UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Simulation von Schnee

Bachelorarbeit

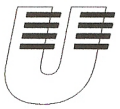
zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Fabian Meyer

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

Koblenz, im März 2015

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



UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Bachelorarbeit

Fabian Meyer

(Matr.-Nr. 211 200 407)

Thema: **Simulation von Schnee**

In keinem Bereich der Informatik hat sich die Hardware-Entwicklung so rasant entwickelt, wie im Bereich der Computergraphik. So ist es heute möglich, selbst komplexe Welten und physikalische Gegebenheiten quasi in Echtzeit zu simulieren und in hoher Qualität darzustellen. Eine Herausforderung dabei ist die Simulation von atmosphärischen Effekten und deren Darstellung, wobei in jüngster Zeit interessante Veröffentlichungen zur Simulation von Schnee publiziert wurden, die bislang noch nicht in Echtzeit umsetzbar sind.

Ziel dieser Arbeit ist es, einen Überblick über Verfahren zur Darstellung und Simulation von Schnee zu recherchieren und entweder ein Verfahren auf seine Möglichkeiten und Grenzen zu untersuchen, oder mehrere Verfahren miteinander zu vergleichen.

Schwerpunkte dieser Arbeit sind:

1. Recherche von Verfahren zur Simulation von Schnee.
2. Einarbeitung in die nötigen Grundlagen
3. Auswahl von einem oder mehreren Verfahren
4. Implementierung des/der Verfahren/s
5. Demonstration der Ergebnisse in ansprechender Qualität
6. Bewertung und Dokumentation der Ergebnisse

Koblenz, den 23.09.2014

F. Meyer

- Fabian Meyer -

S. Müller

- Prof. Dr. Stefan Müller-

Erklärung

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

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

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

Zusammenfassung

Physik-Simulationen erlauben die Erstellung dynamischer Szenen auf dem Rechner. Sie lassen die Computergrafik lebendig werden und finden unter anderem Anwendung in Film, Spiel und Ingenieurwesen. Durch GPGPU-Techniken kann diese Arbeit erstmals auf der Grafikkarte stattfinden. Die dynamische Simulation von Schnee ist ein Gebiet, das aufgrund seiner physikalischen Komplexität wenig erforscht ist. Die Materie-Punkt-Methode ist das erste Modell, das in der Lage ist die Dynamik und verschiedenen Arten von Schnee darzustellen.
Die hybride Nutzung von Lagrange-Partikeln und einem kartesischen Euler-Gitter ermöglichen das Lösen der partiellen Differentialgleichungen. Die Partikel werden dazu auf die Gitterknoten transformiert. Durch Anwendung der Finite-Elemente-Methode auf das Gitter können Gradienten zur Aktualisierung der Geschwindigkeit berechnet werden. Die Geschwindigkeiten werden dann auf die Partikel zurückgewichtet, um diese in der Simulation voranschreiten zu lassen. Gepaart mit einem spezifischen Materialmodell wird die dynamische Natur von Schnee erlangt. Diese schließt Kollision und Bruch mit ein.
Diese Bachelorarbeit verbindet die kürzlich erschienenen GPGPU-Techniken von OpenGL mit der Materie-Punkt-Methode, um die verschiedenen Schneearten dynamisch, visuell ansprechend und effizient zu simulieren.

Abstract

Physic simulations allow the creation of dynamic scenes on the computer. Computer generated images become lively and find use in movies, games and engineering applications. GPGPU techniques make use of the graphics card to simulate physics. The simulation of dynamic snow is still little researched. The Material Point Method is the first technique which is capable of showing the dynamics and characteristics of snow.
The hybrid use of Lagrangian particles and a regular cartesian grid enables solving of partial differential equations. Therefore particles are transformed to the grid. The grid velocities can be updated with the calculation of gradients in an FEM-manner (finite element method). Finally grid node velocities are weight back to the particles to move them across the scene. This method is coupled with a constitutive model to cover the dynamic nature of snow. This includes collisions and breaking.
This bachelor thesis connects the recent developments in GPGPU techniques of OpenGL with the Material Point Method to efficiently simulate visually compelling, dynamic snow scenes.

Inhaltsverzeichnis

1 Einleitung **1**

2 Related Work **3**

3 Grundlagen **5**
3.1 Kontinuumsmechanik 5
3.2 Entstehung und Kategorisierung von Schnee 11
3.3 Dynamik von Schnee 14
3.4 Materie-Punkt-Methode 16
3.5 Materialmodell 19
3.6 Compute-Shader 22

4 Implementation **27**
4.1 Eingebundene Bibliotheken und Programme 28
4.2 Partikel und Gitter 29
4.3 Materie-Punkt-Methode auf der GPU 31
4.4 Kollisionen mit Rigid-Bodies 34
4.5 Rendering 36
4.6 Ablaufdiagramm 38

5 Evaluation **40**
5.1 Kompatibilität 40
5.2 Performance 40
5.3 Visuelle Qualität 43

6 Fazit und Ausblick **50**

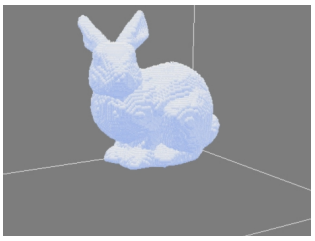
A Grafiken **51**

1 Einleitung

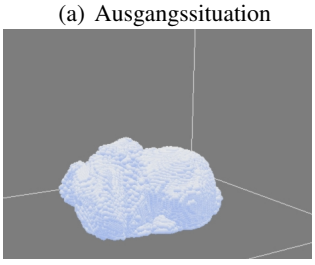
Die Simulation von Physik ist eines der Kernelemente der Computergrafik. Mit ihr ist es möglich die Naturerscheinungen der realen Welt am PC zu imitieren und modifizieren. Die Simulation hat Einzug in jegliche Bereiche gefunden. Das Ingenieurwesen simuliert kostengünstig Situationen, um Abschätzungen, Verbesserungen und höhere Qualität denn je liefern zu können. Dies tritt ein bevor der erste Prototyp gebaut ist. Die Naturwissenschaften simulieren Szenarien am Rechner, um Wissen zu vernetzen und hinzuzugewinnen oder Vorhersagen zu treffen. Designer werden von Simulationen unterstützt, um zu neuen Dimensionen des Erlebnisses durch Film und Spiel zu gelangen. Die Physik auf der GPU[1] mittels GPGPU-Techniken[2] zu berechnen, erweitert die vorstellbaren Möglichkeiten für das Simulieren verschiedenster Problemstellungen.

Schnee ist ein physikalisches Phänomen mit unglaublicher Vielseitigkeit, Dynamik und Schönheit: Sei es gepackter Schnee in der Form eines Schneemanns; aufgewirbelter Schnee durch ein Snowboard, ohne das dieser Sport nicht entstanden wäre; ein Schneesturm, der ein gesamtes Haus innerhalb von Tagen zu schneien kann; der Wurf und Aufprall eines Schneeballs, der je nach Art des Schnees förmlich zerplatzt oder an die Wand pappt; oder das leise Rieseln einzelner Schneeflocken auf ein Fenster.

Die Dynamik von Schnee ist wenig erforscht. Das liegt an verschiedenen Faktoren: Schnee ist ein Phänomen, das weder in allen Gebieten der Welt noch das gesamte Jahr über zu erleben ist. Die Simulation von Soliden, Flüssigkeiten und anderen viskoelastischen[3] Materialien ist schon in verschiedenen Arbeiten gelungen. Schnee hingegen verhält sich mal mehr wie ein solides Material, mal mehr wie eine Flüssigkeit. Dies macht die Simulation von Schnee zu einem komplexen Problem.



(a) Ausgangssituation



(b) Zur Zeit $t = 1.8s$.

Abbildung 1: *Ein simulierter Schneehase.*

Die Materie-Punkt-Methode ist die erste Methodik, die die Simulation verschiedener Schneearten lösen kann. Die Materie-Punkt-Methode ist ein "Teilchen in einer Zelle"(PIC[4])-Verfahren ursprünglich von [SZS95] entwickelt. Der Fakt, das die Materie-Punkt-Methode das erste Mal Anwendung in der Computergrafik gefunden hat, macht deutlich, dass die Suche nach neuen Wegen und Modellen für die Simulation von Physik nötig ist. Die Materie-Punkt-Methode kombiniert Lagrange-Partikel mit einem kartesischen Eulergitter. Dadurch lassen sich Partikel

[1] GPU: Graphical Processing Unit

[2] GPGPU: General Purpose Computation on Graphics Processing Unit

[3] Viskoelastizität: Teilweise flüssig, teilweise fest

[4] Particle-in-Cell: Teilchen in einer Zelle

durch ihre Nachbarn beeinflussen. Keine weitere Struktur, wie die Generierung eines Meshes, ist nötig. Die Materie-Punkt-Methode kümmert sich um Selbstkollisionen und Bruchszenarien, die in den verschiedenen Arten von Schnee auftreten können. Dazu wird die Materie-Punkt-Methode mit einem Materialmodell für Schnee verbunden. Dieses behandelt die weite Bandbreite der Schneearten; von trocken zu nass, von matschig zu eisig. Es ist aber nicht reduziert auf Schnee. Das Schmelzen von Objekten unter Hitze ließ sich bereits verwirklichen [SSJ+14]. [SSC+13]

In dieser Arbeit werden die vor kurzem erschienenen GPGPU-Techniken ausgenutzt, um mithilfe der Materie-Punkt-Methode Schnee effizient auf der Grafikkarte zu berechnen. Dies ermöglicht die Nutzung hoher Parallelisierung durch die zahlreichen GPU-Prozessoren. Die Anwendung lässt sich auf OpenGL 4.3 fähigen Grafikkarten ausführen, was jede aktuelle Grafikkarte leistet.

Kapitel 2 thematisiert die verwandten Arbeiten, die in der Computergrafik für das Modellieren und Simulieren von Schnee erschienen sind. Bevor dann in Kapitel 3 die Grundlagen für das Verständnis der Materie-Punkt-Methode und die Eigenarten von Schnee erläutert werden. Kapitel 4 zeigt die Implementation auf der GPU unter Nutzung von Compute-Shadern auf. In Kapitel 5 wird die Implementation und das Modell evaluiert. Einen abschließenden Überblick über die Arbeit gibt Kapitel 6 und zeigt einen Ausblick für die vorstellbaren Verbesserungen und Erweiterungen auf.

2 Related Work

Modellieren von Schnee mittels Geometrie: In der Computergrafik ist das Modellieren von Schneedecken in einer Szene ist adressiert worden. [NIDN97] nutzt Schneebälle (Metaballs) basierend auf Partikeln, um Schnee auf Oberflächen zu modellieren und mit globaler Beleuchtung zu rendern. [Fea00] bedeckt, durch das Schießen von Partikeln in den Himmel, Szenen mit Schnee und führt einen Stabilitätstest zur Verbesserung der Partikel-Platzierungen durch. [MGG+10] simuliert Winterszenerien durch Schneefall, Schmelzen und Gefrieren unter Einbezug der Temperatur. Das Nutzen von Height-Maps[5] für die Darstellung von deformierenden Oberflächen ist ein populäres Mittel in Spielen, wie Batman: Arkham Origins [6] [SOH99]. Zusätzlich nutzen sie Tesselation[7] zur Geometrieerzeugung, die das Ergebnis weiter verbessern. Dragon Age: Inquisition[8] erweitert dies durch Billboard-Texturen[9], die beim Auftreten aufwirbelnden Schnee darstellen .

Granulare Materie: Schnee wird in der Computergrafik als granulare Materie behandelt. Dies umfasst beispielsweise Sand, Erde, Puder und körniges Material. Jeder der Körner kann entweder einzeln betrachtet werden [MP89, ATO09, BYM05] oder als eine Menge von Körnern, die kontinuierlich approximiert wird. Ein Weg dies zu realisieren ist die Benutzung von geglätter Teilchen-Hydrodynamik (SPH) [LD09, AO11, IWT12]. Dazu wird das zu simulierende Objekt in zufällige Elemente unterteilt, um zwischen diesen Elementen Gradienten berechnen zu können. Eine Alternative, genauere Herangehensweise, ist die Nutzung eines Gitters für die Berechnung von Differentialgleichungen. [ZB05] nutzen inkompressible, implizite Flüssigkeitspartikel (FLIP[10]) zur Simulation von Sand. Das ursprüngliche FLIP zur Simulation von Flüssigkeiten ist veröffentlicht durch [BR86]. FLIP ist wie die Materie-Punkt-Methode eine „Teilchen in einer Zelle“(PIC)-Methode. Inkompressibles FLIP erzwingt die Inkompressibilität. Die Materie-Punkt-Methode ist eine generellere Technik, die die Nutzung von Materialmodellen ermöglicht. [SSC+13]

Materialmodell: Materialmodelle konstatieren, auf welche Weise ein Material auf Einflüsse wie Temperatur oder Kraft reagiert. Meist werden diese auf wenige Parameter reduziert. Dies entspricht nicht der Wirklichkeit. Das Modell bleibt dadurch kontrollierbar und überschaubar. Ein solches Modell fand erstmals Verwendung in der Simulation von plastischen[11] Verformungen, dazu zählen Brüche in Materialien, durch [TF88]. [OBH02] modellieren den dehnbaren Bruch von Materialien, wie Plastik und Metall. Das Material verformt sich erst elastisch[12] und dar-

[5] Height-Map: Textur für den Eindruck von zusätzlicher Geometrie bei der Beleuchtung

[6] Einsicht am 15.03.2015 unter http://www.gdcvault.com/play/1020177/Deformable-Snow-Rendering-in-Batman

[7] Tesselation: Erzeugung von Subgeometrie aus bestehender Geometrie

[8] Einsicht am 15.03.2015 unter http://www.dragonage.com

[9] Billboard-Textur: 2D-Textur, die dem Betrachter immer zugewandt ist

[10] Fluid-Implicit-Particle: Implizites Flüssigkeitspartikel

[11] Plastizität: Irreversible Verformung

[12] Elastizität: Reversible Verformung

aufhin plastisch. Die Verbindungen eines Meshes[13] bei großen Deformationen weisen Fehler auf. Das erfordert die Erneuerung des Meshes(Remeshing). Der Grund liegt darin, dass die Dreiecke durch die Verformungen umgedreht zum Mesh liegen. Eine Fehlerkorrektur ist nötig. [PKA+05] nutzt eine Punkt-basierte meshfreie Methode zur Simulation von Brüchen. Die Materie-Punkt-Methode ist ebenso meshfrei. Die robuste Berechnung von Materialmodellen bei großen Deformation wird von [SHST12, ITF04] adressiert. Eine Arbeit, die Gebrauch von einem Gitter zur Simulation von viskoelastischen Materialien, wie Schleim, Pudding oder Zahnpasta macht, ist [GBO04]. In der Materie-Punkt-Methode halten die Partikel alle über die Zeit veränderlichen Parameter. Das Gitter speichert über einen Simulationsschritt hinaus keine Daten. Das widerspricht den meisten anderen Techniken, die Meshes oder Oberflächen zur Simulation nutzen.

Schneedynamik: Die einzige Arbeit, die speziell die Simulation der Dynamik von Schnee behandelt ist [SSC+13], wie in Kapitel 1 angesprochen. Sie findet im Laufe dieser Bachelorarbeit als Referenz Nutzung. Es existiert eine Implementierung auf der GPU für CUDA[14], was die Nutzung auf Nvidia-Grafikkarten beschränkt. Diese Arbeit nutzt die GPGPU-Techniken von OpenGL unter Verwendung von Compute-Shadern, was die Simulation auf AMD- und Nvidia-Grafikkarten möglich macht. Eine angepasste, ähnliche Methodik zu [SSC+13] wird für das Schmelzen von Materialien unter Hitze verwendet [SSJ+14].

[13] Mesh: Repräsentation eines Objektes in der Computergrafik durch verbundene Dreiecke

[14] Max Liberman, Wil Yegelwel, Eric Jang, Tim Parsons. Einsicht am 15.03.2015 unter http://wyegelwel.github.io/snow/

3 Grundlagen

Dieses Kapitel vermittelt die Grundlagen der Simulation von Schnee, zeigt die Funktionsweise der Materie-Punkt-Methode auf und erläutert die Compute-Shader von OpenGL.

Kapitel 3.1 ist der Lehre der Kontinuumsmechanik gewidmet. 3.2 enthält Information zur Entstehung von Schnee und Abgrenzung verschiedener Schneearten. 3.3 geht im Detail auf die Dynamik von Schnee ein. 3.4 legt die Funktionsweise der Materie-Punkt-Methode dar. In 3.5 wird das für Schnee verwendete Materialmodell spezifiziert. 3.6 gewährt Einblicke in die grundlegende Nutzung von Compute-Shadern, ohne auf die im nächsten Kapitel aufzuzeigende Implementation einzugehen.

3.1 Kontinuumsmechanik

Dieses Kapitel dient die Grundlagen im Umgang mit der Kontinuumsmechanik zu vermitteln. Besprochen werden die Bestimmung von Deformationsgradienten, die materialspezifischen Paramater Youngscher Modulus und Poissonzahl und die Erhaltung von Masse und Impuls in einem Kontrollvolumen.

Spannung beschreibt die Kraft pro Flächeneinheit. Diese Kraft lässt sich in zwei Arten aufteilen:

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

Die Kraft F_{normal} besitzt dieselbe Richtung wie die Normale der Fläche. Sie erzeugt die Normalspannung σ. Die zur Oberfläche parallele Kraft $F_{parallel}$, erzeugt die Scherspannung τ (auch Schubspannung). Im dreidimensionalen Raum existie-

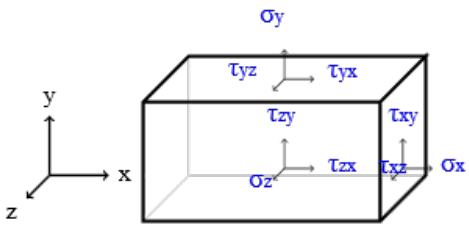


Abbildung 2: *Spannungen im dreidimensionalen Raum.*
Quelle (Einsicht am 27.02.2015):
http://upload.wikimedia.org/wikipedia/de/4/42/Spannung2.svg

ren genau zwei zur Oberfläche parallele Scherspannungen. Eine Fläche, die beispielsweise in xz-Richtung aufgespannt ist, besitzt eine Normalspannung in Richtung der y-Achse und Scherspannungen in Richtung x-Achse und z-Achse. Die Betrachtung eines dreidimensionalen Raumes bedingt ebenfalls, dass für jede der drei Raumrichtungen Normalspannung und zwei Scherspannungen existieren (sie-

he Abbildung 2). Dies lässt sich notieren als der Spannungstensor:

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_{yy} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_{zz} \end{bmatrix} = \begin{bmatrix} \sigma_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{xy} & \sigma_{yy} & \tau_{yz} \\ \tau_{xz} & \tau_{yz} & \sigma_{zz} \end{bmatrix}. \tag{2}$$

Zusätzlich gilt $\tau_{xy} = \tau_{yx}$, da τ_{xy} den Körper gegen den Uhrzeigersinn dreht und τ_{yx} den Körper mit dem Uhrzeigersinn dreht. Analog folgt: $\tau_{xz} = \tau_{zx}$ und $\tau_{yz} = \tau_{zy}$. $\boldsymbol{\sigma}$ ist ein symmetrischer Tensor.[15]

Dehnung ist das direkte Resultat von Spannung. Die Normaldehnung $\boldsymbol{\epsilon}$ oder ingenieurwissenschaftliche Dehnung und die Scherdehnung $\boldsymbol{\gamma}$ sind definiert als:

$$\boldsymbol{\epsilon} = \frac{\Delta L}{L_0} \text{ und } \boldsymbol{\gamma} = \frac{\Delta x + \Delta y}{T}. \tag{3}$$

L_0 ist die initiale Länge des Körpers und ΔL die Differenz zur neuen Länge. Bei einer positiven Normaldehnung spricht man von einer **Streckung**. Eine negative Dehnung entspricht einer **Stauchung**. Für die Dehnung lässt sich, analog zum

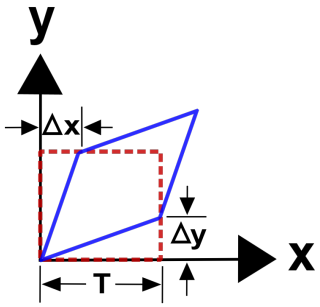


Abbildung 3: *Scherdehnung von Objekten.*
Quelle (Einsicht am 27.02.2015):
http://www.continuummechanics.org/cm/images/pure_shear_def.svg

Spannungstensor, ein Dehnungstensor aufstellen:

$$\boldsymbol{\epsilon} = \begin{bmatrix} \epsilon_{xx} & \gamma_{xy/2} & \gamma_{xz/2} \\ \gamma_{xy/2} & \epsilon_{yy} & \gamma_{yz/2} \\ \gamma_{xz/2} & \gamma_{yz/2} & \epsilon_{zz} \end{bmatrix} \text{ mit } \gamma_{ij} = 2\epsilon_{ij}. \tag{4}$$

Die Scherdehnung γ_{ij} entspricht für die korrekte Koordinatentransformation genau dem doppelten der ingenieurwissenschaftlichen Dehnungsterme ϵ_{ij}. Abbildung 3 veranschaulicht die Scherung eines Körpers mit der Seitenlänge T um Δx und Δy.[16]

[15] Einsicht am 28.02.2015 unter http://www.continuummechanics.org/cm/stress.html
[16] Einsicht am 28.02.2015 unter http://www.continuummechanics.org/cm/strain.html

Youngscher Modul (Elastizitätsmodul E) ist nach dem Hookeschen Gesetz der Quotient von Spannung $\boldsymbol{\sigma}$ zu Dehnung $\boldsymbol{\epsilon}$:

$$E = \frac{\boldsymbol{\sigma}}{\boldsymbol{\epsilon}} = \frac{L_0}{\Delta L}\frac{\boldsymbol{F}}{A}. \tag{5}$$

Höhere Spannung bei gleich bleibender Dehnung hat einen höheren Youngschen Modul zur Folge. Mehr Spannung ist also für die gleiche Längenänderung nötig. Der Youngsche Modul ist materialabhängig.[17]

Poissonzahl (ν) ist ebenso durch das Hookesche Gesetz herleitbar:

$$\epsilon_{xx} = \frac{1}{E}\sigma_{xx} - \frac{1}{E}\nu\sigma_{yy} - \frac{1}{E}\nu\sigma_{zz} = \frac{1}{E}[\sigma_{xx} - \nu(\sigma_{yy} + \sigma_{zz})]. \tag{6}$$

Bei einer Streckung in x-Richtung hat eine zusätzliche Streckung in y(,z)-Richtung die Auswirkung die Streckung in x-Richtung zu verringern. Eine zusätzliche Stauchung in y(,z)-Richtung verstärkt die Streckung in x-Richtung. Die Querspannungen sind proportional zur Spannung in x-Richtung. Proportionalität ermöglicht die Einführung der Konstante ν. Die Poissonzahl ist ein Maß für die Fähigkeit eines Materials zu stauchen bzw. zu strecken. Eine Stauchung hat eine Kompression[18] des Körpers zur Folge. Ein inkompressibles Material wie Gummi besitzt eine hohe Poissonzahl von 0.5. Schaum besitzt eine Poissonzahl von ungefähr 0.0; Die Kompression ist fast unbedingt möglich.[19]

