UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Simulation von Schnee

Bachelorarbeit

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

vorgelegt von

Lars Hoerttrich

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Bastian Krayer, M. Sc. (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im April 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)

Zusammenfassung Mit Hilfe von Physiksimulationen lassen sich viele Naturphänomene auf dem Rechner nachbilden. Ziel ist, eine physikalische Gegebenheit möglichst korrekt zu berechnen, um daraus Schlüsse für die reale Welt zu ziehen. Anwendungsgebiete sind beispielsweise die Medizin, die Industrie, aber auch Spiele oder Filme.
Schnee ist aufgrund seines physikalischen Aufbaus und seinen Eigenschaften ein sehr komplexes Naturphänomen. Um Schnee zu simulieren, müssen verschiedene Materialeigenschaften beachtet werden.
Die wichtigste Methode, die sich mit der Simulation von Schnee und seiner Dynamik befasst, ist die Material-Point-Method. In ihr werden die auf der Kontinuumsmechanik basierenden Lagrange-Partikel mit einem kartesischen Gitter vereint. Das Gitter ermöglicht die Kommunikation zwischen den eigentlich nicht verbunden Schneepartikeln. Zur Berechnung werden Daten der Partikel auf die Gitterknoten übertragen. Dort werden Berechnungen mit Informationen über benachbarte Partikel durchgeführt. Die Ergebnisse werden danach zurück auf die Partikel übertragen.
Durch GPGPU-Techniken lassen sich physikalische Simulationen auf der Grafikkarte implementieren. Verfahren wie die Material-Point-Method lassen sich durch diese Techniken gut parallelisieren.
Diese Arbeit geht auf die physikalischen Grundlagen der Material-Point-Method ein, und implementiert diese mit Hilfe von Compute-Shadern auf der Grafikkarte. Anschließend werden Performanz und Qualität bewertet.

Abstract Using physics simulations natural phenomena can be replicated with the computer. The aim is to calculate a physical feature as correclty as possible in order to draw conclusions for the real world. Fields of Application are, for example, medicine, industry, but also games or films.
Snow is a very complex natural phenomenon due to its physical structure and properties. To simulate snow, different material properties have to be considered.
The most important method that deals with the simulation of snow and its dynamics is the material point method. It combines the Lagrangian particles based on continuum mechanics with a Cartesian grid. The grid enables communication between the snow particles, which are not actually connected. For calculation of particles data is transferred from these particles to the grid nodes. There, calculations are carried out with information about neighboring particles. The results are then transferred back to the original particles. Using GPGPU techniques, physical simulations can be implemented on the graphics card. Procedures like the material point method can be parallelized well with these techniques.
This paper deals with the physical basics of the material point method and implements them on the graphics card using compute shaders. Then performance and quality are evaluated.

Inhaltsverzeichnis

1 Einleitung

Physikbasierte Simulationen sind ein essentieller Bestandteil der Informatik. Hierbei werden natürliche Phänomene und Abläufe möglichst realistisch am Computer nachgestellt. Dies gelingt durch Berechnungen von physikalischen Gegebenheiten wie etwa Kraft oder Strömung. Anhand der entstandenen Simulation und deren Ergebnis, können dann Abschätzungen und Entscheidungen in der realen Welt getroffen werden.
Simulationen können beispielsweise bei Produktionen eingesetzt werden, bei denen Eigenschaften schon ohne einen produzierten Prototypen überprüft werden können, wie etwa die Aerodynamik von neuen Automobil-Modellen. Auch in anderen Bereichen wird mit Simulationen gearbeitet, wie etwa im Ingenieurswesen oder in der Medizin.

Eine Möglichkeit, so eine Simulation zu implementieren, ist über GPGPU[1]-Techniken, bei denen Berechnungen auf der GPU[2] ausgeführt werden können. Dies hat den Vorteil, dass sich viele Berechnungen gleichzeitig und parallel ausführen lassen, wobei jedoch auf die Synchronisierung und den Speicherzugriff geachtet werden muss.
Insbesondere Partikel-Systeme lassen sich auf diese Weise sehr gut konzipieren, da man die einzelnen Berechnungen für viele Partikel parallel ausführen kann.

Diese Arbeit befasst sich mit der Simulation einer bestimmten Art von Partikeln: dem Schnee.
Schnee besteht aus kleinen Eiskristallen aus gefrorenem Wasser. Er ist ein Naturphänomen, das gerade durch seine Vielseitigkeit und seine vielen Eigenschaften sowohl sehr schön als auch sehr komplex sein kann. Diese Vielseitigkeit kann man in den verschiedenen Formen von Schnee erkennen: beispielsweise im trockenen Pulverschnee, der kaum zusammenhält, oder im frisch gefallenen Neuschnee, der sich sehr gut zu Schneebällen oder Schneemännern formen lässt.
Schnee ist also sehr dynamisch, denn je nach Temperatur, Feuchtigkeit oder Dichte hat er verschiedene Eigenschaften, sodass sich beispielsweise
nasser Schnee anders verhält, als trockener Schnee (siehe Abbildung 13 am Schluss). Die erste Methode mit der diese Dynamik simuliert werden kann, ist die „Material-Point-Method“. Sie basiert auf dem PIC[3]-Verfahren von Sulsky et al. [SZS95].

[1] GPGPU: general purpose computation on graphics processing unit
[2] GPU: graphics processing unit, Grafikprozessor
[3] PIC: particle in cell, Partikel in Zelle

Hierbei werden Berechnung mit Hilfe von Lagrange-Partikeln und einem kartesischen Gitter ausgeführt. Durch das Gitter ist es möglich, dass einzelne Partikel von ihren Nachbarn beeinflusst werden, ohne dass sie tatsächlich verbunden sein müssen, wie es in einem Mesh[4] der Fall wäre. Dadurch können auch Kollisionen mit anderen Objekten und anderen Partikeln untereinander simuliert werden. In Verbindung mit einem Schnee-Materialmodell lassen sich zusätzlich die verschiedenen Schneearten und Szenarien nachbilden.

Zur Simulation von Schnee innerhalb dieser Arbeit wird die Material-Point-Method mit Hilfe von Compute-Shadern von OpenGL[5] auf der GPU implementiert, da sich die Simulation der Schneepartikel gut parallelisieren lässt.
Im 2. Kapitel werden andere Arbeiten präsentiert, die sich mit verschiedenen Methoden zur Simulation von Schnee befassen.
Kapitel 3 beschreibt die physikalischen Grundlagen von Schnee, der Material-Point-Method und dem Materialmodell.
Die technischen Grundlagen der Compute-Shader werden in Kapitel 4 vorgestellt.
Danach wird in Kapitel 5 die Implementation auf der GPU erläutert.
In Kapitel 6 werden die Ergebnisse der Simulation hinsichtlich Performanz und Qualität bewertet, während in Kapitel 7 ein Ausblick gegeben wird, welcher zukünftiges Potenzial der Methode aufzeigt.

[4] Mesh: ein verbundenes Netz von Punkten

[5] OpenGL: Open Graphics Library, eine Programmierschnittstelle (API) zur Entwicklung von 3D Anwendungen

2 Verwandte Arbeiten

In diesem Abschnitt werden verschiedene Arbeiten vorgestellt. Teilweise handelt es sich um Grundlagen, auf die die Material-Point-Method aufbaut, teilweise um andere Verfahren, durch die Schnee simuliert werden kann.

2.1 Geometrie

Eine Möglichkeit Schnee zu simulieren, ist ihn mit Hilfe von Geometrie zu modellieren.
Feldman et al. [FO02] nutzen Wind und daraus entstehende Strömungsfelder, um die Menge an Schnee an den Punkten einer Szene zu akkumulieren. Dies geht auf die ursprüngliche Methode von Fearing [Fea00] zurück, in der Schneepartikel in die Luft geschossen werden und auf den Oberflächen der Szene aufsummiert wird, wie viele Schneepartikel an den jeweiligen Stellen landen.
Eine andere Methode von Nishita et al. [NIDN97] nutzt „Metaballs “, also Schneebälle, mit einer Dichte, um Schnee an Oberflächen zu modellieren und zu rendern.
Zhu und Yang [ZY10] erweiterten diese Verfahren, um bessere Interaktion mit externen Objekten und Oberflächen zu ermöglichen.
In vielen Filmen oder Videospielen wie beispielsweise Batman: Arkham Origins[6] werden Height-Maps[7] eingesetzt, um Schnee darzustellen [SOH99]. Es handelt sich um eine simple, aber sehr effiziente Art Schnee darzustellen.

2.2 Granulares Material

Granulare Materialien, wie etwa Sand, sind Materialien, die aus vielen kleinen, festen Partikeln oder Körnern bestehen. Schnee wird in der Computergraphik traditionell als ein solches granulares Material angesehen.
Granulare Materialien können auf zwei Arten simuliert werden. Arbeiten wie von Miller et al. [MP89], Luciani et al. [LHM95] oder Milenkovic [Mil96] nutzen Partikelsysteme, in denen jeder Partikel einzeln behandelt wird. Diese Methoden wurden dann später noch weiterentwickelt [BYM05]. Der Nachteil dieser Methoden ist eine vergleichsweise geringe Effizienz, was sich bei einer höheren Anzahl Partikeln negativ auswirkt.
Im Gegensatz zu diesen Methoden besteht die wesentlich effizientere Möglichkeit das Material als eine Menge von Partikeln anzunähern, wie es von Zhu und Bredson [ZB05], Narain et al. [NGL10] oder

[6] `https://www.gdcvault.com/play/1020177/Deformable-Snow-Rendering-in-Batman`, Einsicht am 04.04.2019

[7] Height-Maps: Höhenfeld, eine Textur mit Höheninformationen

Alduán und Otaduy [AO11] vorgestellt wurde. Gerade die Arbeit von Zhu und Bredson [ZB05] stellt eine FLIP[8]-Methode vor, die Sand als inkompressibles Fluid simuliert. Dies wird in anderen Arbeiten wie der von McAdams et al. [MSW+09] aufgegriffen, um beispielsweise Haare zu simulieren.

2.3 Materialmodell

Eine wichtige Komponente der Simulation von Schnee durch die Material-Point-Method ist das Materialmodell, da Schnee sehr dynamisch ist. Er verhält sich aufgrund seiner Struktur teilweise ähnlich zu Flüssigkeiten. Manchmal kann Schnee aber auch festem Material ähneln.
Das Materialmodell muss also sowohl plastische, irreversible als auch elastische und damit reversible Verformungen zulassen können. Eine der ersten Arbeiten zu diesem Thema stammt aus dem Jahr 1988 [TF88]. Hier kann deformierbare Plastizität bis hin zum Bruch modelliert werden, während Arbeiten wie von Stomakhin et al. [SHST12] versuchen, die Robustheit für elastische Simulationen zu verbessern. 2011 haben Levin et al. [LLJ+11] eine Möglichkeit gefunden, Elastizität mit Hilfe eines kartesischen Gitters, ähnlich dem der Material-Point-Method, zu berechnen.

2.4 Dynamik

Die wichtigste Arbeit, die sich mit der Dynamik von Schnee befasst, ist die von Stomakhin et al. [SSC+13] aus dem Jahr 2013. Hier wird die Material-Point-Method mit Hilfe eines geeigneten Materialmodells genutzt, um dynamischen Schnee zu simulieren.
Die Arbeit ist daher die Hauptreferenz für diese Bachelorarbeit, die versucht, dieses Verfahren durch Compute-Shader zu implementieren.
Zudem lassen sich Implementationen finden, wie etwa in 2D[9], in 3D auf der Grafikkarte über CUDA[10], eine GPGPU-Technik von Nvidea oder auch über Compute-Shader[11] in Verbindung mit der Arbeit von Meyer 2015 [Mey15].

[8] FLIP: fluid implicit particle, eine Weiterentwicklung der PIC-Methode
[9] 2D OpenGL Version : `https://github.com/Azmisov/snow` , Einsicht am 08.04.2019
[10] Link: `https://github.com/wyegelwel/snow` , Einsicht am 08.04.2019
[11] Link: `https://github.com/MeyerFabian/snow` , Einsicht am 08.04.2019

3 Physikalische Grundlagen

Dieses Kapitel behandelt die physikalischen Grundlagen, die zum Verstehen und Implementieren der Material-Point-Method essentiell sind.
Da die Methode auf der Kontinuumsmechanik basiert, werden die Grundlagen dieser in Kapitel 3.1 vorgestellt.
Des Weiteren wird in Kapitel 3.2 Schnee von der physikalischen Seite beschrieben.
Die Material-Point-Method steht in Verbindung mit einem Materialmodell, welches in Abschnitt 3.3 gezeigt wird.
Kapitel 3.4 beschreibt abschließend die physikalischen Berechnungen und den Ablauf der Material-Point-Method.

3.1 Kontinuumsmechanik

Die Kontinuumsmechanik ist ein Teilgebiet der Mechanik. Sie beschreibt das Verhalten von deformierbaren Körpern, das durch Belastung beeinflusst wird.
In diesem Teil der Physik wird Materie als ein Kontinuum angenähert, welches Eigenschaften wie Dichte, Temperatur oder Geschwindigkeit besitzt.
Sämtliche Formeln und Inhalte dieses Kapitels basieren auf McGinty [McG].

Der Spannungstensor ist ein wichtiger Bestandteil der Kontinuumsmechanik. Spannung ist durch Kraft pro Fläche definiert. Der Spannungstensor ist dabei aus zwei Bausteinen aufgebaut: zum einen der Normalspannung, der Kraft σ, die parallel zur Normalen der Fläche wirkt und zum anderen aus der Scherspannung, der Kraft τ, die parallel zur Fläche und damit orthogonal zur Normalen wirkt.

$$\sigma = \frac{F_{\text{normal}}}{A} \qquad \text{und} \qquad \tau = \frac{F_{\text{parallel}}}{A} \tag{1}$$

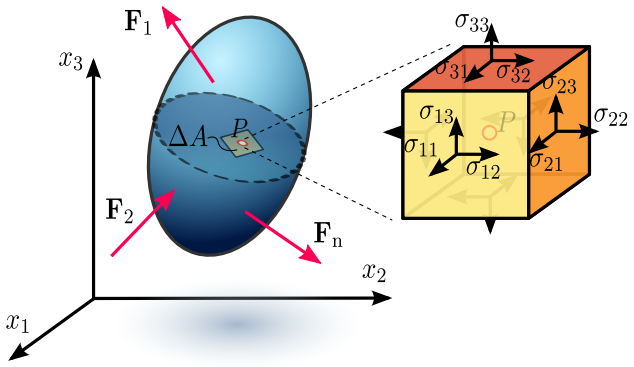


Abbildung 1: Spannungstensor

Quelle: `https://commons.wikimedia.org/wiki/File:Stress_in_a_continuum.svg`, Einsicht am 15.04.2019

Im dreidimensionalen Raum gibt es dementsprechend eine Normalspannung und zwei Scherspannungen (siehe Abbildung 1). Der Spannungstensor kann dann folgendermaßen geschrieben werden:

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_{22} & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_{33} \end{bmatrix} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix} = \begin{bmatrix} \sigma_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_{yy} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_{zz} \end{bmatrix} \tag{2}$$

Hierbei ist zu beachten, dass es auch symmetrische Spannungstensoren gibt. τ_{xy} rotiert den Körper gegen den Uhrzeigersinn. τ_{yx} hingegen rotiert ihn im Uhrzeigersinn. Dadurch ergibt sich $\tau_{xy} = \tau_{yx}$ und dementsprechend auch $\tau_{yz} = \tau_{zy}$ und $\tau_{xz} = \tau_{zx}$. Dann ist σ wie folgt zu schreiben:

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{12} & \sigma_{22} & \sigma_{23} \\ \sigma_{13} & \sigma_{23} & \sigma_{33} \end{bmatrix} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{bmatrix} = \begin{bmatrix} \sigma_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{xy} & \sigma_{yy} & \tau_{yz} \\ \tau_{xz} & \tau_{yz} & \sigma_{zz} \end{bmatrix} \tag{3}$$

Die Dehnung gibt eine relative Längenänderung an und kann in verschiedenen Formen vorkommen. Eine positive Längenänderung wird als Streckung bezeichnet, während eine negative Dehnung Stauchung genannt wird.
Sie kommt im Verzerrungstensor zum Vorschein. Auch dieser besteht aus 2 verschiedenen Teilen, der normalen Dehnung und der Scherdehnung. Diese werden wie folgt berechnet:

$$\epsilon = \frac{\Delta L}{L_0} \qquad \text{und} \qquad \gamma = \frac{\Delta x + \Delta y}{T} \tag{4}$$

Hierbei ist ϵ die Normaldehnung, L_0 ist die initiale Strecke und ΔL ist die Differenz zwischen der neuen Strecke und der initialen Strecke.

Die Scherdehnung γ setzt sich aus dem Verhältnis von $\Delta x + \Delta y$ und T zusammen, wie man in Abbildung 2 sieht.
Im dreidimensionalen Raum setzt sich der Verzerrungstensor analog zum

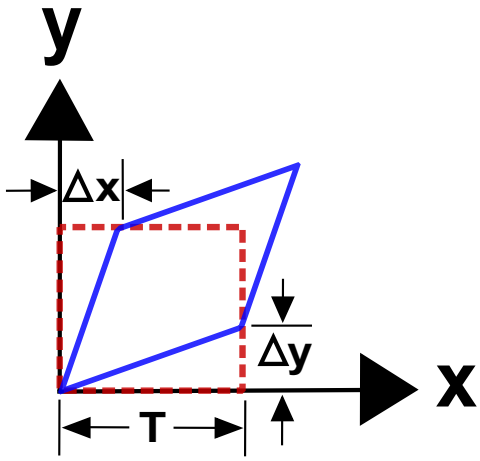


Abbildung 2: Scherung

Quelle: `http://www.continuummechanics.org/images/strain/pure_shear_def.svg`,Einsicht am 15.05.2019

Spannungstensor aus einer Normaldehnung und zwei Scherdehnungen zusammen. Es gilt zu beachten, dass auch dieser Tensor symmetrisch ist, und dass $\boldsymbol{\epsilon_{ij}}$ äquivalent zu $\boldsymbol{\gamma_{ij}}/2$ ist. Die Normaldehnung entspricht also der Hälfte der berechneten Scherdehnung. Dadurch lässt sich folgende Berechnung für einen Verzerrungstensor aufstellen:

$$\begin{bmatrix} \epsilon_{11} & \epsilon_{12} & \epsilon_{13} \\ \epsilon_{12} & \epsilon_{22} & \epsilon_{23} \\ \epsilon_{13} & \epsilon_{23} & \epsilon_{33} \end{bmatrix} = \begin{bmatrix} \epsilon_{xx} & \epsilon_{xy} & \epsilon_{xz} \\ \epsilon_{xy} & \epsilon_{yy} & \epsilon_{yz} \\ \epsilon_{xz} & \epsilon_{yz} & \epsilon_{zz} \end{bmatrix} = \begin{bmatrix} \epsilon_{xx} & \gamma_{xy}/2 & \gamma_{xz}/2 \\ \gamma_{xy}/2 & \epsilon_{yy} & \gamma_{yz}/2 \\ \gamma_{xz}/2 & \gamma_{yz}/2 & \epsilon_{zz} \end{bmatrix} \tag{5}$$

Der Elastizitätsmodul oder Youngscher Modul E ist die Proportionalitätskonstante im Hookeschen Gesetz. Dieses beschreibt die elastische Verformung von Körpern, wenn diese linear zur Belastung auftritt. Der Youngsche Modul beschreibt hierbei das Verhälnis von Spannung zu Dehnung und ist folglich definiert als:

$$E = \frac{\sigma}{\epsilon} \qquad \text{nach Einsetzen:} \qquad E = \frac{F}{A} * \frac{L_0}{\Delta L} \tag{6}$$

Der Youngsche Modul ist abhängig vom Material. Er gibt an, ab wann ein Material anfängt sich nach Belastung zu deformieren, anstatt nur gestreckt zu werden.

Die Poissonzahl oder auch Querkontraktionszahl $\boldsymbol{\nu}$ beschreibt die Möglichkeit eines Materials, gedehnt zu werden. Sie kann über das Hookesche Gesetz hergeleitet werden und definiert sich dann wie folgt:

$$\epsilon_{xx} = \frac{1}{E}\left[\sigma_{xx} - \nu\,\sigma_{yy} - \nu\,\sigma_{zz}\right] \qquad = \qquad \epsilon_{xx} = \frac{1}{E}\left[\sigma_{xx} - \nu\left(\sigma_{yy} + \sigma_{zz}\right)\right] \quad (7)$$

Wir sehen, dass die Dehnung des Objektes abhängig ist von der Spannung und den Dehnungen in andere Richtungen als der Hauptrichtung. Bei einer Streckung entlang der x-Richtung haben weitere Dehnungen in y- oder z-Richtung Einflüsse auf die Streckung. Da diese Einflüsse proportional zur Spannung in x-Richtung sind, kann man die Poissonzahl als Konstante einführen.
So wie jedes Material einen Youngschen Modul hat, so hat auch jedes Material eine Poissonzahl. Beispielsweise ist die Poissonzahl von manchen Schaumstoffen ungefähr 0.1. Diese Schaumstoffe können also sehr einfach gestaucht bzw. komprimiert werden. Bei Gummi ist die Poissonzahl mit annähernd 0.5 nah am Maximum, was zur Folge hat, dass sich Gummi nicht stauchen und komprimieren lässt.

