UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Implementation und Untersuchung von Position Based Dynamics

Masterarbeit

zur Erlangung des Grades Master of Science (M.Sc.)
im Studiengang Computervisualistik

vorgelegt von

Steven Kölzer

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

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

Koblenz, im September 2019

Erklärung

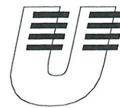
Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

....

(Ort, Datum) (Unterschrift)

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de



UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Masterarbeit
Steven Kölzer
(Mat. Nr. 213 100 814)

Thema: Implementation und Untersuchung von Position Based Dynamics

Position Based Dynamics (PBD) werden zur Simulation von Objekten und Geometrien verwendet. Im Gegensatz zu herkömmlichen Physiksimulationen, die neue Positionen in Abhängigkeit von intern und extern wirkenden Kräften berechnen, manipulieren PBD die Positionen direkt. Dabei werden, je nach Art der zu simulierenden Objekte und Geometrien, sowohl zum Start, als auch zur Laufzeit, Constraints in Form von Gleichungen festgelegt und generiert. Diese dienen dazu, zuvor durch einen Integrationsschritt errechnete, neue Positionen zu korrigieren. Mit iterativen Lösungsverfahren werden die Constraints unabhängig von einander gelöst.

Durch Stabilität, Robustheit und Geschwindigkeit der Simulation findet das Verfahren Einsatz in Echtzeitumgebungen, wie z.B. in Computerspielen oder Virtual Reality, und liefert dabei visuell plausible Ergebnisse.

Ziel dieser Arbeit ist die Implementation eines PBD-Frameworks und die Entwicklung entsprechender Anwendungsbeispiele, inklusive Untersuchung der Möglichkeiten und Grenzen der Methode.

Die inhaltlichen Schwerpunkte der Arbeit sind:

1. Recherche über die Thematik von Position Based Dynamics
2. Implementation eines PBD-Frameworks
3. Bereitstellung von Anwendungsbeispielen
4. Evaluation der Ergebnisse
5. Dokumentation der Ergebnisse

Koblenz, 28.03.2019

– Steven Kölzer –   – Prof. Dr. Stefan Müller –

Zusammenfassung

Simulationen in der Computergraphik haben das Ziel, die Realität so genau wie möglich in einer Szene einzufangen. Dafür werden intern und extern wirkende Kräfte berechnet, aus denen Beschleunigungen berechnet werden. Mit diesen werden letztendlich die Positionen von Geometrien oder Partikeln verändert. *Position Based Dynaimcs* arbeitet direkt auf den Positionen. Durch *Constraints* wird eine Menge von Regeln aufgestellt, die zu jedem Zeitpunkt in der Simulation gelten sollen. Ist dies nicht der Fall, so werden die Positionen so verändert, dass sie den Constraints entsprechen. In dieser Arbeit wird ein PBD-Framework implementiert, in dem Solide und Fluide simuliert werden. Die Constraints werden durch ein Gauss-Seidel-Lösungsverfahren und ein Gauss-Jakobi-Lösungsverfahren gelöst. Die Berechnungen finden dabei komplett auf der GPU statt. Die Ergebnisse sind physikalisch plausible Simulationen, die in Echtzeit laufen.

Abstract

The goal of simulations in computergraphics is the simulation of realistic phenomena of materials. Therefore, internal and external acting forces are accumulated in each timestep. From those, new velocities get calculated that ultimately change the positions of geometry or particles. *Position Based Dynamics* omits thie velocity layer and directly works on the positions. *Constraints* are a set of rules defining the simulated material. Those rules must not be violated throughout the simulation. If this happens, the violating positions get changed so that the constraints get fullfilled once again. In this work a PBD-framework gets implemented, that allows simulations of solids and fluids. Constraints get solved using GPU implementations of Gauss-Seidel and Gauss-Jakobi solvers. Results are physically plausible simulations that are real-time capable.

Inhaltsverzeichnis

1 Einleitung **1**

2 Stand der Technik **2**

3 Grundlagen **3**

3.1 Position Based Dynamics . . . 3

3.2 Lösungsverfahren . . . 5

3.3 Simulationsarten . . . 7

3.3.1 Festkörper . . . 7

3.3.2 Fluide . . . 8

3.4 Offset-Berechnungen . . . 10

3.5 Graphfärbung . . . 13

4 Implementation **16**

4.1 Framework . . . 16

4.1.1 OpenGL-Komponente . . . 16

4.1.2 PBD-Komponente . . . 19

4.2 Partikelsystem . . . 20

4.2.1 Aufbau des Partikelsystems . . . 20

4.2.2 Rendering der Partikel . . . 22

4.3 Simulations-Pipeline auf der GPU . . . 24

4.4 Implementation der Lösungsverfahren . . . 25

4.5 Gauss-Seidel auf der GPU . . . 26

4.6 Gauss-Jakobi auf der GPU . . . 27

4.7 Implementation der Festkörper . . . 33

4.7.1 Graph-Färbung . . . 33

4.7.2 Constraint-Dezimierung . . . 36

4.7.3 Lösen der Distanz-Contraints . . . 38

4.8 Implementation der Fluide . . . 39

4.8.1 Nachbarschaftssuche . . . 39

4.8.2 Lösen der Dichte-Constraints . . . 44

4.9 Viskosität und Wirbelstärke . . . 46

5 Ergebnisse und Auswertung **50**

5.1 Demos und Ergebnisse . . . 50

5.2 Technische Evaluation . . . 58

6 Fazit und Ausblick **66**

1 Einleitung

Simulationen in der Computergraphik haben das Ziel, die Realität so genau wie möglich in einer Szene einzufangen. Dabei sollen die spezifischen Eigenschaften simulierter Materialien so realistisch dargestellt werden. Oftmals führt das dazu, dass dafür in jedem Simulationsschritt alle externen und intern wirkenden Kräfte ermittelt und verrechnet werden müssen, die die Szene manipulieren. Es stellt sich jedoch die Frage, ob so komplexe und genaue Berechnung überhaupt nötig sind, um visuell plausible Ergebnisse zu erzielen.

Das Verfahren *Position Based Dynamics* (PBD) setzt an dieser Stelle an. Die Methode arbeitet auf den Positionen von Geometrien und Partikeln und manipuliert diese in solcher Weise, dass eine Menge an Regeln, die das zu simulierende System beschreibt, durch die komplette Simulation hinweg erfüllt bleibt. Diese Regeln werden in Form von Constraints als Gleichungen aufgestellt. Im Zuge der Simulation werden die Constraint-Gleichungen gelöst, sodass das System bei Veränderung von Positionen immer wieder in den durch die Constraints beschriebenen Zustand zurückgeführt wird. Das Hauptaugenmerk liegt dabei auf visueller Plausibilität. Die physikalische Korrektheit rückt in den Hintergrund.

Die Methode zeichnet sich vor allem durch eine unkomplizierte Implementation aus und liefert zufriedenstellende Ergebnisse, die Echtzeitvorraussetzungen erfüllen. Dadurch eignet sich das Verfahren besonders für *Virtual Reality* und Spiele. In Physik-Engines wie *PhysX* und *Bullet* ist diese Technik bereits integriert.

Ziel dieser Arbeit ist der Aufbau eines PBD-Frameworks und eine vollständige GPU-Implementation des Verfahrens. Anhand von der Simulation von soliden Körpern und Fluiden werden die verwendeten Lösungsverfahren miteinander verglichen und evaluiert. Die simulierten Materialien werden dabei durch Partikel dargestellt. Über eine Nutzeroberfläche können die Simulationen zur Laufzeit manipuliert werden. Dadurch werden verschiedene Effekte sichtbar.

Zu Beginn der Arbeit wird zunächst der aktuelle Stand der Technik besprochen, sowie Anwendungsfelder für PBD vorgestellt. In Kapitel 3 werden die zum Verständnis nötigen Grundlagen besprochen. Dazu zählt nicht nur das PBD-Verfahren an sich, sondern auch die Darstellung von Soliden und Fluiden innerhalb dieses Systems. Das Framework und die Implementation von PBD werden ausführlich in Kapitel 4 besprochen. Dabei wird vor allem auf die Umsetzung der vorgestellten Verfahren auf der GPU eingegangen. Die Ergebnisse und eine Evaluation dieser sind in Kapitel 5.2 zu finden. Zum Abschluss der Arbeit folgt ein Fazit und ein Ausblick auf zukünftige Arbeiten.

2 Stand der Technik

In diesem Kapitel wird ein Überblick über den aktuellen Stand der Technik und die Anwendungsgebiete im Bereich von Position Based Dynamics gegeben.

Extended Position Based Dynamics (*XPBD*) von Macklin *et al.* [MMC16] stellt eine Erweiterung von PBD dar. Durch eine neue Formulierung der Constraints im Rahmen der Konzepte von elastischer potentieller Energie attackiert das Verfahren die zeit- und iterationsabhängige Steifigkeit des PBD-Ansatzes. So lassen sich vor allem elastische Materialien besser simulieren.

In [BML*14] stellen Bouaziz *et al.* *Projektive Dynamik* vor. Durch die Einführung von auf Kontinuum-Mechanik basierenden Constraints in Verbindung mit einem neuen impliziten Lösungsverfahren gelingt ihnen die Verbindung zwischen Finiten Element Methoden und PBD. Ihr Verfahren sticht besonders durch Robustheit, Effizienz und Generalität hervor. Im Gegensatz zur PBD-Methode ist hier die Materialsteifigkeit viel unabhängiger von der Anzahl der Iterationen des Lösungsverfahrens.

Weiss *et al.* [WLJT19] nutzen den PBD-Algorithmus zur Simulation von großen Mengen von Agenten. Sie führen dafür Constraints ein, die Kollisionen zwischen einzelnen Agenten nicht nur vorhersagen, sondern auch entsprechend vermeiden und dadurch mögliche Kollisionen lösen. Sie beobachten dabei, dass die simulierten Agenten automatisch verschiedene Verhaltensmuster entwickeln, wie zum Beispiel das Bilden von Fahrstreifen oder Untergruppen.

Wie Berndt *et al.* in [BTM17] zeigen, findet PBD auch in der Medizin Anwendung. Sie simulieren dabei in einem System verschiedene Gewebearten des menschlichen Körpers, wie Knochen, Muskeln und Fett, auf einmal. Dabei verwenden sie Constraints zur Darstellung von Fluiden und Federkräften, sowie von soften und rigiden Körpern. Mithilfe von PBD schneiden sie durch die verschiedenen Gewebearten und simulieren zusätzlich das Ganze mit interaktiven Raten.

Diese Arbeit richtet sich vor allem nach den älteren Arbeiten von Macklin und Matthias [MM09], Macklin *et al.* [MMCK14], Müller *et al.* [MHHR06] und Jakobsen [Jak03]. Darin werden die Grundzüge des PBD-Algorithmus und die Implementation von verschiedenen Constraint-Arten, wie für zum Beispiel Fluide und Solide gezeigt. Dabei werden auch verschiedene Lösungsverfahren beschrieben und gezeigt, wie verschiedene Constraints in ein einziges Simulationssystem integriert werden können.

3 Grundlagen

In diesem Teil der Arbeit werden die Grundlagen, auf der die Thematik der Ausarbeitung aufgebaut ist, besprochen. Es wird vor allem auf die mathematischen und algorithmischen Hintergründe für die in Kapitel 4 ausgearbeitete Implementation eingegangen.

Zunächst wird in Abschnitt 3.1 der generelle Aufbau von *Position Based Dynamics*-Systemen (PBD) beschrieben. In Abschnitt 3.2 werden die dafür nötigen Lösungsverfahren vorgestellt.

Des Weiteren werden in Abschnitt 3.3 die verschiedenen Simulationsarten illustriert, die bearbeitet worden sind. Abschnitt 3.3.1 erläutert die Grundlagen der implementierten Soliden und Abschnitt 3.3.2 die der Fluide.

In Abschnitt 3.4 werden Offset-Berechnungen mit Hilfe von Präfixsummen erläutert. Diese finden in der Arbeit häufige Anwendung. Abschließend wird in Abschnitt 3.5 die Technik der Graphfärbung in vorgestellt, die vor allem für die GPU-Synchronisierung der Implementation verwendet wird.

3.1 Position Based Dynamics

Nach Müller *et al.* [MHHR06] wird ein Objekt im PBD-Ansatz durch eine Menge von N Vertices und M Constraints definiert. In dieser Arbeit wird ein Vertex durch je ein Partikel repräsentiert. Ein Partikel $p_i,\ i \in [1, ..., N]$ hat dabei eine Position x_i, eine Geschwindigkeit v_i und eine Masse m_i.

Ein Constraint $j \in [1, ..., M]$ besteht, nach [MHHR06], aus

- einer Constraint-Funktion $C_j : \mathbb{R}^{3n_j} \to \mathbb{R}$,
- einer Menge von Partikelindices $\{i_1, ..., i_n\}, i_k \in [1, ..., N]$,
- einem Typ mit entweder *gleich* oder *ungleich*.

Ein Constraint vom Typ *gleich* ist erfüllt, wenn $C(p_{1_x}, ..., p_{i_x}) = 0$, ein Constraint vom Typ *ungleich*, wenn $C(p_{1_x}, ..., p_{i_x}) \geq 0$.

Unter Einbezug eines Zeitschritts Δt wird ein PBD-System mit dem in Algorithmus 1 gezeigten Algorithmus simuliert. Zum Start der Simulation (Zeilen 1-4) werden die Positionen, Geschwindigkeiten und inversen Massen der Partikel auf ihren Ausgangswert gesetzt.

In Zeile 5 beginnt die Simulationsschleife. In jedem Durchlauf werden in einem Integrationsschritt (Euler oder Verlet) die Geschwindigkeiten unter Einbezug extern wirkender Kräften (in dieser Arbeit nur die Gravitation) die Geschwindigkeit aktualisiert und die Positionen der Partikel vorausgesagt.

Das Voraussagen der Position in dabei schon fester Bestandteil der PBD-Pipeline, da über das Lösen von Constraints die vorausgesagte Position so korrigiert wird, dass die Constraints erfüllt sind. Das geschieht in den Zeilen 7-8. Alle Constraints werden dort mit einem Lösungsverfahren, wie z.B

Algorithmus 1: Der PBD-Algorithmus in seiner Basisform. Nach [MHHR06]

```
1  forall particles i do
2  |   x_i = x_i^0
3  |   v_i = v_i^0
4  |_  w_i = 1/m_i
5  loop
6  |   forall particles i do
7  |   |   v_i = v_i + Δt w_i f_ext(x_i)
8  |   |_  p_i = x_i + Δt v_i
9  |   loop solverIteration times
10 |   |_  solveConstraints(C_1, ..., C_m)
11 |   forall particles i do
12 |   |   v_i = (p_i − x_i)/Δt
13 |   |_  x_i = p_i
14 |_  updateVelocities(v_1, ..., v_N)
```

Gauss-Seidel oder Jakobi (siehge Abschnitt 3.2), gelöst. Das Lösungsverfahren wird dabei pro Simulationsschritt *solverIteration*-mal ausgeführt.

Zum Abschluss werden in den Zeilen 11-14 die Geschwindigkeiten über die Position und korrigierte Position der Partikel korrigiert, die Partikelpositionen mit der neu errechneten Positionskorrektur überschrieben. Zum Schluss werden dann auch die eigentlichen Geschwindigkeiten aktualisiert.

Constraint-Projektion In den Lösungsverfahren findet eine Constraint-Projektion statt. Darunter ist, wie Müller *et al.* in [MHHR06] erklären, eine Verschiebung der Positionen anhand der Constraints zu verstehen. Die Positionen werden in jedem Zeitschritt mehrmals projiziert.

Es sei im Folgenden p die Konkatenation $[p_1^T, ..., p_n^T]^T$. Es ist nun das Ziel, eine Korrektur Δp zu finden, sodass $C(p + \Delta p) = 0$. Diese Gleichung kann angenähert werden durch

$$C(p + \Delta p) \approx C(p) + \nabla_p C(p) \cdot \Delta p = 0. \qquad (1)$$

Um Bewegungs- und Drehimpulse zu erhalten wird Δp so eingeschränkt, dass es in Richtung des Constraint-Gradienten $\nabla_p C$ liegt, da dadurch, nach Müller *et al.* [MHHR06] beide Impulse erhalten bleiben.

Demnach ist ein Lagrange-Multiplikator, λ zu finden, sodass

$$\Delta p = \lambda \nabla_p C(p) \qquad (2)$$

erfüllt wird, siehe [BMO*14].

Daraus ergibt sich letztlich für das gesamte Δp

$$\Delta p = -\frac{C(p)}{|\nabla_p C(p)|^2} \nabla_p C(p). \tag{3}$$

Für eine einzelne Position p_i resultiert daraus

$$\Delta p_i = -s \nabla_{p_i} C(p_1, ..., p_n), \tag{4}$$

mit dem Korrekturfaktor

$$s = \frac{C(p_1, ..., p_n)}{\sum_j |\nabla_{p_j} C(p_1, ..., p_n)|^2}, \tag{5}$$

der für alle Punkte, die von einem Constraint beeinflusst werden, gleich ist.

3.2 Lösungsverfahren

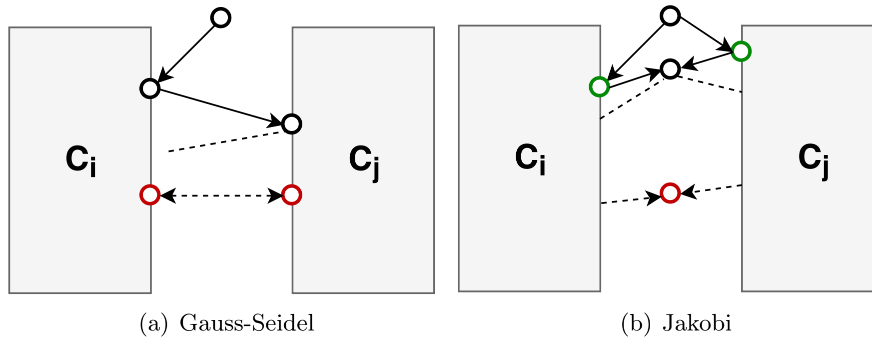
In diesem Abschnitt werden die Grundlagen der in dieser Arbeit verwendeten Lösungsverfahren für PBD besprochen. Im PBD-Algorithmus, siehe Algorithmus 1, wird das verwendete Lösungsverfahren nach dem Vorraussagen der Positionen durch ein Integrationsverfahren gestartet. Es ist die Aufgabe der Lösungsverfahren, die im vorherigen Abschnitt vorgestellte Constraint-Projektion durchzuführen und die vorrausgesagten Positionen so zu ändern, dass alle Constraints erfüllt werden. Pro Zeitschritt wird das aktive Verfahren *solverIteartion*-mal durchgeführt.

Als Lösungsverfahren werden ein Gauss-Seidel und ein Jakobi-Verfahren vorgestellt.

Gauss-Seidel In den Basisarbeiten zu PBD, siehe [Jak03] und [MHHR06] wird ein Gauss-Seidel Verfahren zur Lösung des Systems verwendet. Nach den in Abschnitt 14 gezeigten Formeln wird jeder Constraint einzeln nacheinander gelöst.

Bei dem Verfahren handelt es sich um ein Einzelschrittverfahren, das heißt, dass die errechneten Korrekturen direkt verrechnet werden. Im nächsten Iterationsschritt werden die neu errechneten Werte bereits miteinbezogen, wodurch die Konvergenz des Verfahrens deutlich erhöht wird, siehe [MHHR06]. Dadurch muss die Reihenfolge der Constraints fest sein, da ansonsten andere Voraussetzungen für das Projizieren der Constraints herrschen.

Abbildung 1(a) zeigt ein weiteres Problem des Gauss-Seidel. Da es sich dabei um ein Einzelschrittverfahren handelt, werden Positionsänderungen direkt übernommen. Dadurch alterniert das Verfahren zwischen den möglichen Lösungen der Constraint-Mengen C_i und C_j, da diese keine Überschneidung in ihren Lösungsmengen haben. In einem solchen Fall konvergiert das Verfahren nicht. Die rot umrandeten Punkte in der Abbildung zeigen dies.



(a) Gauss-Seidel (b) Jakobi

Abbildung 1: Der Lösungsunterschied zwischen Gauss-Seidel und Jakobi. Bei Constraint-Sets, die keine Überschneidung in ihren Lösungsmengen haben, alterniert Gauss-Seidel zwischen den möglich Lösungen zum Zeitpunkt der Konvergenz. Jakobi hingegen konvergiert gegen einen Mittelwert der Lösungen. Nach [BML*14]

Jakobi Ein anderes Lösungsverfahren für PBD ist das Jakobi-Verfahren. Im Gegensatz zu dem Gauss-Seidel-Löser handelt es sich hierbei um ein Gesamtschrittverfahren. Demnach werden die Constraints projiziert und die Änderungen für jedes Partikel aufsummiert und erst zum Ende der Iteration in einem Schritt verrechnet.

Wie Macklin *et al.* [MMCK14] beschreiben, haben Jakobi Lösungsverfahren den Nachteil, dass sie zwischen mehreren Lösungen oszillieren. Aus diesem Grund verwenden Macklin *et al.* in ihrer Arbeit lokale und globale Relaxierungsverfahren, die die Gesamtänderung einer Partikelposition zum Ende einer Iteration hin anpassen.

$$\Delta p_i = \frac{\omega}{n} \sum_{n} \nabla C_j \lambda_j \tag{6}$$

Für die lokale Unterrelaxierung wird jede Gesamtänderung durch die Anzahl der auf das Partikel wirkenden Constraints geteilt, wodurch die Änderung gemittelt und Konvergenz erreicht wird. Da die Unterrelaxierung bei hohem n zu stark sein kann, wird ein zusätzlicher globaler Parameter ω eingeführt, der für die globale Überrelaxierung zuständig ist, indem er die lokale Relaxierung abschwächt.

Abbildung 1(b) zeigt das Ergebnis der Constraint-Projektion des Jakobi-Lösungsverfahrens. Da es sich um ein Gesamtschrittverfahren handelt, werden die Änderungen der Projektion aufsummiert, gemittelt und erst zum Ende einer Iteration angewandt. Dadurch konvergiert das Verfahren eindeutig.

Das verwendete Verfahren in dieser Arbeit verbindet, nach Macklin *et al.* [MMCK14], die Ideen des Gauss-Seidel und Jakobi Verfahrens durch einen Gauss-Jakobi Ansatz. Wie bei Gauss-Seidel werden die Constraints zwar

sequentiell projiziert, jedoch wird die Gesamtänderung der Positionen erst zum Ende einer Iteration angewandt. Das Verfahren ist dadurch wesentlich geeigneter zur parallelen Ausführung auf der GPU.

3.3 Simulationsarten

In diesem Abschnitt werden die verschiedenen Simulationsarten und ihre dazugehörigen Constraints vorgestellt. Zunächst werden Festkörper und der Distanz-Constraint erläutert, anschließend Fluide und der Dichte-Constraint.

3.3.1 Festkörper

Festkörper werden über den von Jakobsen in [Jak03] vorgestellten Distanz-Constraint der Form

$$C_i(p_1, p_2) = |p_1 - p_2| - d = 0 \tag{7}$$

dargestellt. Der Constraint C_i wirkt zwischen den beiden Partikeln $\{p_1, p_2\}$, mit dem Ziel, die Distanz zwischen den beiden Partikeln auf d zu halten.

Die partiellen Ableitungen in Abhängigkeit der Partikel $\{p_1, p_2\}$ $\nabla_{p_1} C_i$ und $\nabla_{p_2} C_i$, aus denen sich der Gradient $\nabla C_i(p_1, p_2)$ eines Distanz-Constraints zusammensetzt, sind

$$\begin{aligned} \nabla_{p_1} C_i(p_1, p_2) &= \tfrac{p_1 - p_2}{|p_1 - p_2|}, \\ \nabla_{p_2} C_i(p_1, p_2) &= -\tfrac{p_1 - p_2}{|p_1 - p_2|}, \end{aligned} \tag{8}$$

mit einem Constraint-Fehler von

$$\lambda = \frac{|p_1 - p_2| - d}{2}. \tag{9}$$

Daraus ergeben sich für die Positionsänderungen Δp_1 und Δp_2

$$\Delta p_1 = -\frac{w_1}{w_1 + w_2}(|p_1 - p_2| - d)\frac{p_1 - p_2}{|p_1 - p_2|}, \tag{10}$$

$$\Delta p_2 = \frac{w_2}{w_1 + w_2}(|p_1 - p_2| - d)\frac{p_1 - p_2}{|p_1 - p_2|}, \tag{11}$$

wobei w_i die inverse Masse von Partikel p_i ist.

