UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Bruchsimulation

Masterarbeit

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

vorgelegt von
Raphael Baumeister

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

Koblenz, im September 2014

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)

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

UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Masterarbeit
Raphael Baumeister
(Matr.-Nr. 208 210 293)

Thema: Bruchsimulation

Moderne Rechnersysteme ermöglichen die 3D-Darstellung komplexer, virtueller Umgebungen in sehr hoher Qualität. Eine wichtige Herausforderung besteht nun darin, das Verhalten von Objekten z.B. auf Basis physikalischer Gegebenheiten zu simulieren, um diese virtuellen Umgebungen attraktiv bzw. realitätsnah erscheinen zu lassen. Ein aufgrund seines Aufwands häufig vernachlässigtes Phänomen ist das Bruchverhalten simulierter Objekte das im Rahmen dieser Arbeit speziell untersucht werden soll.

Ziel dieser Arbeit ist die Modellierung von Brüchen starrer Körper. Hierzu werden bestehende Ansätze analysiert und in Bezug auf ihre Realitätsnähe und Echtzeitfähigkeit verglichen. Optional soll aus den Ergebnissen eine eigenständige Weiterentwicklung der Verfahren entstehen.

Im Rahmen der Masterarbeit sind im Einzelnen folgende Grundaufgaben zu lösen:

1. Erarbeitung bekannter Verfahren der Computergrafik zur Bruchsimulation.
2. Untersuchung der Ansätze zur Berechnung eines Bruchs auf Echtzeit und physikalische Plausibilität.
3. Erarbeitung des physikalischen Kontexts.
4. Implementierung.
5. Test und Bewertung.
6. Dokumentation der Ergebnisse.

Koblenz, den 27.03.2014

R. Baumeister
Raphael Baumeister

S. Müller
Prof. Dr. Stefan Müller

Zusammenfassung

Aufgrund ihrer Komplexität wird die Simulation von Brüchen in echtzeitfähigen Anwendungen der Computergraphik häufig gemieden. Durch Methoden aus den Ingenieurswissenschaften können Simulationen geschaffen werden, die Spiele und andere Anwendungen enorm bereichern. Stetig steigende Rechnerleistungen ermöglichen entsprechende Simulationen in Echtzeit und machen diesen Aspekt zunehmend interessanter.

Das Ziel dieser Arbeit ist die Modellierung von Brüchen starrer Körper durch eine Simulation. Der Fokus richtet sich dabei auf die physikalische Plausibilität und Performanz der Anwendung. Durch diese Ausarbeitung soll beantwortet werden, inwiefern eine Simulation von Brüchen mit Mitteln der Computergraphik umgesetzt werden kann.

Es wurden drei bestehende Ansätze und eine eigene Entwicklung implementiert und analysiert. Dieser Arbeit liegen die Verfahren „Real-Time Simulation of Deformation and Fracture of Stiff Materials“ von Müller et al., „Real-Time Simulation of Brittle Fracture using Modal Analysis“ von Glondu et al. und „Fast and Controllable Simulation of the Shattering of Brittle Objects“ von Smith et al. zugrunde. Die vorgestellten Methoden führen voneinander abweichende Bruchbildungen durch. Das eigenständig entwickelte Verfahren baut auf deren Vorzügen auf und erweitert sie mit der Idee der sekundären Risse. Die Implementierung der vier Ansätze erfolgte in der Physik-Engine Bullet.

Die Ergebnisse der Arbeit zeigen, dass physikalisch basierte Brüche in Echtzeit realisierbar sind. Die Untersuchung der physikalischen Methoden auf Performanz zeigte, dass diese vor allem mit der Struktur der Objekte zusammenhängen. Die präsentierten Methoden lieferten für eine Auswahl an Objekten physikalisch plausible Ergebnisse in Echtzeit. Durch die Ausarbeitung wird deutlich, dass die weitere Erforschung der Thematik neue Möglichkeiten aufdecken kann. Die Verbesserung des Realismus in echtzeitfähigen, virtuellen Welten kann mit dem Einsatz von physikalisch plausiblen Methoden erreicht werden.