Die Erhaltungssätze von Masse und Impuls sagen aus, dass Masse und Impuls in einem abgeschlossenem System nicht zusätzlich kreiert oder verloren gehen können. Das heißt, dass verlorene Masse oder Impulse, die das System verlassen, an einer anderen Stelle wieder ins System eintreten müssen.

$$\frac{D\rho}{Dt} = 0, \qquad \rho\frac{D\upsilon}{Dt} = \nabla * \sigma + \rho g \quad (8)$$

Hierbei ist ρ die Dichte, t ist die Zeit, υ ist die Geschwindigkeit, σ ist, wie oben beschrieben, die Spannung und g ist die Schwerkraft.

Der Deformationsgradient F beschreibt die Verformung eines Körpers im Raum. Dabei wird die initiale Position des Körpers $\boldsymbol{X}$ mittels einer Funktion ϕ zum deformierten Zustand $\boldsymbol{x}$ gebracht, also $x = \phi(X)$ oder $x = \phi(X, t)$ in Abhängigkeit von Zeit.
Die Deformationsfunktion ändert sich anhand von den Erhaltungssätzen von Masse und Impuls, die oben beschrieben wurden, sowie den elastoplastischen Eigenschaften des Materials [SSC+13].
Der Deformationsgradient F kann dann definiert werden als:

$$F_{ij} \quad = \quad x_{i,j} \quad = \quad \frac{\partial x_i}{\partial X_j} \quad = \quad \begin{bmatrix} \frac{\partial x_1}{\partial X_1} & \frac{\partial x_1}{\partial X_2} & \frac{\partial x_1}{\partial X_3} \\ \frac{\partial x_2}{\partial X_1} & \frac{\partial x_2}{\partial X_2} & \frac{\partial x_2}{\partial X_3} \\ \frac{\partial x_3}{\partial X_1} & \frac{\partial x_3}{\partial X_2} & \frac{\partial x_3}{\partial X_3} \end{bmatrix} \quad (9)$$

für alle $x_{i,j} = \phi(X_{i,j})$.

Um die Anschaulichkeit zu erhalten, werden die drei Arten von Deformation in (10) nur im zweidimensionalen Raum gezeigt.

$$\mathbf{F_S} = \begin{bmatrix} 3.5 & 0.0 \\ 0.0 & 2.0 \end{bmatrix}, \mathbf{F_{SR}} = \begin{bmatrix} 2 & 0.0 \\ 0.5 & 2.0 \end{bmatrix}, \mathbf{F_{Sch}} = \begin{bmatrix} 2.0 & 0.5 \\ 0.5 & 2.0 \end{bmatrix} \tag{10}$$

F_S ist eine Matrix, die einen gegebenen Punkt skaliert. Die Faktoren sind abzulesen: in x-Richtung wird um den Faktor 3.5 skaliert und in y-Richtung um den Faktor 2.
F_{Sch} ist eine Scherungsmatrix. Die Existenz von Werten auf der Diagonalen, die nicht 0 sind, zeigt eine Scherung an. Da die Matrix symmetrisch ist, findet allerdings keine Rotation statt.
Im Gegensatz dazu steht die Matrix F_{SR}. Multipliziert man einen Punkt eines Objekts mit ihr, so wird das Objekt geschert und rotiert. Das erkennt man an der fehlenden Symmetrie und daran, dass ein Wert existiert, der nicht 0 ist und nicht auf der Hauptdiagonalen liegt.
Zusätzlich zu diesen Deformationen gibt es noch Berechnungen, die das Objekt nicht deformieren: die Translation und die Rotation. Diese verändern zwar die Ausrichtung oder die Position eines Objekts, jedoch nicht die Form. Im Gegensatz zu der Verschiebung, die keinen Einfluss auf den Deformationsgradienten hat, steht die Rotation. Obwohl beide Berechnungen die Form des Objekts nicht verändern, lässt sich die Rotation um den Winkel θ über folgenden Deformationsgradienten ausdrücken:

$$\mathbf{F} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{11}$$

Hierbei handelt es sich um ein großes Problem, da dieser Deformationsgradient nicht der Identitätsmatrix entspricht, aber die Form des Objekts nicht ändert. Eine mögliche Fehlerquelle entsteht, da einfache Rotationen als Deformationen gesehen werden können.
Wenn man einen Punkt skalieren und rotieren will, dann ist es möglich dies mit zwei getrennten Matrizen zu berechnen. Zuerst wird der Punkt skaliert und dann rotiert. Die Skalierung hat einen Einfluss auf den Deformationsgradienten, die Rotation aber nicht. Man kann die Gesamtoperation als Matrixprodukt $F = M_{Rotation} * M_{Skalierung}$ ansehen. Ein Punkt, der mit F multipliziert wird, kann also stattdessen auch mit $M_{Skalierung}$ und danach mit $M_{Rotation}$ multipliziert werden. Dies resultiert darin, dass die Deformation nur von der ersten Matrix abhängig ist und nicht von der Rotationsmatrix, da beide getrennt voneinander sind. Die Lösung für das soeben genannte Problem ist die Polarzerlegung.

Die Polarzerlegung zerlegt den Deformationsgradienten F in zwei Teile, eine Rotationsmatrix R und eine symmetrische Matrix S, die die Deformation beschreibt.
Dabei wird die Eigenschaft genutzt, dass die transponierte Matrix F^T multipliziert mit der Matrix F symmetrisch ist. Der Deformationsgradient ist als Matrixprodukt definiert: $F = R * S$. Man kann folgende Formel herleiten:

$$\mathbf{F}^T \cdot \mathbf{F} \quad = \quad (\mathbf{R} \cdot \mathbf{S})^T \cdot (\mathbf{R} \cdot \mathbf{S}) \quad = \quad \mathbf{S}^T \cdot \mathbf{R}^T \cdot \mathbf{R} \cdot \mathbf{S} \tag{12}$$

Da die inverse Matrix einer Rotationsmatrix gleichzeitig ihre transponierte Matrix ist, lässt sich $R^T \cdot R$ zur Identitätsmatrix kürzen. Damit bleibt:

$$\mathbf{F}^T \cdot \mathbf{F} = (\mathbf{R} \cdot \mathbf{S})^T \cdot (\mathbf{R} \cdot \mathbf{S}) = \mathbf{S}^T \cdot \mathbf{R}^T \cdot \mathbf{R} \cdot \mathbf{S} = \mathbf{S}^T \cdot \mathbf{I} \cdot \mathbf{S} = \mathbf{S}^\mathbf{T} \cdot \mathbf{S} \tag{13}$$

Da die Matrix S symmetrisch ist, ist $S = S^T$. Es gilt jetzt die Matrix S aus S^2 zu bestimmen, und R mit der inversen Matrix von S zu multiplizieren, um R zu erhalten.

$$\mathbf{F} \cdot \mathbf{S}^{-1} \quad = \quad \mathbf{R} \cdot \mathbf{S} \cdot \mathbf{S}^{-1} \quad = \quad \mathbf{R} \tag{14}$$

Die Singulärwertzerlegung ist eine weitere Möglichkeit die Polarzerlegung zu berechnen. Sie ist definiert als die Zerlegung von F:

$$\mathbf{F} = \mathbf{U} \cdot \mathbf{\Sigma} \cdot \mathbf{V}^\mathbf{T} \tag{15}$$

wobei U und V orthogonale Matrizen sind. Ihre transponierten Matrizen sind also auch ihre inversen Matrizen. Σ ist eine Diagonalmatrix, bei der alle Werte, die nicht auf der Hauptdiagonalen liegen, 0 sind. Durch die Multiplikation von F mit F^T erhält man:

$$\mathbf{F}^\mathbf{T} \cdot \mathbf{F} = \mathbf{V}^\mathbf{T} \cdot \mathbf{\Sigma} \cdot \mathbf{U} \cdot \mathbf{U}^\mathbf{T} \cdot \mathbf{\Sigma}^\mathbf{T} \cdot \mathbf{V} = \mathbf{V} \cdot \mathbf{\Sigma} \cdot \mathbf{\Sigma}^\mathbf{T} \cdot \mathbf{V}^\mathbf{T} \tag{16}$$

Diagonalmatrizen multipliziert mit ihrer inversen Matrix resultieren in einer Matrix, in der die Hauptdiagonale einzeln quadriert wurde. Diese Matrix kann als $\mathbf{\Sigma^2}$ geschrieben werden, dann ergibt sich:

$$\mathbf{F} \cdot \mathbf{F}^\mathbf{T} = \mathbf{V}\mathbf{\Sigma}\mathbf{\Sigma}^\mathbf{T}\mathbf{V}^\mathbf{T} = \mathbf{V}\mathbf{\Sigma}^\mathbf{2}\mathbf{V}^\mathbf{T} \tag{17}$$

Die Diagonalmatrix Σ^2 enthält die Eigenwerte von $F \cdot F^T$ und V enthält ihre Eigenvektoren. Die fehlende Matrix U kann berechnet werden in dem wir die Ausgangsgleichung 15 nach U umstellen:

$$\mathbf{U} = \mathbf{F} \cdot \mathbf{V} \cdot \mathbf{\Sigma}^\mathbf{T} \tag{18}$$

Abschließend kann die Polarzerlegung durch die Singulärwertzerlegung ersetzt werden[12]

$$\mathbf{F} = \mathbf{R} \cdot \mathbf{S} \qquad \text{wobei:} \qquad \mathbf{R} = \mathbf{U} \cdot \mathbf{V}^{\mathbf{T}}, \mathbf{S} = \mathbf{V} \cdot \mathbf{\Sigma} \cdot \mathbf{V}^{\mathbf{T}} \tag{19}$$

Die Finite-Elemente-Methode ist ein numerisches Verfahren, das in vielen Gebieten der Physik genutzt werden kann. Hierbei wird die Berechnung eines Problems, beispielsweise eines Körpers, in finite Elemente, also eine endliche Anzahl an Elementen, aufgeteilt. Diese endliche Anzahl lässt sich dann berechnen.

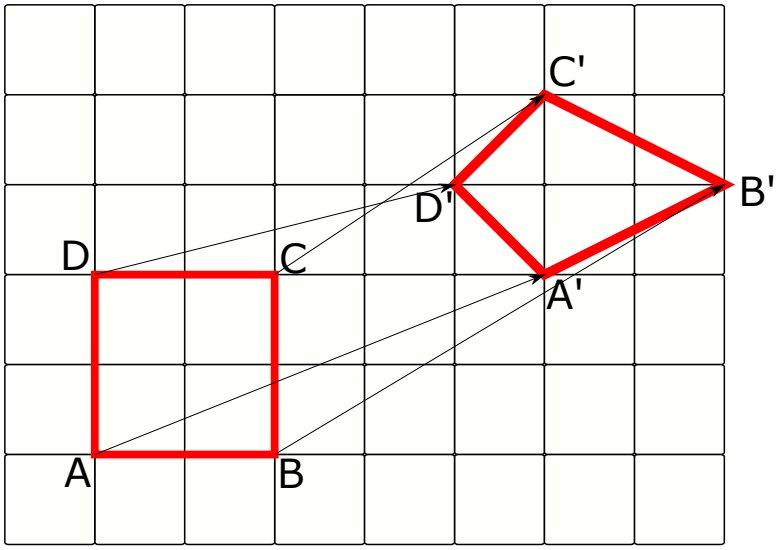


Abbildung 3: Deformationsbeispiel

In Abbildung 3 sehen wir eine Deformation vom ursprünglichen linken Quadrat hin zum neuen Rechteck A'B'C'D'. Ein sehr simples Beispiel zeigt hier eine mögliche Anwendung der Finite Elemente Methode: die Fläche zwischen A, B, C und D wird deformiert. Der Körper wird in einzelne Punkte aufgeteilt, in diesem Fall A, B, C und D. Aus den 4 bekannten Punkten lässt sich die Deformationsfunktion $\mathbf{u}(\mathbf{X})$ errechnen. So ist beispielsweise $u(A) = A'$. Dann kann über lineare Interpolation für alle Punkte innerhalb der Fläche die neue Position bestimmt werden. Der Deformationsgradient an einem Punkt ist wie folgt definiert:

$$\mathbf{F} = \mathbf{I} + \frac{\partial \mathbf{u}}{\partial \mathbf{X}} \tag{20}$$

[12] `http://www.mathematik.uni-ulm.de/m5/balser/Skripten/LA2.pdf`, Einsicht am 13.04.2019

Der Geschwindigkeitsgradient (L) funktioniert ähnlich wie die Deformationsgradienten und ist definiert durch:

$$\mathbf{L} = \frac{\partial \mathbf{v}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial v_x}{\partial x} & \frac{\partial v_x}{\partial y} & \frac{\partial v_x}{\partial z} \\ \frac{\partial v_y}{\partial x} & \frac{\partial v_y}{\partial y} & \frac{\partial v_y}{\partial z} \\ \frac{\partial v_z}{\partial x} & \frac{\partial v_z}{\partial y} & \frac{\partial v_z}{\partial z} \end{bmatrix} \tag{21}$$

Da die Ableitung von den aktuellen Koordinaten $\mathbf{x}$ ist, und nicht von den ursprünglichen Koordinaten $\mathbf{X}$, ist zu erkennen, dass es sich nicht um eine Lagrangesche-, sondern um eine Eulersche Größe handelt.
Dennoch gibt es einen Weg, den Geschwindigkeitsgradienten über F, den Deformationsgradienten, darzustellen. Wie man in Rechnung 22 sehen kann, ist $\dot{\mathbf{F}} = \partial \mathbf{v} / \partial \mathbf{X}$, wobei $\dot{\mathbf{F}}$ als Ableitung des Deformationsgradienten nach der Zeit definiert ist.

$$\dot{\mathbf{F}} \;=\; \frac{d}{dt}\left(\frac{\partial \mathbf{x}}{\partial \mathbf{X}}\right) \;=\; \frac{\partial}{\partial \mathbf{X}}\left(\frac{d\mathbf{x}}{dt}\right) \;=\; \frac{\partial \mathbf{v}}{\partial \mathbf{X}} \tag{22}$$

Durch Anwendung der Kettenregel erhält man:

$$\dot{\mathbf{F}} \;=\; \frac{\partial \mathbf{v}}{\partial \mathbf{X}} \;=\; \left(\frac{\partial \mathbf{v}}{\partial \mathbf{x}}\right)\left(\frac{\partial \mathbf{x}}{\partial \mathbf{X}}\right) \tag{23}$$

und damit ergibt sich:

$$\dot{\mathbf{F}} = \mathbf{L} \cdot \mathbf{F} \tag{24}$$

Durch multiplizieren mit F^{-1} kann L errechnet werden. Das bedeutet, man kann den Geschwindigkeitsgradienten errechnen, wenn der Deformationsgradient bekannt ist, und umgekehrt.

Die materielle Ableitung kann für die Berechnung der Partikel genutzt werden. Die Definition der materiellen Ableitung der Dichte ρ ist gegeben als:

$$\frac{D\rho}{Dt} = \frac{\partial \rho}{\partial t} + v \cdot \frac{\partial \rho}{\partial x} = \frac{\partial \rho}{\partial t} + v \cdot \nabla \rho \tag{25}$$

In einem vorherigen Abschnitt wurden in Gleichung (8) die Erhaltungssätze vorgestellt :

$$\frac{D\rho}{Dt} = 0, \qquad \rho \frac{Dv}{Dt} = \nabla * \sigma + \rho g \tag{26}$$

Anhand des Erhaltungssatzes kann für $\frac{D\rho}{Dt}$ aus Rechnung (25) 0 eingetzt werden. Dies zeigt, dass sich die Dichte eines Punktes in einem Kontrollvolumen über die Zeit nicht verändert. Der Punkt, der sich durch das Volumen bewegt, erhält also eine konstante Dichte.

Aus dem zweiten Teil der Gleichung wird geschlossen, dass der Punkt anhand seiner Bewegung verfolgt werden kann.
Wenn alle Punkte innerhalb des Volumens bleiben, also keiner das Volumen verlässt, so ist damit auch die Erhaltung der Masse gegeben.
In der Material-Point-Methode werden nicht Punkte, sondern Partikel genutzt. Diese haben Eigenschaften wie die Position x_p, die Geschwindigkeit v_p, die Masse m_p und den Deformationsgradienten F_p.
Schnee ist ein kompressibles Material und daher kann sich bei einer Stauchung die Dichte eines Schneepartikels ändern. Die Material-Point-Method ignoriert diese Tatsache, da der in 8 geltende abgewandelte Satz zur Erhaltung der Masse nur für inkompressible Materialien gilt.

Die Cauchy-Impuls-Gleichung geht aus dem zweiten Newtonschen Gesetz hervor:

$$ma = \sum F \Leftrightarrow \int_\Omega \rho \cdot \frac{Dv}{Dt} dV \Leftrightarrow \int_\Omega \nabla \cdot \sigma dV + \int_\Omega \rho \cdot g dV \Leftrightarrow \rho \cdot \frac{Dv}{Dt} = \nabla \cdot \sigma + \rho \cdot g \tag{27}$$

Sie beschreibt, wie ein Partikel bei der Bewegung durch das Feld von Kräften in einem Kontrollvolumen Ω von Geschwindigkeit beinflusst wird.
Dadurch können verschiedene Größen berechnet werden. Die Beschleunigung a ist gleich der materiellen Ableitung von v. Die Masse m kann aus dem Integral der Dichte ρ über Ω errechnet werden. Insgesamt wird die Kraft aus zwei Teilen zusammengesetzt. Die internen Spannungen $\nabla \cdot \sigma$ und die externe Erdanziehungskraft $\rho \cdot g$. Da sich diese Eigenschaften in jedem beliebigen Kontrollvolumen finden lassen, können die Integrale ausgelassen werden [Gra12].

Die Material-Point-Method basiert stark auf der in diesem Kapitel gezeigten Kontinuumsmechanik. Einige Materialparameter lassen sich über den Youngschen Modul und die Poissonzahl realisieren. Die Finite-Elemente-Methode zerlegt das Problem einer Schneemenge, die es zu simulieren gilt, in viele kleine Schneepartikel. Über die Deformationsgradienten dieser Partikel lässt sich dann die Verformung des Schnees berechnen und visualisieren.

3.2 Schnee

Schnee ist eine Form von festem Niederschlag und gehört zu der Gruppe der Hydrometeore[13]. Bei Hydrometeoren handelt es sich um alle Formen von kondensiertem[14] oder sublimiertem[15] Wasserdampf aus der Atmosphäre. Dazu zählen: Wolken, Nebel, Regen, Hagel und eben Schnee.
Dieses Kapitel geht auf die Entstehung und die verschiedenen Arten von Schnee ein. Dann wird die Dynamik beschrieben. Abschließend wird erläutert, welche Eigenschaften mit der Material-Point-Method simuliert werden sollen.

Die Entstehung von Schnee beginnt bei Wasserdampf in der Atmosphäre. Dieser kann zu kleinen Wassertropfen kondensieren, welche dann zum Erdboden fallen. Dieses Phänomen wird Niederschlag genannt. Schnee ist, wie oben geschrieben, fester Niederschlag. Das heißt, dass der Wasserdampf in der Atmosphäre nicht nur kondensiert sondern sogar resublimiert[16]. Er wird direkt zu Eis. Dieses Eis kann sich an kleinsten Partikeln in der Atmosphäre absetzen. Diese Partikel, zu denen beispielsweise kleine Staubpartikel zählen, werden Gefrierkerne genannt.
Mit der Zeit lagert sich dort weiterer Wasserdampf ab, wodurch die bekannten hexagonalen Strukturen der Eiskristalle entstehen [Lib05].
Ab wann der Wasserdampf kondensiert oder resublimiert, hängt von der Temperatur und der Luftfeuchtigkeit der umgebenden Luft ab.
Bei einer relativen Luftfeuchtigkeit von 100% hat die Luft die maximale Menge an Wasser aufgenommen. Wenn das Maximum der relativen Luftfeuchtigkeit überschritten ist, gibt es in der Luft mehr Wasserdampf, als diese aufnehmen kann.
Ab diesem Zeitpunkt kann Kondensation oder Resublimation eintreten, was zu Niederschlag führt.

[13] `https://www.dwd.de/DE/service/lexikon/begriffe/H/Hydrometeore.html`, Einsicht am 14.04.2019
[14] Kondensation: Übergang vom gasförmigen in den flüssigen Aggregatzustand
[15] Sublimation: Übergang vom festen zum gasförmigen Aggregatzustand
[16] Resublimation: Übergang vom gasförmigen in festen Aggregatzustand

Je nach Temperatur und Sättigung bzw. Übersättigung der Luft können die Eiskristalle andere Formen annehmen. Doch auch der Weg eines Eiskristalls aus der Atmosphäre zum Erdboden kann die Form beeinflussen [Lib05]. In Abbildung 4 sind sechs verschiedene Formen von Eiskristallen zu erkennen. Der sehr simpel geformte Kristall (a) ist bei sehr geringen Temperaturen unter $-20°C$ und geringer Luftfeuchtigkeit entstanden. Der Kristall (b) ist schon komplexer als (a).
Bei (c) ist ein stereotypischer Eiskristall zu sehen. Diese Art formt sich normalerweise bei Temperaturen um die $-15°C$ bei einer hohen Luftfeuchtigkeit. Eiskristall (d) ähnelt einem hexagonalen Zylinder. In (e) sind nadelartige Eiskristalle zu sehen, die zusammengewachsen sind oder sich verzweigt haben. Eiskristall (f) begann als hexagonaler Zylinder ähnlich wie (d), doch gegen Ende bildeten sich Platten-ähnliche Kristalle, ähnlich zu (a), an beiden Enden.