Deformationsgradient: Die Verformung eines Körpers wird beschrieben durch die Funktion von initialer Konfiguration $\boldsymbol{X}$ zu deformierter Konfiguration $\boldsymbol{x}$ mit $\boldsymbol{x} = \boldsymbol{\phi}(\boldsymbol{X})$. Eine Konfiguration ist eine Repräsentation eines Körpers im abstrakten, euklidischen Vektorraum, auf dem sich rechnen lässt. Der Deformationsgradient $\boldsymbol{F}$ ergibt sich durch $\boldsymbol{F} = \partial\boldsymbol{\phi}/\partial\boldsymbol{X}$. Die Deformation $\boldsymbol{\phi}(\boldsymbol{X})$ ändert sich nach den Erhaltungssätzen von Masse und Impuls, welche später in diesem Kapitel behandelt werden und dem elasto-plastischem Materialverhalten, welches Thema in 3.5 ist. Nachfolgende Beispiele sind im zweidimensionalen Raum gehalten. [SSC+13]

Eine Streckung um 100% in x-Richtung und eine Stauchung um 100% in $y-$Richtung wird durch den Deformationsgradient $\boldsymbol{F}_{Skalierung}$ dargestellt. Die pure Scherung (keine Rotation) aus Abbildung 3 führt zum Deformationsgradient $\boldsymbol{F}_{Scherung}$:

$$\boldsymbol{F}_{Skalierung} = \begin{bmatrix} 2.0 & 0.0 \\ 0.0 & 0.5 \end{bmatrix}, \boldsymbol{F}_{Scherung} = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.0 \end{bmatrix}. \tag{7}$$

Es gilt jedoch Deformation von Rotationen und Translationen zu unterscheiden, die nicht zur Spannung und Dehnung beitragen. Eine Translation $x = X + 2$ und $y =$

[17] Einsicht am 28.02.2015 unter http://scienceworld.wolfram.com/physics/YoungsModulus.html
[18] Kompression: Abnahme des Volumens bei Zunahme der Dichte
[19] Einsicht am 28.02.2015 unter http://www.continuummechanics.org/cm/hookeslaw.html

$Y + 3$ taucht durch die Ableitung nicht im Deformationsgradient auf. Sie ist die Identitätsmatrix $\boldsymbol{I}$. Translationen sind unabhängig vom Deformationsgradienten. Eine Rotation um den Winkel θ kann beschrieben werden als $x = X\cos(\theta) - Y\sin(\theta)$ und $y = X\sin(\theta) + Y\cos(\theta)$. Der Deformationsgradient sieht aus wie folgt:

$$\boldsymbol{F}_{Rotation} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}. \tag{8}$$

Das entspricht nicht der Identitätsmatrix $\boldsymbol{I}$ und zeigt das grundlegende Problem der Deformation. Eine Rotation ist keine Deformation, trotzdem hat sie Einfluss auf den Deformationsgradienten. Adressiert wird dieses Problem durch die Polarzerlegung.[20]

Polarzerlegung zerlegt den Deformationsgradienten $\boldsymbol{F}$ in eine Rotationsmatrix $\boldsymbol{R}$ und eine symmetrische[21] Matrix $\boldsymbol{S}$ mit $\boldsymbol{F} = \boldsymbol{RS}$. Die Berechnung $\boldsymbol{F}^T\boldsymbol{F} = (\boldsymbol{RS})^T(\boldsymbol{RS}) = \boldsymbol{S}^T\boldsymbol{R}^T\boldsymbol{RS} = \boldsymbol{S}^T\boldsymbol{S}$ eliminiert die Rotationsmatrix $\boldsymbol{R}$. Die besondere Eigenschaft, dass die transponierte Rotationsmatrix gleich ihrer Inversen ist, findet Anwendung: $\boldsymbol{R}^T\boldsymbol{R} = \boldsymbol{R}^{-1}\boldsymbol{R} = \boldsymbol{I}$. Dieser Schritt hinterlässt die Unbekannte $\boldsymbol{S}$ in der Gleichung.

$\boldsymbol{F}^T\boldsymbol{F}$ ist symmetrisch und positiv semidefinit[22]. Die symmetrischen Matrizen gewähren eine weitere Vereinfachung: $\boldsymbol{F}^T\boldsymbol{F} = \boldsymbol{S}^T\boldsymbol{S} = \boldsymbol{S}^2$. Ist $\boldsymbol{S}$ über $\boldsymbol{S}^2$ berechnet, erfolgt das Bestimmen von $\boldsymbol{R}$ über $\boldsymbol{F} = \boldsymbol{RS} \Leftrightarrow \boldsymbol{R} = \boldsymbol{FS}^{-1}$. Eine weitere Variante die Polarzerlegung zu berechnen, folgt aus der Singulärwertzerlegung von $\boldsymbol{F}$.[23]

Singulärwertzerlegung ist definiert als $\boldsymbol{F} = \boldsymbol{U\Sigma V}^T$. Der Deformationsgradient $\boldsymbol{F}$ wird dabei in drei Matrizen aufgeteilt. $\boldsymbol{U}$ und $\boldsymbol{V}$ sind orthogonale[24] Matrizen. $\boldsymbol{\Sigma}$ ist eine Diagonalmatrix[25]. Wieder kann eine Komponente eliminiert werden durch die Multiplikation der Transponierten von $\boldsymbol{F}$ mit $\boldsymbol{F}$: $\boldsymbol{F}^T\boldsymbol{F} = \boldsymbol{V}^T\boldsymbol{\Sigma U}\boldsymbol{U}^T\boldsymbol{\Sigma}^T\boldsymbol{V} = \boldsymbol{V\Sigma\Sigma}^T\boldsymbol{V}^T$. Da $\boldsymbol{\Sigma}$ Diagonalmatrix ist, ergibt sich: $\boldsymbol{V\Sigma\Sigma}^T\boldsymbol{V}^T = \boldsymbol{V\Sigma}^2\boldsymbol{V}^T$, wobei $\boldsymbol{\Sigma}^2$ dem Quadrat der einzelnen Komponenten der Hauptdiagonalen entspricht. $\boldsymbol{V}$ kann als die Eigenvektoren von $\boldsymbol{F}^T\boldsymbol{F}$ bestimmt werden. Die Eigenwerte von $\boldsymbol{F}^T\boldsymbol{F}$ befinden sich in der Diagonalmatrix $\boldsymbol{\Sigma}^2$. $\boldsymbol{U}$ lässt sich bestimmen als: $\boldsymbol{F} = \boldsymbol{U\Sigma V}^T \Leftrightarrow \boldsymbol{U} = \boldsymbol{FV\Sigma}^{-1}$. Eine äquivalentes Ergebnis für $\boldsymbol{U}$ liefern die Eigenvektoren von $\boldsymbol{FF}^T$. Die Polarzerlegung von $\boldsymbol{F}$ ist darstellbar durch die Singulärwertzerlegung als $\boldsymbol{S} = \boldsymbol{V\Sigma V}^T, \boldsymbol{E} = \boldsymbol{UV}^T$ [26].[27]

[20] Einsicht am 28.02.2015 unter
http://www.continuummechanics.org/cm/deformationgradient.html

[21] Symmetrische Matrix: Transponierte Matrix ist gleich der Matrix selbst.

[22] Positiv semidefinite Matrix: Eigenwerte größer/gleich null.

[23] Einsicht am 28.02.2015 unter
http://www.continuummechanics.org/cm/polardecomposition.html

[24] Orthogonale Matrix: Inverse Matrix ist gleich der Transponierten.

[25] Diagonalmatrix: Komponenten, die nicht auf der Hauptdiagonalen liegen sind 0.

[26] Einsicht am 28.02.2015 http://www.mathematik.uni-ulm.de/m5/balser/Skripten/LA2.pdf

[27] Einsicht am 28.02.2015 unter (siehe nächste Seite)

Finite-Elemente-Methode (FEM)[28] kann den Deformationsgradienten $\boldsymbol{F}$ an einem Punkt in einem Element bestimmen. Das Element ist beispielsweise die von den Knotenpunkten 1,2,3 und 4 eingeschlossene Fläche aus Abbildung 4. Für jeden Knotenpunkt lässt sich eine Deformation $\boldsymbol{u} = (u, v)$ in x- und y-Richtung feststellen. u und v repräsentieren die Komponenten X und Y der Verformung. Für den Knotenpunkt 3 ist diese Verformung beispielsweise $u_3 = 3, v_3 = 2$. Jeder Punkt $p(X, Y)$ im Element kann nun mithilfe einer Interpolationsfunktion φ gewichtet werden und von der undeformierten in die deformierte Konfiguration übertragen werden:

$$u(X,Y) = \varphi_1(X,Y)u_1 + \varphi_2(X,Y)u_2 + \varphi_3(X,Y)u_3 + \varphi_4(X,Y)u_4. \quad (9)$$

Analog folgt $v(X, Y)$. Im Falle linearer Interpolation entspricht $\varphi_1(X, Y) = (1 - X)(1 - Y)/4$, $\varphi_2(X, Y) = (1 + X)(1 - Y)/4$, $\varphi_3(X, Y) = (1 + X)(1 + Y)/4$ und $\varphi_4(X, Y) = (1 - X)(1 + Y)/4$. Die Beziehung $\varphi_1 + \varphi_2 + \varphi_3 + \varphi_4 = 1$ sichert ab, dass der Punkt vollständig auf die Gitterknoten gewichtet ist. Der Deformationsgradient an einem Punkt ergibt sich nun durch:

$$\boldsymbol{F} = I + \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{X}} = \begin{bmatrix} 1 + \frac{\partial \boldsymbol{u}(X,Y)}{\partial X} & \frac{\partial \boldsymbol{u}(X,Y)}{\partial Y} \\ \frac{\partial \boldsymbol{v}(X,Y)}{\partial X} & 1 + \frac{\partial \boldsymbol{v}(X,Y)}{\partial Y} \end{bmatrix}. \quad (10)$$

Eine ähnliche Methodik kann für den mit dem Deformationsgradienten in Beziehung stehenden Geschwindigkeitsgradienten aufgestellt werden. [29]

Geschwindigkeitsgradient: Der materielle Geschwindigkeitsgradient einer Konfiguration $\boldsymbol{X}$ zur deformierten Konfiguration $\boldsymbol{x}$ ist definiert als $\dot{\boldsymbol{F}} = \partial \boldsymbol{v}/\partial \boldsymbol{X}$. $\dot{\boldsymbol{F}}$ ist die Ableitung nach der Zeit des Deformationsgradienten $\boldsymbol{F}$, was wie folgt gezeigt werden kann:

$$\frac{d\boldsymbol{F}}{dt} = \frac{d}{dt}\left(\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{X}}\right) = \frac{\partial}{\partial \boldsymbol{X}}\left(\frac{d\boldsymbol{x}}{dt}\right) = \frac{\partial \boldsymbol{v}}{\partial \boldsymbol{X}} = \dot{\boldsymbol{F}} \quad (11)$$

Eine Position $\boldsymbol{x}_t$ in einer Simulationen wird aktualisiert durch $\boldsymbol{x_{t+\Delta t}} = \boldsymbol{x}_t + \Delta t \boldsymbol{v}_{t+\Delta t}$. Analog kann der Deformationsgradient im nächsten Schritt berechnet werden mit $\boldsymbol{F}_{t+\Delta t} = \boldsymbol{F}_t + \Delta t \dot{\boldsymbol{F}}_{t+\Delta t}$.

Wenn der räumliche Geschwindigkeitsgradient $\boldsymbol{L} = \partial \boldsymbol{v}/\partial \boldsymbol{x}$ bestimmt wird, geschieht eine Umrechnung zum materiellen Geschwindigkeitsgradienten über:

$$\dot{\boldsymbol{F}} = \left(\frac{\partial \boldsymbol{v}}{\partial \boldsymbol{X}}\right)\left(\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{x}}\right) = \left(\frac{\partial \boldsymbol{v}}{\partial \boldsymbol{x}}\right)\left(\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{X}}\right) = \boldsymbol{L}\boldsymbol{F}. \quad (12)$$

http://ocw.mit.edu/courses/mathematics/18-06-linear-algebra-spring-2010/video-lectures/lecture-29-singular-value-decomposition/

[28] FEM: Aufteilung des zu berechnenden Volumens in eine endliche Anzahl von Elementen

[29] Einsicht am 27.02.2015 unter
http://www.continuummechanics.org/cm/finiteelementmapping.html

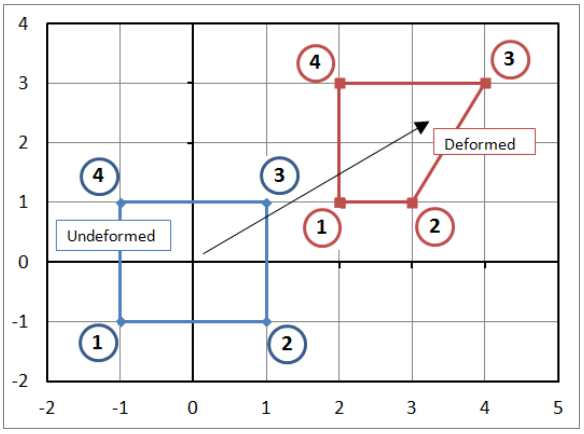


Abbildung 4: *Verformung eines Körpers in einem 2D-Koordinatensystem.*
Quelle(Einsicht am 27.02.2015):
http://www.continuummechanics.org/cm/images/FE_mapping_example.png

Der aktualisierte Deformationsgradient kann direkt durch $\boldsymbol{L}$ berechnet werden als: $\boldsymbol{F}_{t+\Delta t} = (\boldsymbol{I} + \boldsymbol{L}_{t+\Delta t})\boldsymbol{F}_t$.[30]

Erhaltungssätze: Die beiden Sätze aus (13) sind Folgerungen aus der Navier-Stokes-Gleichung. Sie besagen, dass im betrachteten, abgeschlossenen Volumen (Kontrollvolumen) Masse und Impuls weder erschaffen noch zerstört werden können. Masse (oder Impuls), die aus dem System entweichen, müssen wieder hinzugefügt werden. Nachfolgend werden diese Sätze näher erläutert.[31]

$$\frac{D\rho}{Dt} = 0, \quad \rho\frac{D\boldsymbol{v}}{Dt} = \nabla \cdot \boldsymbol{\sigma} + \rho\boldsymbol{g}. \tag{13}$$

Materielle Ableitung der Dichte ρ ist definiert als:

$$\frac{D\rho}{Dt} = \frac{\partial\rho}{\partial t} + \boldsymbol{v}\frac{\partial\rho}{\partial \boldsymbol{x}} = \frac{\partial\rho}{\partial t} + \boldsymbol{v}\nabla p. \tag{14}$$

Zur Veranschaulichung stelle man sich einen beliebigen, statischen Punkt p_0 im Kontrollvolumen vor. Wird der Teil vor der Addition aus Gleichung 13 gleich null gesetzt, so ergibt dies $\partial\rho/\partial t = 0$. Für den Punkt p_0 sagt dies aus, dass die Dichte an diesem Punkt über die Zeit konstant bleibt. Nimmt man den zweiten Teil der Gleichung hinzu, fügt dies die Dynamik im Volumen hinzu. Der vorher statische Punkt wird nun in seiner Bewegung verfolgt: Der zu verfolgende Punkt, der sich durch das Volumen bewegt, erhält eine konstante, unveränderliche Dichte. Eine al-

[30] Einsicht am 27.02.2015 unter
http://www.continuummechanics.org/cm/velocitygradient.html
[31] Einsicht am 27.02.2015 unter
http://www.continuummechanics.org/cm/continuityequation.html

ternative Schreibweise ist durch den Nabla-Operator ∇ gegeben. Wenn kein Punkt jemals das Volumen verlässt, geht die Erhaltung der Masse einher. [31]

In der Materie-Punkt-Methode sind diese Punkte Partikel. Sie halten die physikalischen Variablen, wie Position $\boldsymbol{x}_p$, Geschwindigkeit $\boldsymbol{v}_p$, Masse $\boldsymbol{m}_p$ und Deformationsgradient $\boldsymbol{F}_p$. Erste Gleichung aus (13) ist eine Vereinfachung des Erhaltungssatzes der Masse und gilt für inkompressible Materialien (beispielsweise Flüssigkeiten) [31]. Schnee ist kompressibel und die Dichte eines Partikels ändert sich dementsprechend. Die Materie-Punkt-Methode ignoriert, dass die Dichte eines Partikels Änderungen erfährt.

Cauchy-Impuls-Gleichung zeigt wie die Geschwindigkeiten eines Partikels auf seinem Weg durch das Feld von Kräften in einem Kontrollvolumen Ω beeinflusst werden. Eine Herleitung folgt aus dem zweiten newtonschen Gesetz:

$$\begin{aligned} ma &= \sum F \\ \Leftrightarrow \int_\Omega \rho \frac{D\boldsymbol{v}}{Dt} dV &= \int_\Omega \nabla \boldsymbol{\sigma} dV + \int_\Omega \rho g dV \\ \Leftrightarrow \rho \frac{D\boldsymbol{v}}{Dt} &= \nabla \boldsymbol{\sigma} + \rho g \end{aligned}$$

Die Dichte ρ integriert über das Kontrollvolumen ergibt die Masse m. Die Beschleunigung a ist genau die materielle Ableitung der Geschwindigkeit v. Die Kräfte im Volumen setzen sich aus den internen und den externen Kräften zusammen. Innere Spannungen $\nabla\sigma$ des Körpers sind die internen Kräfte. Die externe Kraft ist die Erdanziehungskraft ρg. Da dies für jedes beliebige Kontrollvolumen gilt, können die Integrale ausgelassen werden. [Gra12]

Dieses Kapitel stellte die Grundlagen der Kontinuumsmechanik vor, die elementar für das Verständnis der Materie-Punkt-Methode sind. Der Youngsche Modul und die Poissonzahl liefern Parameter, die sich abhängig vom Material modifizieren lassen. Der Deformationsgradient bietet eine Möglichkeit die Verformung von Körpern in Zahlen zu fassen. Die Finite-Elemente-Methode gibt eine Anleitung zum Bestimmen dessen. Die Erhaltungssätze zeigen die Rahmenbedingungen für die Simulation von Partikeln in einem Volumen auf.

3.2 Entstehung und Kategorisierung von Schnee

Schnee ist ein Hydrometeor, eine Teilgruppe der Meteore, die bezeichnend ist für den Auftritt von Wetterphänomenen in der Atmosphäre oder auf der Oberfläche der Erde. Kategorisierend für Hydrometeore ist ihre Entstehung aus atmosphärischem Wasserdampf. In diese Kategorie fallen Wolken, Nebel, Regen, Hagel, Tau und auch Schnee. Schnee ist zugehörig zu den (zu Boden) fallenden Hydrometeoren, auch Niederschlag genannt.

Entstehung von Schnee: Wasserdampf kondensiert zu kleinsten Wassertröpfchen, die im Gegensatz zum Wasserdampf dichter sind als die Luft und fallen auf die Erde: Der Auftritt von Niederschlägen. Die Besonderheit bei Schnee ist, dass der kondensierende Wasserdampf direkt zu Eis übergeht. Dieser lagert sich an Gefrierkernen (Staubteilchen oder Rußpartikeln) an [32]. Nach und nach lagert sich weiterer kondensierender Wasserdampf an, der die typischen Strukturen der Eiskristalle formt. [Lib01] [33]

Die für diesen Prozess notwendige Temperatur beträgt zwischen 0 °C und -35 °C. Die Erscheinung der Schneekristalle wird durch die Temperatur und den Übersättigungsgrad beeinflusst (siehe Abbildung 5). Die Sättigung beschreibt, wie viel Wasserdampf sich anteilig in der Luft befindet, auch relative Luftfeuchtigkeit. Ab einer relativen Luftfeuchtigkeit von 100% spricht man von der Übersättigung der Luft. Wasserdampf kondensiert in Folge. Die in Abbildung 5 verwendete Messeinheit ist die absolute Luftfeuchtigkeit (y-Achse); bei der Annahme, dass die Luft vollständig gesättigt ist (= relative Luftfeuchtigkeit 100%). Dies ist, wie oben beschrieben, die Voraussetzung für die Bildung von Schnee. Die absolute Luftfeuchtigkeit wird durch die Dichte des Wasserdampfes in der Luft gemessen.

Um -2 °C bilden sich die ersten Eiskristalle, die eine Plättchenform besitzen, ebenso um -15 °C. Um -6 °C entstehen längliche Säulen bis zu Nadeln. Um -27 °C bilden sich Plättchen und Säulen. Je höher die Übersättigung, desto ausgebildeter sind die entstehenden Strukturen. So entstehen -15 °C die hexagonalen (sechseckigen) Sterne, während die Eiskristalle um die -6 °C langgestreckte Nadeln formen. Die Temperatur bestimmt also die grundlegende Erscheinungsform. Während der Übersättigungsgrad die Wachstumsrate und damit die Ausbildung dieser Erscheinungsform bestärkt.