Abstract

Real-time computing often avoids the simulation of fractures due to its complexity. The field of engineering science provides methods to create these simulations to improve games and other applications. Steadily rising computer capacities allow suitable simulations on a real-time basis and make this aspect increasingly interesting. The topic and aim of this research is to simulate fractures of stiff bodies. The primary objective is the physical plausibility and performance of the application. This thesis analyses the potential of computerscience to realize the simulation of fractures.

Three existing as well as one self-created were implemented and analysed. The works "Real time simulation of deformation and Fracture of Stiff of material" from Müller et al., "real time simulation of Brittle Fracture using Modal analysis" from Glondu et al. and "Almost and Controllable simulation of the Shattering of Brittle Objects" from Smith et al. form the basis of this thesis. The introduced methods use different computation of forces and fractures. The developed procedure uses the idea of generating secondary breaks. The approaches were implemented based on the BULLET physics-engine. The results of the work show that physically based breaks are realizable on a real-time basis.

The analyses of the physical methods demonstrates that their performance mainly depends on the constitution of the used objects. This thesis shows, that the further investigation of this topic can discover new possibilities. The improvement of the realism in virtual worlds can be achieved by executing physically plausible methods.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Seitdem Geschichten durch Filme und neuerdings auch durch Computerspiele erzählt werden, benutzen Regisseure und Spieleentwickler das Zerbrechen als Stilmittel. Es wird genutzt, um die Dramaturgie in Filmen oder virtuellen Welten zu unterstützen. Das Zerbrechen von Gegenständen kann je nach Inszenierung beim Betrachter unterschiedliche Emotionen, wie Trauer oder Wut auslösen.

In Filmen zählt die Simulation vom Zusammenbrechen von Häusern und anderen Objekten zu den so genannten „Special Effects“. Vor allem die in den letzten 15 Jahren entstandenen Action-Filme basieren in vielen Szenen auf computerbasierten Animationen. Die Simulation von Explosionen oder anderen zerstörerischen Handlungen ist eine große Aufgabe für die Entwickler der Spezialeffekte. In Abbildung 1 ist ein Beispiel der Entwicklung des Films 2012 zu sehen. Die Entwickler des Filmes haben für die Zerstörung von Brücken und anderen Bauwerken für jedes Objekt ein dreidimensionales Modell entwickelt, welches je nach Detailgrad in unterschiedlich viele und große Bruchstücke zerfallen kann.

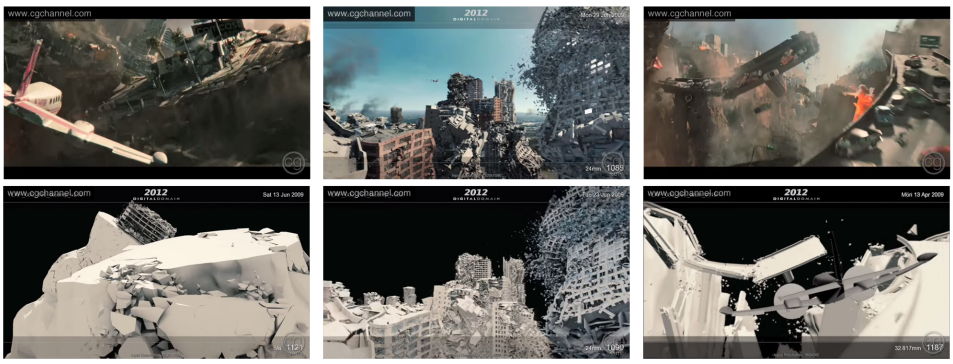


Abbildung 1: Simulation von Brüchen durch Spezialeffekte im Film 2012. Oben: Aufnahme aus dem Film - Unten: Darstellung der zerbrechenden Modelle

Im Bereich der Computerspiele gibt es neben den animierten Geschichten und Zwischensequenzen, welche einem Film ähneln, immer häufiger die Möglichkeit der Interaktion mit der Umwelt. In Spielen, welche aus der Ego- oder „Third-Person“-Perspektive gespielt werden, ist ein wichtiger Bestandteil des Spiels die Immersion. Je realistischer die virtuelle Realität sich verhält, und je mehr Interaktion dem Spieler möglich ist, desto immersiver und intensiver ist das Spielerlebnis. Das Zerbrechen von Gegenständen oder die Zerstörung von Objekten machen eine virtuelle Szenerie deutlich realistischer. Die Möglichkeit seine, Umwelt komplett oder teilweise zu zerstören, wird in Compu-