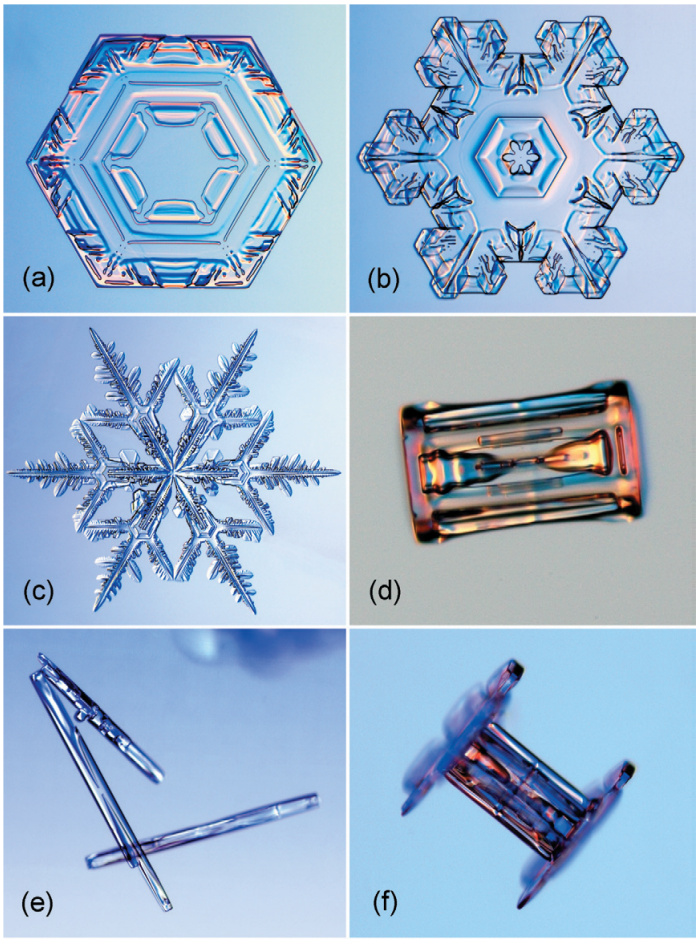


Abbildung 4: Diverse Formen von Eiskristallen [Lib05]

Die Klassifizierung verschiedener Schneearten hängt von dem Merkmal ab, anhand dessen man klassifizieren will. So kann man Schnee beispielsweise nach Feuchtigkeit, Dichte, Alter oder Auftreten kategorisieren. Beispielhaft werden zwei Kategorisierungsarten durchgegangen.
Im ersten Beispiel, der Klassifizierung anhand der Feuchtigkeit, gibt es 4 grobe Arten von Schnee. Zunächst der sehr trockene Pulverschnee, der kaum zusammenhält. Dann kommt der Feuchtschnee, welcher gut zusammenhält und dementsprechend gut geformt werden kann.
Der Nassschnee ist, wie der Name sagt, sehr nass und folglich auch sehr schwer. Er lässt sich Formen, wobei er allerdings Wasser verliert.
Die letzte Kategorie wäre der Faulschnee, bei dem es sich um eine Kombination aus viel Wasser und matschartigen Schneebrocken handelt. Dieser Schnee lässt sich nur noch schlecht formen.

Schneedichte in kg/m^3	Klassifikation
50-150	Neuschnee
100-200	Pulverschnee
150-450	körniger Schnee
350-600	gelagerter Schnee
500-850	Firnschnee
700-900	Gletscherschnee/Gletschereis

Tabelle 1: Klassifizierung von Schnee nach Dichte

Quelle: `https://www.dwd.de/DE/service/lexikon/Functions/glossar.html?lv2=102248&lv3=102438`, Einsicht am 14.04.2019

Tabelle 1 zeigt eine alternative Kategorisierung von Schnee. Hier wird der Schnee anhand seiner Dichte eingeteilt. Auffällig hierbei ist, dass sich die Bereiche, die den Schneearten zugewiesen werden, teilweise überlappen. Dies zeigt, dass sich Schnee nicht immer eindeutig zuordnen lässt, was zur Komplexität dieses Problems beisteuert.
Im Bereich von 50-150 kg/m^3 wird Schnee als Neuschnee bezeichnet. Dieser kann trocken und locker oder auch stark gebunden sein.
Von 100-200 kg/m^3 spricht man von Pulverschnee, der sehr lockeren Schneeart, die kaum zusammenhält und sich daher kaum verformen lässt.
Bei einer Dichte zwischen 150 und 450 kg/m^3 wird der sehr trockene Schnee körniger Schnee genannt, während man bei einer Dichte zwischen 350 und 600 kg/m^3 von feuchtem, gelagertem Schnee spricht. Liegt die Dichte im Bereich von 500-850 kg/m^3, so bezeichnet man den Schnee als Firnschnee. Dieser kann mehrere Jahre alt sein. Er ist schon so vereist, dass er wasserundurchlässig geworden ist. Abschließend wird von Schnee mit einer Dichte von 700-900kg/m^3 von Gletscherschnee gesprochen. Dieser entsteht durch zusätzliche Belastung auf Firnschnee.

Die Eigenschaften von Schnee hängen von Umweltfaktoren wie Alter, Feuchtigkeit oder Dichte ab. Um Schnee physikalisch korrekt zu simulieren, müssten all diese Faktoren betrachtet und in die Simulation über Parameter eingebaut werden.
In einer Simulation können aufgrund von Effizienz und Übersichtlichkeit jedoch nicht alle Faktoren berücksichtigt werden. Dies hat zwei Gründe. Erstens ist die Menge der tatsächlichen Faktoren unbekannt. Zweitens ist diese Menge zu groß, um wirklich alle zu beachten.
Die vier wichtigsten Eigenschaften, die über die Material-Point-Method simuliert werden sollen, werden von Stomakhin et al. [SSC+13] beschrieben:

- Die Erhaltung des Volumens. Flüssigkeiten sind nicht komprimierbar, daher ist die Erhaltung des Volumens immer gegeben. Im Gegensatz dazu steht Schnee, der sich komprimieren lässt. Dennoch ist die Erhaltung des Volumens bei Schnee wichtig, da die Volumen verschiedener Schneesorten unterschiedlich auf Kompression reagieren. Es ist möglich, dass eine Schneeart ihr Volumen besser erhält, als eine andere Art.

- Die Steifheit des Schnees beschreibt den Widerstand eines Körpers gegen eine Verformung. Das Gegenstück zur Steifheit ist die Elastizität, welche angibt, wie gut ein Stoff unter Krafteinwirkung seine Form verändern und nach Wegfall der Kraft in die ursprüngliche Form zurückkehren kann.

- Die Plastizität eines Materials ist dessen Fähigkeit, sich durch die Einwirkung einer Kraft irreversibel zu deformieren und dann in dieser Form zu bleiben.

- Bruch beschreibt eine Deformation eines Körpers, der in mehrere Stücke und damit in neue, einzelne Körper zerteilt wird.

Die Material-Point-Method ist im Vergleich zu anderen Methoden gerade in der Simulation der letzten beiden Eigenschaften, sehr gut [SSC+13].
Die Simulation der vier genannten Haupteigenschaften wird über Parameter realisiert. Diese werden im nächsten Abschnitt 3.3 vorgestellt.

3.3 Materialmodell

Da sich Schnee als Material sehr dynamisch verhält und teilweise Eigenschaften von Wasser, teilweise aber auch von Eis besitzt, ist die Findung eines geeigneten Materialmodells sehr komplex.
Viele Arbeiten, die sich mit der Material-Point-Method oder vergleichbaren Verfahren beschäftigen, handeln von Sand. Einige Abschnitte der Verfahren zur Simulation von Sand können auf Schnee übertragen werden. Aufgrund der verschiedenen Eigenschaften von Schnee gegenüber Sand muss jedoch ein komplexeres Modell für Schnee gefunden werden. Beispielsweise ist Schnee komprimierbar, Sand aber nicht. Des Weiteren hängen die Eigenschaften von Schnee von äußeren Bedingungen wie Temperatur, Feuchtigkeit und Alter ab.
In diesem Abschnitt wird das auf der Arbeit von Stomakhin et al. [SSC$^+$13] beruhende Materialmodell vorgestellt, das in Verbindung mit der Material-Point-Method zum Einsatz kommt. Folglich gilt die Arbeit als Referenz für sämtliche Formeln und Berechnungen.
Es basiert auf der im Kapitel 3.1 gezeigten Kontinuumsmechanik, ist jedoch an einigen Stellen vereinfacht worden, um an mehr Performanz und Kontrolle in der Simulation zu gelangen.

Die totale elastische potentielle Energie kann in Form der Energiedichte $\mathbf{\Psi}$ ausgedrückt werden, wobei Ω^0 die nicht deformierte Ausgangskonfiguration des Materials ist:

$$\int_{\Omega^0} \Psi(F_E(X), F_P(X))dX \tag{28}$$

Die Material-Point-Method nutzt ein kartesisches Gitter. Auf diesem Gitter befinden sich Knoten mit ihren gedachten deformierten Positionen $\hat{x}$. Die Material-Point-Method verformt dieses Gitter jedoch nicht wirklich. Die Deformation einer Position $\hat{x}_i$ eines Gitterknotens i ist definiert als $\hat{x}_i = x_i + \nabla t v_i$ und damit nur von der Geschwindigkeit dieses Gitterknotens abhängig. Mit Hilfe von V_p^0, dem initialen Volumen eines Schneepartikels p kann man die Definition des Zusammenhangs zwischen der Energie Ψ und dem totalen elastischen Potential des Gitters der Material-Point-Method aufstellen:

$$\Phi(\hat{x}) = \sum_p V_p^0 \Psi(\hat{F}_{Ep}(\hat{x}), F_{Pp}^n) \tag{29}$$

F_p wird nun in einen elastischen Teil $\hat{F}_{Ep}$ und einen plastischen Teil F_{Pp} aufgeteilt. F^n_{Pp} ist definiert als der plastische Deformationsgradient zur Zeit t^n. Den elastischen Deformationsgradienten $\hat{F}_{Ep}(\hat{x})$ kann man der Arbeit von Sulsky et al. entnehmen [SZS95] :

$$\hat{F}_{Ep}(\hat{x}) = \left(I + \sum_i (\hat{x}_i - x_i)(\nabla w^n_{ip})^T \right) F^n_{Ep} \tag{30}$$

In der folgenden Gleichung ist die partielle Ableitung des Potentials aus (29) nach $\hat{x}_i$ mit Hilfe der partiellen Ableitung des elastischen Deformatiosngradienten aus der Formel (30) sichtbar. Diese geben die Kraft f_i am Gitterknoten i an, die durch die Spannung entsteht.

$$-f_i(\hat{x}) = \frac{\partial \Phi}{\partial \hat{x}_i}(\hat{x}) = \sum_p V^0_p \frac{\partial \Psi}{\partial F_E}(\hat{F}_{Ep}(\hat{x}), F^n_{Pp})(F^n_{Ep})^T \nabla w^n_{ip} \tag{31}$$

Eine alternative Schreibweise erhält man durch Nutzen der Cauchy-Spannung, die definiert ist als:

$$\sigma_p = \frac{1}{J^n_p} \frac{\partial \Psi}{\partial F_E}(\hat{F}_{Ep}(\hat{x}), F^n_{Pp})(F^n_{Ep})^T \tag{32}$$

und damit ergibt sich :

$$f_i(\hat{x}) = -\sum_p V^n_p \sigma_p \nabla w^n_{ip} \tag{33}$$

wobei V^n_p das Volumen ist, das Partikel p beim Zeitpunkt t^n einnimmt: $V^n_p = J^n_p V^0_p$. Bei J^n_p handelt es sich um die Determinante von F^n_p.

Der letzte unklare Term, ist $\Psi(F_E, F_P)$. Hierbei handelt es sich um die elasto-plastische Energiedichtefunktion. Diese ist von Stomakhin et al. abgewandelt [SHST12] und definiert als:

$$\Psi(F_E, F_P) = \mu(F_P)\|F_E - R_E\|^2_F + \frac{\lambda(F_P)}{2}(J_E - 1)^2 \tag{34}$$

Hierbei ist J_E die Determinante von F_E und J_P die von F_P. Zudem ist F_E durch Polarzerlegung dargestellt als $F_E = R_E S_E$. Zusätlich werden Funktionen des plastischen Deformationsgradienten genutzt, die Lamé-Parameter:

$$\mu(F_P) = \mu_0 e^{\xi(1-J_P)} \qquad \text{und} \qquad \lambda(F_P) = \lambda_0 e^{\xi(1-J_P)} \tag{35}$$

Hierbei ist ξ ein Parameter, der die Härte eines Materials beschreibt. Der Ausdruck $\|F_E - R_E\|^2_F$ bezeichnet die Frobeniusnorm der Matrix $F_E - R_E$.

Parameter	Notation	Wert
Kritische Kompression	θ_c	2.5×10^-2
Kritische Streckung	θ_s	7.5×10^-3
Härte-Parameter	ξ	10
Initiale Dichte (kg/m^3)	ρ_0	4.5×10^2
Initialer Youngscher Modul (Pa)	E_0	1.4×10^5
Poissonzahl	ν	0.2

Tabelle 2: Beispielhafte Ausgangsparameter nach [SSC+13]

Die Frobeniusnorm einer $m \times n$ -Matrix A ist definiert als $\|A\|_F = \sqrt{spur(AA^H)}$ wobei A^H die transponierte, konjugierte Matrix von A ist[17]. μ_0 und λ_0 sind die beiden Lamé-Konstanten die sich wie folgt definieren:[18]

$$\lambda_0 = \frac{\nu E}{(1+\nu)(1-2\nu)} \qquad \text{und} \qquad \mu_0 = \frac{E}{2(1+\nu)} \tag{36}$$

mit dem Youngschen Modul E und der Poissonzahl ν.

Nun werden von Stomakhin et al. [SSC+13] zwei weitere Parameter eingeführt: die kritische Kompression θ_c und die kritische Streckung θ_s. Diese beiden Parameter dienen als Schwellwerte für die plastische Deformation. Das bedeutet, dass Deformationen im Wertebereich von $[1-\theta_c, 1+\theta_s]$ elastische Deformationen sind. Werte, die diesen Bereich verlassen, verursachen eine plastische Deformation beim Objekt. Gleichung (36) sorgt dafür, dass das Material bei Komprimierung fester und gepackter und bei Streckung schwächer wird. Dadurch können Phänomene wie der Bruch von Schnee simuliert werden. Kleine Werte machen den Schnee puderiger, große Werte machen ihn klumpiger.
Der Härte-Parameter ξ beschreibt wie schnell ein Material bricht, nachdem die Deformation in den plastischen Wertebereich gekommen ist. Hohe Werte von ξ machen das Material brüchiger, kleine Werte machen es dehnbarer. Beispielsweise hat trockener Puderschnee eine kleine kritische Kompression und Streckung. Eisiger Schnee hat einen hohen Härte-Parameter und einen hohen Youngschen Modul. Wenn diese Parameter jedoch kleine Werte besitzen, erhält man einen eher schlammartig Schnee.
Eine beispielhafte Konfiguration der Ausgangsparameter ist in Tabelle 2 zu sehen.

[17] `http://mathworld.wolfram.com/FrobeniusNorm.html`, Einsicht am 14.04.2019

[18] `http://scienceworld.wolfram.com/physics/LameConstants.html` , Einsicht am 14.04.2019

Dieser Abschnitt befasste sich mit dem Materialmodell für Schnee nach der Arbeit von Stomakhin et al. [SSC+13]. Es wurde gezeigt, wie Kräfte berechnet werden können, und wie sich die eingeführten Parameter auf die Simulation auswirken.

3.4 Material-Point-Method

In diesem Kapitel wird die Material-Point-Method nach [SSC+13] vorgestellt. Sie basiert auf der Kontinuumsmechanik aus Kapitel 3.1 und nutzt das Materialmodell aus dem vorherigen Kapitel 3.3.
Die Idee der Material-Point-Method ist es, Lagrange-Partikel mit einem kartesischen Gitter zu vereinen. Der Schnee wird hierbei als Lagrange-Partikel implementiert.
Er hat Eigenschaften wie Position x_p, Geschwindigkeit v_p, Masse m_p und Deformationsgradient F_p.
Da diese Partikel nicht miteinander verbunden sind, wären viele Berechnungen ohne Weiteres nicht möglich. Daher wird zusätzlich ein kartesisches Gitter eingesetzt.
Manche Berechnungen müssen über Nachbarinformationen verfügen. Diese können mit Hilfe der Finite-Elemente Methode auf dem Gitter berechnet werden. Die Ergebnisse müssen dann zurück auf die Partikel transferiert werden. Wie in Kapitel 3.3 angesprochen, wird dieses Gitter dabei nicht selber deformiert.
Dadurch werden die Vorteile der Lagrange-Partikel genutzt, während die Nachteile, die durch das Fehlen einer Verbindung zwischen den Partikeln entstehen, durch das Gitter ausgeglichen werden [SSC+13].
In Abbildung 5 ist der Ablauf der Material-Point-Method in Einzelschritten beschrieben und visualisiert. Die obere und die untere Reihe in der Abbildung bezeichnen Operationen, die mit den Daten der Partikel gerechnet werden. Die Schritte zwischen den Hälften bestehen aus jenen Operationen, die auf dem Gitter berechnet werden müssen. Die Methode besteht insgesamt aus zehn Schritten, die im Verlauf dieses Kapitels noch detailliert beschrieben werden.

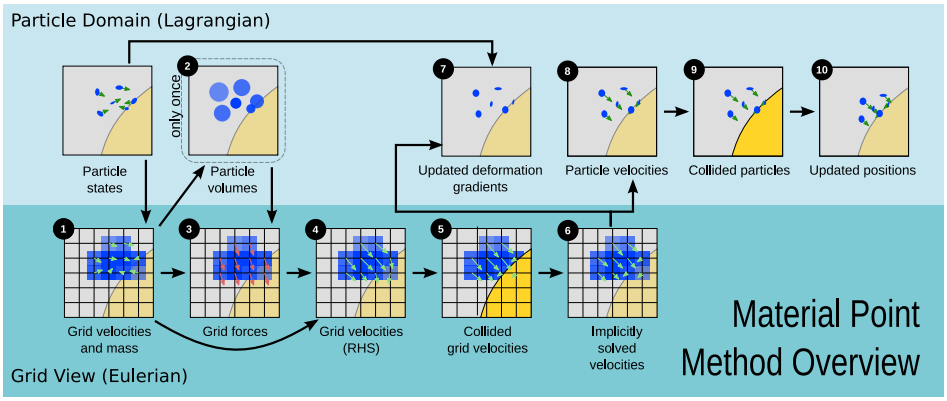


Abbildung 5: Übersicht über die Material-Point-Method nach [SSC+13]

Für das Gitter werden dyadische[19] Produkte als eindimensionale kubische B-Splines[20] [21] als Interpolationsfunktion genutzt [SSC+13] :

$$N_i^h(x_p) = N(\frac{1}{h}(x_p - ih))N(\frac{1}{h}(y_p - jh))N(\frac{1}{h}(z_p - kh)) \tag{37}$$

Wobei $\boldsymbol{i} = (i, j, k)$ die Koordinaten, also der Index des Knotens auf dem Gitter ist. Die relative Position $\boldsymbol{x_p}$ ist gegeben als $\boldsymbol{x_p} = (x_p, y_p, z_p)$. Die konstante h ist der Abstand zwischen den einzelnen Gitterknoten, also die Seitenlänge einer Gitterzelle. Die Funktion $N(x)$ ist definiert [SSC+13] als:

$$N(x) = \begin{cases} \frac{1}{2}|x|^3 - x^2 + \frac{2}{3}, & \text{falls } 0 \leq |x| < 1 \\ -\frac{1}{6}|x|^3 + x^2 - 2|x| + \frac{4}{3}, & \text{falls } 1 \leq |x| < 2 \\ 0, & \text{sonst} \end{cases} \tag{38}$$

Daraus lässt sich der Gradient der Gewichtungsfunktion berechnen, in dem man die 1. Ableitung dieser bildet:

$$\nabla N_i^h(x_p) = \begin{pmatrix} \nabla N(\frac{1}{h}(x_p - ih))N(\frac{1}{h}(y_p - jh))N(\frac{1}{h}(z_p - kh)) \\ N(\frac{1}{h}(x_p - ih))\nabla N(\frac{1}{h}(y_p - jh))N(\frac{1}{h}(z_p - kh)) \\ N(\frac{1}{h}(x_p - ih))N(\frac{1}{h}(y_p - jh))\nabla N(\frac{1}{h}(z_p - kh)) \end{pmatrix} \tag{39}$$

$$\nabla N(x) = \begin{cases} \frac{3}{2}|x|x - 2x, & \text{falls } 0 \leq |x| < 1 \\ -\frac{1}{2}|x|x + 2x - 2\frac{x}{|x|}, & \text{falls } 1 \leq |x| < 2 \\ 0, & \text{sonst} \end{cases} \tag{40}$$

[19] Dyadisches Produkt: ein Produkt von zwei Vektoren mit einer Matrix als Resultat
[20] B-Spline: Basis-Spline
[21] Spline: eine Funktion, die aus Polynomen n-ten Grades zusammengesetzt ist

Im Folgenden werden die Notationen $w_{ip} = N_i^h(x_p)$ und $\nabla w_{ip} = \nabla N_i^h(x_p)$ eingeführt. Um Berechnungen auf dem Gitter durchführen zu können, müssen die Masse und die Geschwindigkeit von den Partikeln auf das Gitter übertragen werden [SSC+13].
Die Interpolationsfunktionen dienen dabei als Gewichtung. Diese Gewichtung basiert auf der Position x_p des Partikels. Nachdem die Informationen des Partikels auf dem Gitter gewichtet wurden, kann eine neue Geschwindigkeit berechnet werden. Diese kann dann auf den Partikel zurückgewichtet werden. Mit der neuen Geschwindigkeit kann die Position des Partikels aktualisiert werden.