Zusätzlich wandern die Schneekristalle auf ihrer Reise zur Erde durch verschiedene Temperaturen und Sättigungsgrade: Dies führt zu einer einzigartigen Erscheinung jedes einzelnen Schneekristalls. [Lib01]

Trotz dieser Einzigartigkeit weisen die Kristalle untereinander eine durchaus ähnliche Struktur auf: Die einheitliche Ausbildung von sechs Ecken ist auf die hexagonale Gitterstruktur der Wassermoleküle zurückzuführen. Die sechs Arme, die Plättchen zu Schneesternen werden lassen, bilden sich durch Instabilitäten beim Wachstum der Kristalle. Die sechs Enden wachsen schneller als die Seiten der Plättchen. Sie ragen weiter aus dem Körper heraus und der Zugang zu umgebenden Wassermolekülen ist einfacher: Arme entstehen an deren Ecken. Bei der oben erwähnten Wanderung durch verschiedene Luftschichten, können sich an diesen Armen wiederum andere Schneekristalle bilden. Die symmetrische Ausführung gründet in den beinahe übereinstimmenden, physikalischen Einflüssen der Kristallenden. [Lib01]

Die Wanderung durch verschiedene Luftschichten kann andere physikalische

[32] Einsicht am 03.02.2015 unter www.planetwissen.de/natur_technik/klima/schnee/

[33] Dr. Markus Duschek. Vorlesung von Dr. Gerhard Karl Lieb. Einsicht am 03.02.2015 unter http://www.markusduschek.com/files/lieb_schnee_lawinen.pdf

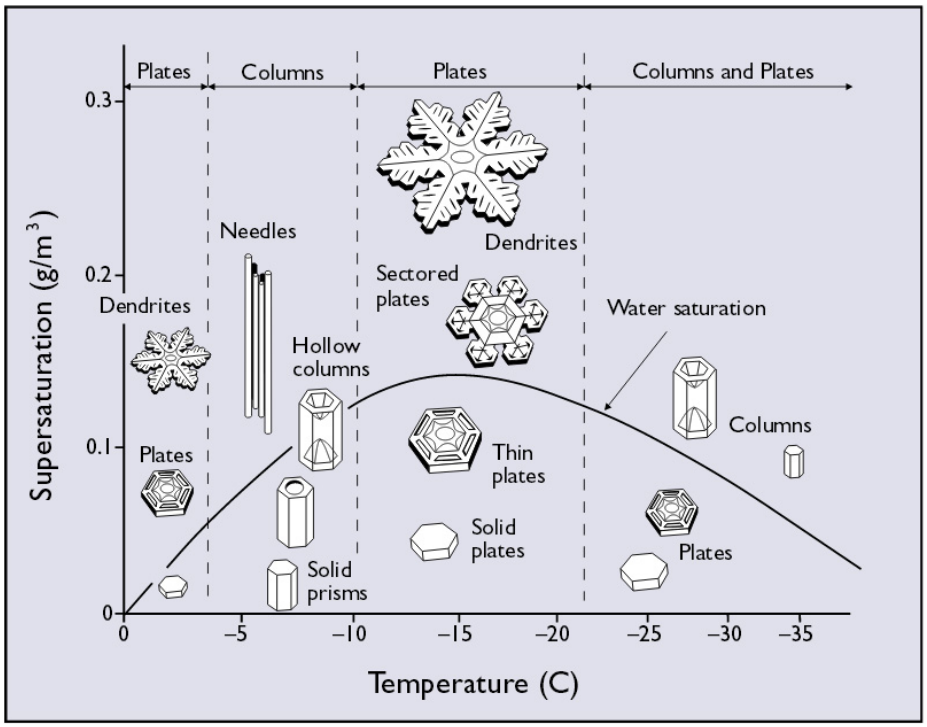


Abbildung 5: *Form der Schneekristalle.* Quelle: [Lib01]

Ereignisse als Wachstum hervorrufen. Wird der Schneekristall Temperaturen um 0 °C ausgesetzt, entstehen durch kleine Wassertröpfchen Verbindungen zwischen den Schneekristallen. Sie kleben bzw. verhaken sich ineinander und werden in ihrem Verbund als Schneeflocke bezeichnet. Sind die Temperaturen noch höher, schmelzen die Schneekristalle und werden zu Regen oder einer Mischung aus Schnee und Regen. Auch können die Schneekristalle Brüche in solchen Phasen erleiden. [34]

Kategorisierung der Schneearten findet über zwei Indikatoren statt: das Alter oder die Dichte des Schnees. Diese korrelieren stark, weswegen im Folgenden nach der Dichte kategorisiert und mit dem Alter in Bezug gesetzt wird. Eine Verbindung von Dichte und Form der Kristalle herzustellen, scheint aufgrund der verschiedenen Erscheinungen naheliegend. Die verzweigten Schneesterne weisen eine geringere Dichte auf, als die einfach anzuordnenden Plättchen und Säulen.

Casson [CSL08] stellt eine Beziehung zwischen Kristalltyp und Dichte der Schneekristalle her. Nach seinen Messungen von neu gefallenem Schnee betrüge die Dichte von Schnee mit überwiegender Anzahl von Schneesternen um 40 kg/m^3. Schnee mit einer größeren Anzahl von Plättchen und Säulen würden eine

[34] Dr. Markus Duschek. Vorlesung von Dr. Gerhard Karl Lieb. Einsicht am 03.02.2015 unter http://www.markusduschek.com/files/lieb_schnee_lawinen.pdf

höhere Dichte um 100 kg/m^3 aufweisen.

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

Tabelle 1: *Einteilung der Schneearten nach Dichtebereichen.* Quelle: [DWD]

Tabelle 1 zeigt eine mögliche Einteilung der Schneearten nach ihrer Dichte. Die Angaben beziehen sich auf unberührten, nur durch die Sonne beeinflussten Schnee; äußere Einflüsse anderer Lebewesen, wie eingebrochener Schnee oder künstliche Wärmequellen sind nicht mit einbezogen. Neuschnee besitzt Dichten von 50 - 150 kg/m^3. Teilweise wird die untere Grenze niedriger bis zu 30 kg/m^3 bei trockenem, lockeren Neuschnee gesetzt.[35] Er ist weniger als drei Tage alt und besteht weitestgehend aus Schneesternen. Pulverschnee besitzt Dichten zwischen 100 - 200 kg/m^3. Er ist trocken und klebt nicht aneinander, da er nicht genügend verdichtet ist. Vorwiegend besteht er aus Plättchen und Säulen. Durch immer wiederkehrendes Auftauen und Gefrieren und den Druck der eigenen Masse des Schnees wird er körnig und verliert die eigentliche Form der Schneekristalle. Körniger Schnee ist trocken. Die Dichten betragen je nach Alter zwischen 150 - 450 kg/m^3. Gelagerter Schnee ist feucht und weist eine Dichte von 350 - 600 kg/m^3 auf. Firnschnee ist über mindestens ein Jahr gealtert, seine Dichte liegt bei 500 - 850 kg/m^3. Firnschnee ist wasserundurchlässig. Unter weiterem Druck wird der Firnschnee zu Gletscherschnee und kann Dichten von massivem Eis (918 kg/m^3) einnehmen. Je öfter der Schnee schmilzt und wieder gefriert, desto schneller vollzieht sich dieser Vorgang. Trockene, kalte Gletscher brauchen deswegen Jahrzehnte, um aus Schnee Eis werden zu lassen. [DWD]

Die ähnliche, aber doch einzigartige Struktur von Schneekristallen durch verschiedene physikalische Einflüsse machen Schnee zu einem komplexen Phänomen. Die Dichte weist ein breites Spektrum auf, ist kategorisierend für die Art und das Alter des Schnees.

3.3 Dynamik von Schnee

Dieses Kapitel verbindet die erworbenen Kenntnisse von 3.1 und 3.2. 3.2 zeigt die Variabilität der Dichte durch verschiedenste Umweltfaktoren. Soll die Simulation von Schnee physikalisch korrekt ablaufen, müsste man alle Umweltfaktoren (Alter,

[35] Einsicht am 18.03.2015 unter
http://www.weltderphysik.de/thema/hinter-den-dingen/winterphaenomene/schneelast/

Wasser/Eis Anteil, Temperatur, etc.) betrachten. Es ist erforderlich die Faktoren bzw. Parameter überschaubar zu halten, um die Methodik nicht zu komplizieren und ineffizient zu machen.

Die grundlegenden, mechanischen Eigenschaften von Schnee sind nachfolgend aufgelistet. Eine Simulation sollte diese Eigenschaften ermöglichen: [SSC+13]:

- Volumenerhaltung: Flüssigkeiten besitzen die Eigenschaft inkompressibel zu sein. Schnee hingegen ist kompressibel, weist aber eine Resistenz zur Komprimierung auf.
- Elastizität: Fähigkeit eines Körpers nach Krafteinwirkung in den Grundzustand zurückzukehren.
- Plastizität: Irreversible Verformung eines Körpers nach Krafteinwirkung; Eine Rückkehr in den Grundzustand wird dementsprechend nicht mehr erreicht.
- Bruch: Körper zerbricht in verschiedene Teile, die wiederum eigene Körper bilden.

Ein Parameter, der durch seine Kompaktheit und die Kennzeichnung verschiedener Schneearten hervorsticht, ist die Dichte ρ. Sigrist [Sig06] stellt Zusammenhänge zwischen Dichte und dem Youngschen Modul wie auch der Poissonzahl her. Der Youngsche Modul und die Poissonzahl beschreiben den materialspezifischen Zusammenhang von Spannung und Dehnung (siehe 3.1).
Der Youngsche Modul E_{dyn} wird demnach durch

$$E_{dyn} = 1.89 * 10^{-6} \rho^{2.94} \text{MPa}, \tag{15}$$

berechnet. Eine Visualisierung dieser Gleichung befindet sich im Anhang A, Abbildung 16.

$$\nu = \nu_0 + (\rho - \rho_0) 5 * 10^{-4} \text{ m}^3/\text{kg} \tag{16}$$

beschreibt die Berechnung der Poissonzahl aus der Dichte ρ. ν_0 ist eine Konstante und standardmäßig mit 0.2 festgelegt. Die Ausgangsdichte ρ_0 beträgt 300kg/m^3. Die Abbildung 17 im Anhang A zeigt ebenfalls einen Graph dieser Funktion.

Schnee wird unter Kompression stärker und schwächer unter Streckung. Pulvriger, trockener(kalter) Schnee bricht schneller auf. Die Bindungen sind aufgrund des niedrigen Wassergehalts schwach und der Schnee verhält sich wie einzelne Körner. Schnee mit hohem Wassergehalt(warm) hat starke Bindungen, bricht langsamer und zeigt beim Bruch Brocken auf. Eisiger Schnee ist härter und besitzt ein höheren Youngschen Modul. Matschiger Schnee ist das Gegenteil: weicher mit geringerem Youngschem Modul.[SSC+13]

Die Materie-Punkt-Methode basiert auf beobachtbaren Parameter und ignoriert die Grundursachen durch die Umweltfaktoren. Vielmehr werden Parameter

verwendet, die der Modellierung unterliegen und eine Kombination aus den umweltbedingten Faktoren sind. Sie werden im Zuge des Materialmodells im Kapitel 3.5 erklärt. [SSC+13]

Die Dynamik von Schnee ist anspruchsvoll und von vielen Parametern abhängig. Für die Simulation von Schnee ist eine Reduzierung auf einfach zu modellierende, beobachtbare Parameter vorzuziehen. Drei Parameter stehen fest: Die Dichte, der Youngsche Modul und die Poissonzahl.

3.4 Materie-Punkt-Methode

In diesem Kapitel wird die Materie-Punkt-Methode behandelt, die der Ausgang der Simulation von Schnee ist. Kapitel 3.1 vermittelt hierzu die nötige Basis. Der Kern der Materie-Punkt-Methode ist die Kommunikation zwischen Lagrange-Partikeln und eulerschem Gitter zur Aktualisierung der physikalischen Parameter.

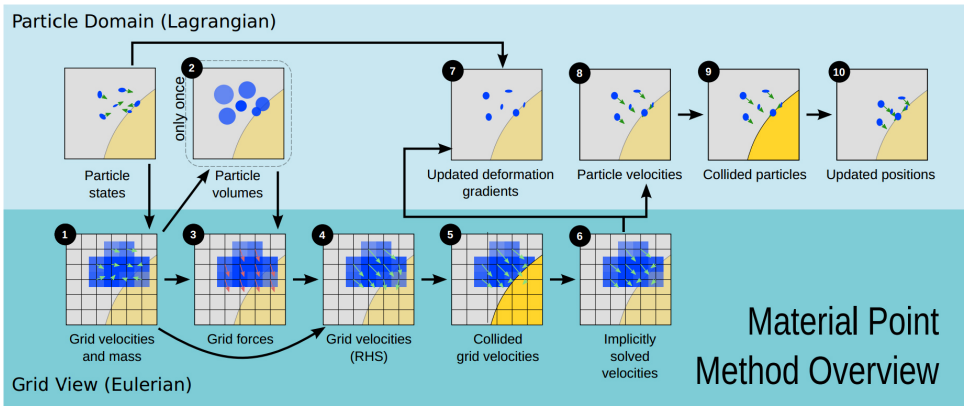


Abbildung 6: *Überblick der Materie-Punkt-Methode. Schritte in der oberen und unteren Hälfte rechnen auf Partikeln, während die Schritte zwischen den Hälften Berechnungen auf dem Gitter vollziehen.* Quelle: [SSC+13]

Als unterliegende, physikalische Einheit der Berechnung dienen Lagrange-Partikel. Sie halten Masse $\boldsymbol{m}_p$, Position $\boldsymbol{x}_p$, Geschwindigkeit $\boldsymbol{v}_p$ und Deformationsgradient $\boldsymbol{F}_p$. Kombiniert werden diese mit einem eulerschen Gitter, das die Verbindungen zwischen den Partikeln herstellt. Dazu werden die physikalischen Parameter auf die Gitterknoten übertragen. Nötige Berechnungen auf dem Gitter für die Aktualisierung der Parameter werden unternommen und auf die Partikel zurück übertragen. Die Gitterknoten werden dabei nicht verformt und behalten ihre Position bei. Die Finite-Elemente-Methode angewandt auf das Gitter ermöglicht das Bilden von Gradienten. [SSC+13]

Als Interpolationsfunktion für das Gitter werden dyadische Produkte von eindimensionalen kubischen B-Splines verwendet: [SSC+13]

$$N_{\boldsymbol{i}}^h(\boldsymbol{x}_p) = N(\frac{1}{h}(x_p - ih))N(\frac{1}{h}(y_p - jh))N(\frac{1}{h}(z_p - kh)). \tag{17}$$

mit Gitterindex $\boldsymbol{i} = (i, j, k)$ und relativer Partikelposition $\boldsymbol{x}_p = (x_p, y_p, z_p)$

$$N(x) = \begin{cases} \frac{1}{2}|x|^3 - x^2 + \frac{2}{3}, & 0 \leq |x| < 1 \\ -\frac{1}{6}|x|^3 + x^2 - 2|x| + \frac{4}{3}, & 1 \leq |x| < 2 \\ 0, & \text{sonst} \end{cases}. \tag{18}$$

Der Gradient der Gewichtungsfunktion berechnet sich aus der 1. Ableitung:

$$\nabla N_{\boldsymbol{i}}^h(\boldsymbol{x}_p) = \begin{pmatrix} \nabla N(\frac{1}{h}(x_p - ih))N(\frac{1}{h}(y_p - jh))N(\frac{1}{h}(z_p - kh)) \\ N(\frac{1}{h}(x_p - ih))\nabla N(\frac{1}{h}(y_p - jh))N(\frac{1}{h}(z_p - kh)) \\ N(\frac{1}{h}(x_p - ih))N(\frac{1}{h}(y_p - jh))\nabla N(\frac{1}{h}(z_p - kh)) \end{pmatrix}. \tag{19}$$

$$\nabla N(x) = \begin{cases} \frac{3}{2}|x|x - 2x, & 0 \leq |x| < 1 \\ -\frac{1}{2}|x|x + 2x - 2\frac{x}{|x|}, & 1 \leq |x| < 2 \\ 0, & \text{sonst} \end{cases}. \tag{20}$$

Im Folgenden wird die kompaktere Notation $w_{\boldsymbol{i}p} = N_{\boldsymbol{i}}^h(\boldsymbol{x}_p)$ und entsprechend $\nabla w_{\boldsymbol{i}p} = \nabla N_{\boldsymbol{i}}^h(\boldsymbol{x}_p)$ verwendet. Die Interpolationsfunktionen ermöglichen die Gewichtung der Masse $\boldsymbol{m}_p$ und Geschwindigkeit $\boldsymbol{v}_p$ von Partikeln auf das Gitter. Die Position $\boldsymbol{x}_p$ des Partikels ist der Indikator für die Gewichtung. h ist die Seitenlänge einer Gitterzelle. Nach Errechnung der neuen Geschwindigkeiten auf dem Gitter kann diese wieder auf die Partikel zurückgewichtet werden. Zuletzt folgt die Aktualisierung der Partikelpositionen $\boldsymbol{x}_p$. [SSC+13]

Es folgt der Update-Prozess durch die Materie-Punkt-Methode, der in Abbildung 6 visualisiert ist. Pfeile in der oberen Hälfte signalisieren Operationen auf Partikeln. In der unteren Hälfte wird hingegen auf Gitterknoten operiert. Die Operationen zwischen Partikeln und Gitterknoten sind die Operationen über das Gitter. Die Nummerierungen entsprechen der folgenden Aufzählung [SSC+13]:

1. **Übertragung der Partikelparameter auf das Gitter**: Die Update-Prozedur beginnt mit der Gewichtung der Masse auf das Gitter. Die Masse eines Gitterknotens zur Zeit t^n errechnet sich als $m_{\boldsymbol{i}}^n = \sum_p m_p w_{\boldsymbol{i}p}^n$. Zusätzlich wird die Geschwindigkeit anhand der Funktion $\boldsymbol{v}_{\boldsymbol{i}}^n = \sum_p \boldsymbol{v}_p m_p w_{\boldsymbol{i}p}^n / m_{\boldsymbol{i}}^n$ auf das Gitter übertragen. Zu beachten ist die Normalisierung durch die Masse des Gitterknoten. Dies sichert die Impulserhaltung. FLIP-Methodiken nutzen in der Regel keine Normalisierung.

2. **Feststellen von Volumen und Dichte eines Partikel.** Diese Berechnung wird initial, einmal am Anfang der Simulation, verrichtet. Die Dichte eines Partikels ändert sich danach, wie in Kapitel 3.1 besprochen, nicht mehr. Die Dichte einer Gitterzelle mit Seitenlänge h lässt sich feststellen als $m_{\boldsymbol{i}}^0/h^3$, wobei $m_{\boldsymbol{i}}^0$ wie in 1. berechnet wird. Der Transfer der Dichte auf die Partikel funktioniert über die Gewichtungsfunktion mit $\rho_p^0 = \sum_i m_i^0 w_{\boldsymbol{i}p}^0 / h^3$. Das Volumen eines Partikel ist $V_p^0 = m_p / \rho_p^0$.

3. **Berechnung der Kräfte auf dem Gitter**: Die Gitterkäfte auf dem Knoten $\boldsymbol{i}$ durch elastische Spannungen ist $\boldsymbol{f_i}(\boldsymbol{x})$. Die Beziehung zum Spannungstensor ergibt sich aus $\boldsymbol{f_i}(\boldsymbol{x}) = -\sum_p V_p^n \boldsymbol{\sigma}_p \nabla w_{\boldsymbol{ip}}^n$. Das Volumen zum Zeitpunkt t^n des Partikels p ist $V_p^n = J_p^n V_n^0$. Die Determinante des Deformationsgradienten ist $\det(F_p^n) = J_p^n$. Die Bestimmung von $\boldsymbol{\sigma}_p$ unterliegt dem Materialmodell in Kapitel 3.5.

4. **Updaten der Gittergeschwindigkeiten** Die neue Geschwindigkeit auf dem Knoten $\boldsymbol{i}$ berechnet sich nach $\boldsymbol{v_i}^{n+1} = \boldsymbol{v_i}^n + \Delta t(m_{\boldsymbol{i}}^{-1} \boldsymbol{f_i}^n + \boldsymbol{g})$. Die Gravitationsbeschleunigung $\boldsymbol{g}$ kann hier hinzugefügt werden.

5. **Rigid-Body Kollisionen mit den Gitterknoten** funktionieren, wie für die Kollision von Rigid-Bodies mit Partikeln üblich. Der Algorithmus wird in Kapitel 4.4 noch einmal vorgestellt.

6. **Semi-implizites Update**: Für eine akkuratere Berechnung von $\boldsymbol{v_i}^{n+1}$ kann ein semi-implizite anstatt einer expliziten Zeitintegration stattfinden. Diese wird in dieser Bachelorarbeit nicht behandelt, ist aber eine wichtige geplante Erweiterung für die Simulation.

7. **Aufstellen des neuen Deformationsgradienten**: Die Idee hinter dem Deformationsgradienten $\boldsymbol{F}_p$ in der Materie-Punkt-Methode ist, dass er zweiteilig ist:
$$\boldsymbol{F}_p^{n+1} = (\boldsymbol{I} + \Delta t \nabla \boldsymbol{v}_p^{n+1}) \boldsymbol{F}_{Ep}^n \boldsymbol{F}_{Pp}^n = \hat{\boldsymbol{F}}_{Ep}^{n+1} \hat{\boldsymbol{F}}_{Pp}^{n+1}. \tag{21}$$
Verwendet wird ein elastischer Deformationsgradient $\boldsymbol{F}_{Ep}$ für die elastischen Spannungen. Die plastische Deformation spiegelt sich im plastischen Deformationsgradienten $\boldsymbol{F}_{Pp}$ wider. Temporär wird jede Änderung des Deformationsgradienten dem elastischen Deformationsgradienten zugewiesen $\hat{\boldsymbol{F}}_{Ep}^{n+1} = (\boldsymbol{I} + \Delta t \nabla \boldsymbol{v}_p^{n+1}) \boldsymbol{F}_{Ep}^n$. Der Geschwindigkeitsgradient berechnet sich durch $\nabla \boldsymbol{v}_p^{n+1} = \sum_i \boldsymbol{v}_i^{n+1} (\nabla w_{\boldsymbol{ip}}^n)^T$.

   Wenn die elastische Deformation nun einen bestimmte Schwellwert überschreitet, wird die restliche Deformation dem plastischen Deformationsgradienten zugeschrieben. Hierzu wird die Singulärwertzerlegung von $\hat{\boldsymbol{F}}_{Ep}^{n+1} = \boldsymbol{U}_p \hat{\boldsymbol{\Sigma}}_p \boldsymbol{V}_p^T$ bestimmt. Die Diagonalmatrix $\boldsymbol{\Sigma}_p$ wird danach durch den Bereich von $\boldsymbol{\Sigma}_p = \text{clamp}(\hat{\boldsymbol{\Sigma}}_p, [1 - \theta_c, 1 + \theta_s])$ beschränkt. θ_c ist die kritische Kompression/Stauchung. Die kritische Streckung ist θ_s. Mehr zu diesen Parametern im Materialmodell (siehe 3.5). Nun können die finalen Deformationsgradienten berechnet werden nach:
$$\boldsymbol{F}_{Ep}^{n+1} = \boldsymbol{U}_p \boldsymbol{\Sigma}_p \boldsymbol{V}_p^T \text{ und } \boldsymbol{F}_{Pp}^{n+1} = \boldsymbol{V}_p \boldsymbol{\Sigma}_p^{-1} \boldsymbol{U}_p^T \boldsymbol{F}_p^{n+1}. \tag{22}$$

8. **Aktualisierung der Partikelgeschwindigkeiten**: Die neuen Geschwindigkeiten werden durch einen PIC-Anteil und einen FLIP-Anteil berechnet. Der PIC-Anteil berechnet sich aus: $\boldsymbol{v}_{\text{PIC}p}^{n+1} = \sum_i \boldsymbol{v}_i^{n+1} w_{\boldsymbol{ip}}^n$. Für den FLIP-Anteil

gilt: $\boldsymbol{v}_{\text{FLIP}p}^{n+1} = \boldsymbol{v}_p^n + \sum_i (\boldsymbol{v}_i^{n+1} - \boldsymbol{v}_i^n) w_{\boldsymbol{ip}}^n$. Die neue Partikelgeschwindigkeit ergibt sich aus der Zusammensetzung $\boldsymbol{v}_p^{n+1} = (1-\alpha)\boldsymbol{v}_{\text{PIC}p}^{n+1} + \alpha \boldsymbol{v}_{\text{FLIP}p}^{n+1}$ mit $\alpha = 0.95$.