Abbildung 2: Spielszene aus BATTLEFIELD 4. Der Spieler hat die Möglichkeit, durch Feuerwaffen den Damm zu zerstören und eine für ihn vorteilhafte Veränderung der virtuellen Welt zu erreichen.[12]

terspielen auch „destructible environment“ genannt. Im Spiel BATTLEFIELD 4 ist es durch das so genannte „LEVOLUTION“-Feature [1] beispielsweise möglich, durch die Zerstörung eines Staudammes (siehe Abbildung 2) die neu entstandenen Verhältnisse in der virtuellen Welt zu seinem Vorteil zu nutzen. Im Anhang in den Abbildungen 41, 42 und 43 auf Seite 88 ist darüber hinaus die Entwicklung der Zerstörungs-Simulation eines Gebäudes zu sehen. Die drei Abbildungen zeigen unterschiedliche Teile der BATTLEFIELD-Computerspielreihe und geben deutlich zu erkennen, dass in diesem Bereich eine sichtliche Veränderung vorliegt. Der Detailreichtum und die Wirklichkeitsnähe der entstandenen Bruchsimulation hat sich enorm verbessert.

Das breite Spektrum der Anwendungsmöglichkeiten von Bruchsimulationen in Spielen und Filmen wird komplettiert durch Simulationen in Medizin und Industrie.

Der wissenschaftliche Bereich der modernen Bruchmechanik nimmt in den 30er und 40er Jahren des 20. Jahrhunderts mit den Anforderungen an Schiffs- und Flugzeugrümpfe seinen Anfang und hat seit dem in vielen anderen Bereichen sein Dasein gefunden. Vor allem die industriell starken Bereiche der Immobilien- und Autoindustrie haben sehr viele Erkenntnisse in der

[1] http://www.battlefield.com/de/battlefield-4/features/levolution

Bruchmechanik hervorgebracht, da mögliche Einstürze von Bauwerken oder Brüche an Verkehrsmitteln unbedingt vermieden werden sollten.

Dank stark gestiegener Rechenleistung in den letzten Jahrzehnten ist es mittlerweile möglich, bestehende Methoden aus diesen Ingenieurswissenschaften für die oben genannten computergraphischen Simulationen zu verwenden. Die Simulation von Brüchen hat hierbei eine besondere Komplexität, welche bereits in vielfältigen Ansätzen diskutiert wurde.

1.2 Problemstellung

Die Film- und Computerspielindustrie hat das Bedürfnis ein wirklichkeitsgetreues Abbild der Natur zu entwerfen. Neben der Visualisierung von Schatten, Texturen und anderen optischen Elementen ist das Zerbrechen ein Phänomen, welches dargestellt werden kann, um realistischer zu wirken. Bruchsimulationen sind in diesem Bereich dementsprechend ebenfalls sehr interessant.

Die Bruchmechanik und die Physik liefern einige Möglichkeiten für diese Bruchsimulationen. Die Herausforderung liegt darin, diese physikalischen Gegebenheiten mit Mitteln der Computergraphik zu simulieren. Die Berechnung der zu Grunde liegenden Modelle haben jedoch teilweise einen sehr hohen Aufwand.

Wie an der immer weiter steigenden Leistung und Qualität der Spiele und „s“[2] zu sehen ist, spielt der Realismus im Bereich der Graphik eine immer wichtigere Rolle. Die Entwickler simulieren Deformationen und Brüche bereits in steigender Anzahl, jedoch ist die Entwicklung dabei nicht so weit fortgeschritten, wie beispielsweise die „Fluid“- oder Partikel-Simulationen.