Nun folgt die Beschreibung der zehn Schritte der Material-Point-Method, die in Abbildung 5 gezeigt wurden. Die Nummerierung stimmt mit der Abbildung überein [SSC+13]:

1. **Übertragung der Partikeldaten auf das Gitter.** Der erste Schritt der Material-Point-Method ist, die Partikeldaten auf das Gitter zu übertragen. Die Masse des Partikels wird über $m_i^n = \sum_p m_p w_{ip}^n$ gewichtet. Nun muss die Geschwindigkeit mittels einer Gewichtungsfunktion auf das Gitter übertragen werden. Die Gewichtung mittels w_{ip}^n verletzt jedoch die Impulserhaltung. Daher wird die Geschwindigkeit mit normalisierten Gewichten $v_i^n = \sum_p v_p^n m_p w_{ip}^n / m_i^n$ übertragen. Dies steht im Gegensatz zu den meisten FLIP-Implementationen.

2. **Initiale Berechnung von Partikelvolumen und -dichte.** Diese initiale Berechnung wird nur ein mal zu Beginn der Simulation durchgeführt. Der Grund hierfür ist, dass die initiale Konfiguration der Partikel ein Ausgangsvolumen haben muss. Über den Term m_i^0/h^3 kann die Dichte einer Zelle geschätzt werden, daher sieht die Gewichtung der Dichte wie folgt aus: $\rho_p^0 = \sum_i m_i^0 w_{ip}^0 / h^3$. Somit lässt sich auch das initiale Volumen eines Partikels abschätzen mit: $V_p^0 = m_p \, \rho_p^0$

3. **Berechnung der Kräfte auf dem Gitter.** Nun werden die Kräfte auf dem Gitter berechnet. Hierfür wird die Gleichung (33) genutzt, die bereits im Kapitel 3.3 des Materialmodells beschrieben wurde. Für $\hat{x}_i$ wird x_i eingesetzt.

4. **Die Aktualisierung der Geschwindigkeiten auf dem Gitter**. Diese lassen sich über die Formel $v_i^{n+1} = v_i^n + \Delta t(m_i^{-1} f_i^n + g)$ berechnen. Hierbei wird die Erdanziehung g mit eingerechnet.

5. **Gitterbasierte Kollision mit Rigid-Bodys[22].** Die Kollision wird in Kapitel 5.5 genauer untersucht.

[22] Rigid-Body: fester Körper, der nicht deformiert werden kann

6. **Das lineare Gleichungssystem der semi-impliziten Integration** wird gelöst. Dadurch erhält man eine genauere Berechnung von v_i^{n+1}. In dieser Bachelorarbeit wird dieser Schritt nicht implementiert. Dies resultiert in einer weniger genauen Berechnung.

7. **Berechnung des aktualisierten Deformationsgradienten.** Wie in Kapitel 3.3 schon beschrieben, wird bei der Material-Point-Method der Deformationsgradient F_p eines Partikels in einen elastischen- (F_{Ep}) und einen plastischen Teil (F_{Pp}) aufgeteilt $F_p = F_{Ep}F_{Pp}$. Analog zu Gleichung (30) wird $\hat{F}_{Ep}^{n+1}$ wie folgt definiert:

$$\hat{F}_{Ep}^{n+1} = (I + \Delta t \nabla v_p^{n+1}) F_{Ep}^n \tag{41}$$

Zusätzlich wird $\hat{F}_{Pp}^{n+1}$ als $\hat{F}_{Pp}^{n+1} = F_{Pp}^n$ definiert. Dadurch werden initiale Änderungen nur im elastischen Teil des Deformationsgradienten sichtbar.
Der gesamte Deformationsgradient ist folglich definiert durch:

$$F_p^{n+1} = (I + \Delta t \nabla v_p^{n+1}) F_{Ep}^n F_{Pp}^n = \hat{F}_{Ep}^{n+1} \hat{F}_{Pp}^{n+1} \tag{42}$$

Der Geschwindigkeitsgradient ∇v_p^{n+1} kann durch ∇v_p^{n+1}
$= \sum_i v_i^{n+1} (\nabla w_{ip}^n)^T$ berechnet werden.
Sobald nun der elastische Teil $\hat{F}_{Ep}^{n+1}$ den Schwellwert der, in Kapitel 3.3 beschriebenen, kritischen Deformation überschreitet, wird dieser überschrittene Anteil an den plastischen Teil $\hat{F}_{Pp}^{n+1}$ des Deformationsgradienten übergeben. Deshalb wird die Singulärwertzerlegung des elastischen Teils berechnet: $\hat{F}_{Ep}^{n+1} = U_p \hat{\Sigma}_p V_p^T$. Die Singulärwerte innerhalb der Diagonalmatrix $\hat{\Sigma}_p$ werden nun auf den in Kapitel 3.3 beschriebenen Wertebereich $[1 - \theta_c, 1 + \theta_s]$ beschränkt, $\Sigma_p = clamp(\hat{\Sigma}_p, [1 - \theta_c, 1 + \theta_s])$. Dieser Wertebereich ist durch die kritische Kompression θ_c und die kritische Streckung θ_s definiert. Nun können die beiden finalen Komponenten des Deformationsgradients berechnet werden:

$$F_{Ep}^{n+1} = U_p \Sigma_p V_p^T \qquad \text{und} \qquad F_{Pp}^{n+1} = V_p \Sigma_p^{-1} U_p^T F_p^{n+1} \tag{43}$$

8. **Die Geschwindigkeiten der Partikel werden aktualisiert.** Die neuen Partikelgeschwindigkeiten setzen sich aus zwei Teilen zusammen:

$$v_p^{n+1} = (1-\alpha)v_{PICp}^{n+1} + \alpha v_{FLIPp}^{n+1} \tag{44}$$

Der erste Teil ist PIC-basiert und wird wie folgt definiert: $v_{PICp}^{n+1} = \sum_i v_i^{n+1} w_{ip}^n$. Der zweite Teil ist FLIP-basiert und ist gegeben als: $v_{FLIPp}^{n+1} = v_p^n + \sum_i (v_i^{n+1} - v_i^n) w_{ip}^n$. Für α wird 0.95 eingesetzt.

9. **Kollisionstest von Rigid-Bodys und Partikeln.** Weil die Partikelgeschwindigkeiten v_p^{n+1} leicht von den Geschwindigkeiten auf dem Gitter abweichen, muss ein weiteres mal auf Kollision getestet werden. Wie bereits im 5. Schritt zu sehen ist, werden die Details der Kollision im Kapitel 5.5 beschrieben.

10. **Neue Partikelpositionen werden berechnet.** Die Partikelpositionen können nun aktualisiert werden. Hierfür berechnet man $x_p^{n+1} = x_p^n + \Delta t v_p^{n+1}$ mit Hilfe der neuen Partikelgeschwindigkeit v_p^{n+1}.

In diesem Kapitel wurde die Material-Point-Method vorgestellt. Sie kombiniert Lagrange-Partikel mit einem kartesischem Gitter. Für einige Rechnungen werden Informationen von Partikeln an das Gitter übergeben und nachdem diese ausgeführt wurden, werden die Daten der Partikel aktualisiert. Hierbei hilft die Finite-Elemente-Methode um Gradienten zu bestimmen, mit der die Geschwindigkeit und damit die Position der Partikel aktualisiert werden kann.

4 Compute-Shader

Während normale Prozessoren (CPU[23]) meist aus einer geringen Anzahl von starken Kernen zusammengesetzt sind, so bestehen Grafikprozessoren meist aus sehr vielen weniger starken Kernen. Dadurch kann bei parallelisierbaren Aufgaben eine sehr hohe Effizienz erreicht werden, da auf jedem dieser Prozessorkerne gleichzeitig eine Berechnung ausgeführt werden kann. Ursprünglich wurde diese Form der Hardware für Rendering[24] in der Computergraphik entwickelt, da sich Berechnungen pro Pixel gut parallel ausführen lassen konnten.
GPGPU-Techniken ermöglichen, auch Rendering-unabhängige Berechnungen auf Grafikkarten durchzuführen. Dieses Kapitel befasst sich mit Compute-Shadern, der GPGPU-Technik innerhalb von OpenGL.
Compute-Shader wurden in OpenGL mit Version 4.3 eingeführt und sind seitdem ein fester Bestandteil. Es handelt sich um zusätzliche Shader, die wie die bisherigen Shader, in GLSL[25] geschrieben und auf der Grafikkarte ausgeführt werden. Im Gegensatz zu anderen Shadern, wie etwa dem Vertex- oder Fragment-Shader, ist der Compute-Shader kein Teil der OpenGL-Rendering-Pipeline. Das bedeutet, dass er separat aufgerufen werden muss und keine vordefinierten Ein- oder Ausgaben hat.
Anwendungen, wie etwa die Material-Point-Method (Kapitel 3.4), deren Berechnung sich gut parallelisieren lassen, können stark von Compute-Shadern profitieren, da mit ihnen viele Berechnungen gleichzeitig ablaufen können. Im Folgenden wird darauf eingegangen, wie Compute-Shader mit Buffern interagieren, wie Compute-Shader ausgeführt werden und wie man die einzelnen Berechnungen synchronisieren kann.

4.1 Buffer

Buffer-Objekte sind von OpenGL genutze Datenstrukturen die ein Array[26] von Daten speichern. Sie werden von verschiedenen Shadertypen verwendet, um auf Daten, wie beispielsweise Vertex-Daten, zuzugreifen[27]. Wie jeder andere Shader innerhalb von OpenGL, können auch Compute-Shader auf Buffer zugreifen.
Zunächst werden im C++-Teil des Programms Shader-Storage-Buffer-Objects (SSBO) angelegt, wie unten am Beispiel des Buffers für die Position und die Masse eines Partikels gezeigt [Quellcode 1]. Durch SSBOs können Arrays von Daten an die Shader geschickt werden.

[23] CPU: central processing unit, zentrale Verarbeitungseinheit
[24] Rendering: Berechnung von Bildern durch den Computer
[25] GLSL: OpenGl-Shading-Language
[26] Array: Listen-Datentyp
[27] `https://www.khronos.org/opengl/wiki/Buffer_Object`, Einsicht 09.04.2019

Quellcode 1: Anlegen eines SSBO

```
1  //Particle Position Buffer
2    glGenBuffers(1, &positionMassBuffer);
3    glBindBuffer(GL_SHADER_STORAGE_BUFFER, positionMassBuffer);
4    glBufferData(GL_SHADER_STORAGE_BUFFER, sizeof(glm::vec4)*
       m_particles.size(), NULL, GL_STATIC_DRAW);
5    positions = (glm::vec4*)(glMapBufferRange(
       GL_SHADER_STORAGE_BUFFER, 0, sizeof(glm::vec4) * m_particles.
       size(), GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT));
6    for (int i = 0; i < m_particles.size(); i++) {
7      positions[i] = glm::vec4(m_particles[i].position, m_particles[
       i].mass);
8    }
9    glUnmapBuffer(GL_SHADER_STORAGE_BUFFER);
10   glBindBufferBase(GL_SHADER_STORAGE_BUFFER,
       PARTICLE_POSITION_MASS_BUFFER, positionMassBuffer);
```

In Zeile 2 wird ein Buffer generiert und in der nächsten Zeile an die Variable „positionMassBuffer“gebunden.
Über den „glBufferData “-Befehl in Zeile 3 wird der nötige Speicher für den Buffer erzeugt. Hierzu wird die Größe des Datentyps für 4-elementige Vektoren, also „glm::vec4 “mit der Anzahl an Partikeln multipliziert. Dann folgt ein Nullpointer[28], der angibt, dass es noch keine Daten gibt, die wir in dem Buffer speichern wollen. GL_STATIC_DRAW ist ein Makro von OpenGL, das für diese Funktion angibt, wie der Zugriff auf diesen Speicher aussehen soll.
In Zeile 5 wird ein Pointer auf die noch leeren Daten des Buffers erzeugt, mit dem dann in den Zeilen 6 bis 8 der Buffer über eine For-Schleife befüllt wird.
Der hier gezeigte Positions- und Massenbuffer enthält Vektoren, die aus vier Elementen bestehen. Die ersten drei Elemete ergeben sich aus der Position des Partikels im Raum. Der letzte Eintrag des Vektors wird von der Masse des Partikels gebildet.
Zeile 9 und 10 geben dem Shader Zugriff auf den erstellten und befüllten Buffer. Um den Buffer zuordnen zu können, wurde ihm hier in Zeile 10 eine eindeutige Indentifikation gegeben, nämlich die des PARTICLE_POSITION_MASS_BUFFER, einer selbst definierten Konstante.

[28]Nullpointer: Ein Pointer (Zeiger), der auf Null (auf Nichts) zeigt. Er wird oft als Spezialfall genutzt

Quellcode 2: SSBO-Zugriff im Shader

```
1 layout(std140, binding = 0) buffer particlePosMass {
2   vec4 ppm[ ];
3 };
```

Im Shader kann dann auf die gewünschten Buffer zugegriffen werden. Dafür muss man, wie in [Quellcode 2] gezeigt, einen „Shader-Storage-Block[29]“ anlegen. Dieser gibt den gebundenen, eindeutigen Ort des Buffers, in diesem Fall 0, und das Layout des Buffers an, in diesem Fall std140. Dann folgt der interne Name „particlePosMass “, der aber nur zur Übersicht gebraucht wird. Den tatsächlichen Inhalt des Buffers sieht man in Zeile 2. Er enthält Elemente vom Typ „vec4 “, wie oben beschrieben, und kann über den Variablennamen „ppm“aufgerufen werden.

4.2 Ausführung

Wie in Kapitlel 4 beschrieben, gehören die Compute-Shader nicht zur OpenGL-Rendering-Pipeline. Daher muss die Ausführung des Shader explizit aufgerufen werden.
Jeder Compute-Shader hat eine dreidimensionale lokale Größe (local size) und eine vom Nutzer definierte Anzahl an „Work-Groups “[30]. Die Work-Groups sind die Menge von Shader-Aufrufen, die den gleichen Code ausführen. Die lokale Größe der Work-Groups gibt an, wie viele Aufrufe des Shaders es innerhalb einer Work-Group gibt. Die gesamte Anzahl der Shader-Aufrufe setzt sich also aus der Anzahl der Work-Groups und der Anzahl der Aufrufe pro Work-Group zusammen.

```
1 void glDispatchCompute(
2   GLuint num_groups_x,
3   GLuint num_groups_y,
4   GLuint num_groups_z);
```

Beide Größen können vom Nutzer festgelegt werden: Über den oben gezeigten „glDispatchCompute()“-Befehl wird die Ausführung des Compute-Shaders gestartet. Gleichzeitig werden als Parameter drei Größen mitgegeben, die die Größe der Work-Groups darstellen.
Die lokale Größe der einzelnen Work-Groups muss dann im Shader angegeben werden, wie in Quellcode [3] zu sehen ist.

[29] https://www.khronos.org/opengl/wiki/Interface_Block_(GLSL)#Shader_storage_blocks, Einsicht am 09.04.2019

[30] https://www.khronos.org/opengl/wiki/Compute_Shader#Compute_space, Einsicht am 09.04.2019

Quellcode 3: Lokale Größen

```
1 layout(local_size_x =1024, local_size_y =1,local_size_z =1)in;
```

Wenn also der glDispatchCompute-Befehl mit den Größen 16, 8 und 64 aufgerufen wird und die lokale Größe im Shader auf (128,1,1) gesetzt wurde, gibt es insgesamt $16 * 8 * 64 * 128 = 1048576$ Aufrufe des Shaders. Um im Shader zu wissen, um welchen Aufruf es sich handelt, gibt es Built-In-Variablen[31] wie „gl_GlobalInvocationId "die die globale Aufruf ID der momentan ausgeführten Shader-Instanz speichert.

4.3 Synchronisation

Bei Compute-Shadern werden viele Aufrufe gleichzeitig und parallel ausgeführt. Hierbei muss beachtet werden, dass alle Shader-Invokationen gleichzeitig auf den geteilten Speicher zugreifen können. Dadurch kann es zu Problemen kommen, wenn zum Beispiel in zwei Berechnungen der selbe Wert in der selben Stelle im Speicher um genau 1 erhöht wird. Denn dann müsste der Wert sich insgesamt um 2 erhöhen, tut es aber nicht, da beide Operationen gleichzeitig ausgeführt werden. Der selbe Wert wird von beiden Aufrufen um 1 erhöht und dann zurück in den Speicher geschrieben.

Das Ziel ist durch Synchronisierung eine Speicherkonsistenz zu schaffen. Das bedeutet, dass Zugriffe auf den geteilten Speicher synchronisiert werden müssen.
Dies gelingt durch zwei Methoden, die „Memory-Barriers "und die „atomaren Operationen ".
Memory Barriers können durch den folgenden Befehl auf C++-Seite ausgelöst werden:

Quellcode 4: Memory Barrier

```
1   glDispatchCompute(countParticles / THREAD_COUNT + 1, 1, 1);
2   glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
```

Die Memory-Barrier kann nach einem glDispatchCompute()-Aufruf folgen und stellt sicher, dass alle Aufrufe des Shaders abgearbeitet wurden, bevor der nächste Shader aufgerufen werden kann.
Die Zweite Möglichkeit die Zugriffe zu synchronisieren besteht in den atomaren Operationen. Hierbei handelt es sich um spezielle Funktionen, die garantieren, dass während der Ausführung, kein anderer Teil des Rechners auf die von der Operation benötigten Speicherstellen zugreifen kann. Die Operation hat zu dem Zeitpunkt der Ausführung das alleinige Lese- und Schreibrecht.

[31] Built-In: intrinsische Variablen

```
1 int atomicAdd( inout int mem, int data);
2 uint atomicAdd( inout uint mem,uint data);
```

Hier sehen wir die Deklaration einer der atomaren Operationen: dem atomaren Addieren (atomic Add). Dabei steht mem für das Ziel der Operation im Speicher und Data für den Wert, der auf mem addiert wird. Das Ergebnis der Addition wird dann im Ziel, also in mem, gespeichert.

5 Implementation

Dieses Kapitel beschäftigt sich mit der Implementation der Material-Point-Method durch Compute-Shader mit OpenGL. Zunächst wird in Kapitel 5.1 beschrieben, welche externe Bibliotheken genutzt wurden.
Dann folgt eine Beschreibung des Aufbaus der Simulation im gleichnamigen Kapitel 5.2.
Die nächsten beiden Kapitel 5.3 und 5.4 handeln jeweils von der Implementation der Partikel bzw. des Gitters.
Im darauffolgenden Kapitel 5.5 wird die Berechnung der Kollision näher erläutert.
Abschnitt 5.6 geht dann auf die Implementation der Material-Point-Method durch Compute-Shader ein.
Abschließend wird in Kapitel 5.7 das Rendering beschrieben.

5.1 Externes

Dieses Kapitel befasst sich mit Bibliotheken, Codeteilen oder sonstige Formen externer Arbeit, die in diese Implementation integriert wurden.
Das Tool, das zum Erstellen des Builds genutzt wurde, ist CMake[32]. Hiermit wird aus dem vorhandenen Quellcode ein Visual-Studio 17[33]-Projekt generiert.
Für die Simulation und das Rendering ist die Einbindung von OpenGL und diversen anderen Bibliotheken nötig. GLEW[34] ermöglicht Zugriff auf alle von der Hardware nutzbaren OpenGL-Funktionen.
GLFW[35] dient sowohl der Erstellung und dem Management von Fenstern als auch den Eingabeformen über Maus und Tastatur.
GLM[36] ist eine Mathematik-Bibliothek, die speziell auf OpenGL abgestimmt ist. Sie wurde für sämtliche Vektor- und Matrixberechnungen genutzt.
Zum Einlesen von ganzen Modellen wurde die Open-Asset-Importer-Library (Assimp)[37] genutzt, während für das Einlesen von Texturen die stb_image-Bibliothek[38] genutzt wurde.
Die in der Arbeit verwendeten Datentypen wie „vec4“,„ivec3“oder „mat4“ stammen alle aus OpenGL. Es handelt sich um Datentypen für Vektoren, Integervektoren und Matrizen.
Zum Debuggen der Ein- und Ausgaben der Compute-Shader wurde RenderDoc verwendet.

[32] `https://cmake.org/`, Einsicht 11.04.2019
[33] `https://visualstudio.microsoft.com/de/`, Einsicht 11.04.2019
[34] `http://glew.sourceforge.net/`, Einsicht 11.04.2019
[35] `https://www.glfw.org/`, Einsicht 11.04.2019
[36] `https://glm.g-truc.net/0.9.9/index.html`, Einsicht 11.04.2019
[37] `http://www.assimp.org/`, Einsicht 11.04.2019
[38] `https://github.com/nothings/stb/blob/master/stb_image.h`, Einsicht 11.04.2019

Eine Implementierung der Polar- und Singulärwertzerlegung wurde mit Erlaubnis von Meyer [Mey15][39] übernommen. Zudem diente die Arbeit als Orientierung bei Berechnungen und Strukturierung.