9. **Kollisionen zwischen Partikeln und Rigid-Bodies**. Auf den neuen Partikelgeschwindigkeiten $\boldsymbol{v}_p^{n+1}$ wird ein weiteres Mal auf Kollisionen getestet; Details zum Algorithmus in Kapitel 4.4. Dies ist aufgrund der minimalen Abweichungen zwischen Gitter- und Partikelgeschwindigkeiten nötig.

10. **Berechnen der neuen Partikelpositionen**: Die neuen Positionen der Partikel lassen sich durch die neue Partikelgeschwindigkeit $\boldsymbol{v}_p^{n+1}$ berechnen: $\boldsymbol{x}_p^{n+1} = \boldsymbol{x}_p^n + \Delta t \boldsymbol{v}_p^{n+1}$.

Dieses Kapitel behandelte die Materie-Punkt-Methode zur Simulation von Lagrange-Partikeln über ein akkumulierendes, eulersches Gitter. Die Lagrange-Partikel werden auf die Gitterknoten übertragen. Mithilfe der Finite-Element Methode lassen sich Gradienten zur Ermittlung der neuen Geschwindigkeiten und Positionen der Partikel berechnen.

3.5 Materialmodell

Das in diesem Kapitel vorgestellte Materialmodell ist basierend auf [SSC+13]. Es unterliegt der physikalischen Basis der Kontinuumsmechanik. Das vereinfacht diese Basis jedoch, um mehr Kontrolle über die Simulation zu gewinnen. Dies ermögliche die große Bandbreite von Schnee unter wenigen Parametern visuell ansprechend darzustellen. Die totale, elastische potentielle Energie der undeformierten Konfiguration Ω^0 ist:

$$\int_{\Omega^0} \Psi(\boldsymbol{F}_E(\boldsymbol{X}), \boldsymbol{F}_P(\boldsymbol{X})) d\boldsymbol{X}. \tag{23}$$

Der Zusammenhang zwischen der Energie Ψ und dem totalen, elastischen Potential[36] des Gitters in der Materie-Punkt-Methode ist unter Einbeziehung der Positionen der Gitterknoten definiert als:

$$\Phi(\hat{\boldsymbol{x}}) = \sum_p V_p^0 \Psi(\hat{\boldsymbol{F}}_{Ep}(\hat{\boldsymbol{x}}), \boldsymbol{F}_{Pp}^n), \tag{24}$$

$\hat{\boldsymbol{x}}$ sind die gedachten, deformierten Positionen. Aufgrund der Definition der Materie-Punkt-Methode werden diese Punkte nicht wirklich verformt. V_p^0 ist das initiale Volumen eines Partikels p. Die deformierte Position $\hat{\boldsymbol{x}}_{\boldsymbol{i}}$ eines einzelnen Gitterknotens $\boldsymbol{i}$ ist gegeben durch $\hat{\boldsymbol{x}}_{\boldsymbol{i}} = \boldsymbol{x}_{\boldsymbol{i}} + \Delta t \boldsymbol{v}_{\boldsymbol{i}}$. Sie ist also allein abhängig von der Geschwindigkeit auf diesem Gitterknoten. $\boldsymbol{F}_p$ wird wie in Kapitel 3.4 aufgeteilt in einen elastischen($\boldsymbol{F}_{Ep}$) und einen plastischen($\boldsymbol{F}_{Pp}$) Anteil. $\boldsymbol{F}_{Pp}^n$ ist der plastische

[36] Potential: Feld(hier: Gitter) lässt den Körper Arbeit (Kraft pro Weg) verrichten

Deformationsgradient zur Zeit t^n. Der elastische Deformationsgradient $\hat{\boldsymbol{F}}_{Ep}(\hat{\boldsymbol{x}})$ ist nach [SZS95]:

$$\hat{\boldsymbol{F}}_{Ep}(\hat{\boldsymbol{x}}) = \left(\boldsymbol{I} + \sum_{\boldsymbol{i}} (\hat{\boldsymbol{x}}_{\boldsymbol{i}} - \boldsymbol{x}_{\boldsymbol{i}})(\nabla w_{\boldsymbol{ip}}^n)^T \right) \boldsymbol{F}_{Ep}^n. \tag{25}$$

Die partielle Ableitung des Potentials aus (24) nach $\hat{\boldsymbol{x}}_{\boldsymbol{i}}$ gibt die durch Spannung entstehende Kraft am Gitterknoten $\boldsymbol{i}$ an. Zusätzlich bedingt das die partielle Ableitung des elastischen Deformationsgradienten aus (25) nach $\hat{\boldsymbol{x}}_{\boldsymbol{i}}$. Das führt zur folgenden Abhängigkeit:

$$-\boldsymbol{f}_{\boldsymbol{i}}(\hat{\boldsymbol{x}}) = \frac{\partial \Phi}{\partial \hat{\boldsymbol{x}}_{\boldsymbol{i}}}(\hat{\boldsymbol{x}}) = \sum_p V_p^0 \frac{\partial \Psi}{\partial \boldsymbol{F}_E}(\hat{\boldsymbol{F}}_{Ep}(\hat{\boldsymbol{x}}), \boldsymbol{F}_{Pp}^n)(\boldsymbol{F}_{Ep}^n)^T \nabla w_{\boldsymbol{ip}}^n. \tag{26}$$

Die Relation zwischen elastischem Potential und der (Cauchy-)Spannung wird durch die Gleichung 27 zum Ausdruck gebracht.

$$\boldsymbol{\sigma} = \frac{1}{J} \frac{\partial \Psi}{\partial \boldsymbol{F}_E} \boldsymbol{F}_E^T. \tag{27}$$

Übrig bleibt die Evaluierung des Terms $\partial\Psi/\partial\boldsymbol{F}_E$. [SHST12] gibt einen robusten Term für die elastische, potentielle Energiedichte an. Er behandelt die bei großen Deformationen auftretenden Fehler in Simulationen, die einem Materialmodell unterliegen:

$$\Psi = \mu \sum_{\boldsymbol{i}} (\sigma_{\boldsymbol{i}} - 1) + \frac{\lambda}{2}(J-1)^2. \tag{28}$$

[SSC+13] modifizieren diesen Term, um die Relation von Spannung und Dehnung für Schnee herzustellen:

$$\Psi(\boldsymbol{F}_E, \boldsymbol{F}_P) = \mu(\boldsymbol{F}_P)||\boldsymbol{F}_E - \boldsymbol{R}_E||_F^2 + \frac{\lambda(\boldsymbol{F}_P)}{2}(J_E - 1)^2, \tag{29}$$

wobei $J_E = det(\boldsymbol{F}_{Ep})$. Analog folgt J_P. Die Frobeniusnorm einer m×n-Matrix A ist $||A||_F = \sqrt{spur(AA^H)}$. A^H ist die transponierte, konjugierte von A[37]. $\boldsymbol{R}_E$ erhält man durch die Polarzerlegung von $\boldsymbol{F}_E$, wie in Kapitel 3.1 beschrieben. Die Lamé-Parameter sind Funktionen des plastischen Deformationsgradienten:

$$\mu(\boldsymbol{F}_P) = \mu_0 e^{\xi(1-J_P)} \text{ und } \lambda(\boldsymbol{F}_P) = \lambda_0 e^{\xi(1-J_P)}. \tag{30}$$

ξ ist ein Indikator für die Härte eines Materials und wird vereinfacht in die Lamé-Parameter mit eingebracht, um ein entsprechendes, visuelles Ergebnis zu erzielen. Die Lamé-Konstanten[38] sind definiert durch den Youngschen Modul E und die Poissonzahl ν:

$$\lambda_0 = \frac{\nu E}{(1+\nu)(1-2\nu)} \text{ und } \mu_0 = \frac{E}{2(1+\nu)}. \tag{31}$$

[37] Einsicht am 06.03.2015 unter http://mathworld.wolfram.com/FrobeniusNorm.html

[38] Einsicht am 06.03.2015 unter http://scienceworld.wolfram.com/physics/LameConstants.html

Die partielle Ableitung $\partial\Psi/\partial\boldsymbol{F}_{Ep}$ lässt sich unter Wegfall der Frobeniusnorm berechnen als:

$$\frac{\partial\Psi}{\partial\boldsymbol{F}_{Ep}} = 2\mu(\boldsymbol{F}_P)(\boldsymbol{F}_E - \boldsymbol{R}_E) + \lambda(\boldsymbol{F}_P)(J_E - 1)J_E\boldsymbol{F}_{\boldsymbol{E}}^{-T}. \tag{32}$$

Das Material Schnee unterliegt bei kleinen Deformationen im Bereich von $[1 - \theta_c, 1 + \theta_s]$ elastischen Spannungen. Der plastische Deformationsgradient bleibt unter dieser Bedingung nach (22) die Identitätsmatrix. Große Werte von θ_c und θ_s veranlasse das Material Brocken zu bilden. Während kleine Werte das Material fein erscheinen ließe. Wenn die Deformation den festgeschriebenen Bereich überschreitet, fängt das Material an plastisch zu deformieren. In Kombination mit (30) mache dies das Material stärker bei Komprimierung und schwächer bei Stauchung. Letzteres bedingt den Bruch des Materials. [SSC$^+$13]

Der Härtekoeffizient ξ bestimme die Geschwindigkeit, mit der das Material breche. Ein hoher Härtekoeffizient mache das Material brüchiger. Gegenteilig mache ein kleiner Härtekoeffizient das Material geschmeidig. [SSC$^+$13]

Trockener Puderschnee habe geringe Kompressions- und Stauchungskonstanten θ_c und θ_s. Das Gegenteil sei nasser, bruckstückhafter Schnee. Eisiger Schnee besitze einen hohen Härtekoeffizient und Youngsches Modul. Matschiger Schnee habe einen geringen Härtekoeffizient und Youngsches Modul. Die Ausgangsparameter, die [SSC$^+$13] anbietet, sind in Tabelle 2 dargestellt.

Parameter	**Notation**	**Wert**
Kritische Komprimierung	θ_c	2.5×10^{-2}
Kritische Stauchung	θ_s	7.5×10^{-3}
Härtekoeffizient	ξ	10
Initiale Dichte (kg/m^3)	ρ_0	4.0×10^2
Initialer Youngscher Modul (Pa)	E_0	1.4×10^5
Poissonzahl	ν	0.2

Tabelle 2: *Nützliche Ausgangsparameter für das Modell nach [SSC$^+$13].*

In diesem Kapitel ist das Materialmodell für die Simulation von Schnee vorgestellt worden. Es basiert auf physikalischen Größen und Konzepten, vereinfacht die Sachverhalte aber. Die Kräfte auf den Gitterknoten werden durch das energetische Potential festgestellt. Mit der angegebenen Theorie soll die visuell, ansprechende Simulation von Schnee möglich sein.

3.6 Compute-Shader

Grafikprozessoren sind Mehrkernprozessoren mit der Fähigkeit Datensätze mit hoher Parallelisierung zu verarbeiten. Grafikprozessoren sind ursprünglich für die Darstellung von Computergrafik entwickelt worden. Neueste Änderungen ermöglichen die Erweiterung des Anwendungsgebietes fernab der Grafik (GPGPU). Die parallele Berechnung von Physik ist einer dieser Anwendungsfälle. Die Khronos Group führte mit der Erweiterung 4.3 von OpenGL die Compute-Shader ein. Compute-Shader werden in OpenGL Shading Language (GLSL) verfasst und erlauben GPGPU-Anwendungen. Von der hohen Anzahl an parallelen GPUs können viele Anwendungen in der Informatik profitieren. Mit Vulkan, vorgestellt von der Khronos Group am 03.03.2015 auf der GDC[39] 2015, wird ein Nachfolger zur OpenGL-API hervorgehen[40].[Hun13]

Die Materie-Punkt-Methode aus Kapitel 3.4 ist eine Anwendung, die durch die von OpenGL spezifizierten Compute-Shader realisiert werden kann. Dieses Kapitel behandelt die Grundlagen im Umgang mit Compute-Shadern. Teil dieses Kapitels sind das Anlegen von Shader-Storage-Buffer-Objekten (SSBOs) und der Zugriff auf diese; weiterhin das Aufrufen und Synchronisieren von Compute-Shadern. Die OpenGL-API wird durch C++ bedient.

Das **Anlegen eines Shader-Storage-Buffer-Objekts** auf C++-Seite ist in Quellcode 1 dargestellt. Die Methodik ähnelt dem Anlegen von Vertex-Buffer Objekten. SSBOs werden verwendet, um Daten in Form eines Arrays an den Shader weiterzugeben:

Quellcode 1: Anlegen eines SSBOs für Partikelgeschwindigkeiten auf C++ Seite.

```
1  glGenBuffers(1,&velB);
2  glBindBuffer(GL_SHADER_STORAGE_BUFFER, velB);
3  glBufferData(GL_SHADER_STORAGE_BUFFER,sizeof (Vector4i) * (particles)->↩
       size(), NULL, GL_STATIC_DRAW);
4  pVelocities = (Vector4i*) (glMapBufferRange(GL_SHADER_STORAGE_BUFFER,0,↩
       sizeof(Vector4i) * (particles)->size(), GL_MAP_WRITE_BIT| ↩
       GL_MAP_INVALIDATE_BUFFER_BIT));
5  for(int i = 0; i<particles->size();i++){
6    pVelocities[i]   = particles->at(i).velocity;
7    pVelocities[i].w = 0;
8  }
9  glUnmapBuffer ( GL_SHADER_STORAGE_BUFFER ) ;
10 glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0 , velB);
```

In Zeile 1 und 2 wird ein Buffer generiert und als SSBO gebunden. Zeile 3 legt die Größe des Buffers fest. Hier: die Anzahl der Partikel multipliziert mit der Größe des Datentyps. Das Befüllen des Buffers passiert in Zeile 4 bis 8 mit der Zuweisung des Buffers auf einen Pointer, der anschließend dazu dient den Datensatz in den Buffer zu schreiben. In Zeile 4 muss GL_MAP_WRITE_BIT gesetzt werden.

[39] GDC: Game Developers Conference

[40] Einsicht am 07.03.2015 unter https://www.khronos.org/news/press/khronos-reveals-vulkan-api-for-high-efficiency-graphics-and-compute-on-gpus

Zeile 9 macht den Buffer verfügbar für die GPU. Um den Buffer im Shader erreichbar zu machen, muss er gebunden werden. Zeile 9 bindet den SSBO velB auf die Zahl 0.

Aufrufen eines Shaders: Das Einbinden eines Shaders und das Mitgeben von Uniform-Variablen funktioniert analog zu den Vertex- und Fragment-Shadern. Anstatt eines Zeichenaufrufs (glDrawArrays(...)) wird für den Aufruf der Compute-Shader unter OpenGL 4.4 folgende Methode genutzt:

void **glDispatchComputeGroupSizeARB**(GLuint global_size_x,
GLuint global_size_y,
GLuint global_size_z,
GLuint local_size_x,
GLuint local_size_y,
GLuint local_size_z)

Die lokalen Gruppengrößen werden durch die drei local_size Variablen festgelegt. Die globalen Gruppengrößen (global_size_(x,y,z)) legen fest, wie viele der lokalen Gruppen existieren. Grafikkarten, die nur OpenGL 4.3 unterstützen, müssen die lokalen Gruppengrößen im Shader definieren (siehe Quellcode 2) und auf folgenden Befehl ausweichen.

void **glDispatchCompute**(GLuint global_size_x,
GLuint global_size_y,
GLuint global_size_z)

Quellcode 2: Lokale Arbeitsgruppengröße

```
1 #version 430
2 layout(local_size_x =1024, local_size_y =1,local_size_z =1)in;
```

Deklaration eines SSBOs: In GLSL werden SSBOs durch das Layout std140 und den gebundenen Index (siehe oben) deklariert:

Quellcode 3: Deklarieren eines SSBOs im Shader

```
1 uniform vec3 gGridPos;
2 layout(std140, binding = 1) buffer pVelVolume {
3 ivec4 pv[ ];
4 };
```

Zusätzlich wird der repräsentierende Datentyp des Buffers im Shader angeben (hier :ivec4). Die Repäsentation kann dabei auf Datentypen gesetzt werden, die nicht dem des Buffers entsprechen. Die Datentypen (i)vec3 und mat3 werden im Speicher immer als (i)vec4 und mat4 angelegt. Dies resultiert in Problemen beim Auswählen des richtigen Index. Ein Weg dies zu umgehen, ist generell von Vektoren und Matrizen der Größe vier auszugehen und die unbenutzten Dimensionen mit

anderen Werten zu füllen: Das spart Speicherauslastung auf der Grafikkarte. Ebenfalls behebt dies ein weiteres Problem: Die vorgestellte Notation funktioniert nicht auf Buffern mit Werten der Dimension 1 (int, float). Zugegriffen wird bei der Notation immer auf den Index 0. Der Grund dieser uneinheitlichen Funktionsweise bleibt fraglich.

Quellcode 4: Lese- und Schreibzugriff auf einen SSBO

```
1  uniform float dt;
2  vec3 vpn1 = vec3(0.0f,0.0f,0.0f);
3  void main(void){
4    uint pI = gl_GlobalInvocationID.x; //Bestimmen der Invokation.
5    vec3 pos = pxm[pI].xyz; //Lesezugriff.
6    pxm[pI].xyz = pos + dt * vpn1; //Schreibzugriff.
7
8    // pxm[pI].xyz += dt * vpn1;
9    //Funktioniert ebenso bei Lese- und Schreibzugriff.
10 }
```

Der **Zugriff auf die SSBOs** gelingt über die von GLSL implementierten Variablen gl_GlobalInvocationID. Sie geben jedem parallelen Aufruf(Invokation) einen Index. Die Anzahl wird durch die Dispatch[41]-Befehle von oben bestimmt. Mit diesem Index werden die Speicherstellen im SSBO angesprochen. Quellcode 4 zeigt den Lesezugriff mithilfe des Index in Zeile 5. Zeile 6 schreibt an der Stelle pI im Buffer die aktualisierte Position zurück.

GLSL kann überraschende Ergebnisse erzeugen, wenn es zum Suchen von Fehlern kommt. Der Compiler lässt viele Konstrukte zu, die in den meisten Fällen nicht intendiert sind. Ein Beispiel ist das Initialisieren einer 3×3 Matrix mit Vektoren der Größe vier:

Quellcode 5: Wahrscheinlich ungewolltes/unerwartetes Ergebnis einer Initialisierung einer Matrix

```
1      vec4 v1=vec4(1.0f,0.0f,0.0f,0.0f);
2      vec4 v2=vec4(2.0f,0.0f,0.0f,0.0f);
3      vec4 v3=vec4(3.0f,0.0f,0.0f,0.0f);
4      mat3 matrix= mat3(v1,v2,v3);  //matrix ist dann:
5      // 1 0 0
6      // 0 2 0
7      // 0 0 3
```

Dieses Konstrukt füllt die ersten vier Werte der Matrix mit den Werten von v1. Die Werte von v2 werden in die nächsten vier Werte eingetragen. Zuletzt, weil die Matrix noch einen freien Wert hat, wird der erste Wert von v3 eingetragen und der Rest abgeschnitten. Erwartungsgemäß sollte der Compiler dieses Konstrukt gänzlich verbieten, da ein solcher Fehler durch Unachtsamkeit entstehen kann. Ob die Implementation auf diese Weise von allen Grafikkarten unterstützt wird, ist fraglich.

[41] Dispatch: Auftrag zur parallelen Berechnung durch Compute-Shader

Die **Synchronisation** zwischen Shadern ist elementar für die Erhaltung von Massen und Impulsen. Der Befehl

glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);

versichert das alle Invokationen eines Shaders abgeschlossen sind, bevor der nächste Shader Invokationen erzeugt.

Ein weiteres Element der Synchronisation sind die atomaren Zugriffe. Diese sind nötig, falls mehrere Invokationen auf die gleiche Speicherstelle im Buffer zugreifen. Die OpenGL-API definiert den atomaren Zugriff nur auf Integern.[42] Da physikalische Größen in den meisten Fällen auf reellen Zahlen beruhen, ist dies ein Genauigkeitsverlust. Nvidia bringt für Grafikkarten seiner Familie eine Erweiterung mit sich, die atomaren Zugriff unter OpenGL auf floats erlauben.[43] AMD zeigt jedoch keine Anzeichen, dass dies jemals für Grafikkarten ihrer Serie in Frage kommt [44]. Eine Lösung für dieses Problem ist zwischen Gleitkommazahlen und Integern hin und her zu skalieren. Dies bringt weitere Schwierigkeiten mit sich, die später angesprochen werden.

Die Nutzung **atomarer Funktionen** ist nur auf einzelnen Integer-Werten definiert:

int **atomicAdd**(inout int mem, int data),
uint **atomicAdd**(inout uint mem, uint data);

mem ist das Ziel im Speicher; data die zu schreibenden Daten. Die einzelnen Werte von Integer-Vektoren der Größe vier lassen sich hiermit ansprechen, wie in Quellcode 6 gezeigt. Das Ansprechen von einzelnen Werten einer Matrix durch **atomicAdd(...)** schließt der Compiler kategorisch aus. Matrizen, auf die ein atomarer Zugriff vollzogen werden soll, werden deshalb durch drei Vektoren der Größe vier dargestellt. Neun Werte einer 4×4 Matrix sind zur Abbildung einer 3×3 Matrix nötig. Es bleiben sieben unbenutzte Werte, von denen vier durch das Nutzen von drei Vektoren eingespart werden können.

Quellcode 6: Nutzung von atomaren Operationen

```
1 vec3 velocity = (1.0f,1.0f,1.0f);
2 /*
3 * Atomarer Zugriff auf Integer: Die floating-point-Variablen werden
4 * skaliert und gecastet.
5 */
6 atomicAdd(gv[index].x,int(velocity.x*1e8f));
7 atomicAdd(gv[index].y,int(velocity.y*1e8f));
8 atomicAdd(gv[index].z,int(velocity.z*1e8f));
```

[42] Einsicht am 07.03.2015 unter https://www.opengl.org/sdk/docs/man/html/atomicAdd.xhtml

[43] Einsicht am 02.02.2015 unter https://www.opengl.org/registry/specs/NV/shader_atomic_float.txt

[44] Einsicht am 07.03.2015 unter http://devgurus.amd.com/message/1264179

OpenGL bietet mit Compute-Shadern die Möglichkeit Physik auf der Grafikkarte berechnen zu lassen. Compute-Shader sind kompatibel zwischen Grafikkarten von verschiedenen Herstellern, die die Implementierung der von OpenGL bereitgestellten Funktionen gewährleisten. Die Sprache GLSL bzw. der Compiler lassen teilweise fragwürdige Konstrukte zu, während andere Konstrukte umständlich erarbeitet werden müssen.

4 Implementation

Die Theorie von Schnee ist Thema in Kapitel 3. Dieses Kapitel beschreibt aufbauend auf dieser Theorie eine auf C++-basierte Anwendung, die Schnee simuliert und visualisiert. Sie kommuniziert für die Ausgabe der Grafik und der Berechnung der Physik mit der OpenGL-API.