Abbildung 2 zeigt die Auswirkung eines Distanz-Constraints. Sind die Partikel zu weit auseinander, bewegt er sie näher zusammen und sind sie zu nah, drückt der Constraint sie auseinander.

Zur Simulation eines Festkörpers wird in dieser Arbeit je ein Distanz-Constraint zwischen jedem Partikel des Körpers erstellt und gelöst, siehe Abschnitt 4.7.3.

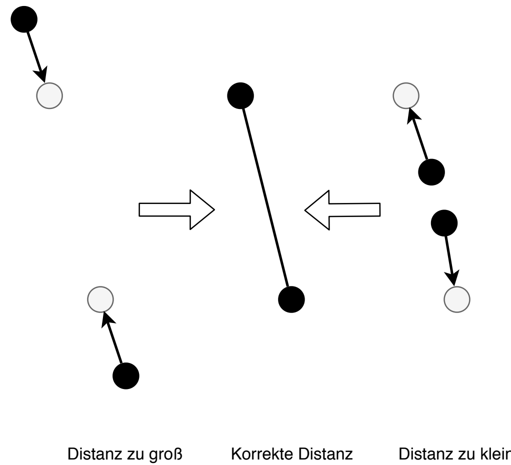


Abbildung 2: Auswirkung eines Distanz-Constraint, nach [Jak03]

3.3.2 Fluide

Fluide werden über den Dichte-Constraint der Form

$$C_i(p_1, ..., p_n) = \frac{\rho_i}{\rho_0} - 1 = 0 \tag{12}$$

mit einem Dichte-Constraint pro Partikel modelliert, siehe [MM09].

Ein Dichte-Constraint C_i ist dabei abhängig von seinem Hauptpartikel p_1 und der Nachbarschaft $\{p_2, ..., p_n\}$. Daneben ist ρ_0 die Ruhedichte des Fluids und ρ_i die geschätzte Dichte des Partikels und seiner Nachbarschaft des SPH-Dichteschätzers

$$\rho_i = \sum_j m_j W(p_i - p_j, h). \tag{13}$$

Damit eine Menge von Partikeln die Constraint-Funktion erfüllt, muss die geschätzte Dichte ρ_i gleich der Ruhedichte ρ_0 sein.

Der Gradient $\nabla_{p_k} C_i$ eines Dichte-Constraints C_i ist

$$\nabla_{p_k} C_i = \frac{1}{\rho_0} \sum_j \nabla_{p_k} W(p_i - p_j, h). \tag{14}$$

Dabei sind zwei verschiedene Fälle zu unterscheiden, je nachdem, ob es sich bei p_k um das Hauptpartikel p_i oder ein Nachbarpartikel p_j handelt

$$\nabla_{p_k} C_i = \frac{1}{\rho_0} \begin{cases} \sum_j \nabla_{p_k} W(p_i - p_j, h) & \text{wenn } k = i \\ -\nabla_{p_k} W(p_i - p_j, h) & \text{wenn } k = j \end{cases} \tag{15}$$

Es ergibt sich demnach ein Constraint-Fehler von

$$\lambda_i = -\frac{C_i(p_1, ..., p_n)}{\sum_k |\nabla_{p_k} C_i|^2 + \varepsilon}. \tag{16}$$

An dieser Stelle verwenden Macklin und Müller [MM09] einen Relaxierungsparameter ε, da der Nenner durch den an hier verwendeten Kernel an seinen Grenzen gegen 0 geht, siehe Abbildung 3(b). Dadurch entstehen Instabilitäten, bevor Partikel in der Simulation letztendlich auseinander gehen.

Die Positionsänderung Δp_i eines Partikels ist somit

$$\Delta p_i = -\frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j) \nabla W(p_i - pj, h). \tag{17}$$

Es ist hervorzuheben, dass in Gleichnung 17 die Constraint-Fehler λ_j der Nachbarpartikel benötigt werden, wodurch die Fehler bei der Implementation vorberechnet werden sollten.

Um SPH-typische Partikelgruppierungen bei der der Existenz weniger Nachbarn zu vermeiden, wird in [MM09] an dieser Stelle nach Monaghan [Mon00] künstlich Druck hinzu geführt, der in Abhängigkeit der Kernelfunktionen definiert ist

$$s_{\text{corr}} = -k \left(\frac{W(p_i - p_j, h)}{W(\Delta q, h)} \right)^n, \tag{18}$$

dabei ist Δq ein fixer Punkt im Kernel. Dieser Term wird schlussendlich in die Positionsaktualisierung mit aufgenommen

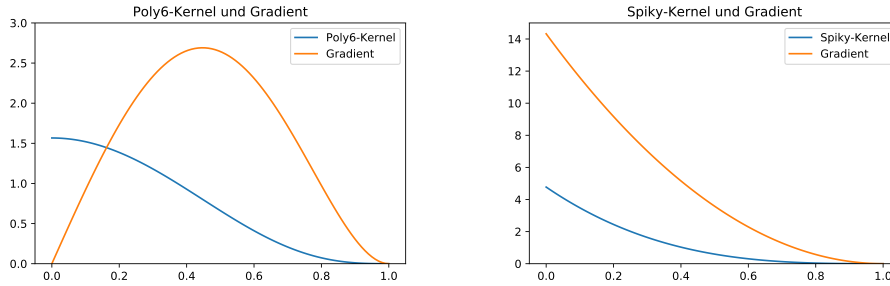
$$\Delta p_i = -\frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j + s_{\text{corr}}) \nabla W(p_i - pj, h). \tag{19}$$

Kernelfunktionen Sowohl bei der Berechnung der geschätzten Dichte ρ_i über den SPH-Dichteschätzer, als auch bei der Berechnung des Constraint-Gradienten $\nabla_{p_k} C_i$ werden Kernel-Funktionen $W(p_i - p_j, h)$ verwendet. Eingabe sind hier der Vektor zwischen den Positionen der Partikel p_i und p_j, sowie der Radius des Kernels h. Macklin und Müller verwenden in ihrer Arbeit zur Berechnung der geschätzten Dichte den *poly6*-Kernel und zur Berechnung des Gradienten den *spiky*-Kernel, siehe [MM09].

Beide Kernelfunktionen stammen aus der Arbeit von Müller *et al.* [MCG03]. Gleichung 20 zeigt die *poly6*-Kernelfunktion, Gleichung 21 zeigt die *spiky*-Kernelfunktion.

$$W_{poly6}(r, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & 0 \leq r \leq h \\ 0 & \text{sonst} \end{cases} \tag{20}$$

$$W_{spiky}(r, h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & 0 \leq r \leq h \\ 0 & \text{sonst} \end{cases} \tag{21}$$



(a) *Poly6*-Kernel (blau) und Gradient (orange)

(b) *Spiky*-Kernel (blau) und Gradient (orange)

Abbildung 3: Die Kernelfunktionen, die zum Berechnen des Dichte-Constraints verwendet werden, nach [MCG03]. Links ist der Graph des *poly6*-Kernels zu sehen, rechts der Graph des *spiky*-Kernels. Beide sind inklusive ihrer Ableitung dargestellt.

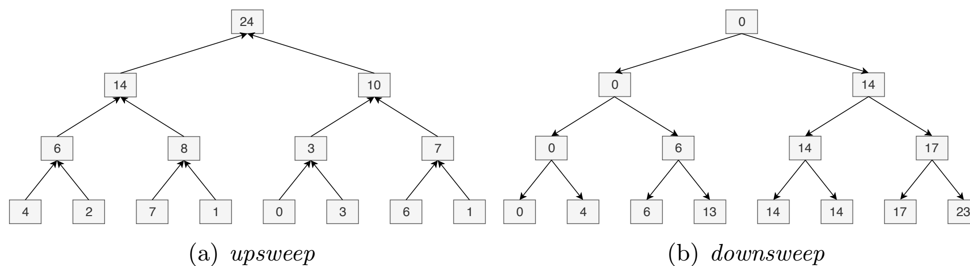
Der Grund für das Benutzen verschiedener Kernelfunktionen bei der Berechnung der geschätzten Dichte ρ_i und des Constraint-Gradienten $\nabla_{p_k} C_i$ lässt sich an den in Abbildung 3 gezeigten Graphen ablesen.

Wie Müller *et al.* [MCG03] beschreiben, dienen die Kernel zum Glätten der Ergebnisse der SPH-Berechnungen. Die *poly6*-Kernelfunktion wurde zu diesem Zweck aufgestellt und glättet, wie an dem blauen Graphen in Abbildung 3(a) zu erkennen ist, bei geringem Partikelabstand konsistent. Da der Abstand r hier auch nur quadratisch auftritt, wird an dieser Stelle eine Wurzelberechnung gespart. Bei der Berechnung des Gradienten wird jedoch der Gradient der Kernelfunktion benötigt. Wie an dem roten Graphen in Abbildung 3(a) zu erkennen ist, verschwindet dieser bei kleinem r, was nach [MCG03] dazu führt, dass nahe Partikel bei der Simulation Gruppen bilden, da die generierte Abstoßkraft für nahe Partikel gegen 0 geht.

Der Wert des Gradienten der *spiky*-Kernelfunktion, siehe roter Graph in Abbildung 3(b), ist sehr hoch bei kleinem r und konvergiert gegen 0 bei steigendem r. Dadurch wird nicht nur die nötige abstoßende Kraft für nahe Partikel generiert, sondern auch für weiter entfernte Partikel verringert, siehe [MCG03].

3.4 Offset-Berechnungen

Für die effiziente Implementation der Algorithmen auf der GPU sind weitere Vorberechnungen nötig. Um eine sequentielle Suche nach den Nachbarn oder der für ein Partikel zu lösenden Constraints zu verkürzen, werden Präfixsummen-Algorithmen nach Blelloch [Ble90] angewandt. Dadurch ist es möglich schon im Vorhinein Offsets für den direkten Zugriff auf eine sortierte Liste zu berechnen.



(a) *upsweep* (b) *downsweep*

Abbildung 4: Links wird ein *upsweep* auf einem Baum ausgeführt, rechts ein *downsweep* auf einem Baum. Nach [Ble90].

Präfixoperationen Im Zentrum steht die in [Ble90] eingeführte *all-prefix-sums*-Operation, mit dem Ziel für eine sortierte Liste alle Präfixsummen zu berechnen. Die Operation erhält als Eingabe einen binären assoziativen Operator $\oplus$ und eine Menge mit n Elementen $[a_0, a_1, ..., a_{n-1}]$ und gibt die Menge $[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus, ..., \oplus a_{n-1})]$ zurück.

So wird beispielsweise aus der Menge $[4, 2, 7, 1, 0, 3, 6, 1]$ und Addition als Operation die Menge $[4, 6, 13, 14, 14, 17, 23, 24]$.

Wird die *all-prefix-sums*-Operation auf einem Vektor ausgeführt, wird sie auch als *scan* bezeichnet.

Als *prescan* wird die Operation bezeichnet, die aus einem binären assoziativen Operator $\oplus$ mit der Identität I und einem Vektor mit n Elementen $[a_0, a_1, ..., a_{n-1}]$ den Vektor $[I, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus, ..., \oplus a_{n-2})]$ zurückgibt, also einen *scan* ohne das letzte Element und der Identität I an Index 0. Die *prescan*-Operation liefert später die benötigten Startoffsets.

Die Operation *reduce* berechnet das letzte Element der *scan*-Operation. Aus einem binären assoziativen Operator $\oplus$ und einer Menge mit n Elementen $[a_0, a_1, ..., a_{n-1}]$ berechnet ein *reduce* den Wert $a_0 \oplus a_1 \oplus, ..., \oplus a_{n-1}$.

Präfixalgorithmen Ziel ist es, den *prescan* möglichst effizient auszuführen, da dieser, wie schon erwähnt, die Start-Offsets berechnet. Blelloch schlägt in [Ble90] eine parallele Implementation mit Hilfe eines balancierten binären Baumes vor.

Dabei wird die *reduce*-Operation auf einer Ebene des Baumes parallel ausgeführt, sodass in jedem Elternknoten einer Ebene die Summe der Kinderknoten steht. Wird dies iterativ für jede Ebene des Baumes mit Start bei maximaler Tiefe ausgeführt, steht im Wurzelknoten das Ergebnis des *prescans*. Das Ausführen des *prescans* auf einem binären balancierten Baum wird als *upsweep* bezeichnet. Abbildung 4(a) zeigt einen Baum mit ausgeführtem *upsweep*.

In der Praxis wird der *upsweep* auf einem eindimensionalen Vektor ausgeführt, indem ein binärer balancierter Baum über den Vektor gelegt wird und auf die entsprechenden Stellen im Vektor zugegriffen wird.

Algorithmus 2: Parallel ausgeführter *+-reduce* (*upsweep*) auf einem Array. Nach [Ble90]

```
1 for d from 0 to maxDepth do
2     do in parallel for i from 0 to n − 1 by 2^(d+1)
3         s[i + 2^(d+1) − 1] = s[i + 2^d − 1] + s[i + 2^(d+1) − 1]
```

Algorithmus 2 zeigt den Algorithmus zum Ausführen eines *upsweeps* auf einem Array. Je nach aktueller Tiefe, auf der in einer Iteration gearbeitet wird, werden andere Stellen des Arrays miteinander verrechnet. So wird ein Baum über den Array gelegt.

Der Baum, der durch die vorangegangene Operation entstanden ist, enthält Teilsummen, die verwendete werden können, um den *prescan* des Ausgangsarrays zu berechnen.

Ausgehend von der Wurzel des Baumes wird ein weiterer Berechnungsschritt bis zu den Blattknoten des Baumes durchgeführt. Dabei wird zunächst in den Wurzelknoten die Identität I eingesetzt, dann werden die Kindknoten berechnet. Der linke Kindknoten erhält den Wert des Elkternknotens, der rechte Kindknoten erhält den vorherigen Wert des linken Kindknotens verrechnet über $\oplus$ mit dem Wert des Elternknotens. Diese Operation wird als *downsweep* bezeichnet, siehe [Ble90]. Ist der *downsweep* auf dem gesamten Baum ausgeführt worden, steht in den Blattknoten das Ergebnis des *prescans*. Abbildung 4(b) zeigt einen Baum mit ausgeführtem *downsweep*.

Algorithmus 3: Parallel ausgeführter *downsweep* auf einem Array. Nach [Ble90]

```
1 s[n − 1] = 0
2 for d from maxDepth downto 0 do
3     do in parallel for i from 0 to n − 1 by 2^(d+1)
4         temp = s[i + 2^d − 1]
5         s[i + 2^d − 1] = s[i + 2^(d+1) − 1]
6         s[i + 2^(d+1) − 1] = temp + s[i + 2^(d+1) − 1]
```

Algorithmus 3 zeigt den parallelen Algorithmus, mit dem der *downsweep* auf einem Array ausgeführt wird. Wie auch bei dem *upsweep* wird über die Iteration über die Tiefe auf die richtigen Einträge im Array zugegriffen.

Abbildung 5 zeigt einen vollständigen parallel ausgeführten *+-prescan*. In den Schritten 1-3 wird ein *upsweep* auf dem Array ausgeführt. Anschließend wird in Schritt 4 die Identität in das letzte Element, was der Wurzel des Baumes entspricht, eingesetzt. Zum Schluss wird in den Schritten 5-7 der *downsweep* ausgeführt.

Schritt	Speicher								
0	4	2	7	1	0	3	6	1	
1	4	6	7	8	0	3	6	7	upsweep
2	4	6	7	14	0	3	6	10	upsweep
3	4	6	7	14	0	3	6	24	upsweep
4	4	6	7	14	0	3	6	0	← löschen
5	4	6	7	0	0	3	6	14	downsweep
6	4	0	7	6	0	14	6	17	downsweep
7	0	4	6	13	14	14	17	23	downsweep

Abbildung 5: Ein parallel ausgeführter $+$-*prescan* nach [Ble90]

Es ist zu beachten, dass die Algorithmen auf einem balancierten Baum ausgeführt werden. Der Array muss somit eine Größe von 2^n haben, damit der Ansatz funktioniert, da ansonsten kein Baum über den Array gelegt werden kann.

3.5 Graphfärbung

Graphfärbung wird von Fratarcangeli und Pellacini in [FP15a] genutzt, um Constraints, die unabhängig voneinander sind, zu Gruppieren und dann effizient auf der GPU zu lösen.

Dabei handelt es sich um ein Problem aus der Graphentheorie. Ein Graph G ist dabei nach Fratarcangeli und Pellacini ein Tupel (V, E), wobei V eine endliche und nicht leere Menge an Vertices ist und E eine Menge von Tupeln mit je verschiedenene Vertices

$$E \subset \{(v_i, v_j) : i \neq j,\ v_i, v_j \in V\},$$

siehe [FP15b].

Das Graphfärbungsproblem ist nach Matula [MB83] folgendermaßen definiert:

Eine k-Färbung von G ist eine Zuweisung von Farben in Form von natürlichen Zahlen mit $1 \leq c(v) \leq k$ für jeden Vertex $v \in V$, sodass benachbarte Vertices verschiedene Farben erhalten. Zwei Vertices gelten dabei als benachbart wenn gilt:

$$\forall v_i, v_j \in V, \exists e \in E : i \neq j \implies (v_i, v_j) \in e,$$

wenn es also mindestens eine Kante $e \in E$ gibt, die die Vertices v_i, v_j verbindet.

Durch eine k-Färbung von G werden k unabhängige Mengen von Vertices ermittelt.

Zudem existiert eine *chromatische Zahl* $\chi(G)$, die das kleinste k ist, für das G eine k-Färbung hat. Das Finden von $\chi(G)$ für einen beliebigen Graphen G ist dabei, nach [MB83], ein NP-vollständiges Problem und wird in dieser Arbeit nicht weiter betrachtet.

Matula stellt in [MB83] die *sequentielle Färbung* (engl. *sequential coloring*) vor, um einen Graphen G zu kolorieren.

Gegeben sei eine Ordnung $v_1, v_2, ..., v_n$ der Vertices von G, dann ist eine *sequentielle Färbung* eine *k-Färbung* von G, wenn

$$c(v_i) = min\{m|1 \leq m \neq c(v_j) \text{ wenn } v_j, v_i \text{ benachbart sind}, j < i\},$$

mit $k = max\{c(v_i)|1 \leq i \leq n\}$.

Eine Ordnung der Vertices ist dabei eine beliebige Reihenfolge zum abarbeiten des Färbens. Diese Heuristik färbt demnach jeden Vertex v_i so, dass er die kleinste mögliche Farbe erhält, die seine Nachbarn nicht haben.

Algorithmus 4: Greedy-Algorithmus zum Färben eines Graphen $G(V, E)$, nach [FP15a]

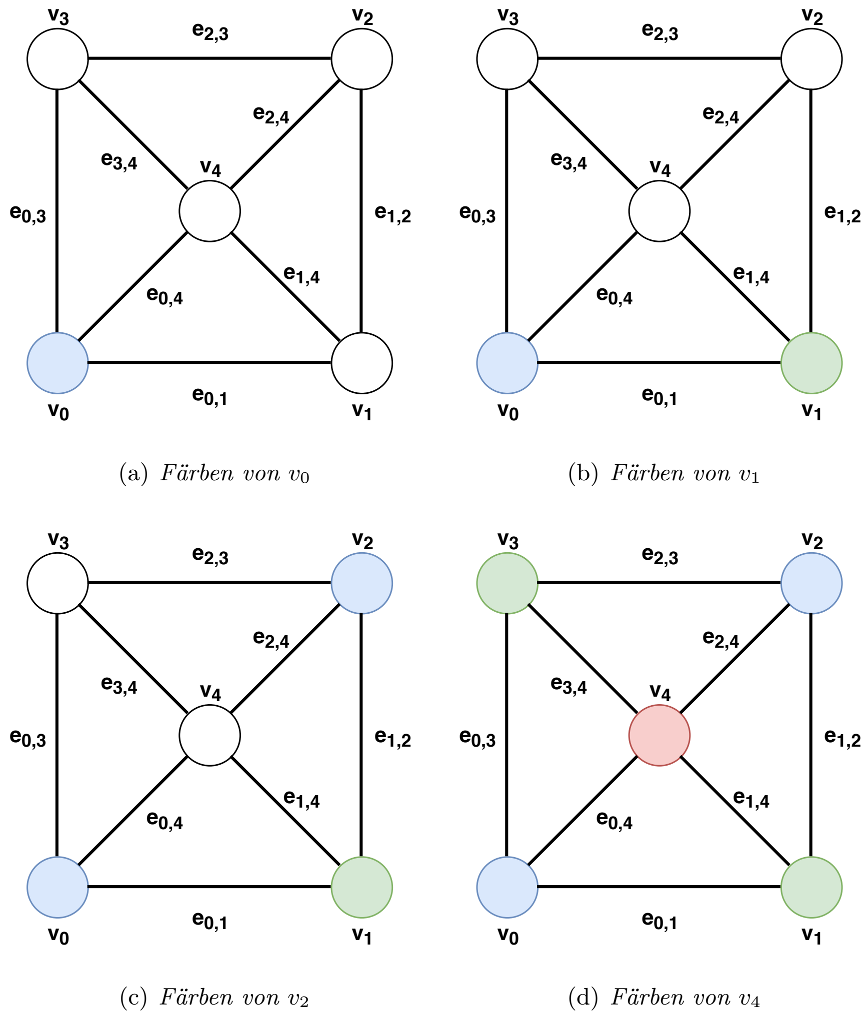
```
1 es sei v_1, v_2, ..., v_n eine Ordnung auf V
2 for i = 1 to n do
3 |  verbotene Farben für v_i ermitteln
4 |  setze v_i auf die kleinste verfügbare Farbe
```

Abbildung 6 zeigt das Färben des Graphen $G(V, E)$, mit $V = \{v_0, v_1, v_2, v_3, v_4\}$ und $E = \{(v_0, v_1), (v_0, v_3), (v_0, v_4), (v_1, v_2), (v_1, v_4), (v_2, v_3), (v_2, v_4), (v_3, v_4)\}$. Die Ordnung der Vertices sei die Reihenfolge, in der diese in V abgelegt sind. Zur besseren Darstellung der Färbung sind die Farben hier als tatsächliche Farben mit blau=0, grün=1 und rot=2 dargestellt.

Zum Start des Algorithmus ist der Graph ungefärbt. Zuerst wird v_0 eingefärbt, da dieser Vertex in der Ordnung an erster Stelle steht, siehe Abbildung 6(a). Die Menge der Nachbarn von v_0 ist $\{v_1, v_3, v_4\}$. Keiner der Nachbarn ist bereits eingefärbt, also wird die Farbe von v_0 auf die kleinste, neue Farbe gesetzt. In diesem Fall ist das Farbe 0.

Im nächsten Schritt wird der nächste Vertex der Ordnung betrachtet, v_1, siehe Abbildung 6(b). Die Nachbarn von v_1 sind $\{v_0, v_2, v_4\}$. Da v_0 bereits eingefärbt ist, darf v_1 nicht mit derselben Farbe eingefärbt werden. Da noch keine weiteren verfügbaren Farben existieren, bekommt v_1 eine neue Farbe zugewiesen, Farbe 1.

Als nächtstes wird v_2 eingefärbt, siehe Abbildung 6(c). Die Nachbarn von v_2 sind $\{v_1, v_3, v_4\}$. v_1 wurde bereits mit Farbe 1 eingefärbt. In diesem Fall gibt es mit Farbe 0 noch eine kleinste verfügbare Farbe. Demnach wird v_2 mit 0 eingefärbt.



(a) *Färben von v_0* (b) *Färben von v_1*

(c) *Färben von v_2* (d) *Färben von v_4*

Abbildung 6: Färben eines Graphen nach der vorgestellten Greedy-Heuristik

Nach dem selben Prinzip werden die restlichen Vertices von G behandelt, siehe Abbildung 6(d). Der vollständige Graph wurde mit 3 Farben eingefärbt. Eine Implementation des Algorithmus und die Übertragung der Graphfärbung auf Constraints sind in Abschnitt 4.7.1 zu finden.

4 Implementation

In diesem Kapitel wird die Implementation vorgestellt. Zunächst werden in Abschnitt 4.1 das für diese Arbeit implementierte Framework und die verwendeten Bibliotheken besprochen. Anschließend wird in Abschnitt 4.2 das Rendering des Partikelsystems behandelt. Das restliche Kapitel behandelt die Implementation der PBD-spezifischen Aspekte, namentlich die Implementation der in Kapitel 3 vorgestellten Pipeline auf der GPU in Abschnitt 4.4, der Festkörper in Abschnitt 4.7 und Fluide in Abschnitt 4.8. Dabei wird zusätzlich auf die jeweils für die Lösung spezifischen Algorithmen und Ansätze eingegangen.