5.2 Aufbau

Dieses Kapitel beschäftigt sich mit dem Aufbau und dem Ablauf der Simulation. Am Anfang des Programms wird eine Szene erstellt. Diese setzt sich je nach Szene aus verschiedenen Bausteinen zusammen.
Ein Baustein, der immer genutzt wird, ist die äußere Kollisionsbox, ein großer durchsichtiger Würfel, in dem sich die ganze Szene abspielen soll. Dies dient dem einfachen Zweck, dass es einen kontrollierbaren, abgegrenzten Raum gibt, in der die Simulation stattfinden kann.
Dann werden je nach Szene Modelle platziert. Diese können entweder als Schnee-Partikelsysteme oder als Kollisionsobjekte definiert sein.
Zudem wird ein Grid[40] der gewünschten Größe, die in einer speziellen Datei für Konstanten definiert werden kann, erstellt.
All diese Daten befinden sich noch auf der C++ Seite der Simulation. Dies ändert sich wenn dann die Simulations-Engine gestartet wird. Nun werden alle SSBOs initialisiert und mit den Werten der Szene befüllt. Außerdem werden alle Compute-Shader initialisiert und die schon bekannten Uniform-Variablen werden zugewiesen.

In den Zeilen 25-27 von Quellcode 5 sieht man schon, dass die ersten drei initialen Berechnungen ausgeführt werden. Hier wird die initiale Berechnung der Dichte ausgeführt.
Nachdem die Simulations-Engine initialisiert wurde, wird noch die Rendering-Engine initialisiert. Hier werden Texturen geladen und Shader für die Darstellung vorbereitet.

[39] `https://github.com/MeyerFabian/snow/`, Einsicht 11.04.2019
[40] Grid: Gitter

Quellcode 5: Initialisierung der Simulation

```
1  //INIT ALL SSBOS
2    scene->particleSystem->initSSBO();  //Initialisierung des Schnee
       -Partikel-SSBO
3    scene->grid->initSSBO();       //Initialisierung des Grid-SSBO
4    scene->collisionSystem->initSSBO(); //Initialisierung des
       Kollisions-SSBO
5
6
7    GLuint particleCount = scene->particleSystem->size(); //Anzahl
       der Partikel
8    GLuint colliderCount = scene->collisionSystem->colliderList.size
       (); //Anzahl der Kollisionsobjekte
9
10   /*
11     INIT ALL COMPUTE-SHADERS
12   */
13   scene->grid->initReset();
14   scene->collisionSystem->initColliders();
15   scene->grid->initGridMass(particleCount);
16   scene->grid->initParticleVolumeAndDensity(particleCount);
17   scene->particleSystem->initParticleToGrid(particleCount);
18   scene->grid->InitForce(colliderCount);
19   scene->grid->initGridToParticles(particleCount);
20   scene->particleSystem->initParticleStep(particleCount,
       colliderCount);
21
22   /*
23    *  INITIAL COMPUTATIONS
24    */
25   scene->grid->resetGrid();
26   scene->grid->calculateGridMass();
27   scene->grid->calculateParticleVolumeAndDensity();
```

Der nächste und letzte wichtige Schritt ist die Hauptschleife (Quellcode6) des Programms nach Glenn Fiedler[41]. Diese ist so konzipiert, dass während eines Schritts der Rendering-Engine dank eines Akkumulators mehrere Schritte der Physik-Engine ausgeführt werden können, ohne abhängig von der Bildrate der Rendering-Engine zu sein.
Die Physik-Engine wird, wie in Zeile 9 zu sehen ist, mit konstanter Delta-Zeit ausgeführt. Es werden so lange Physik-Schritte ausgeführt, bis die akkumulierte Zeit aufgebraucht ist, dann wird ein Frame gerendert und die Schleife beginnt wieder von vorne. Durch die Nutzung des Akkumulators und die konstante Zeit, mit der die Physik-Engine vorangetrieben wird, ist das Ergebnis der Simulation unabhängig von der Bildrate. Auf verschiedenen Rechnern sollte die Simulation damit gleich ablaufen.
Wichtig hierbei ist, dass die Simulation in einer festen Zeit abläuft, die nicht der realen Zeit entspricht.

[41] `https://gafferongames.com/post/fix_your_timestep/`, Einsicht 11.04.2019

In jedem Schritt wird die Simulation um eine Millisekunde vorangetrieben. Das bedeutet, dass nach eintausend Schritten eine Sekunde der realen Welt vergangen ist.

Quellcode 6: Hauptschleife

```
1  while (!glfwWindowShouldClose(window)) {
2      double newTime = glfwGetTime();
3      double frameTime = newTime - currentTime;
4      currentTime = newTime;
5      accumulator += frameTime;
6
7
8      while (accumulator >= DT) {
9        physicsEngine->simulationStep(PHYSIC_DT);
10       accumulator -= DT;
11     }
12
13     renderingEngine->scene->updateCamera(frameTime);
14
15     //Render Step
16     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
17     renderingEngine->render();
18
19     glfwSwapBuffers(window);
20     glfwPollEvents();
21   }
```

Innerhalb der Hauptschleife wechseln sich also Ausführungen der Physik-Engine und der Rendering-Engine ab, bis die Anwendung durch den Nutzer abgebrochen wird.
Die Berechnung ist also in einem sich wiederholendem Schema aufgebaut, dessen Grundlage in Kapitel 3.4 beschriben ist. Der Ablauf ist im Quellcode 7 zu erkennen.

Quellcode 7: Simulationsschritt der Physik-Engine

```
1  void Physics::simulationStep(double time)
2  {
3    scene->collisionSystem->colliderStep(time);
4    scene->collisionSystem->UpdateRender(time);
5    scene->grid->resetGrid();
6    scene->particleSystem->particleToGrid();
7    scene->grid->calculateForce(time);
8    scene->grid->gridToParticles();
9    scene->particleSystem->particleStep(time);
10
11 }
```

Nach der initialen Berechnung, die in den Zeilen 25-27 in Quellcode 5 gezeigt wurde, wird in jedem Simulationsschritt die in Quellcode 7 gezeigte Methode aufgerufen. Sie enthält die Aufrufe aller Compute-Shader, die für die Simulation sorgen.

In Zeile 3 wird der Compute-Shader aufgerufen, der die Kollisionsobjekte aktualisiert. Hier wird beispielsweise die neue Position einer Kugel berechnet, die sich durch die Szene bewegt. In der darauffolgenden Zeile werden die neuen Positionen für das Rendering aktualisiert. Es handelt sich somit nicht um einen Teil der Simulation.
Im nächsten Schritt in Zeile 5 wird ein Compute-Shader ausgeführt, der dafür sorgt, dass die Buffer des Gitters zurückgesetzt werden. Die Werte in diesen Buffern werden im Laufe der Simulation immer weiter aufsummiert. Folglich müssen sie in jedem Simulationsschritt zurückgesetzt werden, um fehlerfrei zu funktionieren.
In Zeile 6 wird ein Compute-Shader angewiesen, die Partikelgeschwindigkeit auf das Gitter zu gewichten und die Gitterkräfte zu berechnen. Der nächste aufgerufene Compute-Shader in Zeile 7 aktualisiert die Gittergeschwindigkeiten, in dem sie mit der Masse normalisiert werden. Zudem wird die erste Kollisionsabfrage durchgeführt. Der Rücktransfer auf die Partikel findet im Compute-Shader aus Zeile 8 statt. In ihm wird die neue Geschwindigkeit der Partikel anhand der aus dem Gitter kommenden aktualisierten Geschwindigkeit berechnet. Zudem wird der Geschwindigkeitsgradient berechnet. Der letzte Compute-Shader aus Zeile 9 führt nun die Aktualisierung der Partikel aus. Zunächst werden die neuen Deformationsgradienten berechnet. Dann erfolgt der zweite Kollisionstest. Die Partikelgeschwindigkeit wird aktualisiert und mit ihr dann auch die Partikelposition. Zusätzlich werden die Buffer für die aktualisierte Geschwindigkeit und den Geschwindigkeitsgradienten zurückgesetzt, um für die nächste Berechnung bereit zu sein. Wie einzelne Berechnungen innerhalb der Compute-Shader detailliert funktionieren wird in Kapitel 5.6 aufgezeigt.

5.3 Partikel

Dieses Kapitel beschreibt das Hauptelement, welches es zu simulieren gilt: die Schneepartikel.
Zur Simulation wird ein Partikelsystem erstellt, das aus vielen einzelnen Partikeln besteht. Diese haben ,basierend auf Kapitel 5.6, die folgenden Eigenschaften: die Position x_p, die Masse m_p die Geschwindigkeit v_p und abschließend der Deformationsgradient. Dieser besteht aus den zwei Hälften F_{Ep} und F_{Pp}.
Die Schneepartikel werden auf CPU-Seite mit initialen Werten befüllt. Dies ist in Quellcode 8 zu sehen.

Parameter	Buffernotation	Form
$\boldsymbol{x_p}$ und m_p	ppm	vec4(x_p, y_p, z_p, m_p)
v_p und ρ_{p0}	pv	ivec4($v_{px}, v_{py}, v_{pz}, \rho_{p0}$)
F_{Ep}	pFE	mat4(F_{Epxx}, F_{Epyx}, F_{Epzx}, 0.0, F_{Epxy}, F_{Epyy}, F_{Epzy}, 0.0, F_{Epxz}, F_{Epyz}, F_{Epzz}, 0.0, 0.0, 0.0, 0.0, 0.0)
F_{Pp}	pFP	mat4(F_{Ppxx}, F_{Ppyx}, F_{Ppzx}, 0.0, F_{Ppxy}, F_{Ppyy}, F_{Ppzy}, 0.0, F_{Ppxz}, F_{Ppyz}, F_{Ppzz}, 0.0, 0.0, 0.0, 0.0, 0.0)
v_p^{n+1}	pvn	ivec4(v_{px}^{n+1}, v_{py}^{n+1},v_{pz}^{n+1}, 0.0)
Δv_p^{n+1}	pdvp0, pdvp1 und pdvp2	ivec4(Δv_{pxx}^{n+1} ,Δv_{pyx}^{n+1} , Δv_{pzx}^{n+1}, 0.0) ivec4(Δv_{pxy}^{n+1} ,Δv_{pyy}^{n+1} , Δv_{pzy}^{n+1}, 0.0) ivec4(Δv_{pxz}^{n+1} ,Δv_{pyz}^{n+1} , Δv_{pzz}^{n+1}, 0.0)

Tabelle 3: Datentypen der Partikelparameter innerhalb der Buffer im Shader

Quellcode 8: Partikel-Konstruktor auf CPU-Seite

```
1 Particle::Particle(glm::vec3 pos, glm::ivec3 vel, float m, float
      vol)
2 {
3   position = pos;
4   velocity = vel;
5   mass = m;
6   volume = vol;
7   elastic = glm::mat3(1.f);
8   plastic = glm::mat3(1.f);
9 }
```

Im Laufe der Berechnung auf der GPU, werden sie allerdings neu berechnet und aktualisiert. Zudem werden in der Berechnung nach Kapitel 3.4 auf der GPU weitere Variablen wie die Dichte ρ_p, der Geschwindigkeitsgradient Δv_p^{n+1} und ein Buffer für die Berechnung der aktualisierten Geschwindigkeiten v_p^{n+1} eingeführt. Alle Eigenschaften werden über Variablen in Buffern gespeichert. Die Entstehung eines solchen Buffers ist in Kapitel 4.1 zu sehen.
Aufgrund des Aufbaus können zur Laufzeit der Simulation keine neuen Partikel zum System hinzugefügt werden.

In Tabelle 3 sind die Datentypen der Partikelparameter innerhalb der Buffer im Shader zu sehen. In der ersten Spalte sind die physikalischen Parameter zu sehen, in der zweiten Spalte sind die Notationen der Buffer in den Shadern aufgelistet. Es handelt sich um die Variablennamen der Buffer. So kann beispielsweise über die Befehle aus Quellcode 9 auf Buffer zugegriffen werden.

Quellcode 9: Zugriff auf Buffer

```
1 vec4 particle = ppm[pIndex];
2 vec4 particleVelocity = pv[pIndex];
```

Hierbei ist pIndex gegeben als Partikelindex.

5.4 Gitter

Dieses Kapitel beschreibt das kartesische Gitter, das in der Material-Point-Method zum Einsatz kommt.
Das Gitter verfügt, wie bereits in Abschnitt 3.4 gezeigt, über Eigenschaften wie die Position im Raum, den Abstand h zwischen den einzelnen Gitterknoten und der Dimension des gesamten Gitters. Diese sind konstant und können nicht während der Simulation geändert werden. Jeder einzelne Gitterknoten hat hierbei weitere Eigenschaften wie die Position x_i, die gewichtete Masse m_i, die Geschwindigkeit v_i, die Kraft f_i und die aktualisierte Geschwindigkeit v_i^{n+1}.
Im Konstruktor des Gitters auf CPU-Seite werden die Konstanten wie die Position, Dimension und der Zellenabstand des Gitters gesetzt. Dann wird der in Kapitel 5.2beschriebene Kontrollwürfel generiert.

Quellcode 10: Gitterknoten-Konstruktor auf CPU-Seite

```
1 GridNode() {
2     position = glm::vec3(0.f, 0.f, 0.f);
3     mass = 0;
4     velocity = glm::vec3(0.f, 0.f, 0.f);
5     velocityChange = glm::vec3(0.f, 0.f, 0.f);
6     forceElastic = glm::mat3(0);
7     forcePlastic = glm::mat3(0);
8   };
```

Wie in Quellcode 10 zu sehen ist, wird im Konstruktor der Gitterknoten auf CPU-Seite kein Wert tatsächlich gesetzt. Diese werden erst zur Laufzeit der Simulation mit den Daten der Partikel gefüllt und berechnet.
Auch das Gitter verfügt über Bufferobjekte, die für die Simulation notwendig sind. Tabelle 4 zeigt diese Parameter.

Parameter	Buffernotation	Form
$\boldsymbol{x_i}$	gp	vec4($x_i, y_i, z_i, 0.0$)
v_i und m_i	gv	ivec4($v_{ix}, v_{iy}, v_{iz}, m_i$)
f_i	gf	ivec4($f_{ix}, f_{iy}, f_{iz}, 0.0$)
v_i^{n+1}	gvn	ivec4(v_{ix}^{n+1}, v_{iy}^{n+1},$v_{iz}^{n+1}, 0.0$)

Tabelle 4: Datentypen der Gitterparameter innerhalb der Buffer im Shader

5.5 Kollision

Dieses Kapitel beschäftigt sich mit der Kollision von Schnee mit Rigid-Bodys. Auch die Kollision wird auf der GPU simuliert. Momentan unterstützt die Anwendung nur Kugeln und Ebenen.
Kollisionsobjekte haben diverse Eigenschaften: die Position x_c, der Typ typ_c von Objekt, also beispielsweise Kugel ($typ_c = 1$= oder Ebene ($typ_c = 0$), die Geschwindigkeit v_c, die Normale n_c und der Reibungskoeffizient μ. Dieser ist vom Material des Kollisionsobjekts abhängig. Eine Übersicht über die Parameter innerhalb der Shader gibt 5.

Parameter	Buffernotation	Form
$\boldsymbol{x_c}$	cpos	vec4(x_c, y_c, z_c, typ_c)
v_c	cv	vec4($v_{cx}, v_{cy}, v_{cz}, 0.0$)
n_c und μ	cn	vec4($n_{cx}, n_{cy}, n_{cz}, \mu$)

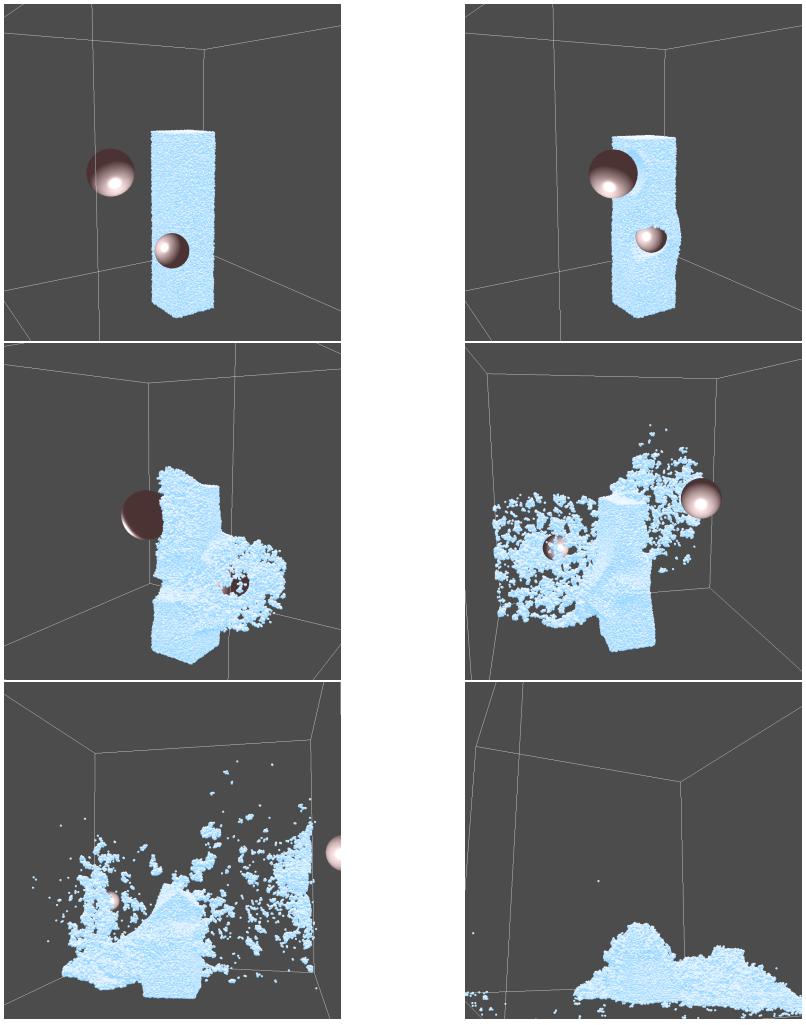
Tabelle 5: Datentypen der Kollisionsobjektparameter innerhalb der Buffer

Quellcode 11: Kollisionsabfragen auf der GPU

```
1  bool collisionTestHalfPlane(const vec3 particlePos,const int i){
2    return dot((particlePos-colliderPos[i].xyz),cn[i].xyz) <= 0;
3  }
4  bool collisionTestSphere(const vec3 particlePos,const int i, inout
       vec3 n){
5    float radius = cv[i].w;
6    n = normalize(particlePos-colliderPos[i].xyz);
7    return length(particlePos-colliderPos[i].xyz) < radius;
8  }
9
10 bool collisionTest(const vec3 particlePos,const int i, inout vec3
     n){
11   return (uint( colliderPos[i].w) ==0)? collisionTestHalfPlane(
       particlePos,i)
12     :(uint(colliderPos[i].w) ==1)?collisionTestSphere(particlePos,
       i,n)
13     :false;
14 }
```

Die Kollisionsabfragen für Kugeln und Ebenen sind relativ simpel, wie man in 11 sieht. Die Kollisionstest-Methode „collisionTest “entscheidet je nach Typ des Kollisionsobjekt, welche der beiden anderen Funktionen aufgerufen wird. Zur Abfrage einer Kollision wird bei einer Ebene deren Normale genutzt. Bei einer Kugel wird neben der Normalen auch der Radius der Kugel betrachtet.
Die folgende Abbildung zeigt die Kollision eines Schneeturms mit zwei Kugeln. Der weggeschobene Schnee kollidiert zudem mit den Wänden der Kontrollwürfel.

Abbildung 6: Kollision eines Turms aus Schnee mit zwei Kugeln



Die eigentliche Berechnung der Kollision findet in Quellcode 12 statt. Wir sehen eine for-Schleife über die Anzahl von Kollisionsobjekten der Szene.
In Zeile 5 wird mit den in Quellcode 11 gezeigten Methoden überprüft, ob eine Kollision vorliegt.
Über die if-Abfrage in Zeile 9 wird geprüft, ob sich das Objekt und der Partikel voneinander entfernen. Ist dies der Fall, entsteht keine Kollision. Für den anderen Fall wird nun der tangentiale Anteil der Geschwindigkeit berechnet. Über diese kann getestet werden (Zeile 11-16), ob eine Kollision entsteht und wie viel Reibung dabei entsteht (Zeile 18). In Zeile 20 wird dann die alte Geschwindigkeit unter Rücksichtname der Reibung aktualisiert. Falls keine Reibung entstanden ist, wird hier also ein Nullvektor addiert.

Quellcode 12: Kollisionsalgorithmus

```
1  for(int i = 0 ; i<NumColliders; i++){
2    vec3 p = cpos[i].xyz;
3    vec3 n = cn[i].xyz;
4    vec3 particlePos = ppm[pI].xyz;
5    if(collisionTest(particlePos,i,n)){
6      vec3 vco = cv[i].xyz;
7      vec3 vrel = vpn1 - vco;
8      float vn = dot(vrel,n);
9      if(vn<0.0f){
10       vec3 vt = vrel - n*vn;
11       float muvn = cn[i].w * vn;
12       vec3 vrelt=vt;
13       float lengthvt=length(vt);
14       if(lengthvt<= - muvn){
15         vrelt = vec3(0.0f);
16       }
17       else{
18         vrelt+= muvn*vt/(lengthvt);
19       }
20       vpn1 = vrelt + vco;
21     }
22   }
23 }
```

Wie in Kapitel 3.4 beschrieben, wird der Kollisionstest jeweils ein Mal auf der Gitter-Seite und ein mal auf der Partikel-Seite berechnet. Der Zeitpunkt der Kollisionsabfrage ergibt sich, sobald die aktualisierten Geschwindigkeiten auf dem Gitter oder im zweiten Fall von den Partikeln berechnet wurden.