Die Architektur der Anwendung ist geteilt in Rendering-Engine und Physik-Engine. Die Physik-Engine simuliert die Partikel, die anschließend von der Rendering-Engine visualisiert werden. Die Synchronisation von Physik-Engine und Rendering-Engine soll weitestgehend deterministisch sein: Eine erneute Ausführung soll das gleiche Ergebnis erzeugen. Eine wichtige Komponente hierzu ist, die Zeitschritte der Physik-Engine unabhängig von den Rendering-Schritten zu machen.

Glenn Fiedler[45] zeigt den folgenden Algorithmus 4 unter Verwendung eines Akkumulators auf. Die Physik-Engine wird mit konstanter, fixierter Delta-Zeit ausgeführt. Wenn die Physik innerhalb der Zeit STEP_DT berechnet ist, kann die übrige Zeit genutzt werden, um das Rendering auszuführen:

```
1  rE->init(); //Initialisiere Rendering-Engine.
2  pE->init(); //Initialisiere Physik-Engine.
3  double currentTime = glfwGetTime();
4  double accumulator = 0.0;
5    while (rE->shouldClose()){ // Falls kein Programmabbruch erzeugt wird.
6          double newTime = glfwGetTime(); // Zeit zum Anfang der Methode.
7          double frameTime = newTime - currentTime; //Zeit, die das letzte ↩
               Frame gebraucht hat.
8          currentTime = newTime; // Setze currentTime fuer naechsten Frame.
9          accumulator += frameTime;
10         // Fuehre solange Physikschritte aus bis es synchron zur STEP_DT ↩
               ist.
11         while (accumulator >= STEP_DT){ //STEP_DT = Physikschritte/↩
               Sekunde.
12           pE->update(PHYSIC_DT); // PHYSIC_DT = Delta-Zeit fuer die ↩
                 Simulation.
13           accumulator -= STEP_DT;
14         }
15         // Fuehre solange Renderschritte aus, wie Zeit nach Berechnung ↩
               der Physik bleibt.
16         rE->render();
17     }
18     rE->stop();
19 }
```

Dies garantiert, solange die Berechnung der Physik unter der STEP_DT bleibt, eine konstante Anzahl von Simulationsschritten pro Sekunde. Zu beachten ist, dass die Physik mit einer niedrigeren Delta-Zeit(PHYSIC_DT) berechnet wird. Diese ist von der Materie-Punkt-Methode vorgegeben: Eine Sekunde in der realen Welt entsprechen demnach nicht einer Sekunde in der Simulation. Die Zeit ist konstant und nie langsamer oder schneller. Dies adressiert der Algorithmus.

[45] Glenn Fiedler. Einsicht am 08.03.2015 unter http://gafferongames.com/game-physics/fix-your-timestep/

Im Kapitel 4.1 werden die Bibliotheken und Programme aufgezeigt, die die C++-Anwendung verwendet. 4.2 behandelt die Erstellung von Speichern auf der Grafikkarte. 4.3 stellt die Arten der parallelen Aufrufe in der Materie-Punkt-Methode vor. In 4.4 wird die Detektion und Behandlung von Kollisionsobjekten mit Schnee veranschaulicht. Gerendert wird die Simulation mit der in 4.5 vorgestellten Methodik. Abschließend gibt das Kapitel 4.6 einen Überblick über die Implementation.

4.1 Eingebundene Bibliotheken und Programme

Dieses Kapitel behandelt die Bibliotheken und Programme, die im Laufe der Entwicklung der C++-Anwendung integriert bzw. verwendet worden sind. Der Build-Prozess wird durch das plattformunabhängige Cmake[46] begleitet. Cmake erstellt aus dem Source-Code die nötigen, ausführbaren Dateien und organisiert das Projekt. Qt-Creator[47] ist ein Beispiel für eine Entwicklungsumgebung, die Cmake integriert. Cmake kann aber unabhängig von der Entwicklungsumgebung genutzt werden. Die Anwendung ist auf Windows unter Benutzung des MinGW-w64-Compilers[48] entstanden. Dieser bringt C++11/14-Unterstützung mit sich.

GLEW[49] stellt fest, welche OpenGL-Erweiterungen vom Zielsystem unterstützt werden. Der Fenstermanager GLFW[50] dient zum Anzeigen, zur Verwaltung und Erstellung von OpenGL-Kontext. Es erlaubt Interaktionen und Eingaben per Maus und Tastatur. Etay Meiri[51] bietet ein grundlegendes Tutorial zum Rendern von OpenGL-Kontext. Einige Ideen dieses Tutorials sind bei der Erstellung der Rendering-Engine mit eingeflossen. GLM[52] ist eine Mathematikbibliothek, die sehr rar in der Anwendung benutzt wird, da eine eigens verfasste Bibliothek für Vektoren und Matrizen verwendet wird. Die Bibliothek stb-image.c[53] lädt Texturen in die Anwendung, die mit OpenGL gerendert werden können. Die Open Asset Import Library (Assimp)[54] unterstützt das Laden von gesamten Meshes und die Verbindung dieser mit Texturen.

Das Aufnehmen von Videos ist mit der Open Broadcast Software[55] geschehen. Die Open Broadcast Software erlaubt das Aufnehmen von Videos in 1080p[56] bei 60 FPS[57]. Das Blender Project[58] ist für das Bearbeiten von Objektmeshes konzipiert. Mittlerweile erlaubt Blender aber viele andere Anwendungsmöglichkeiten.

[46] Kitware. Einsicht am 02.02.2015 unter http://www.cmake.org/
[47] Qt. Einsicht am 08.03.2015 https://www.qt.io/
[48] MinGW. Einsicht am 08.03.2015 http://sourceforge.net/projects/mingw-w64/
[49] Milan Ikits, Nigel Stewart. Einsicht am 02.02.2015 unter http://glew.sourceforge.net/
[50] Camilla Berglund. Einsicht am 02.02.2015 unter http://www.glfw.org/
[51] Etay Meiri. Einsicht am 02.02.2015 unter http://ogldev.atspace.co.uk/
[52] G-Truc Creation. Einsicht am 02.02.2015 unter http://glm.g-truc.net/0.9.6/index.html
[53] Sean Barret. Einsicht am 02.02.2015 unter http://nothings.org/
[54] assimp team. Einsicht am 02.02.2015 http://assimp.sourceforge.net/
[55] Open Broadcaster Software. Einsicht am 08.03.2015 unter https://obsproject.com/
[56] 1080p: Entspricht einer Auflösung von 1920×1080 Bildpixeln
[57] FPS: Frames/Bilder pro Sekunde
[58] Blender Foundation. Einsicht am 02.02.2015 unter http://www.blender.org/

Die Nutzung Blenders als Videobearbeitungssoftware erfüllt alle Anforderungen. Die Simulation von Schnee wird mit niedrigeren Delta-Zeiten(Δt) ausgeführt (siehe Kapitel 4). Das Beschleunigen der entstehenden Videos auf reelle Zeit erledigt Blender. Zusätzlich lassen sich viele, andere Effekte mit der Bearbeitungssoftware erledigen.

In der Simulation kann eine hohe Anzahl von Partikeln verwendet werden. Sämtliche Partikel per Hand zu setzen ist aufwendig. Um komplexere Objekte aus Schnee darzustellen, wird ein Voxelisierer verwendet. Dieser wandelt ein Mesh in Partikel um. Der Voxelisierer von Masayuki Takagi[59] füllt dabei, ähnlich der Scanline Konvertierung, das Innere des Meshes auf. Keiner der anderen ausprobierten Voxelisierer erledigt dies. Der Voxelisierer basiert auf der funktionalen Programmiersprache Common Lisp[60]. Dazu wird für den Voxelisierer der Bibliotheksmanager Quicklisp[61] und die Umgebung Emacs[62] benötigt. Außerdem muss das Mesh frei von Löchern sein. Volfill[63] adressiert dieses Problem. Der Voxelisierer überlässt dem Anwender das Einstellen der Partikeldichte, mit der das Mesh gefüllt werden soll. Zur Verbindung des Voxelisierers mit dem Projekt wird die Ausgabe modifiziert, um das Parsen[64] zu vereinfachen. Dies erlaubt die Nutzung von Objekten, die aus Partikeln bestehen. Die Singulärwertzerlegung ist von Eric Jang[65] übernommen und auf GLSL angepasst. Sie nutzen die in [MST+11] verwendete Methodik zur Berechnung der Singulärwertzerlegung von 3×3 Matrizen. Wie in Kapitel 3.1 besprochen wird erst eine Eigenanalyse auf $\boldsymbol{A}^T\boldsymbol{A}$ vollzogen. Die singulären Werte werden sortiert. Danach wird die QR-Zerlegung U und Σ durch die Givens-Rotation[66] bestimmt. Die Polarzerlegung wird, wie in 3.1 aufgezeigt, über die Singulärwertzerlegung berechnet.

4.2 Partikel und Gitter

Dieses Kapitel enthält, die zu erstellenden Strukturen und die Architektur der Anwendung.

Das Partikelsystem besteht aus vielen, einzelnen Partikeln. Ein Partikel hält dabei initial dessen Position $\boldsymbol{x}_p$, Masse m_p, Geschwindigkeit $\boldsymbol{v}_p$ und Deformationsgradient $\boldsymbol{F}_{Ep}$ und $\boldsymbol{F}_{Pp}$. Hinzu kommen im Verlaufe der Berechnung nach Kapitel 3.4 die Dichte ρ_{p0}, der Geschwindigkeitsgradient $\Delta\boldsymbol{v}_p^{n+1}$ und ein Buffer für die Berechnung der neuen Geschwindigkeiten $\boldsymbol{v}_p^{n+1}$.

[59] Masayuki Takagi. Einsicht am 02.02.2015 unter https://github.com/takagi/cl-voxelize/

[60] The Common Lisp Foundation. Einsicht am 02.02.2015 unter https://www.common-lisp.net/

[61] Zach Beane. Einsicht am 02.02.2015 unter http://www.quicklisp.org/

[62] Free Software Foundation, Inc. Einsicht am 02.02.2015 unter http://www.gnu.org/software/emacs/

[63] Volfill: Einsicht am 08.03.2015 unter http://graphics.stanford.edu/software/volfill/

[64] Parsen: Einlesen einer Datei

[65] Eric Jang. Einsicht am 08.03.2015 unter https://github.com/ericjang/svd3

[66] Givens-Rotation: Rotation der Matrix, um hintereinander die einzelnen Werte der Matrix, die sich nicht auf der Hauptdiagonalen befinden, auf 0 zu setzen (≙Diagonalmatrix).

Das Gitter enthält die Gitterknoten. Spezifiziert wird das Gitter über die Dimension in x-,y- und z-Richtung, der Position im Raum und der Seitenlänge h einer Gitterzelle. Die Gitterknoten halten fünf weitere physikalischen Parameter: Die Position des Knotens $\boldsymbol{x}_i$, die gewichtete Masse m_i und Geschwindigkeit $\boldsymbol{v}_i$, die Kraft $\boldsymbol{f}_i$ und die neue Geschwindigkeit $\boldsymbol{v}_i^{n+1}$.

Tabelle 3 und 4 zeigen, wie die Parameter im Shader repräsentiert werden. Die OpenGL eigenen ivec4, vec4 und mat4 entsprechen Vector4i, Vector4f und Matrix4f, der eigens geschriebenen Mathe-Bibliothek auf C++-Seite. Die Datentypen ergeben sich aus den in Kapitel 3.6 angesprochenen Punkten. Das Erstellen der Shader-Storage-Buffer-Objekte ist im selben Kapitel beschrieben. Quellcode 1 zeigt die Erstellung des Buffers für $\boldsymbol{v}_p$ und ρ_{p0}. Die Größe eines Partikel-SSBOs beträgt: (Größe des Datentyps)×(Partikelanzahl). Ein Gitter-SSBOs hat die Größe: (Größe des Datentyps)×(Anzahl der Gitterknoten). Die Anzahl der Gitterknoten ergibt sich aus: (x-Dimension)×(y-Dimension)×(z-Dimension).

Parameter	**Repräsentation im Shader**
$\boldsymbol{x}_p$, m_p	vec4(x_p,y_p,z_p,m_p)
$\boldsymbol{v}_p$, ρ_{p0}	ivec4(v_{px},v_{py},v_{pz},ρ_{p0})
$\boldsymbol{F}_{Ep}$	mat4(F_{Epxx},F_{Epyx},F_{Epzx}, 0.0f, F_{Epxy},F_{Epyy},F_{Epzy}, 0.0f, F_{Epxz},F_{Epyz},F_{Epzz}, 0.0f, 0.0f , 0.0f , 0.0f , 0.0f)
$\boldsymbol{F}_{Pp}$	mat4(F_{Ppxx},F_{Ppyx},F_{Ppzx}, 0.0f, F_{Ppxy},F_{Ppyy},F_{Ppzy}, 0.0f, F_{Ppxz},F_{Ppyz},F_{Ppzz}, 0.0f, 0.0f , 0.0f , 0.0f , 0.0f)
$\boldsymbol{v}_p^{n+1}$	ivec4(v_{px}^{n+1},v_{py}^{n+1},v_{pz}^{n+1},0.0f)
$\Delta\boldsymbol{v}_p^{n+1}$	ivec4(Δv_{pxx}^{n+1},Δv_{pyx}^{n+1},Δv_{pzx}^{n+1}, 0.0f), ivec4(Δv_{pxy}^{n+1},Δv_{pyy}^{n+1},Δv_{pzy}^{n+1}, 0.0f), ivec4(Δv_{pxz}^{n+1},Δv_{pyz}^{n+1},Δv_{pzz}^{n+1}, 0.0f)

Tabelle 3: *Repräsentation der Partikel-Parameter im Shader*

Sowohl Partikel als auch Gitter sind nach dieser Interpretation statisch angelegt und erlauben es nicht, während der Laufzeit neue Partikel zu erschaffen oder das Gitter zu vergrößern. Bei Rendering-Schritten werden die Partikel-Positionen von der Rendering-Routine als Vertex-Buffer interpretiert. Dies kann wahlweise mit den Positionen der Gitterknoten für ein Gitter-Rendering geschehen. Es ist nicht ausgeschlossen weitere physikalische Konstanten zu übergeben, um die Erscheinung des Renderings zu verbessern.

In diesem Kapitel sind die konkreten Shader-Storage-Buffer-Objekte aufgezählt, auf denen die Berechnungen der Materie-Punkt-Methode stattfinden. Diese ist Thema des nächsten Kapitels.

Parameter	Repräsentation im Shader
$\boldsymbol{x}_i$	vec4(x_i,y_i,z_i,0.0f)
$\boldsymbol{v}_i$, m_i	ivec4(v_{ix},v_{iy},v_{iz},m_i)
$\boldsymbol{f}_i$	ivec4(f_{ix},f_{iy},f_{iz},0.0f)
$\boldsymbol{v}_i^{n+1}$	ivec4(v_{ix}^{n+1},v_{iy}^{n+1},v_{iz}^{n+1},0.0f)

Tabelle 4: *Repräsentation der Gitterknoten-Parameter im Shader*

4.3 Materie-Punkt-Methode auf der GPU

Dieses Kapitel vermittelt die Implementation der aus 3.4 und 3.5 besprochenen Methodik auf der GPU mithilfe von Compute-Shadern. Aufgezeigt werden die Arten der Aufrufe.

Wenn auf C++-Seite ein Physik-Update eingereiht wird, werden eine Reihe von Dispatches hintereinander gestartet. Getrennt sind einige durch den Befehl **glMemoryBarrier**(GL_SHADER_STORAGE_BARRIER_BIT). Dies stellt sicher, dass die vorherige Phase abgeschlossen ist. Die Phase, die auf Werte aus der vorherigen zugreift, hat somit immer dasselbe Ergebnis. Eine Wettlaufsituation (Race-Condition) entsteht also nicht.

Die lokale Gruppengröße ist unabhängig von der globalen Gruppengröße (im Shader) wie in Quellcode 2 mit 1024 Threads in x-Richtung und einem Thread in jeweils y- und z- Richtung gewählt.

Im Laufe eines Physikschrittes werden auf drei verschiedenen globalen Gruppengrößen parallele Berechnungen ausgeführt. Die erste Größe sind die **Partikelbasierten Operationen**, die wie folgt aufgerufen werden:

```
glDispatchCompute(particlesystem->particles->size()
                  /NUM_OF_GPGPU_THREADS_X+1,
                  1,
                  1)
```

Dazu wird ein Dispatch auf der Anzahl der Partikel ausgeführt. Dieser wird durch die Größe der lokalen Gruppengröße geteilt, um die globale Gruppengröße festzustellen. Da beim Dividieren der Rest nicht mit einbezogen wird, wird eine zusätzliche Gruppe benötigt. So werden auch die restlichen Partikel simuliert. Ein Beispiel für eine solche Operation ist das Aktualisieren der Position $\boldsymbol{x}_p$ eines Partikels.

Für **Gitterknoten-basierte Operationen**, die ausnahmslos auf dem Gitter arbeiten, wird folgender Dispatch verwendet:

glDispatchCompute(GRID_DIM_X * GRID_DIM_Y * GRID_DIM_Z
/NUM_OF_GPGPU_THREADS_X+1,
1,
1)

Die Anzahl der Gitterknoten entsprechen den Dimensionen in die drei Raumrichtungen. Eine Berechnung, die diesen Dispatch benötigt, ist die Berechnung von $\boldsymbol{v}_i^{n+1}$.

Die **Gitter-basierten Operationen**, die zwischen Gitter und Partikel kommunizieren, bilden die letzte Größe. Eine naive Herangehensweise wäre einen Dispatch über alle Partikel zu starten und im Shader mit einer for-Schleife über die 64 benachbarten Gitterknoten zu iterieren. Dies widerspricht, aber dem Parallelisierungsgedanken der GPU: Die Performanz sinkt ab. Der Grad der Parallelisierung sollte so hoch wie möglich gewählt werden. Eine bessere Variante zeigt folgender Dispatch-Befehl:

glDispatchCompute(GRID_DIM_X * GRID_DIM_Y * GRID_DIM_Z
/NUM_OF_GPGPU_THREADS_X+1,
PARTICLE_TO_GRID_SIZE,
1)

Quellcode 7: Berechnung zwischen Gitterknoten und Partikel

```
1  /**
2    * Nimmt einen Integer von [0,63], entsprechend einem der 64
3    * Gitterknoten-Nachbarn (4x*4y*4z = 64 Nachbarn) und gibt die relative,
4    * gerundete Position  (mit Vorzeichen) zum Knoten zurueck
5    * (i,j,k Element von [-1,2]).
6    */
7  const int width = 4;
8  const ivec3 windowOffset=ivec3(-1,-1,-1);
9
10 void getIJK(const int index,inout ivec3 ijk){
11     int temp = index%(width*width);
12     ijk= ivec3(temp%width,temp/width,index/(width*width))+windowOffset;
13 }
14 /**
15  * Nimmt einen Gitterindex (ijk) und gibt den respektiven Index im Buffer↩
        zurueck.
16  * i im Bereich von [0,Gitterdimension in x-Richtung]
17  * j im Bereich von [0,Gitterdimension in y-Richtung]
18  * k im Bereich von [0,Gitterdimension in z-Richtung]
19  */
20 void getIndex(const ivec3 ijk,inout int index){
21     index = ijk.x + (ijk.y * int(gGridDim[0].x)) + (ijk.z *int(gGridDim↩
          [1].x) * int(gGridDim[0].x));
22 }
```

PARTICLE_TO_GRID_SIZE ist die Anzahl der Gitterknoten-Nachbarn. Bei der Materie-Punkt-Methode genau 64. Jedes Partikel führt also 64 „voneinander unabhängige“ Berechnungen durch, um mit dem Gitter zu kommunizieren. Beispielhaft sei hier die Berechnung der Massen des Gitters m_i genannt; Generalisierend ist dies jede Berechnung mit einer Summe über Partikel oder Gitterknoten.

Nachfolgend wird gezeigt, wie eine Shader-Invokation den betreffenden Gitterknoten auswählt. Dies stellt die Verbindung von Partikeln zu Gitterknoten her. Dazu werden zwei Funktionen benötigt, die in Quellcode 7 abgebildet sind.

Die erste Funktion getIJK(...) nimmt einen eindimensionalen Index für einen der 64 Gitternachbarn entgegen und kreiert daraus einen dreidimensonalen Indexvektor in $\mathbb{Z}$ mit $(i_r,j_r,k_r) \in [-1,2]$.

getIndex(...) nimmt wiederum einen Indexvektor (i_g,j_g,k_g) entgegen. Dieser Vektor gibt die dreidimensionale Position eines Gitterknotens im Gitter an. Zurückgegeben wird der eindimensionale Index für die Speicherstelle im Buffer.

Quellcode 8: Aufbau eines zwischen Partikel und Gitter kommunizierenden Shaders

```
1  layout(std140, binding = 0) buffer pPosMass {
2      vec4 pxm[ ];
3  };
4  layout(std140, binding = 3) buffer gVel {
5      ivec4 gv[ ];
6  };
7  vec4 particle = pxm[gl_GlobalInvocationID.x];
8  vec3 xp= particle.xyz; // Position des Partikels.
9  float mp = particle.w; // Masse des Partikels.
10
11 int gridOffsetOfParticle = int(gl_GlobalInvocationID.y); //(zwischen ↩
       0-63)
12 ivec3 gridOffset;
13 getIJK(gridOffsetOfParticle,gridOffset ); //Berechne den Offset in IJK-↩
       Koordinaten des Gitterknotens fuer diese Invokation.
14 vec3 ParticleInGrid= (xp- gGridPos)/gridSpacing; // Transformiere das ↩
       Partikel in das Koordinatensystem des Gitters.
15 ivec3 gridIndex = ivec3(ParticleInGrid) + gridOffset; // die IJK- ↩
       Koordinaten der Gitterzelle in der das Partikel ist, werden addiert ↩
       mit dem Offset des Gitterknotens. Es ergibt sich der entgueltige ↩
       Gitterknoten fuer diese Invokation.
16
17 if(gridIndex.x>= n && gridIndex.y>=n && gridIndex.z>=n && gridIndex.x< ↩
       gGridDim[0].x && gridIndex.y <gGridDim[1].x &&gridIndex.z< gGridDim↩
       [2].x ){
18         // Wenn der berechnete Gitterpunkt ausserhalb der Dimension des ↩
               Gitters liegt, wird er emittiert.
19         vec3 gridDistanceToParticle = vec3(gridIndex)- ParticleInGrid; //↩
               Berechne Abstand von Partikel zu Gitterknoten.
20         float wip = .0f;
21         weighting (gridDistanceToParticle,wip);// Berechne die Gewichtung↩
               fuer den korrespondierenden Gitterknoten.
22         int gI;
23         getIndex(gridIndex,gI);
24         atomicAdd(gv[gI].w,  int(mp * wip* 1e8f)); //Fuehre Addition auf ↩
               dem Gitterknoten aus.
25 }
```

Quellcode 8 zeigt den Aufbau eines Shaders, der die Massen der Partikel auf die Gitterknoten transformiert. Zeile 7 identifiziert die Partikel-Position und Masse für diese Invokation. Zeile 11 gibt den Gitternachbarn dieses Partikels an. In Zeile 13 wird diese in dreidimensionale Koordinaten umgewandelt. Durch Zeile 14 wird das Partikel in das Koordinatensystem des Gitters übertragen. Die gerundete Partikelposition addiert mit dem Offset des Gitternachbarn in Zeile 15 ergibt den Gitterknoten für diese Invokation. Wenn der errechnete Gitterknoten tatsächlich innerhalb des Gitters liegt (Zeile 17), wird die Distanz von Partikel zu Gitterknoten berechnet (Zeile 19). Anhand der Distanz wird die Gewichtung w_{ip} bestimmt. Mit getIndex wird die dreidimensionale Gitterposition in den eindimensionalen Index umgewandelt, der die Stelle für diese Gitterknoten im SSBO angibt. Mit einem atomaren Zugriff (Zeile 24) wird die Masse auf diesen Knoten aufaddiert. Jedes Partikel in der Umgebung von $[-2h, 2h]$ in alle drei Raumrichtungen um den Gitterknoten addiert seine Masse auf, gewichtet durch den Abstand zum Knoten. Jedes Partikel besitzt dafür 64 Invokationen.