Eine Bruchsimulation müsste ein sehr hoch aufgelöstes Objekt-Modell und eine exakt arbeitende Physik-Simulation als Grundlage haben, um optimale Ergebnisse zu erzielen. Die Kombination dieser beider Faktoren hat jedoch zur Folge, dass die Berechnungen in Echtzeit nicht zu bewältigen sind. Für Anwendungen wie Computerspiele müssen daher beispielsweise einfachere Lösungen gefunden werden. Das Anforderungsprofil an eine Bruchsimulation ist also sehr vielschichtig. Berechnung in Echtzeit, realistische Optik und physikalische Korrektheit müssen hierbei in Betracht gezogen werden.

Die bisher am häufigsten vorzufindende Simulation von Brüchen basiert meist auf Objekten, welche für die Bruchsimulation bereits vorher in Einzelteile zerlegt worden sind. Diese Vorfertigung der Objekte hat in großen virtuellen Welten einen hohen Modellierungsaufwand. Um diese händische Vorbereitung zu minimieren, bieten sich physikalisch basierte Simulationen, welche auf den normalen Objekten arbeiten, an.

[2] Programm zur Simulation physikalischer Prozesse

1.3 Zielsetzung

Das Ziel dieser Arbeit ist es, verschiedene Ansätze zur Simulation von Brüchen zu implementieren, die Resultate zu analysieren und aus den Ergebnissen eine eigenständige Methode zu entwickeln. Jedes Verfahren soll anschließend auf seine physikalische Plausibilität und Echtzeitfähigkeit überprüft werden. Erstrebenswert sind Ergebnisse mit schneller Laufzeit und Objekte die realistisch in Bruchstücke zerfallen. Für den Benutzer der entwickelten Anwendung soll es möglich sein, zwischen den implementierten Verfahren zu wechseln und diese durch unterschiedliche Parameter interaktiv zu steuern.

Die Anwendung soll unabhängig von konkreten Szenarien sein, um eine Reproduzierbarkeit zu ermöglichen. Bei der Umsetzung wird nur in zweiter Linie auf grafische Feinheiten, wie Texturierung und „Shading“ geachtet. Das Hauptaugenmerk wird auf die grundsätzliche Modellierung von Brüchen starrer Körper gelegt. Zum Abschluss dieser Arbeit soll entschieden werden, ob die umgesetzten Verfahren den in der Problemstellung angesprochenen Anforderungen der Unterhaltungsindustrie gerecht werden.

1.4 Aufbau der Arbeit

Nach dem einleitenden Abschnitt wird als Erstes im Abschnitt 2 ein Überblick über bestehende Ansätze zur Simulation von Brüchen gegeben und themenverwandte Arbeiten vorgestellt. Im darauf folgenden Abschnitt werden physikalische Grundlagen, welche für die Entwicklung einer Festkörpersimulation wichtig sind, erläutert und deren mathematische Zusammenhänge dargelegt. Zusätzlich werden die für die Bruchbildung grundlegenden Kenntnisse über Materialeigenschaften, Belastung und Kollision ausführlich erklärt. Der anschließende Abschnitt 4 zeigt die Auswahl der getesteten Bruchsimulationen und stellt die einzelnen Ansätze vor. Die für das Verständnis notwendigen Formeln und Methoden werden dabei vertieft. Die beiden darauf folgenden Abschnitte zeigen die Umsetzung der Implementationen. Es werden die zugrunde liegende Physik- und die programmierten Methoden erläutert. Im Anschluss werden die Ergebnisse (Abschnitt: 7) präsentiert und eine Wertung im Hinblick auf die aufgestellten Anforderungen abgegeben. Die schriftliche Ausarbeitung wird mit dem Fazit abgeschlossen.

2 Verwandte Arbeiten

Die Thematik der Bruchsimulation ist bereits Gegenstand intensiver computergraphischer Betrachtung gewesen. Es gibt in diesem Bereich mehrere thematische Ansätze, welche in der Literatur bereits behandelt wurden. Die Arbeiten diesbezüglich reichen von Feder-Masse-Modellen über abgeleitete Verfahren aus der Kontinuumsmechanik bis hin zu partikelbasierten Verfahren. Es existieren unter anderem auch Werke, die Verfahren ohne physikali-

sche Grundlage vorstellen. In dieser Ausarbeitung werden lediglich diejenigen Ansätze betrachtet, welche eine physikalische Basis besitzen.