4.1 Framework

In dem Framework werden hauptsächlich *OpenGL* in Version 4.6 und *C++* verwendet. Generiert werden die Projekte mit *CMake*, zusätzliche Bibliotheken werden mithilfe von *vcpkg*[1] eingebunden.

Verwendete externe Bibilotheken sind *Dear ImGui*[2] und *tiny file dialogs*[3] für das Erstellen von Benutzeroberflächen, *OpenMP*[4] zur Parallelisierung von Algorithmen auf der CPU, *JSON for Modern C++*[5] um Einstellungen und Vorberechnungen zu serialisieren und deserialisieren und *GLShader Processor*[6], welche als als Präprozessor für *Shader*-Dateien verwendet wird.

Das Framework ist weitesgehend in zwei Komponenten aufgeteilt. Zum einen in eine *OpenGL*-Komponente, zum anderen in eine PBD-Komponente.

4.1.1 OpenGL-Komponente

Die *OpenGL*-Komponente dient zur Abstrahierung von *OpenGL*-Befehlen und -Objekten, sodass ein einfaches Arbeiten möglich ist. Es handelt sich hierbei zwar um ein vollständiges Rendering-Framework mit vordefinierten Geometrien und der Möglichkeit Modelle zu laden, es werden jedoch diverse Standardkomponenten, wie Texturen oder FBOs in dieser Arbeit nicht benutzt. Die wichtigsten Komponenten und deren Benutzung sind im Folgenden aufgeführt.

Hauptsächlich werden in der Arbeit *Shader Storage Buffer Objects* (*SSBOs*) verwendet, die über die Klasse *SSBO* abstrahiert werden. Diese dienen dazu, alle nötigen Daten von der CPU auf die GPU zu laden und dort weiter zu verarbeiten. Dabei gibt es allgemein zwei verschiedene Arten von Konstruktoren, um ein *SSBO* anzulegen. Auf der einen Seite solche, die ein

[1] https://github.com/Microsoft/vcpkg , letzter Aufruf am 27.08.2019
[2] https://github.com/ocornut/imgui , letzter Aufruf am 27.08.2019
[3] https://github.com/native-toolkit/tinyfiledialogs , letzter Aufruf am 27.08.2019
[4] https://www.openmp.org/ , letzter Aufruf am 27.08.2019
[5] https://github.com/nlohmann/json , letzter Aufruf am 27.08.2019
[6] https://gitlab.uni-koblenz.de/johannesbraun/glshader , letzter Aufruf am 27.08.2019

beliebiges *Struct* fester Größe als Daten übergeben bekommen. Auf der anderen Seite solche, die Daten unbekannter Größe übergeben bekommen.

```
//Konstruktor fuer ein Struct mit fester Groesse...
template <class T>
SSBO(const std::shared_ptr<T>& instance, const std::string&
    instance_name, GLuint binding_point, GLenum usage);

//mit Beispielaufruf fuer ein SSBO mit Kameradaten, die als Struct
    vorliegen
auto camera = std::make_shared<SSBO>(CameraBlock, "CameraBlock", 3,
    GL_DYNAMIC_DRAW);

//Konstruktor fuer eine Menge an Daten...
SSBO(const std::shared_ptr<std::vector<int>>& instance, const
    std::string& instance_name, GLuint binding_point, GLenum usage);

//mit Beispielaufruf fuer einen std::vector<int>, dessen Groesse
    erst zur Laufzeit bekannt ist
auto gridStartIndex = std::make_shared<SSBO>(gridStartVec,
    "GridStartIndex", 11, GL_DYNAMIC_DRAW);
```

Quellcode 1: SSBO Konstruktoren

Durch `template <class T>` in Quellcode 1 können beliebige, aber feste Structs als Eingabedaten übergeben werden.

Prinzipiell folgt das Framework dem Grundsatz, Daten zur Laufzeit so wenig wie möglich zwischen GPU und CPU zu transferieren und diese im Optimalfall einmal zu Beginn auf die Grafikkarte zu laden. Update-Funktionen zum Ändern der Daten auf GPU oder CPU, sowie für einzelne Uniform-Variablen sind zwar auch vorhanden, jedoch nur für Typen die explizit ein Update benötigen. So gibt es beispielsweise eine Update-Funktion für SSBOs mit Partikeldaten, damit die auf der GPU veränderten Daten für späteres Rendering CPU-seitig in entsprechende *Vertex Array Objects* (*VAO*) geschrieben werden können. Es gibt aber keine Update-Funktion für, zum Beispiel, die Daten der Ausmaße eines Uniformgrids, welches in Abschnitt 4.8.1 weiter beschrieben wird, da diese nach dem ersten Upload auf der Grafikkarte bleiben können und nicht verändert werden müssen.

Die verwendeten Structs folgen alle einem festen Schema. Sie erben alle von einem leeren Struct `struct structure {};`, sodass Funktionen mit diesem Typ als Eingabe anstelle der einzelnen Struct-Typen aufgerufen werden können. Es ist dabei jedoch zu beachten, dass dadurch nicht auf nicht existierende Felder zugegriffen wird. Um eventuelle Probleme durch *Padding* zu vermeiden, werden keine Felder genutzt, deren Ausrichtung und Größe unterschiedlich sind. So werden beispielsweise `vec4` statt `vec3` verwendet, da diese eine Aurichtung von 16 Bytes, aber eine Größe von 12 Bytes haben. Auch die Gesamtgröße jedes festen Structs ist festgelegt, sodass diese mit

der
Ausrichtung von `vec4` übereinstimmen. Da dies aber nicht immer automatisch der Fall ist, werden, wenn nötig, manuell zusätzliche Felder eingefügt.

```
struct systemData : structure {
  float time;
  float deltatime;
  float pad[2];
};
```

Quellcode 2: Ein valides Struct im Framework

Quellcode 2 zeigt ein valides Struct. Es erbt von dem leeren Struct, verwendet keine unerlaubten Felder und hat im ganzen die korrekte Ausrichtung. Angelegt wird ein einzelnes Struct als Objekt vom Typ `std::shared_ptr<T>`, wobei `T` der Typ des jeweiligen Structs ist.

```
//Anlegen
auto SystemBlock = std::make_shared<systemData>();

//Befuellen
SystemBlock->time = currentFrame;
SystemBlock->deltatime = deltaTime;
```

Quellcode 3: Anlegen und befüllen eines Structs vom Typ `systemData`

Quellcode 3 zeigt beispielhaft das Anlegen und Befüllen eines Structs vom Typ `systemData`.

Die genutzten Shader werden als Objekt der Klasse `shaderHandle` angelegt. Die Eingabewerte für den Konstruktor sind Referenzen des Typs `std::filesystem::path`. In Kombination mit global gesetzten Pfadvariablen, die auf den Ordner mit den Shader-Dateien zeigen, ist es nur nötig, den Namen der Shaderdatei, die geladen werden soll, zu übergeben.

```
shaderHandle renderShader("pbdRender.vert", "pbdRender.frag");
```

Quellcode 4: Erstellen eines Shaderobjektes mit Vertex- und Fragment-Shader im Framework

Quellcode 4 zeigt das Erstellen eines Shaderobjektes. Es reicht an dieser Stelle aus, lediglich die Dateinamen von Vertex- und Fragment-Shader zu übergeben. Neben diesem Konstruktor existiert noch ein weiterer für Compute-Shader, die häufigen Einsatz finden. Eine Erweiterung für andere Shadertypen, wie z.B Geometry-Shader, ist natürlich möglich und sind zum Teil auch bereits vorhanden, nur werden solche aktuell nicht im Framework verwendet. Beim erstellen der Shader wird, wie anfangs erwähnt, der Präprozessor aus der Bibliothek *GLShader Processor* verwendet. Dafür muss zunächst ein Objekt vom Typ `glsp::state` erstellt werden. Darüber werden dann nicht

nur über den Befehl `preprocess_file(const files::path& file_path, ...)` Shader-Dateien eingelesen werden, sondern es wird auch mit dem Befehl `add_include_dir(const files::path& dir)` ein Include-Verzeichnis gesetzt. Dadurch können Shaderdateien (**.glsl*) in andere (**.comp*, **.vert* etc.) inkludiert werden, sodass auch auf der GPU eine Abkapselung von Befehlen möglich ist.

Für den Shader benötigte SSBOs können dann über die Funktion `assignSSBO(std::shared_ptr<SSBO> sptr)` an den Shader gebunden werden. Dabei handelt es sich lediglich um eine Abstraktion der dazu nötigen OpenGL-Befehle. Die nötigen Bindings werden dabei schon bei der Erstellung des SSBOs gesetzt und hier an den Shader weitergereicht. Zusätzlich wird dann auch die Eingabe vom Typ
`std::shared_ptr<SSBO>` in einer Member-Variable des `ShaderHandles` gespeichert. Für Uniform-Variablen existiert eine ähnliche Abstrahierung, nur werden solche selten verwendet, da im Framework eher mit Structs und SSBOs gearbeitet wird.

4.1.2 PBD-Komponente

In der PBD-Komponente des Frameworks befinden sich die für den PBD-Teil notwendigen Klassen. Da der Großteil der Arbeit auf der GPU stattfindet, sind die hier vorhandenen Klassen für die Vorverarbeitung der Daten zuständig, sei es für die zum Rendern nötigen Geometrien im Falle des Partikelsystems und des Uniform-Grids, oder explizite Vorberechnungen, die auf der CPU getätigt werden, wie für die Graphfärbung. Auf deren Funktionsweisen wird in den folgenden Abschnitten näher eingegangen.

An dieser Stelle lohnt es sich dennoch, auf die Constraint-Klassen einzugehen. Für die Constraints existiert das in Quellcode 5 gezeigte Struct.

```
struct UberConstraintData : structure {
  glm::vec4 min;
  glm::vec4 max;
  float stiffness;
  float dist;
  int color;
  int type;
  int indices[499];
  int equality;
};
```

Quellcode 5: `UberConstraintData` ist das eine Struct für alle möglichen Constraints

Ein Struct enthält dabei alle Felder, die für verschiedene Constraint-Typen benötigt werden. Demnach enthält ein Constraint für Fluide, welches lediglich eine Liste für Nachbarn benötigt, hier `int indices[499]`, auch ein

Feld für eine Distanz (hier das Feld `float dist`), die natürlich in diesem Fall keine Anwendung findet. Der Sinn dahinter ist die Möglichkeit, alle Constraints unabhängig ihrer Art in einer Liste abspeichern und diese dann in einer Operation auf die GPU laden zu können.

Das Setzen der richtigen Felder für einen Constraint übernimmt die Klasse *constraint* in Verbindung mit der Klasse *ConstraintFunction* und deren Unterklassen, wobei davon für jede Art von Constraint eine existiert. So gibt es beispielsweise die Klasse *DistanceEq*, die den Distanz-Constraint darstellt. Die Membervariablen der Unterklassen entsprechen den für den Constraint nötigen Attributen, wie zum Beispiel die Distanz im Fall der Klasse *DistanceEq*.

Jedes Objekt vom Typ `ConstraintFunction` kennt die Art des Constraints, die es modelliert. Dieser wird beim Aufruf des jeweiligen Konstruktors einer Unterklasse über den Befehl `typeid(T).name()` in einer weiteren Membervariable gesetzt. `T` ist hierbei die Unterklasse. Dadurch können die Constraints voneinander unterschieden werden, wenn die einzelnen Structs erstellt werden. Das geschieht in der Funktion `generateUberConstraint()` der Klasse *Constraint*. Ermöglicht wird das auch durch Vorwärtsdeklarierungen innerhalb dieser Klassen. In der Anwendung werden zunächst alle nötigen Constraints erstellt und in einem Vektor vom Typ `<Constraint>` gespeichert. In einer Schleife wird dann für jedes Objekt in der Liste der jeweilige Uber-Constraint generiert und in einer weiteren Liste abgelegt, die als SSBO an die Grafikkarte übergeben wird.

4.2 Partikelsystem

Die in dieser Arbeit vorgestellte Simulationsmethode arbeitet in diesem Fall auf Partikeln, statt auf Geometrien oder Modellen. Aus diesem Grund ist ein Partikelsystem implementiert worden. Im folgenden wird zunächst in Abschnitt 4.2.1 der Aufbau des Partikelsystems beschrieben, abschließend wird in Abschnitt 4.2.2 auf das Rendering der Partikel eingegangen.

4.2.1 Aufbau des Partikelsystems

Die Klasse *particleSystem* ist Teil der in Abschnitt 4.1.2 beschriebenen PBD-Komponente des Frameworks. Ein einzelnes Partikel wird durch das in Quellcode 6 gezeigte Struct `particleData` beschrieben.

```
struct particleData : structure{
  glm::vec4 position;
  glm::vec4 velocity;
  glm::vec4 oldPosition;
  float lifeTime;
  float mass;
  float phase;
  float timestep;
};
```

Quellcode 6: Struct mit Daten eines einzelnen Partikels

Am wichtigsten für die Simulation sind hierbei die Attribute `position`, `velocity` und `oldPosition`, welche die Position (aktuelle und vorherige) und die Geschwindigkeit des Partikels beschreiben. Die Masse `mass` jedes Partikels ist zwar durchweg ein und derselbe Wert, jedoch wird dieser Wert in die Berechnungen der Simulation miteinbezogen. Das Attribut `phase` dient zur Gruppierung von Partikeln. Sowohl das Feld `timestep`, als auch das Feld `lifeTime` dienten bisher lediglich Testzwecken.

Ein Objekt der Klasse *particleSystem* verwaltet hauptsächlich einen Vektor vom Typ `particleData`. Dieser kann entweder durch verschiedene Funktionen der Klasse mit Partikeln befüllt werden, oder es kann ein bereits existierender Vektor mit Partikeln dem Objekt bei dem Aufruf des entsprechenden Konstruktors zugewiesen werden. So zum Beispiel erstellt die Funktion `addNParticles(int nr)` zufällig `nr`-viele Partikel, die dem Partikelsystem zugewiesen werden. Die Verteilung der Partikel steuert beispielsweise das Objekt `std::uniform_real_distribution<float>(0.0, 1.0);` in Verbindung mit einem Objekt des Typs `std::random_device` und einem Zufallszahlengenerator, wie einem Mersenne-Twister `std::mt19937`. Dieser benötigt bei der Erstellung das `std::random_device`-Objekt als Eingabe, die Verteilung wiederum benötigt den Zufallszahlengenerator als Eingabe.

Es ist auch möglich, Partikel zielgerichteter zu erzeugen, indem Partikel von Hand Eingaben erhalten. Vorgefertigte Funktionen dafür sind auch vorhanden. So erstellt die Methode `addStick(float distance)` zwei zufällige Partikel mit einem Abstand von `distance`, die Methode `addCube(float length)` erstellt Partikel an den Ecken eines Würfels mit Seitenlänge `length`. Auch das Setzen von Partikeln anhand von definierter Geometrie ist über die Funktion `addFromGeometry(Geometry g)` möglich, wobei `Geometry` ein eigenständiger Typ des Frameworks ist. Mit jedem erstellten Partikel wird ein Zähler inkrementiert, der dem Partikelsystem als Member-Variable zugewiesen ist. Dadurch kennt jedes Partikelsystem die Anzahl seiner Partikel.

Des weiteren besitzt ein Partikelsystem ein Feld für eine Geometrie, die angibt, als was die Partikel dargestellt werden. Darauf wird in Abschnitt 4.2.2 genauer eingegangen.

Das Partikelsystem kümmert sich auch um das Erstellen der Constraints,

bevor sie in *Uber-Constraints* umgewandelt werden. Dafür gibt es die Funktionen `createDistanceConstraints(std::vector<Constraint>& vec)` und `createFluidConstraints(std::vector<Constraint>& vec)`. Beide Funktionen erstellen die angegebenen Constraints für alle Partikel, die ein Partikelsystem hat und schiebt diese in den referenzierten Vektor. Durch Funktionsüberladung kann der Bereich der bearbeiteten Partikel mit weiteren Parametern abgegrenzt werden.

Im Falle der Distanz-Constraints wird von jedem Partikel zu jedem anderen ein Constraint mit den jeweiligen Partikelindices und der Distanz zwischen diesen erstellt. Für die Dichte-Constraints wird für jeden Partikel ein Constraint erstellt und der Index des Partikel gespeichert. Die Nachbarschaften werden später zur Laufzeit hinzugefügt, siehe Abschnitt 4.8.

4.2.2 Rendering der Partikel

Die Partikel eines Partikelsystems werden instanziert mit einer zugewiesenen Geometrie gezeichnet. Über den Befehl
`prepareGeometry(std::shared_ptr<Geometry> geom)`
kann ein `std::shared_ptr<Geometry>` dem Partikelsystem zugewiesen werden. Im selben Befehl werden dann die Buffer der Geometrie mit den nötigen Daten, die bei dem Anlegen der Geometrie gesetzt werden, gefüllt.

Gezeichnet werden die Partikel mit der Funktion
`drawGeometryInstanced()`. Gebunden wird an dieser Stelle der Vertex-Array der zugewiesenen Geometrie, der Zeichenbefehl ist dann
`glDrawArraysInstanced(GL_TRIANGLES, 0, m_assignedGeometry->m_vertCount, m_count);`. Dadurch wird die Geometrie `m_count`-mal, also so oft, wie es Partikel gibt, gezeichnet.

Wichtig ist, dass die Geometrie im Vertex-Shader an die Position der Partikel geschoben wird. Dafür kann im Shader mit der Variable `gl_InstanceID` auf die ID des aktuellen Aufrufs zugegriffen werden. Diese ID wird benutzt, um auf die Position des i-ten Partikels zuzugreifen. Um diesen Wert wird jeder Vertex der Geometrie verschoben. Quellcode 7 zeigt die Translation der Geometrie im Vertex-Shader.

```
1 vec4 new_pos = projection * view *
    vec4(pos.xyz+particles[gl_InstanceID].position.xyz, 1.0);
```

Quellcode 7: Errechnen der Vertexpositionen

```cpp
//Geometrien anlegen
auto sphere_ptr = std::make_shared<Sphere>(1.f, 10, 10, false);
auto sphere_ptr2 = std::make_shared<Sphere>(10.f, 15, 15, true);

//Shader anlegen
shaderHandle renderShader("pbdRender.vert", "pbdRender.frag");

//Partikelsystem anlegen
auto renderPs = std::make_shared<particleSystem>(nrParticles, true);

//Partikel hinzufuegen
renderPs->addFromGeometry(*sphere_ptr2);

//SSBO anhand der Partikeldaten fuellen
auto pRender = std::make_shared<SSBO>(renderPs->m_particlePointer,
    "RenderPSBlock", 4, GL_STREAM_DRAW);

//SSBO dem Shader zuweisen
renderShader.assignSSBO(pRender);

//Partikelgeometrie zuweisen
renderPs->prepareGeometry(sphere_ptr);

//Renderschleife
while(!glfwWindowShouldClose(window)){
  (...)
  //Shader binden
  renderShader.use();

  //Instanziert zeichnen
  renderPs->drawGeometryInstanced();
  (...)
}
```

Quellcode 8: Schritte zum Rendern des Partikelsystems

Quellcode 8 zeigt die Schritte, die zum Rendern eines Partikelsystems notwendig sind. Zuerst werden die Geometrien, Shader und das Partikelsystem angelegt. Danach werden die Partikel anhand von der Geometrie von `sphere_ptr2` erzeugt und abgespeichert. Daraufhin wird ein SSBO mit den Partikeldaten erstellt und an den Shader gebunden. Anschließend wird eine Geometrie für die Partikel festgelegt. In diesem Fall erhalten alle Partikel die Geometrie von `sphere_ptr`. In der Renderschleife folgt dann das Binden der Shader und der Befehl zum Zeichnen.

Abbildung 7 zeigt das gerenderte Partikelsystem nach Quellcode 8. Die Partikelpositionen sind durch die in `sphere_ptr2` erstellte Kugel gegeben. Jedes einzelne Partikel ist als Kugel gerendert. Die Geometrie der Kugel ist

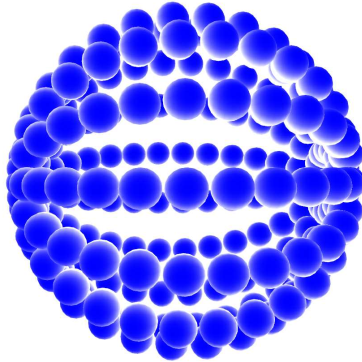


Abbildung 7: Gerendertes Partikelsystem nach Quellcode 8

als `sphere_ptr` angelegt worden ist. Zur besseren Unterscheidung einzelner Partikel wird *Rim-Shading* verwendet.

4.3 Simulations-Pipeline auf der GPU

Für die Simulation auf der GPU werden nacheinander die einzelnen Schritte der in Abschnitt 3 vorgestellten Pipeline ausgeführt. Jeder einzelne Schritt besteht dabei aus teils mehreren Compute-Shader-Aufrufen. Alle benötigten Daten und freie Felder zum Beschreiben werden zuvor in SSBOs gepackt und an die Shader gebunden, in denen sie gebraucht werden.

Vor dem Eintritt in die Simulationsschleife werden zunächst Vorberechnungen ausgeführt, um die Constraints nach zugehörigen Partikeln und Farben, siehe Abschnitt 3.5, zu gruppieren. Während der Simulation kann dadurch direkt auf die im aktuellen Schritt relevanten Constraints zugegriffen werden und es werden unnötige Iterationen über den Constraint-Buffer vermieden. Dabei wird das in Abschnitt 3.4 beschriebene Verfahren nach [Ble90] verwendet, um korrekte Offsets für den Zugriff zu berechnen. In Abschnitt 4.8.1 wird näher auf die Implementation der Offset-Berechnung eingegangen. Die Implementation der Constraint-Farben wird in Abschnitt 4.7.1 behandelt.

In der Simulationsschleife wird zuerst ein Integrationsschritt für jedes Partikel ausgeführt, wodurch die nächste Position eines Partikels errechnet wird, bevor diese durch die Constraints korrigiert wird. Zur Auswahl stehen hier ein Euler- und ein Verlet-Integrator.

Als nächstes werden die Nachbarschaften für die Fluid-Simulation berechnet. Da die Nachbarschaften sich mit jedem Simulationsschritt ändern können, müssen diese auch mit jedem Schritt neu berechnet werden. Der hier verwendete Ansatz ist ein *Uniform-Grid* in Verbindung mit den schon vorher erwähnten Offset-Berechnungen. Eine genaue Erläuterung zu der Implementation der Nachbarschaftssuche ist in Abschnitt 4.8.1 zu finden.

Im darauf folgenden Schritt befinden sich die in Abschnitt 3.2 vorgestellten Lösungsansätze. Es ist anzumerken, dass der vorgestellte

Jakobi-Ansatz hier als iteratives *Gauss-Jakobi*-Verfahren auf der GPU implementiert worden ist. Zudem wurden, nach Macklin *et al.* [MMCK14], sowohl eine constraint-zentrische, als auch eine partikelzentrische Variante des *Gauss-Jakobi*-Lösers umgesetzt. In Abschnitt 4.4 werden die Implementationen der Lösungsansätze deutlicher vorgestellt.

Nach dem Lösen der Constraints werden weitere, für die Simulation notwendige, Shader aufgerufen. Darunter befinden sich Korrekturen für die Integration und die Berechnungen von zusätzlichen Simulationsparametern, wie die Wirbelstärke und Viskosität für die Fluidsimulation (siehe Abschnitt 4.8).

Ein wichtiger Schritt für die Simulation ist an dieser Stelle eine Implementation des in [Jak03] vorgestellten *Box*-Constraint. Dieser sorgt dafür, dass alle Partikel in einem bestimmten Bereich bleiben und diesen nicht verlassen. Dadurch werden feste Wände und Böden simuliert, mit denen die Partikel in der Simulation interagieren. Es existiert zwar eine explizite Implementation des Box-Constraints, jedoch wurde sich entschieden, diesen einfach durch einen Shader zu ersetzen, welcher für jedes Partikel die in Quellcode 9 gezeigte Funktion ausführt.

```
1 vec4 cheatBox (vec4 p, vec4 minB, vec4 maxB){
2   vec4 updpos = vecMin4(vecMax4(p,minB),maxB);
3   return updpos;
4 }
```

Quellcode 9: Implementation des Box-Constraints

Sollte die Position eines Partikels außerhalb der durch `minB` und `maxB` festgelegten Grenzen liegen, wird die entsprechende Komponente der Position auf die jeweilige Komponente der übertretenen Grenze gesetzt. Die Funktionen `vecMin4()` und `vecMax4()` geben dabei den Vektor zurück, der durch Komponentenweises `min()` und `max()` der Eingabevektoren entsteht.