Dieses Kapitel behandelt die Arten der Aufrufe. Partikel-basierte Operationen rechnen auf Partikeln; Gitterknoten-basierte Operationen auf Gitterknoten. Die Kommunikation zwischen Gitter und Partikel findet über Gitter-basierte Operationen statt. In Kapitel 4.6 werden anhand des Ablaufdiagramms einige Besonderheiten und Anpassungen der Implementation besprochen.

4.4 Kollisionen mit Rigid-Bodies

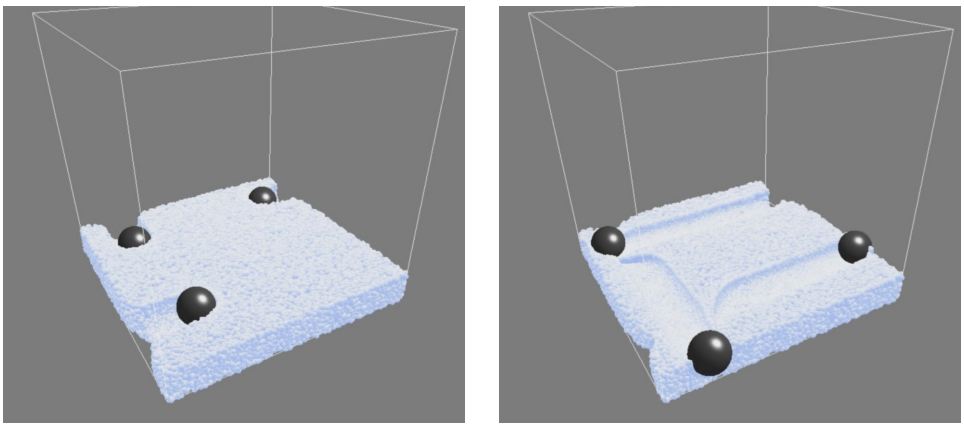


Abbildung 7: *Kollisionsobjekte, die durch eine Schneedecke wandern.*

In diesem Kapitel wird die Interaktion des Schnees mit Kollisionsobjekten gezeigt. Die Kollisionsobjekte werden auf der GPU simuliert. Unterstützt in der Anwendung werden Kugeln und Ebenen. Eine Anbindung an eine Physik-Engine zur Kollisionsdetektion ist denkbar.
Tabelle 5 zeigt die Shader-Storage-Buffer-Objekte für die Kollisionsobjekte. Die

Parameter	Repräsentation im Shader
$\boldsymbol{x}_c$, $type_c$	vec4(x_c,y_c,z_c,$type_c$)
$\boldsymbol{v}_c$	vec4(v_{cx},v_{cy},v_{cz},0.0f)
$\boldsymbol{n}_c$, μ	vec4(n_{cx},n_{cy},n_{cz},μ)

Tabelle 5: *Repräsentation der Kollisionsobjekte-Parameter im Shader*

Parameter sind die Position $\boldsymbol{x}_c$, der Typ $type_c$, die Geschwindigkeit $\boldsymbol{v}_c$, die Normale $\boldsymbol{n}_c$ und der materialabhängige Reibungskoeffizient μ. Der Typ ist entweder eine Ebene($type_c = 0$) oder eine Kugel($type_c = 1$). Die Kollisionen für Kugeln und Ebenen lassen sich mit diesen Parametern detektieren und behandeln.

Die Positionen der Kollisionsobjekte werden mit folgendem Compute-Shader aktualisiert, der über die Zahl der Kollisionobjekte aufgerufen wird:

Quellcode 9: Updaten der Position der Kollisionsobjekte

```
1  #version 440
2  uniform float dt;
3  layout(local_size_x =1, local_size_y =1,local_size_z =1)in;
4
5  layout(std140, binding = 8) buffer pPosMass {
6      vec4 cx[ ];
7  };
8  layout(std140, binding = 9) buffer pVelVolume {
9      vec4 cv[ ];
10 };
11
12 void main(void){
13     uint pI = gl_GlobalInvocationID.x; // Index des Kollisionsobjekts.
14     cx[pI].xyz += dt *  cv[pI].xyz; // Update der Position.
15 }
```

In Zeile 14 wird die Position eines Kollisionsobjekt anhand seiner Geschwindigkeit aktualisiert.

Der nachfolgende Algorithmus 10 behandelt die Kollision von Schnee mit Kollisionsobjekten [SSC+13].

In Zeile 5 wird abhängig vom Typ getestet, ob eine Kollision stattfindet. Falls das Partikel und das Objekt sich voneinander entfernen, passiert nichts (Zeile 9). Andernfalls wird eine Behandlung nötig. Der tangentiale Anteil der Geschwindigkeit wird in Zeile 10 berechnet. Wenn er die Reibung überwinden kann, wird kinetische Reibung anhand der Gleichung in Zeile 18 angewandt. Andernfalls wird die relative Geschwindigkeit des Partikels wie in Zeile 15 gestoppt. Die neue Partikelgeschwindigkeit entsteht aus der Geschwindigkeit des Kollisionsobjektes und der relativen Geschwindigkeit nach Anwendung der Reibung. Zeile 1 ist eine Schleife über alle Kollisionsobjekte. Besser wäre es diese zu parallelisieren. In der Schleife wird kein Schreibzugriff vollzogen, daher ist die Performance bei wenigen Kollisionsobjekten nicht beeinflusst. [SSC+13]

Die Kollisionsbehandlung wird genau zweimal ausgeführt. Einmal auf den Git-

terknoten, nachdem die neue Gittergeschwindigkeit $\boldsymbol{v}_i^{n+1}$ festgestellt wurde; das zweite Mal, wenn die Partikel ihre neuen Geschwindigkeiten $\boldsymbol{v}_p^{n+1}$ erhalten. Dies behebt die Ungenauigkeiten zwischen Gitter und Partikel durch das Interpolieren. [SSC+13]

Quellcode 10: Algorithmus für die Berechnung von Kollisionen zwischen Partikeln und Rigid-Bodies

```
1  for(int i = 0 ; i<gNumColliders; i++){
2    vec3 p = cx[i].xyz;
3    vec3 n = cn[i].xyz;
4    vec3 particlePos = pxm[pI].xyz; // Position des Partikels bzw. ↩
         Gitterknotens.
5    if(collides(particlePos,i,n)){ //Testet, ob eine Kollision vorliegt.
6      vec3 vco = cv[i].xyz;
7      vec3 vrel = vpn1 - vco;
8      float vn = dot(vrel,n);
9      if(vn<0.0f){ //Falls sich die Objekte voneinander entfernen, ↩
           unternehme nichts.
10       vec3 vt = vrel - n*vn; //Tangentialer Anteil der Geschwindigkeit.
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); // Reibung kann nicht ueberwunden werden.
16       }
17       else{
18         vrelt+=  muvn*vt/(lengthvt); // Dynamische Reibung.
19       }
20       vpn1 = vrelt + vco; //Aktualisieren der Geschwindigkeit.
21     }
22   }
23 }
```

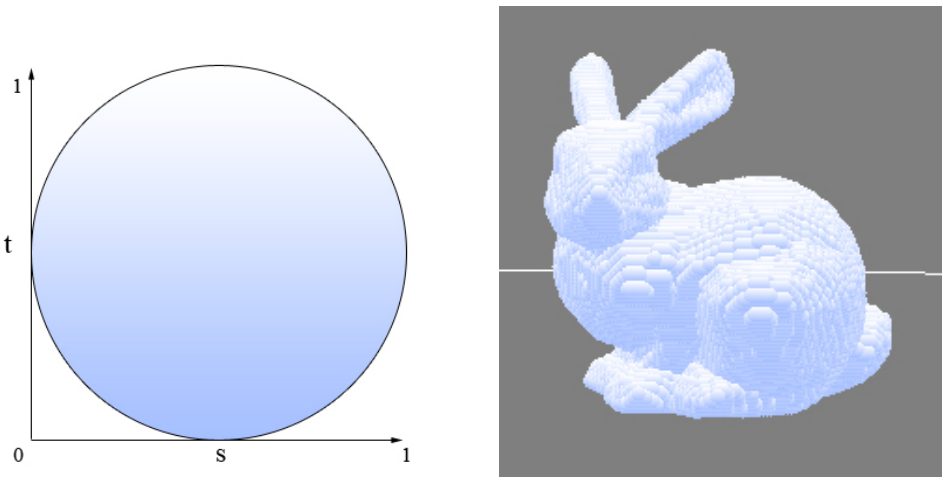
Dieses Kapitel behandelt die Detektion und Behandlung von Kollisionsobjekten mit Schneepartikeln. Abbildung 7 zeigt die Interaktion von Schnee mit drei Kugeln und sechs Ebenen. Kollisionsobjekte kollidieren nicht miteinander.

4.5 Rendering

Das Rendering ist einfach gehalten. Es kann beispielsweise durch ein Gitter-Rendering ersetzt werden.

Mit **glDrawArrays**(GL_POINTS,0,(particles)->size()); werden die Positionen $\boldsymbol{x}_p$ der Partikel gerendert. Das Shader-Storage-Buffer-Objekt von $\boldsymbol{x}_p$ lässt sich als Vertex-Buffer-Objekt interpretieren. Quellcode 11 zeigt die entstehenden Shader. Die Bits GL_POINT_SPRITE und GL_VERTEX_PROGRAM_POINT_SIZE werden mit glEnable für das Aktivieren von gl_Pointsize (Zeile 8) gesetzt. In Zeile 13 wird ein Verlauf über den Punkt gelegt. Temporär wird der Punkt in den Ursprung gelegt und ein Kreis mit Radius 0.5 ausgeschnitten. Abbildung 8 (a) zeigt das Ergebnis.

Das Rendering erlaubt die Simulation visuell darzustellen. Abbildung 8 (b) zeigt das Rendering eines Hasen nach dieser Methode.



(a) (b)

Abbildung 8: *Partikelrendering: (a) Visualisierung des Fragment-Shaders (b) Rendering des Stanford-Hasen mit 183029 Partikeln*

Quellcode 11: Primitives Rendering für ein Schnee-Partikel

```
1  //Vertex Shader
2  #version 440
3  layout (location = 0) in vec4 Position;
4  uniform mat4 gMVP;
5  void main(void)
6  {
7      gl_Position = gMVP* vec4(Position.xyz,1.0);
8      gl_PointSize = 8; // Lege Groesse des Partikels fest.
9  }
10
11 //Fragment Shader
12 void main(void){
13   gl_FragColor.rgba= vec4(1.0-gl_PointCoord.t*0.35,1.0f-gl_PointCoord.t↩
         *0.25,1.0,1.0); //Lege Verlauf ueber die Partikelkoordinaten.
14   if((gl_PointCoord.t*2.0-1.0)*(gl_PointCoord.t*2.0-1.0)+ (gl_PointCoord.↩
         s*2.0-1.0)*(gl_PointCoord.s*2.0-1.0)>1.0f){
15     discard; //Schneide einen Kreis aus dem viereckigen Punkt aus.
16   }
17 }
```

4.6 Ablaufdiagramm

Skizziert wird in diesem Kapitel der Ablauf der Simulation in der Anwendung. Sie nimmt stark Bezug auf die bestehende Skizze nach Kapitel 3.4. Abbildung 9 visualisiert den Ablauf mithilfe eines Aktivitätsdiagramms.

Der Ablauf ist in drei Phasen aufgeteilt: Die initiale Berechnung der Dichte, den Simulationsschritt und den Renderschritt. Die Compute-Shader sind aufgeteilt nach der Art des Aufrufs aus Kapitel 4.3. Die Wahl zwischen Simulationsschritt und Renderschritt wird nach Kapitel 4 getroffen. Zwischen jedem gezeigten Shader findet ein Synchronisationsschritt statt.

Ein Punkt, der nicht besprochen ist, ist das Zurücksetzen von Buffern. Jeder Buffer, der durch Gitter-basierte Operationen aufsummiert wird, wird auf null gesetzt. Dies passiert, um die Buffer für die nächste Aufsummierung zurücksetzen. Im Ablauf geschieht dies einmal initial und einmal in jedem Simulationsschritt für den zusammengesetzten Buffer aus $\boldsymbol{v}_i$ und m_i und für f_i mit *Gitterparameter zurücksetzen*. Der Buffer für $\boldsymbol{v}_p^{n+1}$ und die drei Buffer für $\nabla \boldsymbol{v}_p^{n+1}$ werden am Ende des Simulationsschritts zurückgesetzt. Zu diesem Zeitpunkt wird auf der richtigen Größe (Partikeln) operiert.

Die Berechnung des Partikelvolumens V_p ist in drei Shader aufgeteilt. Die *Berechnung der Gittermasse* m_i kann nicht in derselben Aufsummierung passieren wie für die *Partikeldichte* ρ_{p0}. Eine Synchronisierung muss zwischendurch stattfinden. Da die Partikeldichte im Nenner der Berechnung für das Volumen steht($V_p^0 = m_p/\rho_p^0$), wird das Volumen erst berechnet, wenn es benötigt wird (siehe *Volumen berechnen*).

Die Normalisierung der Gittergeschwindigkeit $\boldsymbol{v}_i$ mit m_i kann erst in späteren Shadern geschehen. Signalisiert wird dies im Ablaufdiagramm mit *Gittergeschwindigkeit normalisieren*.

Beim Dividieren durch Partikel-Dichte oder Gittermasse wird überprüft, ob der Wert positiv und ungleich null ist. Bei Floating-Point-Genauigkeit und speziell bei den verwendeten Integern können diese, wenn sie sehr gering gewichtet werden, den Wert null annehmen.

Die Berechnung der neuen Partikel-Geschwindigkeit lässt sich auch anders für eine verminderte Anzahl von Speicherzugriffen aufschreiben: Die *Berechnung der temporären Partikel-Geschwindigkeit* $\boldsymbol{v}_{\text{TEMP}p}^{n+1} = \sum_i (1-\alpha)\boldsymbol{v}_i^{n+1} w_{\boldsymbol{ip}}^n + \alpha(\boldsymbol{v}_i^{n+1} - \boldsymbol{v}_i^n) w_{\boldsymbol{ip}}^n$. Die *neue Partikel-Geschwindigkeit* ergibt sich mit $\boldsymbol{v}_p^{n+1} = \alpha \boldsymbol{v}_p^n + \boldsymbol{v}_{\text{TEMP}p}^{n+1}$. Dies reduziert die Anzahl der nötigen Schreibzugriffe.

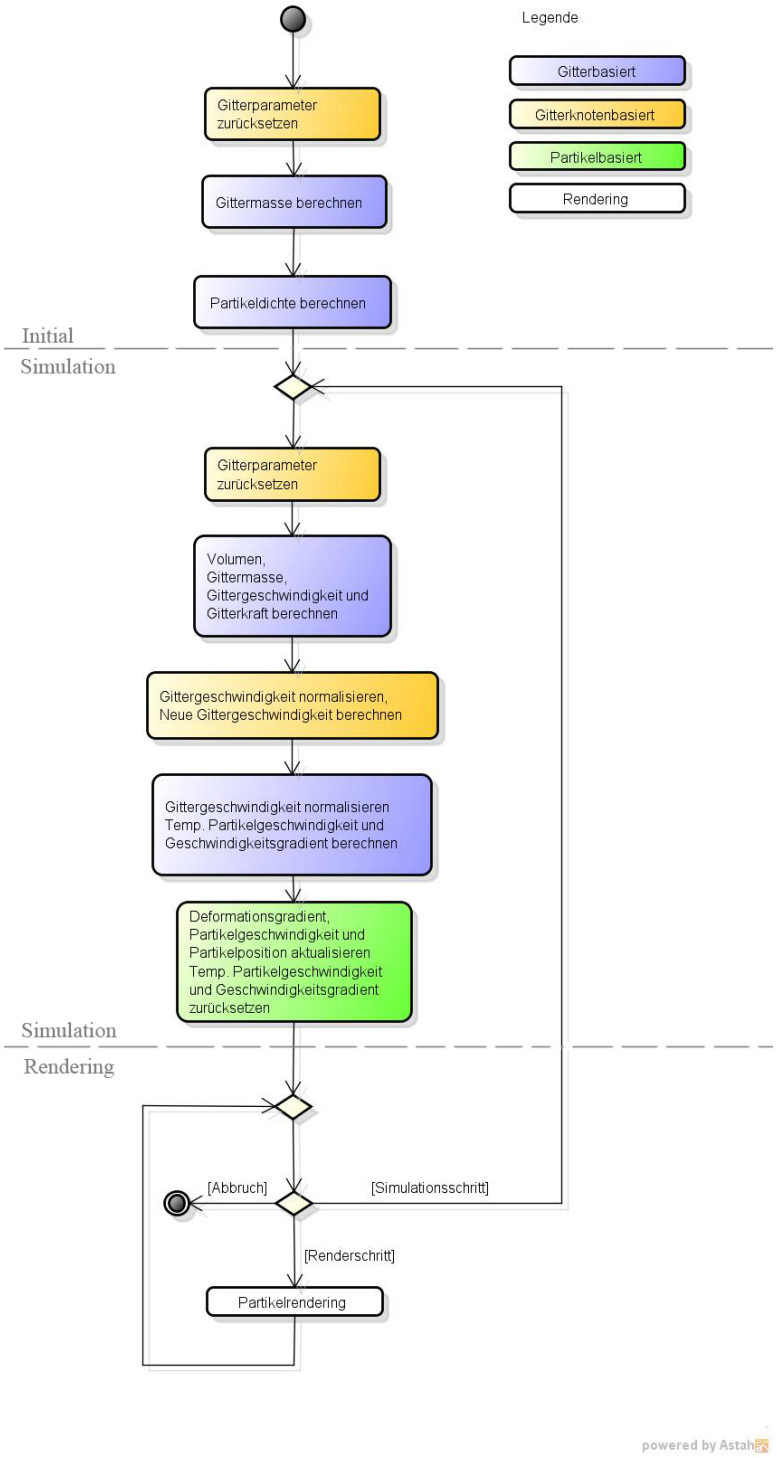


Abbildung 9: *UML-Aktivitätsdiagramm: Überblick über die Abfolge der OpenGL-Shader für die Simulation von Schneepartikeln.*

5 Evaluation

In diesem Kapitel wird das vorgestellte Modell unter den Kriterien Kompatibilität, Performance und Visuelle Qualität bewertet. 5.1 zeigt auf unter welchen Bedingungen die Anwendung ausgeführt werden kann. Das Kapitel 5.2 behandelt die Performance mit größer werdender Partikel- und Gitterknotenanzahl. In 5.3 wird das visuelle Ergebnis und die Beeinflussung durch die verschiedenen Parameter bewertet.

5.1 Kompatibilität

Für die Unterstützung der Compute-Shader ist OpenGL 4.3 notwendig. Die Anwendung ist getestet auf einer AMD Radeon HD7850 mit 2 Gigabyte(GB) RAM und einer Nvidia GTX 970 mit 4GB RAM. Beide sind im Stande die Anwendung in vollem Umfang auszuführen. Für die Nutzung der Implementation auf einer AMD Radeon HD7850 muss eine kleine Änderung im Code vollzogen werden. Die HD7850 unterstützt genau 16 Bindungen von Shader-Storage-Buffer-Objekt. Diese Anzahl lässt sich abrufen mit dem Befehl:

glGetIntegerv(0x90DD,GLint* param);

Die Bitmaske 0x90DD steht für die Variable MAX_SHADER_STORAGE_BUFFER_BINDINGS[67]. In der Implementation werden derzeit genau 17 Bindungen verlangt. Das Zusammenlegen von Buffern bereitet kein Problem. Ein Beispiel wäre die drei $\Delta \boldsymbol{v}_p^{n+1}$ Buffer zusammenzulegen.

Durch die Nutzung von atomaren Funktionen auf Integern und die Nutzung von OpenGL ist die Kompatibilität auf Grafikkarten von AMD und Nvidia unter den beschriebenen Bedingungen gewährleistet.

5.2 Performance

Zur Beurteilung der Performance werden die Simulationsschritte (siehe Abbildung 9) pro Sekunde gemessen. Dazu wird der Algorithmus aus 4 für die Auswahl von Render-und Simulationsschritten ausgelassen. Anstatt dessen werden jeweils ein Renderschritt und ein Simulationsschritt vollzogen. Messungen zeigen dasselbe Ergebnis, nur fällt das Justieren der STEP_DT bis zur Auffindung des maximalen Werts weg. Die Simulationsschritte pro Sekunde lassen sich, beispielsweise mit Fraps[68], ablesen. Die Anzahl der Renderschritte ist gleich der Anzahl der Simulationsschritte. Zusätzlich wird vertikale Synchronisation in der Nvidia-Systemsteuerung ausgestellt, damit die FPS nicht ausgebremst werden. Dies ist bei der hier vorgenommen Anpassung notwendig: Die Anzahl der Simulationsschritte ist ansonsten unter Einhaltung der Rahmenbedingungen unabhängig und konstant.