5.6 Material-Point-Method

In diesem Kapitel wird die Implementierung der Material-Point-Method auf der GPU genauer beschrieben. Hierfür wird zunächst auf den glDispatchCompute()-Befehl eingegangen. Dann werden Teile der Berechnung in den Compute-Shadern gezeigt und erläutert.
Der glDispatchCompute()-Befehl ist in Kapitel 4.2 bereits beschrieben. In dieser Simulation gibt es drei Arten von Berechnungen: jene die auf Partikeln basieren, jene die auf Gitterknoten basieren und jene die auf dem Gitter basieren.
Dementsprechend müssen geeignete Werte für die lokale Größe innerhalb des Shaders und die Anzahl an Work-Groups gefunden werden. Die lokale Größe, die unabhängig von der Anzahl von Work-Groups ist, wurde mit 1024 Threads für x, und jeweils einem für y und z definiert.
Die Wahl der Anzahl der Work-Groups und deren Verteilung hängt davon ab, welche Art von Berechnung durchgeführt werden soll.

Für alle Berechnungen die auf Partikeln basieren, kann eine einfache Größe festgelegt werden, da wir für jeden Partikel genau eine Berechnung ausführen wollen. Es ergibt sich daher der Aufruf:

```
glDispatchCompute( countParticles / THREAD_COUNT +1, 1, 1);
```

Dieser Aufruf wird beispielsweise bei dem letzten Compute-Shader des Simulationsschritts eingesetzt. Dieser aktualisiert die Partikelgeschwindigkeit und -Position (siehe Kapitel 3.4). Teile dieses Shaders sind in Quellcode REF zu sehen. Dieser wird dann noch einmal genauer erklärt. Bei der Variablen „countParticles “handelt es sich um die Anzahl aller Partikel. Da diese Anzahl im Compute-Shader allerdings auf 1024 Threads verteilt wird, wird sie noch durch die lokale Größe (THREAD_COUNT) geteilt. Falls bei dieser Division ein Rest entsteht, der nicht behandelt werden würde, wird sie noch um eins inkrementiert. Die anderen beiden Dimensionen der Work-Group können bei 1 gelassen werden. Somit ist eine Berechnung pro Partikel garantiert.

Analog benötigt man für die Berechnungen, die auf den Gitterknoten basieren, eine ähnliche Anzahl an Work-Groups. Anstatt einer Variable, die die Anzahl der Partikel angibt, müssen wir nun die Anzahl aller Gitterknoten angeben. Diese lässt sich über die Multiplikation der drei Dimensionswerte berechnen:

```
glDispatchCompute(GRID_DIM_X * GRID_DIM_Y * GRID_DIM_Z /
    THREAD_COUNT + 1, 1, 1);
```

Die Größe der Work-Groups für diejenigen Berechnungen, die nur auf dem Gitter basieren, ist komplexer zu berechnen als die der anderen beiden. Das Ziel ist, für jeden Partikel eine Berechnung auszuführen, die allerdings Informationen von den Nachbarknoten des Gitters besitzt.
Aus diesem Grund kann die Größe der Work-Groups nicht auf die Anzahl der Gitterknoten oder Partikel gesetzt werden. Es ist effizienter und damit besser, wenn man die anderen Dimensionen der Work-Groups miteinbezieht. Die Work-Group-Größe ist auf drei Dimensionen aufgeteilt. Bis jetzt nutzten wir nur die erste Dimension. Nun kann aber die Indexierung des Nachbarn mit Hilfe der zweiten Dimension erreicht werden. Wenn beispielsweise 27 Nachbarn untersucht werden sollen, so kann der Aufruf so aussehen:

```
glDispatchCompute(countParticles / THREAD_COUNT + 1,
    GRID_PARTICLE_NEIGHBORHOOD, 1);
```

Die Konstante „GRID_PARTICLE_NEIGHBORHOOD“beschreibt die Anzahl der Gitterknoten in der Nachbarschaft des Partikels, mit denen Berechnungen durchgeführt werden sollen.
Compute-Shader, die Informationen zwischen Partikeln und Gitterknoten hin und her gewichten, nutzen Indexierungsfunktionen für eindeutige Zugriffe.

Im Folgenden Abschnitt werden Codebeispiele aus einem Compute-Shader gezeigt. Es handelt sich um den letzten Shader, der pro Simulationsschritt aufgerufen wird. Seine Funktion ist es, die neuen Deformationsgradienten, Partikelgeschwindigkeiten und -positionen zu berechnen Um übersichtlich zu bleiben, wurden einige Zeilen weggelassen oder gekürzt. Die Main-Funktion dieses Shader beginnt mit Wertezuweisungen von Bufferinhalten an Hilfsvariablen:

Quellcode 13: Zuweisungen im Compute-Shader

```
1  mat4 FEp4 = mat4(pFE[pI]);
2  mat3 FEp = mat3(FEp4);
3  mat4 FPp4 = mat4(pFP[pI]);
4  mat3 FPp = mat3(FPp4);
5  mat3 dvp =mat3(vec3(pdvp0[pI]),vec3(pdvp1[pI]),vec3(pdvp2[pI]));
```

Der Zugriff auf die Buffer erfolgt wie in Kapitel 4.1, 5.3, 5.4 und 5.5 beschrieben. Wie man sieht werden die beiden Deformationsgradienten und der Geschwindigkeitsgradient aus dem Buffer eingelesen.
Nun können die Deformationsgradienten aktualisiert werden:

Quellcode 14: Aktualisieren der Deformationsgradienten

```
6   mat3 FEpn = (mat3(1.0f) + dt * dvp)*FEp;
7   mat3 Fpn = (mat3(1.0f) + dt * dvp)* (FEp*FPp);
8   mat3 FPpn = FPp;
9
10  mat3 W =mat3(0.0f);
11  mat3 S =mat3(0.0f);
12  mat3 V =mat3(0.0f);
13  SVD(FEpn,W,S,V);
14
15  sclamp(S[0][0], 1.0f-critComp,1.0f+critStretch);
16  sclamp(S[1][1], 1.0f-critComp,1.0f+critStretch);
17  sclamp(S[2][2], 1.0f-critComp,1.0f+critStretch);
18
19  FEpn = W*S*transpose(V);
20
21  mat3 S_I = mat3(0.0f);
22
23  S_I[0][0]= 1.0f/S[0][0];
24  S_I[1][1]= 1.0f/S[1][1];
25  S_I[2][2]= 1.0f/S[2][2];
26  FPpn =V    * S_I * transpose(W) *Fpn;
```

In den Zeilen 6-8 werden die Deformationsgradienten nach (41) und (42) berechnet. Für die Berechnung des plastischen Teils des Deformationsgradienten wird in Zeile 13 die Singulärwertzerlegung des Elastischen Teils ausgeführt. In Zeile 26 wird dann der plastische Teil nach (43) berechnet.
In den Zeilen 15 bis 17 wird der Wertebereich der Diagonalmatrix beschränkt (siehe Kapitel 3.3)

Anschließend werden die aktualisierten Deformationsgradienten in die Buffer geschrieben:

Quellcode 15: Aktualisieren der Deformationsgradienten-SSBOs

```
27  pFE[gl_GlobalInvocationID.x] = mat4( FEpn);
28  pFP[gl_GlobalInvocationID.x] = mat4( FPpn);
```

Als nächstes wird die zweite Kollisionsabfrage gestartet. Diese wird an dieser Stelle übersprungen, da sie schon in Kapitel 5.5 genauer beschrieben wurde.

Quellcode 16: Aktualisieren und Zurücksetzen verschiedener SSBOs

```
30  pv[pI].xyz = ivec3(vpn1);
31
32  ppm[pI].xyz += dt *  vpn1;
33
34  pvn[pI].xyz = ivec3(0,0,0);
35  pdvp0[pI].xyz = ivec3(0,0,0);
36  pdvp1[pI].xyz = ivec3(0,0,0);
37  pdvp2[pI].xyz = ivec3(0,0,0);
```

Der Buffer, der die Partikelgeschwindigkeit enthält wird in Zeile 30 mit der neuen Geschwindigkeit vpn1 überschrieben. Diese wurde zuvor durch die Kollisionsabfrage verändert.
Der Buffer für die Partikelposition und -masse wird anschließend in Zeile 32 aktualisiert. Hierfür wird die Geschwindigkeit pro Zeit drauf addiert.

In den Zeilen 34-37 werden die vier Buffer zurückgesetzt, die im nächsten Shader-Durchlauf benötigt werden (siehe Kapitel 5.2).

5.7 Rendering

Dieses Kapitel befasst sich mit der Rendering. In der Applikation gibt es drei Elemente die, gerendert werden können.
Als erstes wäre da der Kollisionswürfel um die Szene herum. Dieser wird im wireframe[42]-Modus auf den grauen Hintergrund gezeichnet.
Andere, bewegliche Kollisionsobjekte wie Kugeln, die durch die Szene schießen, werden über einen simplen Phong-Shader beleuchtet.
Das Rendering der Schneepartikel ist von Meyer [Mey15] übernommen, da der Fokus eher auf der Simulation lag und diese Form des Renderings simpel gehalten wurde und visuell trotzdem ansprechend ist. Einzelne Schneepartikel werden einzeln als große Punkte gerendert. Diese Punkte wurden mit einem bläulichen Farbverlauf versehen, sodass als Gesamteindruck eine simple Form von Schnee zu erkennen ist, siehe Abbildung 7.

[42] wireframe: Drahtgitter

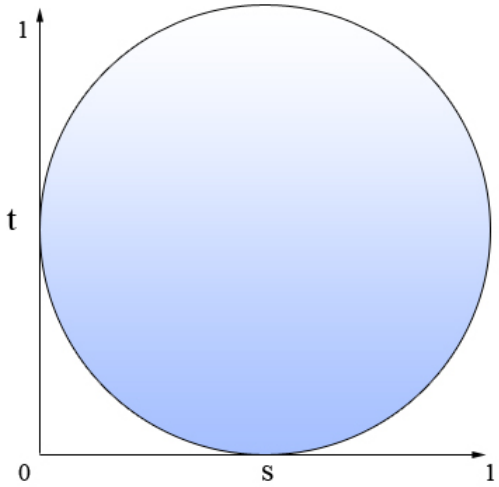


Abbildung 7: Farbverlauf von Meyer [Mey15]

6 Evaluation

Dieses Kapitel bewertet die Ergebnisse dieser Arbeit. Da die Simulationsergebnisse von der genutzten Hardware abhängig sind, wird diese in Kapitel 6.1 beschrieben.
Im nächsten Abschnitt wird festgestellt, wie leistungsfähig die Simulation ist. Hierfür werden Szenen mit verschiedenen Parametern für Partikel- oder Gitterknotenanzahl ausgewählt.
In Kapitel 6.3 wird die Qualität der Simulation bewertet. Hierfür wird zum einen die visuelle Qualität untersucht, zum anderen wird beschrieben, inwieweit die in Kapitel 3.3 erwähnten Eigenschaften von Schnee realisiert worden sind.

6.1 Hardware

Hier wird kurz beschrieben, mit welcher Hardware die Simulation ausgeführt wurde. Alle Bewertungen, die innerhalb dieser Arbeit entstehen, basieren auf den Ergebnissen der Simulation auf einer Nvidia Geforce GTX 1070- Grafikkarte mit 8GB RAM.
Die Anwendung wird aber grundsätzlich von allen Grafikkarten unterstützt, die Compute-Shader unterstützen. Diese wurden in OpenGL-Version 4.3 eingeführt.

Für Nvidea Grafikkarten gibt es eine Erweiterung, welche atomare Operationen auf Floating-Point[43]-Werten ermöglicht[44]. Damit die Simulation unabhängig vom Hersteller der Grafikkarte ist, wurden die standardmäßig vorhandenen, atomaren Operationen auf Integern[45] genutzt. Daher wurden einige Werte innerhalb der Shader skaliert, dann berechnet und wieder zurückskaliert. Die Werte werden skaliert und dann auf Integer gerundet. Durch das Runden entsteht ein minimaler Fehler. Nach der Berechnung kann der Wert wieder zurück skaliert werden.

6.2 Performanz

Um eine aussagekräftige Beurteilung der Leistungsfähigkeit abzugeben, werden in diesem Abschnitt die durchschnittlichen FPS[46] unter verschiedenen Bedingungen verglichen.
Hierbei ist zu beachten, dass die Hauptschleife der Simulation, wie sie in 5.2 beschrieben ist, abgeändert wurde. Anstatt die Aufrufe über einen Akkumulator zu regeln, wird in der Hauptschleife nun jeweils ein Simulations- und ein Renderingschritt vollzogen.
Da das Rendering der Schneesimulation simpel gehalten wurde, dauert die Berechnung des Simulationsschritts signifikant länger als ein Renderingschritt. Daraus resultiert, dass die FPS hauptsächlich aussagen, wie viele Physiksimulationsschritte pro Sekunde ausgeführt werden können.
Da in der Hauptschleife für den Akkumulator bereits Zeitwerte wie die Frame-Time festgestellt werden, können diese genutzt werden, um die FPS zu berechnen.Die Frame-Time ist definiert als die Zeit, die benötigt wurde, um ein Bild zu berechnen. Zur Berechnung der FPS wird einfach eine Sekunde durch die Frame-Time in Sekunden geteilt: $FPS = 1s/frameTime$.
Das Ergebnis der FPS-Berechnung jedes Schritts wird gesammelt, um am Ende der Simulation den Durchschnitt zu bilden.
Als Modell für den Leistungstest wird ein einfacher Schneeball verwendet, der durch die Schwerkraft auf den Boden fällt. Nach zwei Sekunden werden die durchschnittlichen FPS ausgelesen.

[43] Floating-Point: Datentyp für Gleitkommazahlen

[44] `https://www.khronos.org/registry/OpenGL/extensions/NV/NV_shader_atomic_float.txt`, Einsicht am 15.04.2019

[45] Datentyp für ganze Zahlen

[46] FPS: frames per second, Bilder pro Sekunde

In Diagramm 8 ist die Performanz in Abhängigkeit von der Anzahl von Partikeln zu sehen. Die Gitterdimension bleibt konstant bei $256 \times 256 \times 256$. Die minimale Anzahl der Partikel wurde auf 1024, also auf die lokale Größe innerhalb der Shader gesetzt. Der nächste Wert wurde bei 8192 Partikeln, also 2^{13} Partikeln, gemessen. Ab dann wurde die Anzahl der Partikel in jedem Schritt verdoppelt. Die Partikelanzahl variiert von 2^{10}, also 1024, bis 2^{20}, also $1,048,676$ Partikeln.
Da die Simulation bei manchen Partikelmengen sehr gute FPS-Werte (über 60 FPS) liefert, könnte man denken, die Simulation liefe in Echtzeit. Dies ist aber nicht der Fall.
Wichtig ist hierbei, dass die angegebene FPS die Anzahl der Simulationsschritte pro Sekunde angibt. Da jeder Schritt die Simulation jedoch nur um eine Millisekunde vorantreibt, werden in einer realen Sekunde bei 151 FPS tatsächlich nur 151 Millisekunden simuliert.
Die Simulation läuft damit selbst bei einer sehr kleinen Anzahl von Partikeln nicht in Echtzeit.

Die Performanz der Simulation ist abhängig von der Anzahl der Gitterknoten und kann in Diagramm 9 gesehen werden.
Hier wurde bei gleichbleibender Anzahl von Partikeln, nämlich $2^{16} = 65,536$, die Anzahl der Gitterknoten variiert. Der Wertebereich besteht aus den folgenden vier Werten: $64^3 = 262,144$, $128^3 = 2,097,152$, $256^3 = 16,777,216$ und abschließend $512 \times 512 \times 256 = 67,108,864$.
Da auf der CPU-Seite eine Liste von Gitterknoten und deren Eigenschaften angelegt wird, waren 16GB RAM für die Dimension $512 \times 512 \times 512$ zu wenig. Deshalb wurde bei $512 \times 512 \times 256$ aufgehört.

In beiden Grafiken ist eindeutig zu sehen, dass die Performanz stark von der Anzahl der Partikel und der Gitterknoten abhängt. Dies liegt vor allem daran, dass mit jedem Partikel oder Gitterknoten mehr Zugriffe auf SSBOs passieren müssen, was am meisten Zeit kostet. Solange alle Berechnungen auf der GPU ausgeführt werden, sind andere Faktoren zu vernachlässigen. Die Nutzung der atomaren Floating-Point-Operationen führte nur zu minmalen Unterschieden. Auch veränderte Buffergrößen haben keinerlei Auswirkung, da sich durch andere Größen die Anzahl der benötigten Zugriffe nicht verändert.

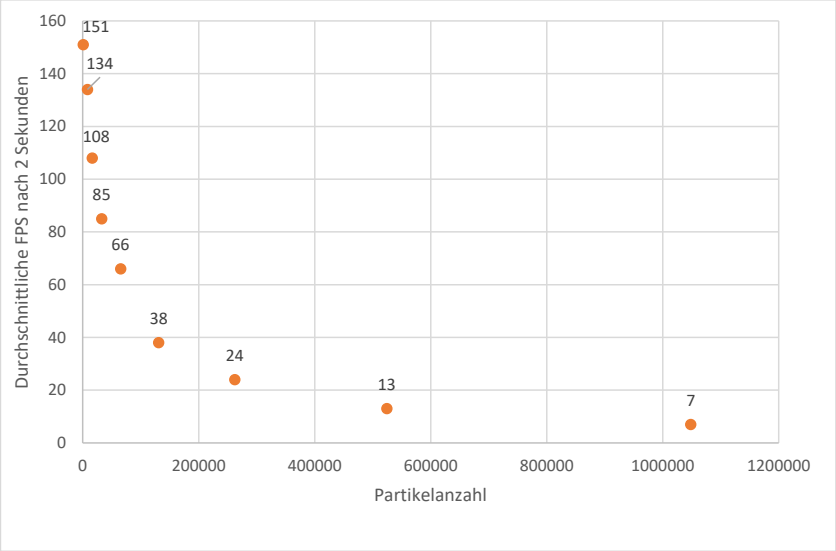


Abbildung 8: Leistung nach Partikelanzahl

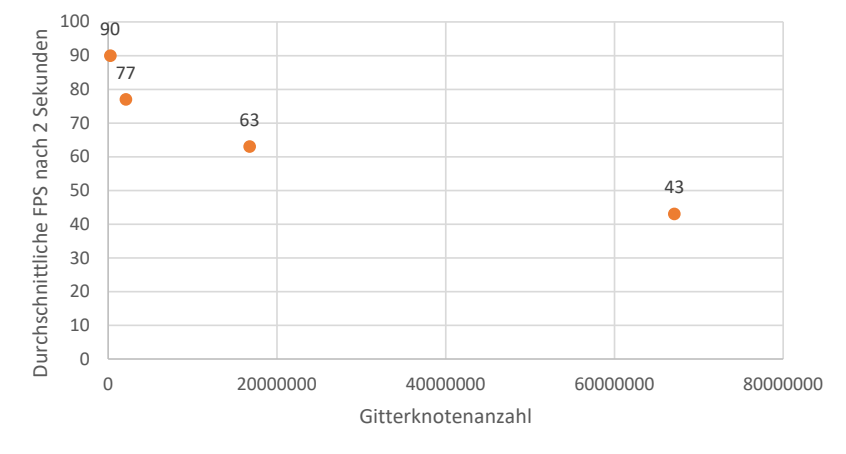


Abbildung 9: Leistung nach Gitterknotenanzahl

6.3 Qualität

In diesem Abschnitt wird die Qualität der Simulation bewertet. Hierfür wird untersucht, welche Leistungen die Simulation erbringen kann.
Zunächst kann gesagt werden, dass Schneepartikel dem Partikelsystem auf zwei Arten hinzugefügt werden können.
Zum einen gibt es die Option, einzelne Partikel direkt zu definieren und an das Partikelsystem anzuhängen. So können im Quellcode simple geometrische Formen wie Kugeln oder Ebenen aus Schnee erzeugt werden.
Zum anderen können auch komplexe Modelle in die Szene geladen werden, solange sie voxelisiert und richtig formatiert sind.
Beispielsweise ist in Abbildung 10 der voxelisierte Stanford-Hase[47] aus Schnee zu sehen. Er besteht aus $183,029$ Partikeln.