In der Literatur existieren sehr viele unterschiedliche Begriffe, welche die Art des Zerbrechens definieren. Im englischen werden hierbei die Begriffe „shattering“, „crumbling“ und „tearing“ unterschieden. In der Ausarbeitung „Physically-based and Real-time Simulation of Brittle Fracture for Interactive Applications“[6] von Glondu et al. finden sich passende Erläuterungen.

2.1 Ansätze aus der Kontinuumsmechanik

Die meisten Verfahren der Simulation von Belastungen basieren auf Berechnungen der Kontinuumsmechanik. In dieser Wissenschaft werden vereinfachte Annahmen über Eigenschaften von Materialien gemacht, welche sich gut für Objekt-Deformationen eignen. In Abschnitt 3.3 werden Eigenschaften unterschiedlicher Materialien vorgestellt. Materialien sind in ihrem natürlichen, atomaren Aufbau für computergraphische Darstellungen zu komplex, da die Menge an unbekannten Parametern einen nicht zu bewältigenden Rechenaufwand zur Folge hätte. Objekte werden daher meist vereinfacht dargestellt und über einzelne Elemente simuliert. Innerhalb eines Elements sind die Auswirkungen von wirkenden Kräften über endlich viele Variablen definiert. Der Berechnungsaufwand wird durch Aufteilung der Objekte reduziert und die Materialeigenschaften durch Parameter angenähert.

O'Brien und Hodgins erweiterten 1999 [17] eine „Finite Elemente Methode“ (siehe Abschnitt 3.2) um ein Bruchkriterium und simulierten die Brüche auf eine neue Art und Weise. Sie entwickelten, mit Hilfe einer Starrkörpersimulation die Basis für viele nachfolgende Ansätze. Nach jedem Zeitschritt wird bei O'Brien und Hodgins die auf die Elemente wirkende Belastung mit einem Materialparameter verglichen und bei dessen Überschreitung ein Bruch ausgelöst. Die Massenpunkte des Objektes werden danach durch eine Bruch-Ebene geteilt und die Tetraeder an den Bruchkanten neu berechnet.

Ein weiterer Ansatz, der auf der „Finiten Elemente Methode“ basiert, wurde 2007 von Bao et al. [2] vorgestellt. In diesem Ansatz wird eine „Stress-Map“ durch die auftretenden Belastungen generiert und als Bruchkriterium genutzt. Sollte ein festgelegter Schwellwert, innerhalb eines Zeitschrittes überschritten, werden wird ein Bruch ausgelöst. Beim Bruch eines Objekts berechnet der vorgestellte „virtual node algorithm“ einfache und stabile neue Formen. Je höher die Belastung des Objektes an einer Stelle des Objektes ist, um so mehr Bruchstücke werden in diesem Gebiet erzeugt. Die Richtung, in der ein Bruch fortgeführt wird, wird in diesem Ansatz von einzelnen Energiefunktionen der umliegenden Knoten beeinflusst. Dieses Verfahren eignet sich vor allem für starre Körper mit dünner Schale, welche durch ein Netz von Dreiecken repräsentiert werden.

Sehr gute Ergebnisse brachte die zwei Jahre später (2009) entwickelte Bruchsimulation von Parker und O'Brien hervor [19]. Ihr Ansatz wurde für

ein Echtzeit-Videospiel entwickelt und musste somit strengen Kriterien wie Robustheit und Performanz entsprechen. Das Herzstück des Systems ist eine Methode, welche mehrere Ideen anderer Ansätze kombiniert. Zur Bestimmung eines Bruches wird die Deformation der Tetraeder betrachtet. Verändert sich das Volumen eines Tetraeders um einen bestimmtem Prozentwert (Beispielwert aus [19]: 6%), so wird ein Bruch initiiert. Der Bruch an sich ist eine simple Version des in [17] vorgestellten Ansatzes, jedoch ohne das Aufteilen der Tetraeder an den Bruchkanten. Diese Neugenerierung der Meshs ist nicht notwendig, da die graphische und physikalische Repräsentation des Objektes über so genannte „Splinter“ verbunden ist. Diese „Splinter“ sind Einheiten, welche höher aufgelöste, polygonale Oberflächen beinhalten, und so eine sichtbar detailliertere Geometrie darstellen. Diese kleineren Einheiten werden dann -je nach Schwerpunkt- einem Tetraeder im gröberen Mesh zugeordnet. Zur Bruchsimulation wird dann das gröbere Tetraeder-Mesh genutzt. In Abbildung 3 sind die beiden Repräsentationen und ihr Zusammenhang sichtbar.