[67] Einsicht am 11.03.2015 unter
https://www.opengl.org/registry/specs/ARB/shader_storage_buffer_object.txt

[68] Fraps. Einsicht am 11.03.2015 unter http://www.fraps.com/

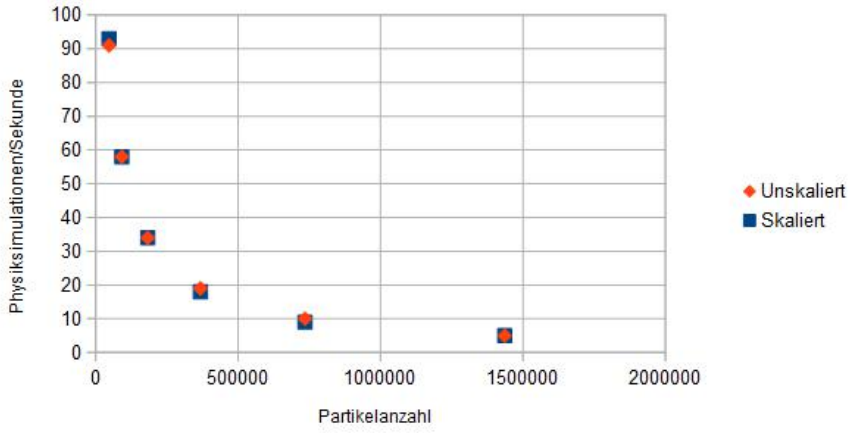


Abbildung 10: *Die Performance der Anwendung bei größer werdender Partikelanzahl. Gitterknotenanzahl: 8 Millionen. Skaliert: Immer 8 Partikel pro Zelle. Unskaliert: Partikel pro Zelle verdoppeln sich pro Schritt beginnend mit 4 bei 47063 Partikeln. Datensatz ist der Stanford-Hase. Grafik erstellt mit https://www.openoffice.org/de/.*

Die Simulation ist auf einer Nvidia GTX 970 mit 4GB RAM durchgeführt. Der mit Partikeln aufgefüllte Stanford-Hase[69] ist der verwendete Datensatz. Abbildung 10 zeigt die Anzahl der maximal möglichen Simulationsschritte pro Sekunde bei steigender Anzahl der Partikel. Bei der ersten Testreihe wird die Partikelanzahl erhöht, ohne die Größe des Hasen zu verändern. Bei der zweiten Reihe ist die Größe des Hasen angepasst, sodass sich immer acht Partikel in einer Gitterzelle befinden. Die Unterschiede sind minimal, jedoch kann im Laufe der Simulation die Größe enorm zunehmen, wenn die Parameter extrem gewählt werden, was die Performance beeinträchtigt.

Entscheidend ist die Anzahl der Schreibzugriffe. Diese skaliert mit der Anzahl der Partikel und mit der Anzahl der Gitterknoten (siehe Abbildung 11) mit. Für die Performance gilt: Die Berechnungen, die innerhalb eines Shaders passieren, sind in der Regeln nicht so ausschlaggebend, wie die Minimierung der Schreibzugriffe.

Größere Buffer, wie zum Beispiel beim Zusammenlegen von Buffern, wie es in 5.1 vorgestellt wird, haben deswegen keine Auswirkung auf die Performance. Der Unterschied bei separaten Shadern, die auf der gleichen Größe arbeiten, ist geringfügig messbar gewesen. Einige Berechnungen werden zweimal ausgeführt. Die Anzahl der Schreibzugriffe unterscheidet sich nicht.

Zugleich ist der Kontakt von CPU und GPU, während der main-Schleife zu ver-

[69] Einsicht am 11.03.2015 unter http://graphics.stanford.edu/data/3Dscanrep/#bunny

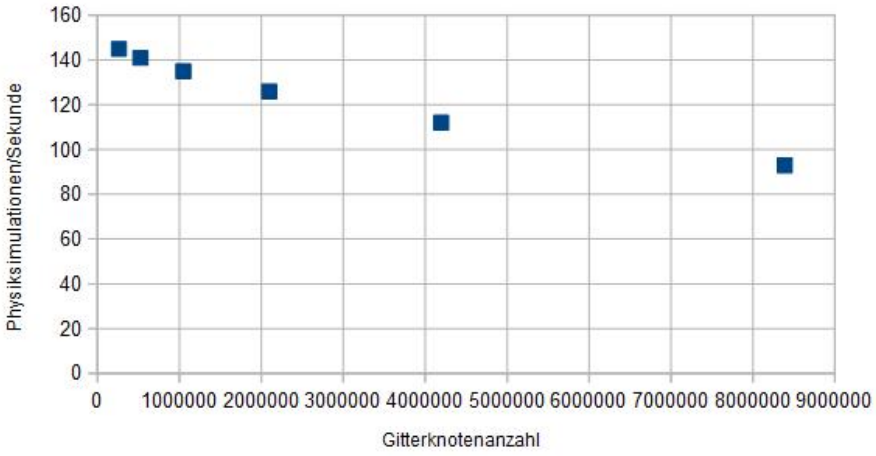


Abbildung 11: *Die Performance der Anwendung bei größer werdender Gitterknotenanzahl, 47063 Partikel. Datensatz ist der Stanford-Hase. Grafik erstellt mit https://www.openoffice.org/de/.*

meiden. Das Verändern, Binden oder Auslesen von Buffern durch die CPU bremst die GPU aus. Einen Ansatz, der die Gitteroperationen von der Shared-Memory[70] lokaler Arbeitsgruppen profitieren lässt, konnte nicht gefunden werden. Die Kommunikation zwischen Partikeln und Gitter ist in diesem Sinne komplex.

Das Gitter kann im Verlaufe der Simulationen verschoben werden. Alle Partikel, die sich außerhalb des Gitters befinden, müssen nicht mehr den gesamten Simulationsprozess durchlaufen. Sie bewegen sich nicht mehr. Dies würde Zeit sparen.

Zusätzlich war beim Vergleich zwischen atomaren Operationen auf Floating-Point Zahlen und atomaren Operationen auf Integern kein Unterschied festzustellen. Der Vergleich kann aus den in 3.6 beschriebenen Bedingungen nur auf Nvidia-Grafikkarten bezogen werden.

Der semi-implizite Zeitschritt berechnet die Änderungsrate der Kraft $\boldsymbol{f_i}$ mit ein. Damit werden 50 Mal schnellere Δt möglich. Der Vergleich mit der Implementierung des expliziten Zeitschritts auf der Grafikkarte ist schwer. Angenommen der Fall des Schneeballs wird auf 30 Frames pro Sekunde gerendert und der Frame-Begriff in der Tabelle von [SSC+13] entsprechen einem gerenderten Frame. Abgelesen aus Abbildung 10 ergibt die Performance bei 300000 Partikeln mit einem 200×200×200 Gitter 20 Physiksimulationen pro Sekunde mit einer $\Delta t = 1.0{\times}10^{-5}$s. Ein Frame würde 3.7 Minuten dauern. [SSC+13] verwendet ein

[70] Einsicht am 11.03.2015 unter
https://www.opengl.org/wiki/Compute_Shader#Shared_memory_coherency

größeres Gitter von 600×300×600 und braucht 5.2 Minuten für die Berechnung eines Frames. Dieser Vergleich ist aber aus der fehlenden Übereinstimmung vage. Es tendiert aber dahin, dass mit der Implementation des Semi-Impliziten Zeitschritts auf der GPU die Simulation schneller berechnet wird.

Die Performance ist abhängig von der Anzahl und Streuung der Schreibzugriffe. Sie steigen mit der Anzahl und Verteilung der Partikel und der Größe des verwendeten Gitters.

5.3 Visuelle Qualität

Die Kriterien für die Bewertung der visuellen Qualität sind die in 3.3 aufgestellten Eigenschaften: Volumenerhaltung, Elastizität, Plastizität und Bruch.

In den Abbildungen 12 und 13 sind die Einflüsse der verschiedenen Parameter auf die Simulation dargestellt. Die Ausgangssituation ist jeweils in der obersten Reihe dargestellt. Bis auf das Bild oben links ist jedes der Bilder nach 1.8 Sekunden nach beschleunigter, realer Laufzeit aufgenommen worden. Die verwendeten Parameter in der Ausgangssituation sind $h = 0.05m$, $\Delta t = 1 \times 10^{-4}s$, $E = 1 \times 10^6$ Pa, $t = 1.8s$, $\rho = 400$kg/m^3, $\nu = 0.25$, $\xi = 10$, $\theta_c = 2.5 \times 10^{-2}$, $\theta_s = 7.5 \times 10^{-3}$, 200×200×200 Gitter und 183029 Partikel. Die jeweilige Änderung in den darunter liegenden Bildern, ist in der oberen, linken Ecke festgehalten.

Die **Volumenerhaltung** ist durch die Materialkennwerte Youngscher Modul, Dichte und Poissonzahl bestimmt. Youngscher Modul und Dichte korrelieren, wie die Analyse von [Sig06] zeigt. Benutzt wird mit E=1 $\times 10^6$ Pa ein Wert, der zwischen dem von [SSC$^+$13] und [Sig06] vorgeschlagenem liegt. Wie in der Mitte in Abbildung 12 gezeigt bestimmt er die Resistenz des Körpers zur Deformation. Das Volumen der Körper bleibt erhalten, da der Körper sich plastisch weniger stark verformt. Die Verringerung der Dichte, wie unten links, hat einen ähnlichen Effekt. Da sich Dichte und Youngscher Modul direkt entgegen stehen, erreicht der relativ zur Dichte hohe Youngscher Modul mehr Einfluss.

Die Poissonzahl ist der zweite Kennwert, der zur Erhaltung des Volumens beiträgt. Die Auswirkungen einer hohen Poissonzahl können in der Abbildung unten rechts betrachtet werden. Bei einer Poissonzahl von 0.4 sind die Querspannungen zusammen beinahe so stark wie die Normalspannung. Das Material wird automatisch bei starker Einwirkung der Normalspannung nach außen/innen gedrückt, was das Volumen bei Einfüssen von Spannungen erhält. Kompressible Materialien besitzen niedrigere Poissonzahlen. In der Referenz und dem Bild unten links ist die in [Sig06] genutzte Abhängigkeit von Dichte und Poisson eingehalten.

Die **Elastizität** ist ein Zusammenspiel von kritischer Streckung θ_s und kritischer Stauchung/Kompression θ_c. Abbildung 13 zeigt die Einflüsse dieser Parameter. Bei der Kompression des Hasen entstehen Querspannungen, die den Körper nach außen drücken. In der Referenz ist die kritische Streckung groß genug und verhindert die plastische Verformung in diese Richtung. Die Auswirkungen einer höheren Kompression sind unten links in der Abbildung gezeigt. Der Körper hat eine höhere Resistenz sich bei Normalspannungen zu verformen. Er bewegt sich

weiter in die Ausgangsposition zurück.

Die **Plastizität** ist das Phänomen, das entsteht, wenn die elastischen Kräfte überwunden werden. Die Verringerung der kritischen Streckung (Mitte, links in der Abbildung) hält den Körper weniger gegen die Querspannungen zusammen. Die Elastizität leidet darunter und kann die Streckung erst nach längerer, plastischer Verformung aufhalten. Der Härtekoeffizient schwächt/stärkt die Auswirkungen der plastischen Deformation. In den Abbildungen unter der Referenz können diese Auswirkungen gesehen werden.

Der **Bruch** ist deutlich in Abbildung 12 in den beiden, mittleren Bildern sichtbar. Die Hasenohren stehen weit ab vom Kopf und erfahren als erste den Bruch. Auch der Kopf ist zu schwer und zu weit vorne platziert. Das Ergebnis ist, dass der Körper an diesen Stellen bricht. In der linken von beiden Abbildungen, fällt das linke Ohr auf den Rücken des Hasen. Danach rollt er sich seitlich nach hinten ab, bis er auf dem Boden landet.

Der Youngsche Modul und die Poissonzahl haben aus den oben genannten Gründen Einfluss auf das elastische und plastische Verhalten. Genauso können extreme Parameter für kritische Kompression und Stauchung Auswirkungen auf die Volumenerhaltung haben. Eine eindeutige Differenzierung ist nicht möglich. Empfehlenswert ist es zuerst den Youngschen Modul, Dichte und Poissonzahl festzulegen. Anschließend werden kritische Streckung und Kompression variiert, um das Ergebnis zu verfeinern. Speziell für die Erscheinung von Schnee lassen sich folgende Beobachtungen machen: Höhere kritische Streckung und Kompression lassen das Ergebnis nass und klebrig erscheinen. Der Schnee formt Brocken und weist Bruchkanten auf. Das Gegenteil macht den Schnee trocken und loser. Der Schnee erscheint wie ein Puder. Visuell dargestellt in Abbildung 14.

Die Kollision zwischen zwei Schneebällen ist in Abbildung 15 gezeigt. Die Parameter sind leicht modifiziert: $E = 5 \times 10^6$ Pa, $\nu = 0.3$, $\xi = 30$, $\theta_c = 2.5 \times 10^{-2}$, $\theta_s = 2.0 \times 10^{-3}$, 150×100×100 Gitter und 32768 Partikeln.

Wie in Abbildung 12 veranschaulicht, hat ein höhere Youngscher Modul eine höhere Resistenz zur Deformation, was den Schnee eisiger macht. Das Gegenteil ist matschiger Schnee, was in der Abbildung 12 in der Referenz zur Geltung kommt.

Die Nutzung einer Δt von $1 \times 10^{-4} s$ lässt die Simulation sehr elastisch erscheinen. Empfohlen ist eine Δt von $1 \times 10^{-5} s$, was dem Deformationsgradient mehr Zeit zur Anpassung gibt. Die Nutzung dieser geringeren Δt ist aufgrund der Konvertierung in Integer nicht möglich. Der Bereich der darzustellenden Geschwindigkeiten wird zu groß. Mit der Nutzung des Semi-Impliziten Zeitschritts werden Δt von $0.5 \times 10^{-3} s$ möglich, die dieses Problem beheben würden.

Lange Laufzeiten weisen einen weiteren Fehler auf: Die Kompression, ausgelöst durch die Gravitation, bringt über lange Laufzeiten mehr Kraft auf als die Streckung aufbringen kann. Der Körper bläht sich auf. Dies kann durch den im Vergleich zu [SSC+13] höheren Youngschen Modulus ausgelöst worden sein. Ohne diese Anpassung weisen die Körper kaum Resistenz zur Schwerkraft auf und werden auf den Boden der Simulation gedrückt. Die Ungenauigkeit bei der Be-

rechnung der Singulärwertzerlegung könnte ein weiterer Grund sein.

Die visuelle Qualität des Renderings ist sehr limitiert. Es findet weder eine Beleuchtung statt noch ermöglichen Schatten eine bessere räumliche Orientierung.

Die visuelle Erscheinung der Simulation lässt sich durch wenige Parameter in verschiedene Richtungen lenken. Eine kleine Änderung kann schon Auswirkungen im Ergebnis erzielen. Die Nutzung extremer Parameter hat Auswirkungen auf die grundlegenden Eigenschaften und macht das Ergebnis schnell ungläubig. Die Differenzierung und Verfeinerung der Parameter ist schwierig und wird durch die lange Laufzeit (10 Minuten für 1.8 Sekunden) nicht begünstigt.

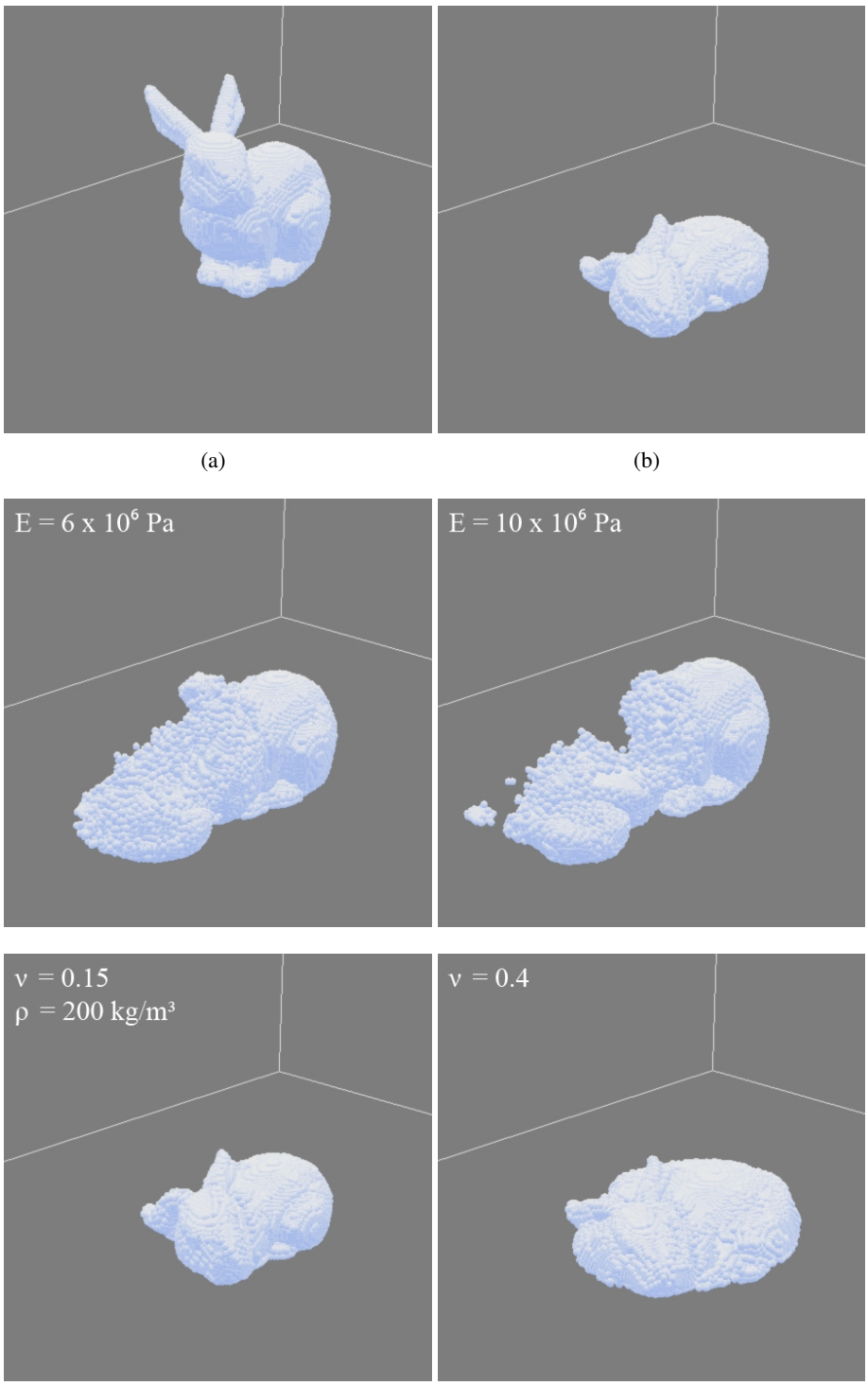


Abbildung 12: *Ein Schneehase fällt auf eine ebene Fläche. Verschiedene Parameter werden angepasst, um das Ergebnis zu beeinflussen. (a) Ausgangssituation der Simulation (b) Referenzbild.*

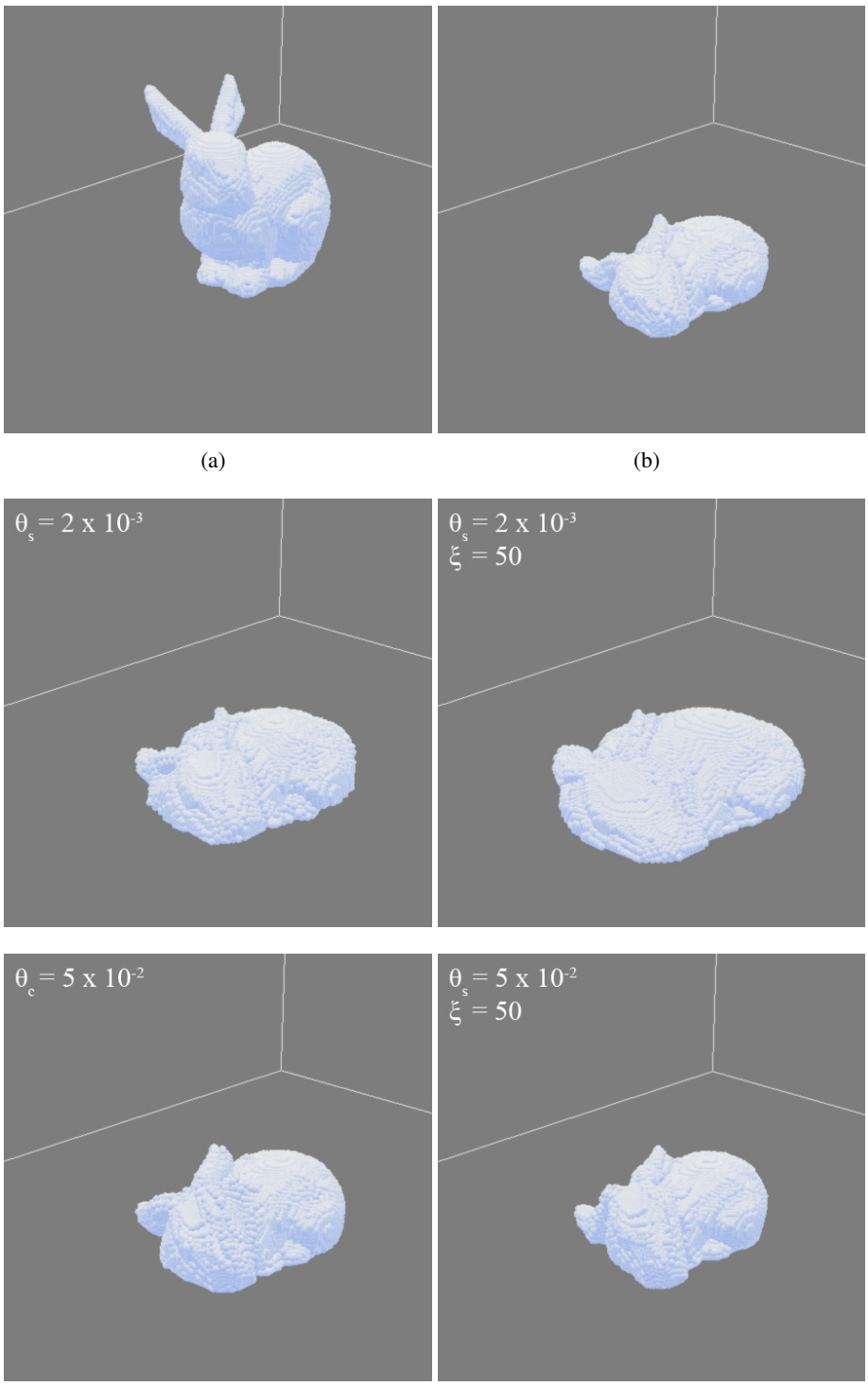
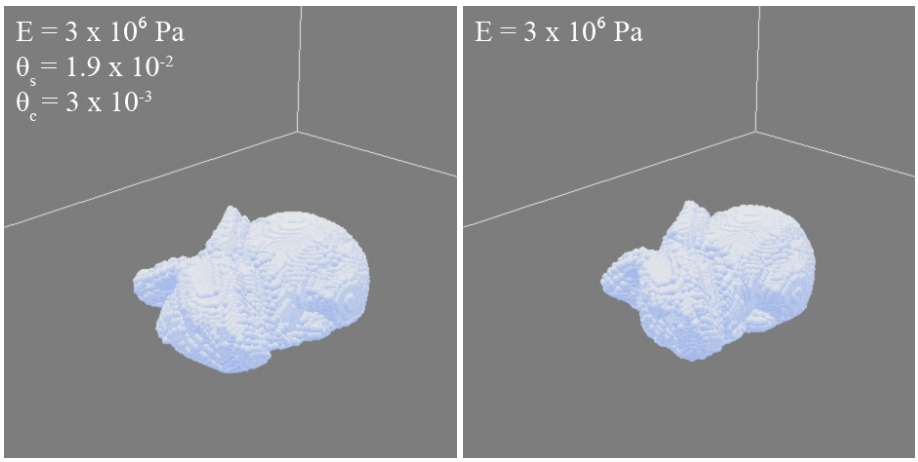
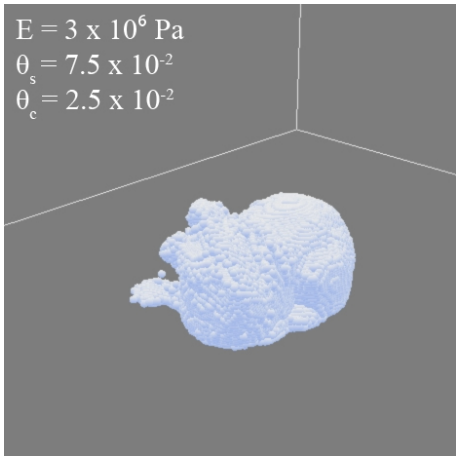