Hier endet die eigentliche Simulationspipeline. Es folgt das Updaten der benötigten Buffer. Dabei muss dies lediglich für die Buffer der Partikel, der Kamera und der ImGui-Nutzeroberfläche getan werden. Der Partikelbuffer enthält die neu errechneten Positionen, welche für das Rendering benötigt werden. Aus diesem Grund müssen Daten auf die CPU übertragen werden. Die Kamera und ImGui-Parameter werden auf der CPU geändert und werden deshalb auf die GPU übertragen. Nach dem Rendering des Partikelsystems, siehe Abschnitt 4.2.2, tritt die Simulation in die nächste Iteration ein.

4.4 Implementation der Lösungsverfahren

Wie in Abschnitt 4.3 angesprochen, sind sowohl ein *Gauss-Seidel*-, als auch ein *Gauss-Jakobi*-Verfahren zum Lösen der Gleichungssysteme auf der GPU implementiert worden. Das *Gauss-Jakobi*-Verfahren existiert in einer partikelzentrischen und einer constraint-zentrischen Variante, siehe [MMCK14].

Implementiert wurden die Lösungsverfahren jeweils in einzelnen Compute-Shadern, sodass jeder Constraint parallel auf der GPU gelöst werden kann. Über das Feld `int type` eines *UberConstraint*-Blocks, siehe Quellcode 5, wird der Typ eines Constraints herausgelesen. Durch ein `switch-case`-Statement wird unterschieden, welche Berechnungen für den Aufruf durchgeführt wird. Im Grunde folgt jede Berechnung dem in Abschnitt 3 vorgestellten Ansatz. Es wird jeweils der Gradient $\nabla C_i(x_0, ..., x_n)$ und der Korrekturfaktor λ_i des aktuellen Constraints berechnet. Anschließend werden darüber Änderungen für die betroffenen Partikel berechnet. Auf die genauen Berechnungen für die verschiedenen Constraint-Arten wird jeweils in Abschnitt 4.7 und Abschnitt 4.8 eingegangen. Die Anzahl der individuellen Solver-Iterationen lässt sich über die Nutzeroberfläche steuern.

4.5 Gauss-Seidel auf der GPU

Der Gauss-Seidel-Löser folgt einem constraint-zentrischen Ansatz. Das heißt, dass das Verfahren für jeden vorhandenen Constraint ausgeführt wird.

Im Fall des Gauss-Seidel Ansatzes werden diese Änderungen direkt mit der aktuellen Position der Partikel verrechnet.

Bei den constraint-zentrischen Ansätzen muss vor allem auf die Synchronisierung der Daten geachtet werden. Da Constraints parallel bearbeitet werden, kann es vorkommen, dass mehrere Threads gleichzeitig auf die selben Daten eines Partikels zugreifen. Im Falle des Lesens von Daten stellt das kein Problem dar, jedoch beim Beschreiben der Buffer.

Vor allem bei Einzelschrittverfahren, wie dem Gauss-Seidel-Verfahren, führt dies zu fehlerhaften Ergebnissen, da hier die in einer Iteration errechneten, neuen Werte direkt in der nächsten Iteration miteinbezogen werden.

Durch den parallelen Zugriff auf die Daten tritt an dieser Stelle eine Wettlaufsituation (engl. *race condition*) auf, da nicht garantiert ist, welcher Thread zuerst seine Berechnungen fertigstellt und die Daten beschreibt, wodurch andere Threads auf ungeeignete, bzw. inkonsistente Daten zugreifen. Dadurch kommt es zu unbeabsichtigten Ergebnissen [NM92].

Zwischen dem Aufrufen unterschiedlicher Shader kann dies CPU-seitig durch Speicherbarrieren (engl. *memory barrier*) abgefangen werden, wie zum Beispiel `glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT)`[7]. Dadurch können folgende Shader erst auf die Daten in den SSBOs zugreifen, wenn der Shader vor der Barriere fertig mit dem Beschreiben dieser ist.

[7]https://www.khronos.org/opengl/wiki/GLAPI/glMemoryBarrier, letzter Aufruf 2.9.2019

```
while(!glfwWindowShouldClose(window)){
  (...)
  //shader binden
  constraintSolverGaussSeidelCompute.use();

  //dispatch aufruf
  glDispatchCompute(constraintCount, 1, 1);

  //Speicherbarriere setzen
  glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
  (...)
}
```

Quellcode 10: Speicherbarriere zwischen verschiedenen Shadern

Quellcode 10 zeigt den Einsatz einer Speicherbarriere. Dadurch können folgende Shader erst auf den Speicher zugreifen, wenn der Compute-Shader `constraintSolverGaussSeidelCompute` seine Berechnungen beendet hat.

In ein und demselben Shader muss dies jedoch anders gehandhabt werden. In diesem Fall werden alle Constraints so gruppiert, dass gleichzeitig nur diejenigen Constraints parallel bearbeitet werde, die unterschiedliche Partikel verändern, also unterschiedliche Daten beschreiben.

Die Gruppierung erfolgt über das „Färben" von Constraints mit Hilfe von *Graph-Färbung*. Diese wurde bereits in Abschnitt 3.5 beschrieben, die Umsetzung ist in Abschnitt 4.7.1 zu fiden.

4.6 Gauss-Jakobi auf der GPU

Der Gauss-Jakobi-Löser ist, wie der Gauss-Seidel-Löser, ein iteratives Verfahren, welches auf die parallele Verarbeitung ausgelegt ist. Im Gegensatz zu dem Gauss-Seidel-Verfahren handelt es sich hierbei aber um ein Gesamtschrittverfahren. Das heißt, dass die neu errechneten Werte erst in der nächsten Iteration verwendet werden.

Deswegen werden die hier die Änderungen der Positionen in einem zusätzlichen SSBO gespeichert und aufaddiert. Zum Ende eines Simulationsschritts werden die aufsummierten Positionsänderungen parallel mit den eigentlichen Positionen der Partikel verrechnet.

```
struct DeltaData : structure {
  glm::vec4 delta;
  glm::vec4 posCache;
  float countPC;
  float countCC;
  float countContacts;
  float pad;
};
```

Quellcode 11: Das Delta-Struct

Quellcode 11 zeigt das Delta-Struct, was hauptsächlich im Gauss-Jakobi-Löser Anwendung findet. In `delta` werden die errechneten Positionsänderungen aufaddiert, `posCache` speichert direkte Positionsänderungen durch zum Beispiel Integration und die Kollision mit der Box. Das ist notwendig, da das errechnete Delta zum Ende durch die Anzahl der verarbeiteten Constraints geteilt wird, welche in den `count`-Feldern hochgezählt werden. Die direkten Änderungen sollen davon unbeeinflusst bleiben. Es existieren einzelne Felder für partikelzenstrische (`float countPC`) und constraint-zentrische (`float countCC`) verarbeitete Constraints. Für jedes Partikel gibt es ein solches Struct, die Zuweisung erfolgt über die Buffer-Position, d.h. das i-te `DeltaData` gehört zu dem i-ten Partikel.

Nachdem die Constrainst verarbeitet worden sind, folgt eine Relaxierung des Gesamtdeltas. Diese ist notwendig, da Jakobi-Verfahren dazu tendieren, zwischen Lösungen zu oszillieren, wie in [MMCK14] beschrieben. Damit ist eine Konvergenz des Verfahrens nicht garantiert. Wie in Abschnitt 3.2 wird die Relaxierung jedes Partikels in einem lokalen (Unterrelaxierung) und in einem globalen Schritt (Überrelaxierung) durchgeführt.

Im lokalen Schritt wird die Gesamtänderung eines Partikels einfach durch die Anzahl der auf das Partikel wirkenden Constraints geteilt, sodass sich die Änderung schlicht mittelt. Wie weiter oben beschrieben, sind dafür die `count`-Felder in `DeltaData` da, die die Anzahl der wirkenden Constraints hochzählen.

Die globale Relaxierung ist ein ein einziger globaler Parameter, der nach dem lokalen Schritt an das unterrelaxierte Ergebnis heran multipliziert wird, um die Mittlung abzuschwächen. Dieser kann durch die Nutzeroberfläche zur Laufzeit verändert werden. Mackiln *et al.* empfehlen dabei einen Parameter in dem Intervall $[1; 2]$ [MMCK14]. Ein Wert von ≤ 1 führt dabei zu zusätzlicher Unterrelaxierung.

Die Relaxierung und das letztendliche Update der Partikelpositionen findet in zwei Shadern statt. Im ersten Shader finden beide Relaxierungsschritte statt. Das Ergebnis wird auf `posCache` in `DeltaData` addiert.

```
void main(){
  //Sicherheitsvorkehrung: wenn count ungleich 0...
  (deltas[coord].count != 0)?

  //... teile das delta durch count und addiere auf posCache...
  (deltas[coord].posCache +=
      imgui.overRelaxation*deltas[coord].delta /
      deltas[coord].count):

  //...ansonsten addiere delta einfach auf posCache
  (deltas[coord].posCache +=
      imgui.overRelaxation*deltas[coord].delta);
}
```

Quellcode 12: Lokale und globale Relaxierung auf der GPU

Quellcode 12 zeigt die lokale und globale Relaxierung. Um ein Teilen durch 0 abzufangen, wird überprüft, ob mindestens ein Constraint gewirkt hat. Wenn nicht, wird das delta ohne Division, welches in dem Fall ein `vec4(0.0f)` sein sollte, auf `posCache` addiert.

Zusätzlich zur Relaxierung wird in dieser Arbeit ein weiterer Parameter für eine allgemeine Abschwächung der Positionsänderung eingeführt. Dieser wird auf die Gesamtposition inklusive Integration und Box-Kollision angewandt, was zu einem stabilerem Ergebnis führt.

```
void main(){
  (...)
  //Wende globale Relaxierung auf die neu errechnete Position an
  particles[coord].position = imgui.damping*deltas[coord].posCache;
  (...)
}
```

Quellcode 13: Globale Relaxierung auf der GPU

Quellcode 13 zeigt die abschließende Abschwächung. Der nutzergesteuerte Parameter `imgui.damping` wird auf die komplette neue Position multipliziert. Anfänglich zieht dies die Partikel zwar in die Ecken der Box, jedoch führt dies auch während der Simulation zu stabileren Ergebnissen. Dieser Parameter hat sich aus dem Experimentieren mit der Anwendung der globalen Relaxierung ergeben.

Partikelzentrischer Gauss-Jakobi Bei der partikelzentrischen Variante des Verfahrens gibt es einen Thread für jeden Partikel. In jedem werden dann alle Constraints gelöst, die den Partikel beeinflussen.

Der naive Ansatz wäre hier, für den aktuellen Partikel `p` über alle Constraints `c` zu iterieren und zu überprüfen, ob ein `c` den Partikel beeinflusst.

Hier bedeutet das soviel wie zu überprüfen, ob `p` ein Element von `c.indices[]` ist.

Um das Suchen der zugehörigen Constraints effizient zu halten, werden die Constraints in der Vorverarbeitung, siehe Abschnitt 4.3, so in eine Liste einsortiert, dass alle für ein Partikel wichtigen Constraints hintereinander stehen. Über die in Abschnitt 3.4 vorgestellte Präfixsummenberechnung werden Start- und Endoffsets errechnet. Dadurch steht für jeden Partikel fest, wo die Constraints in dem sortierten Buffer liegen, und somit auch, auf welche in der Liste der Constraints zugegriffen werden muss. Da die Constraints in dem sortierten Buffer hintereinander liegen, kann auf der GPU in eine Schleife über den Bereich, den Start- und Endoffset angeben, iteriert werden und die Constraints können nacheinander, wie beschrieben, in Abhängigkeit ihres Typs gelöst werden.

```
1  void main(){
2    //index des Partikels, ermittlung von Start- und Endoffset
3    uint coord = gl_GlobalInvocationID.x;
4    int start = startInd[coord];
5    int end = start + count[coord];
6
7    //vermeide endlosschleife
8    if(start > end){
9      return;
10   }
11
12   //iteriere ueber constraints...
13   for(int i = start; i<end;i++){
14     //...loese Constraint an constraints[sorted[i]], addiere delta
           auf
15 }}
```

Quellcode 14: Struktur des partikelzentrischen Gauss-Jakobi

Quellcode 14 zeigt den Aufbau des partikelzentrischen Gauss-Jakobi-Solvers. In dem Buffer `int startInd[]` liegen die Startindizes aus der Präfixsummenberechnung. In einem anderen Buffer `int count[]` steht die Anzahl der Constraints, die das `p`-te Partikel beeinflussen. Aus den Einträgen `startInd[p]` und `count[p]` ergibt sich der exklusive Endindex für die sortierte Constraintliste für Partikel `p`. Bevor über die Indizes iteriert werden kann, sollte sicherheitshalber getestet werde, ob die Indizes valide sind. Sollte `start > end` gelten, wird die folgende Schleife endlos ausgeführt. Letzendlich wird über die Indizes iteriert und der Constraint an Stelle `constraints[sorted[i]]` kann bearbeitet werden.

Es ist jedoch bei dem Anlegen der Buffer zu beachten, dass der Buffer für die sortierten Constraint-Indizes groß genug ist. Da ein Constraint mehrere Partikel beeinflussen kann, wie z.B. 2 im Fall von Distanz-Constraints, wird ein Constraint auch mehreren Partikeln zugeordnet und dieser wird somit

mehrmals in die sortierte Liste einsortiert. Letztendlich sollte der zugehörige Buffer also so groß sein, wie die Anzahl der Partikel, die insgesamt von allen Constraints beeinflusst werden, unabhängig von Doppelungen.

Constraint-zentrischer Gauss-Jakobi Der constraint-zentrische Ansatz des Gauss-Jakobi-Verfahrens ist dem Gauss-Seidel-Verfahren ähnlich. Es wird für jeden Constraint ein Thread gestartet. Der Constraint wird dann auf der GPU gelöst und die Positionen der beeinflussten Partikel werden geupdated.

```
void main(){
  //index des aktuellen constraints
  uint coord = gl_GlobalInvocationID.x;

  //erstelle benoetigte Variablen
  vec4 update;
  (...)

  switch(constraints[coord].type){
    //loese constraint nach typ t, schreibe in update
  }

  //iteriere ueber indizes und update deltas aller Partikel
  for (int j = 0; j < constraints[coord].indices.length();j++){

    //aktualisiere das delta ueber atomicAdd
    atomicAdd(deltas[currentParticleIndex].delta,update);
}}
```

Quellcode 15: Struktur des constraint-zentrischen Gauss-Jakobi

Quellcode 15 zeigt den Aufbau des constraint-zentrischen Gauss-Jakobi-Lösers. Wie bei Gauss-Seidel ist es möglich, dass mehrere Constraints die selben Partikel beeinflussen. Macklin *et al.* lösen hier das Synchronisierungsproblem, indem die zugehörigen Deltas über *atomic*-Operationen aktualisiert werden, siehe [MMCK14]. Damit dies jedoch funktioniert, muss zuvor eine zusätzliche Erweiterung für atomische `float` Operationen aktiviert werden, da diese standardmäßig nicht unterstützt werden. In diesem Fall handelt es sich um die Erweiterung `GL_NV_shader_atomic_float`.

Eine andere Lösung für das Synchronisierungsproblem sind an dieser Stelle die schon in Abschnitt 4.5 angesprochenen Constraint-Farben. Diese gruppieren schliesslich die Constraints so, dass in einer Gruppe von Constraints nur diejenigen vorhanden sind, die unterschiedliche Partikel beeinflussen.

Stehen die Farben der Constraints bereits fest, können diese auf die selbe Weise gruppiert werden, wie für die Nachbarschaftssuche oder die Partikelgruppierung für den partikelzentrischen Ansatz.

Die Zuordnung an dieser Stelle ist Constraints zu Farben. Der `count`-Buffer ist so groß, wie die Anzahl an Farben. In diesem wird hochgezählt,

wie oft eine Farbe auftaucht. Anschließend werden Offsets berechnet und die Constraints so in einen Buffer einsortiert, dass die zu einer Farbe gehörigen Constraints hintereinander stehen.

Der Löser wird dann so oft aufgerufen, wie es Farben gibt. Die aktuelle Farbe wird als Uniform-Variable an den Shader übergeben und die entsprechenden Constraints werden behandelt.

```
for (int col = 0; col<= maxColor; col++) {
  constraintSolver.setInt("u_color", col);
  constraintSolver.use();
  glDispatchCompute(constraintCount, 1, 1);
  glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
}
```

Quellcode 16: Aufruf eines Lösers mit Constraint-Farben

Quellcode 16 zeigt den CPU-seitigen Aufruf des Lösers mit Constraint-Farben. In der `for`-Schleife wird die Farbe hochgezählt und über den Befehl `setInt()` als Uniform-Variable an den Shader übergeben.

```
uniform int u_color;

void main(){
  //index des Constraints, ermittlung von Start- und Endoffset
  uint coord = gl_GlobalInvocationID.x;

  //pruefe, ob coord behandelt werden darf
  if (coord > count[u_color]){
    return;
  }

  //Position des Constraints im Constraint-Buffer
  int constraintCoord = sorted[startInd[u_color]+coord];

  //loese nach Typ des Constraints an Stelle
      constraints[constraintCoord]
  //aktualisiere Position
  (...)
}
```

Quellcode 17: Struktur des farbbasierten Lösers

Quellcode 17 zeigt das Gerüst eines farbbasierten Lösers. Über die aktuelle Farbe `u_color`, die Aufrufs-ID `gl_GlobalInvocationID.x` und den Start-Offset der Farbe in dem Buffer `int startInd[]` kann auf den Index des zu behandelnden Constraints im sortierten Buffer `int sorted[]` zugegriffen werden. Der Constraint an dieser Stelle des Constraint-Buffers wird in diesem Aufruf gelöst.

4.7 Implementation der Festkörper

Wie in Abschnitt 3.3.1 beschrieben, werden Festkörper über Distanz-Constraints zwischen allen zu dem Körper gehörigen Partikeln simuliert. Das heißt, dass für ein Objekt, das durch n-viele Partikel dargestellt wird, ohne Dopplung von Constraints, eine Anzahl von $\sum_{i=0}^{n-1} i$-vielen Constraints zu lösen sind.

In einem Struct vom Typ `UberConstraintData` besetzte Parameter sind hier `type` mit 1 für die Markierung als Distanz-Constraint, `indices` mit den Indizes der beeinflussten Partikel $\{p_i, p_j\}$ und `dist` mit dem euklidischen Abstand der Partikel $\{p_i, p_j\}$ in der Startkonfiguration. Darüber können auf die Werte für die Constraint-Berechnung zugegriffen werden.

Da in diesem Fall mehrere Constraints gleichzeitig auf ein Partikel einwirken, muss für den in Abschnitten 3.2 und 4.5 behandelten Gauss-Seidel-Löser ein zusätzlicher Synchronisierungsschritt vorgenommen werden, um Wettlaufbedingungen der GPU-Threads zu vermeiden. Dafür wird die in Abschnitt 3.5 behandelte Graph-Färbung verwendet.

In den folgenden Abschnitten wird zunächst auf die Implementation der Graph-Färbung eingegangen, siehe Abschnitt **??**, danach wird das Verringern der Anzahl von verwendeten Constraints besprochen und abschließend das explizite Lösen der Distanz-Constraints auf der GPU, siehe Abschnitt 4.7.3. Die eigentlichen Implementationen der Lösungsverfahren werden in Abschnitt 4.4 besprochen.

4.7.1 Graph-Färbung

Die Graph-Färbung wird dazu verwendet, Constraints so zu gruppieren, dass in einer Gruppe von Constraints nur solche enthalten sind, die unterschiedliche Partikel behandeln. Das Grundprinzip hinter der Graph-Färbung wurde bereits in Abschnitt 3.5 beschrieben. Im Weiteren wird zunächst der Aufbau des Graphen aus Constraints und Partikeln besprochen, abschließend der Algorithmus zum Färben des Graphen und der Constraints.

Aufbau des Graphen Die Konstruktion des Graphen folgt der in der Arbeit von Fratarcengeli und Pellacini beschriebenen Methode [FP15a]. Jeder Constraint wird in dem Graph durch einen Knoten repräsentiert. Zwei Knoten werden durch eine Kante Verbunden, wenn sich die entsprechenden Constraints mindestens einen Partikel teilen.

Im Framework ist der Graph als eigenständige Klasse implementiert. Als Eingabeparameter erhält der Graph eine Liste von Constraints, die die Knoten des Graphs darstellen. Der vollständige Graph wird zunächst als zweidimensionaler Vektor der Größe $n \times n$ initialisiert, wobei n der Anzahl der Knoten entspricht. Dies ist nötig, da das Berechnen der Nachbarschaften der Knoten mit Hilfe der Bibliothek *OpenMP* parallel auf der CPU ausgeführt wird und deshalb der Speicher von Beginn an allokiert sein muss. Für jeden

Knoten `i` wird ein Thread gestartet. Dieser prüft für jeden anderen Knoten `j`, ob es eine Überschneidung zwischen den Indizes von `i` und `j` gibt. Die gefundenen Nachbarn `j` werden hintereinander in den Kontainer von `i` geschrieben. Ein Parameter zählt hoch, wie viele Nachbarn gefunden worden sind, sodass über diesen die Liste der Nachbarn verkleinert werden kann.

```
1 bool Graph::checkOverlap(std::vector<int>& c1, std::vector<int>&
    c2) {
2   std::vector<int> intersect;
3   std::set_intersection(c1.begin(), c1.end(), c2.begin(), c2.end(),
      std::back_inserter(intersect));
4   return !intersect.empty();
5 }
```

Quellcode 18: Funktion zum Überprüfen auf Überschneidung von zwei Vektoren

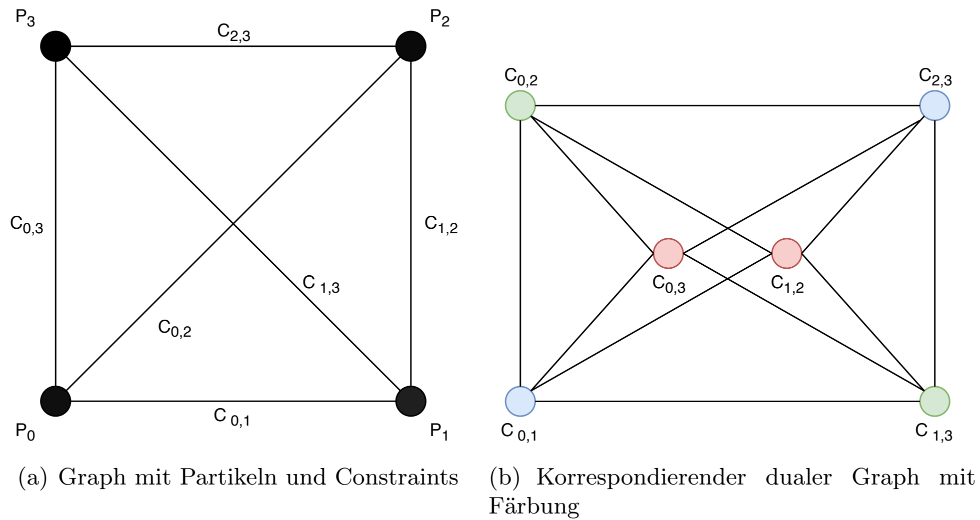
Quellcode 18 zeigt die Funktion, mit der auf eine Überschneidung der Indizes geprüft wird. Es wird zunächst ein leerer Container angelegt. In diesen werden die gleichen Einträge der Eingabevektoren über die Funktion `std::back_inserter(intersect)` eingetragen, wobei diese Funktion die Größe des Containers `intersect` erhöht und das Schnittelement an dieser Stelle einträgt, wenn eine Überschneidung über `std::set_intersection()` gefunden worden ist. So kann zum Schluss durch die Abfrage, ob `intersect` leer ist, entschieden werden, ob es eine Überschneidung gibt, oder nicht.