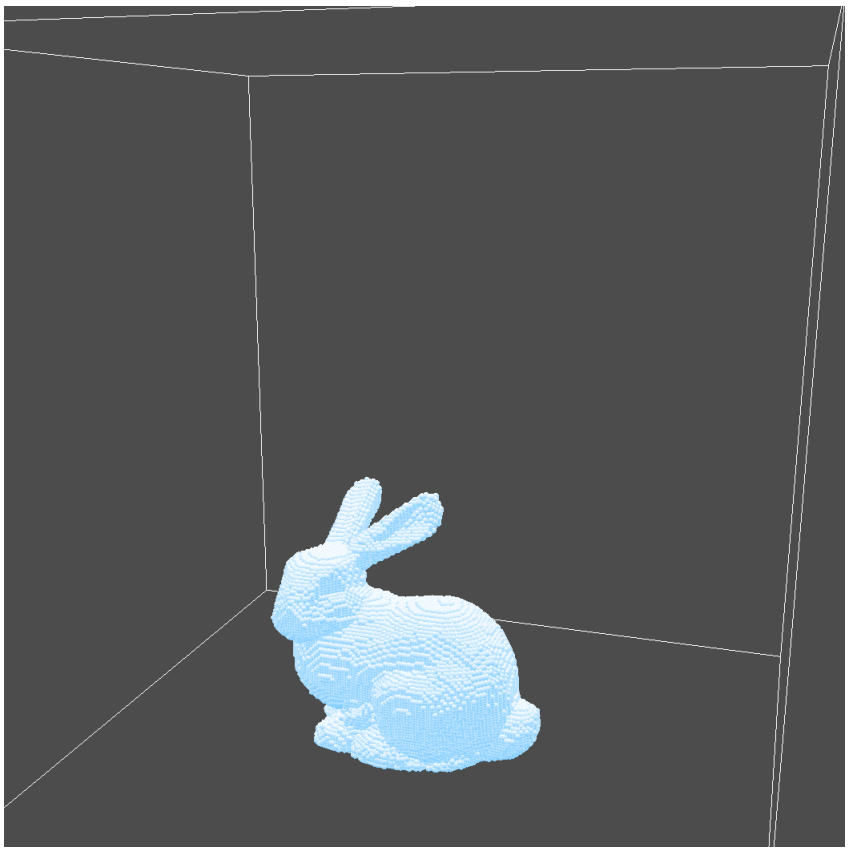


Abbildung 10: Stanford-Hase

[47] `http://graphics.stanford.edu/data/3Dscanrep/`, Einsicht am 15.04.2019

In Kapitel 3.2 wurden vier Eigenschaften von Schnee beschrieben, die mit dieser Anwendung simuliert werden sollen. Um zu testen, ob diese Eigenschaften tatsächlich simuliert werden können, werden Testszenen benötigt. Hauptsächlich wird eine Testszene genutzt, bei der ein Schneeball an einer Wand abprallt.
Anhand der verschiedenen Reaktion der Schneepartikel auf den Aufprall kann die Simulation der Dynamik bewertet werden.
In den folgenden Abbildungen 14, 15, 16, 17 und18 ist die Testszene zu sehen. Diese ist in 6 Standbilder unterteilt, die den Verlauf nachvollziehbar machen. Das erste Standbild zeigt die Ausgangsposition, die bei allen Testszenen gleich ist. Sie wird daher in Abbildung 12 nur einmal gezeigt.

Zunächst werden kurz die initialen Werte der Testparameter genannt. Der Youngsche Modul wurde auf 1×10^6 festgelegt. Dies ist ein bisschen mehr als von Stomakhin et al. empfohlen (Tabelle 2) wurde. Die Poissonzahl wurde als 0.3 definiert. Der Härte-Parameter ξ wurde mit 0.1 belegt. Es bleiben die kritische Kompression und die kritische Streckung. Für die Kompression wurde der Wert 2.5×10^{-2} gewählt, während die Streckung auf 7.5×10^{-3} gesetzt wurde.
In jeder dieser Abbildungen ist Bild (a) die Referenz mit den initial genannten Werten.
Für die Bilder aus Reihe (b) wurde der Youngsche-Modul auf 5×10^6 gesetzt. In Bildreihe (c) ist die Änderung der Poissonzahl zu erkennen. Sie erhielt den neuen Wert 0.41. Alle Bilder, die mit (d) markiert sind, zeigen den Einfluss des kritischen Streckungsparameters, während die Bilder (e) den Einfluss der kritischen Kompression zeigt. Die kritische Streckung wurde in der Bilderreihe (d) auf 1.75×10^{-3} gesetzt und damit verringert. Die kritische Kompression wird von 2.5×10^{-2} auf 5×10^{-2} verdoppelt.

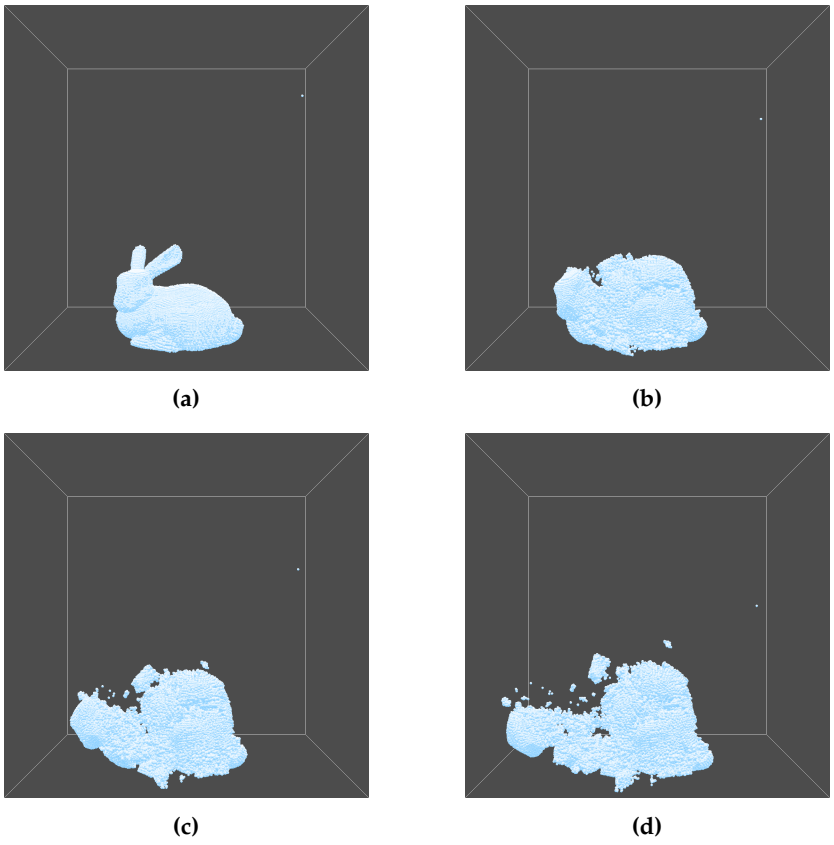
Erhaltung des Volumens Einfluss auf das Volumen des Schnees haben vor allem der Youngsche Modul und die Poissonzahl. Dies ist erkennbar wenn man (a) und (b) und (a) und (c) miteinander vergleicht. Besonders deutlich wird dies in den Abbildungen 15 und16, da hier das größere Volumen der beiden veränderten Varianten auffällt. Dies liegt daran, dass über den Youngschen Modul der Schneepartikel eine gewisse Resistenz gegenüber Kompression entwickelt wird.
Die Poissonzahl definierte, inwieweit Querspannungen Einfluss auf die Hauptspannung haben und damit, wie sehr sich das Material dehnen lässt. Ein Material mit einer hohen Poissonzahl kann besser gedehnt werden. Komprimierbare Materialien haben also eher niedrige Poissonzahlen.
Das Volumen eines gedehnten Materials wird erhalten.
Die Erhaltung des Volumens ist damit also über Parameter steuerbar.

Steifheit bzw. Elastizität Die Steifheit beschreibt den Widerstand eines Materials gegen Verformungen. Ihr Gegenstück die Elastizität sagt aus, wie gut sich ein Material nach Verformung wieder in den Ursprungszustand zurückversetzen kann.
Diese Eigenschaften werden von der kritischen Kompression und -Streckung parametrisiert. In den Bilderreihen (d) und (e), in denen diese beiden Werte abgeändert wurden, ist im Vergleich zu (a) ein deutlicher Unterschied zu erkennen. Gerade in Abbildung 16 (e) ist der Schneeball fast wieder in Halbkugelform angelangt.
Die in (d) gezeigten Bilder zeigen einen sehr viel plastischeren Schnee, als in (a) zu sehen ist. In der Bilderreihe (e) ist der Schnee weitaus elastischer, als in der Bilderreihe (d) und (a). Dies liegt an der erhöhten kritischen Kompression.
Über diese beiden Parameter ist also die Elastizität kontrollierbar.

Plastizität Die Plastizität beschreibt die Fähigkeit des Materials zu irreversiblen Verformungen.
Diese ist besonders gut in (d) zu sehen, da hier die kritische Streckung verringert wurde. Von der ursprünglichen Form ist in (d) am wenigsten zu erkennen.
Dadurch wird weniger Kraft benötigt, um bleibende Deformationen zu hinterlassen. Der Härte-Parameter ξ spielt hierbei auch eine wichtige Rolle.
Ein Material, dem mehr Plastizität gegeben wird, verliert im Umkehrschluss Elastizität. Auch die Plastizität kann mit der Material-Point-Method simuliert werden.

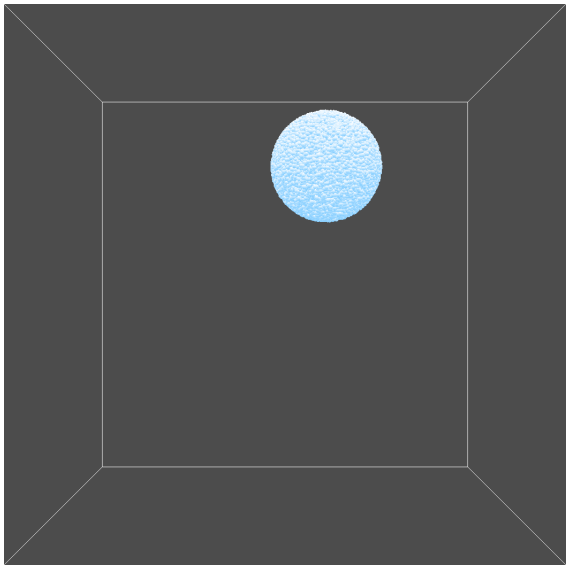
Bruch Der Bruch ist in den gezeigten Abbildungen nicht optimal zu erkennen. Am besten ist er wohl noch in dem dritten Standbild in Abbildung 16 (b) zu erkennen.
Um den Bruch noch deutlicher zu zeigen, wurde in Abbildung 11 eine zweite Testszene mit dem Stanford-Hasen kreiert. In den vier gezeigten Bildern ist der Verlauf eines Bruchs des Stanford-Hasen gezeigt. In Bild (a) ist der Hase noch nicht auf dem Boden aufgekommen. In Bild (b) beginnt sich der Kopf des Hasen vom Rest des Körpers zu trennen. Dieser fängt an, in Bild (c) komplett auseinander zu fallen, dennoch ist der Kopf als ganzes Stück noch erkennbar. In Abbildung (d) ist der Hase dann noch weiter zerfallen.
Beim Bruch gibt es keinen Parameter, der die Eigenschaft stärker kontrolliert als andere. Alle genannten Parameter sind daher wichtig, um das gewünschte Bruchverhalten zu erzeugen.
Der Youngsche Modul und die Poissonzahl haben Einflüsse auf die elastischen und plastischen Verformungen. Die beiden kritischen Parameter können gerade in Extremfällen starke Auswirkungen zeigen.

Abbildung 11: Bruch



(a) (b) (c) (d)

Abbildung 12: Ausgangspunkt der Testszene

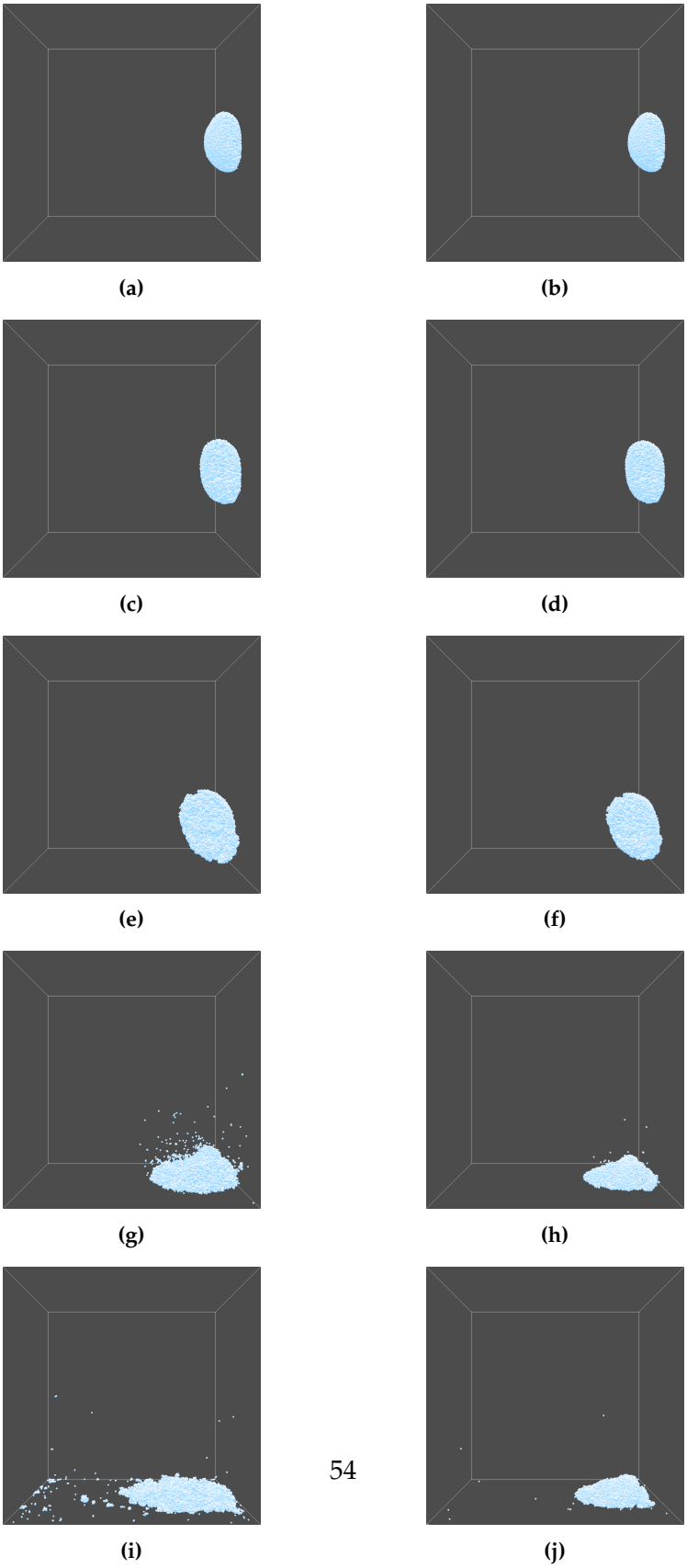


Das Bruchverhalten des Materials zu steuern ist also möglich, wobei eine geeignete Wertebelegung für die einzelnen Parameter gefunden werden muss.

Abschließend wird ein Vergleich zweier Schneearten vollzogen. In Abbildung 13 sieht man einen minimalen Unterschied zwischen trockenem Schnee in der linken Spalte und nassem Schnee in der rechten Spalte. Der nasse Schnee bleibt einen kleinen Teil länger an der Wand kleben als der trockene. Beim Herunterfallen ist der nasse Schnee gepackter als der trockene.
Den größten Unterschied erkennt man in den letzten beiden Zeilen, also beim Vergleich von (g) und (h) und (i) und (j). Der trockene Schnee zerspringt nach dem Aufprall. Einzelne Schneepartikel fliegen weiter weg.
Der nasse Schnee, der schon beim Fallen kompakt war, bleibt beinahe genauso liegen, wie er aufkam. Er verändert seine Form kaum.

Nasser Schnee hat im Vergleich zu dem trockenen eine hohe kritische Spannung und -Kompression. Der trockene Schnee hat dementsprechend gegenteilige Werte.
Damit wurden alle Eigenschaften, die es zu simulieren galt, realisiert. Die Simulation funktioniert.

Abbildung 13: Vergleich zwischen trockenem (a) und nassem (b) Schnee



(a) **(b)**

(c) **(d)**

(e) **(f)**

(g) **(h)**

(i) **(j)**

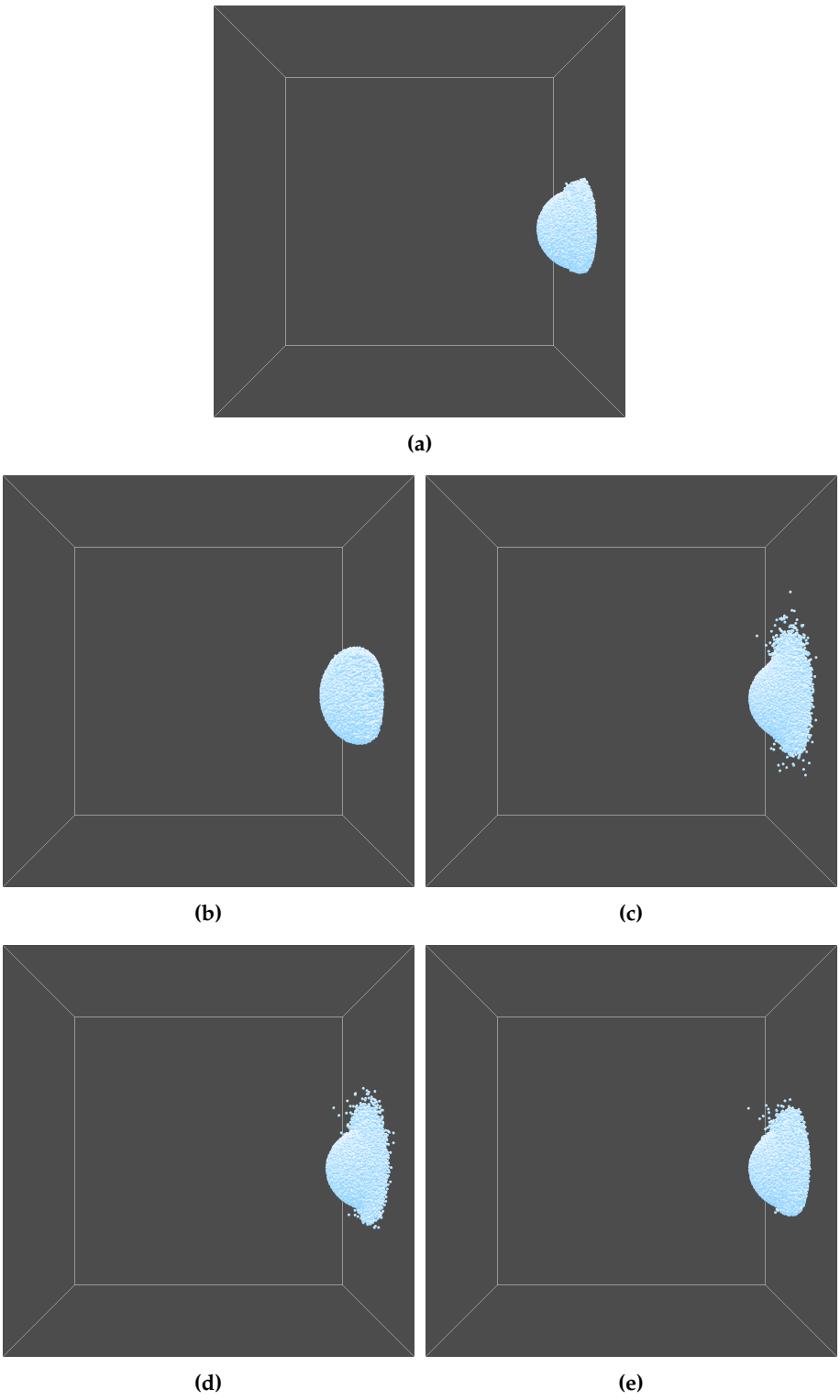
7 Fazit und Ausblick

Das Ziel dieser Arbeit war es, ein Verfahren zur Simulation von Schnee zu finden. Die Funktionsweise einer Physiksimulation sollte erarbeitet werden, um das gefundene Verfahren zu implementieren.
Die bedeutendste Arbeit zu dem Thema der dynamischen Simulation von Schnee ist die von Stomakhin et al. aus dem Jahr 2013 [SSC+13]. In ihr wird die Material-Point-Method vorgestellt, mit der sich das Verhalten dynamischer Materialien wie Schnee simulieren lassen. Hierfür wurde auch ein geeignetes Materialmodell benötigt.
Die Methode wurde mittels Compute-Shader erfolgreich auf der Grafikkarte implementiert.
Die Simulation an sich funktioniert. Damit ist das Hauptziel dieser Arbeit erreicht, obwohl noch Verbesserungen möglich sind.
Das Rendering der Partikel wurde bewusst simpel gehalten, um sich auf die Implementation der Physik in den Compute-Shadern zu konzentrieren. Da Schnee ein visuell sehr beeindruckendes Naturphänomen sein kann, ist dies sehr schade.
Stomakhin et al. nutzen beispielsweise einen volumetrischen Path-Tracer[48], um ihre Szenen zu rendern. Da Materialinformationen wie zum Beispiel die Partikeldichte vorliegen, kann ein fest gepackter Schnee anders gerendet werden, als ein trockener, loser Schnee. Die Materialinformation fließt also physikalisch ins Rendering ein.
Das Einlesen von Modelldaten kann auch noch verbessert werden. Momentan können Partikel manuell an das Partikelsystem angehangen werden, oder es wird ein voxelisiertes Modell in die Szene geladen. Dieses muss allerdings auf eine bestimmte Art formatiert sein, damit es korrekt geparst wird. Im Idealfall könnten nicht voxelisierte Modelle geladen werden, die dann von der Anwendung voxelisiert und in die Szene gebracht werden.
Außerdem fehlt es der Anwendung an Optimierung. Der momentane Aufbau garantiert die Funktion, ist dafür aber an manchen Stellen langsamer als er sein müsste. Das beste Beispiel hierfür ist das nicht vorhandene semi-implizite Update, das die Geschwindigkeiten akkurater berechnen kann.