Abbildung 13: *Ein Schneehase fällt auf eine ebene Fläche. Verschiedene Parameter werden angepasst, um das Ergebnis zu beeinflussen. (a) Ausgangssituation der Simulation (b) Referenzbild.*



(a) Trockener Schnee (b) Referenz



(c) Nasser Schnee

Abbildung 14: *Ein Schneehase fällt auf eine ebene Fläche. Das Ergebnis zeigt trockenen und nassen Schnee.*

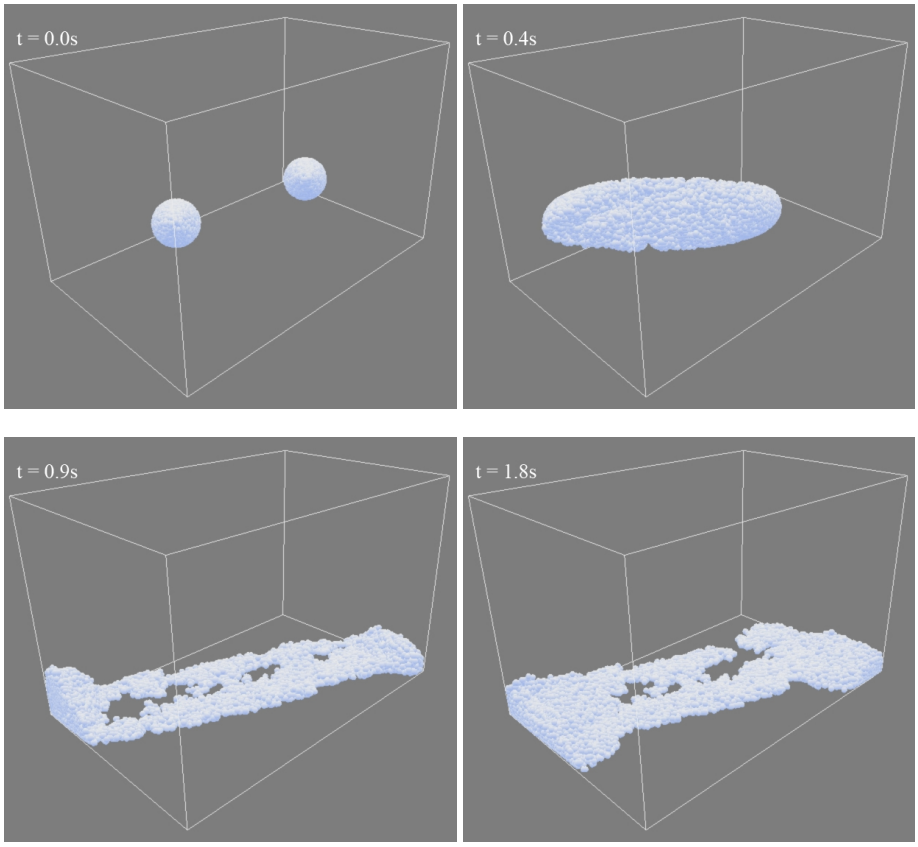


Abbildung 15: *Kollision zwischen zwei Schneebällen.*

6 Fazit und Ausblick

Vorgestellt wurde die Simulation von dynamischem Schnee unter Verwendung neuester GPGPU-Techniken. Gezeigt wurde die Materie-Punkt-Methode unter Verwendung eines für Schnee entwickelten Materialmodells nach [SSC+13]. Es ist in der Lage die Verschiedenheit von Schnee darzustellen. Darunter fällt nasser, trockener, eisiger und matschiger Schnee. Mit der Implementierung wurde die Methodik auf der GPU unter Nutzung der von OpenGL bereitgestellten Compute-Shader verwirklicht. Dies erlaubt die effiziente, parallele Ausnutzung der Grafikkarte. Ein direkter Vergleich ist vage: In etwa werden die gleichen Zeiten ohne Implementation des semi-impliziten Zeitschritts auf der GPU erreicht.

Der semi-implizite Zeitschritt ermöglicht die Nutzung einer höheren Δt von $0.5{\times}10^{-3}s$. Das beschleunigt sowohl die Laufzeit der Simulation als auch die genauere Berechnung für ein weniger elastisches Erscheinen der Simulation, wie in 5.3 erklärt. Eine Erweiterung ist es einzelne Partikel, die in der Umgebung von zwei keinen Nachbar besitzen, von der Berechnung über das Gitter auszunehmen. Diese Partikel profitieren nicht von der Berechnung über das Gitter, da sie zu diesem Zeitpunkt zu keinem Körper gehören. Dies verbessert die Performance. Das Modell erfordert weiteres Testen. Der entstehende Fehler bei langen Laufzeiten limitiert die Anwendung. Er fällt erst nach tausenden von Simulationsschritten statt. Dies entspricht in etwa sechs bis acht Sekunden realer Zeit. Das Rendering lässt sich ersetzen durch ein Gitter-Rendering. Ein Beispiel ist die Nutzung eines volumetrischen Path-Tracer[71] oder das Nutzen des Marching-Cubes-Verfahren zur Erstellung von Polygonen, die beleuchtet werden. Das Rendering Verfahren sollte sowohl Beleuchtung und Schatten als auch das visuelle Spektrum von Schnee ermöglichen. Eine Anbindung an einer vorhandene Physik-Engine könnte die in 4.4 besprochene Kollisionsdetektion mit Rigid-Bodies ausweiten. Ein Videoexporter bringt eine bessere Performance, indem gerendert wird, wenn wirklich ein Bild benötigt ist. Im Moment werden viele Bilder durch die Beschleunigung des Videos aussortiert. Möglich wäre das Rendern in ein Frame-Buffer-Objekt zur Erstellung einzelner Bilder. Ein Szenen-Exportierer könnte ein Abbild der Buffer speichern und laden. Von diesem Zeitpunkt ließe sich weitersimulieren. Ein Interface zur Erstellung einer Szene würde den Umgang mit der Simulation vereinfachen. Objekte ließen sich in der Welt platzieren, Geschwindigkeiten zuweisen, um anschließend die Szene zu simulieren.

Die Simulation von Physik erfordert Modelle, die effiziente und visuell ansprechende Ergebnisse liefern. GPGPU-Techniken ermöglichen die hohe Parallelisierung von Physik, die diese Modelle ausnutzen können. Die Materie-Punkt-Methode ist die erste Technik, die die Simulation von Schnee ermöglicht. Diese Implementierung zeigt unter den beschriebenen Limitierungen, dass die Materie-Punkt-Methode effizient auf der GPU simuliert werden kann, um die Dynamik von Schnee darzustellen.

[71] Path-Tracer: globale Beleuchtung durch Schießen von Strahlen in die Szene

Anhang

A Grafiken

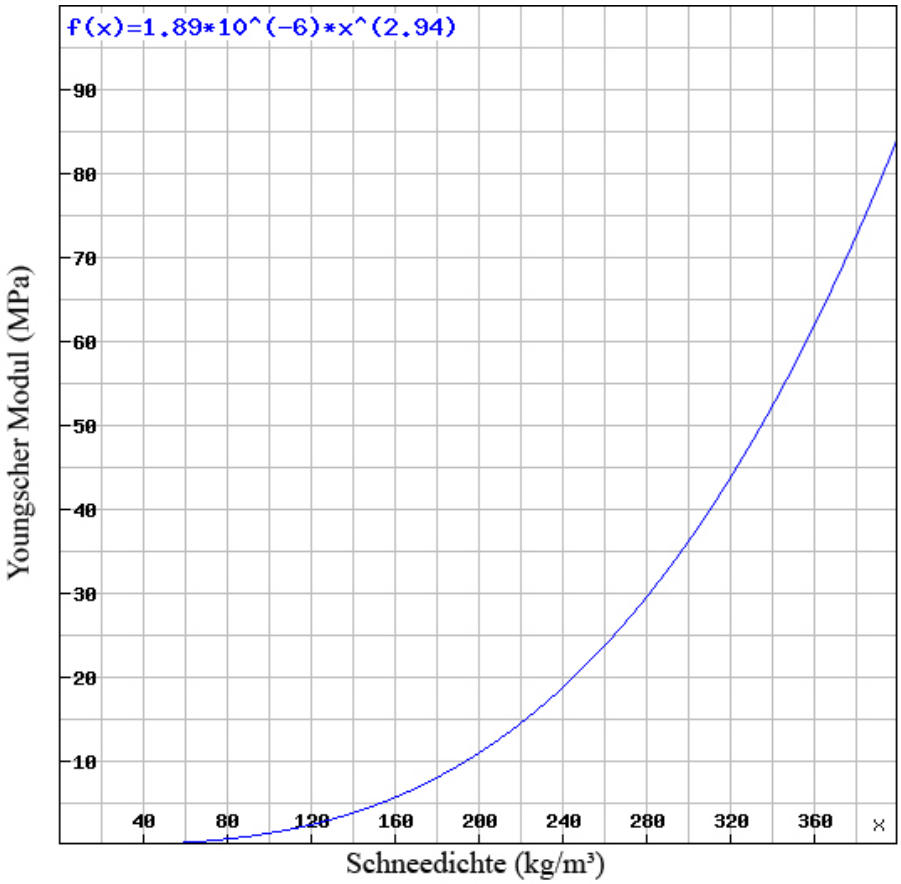


Abbildung 16: *Abhängigkeit zwischen Youngschem Modul und Schneedichte nach [Sig06].* Grafik erstellt mit http://rechneronline.de/function-graphs/.

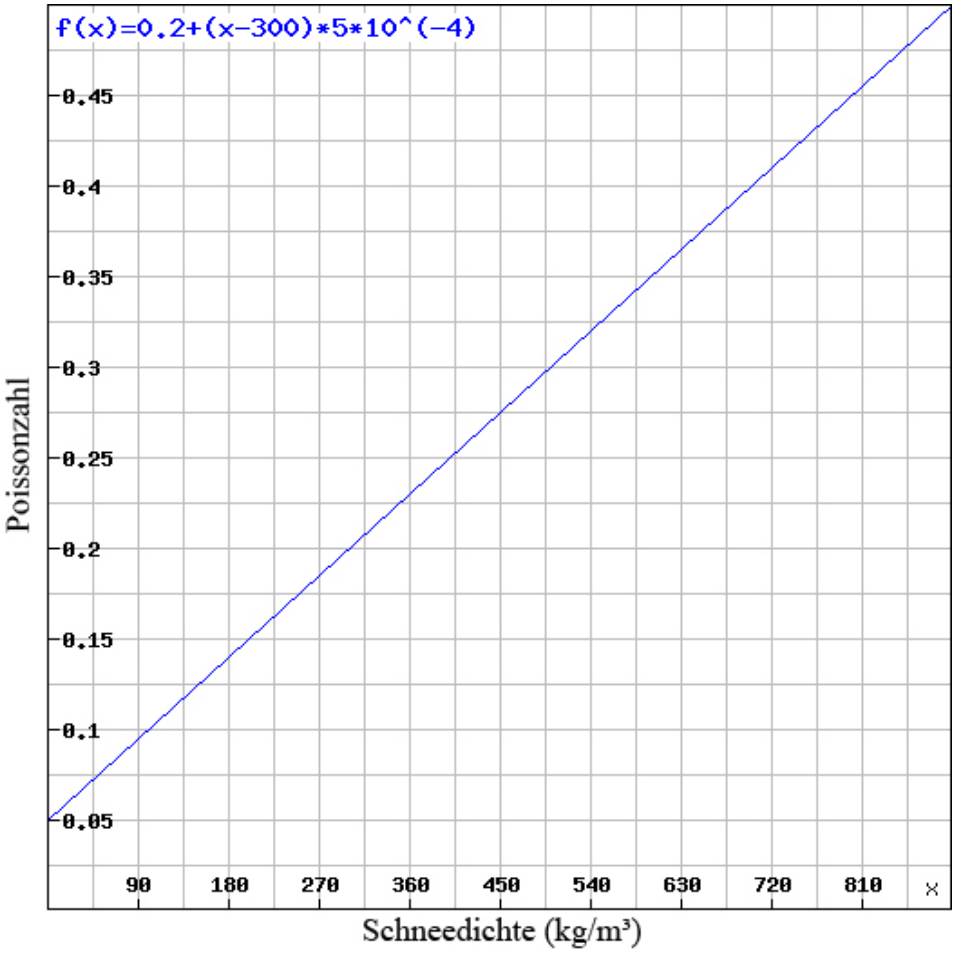


Abbildung 17: *Abhängigkeit zwischen Poissonzahl und Schneedichte nach [Sig06].* Grafik erstellt mit http://rechneronline.de/function-graphs/.

Abbildungsverzeichnis

1 *Ein simulierter Schneehase.* 1
2 *Spannungen im dreidimensionalen Raum.* 5
3 *Scherdehnung von Objekten.* 6
4 *Verformung eines Körpers in einem 2D-Koordinatensystem.* 10
5 *Form der Schneekristalle.* Quelle: [Lib01] 13
6 *Überblick der Materie-Punkt-Methode. Schritte in der oberen und unteren Hälfte rechnen auf Partikeln, während die Schritte zwischen den Hälften Berechnungen auf dem Gitter vollziehen.* Quelle: [SSC+13] 16
7 *Kollisionsobjekte, die durch eine Schneedecke wandern.* 34
8 *Partikelrendering: (a) Visualisierung des Fragment-Shaders (b) Rendering des Stanford-Hasen mit 183029 Partikeln* 37
9 *UML-Aktivitätsdiagramm: Überblick über die Abfolge der OpenGL-Shader für die Simulation von Schneepartikeln.* 39
10 *Die Performance der Anwendung bei größer werdender Partikelanzahl. Gitterknotenanzahl: 8 Millionen. Skaliert: Immer 8 Partikel pro Zelle. Unskaliert: Partikel pro Zelle verdoppeln sich pro Schritt beginnend mit 4 bei 47063 Partikeln. Datensatz ist der Stanford-Hase. Grafik erstellt mit https://www.openoffice.org/de/.* 41
11 *Die Performance der Anwendung bei größer werdender Gitterknotenanzahl, 47063 Partikel. Datensatz ist der Stanford-Hase. Grafik erstellt mit https://www.openoffice.org/de/.* 42
12 *Ein Schneehase fällt auf eine ebene Fläche. Verschiedene Parameter werden angepasst, um das Ergebnis zu beeinflussen. (a) Ausgangssituation der Simulation (b) Referenzbild.* 46
13 *Ein Schneehase fällt auf eine ebene Fläche. Verschiedene Parameter werden angepasst, um das Ergebnis zu beeinflussen. (a) Ausgangssituation der Simulation (b) Referenzbild.* 47
14 *Ein Schneehase fällt auf eine ebene Fläche. Das Ergebnis zeigt trockenen und nassen Schnee.* 48
15 *Kollision zwischen zwei Schneebällen.* 49
16 *Abhängigkeit zwischen Youngschem Modul und Schneedichte nach [Sig06].* Grafik erstellt mit http://rechneronline.de/function-graphs/. 51
17 *Abhängigkeit zwischen Poissonzahl und Schneedichte nach [Sig06].* Grafik erstellt mit http://rechneronline.de/function-graphs/. 52

Tabellenverzeichnis

1 *Einteilung der Schneearten nach Dichtebereichen.* Quelle: [DWD] 14
2 *Nützliche Ausgangsparameter für das Modell nach [SSC+13].* . . 21
3 *Repräsentation der Partikel-Parameter im Shader* 30
4 *Repräsentation der Gitterknoten-Parameter im Shader* 31
5 *Repräsentation der Kollisionsobjekte-Parameter im Shader* 35

Quellcodeverzeichnis

1 Anlegen eines SSBOs für Partikelgeschwindigkeiten auf C++ Seite. 22
2 Lokale Arbeitsgruppengröße 23
3 Deklarieren eines SSBOs im Shader 23
4 Lese- und Schreibzugriff auf einen SSBO 24
5 Wahrscheinlich ungewolltes/unerwartetes Ergebnis einer Initialisierung einer Matrix 24
6 Nutzung von atomaren Operationen 25
7 Berechnung zwischen Gitterknoten und Partikel 32
8 Aufbau eines zwischen Partikel und Gitter kommunizierenden Shaders 33
9 Updaten der Position der Kollisionsobjekte 35
10 Algorithmus für die Berechnung von Kollisionen zwischen Partikeln und Rigid-Bodies 36
11 Primitives Rendering für ein Schnee-Partikel 37

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

[ATO09] ALDUÁN, Iván ; TENA, Angel ; OTADUY, Miguel A.: Simulation of high-resolution granular media. In: *Proc. of Congreso Español de Informática Gráfica* Bd. 1, 2009

[BR86] BRACKBILL, J U. ; RUPPEL, H M.: FLIP: A Method for Adaptively Zoned, Particle-in-cell Calculations of Fluid Flows in Two Dimensions. In: *J. Comput. Phys.* 65 (1986), August, Nr. 2, 314–343. `http://dx.doi.org/10.1016/0021-9991(86)90211-1`. – DOI 10.1016/0021–9991(86)90211–1. – ISSN 0021–9991

[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* ACM, 2005, S. 77–86

[CSL08] CASSON, Jerry ; STOELINGA, Mark ; LOCATELLI, John: Evaluating the Importance of Crystal-Type on New Snow Instability: a Strength vs. Stress Approach Using the SNOSS Model. In: *Proc. Int. Snow Science Workshop*, 2008

[DWD] Deutscher Wetterdienst: Schneedichte. `http://www.deutscher-wetterdienst.de/lexikon/index.htm?ID=S&DAT=Schneedichte`. – Einsicht am 02.02.2015

[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

[GBO04] GOKTEKIN, Tolga G. ; BARGTEIL, Adam W. ; O'BRIEN, James F.: A Method for Animating Viscoelastic Fluids. In: *ACM SIGGRAPH 2004 Papers*. New York, NY, USA : ACM, 2004 (SIGGRAPH '04), 463–468

[Gra12] GRANGER, R.A.: *Fluid Mechanics*. Dover Publications, 2012 (Dover Books on Physics). – 259 S. `https://books.google.de/books?id=VWG8AQAAQBAJ`. – ISBN 9780486135052

[Hun13] HUNZ, Jochen: *The Possibilities of Compute Shaders - an Analysis*, Universitaet Koblenz-Landau, Campus Koblenz, Diplomarbeit, 2013

[ITF04] IRVING, Geoffrey ; TERAN, Joseph ; FEDKIW, Ron: Invertible finite elements for robust simulation of large deformation. In: *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation* Eurographics Association, 2004, S. 131–140

[IWT12] IHMSEN, Markus ; WAHL, Arthur ; TESCHNER, Matthias: High-Resolution Simulation of Granular Material with SPH. In: *VRIPHYS*, 2012, S. 53–60

[LD09] LENAERTS, Toon ; DUTRÉ, Philip: Mixing fluids and granular materials. In: *Computer Graphics Forum* Bd. 28 Wiley Online Library, 2009, S. 213–218

[Lib01] LIBBRECHT, Kenneth G.: Morphogenesis on ice: The physics of snow crystals. In: *Engineering and Science* 64 (2001), Nr. 1, S. 10–19

[MGG+10] MARECHAL, Nicolas ; GUÃ©RIN, Eric ; GALIN, Eric ; MERILLOU, Stephane ; MERILLOU, Nicolas: Heat Transfer Simulation for Modeling Realistic Winter Sceneries. In: *Computer Graphics Forum* 29 (2010), Nr. 2. http://liris.cnrs.fr/publis/?id=4585

[MP89] MILLER, Gavin ; PEARCE, Andrew: Globular dynamics: A connected particle system for animating viscous fluids. In: *Computers & Graphics* 13 (1989), Nr. 3, S. 305–309

[MST+11] MCADAMS, Aleka ; SELLE, Andrew ; TAMSTORF, Rasmus ; TERAN, Joseph ; SIFAKIS, Eftychios ; STUDIOS, Walt Disney A.: Computing the Singular Value Decomposition of $3\times$ 3 matrices with minimal branching and elementary floating point operations / Technical Report, University of Wisconsin-Madison. 2011. – Forschungsbericht

[NIDN97] NISHITA, Tomoyuki ; IWASAKI, Hiroshi ; DOBASHI, Yoshinori ; NAKAMAE, Eihachiro: *A Modeling and Rendering Method for Snow by Using Metaballs*. 1997

[OBH02] O'BRIEN, James F. ; BARGTEIL, Adam W. ; HODGINS, Jessica K.: Graphical Modeling and Animation of Ductile Fracture. In: *ACM Trans. Graph.* 21 (2002), Juli, Nr. 3, 291–294. http://dx.doi.org/10.1145/566654.566579. – DOI 10.1145/566654.566579. – ISSN 0730–0301

[PKA+05] PAULY, Mark ; KEISER, Richard ; ADAMS, Bart ; DUTRÉ, Philip ; GROSS, Markus ; GUIBAS, Leonidas J.: Meshless animation of fracturing solids. In: *ACM Transactions on Graphics (TOG)* Bd. 24 ACM, 2005, S. 957–964

[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*. Aire-la-Ville, Switzerland, Switzerland : Eurographics Association, 2012 (SCA '12). – ISBN 978–3–905674–37–8, 25–32

[Sig06] SIGRIST, Christian: *Measurement of fracture mechanical properties of snow and application to dry snow slab avalanche release*, University of Bern, Diss., 2006

[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://graphics.berkeley.edu/papers/Sumner-ASM-1999-03/

[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

[SSJ+14] STOMAKHIN, Alexey ; SCHROEDER, Craig ; JIANG, Chenfanfu ; CHAI, Lawrence ; TERAN, Joseph ; SELLE, Andrew: Augmented MPM for Phase-change and Varied Materials. In: *ACM Trans. Graph.* 33 (2014), Juli, Nr. 4, 138:1–138:11. http://dx.doi.org/10.1145/2601097.2601176. – DOI 10.1145/2601097.2601176. – 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, S. 236–252

[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 Transactions on Graphics (TOG)* 24 (2005), Nr. 3, S. 965–972