Die in [14] und [7] vorgestellten Ansätze basieren ebenfalls auf Methoden der Kontinuumsmechanik. Sie sind zwei der in dieser Ausarbeitung implementierten Ansätze und werden in Abschnitt 4 detailliert beschrieben.

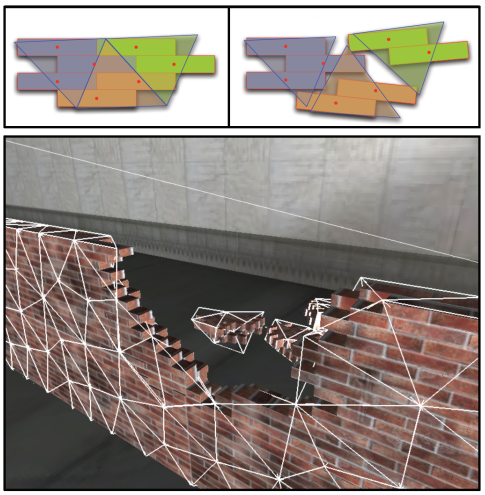


Abbildung 3: Geometrierepräsentation nach Parker und O‘Brien.[19]

2.2 Feder-Masse-Modelle

Modelle, die ein Feder-Masse-System als Grundlage besitzen, sind in den letzten Jahren kaum noch vorgestellt worden. Diese Methode ist jedoch die Grundlage aller physikalisch nicht korrekten Bruchsimulationen.

Bereits 1988 haben Terzopoulos und Fleischer[22] einen Ansatz vorgestellt, bei dem ein Papier, welches als zweidimensionales Mesh modelliert ist, zerreißt. Hierfür wurde ein Masse-Feder-Netz simuliert, indem die Federn bei Überschreitung eines elastischen Limits entfernt werden. Hierfür wurde der Abstand zwischen zwei verbundenen Knoten mit einem vorgegebenen Schwellwert verglichen. Norton et al. [15] haben diesen Ansatz aufgegriffen und für dreidimensionale Objekte erweitert. Zusätzlich wurde die Zusammenfassung von Federn in Blöcke vorgestellt. Dies führt dazu, dass mehrere Federn gleichzeitig entfernt werden und dadurch bessere Fragmente

entstehen können.

Eine ähnliche Herangehensweise stellen Smith et al. [21] vor. In ihrem Ansatz wird das physikalische Mesh und dessen Elemente durch so genannte „cohesive forces“ zusammengehalten. Dieser Ansatz ist in dieser Ausarbeitung implementiert und wird in Abschnitt 4 genauer erläutert.

3 Grundlagen

3.1 Allgemeine physikalische Grundlagen

Das Verhalten von Objekten, welche aus einem spröden Material bestehen, wird häufig durch so genannte Starrkörpersimulationen[3] realisiert. Um eine Deformation eines beliebigen Objektes durchführen zu können, werden diese durch Massenpunkte[4] dargestellt. Je mehr Massenpunkte ein Objekt besitzt umso mehr Möglichkeiten der Deformation gibt es. Ein Massenpunkt eines Objektes hat keine Ausdehnung und wird durch die Parameter der Masse m, seiner Position $p(t)$ und seiner Geschwindigkeit $v(t)$ definiert. Der Bewegungszustand kann durch eine einwirkende Kraft f verändert werden. In seinem Buch „Game Physics Developement“ [13] stellt Millington im Abschnitt „Laws of Motion“ die Formeln zur Berechnung von Masse, Geschwindigkeit und Beschleunigung vor.

Das zweite Newtonsche Bewegungsgesetz besagt:

Eine Änderung der Bewegung einer Masse ist proportional zu Kraft und Richtung der einwirkenden Masse.

$$f = m\dot{v} \tag{1}$$

$\dot{v}$ bezeichnet hierbei die Änderung/Ableitung der Geschwindigkeit, also die Beschleunigung eines Massenpunktes.