Letztendlich hat diese Bachelorarbeit ihr Ziel dennoch erreicht. Mit Hilfe von Compute-Shadern wurde eine funktionierende Physiksimulation von Schnee auf der Grafikkarte implementiert.
Da die Material-Point-Method nicht nur auf Schneepartikel begrenzt ist, wird sie in Zukunft auch weiterhin relevant bleiben und sich weiter verbessern.

[48] Path-Tracer: Rendering-Methode die Lichtpfade verfolgt.

Abbildung 14: Erstes Standbild nach 901 Frames, 901ms



(a)

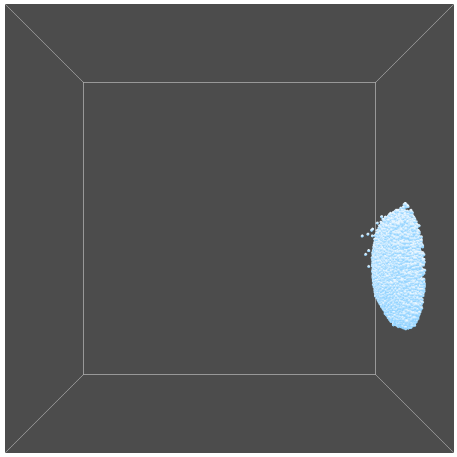
(b)

(c)

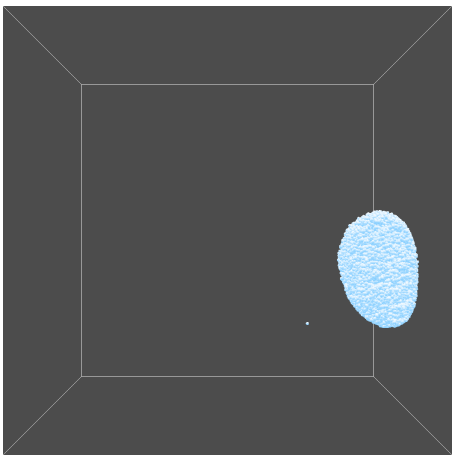
(d)

(e)

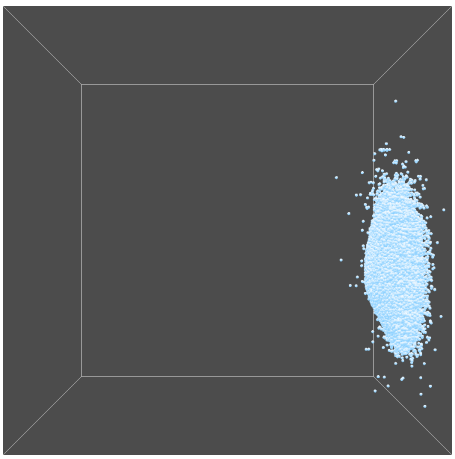
Abbildung 15: Zweites Standbild nach 1000 Frames, 1s



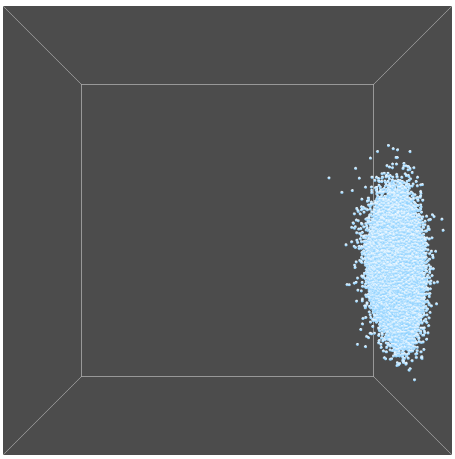
(a)



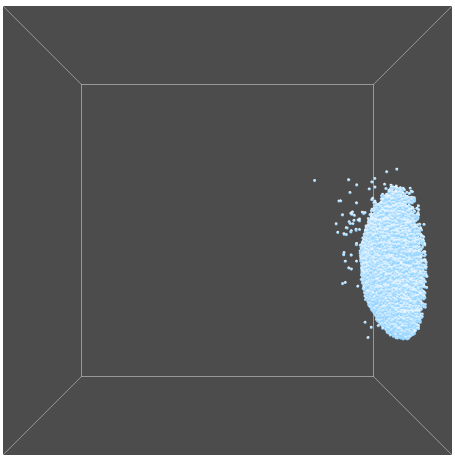
(b)



(c)

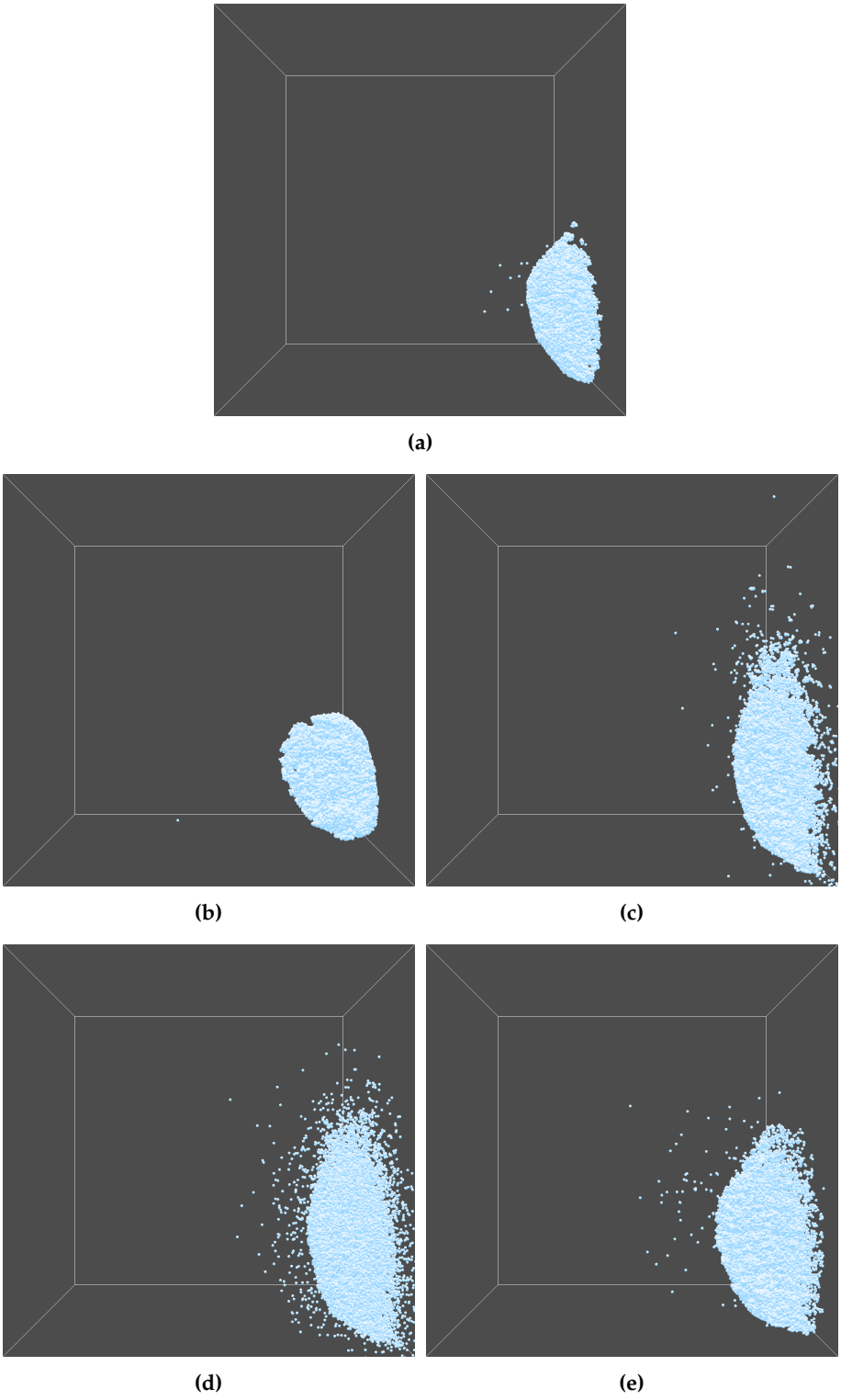


(d)



(e)

Abbildung 16: Drittes Standbild nach 1233 Frames, 1.233s



(a)

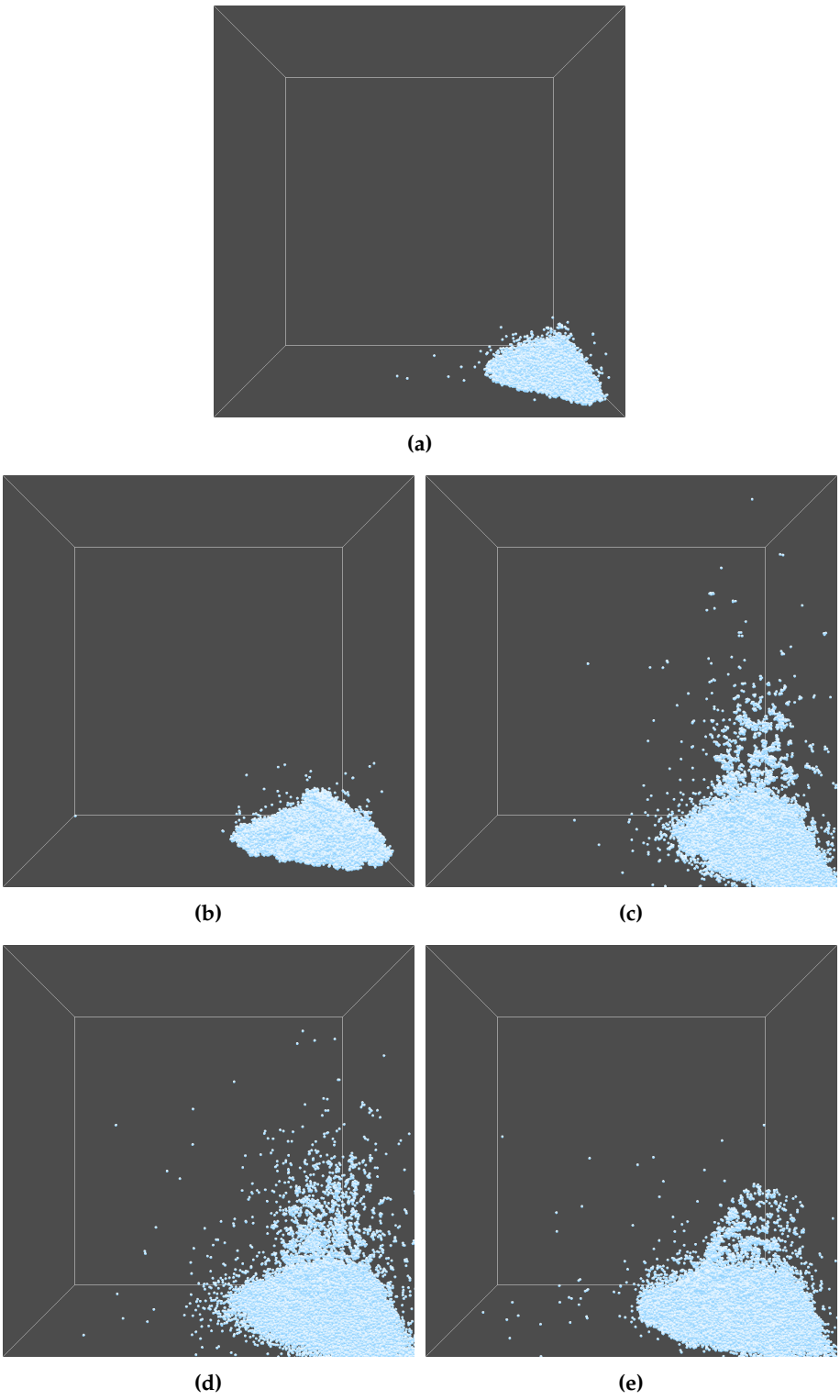
(b)

(c)

(d)

(e)

Abbildung 17: Viertes Standbild nach 1644 Frames, 1.644s



(a)

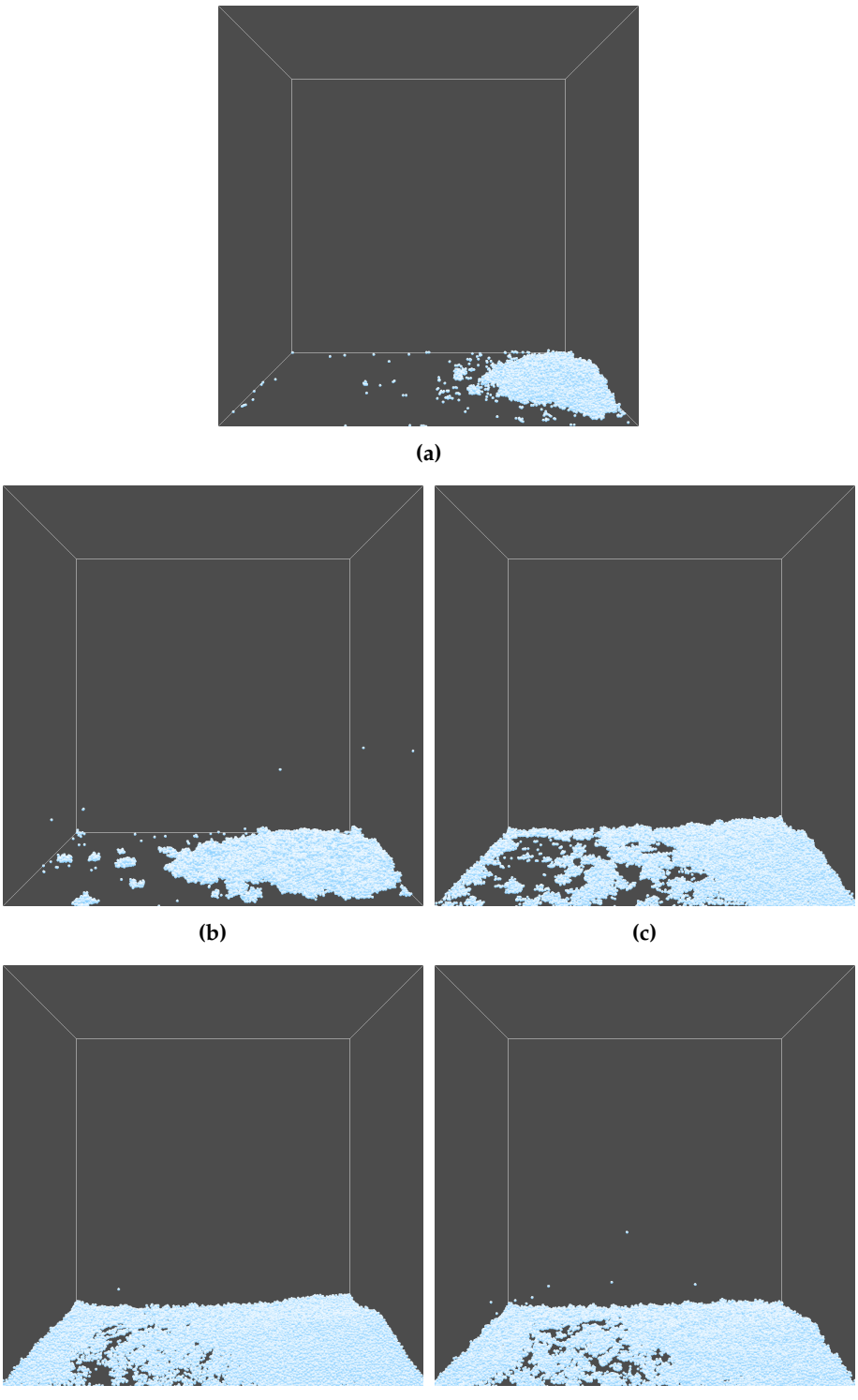
(b)

(c)

(d)

(e)

Abbildung 18: Fünftes Standbild nach 4054 Frames, 4.054s



(a) **(b)** **(c)** **(d)** **(e)**

8 Anhänge

Quellcodeverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[AO11] ALDUÁN, Iván ; OTADUY, Miguel A.: SPH granular flow with friction and cohesion. In: *Proceedings of the 2011 ACM SIGGRAPH/Eurographics symposium on computer animation* ACM, 2011, S. 25–32

[BYM05] BELL, Nathan ; YU, Yizhou ; MUCHA, Peter J.: Particle-based Simulation of Granular Materials. In: *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. New York, NY, USA : ACM, 2005 (SCA '05). – ISBN 1–59593–198–8, 77–86

[Fea00] FEARING, Paul: Computer Modelling of Fallen Snow. In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 2000 (SIGGRAPH '00). – ISBN 1–58113–208–5, 37–46

[FO02] FELDMAN, Bryan E. ; O'BRIEN, James F.: Modeling the Accumulation of Wind-driven Snow. In: *ACM SIGGRAPH 2002 Conference Abstracts and Applications*. New York, NY, USA : ACM, 2002 (SIGGRAPH '02). – ISBN 1–58113–525–4, 218–218

[Gra12] GRANGER, R.A.: *Fluid Mechanics*. Dover Publications, 2012 (Dover Books on Physics). `https://books.google.de/books?id=VWG8AQAAQBAJ`. – ISBN 9780486135052

[LHM95] LUCIANI, Annie ; HABIBI, Arash ; MANZOTTI, Emmanuel: A multi-scale physical model of granular materials. In: *Graphics interface'95*, 1995, S. 136–146

[Lib05] LIBBRECHT, Kenneth G.: The physics of snow crystals. In: *Reports on progress in physics* 68 (2005), Nr. 4, S. 855

[LLJ+11] LEVIN, David I. ; LITVEN, Joshua ; JONES, Garrett L. ; SUEDA, Shinjiro ; PAI, Dinesh K.: Eulerian solid simulation with contact. In: *ACM Transactions on Graphics (TOG)* 30 (2011), Nr. 4, S. 36

[McG] MCGINTY, Bob: *Continuum Mechanics*. http://www.continuummechanics.org/

[Mey15] MEYER, Fabian: *Simulation von Schnee*. 2015

[Mil96] MILENKOVIC, Victor J.: Position-based physics: simulating the motion of many highly interacting spheres and polyhedra. In: *SIGGRAPH* Bd. 96 Citeseer, 1996, S. 129–136

[MP89] MILLER, Gavin ; PEARCE, Andrew: Globular dynamics: A connected particle system for animating viscous fluids. In: *Computers & Graphics* 13 (1989), 12, S. 305–309. http://dx.doi.org/10.1016/0097-8493(89)90078-2. – DOI 10.1016/0097–8493(89)90078–2

[MSW+09] MCADAMS, Aleka ; SELLE, Andrew ; WARD, Kelly ; SIFAKIS, Eftychios ; TERAN, Joseph: Detail preserving continuum simulation of straight hair. In: *ACM Transactions on Graphics (TOG)* Bd. 28 ACM, 2009, S. 62

[NGL10] NARAIN, Rahul ; GOLAS, Abhinav ; LIN, Ming C.: Free-flowing granular materials with two-way solid coupling. In: *ACM Transactions on Graphics (TOG)* Bd. 29 ACM, 2010, S. 173

[NIDN97] NISHITA, Tomoyuki ; IWASAKI, Hiroshi ; DOBASHI, Yoshinori ; NAKAMAE, Eihachiro: A Modeling and Rendering Method for Snow by Using Metaballs. In: *Computer Graphics Forum* 16 (1997), Nr. 3, C357-C364. http://dx.doi.org/10.1111/1467-8659.00173. – DOI 10.1111/1467–8659.00173

[SHST12] STOMAKHIN, Alexey ; HOWES, Russell ; SCHROEDER, Craig ; TERAN, Joseph M.: Energetically consistent invertible elasticity. In: *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* Eurographics Association, 2012, S. 25–32

[SOH99] SUMNER, Robert W. ; O'BRIEN, James F. ; HODGINS, Jessica K.: Animating Sand, Mud, and Snow. In: *Computer Graphics Forum* 18 (1999), Nr. 1, 17-26. http://dx.doi.org/10.1111/1467-8659.00299. – DOI 10.1111/1467–8659.00299

[SSC+13] STOMAKHIN, Alexey ; SCHROEDER, Craig ; CHAI, Lawrence ; TERAN, Joseph ; SELLE, Andrew: A Material Point Method for Snow Simulation. In: *ACM Trans. Graph.* 32 (2013), Juli, Nr. 4, 102:1–102:10. http://dx.doi.org/10.1145/2461912.2461948. – DOI 10.1145/2461912.2461948. – ISSN 0730–0301

[SZS95] SULSKY, Deborah ; ZHOU, Shi-Jian ; SCHREYER, Howard L.: Application of a particle-in-cell method to solid mechanics. In: *Computer Physics Communications* 87 (1995), Nr. 1, 236 - 252. http://dx.doi.org/https://doi.org/10.1016/0010-4655(94)00170-7. – DOI https://doi.org/10.1016/0010–4655(94)00170–7. – ISSN 0010–4655. – Particle Simulation Methods

[TF88] TERZOPOULOS, Demetri ; FLEISCHER, Kurt: Modeling inelastic deformation: viscolelasticity, plasticity, fracture. In: *ACM Siggraph Computer Graphics* Bd. 22 ACM, 1988, S. 269–278

[ZB05] ZHU, Yongning ; BRIDSON, Robert: Animating Sand As a Fluid. In: *ACM Trans. Graph.* 24 (2005), Juli, Nr. 3, 965–972. http://dx.doi.org/10.1145/1073204.1073298. – DOI 10.1145/1073204.1073298. – ISSN 0730–0301

[ZY10] ZHU, Bo ; YANG, Xubo: Animating Sand as a Surface Flow. In: *Eurographics (Short Papers)*, 2010, S. 9–12