Färbung der Constraints Ausgehend von den gefundenen Nachbarschaften kann der Graph, und somit die Constraints, koloriert werden. Als Grundlage dient hier die in [FP15a] vorgestellte Greedy-Heuristik. Dabei wird zunächst für jeden Knoten die Farben der Nachbarn ermittelt. Das sind diejenigen Farben, die der Knoten nicht haben darf. Die Farben werden durch Zahlen, angefangen bei 0, repräsentiert. Die Farbe des aktuellen Knoten wird dann auf die kleinstmögliche Farbe gesetzt.

In der Arbeit wird dies auf folgende Weise gelöst. Zunächst wird ein Objekt vom Typ `std::vector<glm::ivec2>` angelegt. Dieses wird mit `n` Einträgen gefüllt. Jeder Eintrag entspricht dem Index eines Knotens und der zugewiesenen Farbe, welche als `int` repräsentiert wird. Die Farbe wird zunächst für alle Einträge auf `MAXINT` gesetzt, ein zusätzlicher Parameter überwacht die aktuell kleinste mögliche neue Farbe.

In einer Schleife werden alle Knoten nacheinander abgearbeitet. Die aktuellen Farben der Nachbarn werden in einem Vektor `std::vector<int> col` gespeichert. Über die Funktionen `std::sort()` und `std::unique()` wird der `col` so verarbeitet, dass jede Farbe der Nachbarn aufsteigend sortiert nur einmal vorkommt. Das ist die Liste der für den Knoten verbotenen Farben.

Die kleinste für den Knoten mögliche Farbe kann nun sehr einfach ermittelt werden. Es wird über die Liste der verbotenen Farben iteriert. Da die



(a) Graph mit Partikeln und Constraints

(b) Korrespondierender dualer Graph mit Färbung

Abbildung 8: Nach [FP15a]. Links ist der Graph mit Partikeln als Knoten und Constraints als Kanten. Rechts ist der korrespondierende Duale Graph mit mit Constraints als Knoten, inklusive Färbung

Einträge einzigartig und sortiert sind, wird angenommen, dass von Beginn an eine Serie an natürlichen Zahlen inklusive 0 existiert, also dass `col[i]==i`.

An der ersten Stelle, an der diese Annahme nicht mehr hält, endet die Serie und die Farbe wird auf den Index gesetzt, an dem die Bedingung gebrochen ist. Sind die ermittelten verbotenen Farben beispielsweise `col=[0,1,4,5,...]`, bricht die Serie an Index 2, da `col[2]!=2`. Somit ist `2` die kleinste für den Knoten verfügbare Farbe und der Knoten wird auf diese Farbe gesetzt.

Abbildung 8 zeigt den auf diese Weise aufgebauten und kolorierten dualen Graphen. Abbildung 8(a) zeigt den Ausgangsgraphen. Die Knoten stellen Partikel dar, die Kanten Constraints. Es existiert eine Kante zwischen zwei Knoten, wenn die Partikel von einem Constraint beeinflusst werden. Abbildung 8(b) zeigt den korrespondierenden dualen Graphen. Die Knoten sind dieses mal Constraints, welche durch den beschriebenen Algorithmus eingefärbt worden sind. Eine Kante symbolisiert, dass die Constraints mindestens einen gemeinsamen Partikel beeinflussen.

Dies gefundene Kolorierung kann nun auf die Constraints übertragen werden. Dafür wird die gefundene Farbe, repräsentiert als `int`, in das Feld `int color` des in Quellcode 5 dargestellten `UberConstraintData`-Structs geschrieben.

Zusätzlich ist es möglich, eine gefundene Färbung zu serialisieren und dezuserialisieren. Mit Hilfe der Bibliothek *JSON for Modern C++* wird eine JSON-Datei angelegt, in der die Kolorierung gespeichert wird. Liegt eine solche Datei vor, kann diese geladen und direkt auf die vorhandenen

Constraints übertragen werden. Dadurch fällt das Erstellen und Kolorieren des Graphen weg.

4.7.2 Constraint-Dezimierung

Wie weiter oben beschrieben, werden solide Körper in dieser Arbeit mit Distanz-Constraints zwischen jedem Partikel zu jedem anderen beschrieben. Es stellt sich an dieser Stelle die Frage, ob diese Menge an Distanz-Constraint überhaupt benötigt wird, um den Körper in seiner Form zu halten. Im Grunde genommen ist der Ansatz, keine Constraints doppelt zu erstellen, schon eine Art der Dezimierung. Im naiven Ansatz würde zwischen jedem Partikel zu jedem anderen Partikel ein Distanz-Constraint erstellt werden. Demnach werden für ein Objekt mit einer Anzahl von n Partikeln eine Anzhal von n^2 Distanz-Constraints erstellt.

Durch das Entfernen von doppelten Constraints reduziert sich die Anzahl, wie schon erwähnt, auf $\sum_{i=0}^{n-1} i$ Distanz-Constraints. Dadurch bilden die Constraints die konvexe Hülle des Objekts, sowie Querstreben innerhalb des Objekts.

```
void createDistanceConstraints(std::vector<Constraint>& vec) {
  //gehe ueber alle Partikel i des Partikelsystems
  for (int i = 0; i < m_count; i++) {
  glm::vec4 currentPos = m_particlePointer->at(i).position;

  //fuege Constraint zu jedem anderen Partikel j hinzu
  for (int j = i + 1; j < m_count; j++) {
    vec.emplace_back(1, 1.0f, true, std::vector<int>{i, j},
     std::make_shared<DistanceEq>(glm::distance(
      m_particlePointer->at(i).position,
      m_particlePointer->at(j).position)));
}}}
```

Quellcode 19: Funktion zum Erstellen von Distanz-Constraints ohne Dopplungen

Quellcode 19 zeigt das Erstellen von Distanz-Constrinst für ein Partikelsystem. Dadurch gibt es zwischen allen Partikeln p_i und p_j nur jeweils einen Distanz-Constraint. So entstehen die konvexe Hülle außerhalb und Querstreben innerhalb des Partikelsystems.

Ein weiterer Ansatz entsteht aus der Idee, die Anzahl der inneren Querstreben zu reduzieren, dabei aber die konvexe Hülle möglichst zu erhalten. So soll es Distanz-Constraints innerhalb der Nachbarschaft eines Partikels geben und jeweils nur Querstreben zu weit entfernten Partikeln. Diese sollen dabei zur Stabilisation des Objektes dienen.

Abbildung 9 zeigt die Idee verbildlicht. Es werden Distanz-Contraints zu den Partikel dargestellten im Umkreis erstellt, sowie zu einem anderen Partikel, der weit entfernt ist.

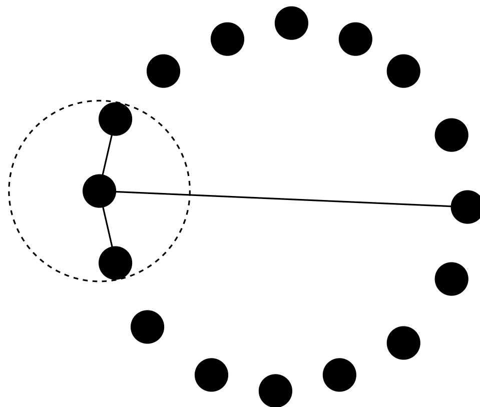


Abbildung 9: Wahl der Partikel für Distanz-Constraints bei der Constraint-Dezimierung.

```
void createDistanceConstraintsDecimated(std::vector<Constraint>&
    vec, float maxDistance, float furthest, int farMax) {
  //gehe ueber alle Partikel i des Partikelsystems
  for (int i = 0; i < m_count; i++) {
  glm::vec4 currentPos = m_particlePointer->at(i).position;
  int count = 0;

  //fuege Distanz-Constraint zu j hinzu...
  for (int j = i + 1; j < m_count; j++) {
    float dist = glm::distance(m_particlePointer->at(i).position,
        m_particlePointer->at(j).position);

    //...wenn die Distanz zwischen i und j klein genug ist
    if (dist < maxDistance) {
    vec.emplace_back(1, 1.0f, true, std::vector<int>{i, j},
        std::make_shared<DistanceEq>(dist));
    }

    //...wenn die Distanz zwischen i und j gross genug ist
    else if (dist > furthest && count <= farMax) {
    vec.emplace_back(1, 1.0f, true, std::vector<int>{i, j},
        std::make_shared<DistanceEq>(dist));
    count++;
}}}}
```

Quellcode 20: Funktion zum Erstellen von Constraints zu Partikeln in der Nähe und von wenigen Querstreben

Quellcode 20 realisiert die Idee. Es werden Distanz-Constraints von einem Partikel zu allen anderen Partikeln im Radius von `maxDistance` erstellt, sowie zu maximal `farMax`-vielen Partikeln, die mindestens eine Entfernung

von `furthest` haben.

Der Nachteil an dieser Variante der Dezimierung ist jedoch, dass es doppelte Constraints gibt. Diese entstehen durch das Verbinden von Partikeln in der jeweiligen Nachbarschaft. dafür werden jedoch die Anzahl der Constraints, die durch das Partikelsystem laufen, stark reduziert, solange `farMax` entsprechend klein gewählt wird.

4.7.3 Lösen der Distanz-Contraints

Das Lösen der Distanz-Constraints folgt den in Abschnitt 3.3.1 vorgestellten Formeln. Es ist zu beachten, dass ein Distanz-Constraint, wie erwähnt, mehrere Partikel beeinflusst. Je nachdem, welches Lösungsverfahren verwendet wird, muss zusätzlich auf die Synchronisierung auf der GPU geachtet werden.

Wie in Abschnitt 3.3.1 gezeigt, hat ein Distanz-Constraint die Form $C(p_i, p_j) = |p_j - p_i| - d$, dabei sind p_i und p_j die aktuellen Positionen der Partikel, jeweils aus `particleData.position` und d der Abstand der Partikel aus ihrem Constraint `c` an der Stelle `c.dist`.

Die Formeln zur Berechnung des Constraint-Fehlers λ_i und der eigentlichen Positionsänderung werden in den Shader übernommen.

```
1  case 1:
2    //Abstaende
3    vec4 deltaPosition = p1.position-p2.position;
4    float deltalength = length(deltaPosition);
5
6    //Gradient berechnen
7    vec4 partials[2];
8    partials[0] = delta/deltalength;
9    partials[1] = -delta/deltalength;
10
11   //Inverse Masse
12   float im1 = 1.0f/p1.mass;
13   float im2= 1.0f/p2.mass;
14
15   //Constraint-Fehler
16   float scaling = (deltalength - c.dist) / (im1+im2);
17
18   //update der Positionen
19   vec4 up1 = scaling * im1 * partials[0];
20   vec4 up2 = scaling * im2 * partials[1];
21
22   //aktualisiere position oder delta der Partikel
23   (...)
```

Quellcode 21: Lösen eines Distanzconstraints auf der GPU

Quellcode 21 zeigt das Lösen der Distanz-Constraints im Shader. Die Berechnungen folgen den in Abschnitt 3.1 und Abschnitt 3.3.1 vorgestellten Formeln. Die letztendliche Aktualisierung der Positionen erfolgt lösungsverfahrenspezifisch.

4.8 Implementation der Fluide

Fluide werden, wie in Abschnitt 3.3.2 beschrieben über einen Dichte-Constraint pro Partikel dargestellt. Für eine aus aus p-vielen Partikeln bestehende Flüssigkeit sind demnach p-viele Constraints zu lösen.

Die in `UberConstraintData` wichtigen besetzten Parameter sind hier `type` mit 2 und `indices` mit dem Index des beeinflussten Partikels p_i und den Indizes der benachbarten Partikel $\{p_0^n, ..., p_n^n\}$.

In den folgenden Abschnitten wird zunächst auf Implementation der Nachbarschaftssuche eingegangen, anschließend auf das explizite Lösen der Dichte-Constraints. In beiden Fällen liegt eine Implementation auf der GPU vor.

4.8.1 Nachbarschaftssuche

Da das Lösen von Dichte-Constraints von der Nachbarschaft des zugehörigen Partikels abhängt, muss diese zunächst berechnet werden. Dafür ist in der Arbeit ein *Uniform-Grid* auf der GPU implementiert worden. Im Speicher ist das Grid als eindimensionaler Container abgelegt, auf den über Offset-Berechnungen zugegriffen wird. Dafür werden die in Abschnitt 3.4 vorgestellten Präfix-Operationen verwendet.

Im Folgenden wird zunächst die Implementation des *Uniform-Grids*, inklusive Offsetberechnung und Nachbarschaftssuche vorgestellt. Die Implementationen und Ausführungen finden vollständig auf der GPU statt.

Uniform-Grid Bei einem Uniform-Grid handelt es sich um eine Datenstruktur, die einen Bereich in Zellen aufteilt. Dabei hat jede Zelle die selbe Größe. Das Grid ist somit definiert über die Anzahl der Zellen in x, y und z-Richtung, sowie einen Parameter für die Seitenlänge einer Zelle.

```
1 struct GridData : structure {
2   float cellSize;
3   int cellsX;
4   int cellsY;
5   int cellsZ;
6 };
```

Quellcode 22: Struct mit Grid-Parametern

Quellcode 22 zeigt das Struct, in dem die Parameter des Grids abgelegt werden.

Zusätzlich werden zwei eindimensionale Buffer benötigt. In dem Buffer `int count[]` mit der Größe `cellsX*cellsY*cellsZ` werden die Partikel gezählt, die in die Zellen Fallen. In einem anderen Buffer `int sorted[]` der Größe $\#Partikel$ werden die Indizes der Partikel so einsortiert,dass die Indizes aller Partikel einer Zelle hintereinander stehen. Über Offset-Berechnungen kann dann der Bereich einer Zelle ermittelt werden. Alle Buffer liegen dabei als SSBO auf der Grafikkarte.

Im ersten Schritt wird `count[]` gefüllt. Für jeden Partikel wird anhand der Position seines Mittelpunktes berechnet, in welche Zelle dieser fällt. Die Berechnungen dafür stammen aus [HVDM*14].

```
1 ivec3 pointInCell(vec3 p, float size) {
2  ivec3 r;
3  r.x = int(floor(p.x / size));
4  r.y = int(floor(p.y / size));
5  r.z = int(floor(p.z / size));
6  return r;
7 }
```

Quellcode 23: Nach [HVDM*14]. Berechnung der Partikel-Zelle

Quellcode 23 zeigt die Berechnung der Zelle, in die ein Partikel fällt, `p` ist die Position des Partikels, `size` ist die Zellengröße. Wenn die so ermittelte 3D-Zelle im Grid liegt, muss diese auf einen Index `ci` für den Buffer `count` umgerechnet werden.

```
1 int cellIndex (ivec3 index, int x, int y){
2     return (index.x + x * (index.y + y * index.z));
3 }
```

Quellcode 24: Berechnung des Bufferindex einer Zelle. Nach [HVDM*14]

Das übernimmt die in Quellcode 24 gezeigte Funktion. Eingabe Parameter sind der errechnete 3D-Zellenindex, `cellsX` und `cellsY`, also Parameter des Grids.

Anschließend wird der `count[ci]` um 1 erhöht. Da diese Berechnung parallel für alle Partikel ausgeführt wird, geschieht dies über `atomicAdd()`.

Zum Einsortieren der Partikel in den Buffer `sorted` werden die Schritte wiederholt. Der Partikelindex wird dann an die Stelle `sorted[startInd[ci]+prev]` geschrieben, `startInd[ci]` ist dabei der durch Offset-Berechnungen ermittelte Startindex der Zelle `ci`, siehe Abschnitt 4.8.1. `prev` ist der Rückgabewert von `atomicAdd(count[ci],1)`, also der Wert, der in `count[ci]` vor dem `atomicAdd()` steht.

Abbildung 10 zeigt ein 2D-Uniform-Grid mit entsprechender Umwandlung in einen eindimensionalen Buffer.Die Zeilen des Grids werden hintereinander abgelegt. Die in die Zellen fallenden Partikel werden hochgezählt.

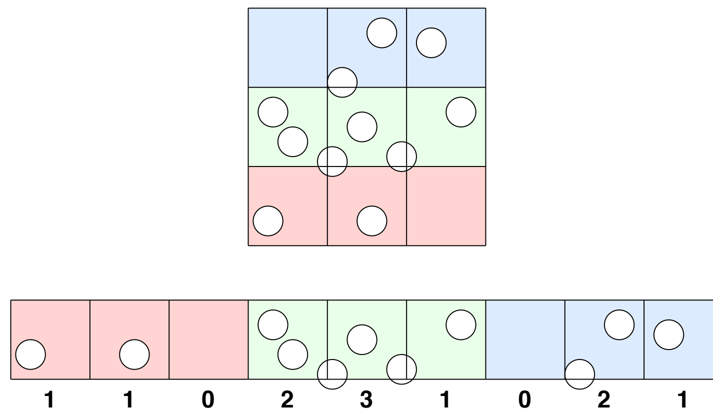


Abbildung 10: Ein 2D-Uniform-Grid wird als eindimensionaler Buffer repräsentiert.

Offset-Berechnungen Die Offset-Berechnung richtet sich nach der in Abschnitt 3.4 vorgestellten Präfixsummenberechnung und wird auf dem hochgezählten Buffer `count[]` durchgeführt, welcher für die Berechnung in einen anderen Buffer `int startInd[]` kopiert wird.

Da der hier implementierte Algorithmus auf einem balancierten Baum arbeitet, siehe [Ble90], wird der Buffer mit einer Größe der nächsthöheren Zweierpotenz der Größe von `count[]` angelegt.

Es ist das Ziel, Start- und End-Offsets der Zellen für den Zugriff auf `sorted[]` zu errechnen.

Dafür wird die in Quellcode 25 gezeigte Pipeline verwendet.

```
//upsweep
for (int up = 0; up < treedepth; up++) {
  upsweepCompute.setInt("u_depth", up);
  upsweepCompute.use();
  glDispatchCompute(cellCount, 1, 1);
}

//setze letztes Element des Buffers auf 0 fuer den downsweep
(...)

//downsweep
for (int dwn = treedepth - 1; dwn >= 0; dwn--) {
  downsweepCompute.setInt("u_depth", dwn);
  downsweepCompute.use();
  glDispatchCompute(cellCount, 1, 1);
}
```

Quellcode 25: Pipeline für die Offset-Berechnung nach [Ble90]

`treedepth` ist dabei die Tiefe des Baums, der über den Buffer gelegt wird

und steuert den Zugriff auf die Positionen, die miteinander verrechnet werden. Die Shader für *upsweep* und *downsweep* werden `treedepth`-mal aufgerufen, die aktuelle Tiefe wird als Uniformvariable an den Shader übergeben. Dadurch können die Shader jeweils parallel ausgeführt werden.

```
uniform int u_depth;
//upsweep
void main(){
  if(int(mod(coord,pow(2,u_depth+1)))==0){
  startInd[int(coord+pow(2,u_depth+1)-1)] =
      startInd[int(coord+pow(2,u_depth)-1)] +
      startInd[int(coord+pow(2,u_depth+1)-1)];
}}
//downsweep
void main(){
  if(int(mod(coord,pow(2,u_depth+1)))==0){
    int temp = startInd[coord+int(pow(2,u_depth))-1];
    startInd[coord+int(pow(2,u_depth))-1] =
        startInd[coord+int(pow(2,u_depth+1))-1];
    startInd[coord+int(pow(2,u_depth+1))-1] = temp +
        startInd[coord+int(pow(2,u_depth+1))-1];
}}
```

Quellcode 26: Implementation von upsweep und downsweep im Shader nach [Ble90]

Quellcode 26 zeigt die GPU-Implementation von *upsweep* und *downsweep* nach [Ble90]. Die Uniform-Variable `u_depth` steuert den Zugriff auf die korrekten Positionen für jeden Aufruf.

Nach dem letzten *downsweep* stehen die korrekten Start-Offsets der Zellen für den Zugriff auf `sorted[]` in `startInd[]`. Der exklusive End-Offset lässt sich letztendlich durch Addition der Partikel pro Zelle aus `count[]` und dem Start-Offset ermitteln.

Es ist hervorzuheben, dass dieser Ansatz nicht nur für das Uniformgrid geeignet ist. Derselbe Prozess wird in der Arbeit zudem als Vorberechnung für die Zuweisung von Constraint zu Partikel für den partikelzentrischen Gauss-Jakobi-Löser und der Zuordnung von Constraint zu Farbe für den Gauss-Seidel-Löser und den constraint-zentrischen Gauss-Jakobi-Löser verwendet, um auf die richtigen, bzw. geeigneten Constraints zugreifen zu können, siehe Abschnitt 4.4.

Nachbarschaft Da nun in `sorted[]` korrekt in die Zellen einsortierte Partikel stehen und die entsprechenden Start- und End-Offsets durch `startInd[]` und `count[]` ermittelt werden können, ist es möglich, Partikelnachbarschaften über das Uniformgrid zu bestimmen.

Die Suche wird für jedes Partikel `p` aufgerufen. Ein weiteres mal wird die Zelle, in die `p` fällt bestimmt. In einer verschachtelten Schleife wird über alle

Zellen iteriert, die in einem Suchradius liegen, der durch Nutzereingaben angepasst werden kann. Die 27er-Nachbarschaft würde durch einen Suchradius von 1 abgearbeitet werden.

Liegt die aktuelle Zelle im Grid, werden die Offsets der Zelle wie oben beschrieben ermittelt. Die in der Zelle liegenden Partikel sind Nachbarn von `p`. Dies wird für alle Zellen im Suchradius durchgeführt.

Die so ermittelten Partikelindizes werden für die Fluid-Simulation in `indices[]` des zu `p` gehörigen Dichte-Constraints geschrieben.

```
void main(){
  //Zelle in die das aktuelle Partikel p faellt
  ivec3 i = pointInCell(p.position, gridSize);

  // Gehe ueber Nachbarzellen
  for (int x = i.x-searchRadius; x <= i.x+searchRadius; x++){
    for(int y = i.y-searchRadius; y <= i.y+searchRadius; y++){
      for(int z = i.z-searchRadius; z <= i.z+searchRadius;z++){

        //Zelle, die untersucht wird
        ivec3 lookAtCell2 = ivec3(x,y,z);

        //Pruefe, ob Zelle im Grid liegt
        if(inBounds(lookAtCell2,cellsX,cellsY,cellsZ)) {

          //Berechne offsets der Zelle
          int ci = cellIndex(lookAtCell2,cellsX,cellsY);
          int startOffset = startInd[ci];
          int endOffsetEx = startOffset + count[ci];

          // Gehe ueber offsets und schreibe Partikelindex in den
              Constraint
          for (int it = startOffset; it <= endOffsetEx-1; it++){
            int neighborIndex = sorted[it];
            constraints[currentConstraintIndex].indices[nextWrite] =
                neighborIndex;
}}}}}}
```

Quellcode 27: Die Nachbarschaftssuche im Uniform-Grid

Quellcode 27 zeigt die beschriebene Nachbarschaftssuche. Es lassen sich hier noch weitere Optimierungen vornehmen. Der zu dem Partikel gehörige Constraint lässt sich ebenfalls vorberechnen, wie im vorherigen Abschnitt beschrieben. Auch lässt sich die Anzahl der gefundenen Nachbarn im Gesamten oder pro Zelle begrenzen. Dadurch wird die Suche verkürzt.

Abbildung 11 zeigt ein 2D-Uniform-Grid mit Partikeln. Alle Zellen haben die gleiche Größe. Es ist das Ziel, die Nachbarn von Partikeln zu ermitteln. Ausgehend von der Zelle des aktiven Partikel, werden die benachbarten Zellen nach Partikeln durchsucht. Das blaue Partikel ist hier beispielsweise das

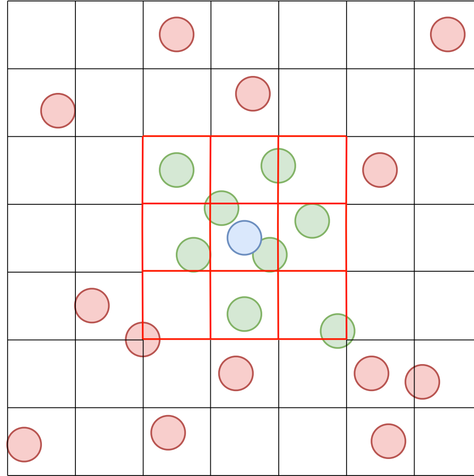


Abbildung 11: Uniform-Grid mit einsortierten Partikeln, inklusive Nachbarschaft des zentralen blauen Partikels. Grüne Partikel sind die Nachbarn, die roten Partikel nicht. Die roten Zellen sind dir Zellen, in denen die Nachbarn gesucht werden.

aktive Partikel, die benachbarten Zellen sind rot umrandet, die resultierende Nachbarschaft bilden die grünen Partikel.