Zusätzlich hat das dritte Newtonsche Bewegungsgesetz eine für die Bruchsimulation wichtige Aussage:

Eine Kraft, die von einem Objekt A auf ein Objekt B gerichtet ist, erzeugt immer auch eine gleich große, aber entgegen gesetzt wirkende Kraft von B in Richtung A.

$$F_{A\to B} = -F_{B\to A} \tag{2}$$

Um die Position eines Massenpunktes zu verändern, ist die Integration eines Zeitschrittes nötig. Ein Integrationsschritt besteht dann aus den Berechnungen der neuen Position, der einwirkenden Kraft und der resultierenden Geschwindigkeit.

[3] engl. Rigid-Body Simulation/Dynamics

[4] In der Literatur häufig auch Knoten genannt.

Die folgende Formel beschreibt das Integral der Geschwindigkeit v über die Zeit t. Hierdurch wird die neue Position des Punktes berechnet.

$$p' = p + vt \tag{3}$$

Die neue Geschwindigkeit berechnet sich analog durch:

$$v' = v + \dot{v}t \tag{4}$$

Die hierfür benötigte Beschleunigung $\dot{v}$ ergibt sich aus den auf das Objekt einwirkenden Kräften f und der Masse m. f ist dabei gemäß dem D'Alembert-Prinzip die Summe aller auf das Objekt wirkenden Teilkräfte (Formel 5). Das Prinzip von D'Alembert ist in [13] im Abschnitt „Adding General Forces" nachzulesen.

$$f = \sum_i f_i \tag{5}$$

Die exakte Berechnung der Kräfte, welche auf einen Punkt Einfluss, haben folgt im Abschnitt 4.1.1.

3.2 Aufteilung eines Objekts

Ein möglicher Ansatz, um eine Bruchsimulation durchzuführen, ist die Unterteilung des Objektes in viele Teilobjekte. Die Elemente $e_j (1 \leq j \leq n)$ teilen sich Massenpunkte x_i und bilden dadurch ein zusammenhängendes Objekt.

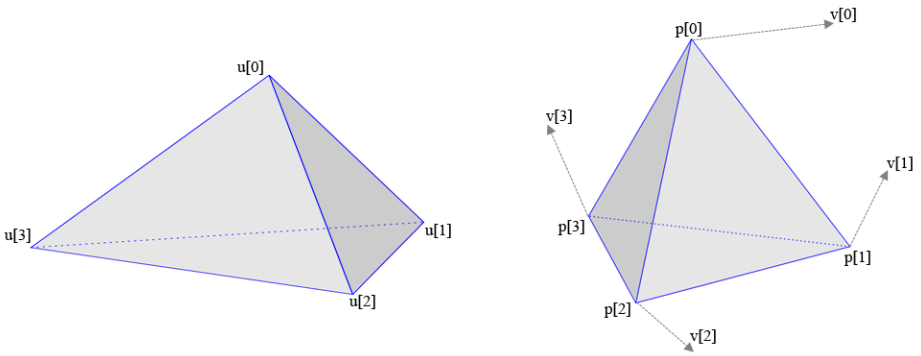


Abbildung 4: Material- und Weltkoordinatensystem

Es wird so der Aufbau eines Gegenstandes angenähert, ohne den Rechenaufwand sehr groß werden zu lassen. [6]

Für jedes Element sind die Auswirkungen der Belastungen und Kräfte definiert. Durch die Verbindungen der einzelnen Elemente ist die Möglichkeit gegeben, die physikalischen Kräfte an einem Objekt zu simulieren.