4.8.2 Lösen der Dichte-Constraints

Das Lösen der Dichte-Constraints folgt den in Abschnitt 3.3.2 vorgestellten Formeln nach Macklin und Müller [MM09]. Da ein Dichte-Constraint nur einen Partikel beeinflusst, muss keine zusätzliche Vorkehrung für Synchronisation vorgenommen werden.

Ein Dichte-Constraint hat die Form
$C(p_i, p_0^n, ..., p_n^n) = \frac{\rho_i}{\rho_0} - 1$. p_i ist der beeinflusste Partikel, $p_0^n, ..., p_n^n$ sind die ermittelten Nachbarpartikel, ρ_i die Dichte gegeben durch den SPH-Dichte-Schätzer und ρ_0 die Ruhedichte, die über die Nutzeroberfläche eingestellt werden kann.

Im ersten Schritt zum Lösen der Dichte-Constraints müssen für jeden Partikel p_i die Nachbarschaften neu berechnet werden. Dabei werden in jedem neuen Simulationsschritt die in den Abschnitt 4.8.1 gezeigten Berechnungsschritte durchgeführt. Jeder der einzelnen Schritte wird dabei parallel auf der GPU ausgeführt.

Anschließend werden für alle Constraints C_i der Gradient $\nabla_{p_k} C_i$ und der Constraint-Fehler λ_i ausgerechnet.

Für die Gradienten und die SPH-Dichte werden SSBOs angelegt, in denen die errechneten Werte gespeichert werden. Der Buffer für den Gradienten und die SPH-Dichte sind dabei beide vom Typ `float []`.

Der eigentliche Gradient $\nabla_{p_k} C_i$ wird nur im Nenner der Berechnung des

Constraint-Fehlers benötigt, siehe [MM09], und dass auch nur in seiner Länge aufsummiert und quadriert. Deswegen wird $\sum_k |\nabla_{p_k} C_i|^2$ in dem Gradienten-Buffer gespeichert.

Weil in diesem Schritt bereits über alle Partikel eines Constraints iteriert werden muss, kann in dem selben Schritt schon die Dichte ρ_i über den SPH-Schätzer berechnet werden, wodurch ein kompletter Compute-Shader-Dispatch, bzw. eine zusätzliche Iteration über die Nachbarn gespart wird.

Im nächsten Schritt wird dann der Constraint-Fehler λ_i Vorberechnet. Dafür werden lediglich die im Schritt zuvor Vorberechneten Werte benötigt, inklusive einem Wert für eine Relaxierung und der Ruhedichte, die beide über die Nutzeroberfläche eingestellt werden können.

```
1 void main(){
2  uint i = gl_globalInvocationID.x
3  //Berechnung des Constraints
4  float cx =
       constraintDensity(sph[constraints[i].indices[0]],restDensity);
5
6  //Berechnung des Fehlers
7  float error = -cx/(densityGrad[i]+fluidRelaxation);
8 }
```

Quellcode 28: Berechnung der Constraint-Fehlers nach [MM09]

Quellcode 28 zeigt die Berechnug des Constraint-Fehlers auf der GPU. Die Funktion `constraintDensity()` berechnet den Wert des Constraints. Der Fehler wird abschließend in einem eindeutigen und noch freien Feld abgespeichert. Der beinflusste Partikel steht bei den Dichte-Constraints immer an `indices[0]`.

Für diese Berechnung gibt es allerdings zwei Variationen. Die in Quellcode 28 gezeigte Variante ist nach [MM09]. In [MMCK14] wird der Constraint-Wert `cx` auf 0 gesetzt, wenn dieser negativ sein sollte. Dadurch wirkt dieser nur zum Separieren von Partikeln. In der Arbeit sind beide Varianten vorhanden.

Letztendlich wird die Dichte-Constraint von den in Abschnitt 4.4 gezeigten Lösern gelöst und entprechend die Position oder das delta aktualisiert. Dies geschieht nach der in Abschnitt 3.3.2 vorgestellten Formel nach Macklin und Müller [MM09].

Quellcode 29 zeigt das Lösen der Dichte-Constraints. Der Ansatz nach Monaghan [Mon00] zum Hinzufügen von Druck, um die abstoßende Kraft zu erhöhen, wird auch hier verwendet und in `scorr` berechnet. Die verwendeten Konstanten dafür sind entnommen aus [MM09].

```
case 2:
  float kernelVal =0.0f;
  //gehe ueber alle Nachbarn
  for (int ind=0; ind < constraints[i].indices.length(); ind++){

    //Vektor zwischen Haputpartikel und Nachbar
    vec4 diff = particles[coord].position -
        particles[constraints[i].indices[ind]].position;

    //Berechne korrektur
    float scorr = -0.1f*pow((poly6B(diff,kernelRadius)/
              poly6Bcorrection(0.2*kernelRadius,kernelRadius)),4);

    //addiere Teilsummen auf
    kernelVal += (deltas[constraints[i].indices[0]].pad
                +deltas[constraints[i].indices[ind]].pad+scorr)
                *gradientSpikeyB(diff,kernelRadius);
  }
  kernelVal /=restDensity;
  //aktualisiere Position oder Delta
```

Quellcode 29: Lösen der Dichte-Constraints nach [MM09]

4.9 Viskosität und Wirbelstärke

Zum Abschluss der Fluid-Simulation werden, wie in [MM09], noch die Wirbelstärken korrigiert[8] und XSPH Viskosität nach [SB12] angewandt. Dadurch wird das Verhalten der Flüssigkeit realisitischer.

Wirbelstärke Wie Fedkiw *et al.* [FSJ01] in ihrer Arbeit beschreiben, neigen numerische Ableitungen, welche schließlich im PBD-Verfahren verwendet werden, dazu, Strömungseigenschaften abzuschwächen.

Ziel ist es an der Stelle, diese künstlich wieder hinzuzufügen, indem die bereits vorhandenen Wirbel verstärkt werden.

Zunächst wird für jeden Punkt die Wirbelstärke ω_i berechnet:

$$\omega_i = \nabla \times v_i = -\sum_j \frac{m_j}{\rho_j}(v_i - v_j) \times \nabla W_{ij} \tag{22}$$

Diese wird anschließend durch eine korrigierende Kraft verstärkt:

$$F_i = \epsilon(\frac{\eta}{\|\eta\|} \times \omega_i) \tag{23}$$

[8]https://interactivecomputergraphics.github.io/SPH-Tutorial/slides/06_vorticity.pdf, letzter Aufruf 4.9.2019

Dabei ist η der Ortsvektor der Wirbelstärke:

$$\eta = \sum_j \frac{m_j}{\rho_j} \|\omega_j\| \nabla W_{ij} \tag{24}$$

In der Implementation wird das in zwei Shadern gelöst. Im ersten Shader wird zunächst parallel die Wirbelstärke an den Partikelpositionen nach Formel 22 berechnet und in einem SSBO gespeichert. Als Kernel wird hier der in Abschnitt 3.3.2 vorgestellte *Spikey*-Kernel verwendet.

Bevor ein Summand final berechnet wird, wird über ein Skalarprodukt zwischen dem normalisierten Geschwindigkeitsdifferenzvektor $v_i - v_j$ und Kernel-Gradienten ∇W_{ij} geprüft, ob diese in die selbe Richtung zeigen. Ist dies der Fall, wird die Berechnung des Summanden abgebrochen, da so die Berechnung des Kreuzprodukts fehlschlägt.

In dem zweiten Shader wird anschließend parallel die Kraft aus Formel 23 berechnet. Im Gegensatz zu Formel 22 wird, wie in [FSJ01] beschrieben, nicht der normalisierte Ortsvektor verwendet, da so nur dort Wirbelstärke hinzugefügt wird, wo auch eine vorliegt.

Abschließend wird die Geschwindigkeit der Partikel mit der errechneten Kraft aktualisiert, indem diese mit Zeitschritt verrechnet und auf die vorhandene Geschwindigkeit aufaddiert wird.

XSPH-Viskosität Die zusätzliche Viskosität sorgt, nach [MM09], für eine zusammenhängende Bewegung der Partikel. Die von Schechter und Bridson verwendete XSPH-Viskosität vermischt an dieser Stelle die Geschwindigkeiten der benachbarten Partikel in Abhängigkeit einer Kernel-Funktion, siehe [SB12].

$$v_i^{new} = v_i^* + \epsilon \sum_j \frac{m_j}{\rho_j} (v_j^* - v_i^*) * W_{ij} \tag{25}$$

Formel 25 zeigt die Berechnung der neuen Geschwindigkeit nach XSPH-Viskosität, siehe [SB12]. Dabei sind v_j^* und v_i^* die Geschwindigkeiten des Hauptpartikels p_i und der Nachbarn p_j des nächsten Zeitschritts, m_j die Masse und ρ_j die SPH-Dichte von p_j. W_{ij} ist der verwendete Kernel. Die Konstante ϵ dient zum verteilen der Viskosität und wird hier, nach [MM09], zunächst auf 0.01 gesetzt

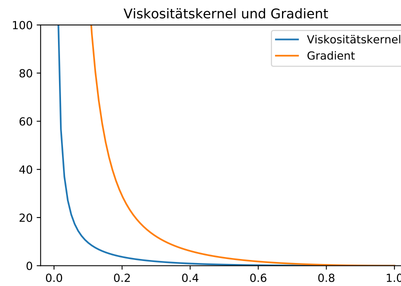


Abbildung 12: Viskositätskernel (blau) und Gradient (orange), nach [MCG03]

```
void main(){
  vec4 velSum = vec4(0.0f);

  //Gehe ueber alle Nachbarn
  for (int i = 1; i < constraints[coord].indices.length(); i++){

    //Berechne Viskositaetsanteil des Nachbarn
    velSum += (particles[constraints[coord].indices[i]].velocity -
        particles[coord].velocity) *
        viscosityB(particles[coord].position -
        particles[constraints[coord].indices[i]].position,
        kernelRadius);
  }

  //Aktualisiere die Geschwindigkeit des Partikels
  particles[coord].velocity += 0.01 * velSum;
}
```

Quellcode 30: Implementation der XSPH-Viskositaet nach [SB12] und [MM09]

Quellcode 30 zeigt die die Implementation von XSPH-Viskosität, die die Geschwindigkeit eines Partikels in Abhängigkeit seiner Nachbarn ändert. Hierbei wird jedoch die aktuelle Geschwindigkeit der Partikel verwendet und der Faktor $\frac{m_j}{\rho_j}$ fällt weg, siehe [MM09].

Der verwendete Kernel W_{ij} ist der Viskositätskernel nach Müller *et al.* [MCG03], siehe Gleichung 26.

$$W_{viscosity}(r,h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & 0 \leq r \leq h \\ 0 & \text{sonst} \end{cases} \tag{26}$$

Abbildung 12 zeigt den Graph des durch Formel 26 beschriebenen Viskositätskernel. Ist der Abstand zwischen zwei Partikeln sehr nahe, ist der Wert des Kernels hoch. Der Wert nimmt aber sehr rasch ab, sodass effektiv nur sehr nahe Partikel einen Einfluss haben.

Die Implementation ist somit in ihren wichtigsten Punkten besprochen. Die daraus entstehenden Simulationen werden im folgenden Kapitel vorgestellt und evaluiert.

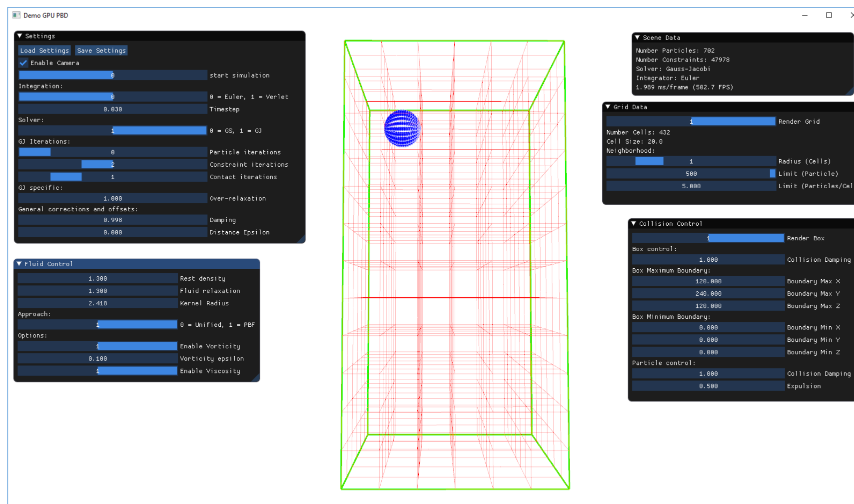


Abbildung 13: Überblick über die Demo.

5 Ergebnisse und Auswertung

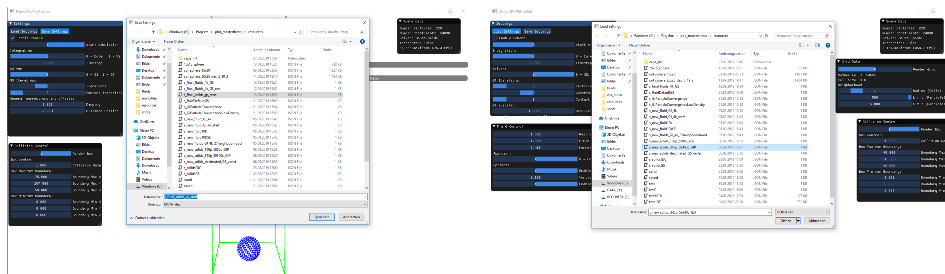
In diesem Kapitel werden die Ergebnisse der Arbeit vorgestellt, siehe Abschnitt 5.1, und anschließend in Abschnitt 5.2 evaluiert. Die Simulationen werden auf einem Laptop mit 16GB Arbeitsspeicher, einer *NVIDIA GeForce GTX 1060* Grafikkarte und einem *Intel Core i7-8750H CPU* Prozessor mit 2.20 GHz ausgeführt. Die Demos und Messungen erfolgten im Release-Build.

5.1 Demos und Ergebnisse

General Abbildung 13 zeigt den Start einer Demo. Die blauen Kugeln sind das in Kapitel 4.2 beschriebene Partikelsystem. Die roten Linien zeigen das gerenderte Uniform-Grid, siehe Kapitel **??**. Die grüne Box stellt den Simulationsbereich dar, welcher über den in Abschnitt 4.3 angesprochenen Box-Constraint nach [Jak03] kontrolliert wird.

Die Nutzeroberfläche wurde mit *Dear ImGui* erstellt und ermöglicht die Parameter der Simulation zur Laufzeit zu ändern. Wie in Abbildung 13 zu sehen ist, ist die Nutzeroberfläche in verschiedene Blöcke aufgeteilt.

Über den Block *settings* werden generelle Einstellungen festgelegt, wie das Starten und Stoppen der Simulation, das verwendete Lösungsverfahren mit Anzahl der Iterationen oder Relaxierungsparameter. Über die Buttons *Save settings* und *Load settings* können aktuell gesetzte Einstellungen gespeichert oder gespeicherte Einstellungen geladen werden. Dabei werden Einstellungen serialisiert und in einer *.json*-Datei gespeichert, bzw. von einer *.json*-Datei deserialisiert und geladen. Über die Bibliothek *tiny file dialogs* wird jeweils ein Fenster geöffnet, in dem das Verzeichnis zum Speichern oder



(a) Speichern von Einstellungen (b) Laden von Einstellungen

Abbildung 14: Einstellungen können gespeichert und geladen werden, was durch eine Serialisierung, bzw. Deserialierung einer *.json*-Datei mithilfe der Bibliotheken *JSON for Modern C++* und *tiny file dialogs* realisiert wird.

die Datei zum Laden ausgewählt werden kann. Abbildung 14(a) zeigt das Speichern und Abbildung 14(b) das Laden von Einstellungen.

Der Block *Fluid Control* verwaltet die Parameter für die Fluid-Simulation. Hier können unter anderem die Ruhedichte λ_0, der Radius der Kernel oder Wirbelstärke und Viskosität eingestellt werden.

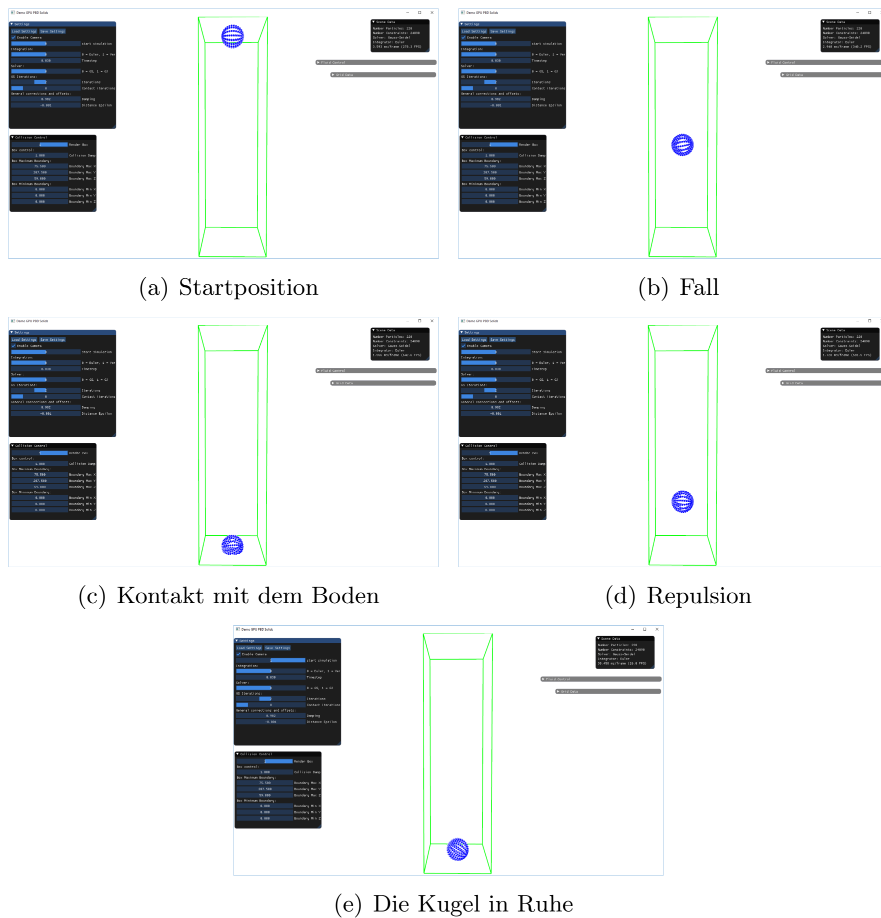
In *Scene Data* sind Daten der Szene zu sehen. Wichtig sind an dieser Stelle vor allem die Anzahl an Partikeln und Constraints und Zeit, die benötigt wird, um einen Simulationsschritt inklusive des Renderings durchzuführen. Zusätzlich ist das aktuell verwendete Integrations- und Lösungsverfahren zu sehen.

Grid Data steuert die Anzeige des Uniform-Grids, sowie die Parameter für die Nachbarschaftssuche in Form von dem Zellenradius der Suche und der Anzahl der Nachbarn, die maximal pro Zelle betrachtet werden sollen. Die Einstellungen sind nur für die Fluid-Simulation relevant.

Der letzte Block *Collision Control* steuert hauptsächlich den Simulationsbereich. Hier kann die Größe und das Anzeigen der Box geändert werden. Durch das Ändern des Simulationsbereichs kann Einfluss auf die Simulation genommen werden, wie im Folgenden gezeigt wird.

Solide Solide folgen der in Abschnitt 4.7.3 gezeigten Implementation. Ein Objekt wird durch eine Menge von Partikeln repräsentiert. Im Basisfall ist jeder Partikel über einen Distanz-Constraint mit jedem anderen verbunden, wobei es keine Dopplungen von Partikelpaaren gibt. Somit sind, wie in Abschnitt 4.7.3 erwähnt, für ein Objekt mit n Partikeln eine Anzahl von $\sum_{i=1}^{n-1} i$ Constraints in jeder Iteration des jeweiligen Lösungsverfahrens zu lösen. Die zum Anfang gesetzten Constraints stellen dabei die Konfiguration dar, die während der Simulation gehalten werden soll.

In der Folgenden Simulation wird eine Kugel mit 220 Partikeln und 24k Constraints verwendet. Als Integrationsverfahren wird Euler verwendet und



(a) Startposition

(b) Fall

(c) Kontakt mit dem Boden

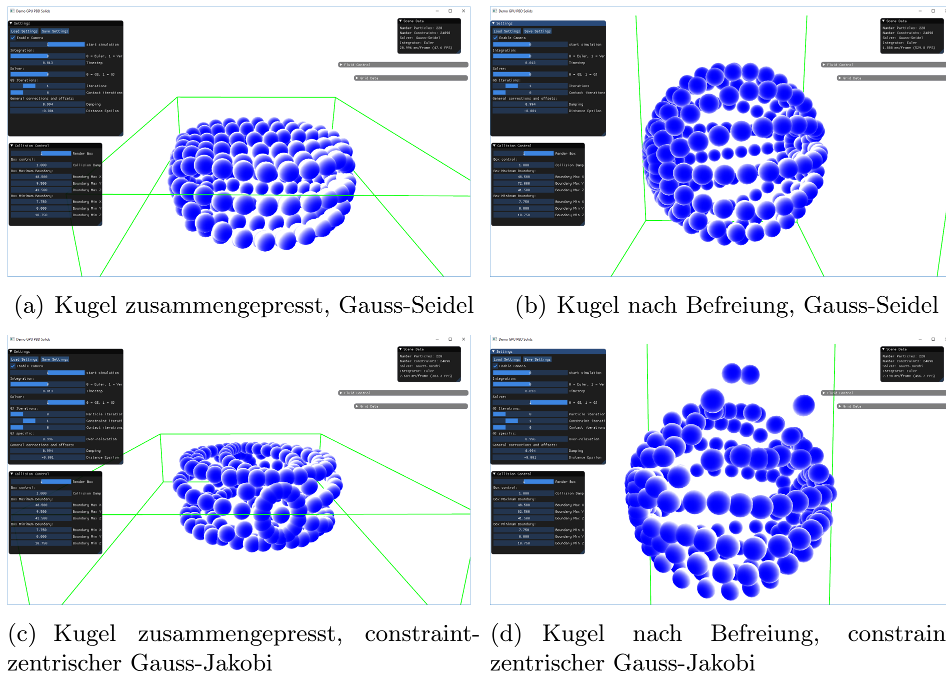
(d) Repulsion

(e) Die Kugel in Ruhe

Abbildung 15: Ablauf einer Simulation für Solide. Eine Kugel mit 220 Partikeln und 24k Constraints. Integrationsverfahren: Euler, Lösungsverfahren: Gauss-Seidel

als Lösungsverfahren Gauss-Seidel mit zwei Iterationen für einen Simulationsschritt. Abbildung 15 zeigt die wichtigsten Schritte der Simulation.

Zum Start wird die Kugel in ihrer Startposition generiert (Abbildung 15(a)). Durch die Gravitation im Integrationsschritt fällt die Kugel (Abbildung 15(b)), bis sie auf den Boden des Simulationsbereichs trifft. An dieser Stelle kollidieren die Partikel mit dem Boden (Abbildung 15(c)), bleiben zunächst durch die Box hängen und erfahren dann durch die Kollisionserkennung eine Änderung ihrer Geschwindigkeit. Da die Startkonfiguration an dieser Stelle verletzt wird und die Partikel in die entgegengesetzte Richtung bewegt werden, sorgen die Distanz-Constraints dafür, dass sich die komplette Kugel mit nach oben bewegt (Abbildung 15(d)). Wäre dies nicht der Fall, würde jedes einzelne Partikel auf den Boden fallen. Dieser Vorgang wiederholt sich so lange, bis die Kugel auf dem Boden liegen bleibt und die Berechnungen konvergieren.



(a) Kugel zusammengepresst, Gauss-Seidel

(b) Kugel nach Befreiung, Gauss-Seidel

(c) Kugel zusammengepresst, constraint-zentrischer Gauss-Jakobi

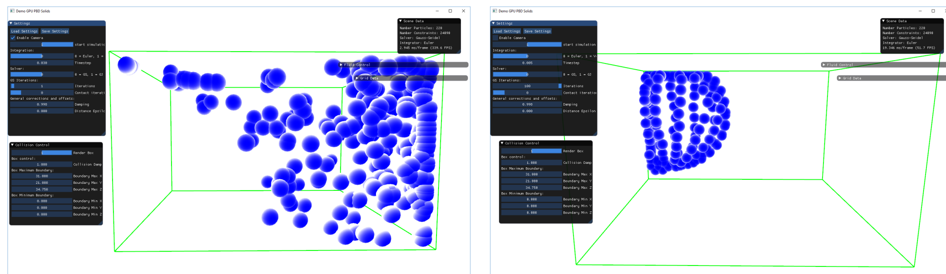
(d) Kugel nach Befreiung, constraint-zentrischer Gauss-Jakobi

Abbildung 16: Vergleich zwischen Gauss-Seidel und constraint-zentrischem Gauss-Jakobi mit jeweils einer Iteration pro Simulationsschritt. Eine Kugel wird zusammengepresst, anschließend wird der Kugel Freiraum gegeben. 220 Partikeln und 24k Constraints.

Durch die Constraints wird die Kugel-Form eingehalten (Abbildung 15(e)).

Die unterschiedlichen Auswirkungen der Lösungsverfahren machen sich vor allem bemerkbar, wenn ein Objekt wieder in die Startkonfiguartion bewegt werden soll.

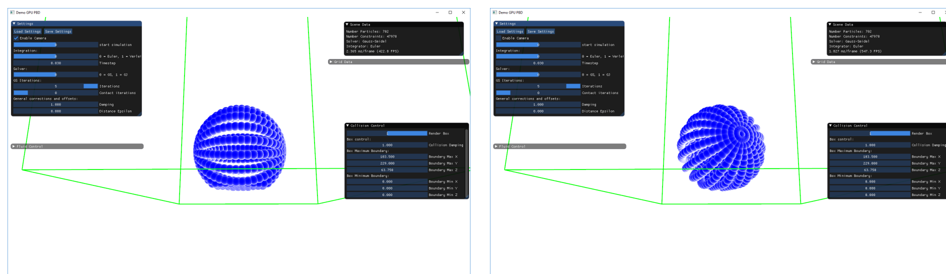
In Abbildung 16 wird eine Kugel zusammengepresst, anschließend wird die Begrenzung aufgehoben. In Abbildungen 16(a) und 16(b) wird dabei Gauss-Seidel mit einer Iteration verwendet, in den Abbildungen 16(c) und 16(d) der constraint-zentrische Gauss-Jakobi mit einer Iteration. In den Abbildungen 16(a) und 16(c) wird die Kugel durch eine Begrenzung des Simulationsbereichs in der Höhe zusammengedrückt. Sobald die Begrenzung aufgehoben ist, soll die Kugel durch Erfüllung der Constraints in die Ausgangskonfiguration überführt werden, siehe Abbildungen 16(b) und 16(d). Durch die Constraints der Partikel am Boden wird die Kugel in die Luft geschossen. Das Gauss-Seidel-Verfahren stellt bei einer Iteration die Ausgangsform wieder her, siehe Abbildung 16(b). Das constraint-zentrische Gauss-Jakobi-Verfahren konvergiert hingegen nicht so schnell bei einer Iteration, siehe Abbildung 16(d). Der anfängliche Schock propagiert durch die gesamte Kugel und einzelne Partikel werden weiter von ihrer Zielposition weg gedrückt. Das Verfahren konvergiert somit noch nicht zu diesem Zeitpunkt.



(a) Gauss-Seidel ohne Constraint-Gruppierung, eine Iteration

(b) Gauss-Seidel ohne Constraint-Gruppierung, 100 Iterationen

Abbildung 17: Gauss-Seidel auf Soliden ohne Constraint-Gruppierung durch Graphfärbung. Das Verfahren konvergiert nicht.



(a) Kugel mit dezimierten Constraints, Kontakt mit dem Boden nach dem Fall

(b) Kugel mit dezimierten Constraints, Ruheposition

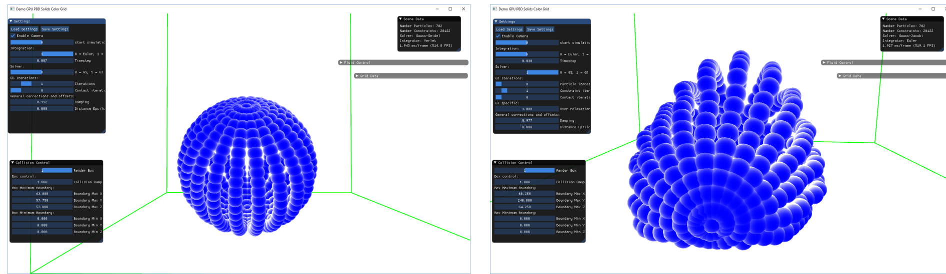
Abbildung 18: Eine Kugel mit 702 Partikeln und 48k Distanz-Constraints bei initialem Kontakt mit dem Boden und in Ruheposition. Gelöst mit 5 Gauss-Seidel-Iterationen.

Es ist anzumerken, dass die Constraints in den bisher gezeigten Ergebnissen durch Graphfärbung gruppiert worden sind. Das Gauss-Seidel-Verfahren benötigt eine Gruppierung der unabhängigen Constraints, damit es konvergiert.

In Abbildung 17 wird eine die Kugel mit Gauss-Seidel ohne Constraint-Gruppierung durch Graphfärbung gezeigt. In Abbildung 17(a) wird eine Iteration des Verfahrens pro Simulationsschritt verwendet, in Abbildung 17(b) 100 Gauss-Seidel-Iterationen pro Simulationsschritt. Das Verfahren konvergiert nicht. Bei 100 Iterationen kann zwar in einem einzelnen Bild eine Kugelform ausgemacht werden, während laufender Simulation ist dies aber nicht der Fall.

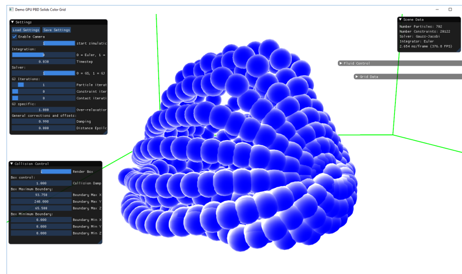
Für die Constraint-Dezimierung ist wichtig zu überprüfen, wie viele Constraints ausreichen, damit das durch das Partikelsystem dargestellte Objekt seine Form behält.

Abbildung 18 zeigt eine Kugel mit 702 Partikeln. Statt der ursprünglichen $\sim$ 246k Distanz-Constraints werden hier durch die Dezimierung $\sim$ 48k



(a) Eine Gauss-Seidel-Iteration nach Kontakt mit dem Boden

(b) Eine constraint-zentrische Gauss-Jakobi-Iteration nach Kontakt mit dem Boden



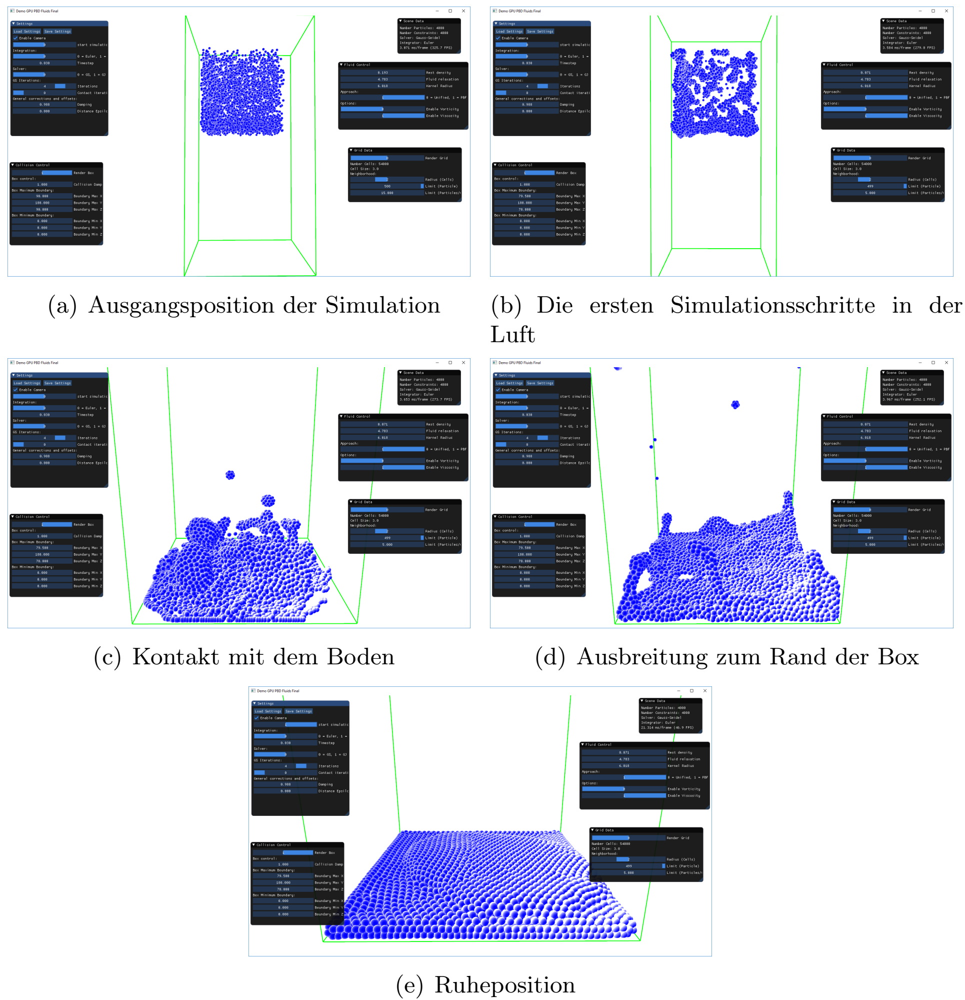
(c) Eine partikel-zentrische Gauss-Jakobi-Iteration nach Kontakt mit dem Boden

Abbildung 19: Eine Kugel mit 702 Partikeln und 20k Distanz-Constraints in Ruheposition nach initialem Kontakt mit dem Boden. Vergleich der Lösungsverfahren.

Distanz-Constraints verwendet. Wie in Abbildung 18(a) an der Unterseite der Kugel zu sehen ist, gibt diese trotz stabilem Gauss-Seidel bei initialem Kontakt mit dem Boden der Box nach. Trotz der nur $\sim \frac{1}{5}$ der Ursprünglichen Distanz-Constraints kann die gewünschte stabile Ruheposition erreicht werden, siehe Abbildung 18(b).

Zum Vergleich der Lösungsverfahren bei dezimierten Constraints wurde die folgende Simulation mit einer Kugel mit 702 Partikeln erstellt. Als Parameter für die Constraint-Dezimierung wurden maxDistance $= 5.0$f, furthest $= 19.0$f und farMax $= 2$ gewählt. Damit verringert sich die Anzahl der Constraints auf ~ 20k.

Abbildung 19 zeigt einen Vergleich zwischen dem Gauss-Seidel (Abbildung 19(a)) und dem Gauss-Jakobi-Verfahren in beiden Varianten (Abbildungen 19(b) und 19(c)), bei jeweils einer Iteration pro Simulationsschritt. Das Gauss-Seidel-Verfahren erhält die ursprüngliche Form der Kugel, trotz der nur $\sim \frac{1}{12}$ der ursprünglichen Constraints. Beiden Varianten des Gauss-Jakobi-Verfahrens gelingt das nicht, selbst bei mehreren Iterationen pro Simulationsschritt.



(a) Ausgangsposition der Simulation

(b) Die ersten Simulationsschritte in der Luft

(c) Kontakt mit dem Boden

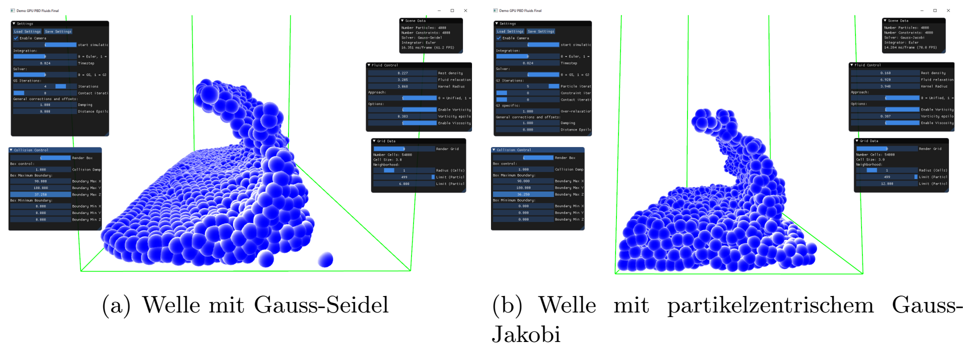
(d) Ausbreitung zum Rand der Box

(e) Ruheposition

Abbildung 20: Fluid-Simulation mit 4000 Partikeln. Gelöst mit 4 Gauss-Seidel-Iterationen pro Simulationsschritt.

Fluide Fluide folgen der in Abschnitt 4.8 vorgestellten Implementation. Jeder Partikel eines Partikelsystems stellt ein Fluid-Partikel dar. Jedes Fluid-Partikel hat einen Dichte-Constraint. Für ein Partikelsystem mit n Fluid-Partikeln sind demnach mit jedem Iterationsschritt n Dichte-Constraints zu lösen.

In der in Abbildung 20 gezeigten Simulation wurden 4000 Fluid-Partikel verwendet. Als Lösungsverfahren wurde Gauss-Seidel mit 4 Iterationen pro Simulationsschritt gewählt. Für jeden Partikel wird die 27-er Nachbarschaft mit 5 Partikeln pro Zelle in Betracht gezogen. Zum Start ist in Abbildung 20(a) das verwendete Partikelsystem zu sehen, die Simulation wurde hier noch nicht eingeschaltet. Sobald dies geschieht, startet die eigentliche Simulation und die Partikel fangen an sich zu gruppieren, siehe



(a) Welle mit Gauss-Seidel

(b) Welle mit partikelzentrischem Gauss-Jakobi

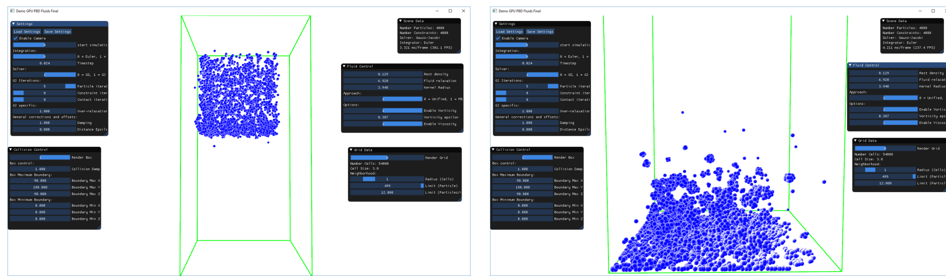
Abbildung 21: Vergleich zwischen Gauss-Seidel und Gauss-Jakobi bei der Fluid-Simulation. Durch ein Zusammenstauchen der Simulationsumgebung wird eine Welle erzeugt.

Abbildung 20(b). Die Klumpenbildung in der Luft ist typisch für die Fluid-Simulation, da der Constraint-Wert eines Dichte-Constraints negativ werden kann [MMCK14], was die Partikel zusammenzieht. In Verbindung mit Gauss-Seidel verstärkt sich dieser Effekt, da die neuen Positionen der Nachbarn direkt in die Berechnungen einbezogen werden. Abbildungen 20(c) und 20(d) interagieren die Partikel mit der Box. Sobald die Partikel den Boden berühren, beginnen sie sich auszubreiten, um die Ruhedichte zu erreichen. Die Kollision mit den Seiten der Box wirft die Partikel entsprechend wieder zurück. Abbildung 20(e) zeigt die erreichte Ruheposition. Durch eine Verkleinerung der Ruhedichte fangen die Partikel an, sich in dieser Lage zu stapeln. Eine Vergrößerung dieser sorgt dafür, dass die Partikel sich weiter zusammenziehen

Abbildung 21 zeigt einen Vergleich der Ergebnisse der Lösungsverfahren bei der Fluid-Simulation. Zunächst wurde der Simulationsbereich in der Horizontalen eingegrenzt und die Ruheposition abgewartet. Dadurch haben sich die Partikel auf engerem Raum aufgestapelt. Durch eine weitere plötzliche Verringerung werden die Partikel in Richtung der Verschiebung gedrückt. Es entsteht an dieser Stelle eine Welle. Wie in Abbildung 21(a) zu erkennen ist, bildet die Welle mit Gauss-Seidel eine viel weichere und zusammenhängendere Form, als bei dem partikelzentrischen Gauss-Jakobi, siehe Abbildung 21(b). Bei diesem bewegen sich die Fluid-Partikel mehr vereinzelt und unabhängiger, da die Partikel im Gesamtschritt aktualisiert werden. Dadurch wirken die Ergebnisse zwar um einiges unruhiger, die Partikel lösen sich aber auch wesentlich leichter voneinander.

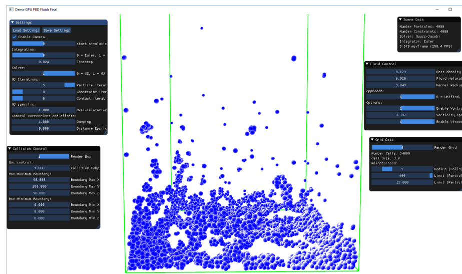
Das Lösen der Partikel ist bereits zu Beginn einer Simulation zu erkennen.

Abbildung 22 zeigt die ersten Simulationsschritte, dieses Mal mit Gauss-Jakobi, statt mit Gauss-Seidel (Abbildung 20). Nach Beginn der Simulation bilden die Fluid-Partikel deutlich weniger zusammenhängende Klumpen in



(a) Erste Simulationsschritte in der Luft, Gauss-Jakobi

(b) Initialer Kontakt mit dem Boden, Gauss-Jakobi



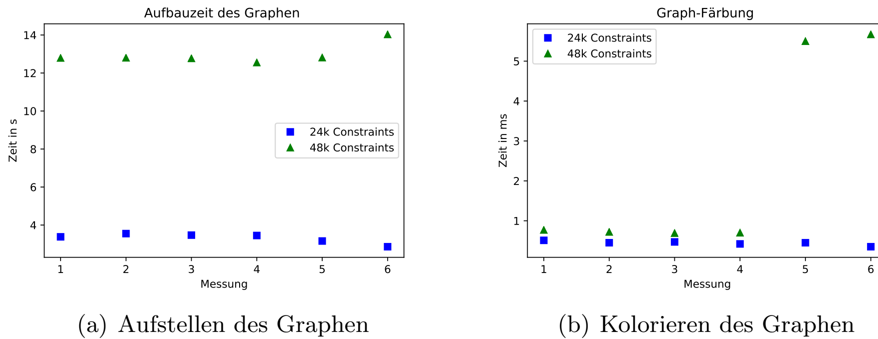
(c) Ausbreitung zum Rand nach Kontakt, Gauss-Jakobi

Abbildung 22: Fluid-Simulation mit 4000 Partikeln. Gelöst mit 4 Gauss-Jakobi-Iterationen.

der Luft (Abbildung 22(a)). Dieses Verhalten ist wieder bei Kontakt mit dem Boden (Abbildung 22(b)) und bei Kontakt mit den Seiten der Box zu erkennen (Abbildung 22(c)). Die Fluid-Partikel lösen sich nicht nur viel leichter voneinander, sondern bilden auch mehr vereinzelte Gruppen, als eine homogene Masse, wie bei dem Gauss-Seidel-Verfahren. Die Partikel bewegen sich also deutlich unabhängiger voneinander. Das Verfahren produziert somit wesentlich instabilere Ergebnisse. Es ist anzumerken, dass es für die visuellen Ergebnis an dieser Stelle egal ist, ob die partikelzentrische oder constraintzentrische Variante des Gauss-Jakobi-Verfahrens gewählt wird. Jeder Partikel hat nur einen Dichte-Constraint. Beide Varianten arbeiten die Partikel, bzw. die Constraints, bei der Fluid-Simulation in gleicher Weise ab.

5.2 Technische Evaluation

In diesem Abschnitt werden die Laufzeiten der implementierten Verfahren und der Simulationen evaluiert. Zunächst wird auf die nötigen Vorberechnungen eingegangen, wie die Graph-Färbung und die Zuordnungen von Constraints zu Partikeln und Farben. Anschließend werden die Simulationszeiten für Solide und Fluide besprochen. Die Zeiten wurden mit Hilfe von



(a) Aufstellen des Graphen

(b) Kolorieren des Graphen

Abbildung 23: Zeitmessungen für den Aufbau und das Kolorieren des Graphen.

timer queries[9] gemessen. Die Messungen für Vorberechnungen und Simulationen wurden jeweils zusammen aufgenommen.

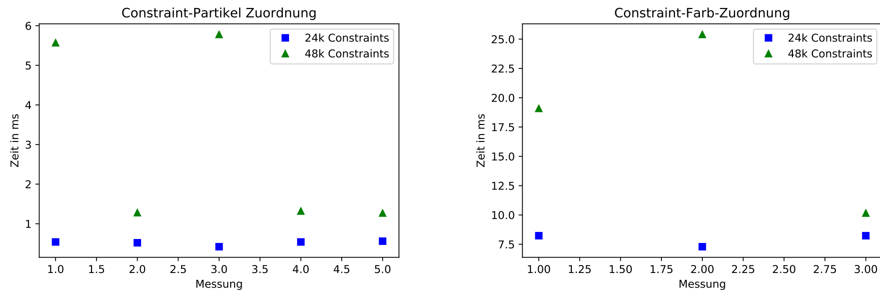
Vorberechnungen Unter den Vorberechnungen ist alles zu verstehen, was nicht innerhalb der Simulationsschleife geschieht. Das trifft auf die Graph-Färbung und das Zuordnen von Constraints zu Partikeln und Farben zu.

Wie bereits in Abschnitt 4.7.1 erwähnt, ist die Graph-Färbung notwendig, damit vor allem das Gauss-Seidel-Verfahren auf der GPU funktioniert. Die in Abbildung 23 gezeigten Messungen sind für den Aufbau des Constraint-Graphen (Abbildung 23(a)) und das anschließende Färben (Abbildung 23(b)) durchgeführt worden. Es wurden jeweils Messungen für 24k Constraints und 48k Constraints gemacht, also dieselben Anzahlen, wie sie auch in der Simulation der Solide vorhanden sind.

Für den Aufbau des Graphen liegen alle Messungen trotz paralleler Implementation auf der CPU im Sekundenbereich. Je mehr Constraints vorliegen, desto länger dauert auch der Aufbau des Graphen. Das ist nicht verwunderlich, da zum Erstellen der Kanten des Graphen jeder Constraint mit jedem folgenden auf eine Überschneidung in den Partikeln getestet werden muss. Es liegt also ein quadratischer Aufwand beim Erstellen des Graphen vor. Aus diesem Grund ist es möglich, einen bereits vorberechneten Graphen mit einer entsprechenden Färbung zu laden und auf die Constraints zu übertragen.

Das Färben des Graphen geschieht jedoch im Millisekundenbereich. Nach Matula und Beck [MB83] hat die hier verwendete sequentielle Graphfärbung einen konstanten Zeitaufwand, der abhängig von der Anzahl der Knoten und Kanten des Graphen ist. Die Ausreißer bei Messungen 5 und 6 bei 48k Constraints lassen sich demnach nur so erklären, dass mehr Kanten erstellt worden sind. Das könnte daran liegen, dass für die 48k Constraints eine dezimierte Constraintmenge verwendet worden ist, die nicht immer gleich

[9] https://www.khronos.org/opengl/wiki/Query_Object



(a) Zuordnung Constraints zu Partikeln (b) Zuordnung Constraints zu Farben

Abbildung 24: Zeitmessungen für die Zuordnungen von Constraints zu Partikeln und Farben.

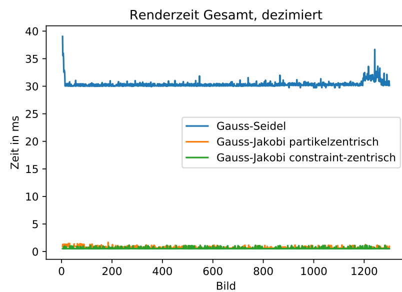
sein muss.

Für die Fluide ist Graph-Färbung zu diesem Zeitpunkt unnötig, da jeder Constraint nur einen Partikel beeinflusst.
Es können also keine Wettlaufbedingungen mit negativem Einfluss wie bei den Soliden auftreten.

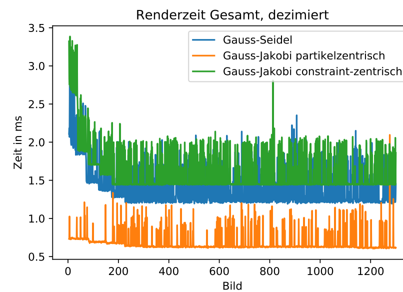
Abbildung 24 zeigt Zeitmessungen für das Zuordnen von Constraints zu Partikeln (Abbildung 24(a)) und zu den Farben aus der Graphfärbung (Abbildung 24(b)). Diese werden mit Hilfe der in Abschnitt 4.8.1 gezeigten Implementation der Offset-Berechnungen auf der GPU durchgeführt. Da sich die Zuordnungen während der Simulation nicht ändert, werden diese vorberechnet. Das Ziel dieser ist es, einen schnelleren Zugriff auf die zu lösenden Constraints zu gewähren. Dadurch soll das Lösen beschleunigt werden. Die starken Unterschiede bei den unterschiedlichen Constraint-Anzahlen, lässt sich erklären. Da die Offset-Berechnung über einen balancierten Baum ausgeführt wird, funktioniert der Algorithmus nur, wenn die Anzahl der Partikel oder Farben eine 2er Potenz ist. Ist dies nicht der Fall, wird die Anzahl künstlich auf die nächste 2er Potenz gesetzt, damit der Algorithmus immer ausgeführt werden kann. Die Größe des Vektors, mit dem gearbeitet wird ist also exponentiell.

Letztendlich sind die ausschlagenden Zeiten aber zu vernachlässigen, da es sich, wie erwähnt, um einen einmal ausgeführten Schritt vor der Simulation handelt. Zudem sind die Werte durch die Implementation auf der GPU im Millisekundenbereich.

Solide Abbildung 25 zeigt die Laufzeiten der gesamten Solid-Simulation. Simuliert wurde hier eine Kugel mit 702 Partikeln. Die Constraints wurden auf 48k dezimiert. Die Lösungsverfahren wurden zum Messen mit einer Iteration pro Simulationsschritt ausgeführt. Ein Simulationsschritt besteht bei der Solid-Simulation aus einem Integrationsschritt, dem

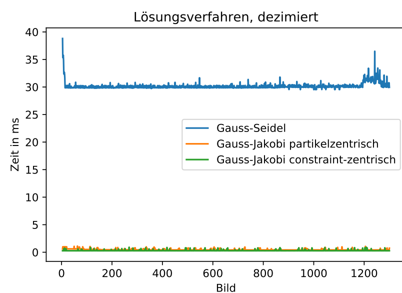


(a) Laufzeiten der Solid-Simulation

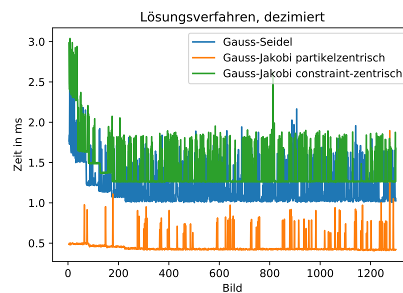


(b) Laufzeiten optimierter Solid-Simulation

Abbildung 25: Zeitmessungen der Laufzeiten der Solid-Simulation.



(a) Laufzeiten der Lösungsverfahren



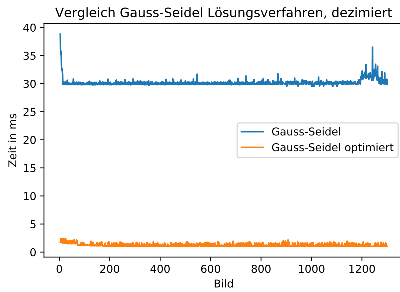
(b) Laufzeiten optimierter Lösungsverfahren

Abbildung 26: Zeitmessungen Lösungsverfahren bei der Solid-Simulation.

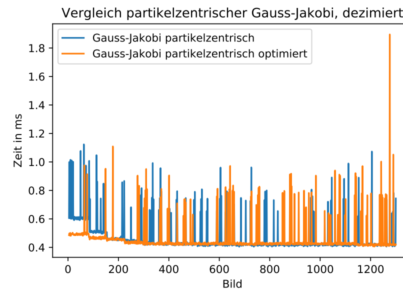
Lösungsverfahren und das anschließende Rendern. Die Integration und das Rendern sind jedoch vernachlässigbar, da das Lösen der Constraints fast den gesamten Anteil der gemessenen Zeit beansprucht. Das eigentliche Rendern macht den zweitgrößten Teil aus. Dieses ist nur von der Anzahl der Partikel abhängig, von denen es in der Solid-Simulation weitaus weniger gibt, als Constraints.

Die Graphen in Abbildung 26 zeigen die Laufzeiten der Lösungsverfahren. Im Vergleich mit den Graphen in Abbildung 25 ist zu sehen, dass die Lösungsverfahren, wie erwähnt, den Großteil der Laufzeit ausmachen.

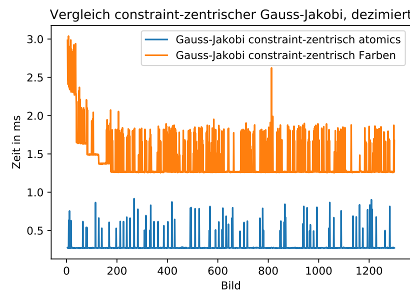
Abbildung 26(a) zeigt die unoptimierten Lösungsverfahren. Es werden an dieser Stelle keine Constraint-Zuweisungen für das Lösen verwendet. Das Lösen nach Farben geschieht bei dem Gauss-Seidel naiv über eine Abfrage, ob ein Constraint aktuell gelöst werden kann, während über die Farben iteriert wird. Die Gesamtschritt-Verfahren sind an dieser Stelle um einiges schneller, da diese keine Farben benötigen. Durch das Zuordnen der Constraints zu den Farben wird das Gauss-Seidel Verfahren sehr effizient. Die Zeiten für eine Iteration liegen nicht mehr bei mehr als 30 ms pro Bild, sondern im Schnitt zwischen 1.0 ms und 2.0 ms, siehe Abbildung 26(b). Das Verfahren



(a) Vergleich Gauss-Seidel



(b) Vergleich Gauss-Jakobi, partikelzentrischzentrisch



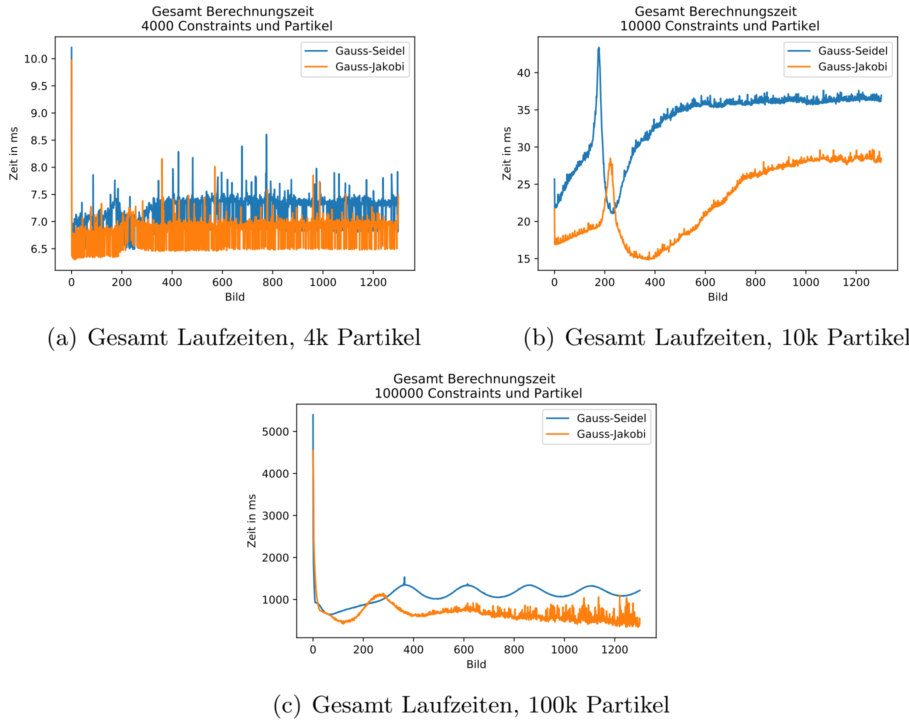
(c) Vergleich Gauss-Jakobi, constraint-zentrisch

Abbildung 27: Direkter Vergleich zwischen den Lösungsverfahren in ihrer normalen und veränderten Variante.

hat somit eine Beschleunigung um das fast 30-fache erfahren.

Die Messungen in Abbildung 27 zeigen die Laufzeiten der einzelnen Lösungsverfahren vor und nach der Optimierung im direkten Vergleich. Gauss-Seidel (Abbildung 27(a)) wurde bereits besprochen. Abbildung 27(b) zeigt das partikelzentrische Gauss-Jakobi-Verfahren. Durch die Zuordnung der Constraints zu den Partikeln kann für jedes Partikel direkt bestimmt werden, welche Constraints für dieses zu lösen sind. Gerade zum Anfang der Simulation, wenn die Kugel in Kontakt mit dem Boden kommt, benötigt das Verfahren nur knapp die Hälfte der ursprünglichen Zeit. Anschließend sind beide Varianten gleich auf.

Die gedachte Optimierung für den constraint-zentrischen Ansatz hat jedoch nicht funktioniert, siehe Abbildung 27(c). Im Normalfall arbeitet dass Verfahren mit atomischen Operationen zum Aufaddieren der Deltas. Ziel war es, mit Ersetzen der atomischen Operationen durch die schon vorhandenen Constraint-Farben das Verfahren von den Operationen zu befreien. Das Resultat ist jedoch sowohl zeitlich, als auch in der Simulation ernüchternd. Unter Verwendung der Farben und der dazugehörigen Zuordnung der Constraints ist der Löser nicht nur deutlich langsamer geworden, sondern auch



(a) Gesamt Laufzeiten, 4k Partikel

(b) Gesamt Laufzeiten, 10k Partikel

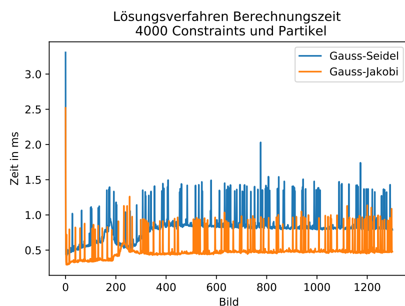
(c) Gesamt Laufzeiten, 100k Partikel

Abbildung 28: Vergleich der Laufzeiten der Fluid-Simulation bei unterschiedlichen Anzahlen an Partikeln.

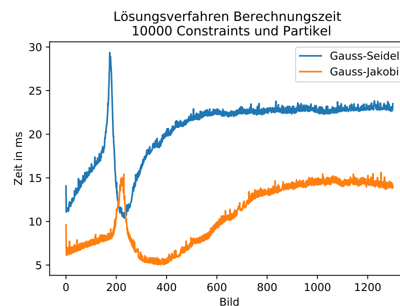
instabiler in der Simulation. Die Verlangsamung lässt sich durch die höhere Anzahl an Shader-Dispatches erklären, da für jede Farbe ein einzelner Dispatch ausgeführt wird. Die unveränderte Variante hingegen hat nur einen für alle Constraints. Die Instabilität in der Simulation kann jedoch durch zusätzliche Unterrelaxierung ausgeglichen werden.

Fluide Für die Fluide wurden Simulationen mit jeweils 4k, 10k und 100k Partikeln evaluiert. Die Simulationen folgen den in Abbildung 20 gezeigten Ablauf.

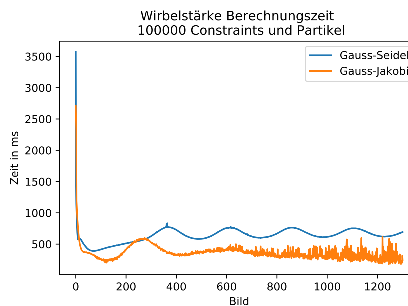
Abbildung 28 zeigt die jeweiligen gesamten Laufzeiten für die verschiedenen Anzahlen an Partikeln. Es wurden jeweils Zeiten für Gauss-Seidel und Gauss-Jakobi gemessen. An den Graphen lässt sich der Simulationsablauf rekonstruieren. Zum Start ist die Laufzeit am höchsten. Die Fluid-Partikel befinden sich dann alle in der Luft auf einem Haufen. Die Nachbarschaften sind für jeden Dichte-Constraint gezwungenermaßen voll besetzt. Beim Fall fangen die Constraints an zu wirken. Die Partikel beginnen, sich zusammen zu ziehen, breiten sich jedoch insgesamt mehr aus. Dadurch gibt es mehr freie Zellen in der jeweiligen Nachbarschaft. Da die Nachbarschaften auf eine maximale Anzahl der Partikel pro Zelle limitiert ist, machen diese freien



(a) Laufzeiten der Lösungsverfahren, 4k Partikel



(b) Laufzeiten der Lösungsverfahren, 10k Partikel

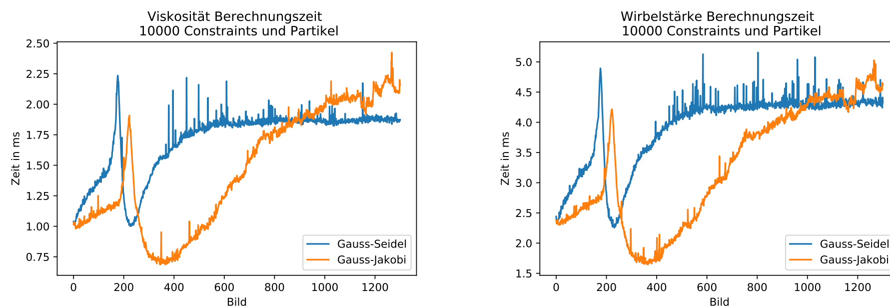


(c) Laufzeiten der Lösungsverfahren, 100k Partikel

Abbildung 29: Vergleich der Laufzeiten der Lösungsverfahren bei unterschiedlichen Anzahlen an Partikeln.

Zellen einiges aus. Die Spitzen, die vor allem in Abbildung 28(b) bei um Bild 200 herum zu erkennen sind, treten bei Kontakt mit dem Boden auf. Die Partikel sind hier wieder dichter zusammen, sodass die Nachbarschaften wieder annähernd voll besetzt sind. Anschließend breiten sich die Partikel zur Seite aus und fließen danach wieder zusammen, bis es zur Konvergenz kommt. In diesem Zustand sind die Constraints erfüllt. Die Partikel sind gemäß der Ruhedichte weit voneinander entfernt. Dadurch sind die Nachbarschaften wieder nicht voll besetzt und bleiben auch in etwa in dem Maße. Durch die Bewegungen der Flüssigkeit verändern sie sich zwar teilweise, aber periodisch. Daran ist auch die Konvergenz in den Abbildungen **??** und 28(c) am Ende zu erkennen.

Zudem lässt sich feststellen, dass das Gauss-Jakobi-Verfahren durchweg schneller als das Gauss-Seidel-Verfahren ist. Auf der anderen Seite konvergiert das Gauss-Seidel-Verfahren nicht nur schneller, sondern ist auch stabiler. Das ist vor allem in Abbildung 28(c) zu erkennen. Während bei dem Gauss-Seidel ab ungefähr Bild 300 die Konvergenz eintritt und die Zeiten anschließend periodisch sind, erreicht Gauss-Jakobi zwar die erste Periode ein wenig früher, jedoch wird danach kein endgültiger Ruhezustand erreicht.



(a) Laufzeiten der Viskositätsberechnung, 10k Partikel

(b) Laufzeiten der Wirbelstärkenberechnung, 10k Partikel

Abbildung 30: Laufzeiten der Nachbarschaftsabhängigen Berechnungen bei 10k Partikeln.

Das Verfahren ist in dem Fall also instabiler. Das liegt auch daran, dass Gauss-Jakobi mehr Nachbarn benötigt, als in dieser Simulation eingestellt worden sind. Gauss-Seidel benötigt weniger Nachbarschaften, da es ein Einzelschrittverfahren ist somit die Nachbarn der Nachbarn implizit miteinbezogen werden. Dadurch gewinnt das Verfahren auch an Stabilität.

Als Beleg für die These, dass die Laufzeiten der Fluid-Simualtion stark von der Anzahl der einbezogenen Nachbarschaft abhängig ist, sind Abbildungen 29 und 30 zu nehmen.

Abbildung 29 zeigt die Laufzeiten der Lösungsverfahren und Abbildung 30 die Laufzeiten der Viskösitätsberechnung (Abbildung 30(a)) und der Wirbelstärkenberechnung (Abbildung 30(b)) für 10k Partikel. Dabei handelt es sich jeweils um Berechnungen, für die einzelnen Nachbarschaften der Partikel miteinbezogen werden müssen. Die Graphen für die 10k Partikel ähneln sich sehr stark in ihrem Verlauf.

Insgesamt lässt sich zeigen, dass beide Verfahren durchaus echtzeit-tauglich sind, wobei das Gauss-Seidel Verfahren deutlich stabilere Ergebnisse liefert. Es ist dabei jedoch auf die Anzahl der Partikel zu achten.

6 Fazit und Ausblick

Ziel dieser Arbeit ist die Implementation von *Position Based Dynamics*, sowie deren Untersuchung und Evaluation anhand von passenden Beispielanwendungen. Dafür sind sowohl solide Körper und Fluide implementiert worden. Alle Berechnungen finden auf der GPU statt.

Das Framework bietet genug Schnittstellen, an denen das System erweitert werden kann. Die Einführung neuer Constraints, und damit neuer Simulationsmethoden, sollte kein Problem darstellen. Es wäre noch wünschenswert, wenn die Szenen lebendiger gestaltet werden könnten, durch z.B. Distanzfelder, mit denen die Simulationen interagieren können. Eine Implementation dieser in den Shadern sollte kein allzu großer Aufwand sein. Ein Beispiel, wie Kollisionen geahndet werden, ist schließlich auch schon durch die Box vorhanden.

Solide Objekte sind erfolgreich durch die Verwendung von Distanz-Constraints simuliert worden. Sowohl das Gauss-Seidel-verfahren, als auch beide Varianten des Gauss-Jakobi-Verfahrens lösen das System zufriedenstellend in Echtzeit. Das ist jedoch abhängig von der Anzahl der Constraints. Dennoch gibt es Unterschiede zwischen den Ergebnissen der Verfahren. Während Gauss-Seidel das stabilere der Verfahren ist, ist Gauss-Jakobi in beiden Varianten, trotz optimiertem Gauss-Seidel, deutlich schneller.

Die Optimierung des Gauss-Jakobi-Verfahrens hat hat nur bei dem partikelzentrischen Ansatz funktioniert. Eine Vorberechnung der zu bearbeitenden Constraints ist bei diesem, sowie bei Gauss-Seidel, der richtige Weg, um das Verfahren zu beschleunigen. Das Verwenden von Farben anstatt von atomischen Operationen bei dem constraint-zentrischen Ansatz, hat die Simulation verlangsamt. Ein Grund dafür ist die vermehrte Anzahl an Shader-Dispatches bei Verwendung der Farben.

Ein Ansatz zur generellen Beschleunigung der Verfahren liegt in der Constraint-Dezimierung. Die Ergebnisse haben gezeigt, dass es nicht nötig ist, alle Partikel untereinander durch Distanz-Constraints zu Verbinden. Es ist an der Stelle jedoch hervorzuheben, dass Gauss-Seidel dann auch wesentlich stabilere Ergebnisse liefert, als die anderen Verfahren. Es könnte sich lohnen zu untersuchen, wie sich die Anzahl der Constraints weiter reduzieren lässt. Mit weniger Distanz-Constraints würde es nicht nur weniger Farben geben, sondern der Aufbau des Graphen würde auch schneller werden. Das ist mit Abstand dasjenige, was am längsten dauert. Durch die Möglichkeit, eine bereits gefundene Kolorierung zu laden, muss diese aber nur einmal vorberechnet werden.

Auch die Fluide sind zufriedenstellend simuliert worden. Effekte wie Tropfenbildung und Kollision mit starren Wänden sind deutlich zu erkennen. Die Verfahren liefern dabei visuell unterschiedliche Ergebnisse. Während die Flüssigkeit bei Gauss-Seidel mehr homogen und zusammenhängend erscheint, ist dies bei Gauss-Jakobi nicht der Fall. Die Partikel bilden da

mehr kleinere Gruppen, wodurch das Fluid lebendiger und unabhängiger erscheint.

Trotz dessen ist auch hier wieder Gauss-Seidel das stabilere Verfahren. Das System konvergiert hier nicht nur früher, sondern auch überhaupt. Gerade bei sehr vielen Partikeln ist das anhand der gezeigten Graphen zu erkennen. Das Gauss-Jakobi-Verfahren schaukelt sich selbst immer wieder hoch.

Laufzeittechnisch ist aber auch hier das Gauss-Jakobi-Verfahren um einiges schneller. Dennoch sind auch wieder für den Gauss-Seidel in Echtzeit laufende Simulationen möglich. Das eigentliche Problem bei der Geschwindigkeit der Fluid-Simulation ist, dass viele Berechnungen für alle Nachbarn eines Partikels ausgeführt werden müssen. Durch eine Begrenzung der Nachbarschaft lässt sich zwar dagegen angehen, dennoch sind die Nachbarn für realistische Fluid-Effekte nötig.

Insgesamt lässt sich somit sagen, dass die gesetzten Ziele erreicht worden sind. Die Simulationen laufen nicht nur in Echtzeit, sondern liefern auch physikalisch plausible Ergebnisse. Auch das Framework bietet genug Schnittstellen zur Erweiterung, sodass in Zukunft mehr Simulationen möglich sein können.

Literatur

[Ble90] Blelloch G. E.: Prefix sums and their applications. 35–60.

[BML*14] Bouaziz S., Martin S., Liu T., Kavan L., Pauly M.: Projective dynamics : Fusing constraint projections for fast simulation. *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2014 33*, 4 (2014), 154:1–154:11.

[BMO*14] Bender J., Matthias M., Otaduy M. A., Teschner M., Macklin M.: A survey on position-based simulation methods in computer graphics. 228–251.

[BTM17] Berndt I., Torchelsen R., Maciel A.: Efficient surgical cutting with position-based dynamics. *IEEE Computer Graphics and Applications 38*, 3 (2017), 24–31.

[FP15a] Fratarcangeli M., Pellacini F.: A gpu-based implementation of position based dynamics for interactive deformable bodies. *Journal of Graphics Tools 17*, 3 (2015), 59–66.

[FP15b] Fratarcangeli M., Pellacini F.: Scalable partitioning for parallel position based dynamics. *Computer Graphics Forum 34*, 2 (2015), 405–413.

[FSJ01] Fedkiw R., Stam J., Jensen H. W.: Visual simulation of smoke. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM, pp. 15–22.

[HVDM*14] Hughes J. F., Van Dam A., McGuire M., Sklar D. F., Foley J. D., Feiner S. K., Akeley K.: *Computer Graphics: Principles and Practice, Third Edition.* Addison-Wesley Professional, 2014.

[Jak03] Jakobsen T.: Advanced character physics. *Game Developer Conference* (2003), 1–16.

[MB83] Matula D. W., Beck L. L.: Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM 30*, 3 (July 1983), 417–427.

[MCG03] Müller M., Charypar D., Gross M.: Particle-based fluid simulation for interactive applications. *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2003), 154–159.

[MHHR06] Müller M., Heidelberger B., Hennix M., Ratcliff J.: Position based dynamics. *Primark.com* (2006).

[MM09] Macklin M., Matthias M.: Position based fluids. 1–5.

[MMC16] Macklin M., Müller M., Chentanez N.: Xpbd: Position-based simulation of compliant constrained dynamics. *Proceedings of the 9th International Conference on Motion in Games - MIG '16* (2016), 49–54.

[MMCK14] Macklin M., Müller M., Chentanez N., Kim T.-Y.: Unified particle physics for real-time applications. *ACM Transactions on Graphics 33*, 4 (2014), 1–12.

[Mon00] Monaghan J. J.: Sph without a tensile instability. *Journal of Computational Physics 159*, 2 (2000), 290–311.

[NM92] Netzer R. H., Miller B. P.: What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS) 1*, 1 (1992), 74–88.

[SB12] Schechter H., Bridson R.: Ghost sph for animating water. *ACM Transactions on Graphics (TOG) 31*, 4 (2012), 61.

[WLJT19] Weiss T., Litteneker A., Jiang C., Terzopoulos D.: Position-based real-time simulation of large crowds. *Computers and Graphics (Pergamon) 78* (2019), 12–22.