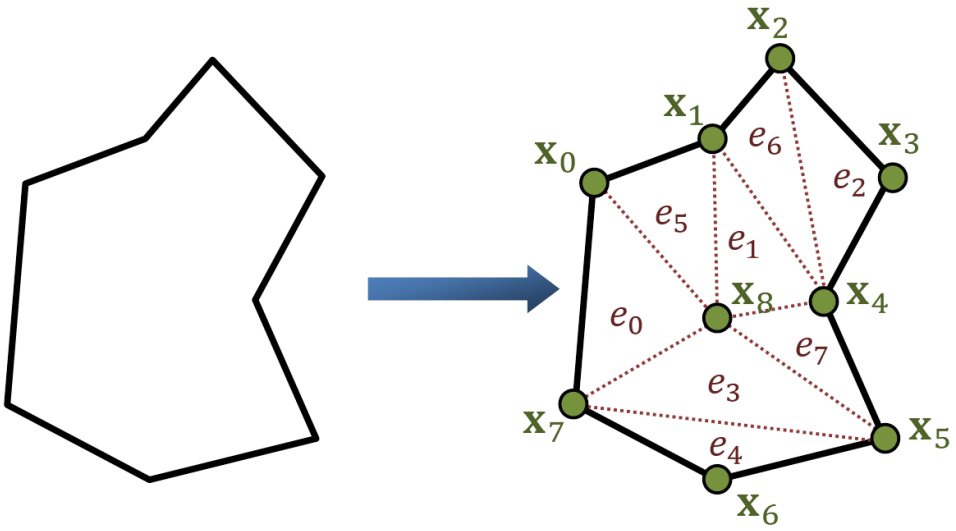


Abbildung 5: *links:* Ein zweidimensionales Objekt, *rechts:* Unterteilung des Objekts in die Elemente e_j.[6]

Für die Darstellung der einzelnen Elemente werden zwei Koordinatensysteme (Abbildung 4) genutzt.

1. Materialkoordinatensystem: $u = [U, V, W]^T$
2. Weltkoordinatensystem: $x(u) = p = [X, Y, Z]^T$

Jedes Element besteht aus vier Massenpunkten mit jeweils Material-, Welt- und Geschwindigkeitsinformationen. $u[i]$ beschreibt die Koordinaten der einzelnen Punkte in Materialkoordinaten[5]. Diese Koordinaten beinhalten die Informationen über die nicht deformierte Ausgangskonfiguration des Objektes. Alle angewandten Kräfte und mögliche Deformationen sind in $p[i]$ zu finden. Hier wird die Position in Weltkoordinaten gespeichert. Zusätzlich wird zu jedem Knoten die entsprechende Geschwindigkeit $v[i]$ angegeben.

In Abschnitt 4.1.1 wird durch diese Informationen die Belastung der einzelnen Elemente berechnet. Hierzu wird der Vergleich zwischen ursprünglicher ($u[i]$) und potentiell veränderter Form ($p[i]$) gezogen, und zusätzlich die Belastung durch auftretende Beschleunigungen - Ableitungen von $v[i]$ - ermittelt.

Ein Werkzeug für die Simulation physikalischer Phänomene ist die „Finite Elemente Methode“ (FEM). Im Gegensatz zu der hier erläuterten Struktur sind die Elemente dabei endlich klein (finit). In der Medizin wird diese Struktur beispielsweise zur Darstellung von Gewebedeformationen genutzt. Je nach Größe der Elemente kann der atomare Aufbau eines Objekts nachgeahmt werden.

[5] in der Literatur oft auch als lokales Koordinatensystem bezeichnet

3.3 Material

In der Bruchmechanik sind die Material- und Werkstoffeigenschaften der Objekte sehr wichtig. Ein Bruch kann auch als Deformation durch Materialversagen verstanden werden. Für eine Bruchsimulation ist es notwendig zu wissen, wann ein Material bricht, und was die Gründe dafür sind. Zusätzlich zu den Eigenschaften des Materials an sich spielen äußere Einflüsse, wie Druck und Temperatur des Materials, eine große Rolle. Diese Parameter werden in den in dieser Arbeit untersuchten Verfahren jedoch nicht berücksichtigt.

Bei einer Materialdeformation wird unterschieden, ob eine spröde oder eine duktile[6] Verformung durchgeführt werden muss. Für diesen Zweck wird zwischen elastischer und plastischer Verformung differenziert. In der Werkstoffkunde wird das so genannte Spannungs-Dehnungs-Diagramm genutzt, um Elastizität, Festigkeit und andere Eigenschaften darzustellen.

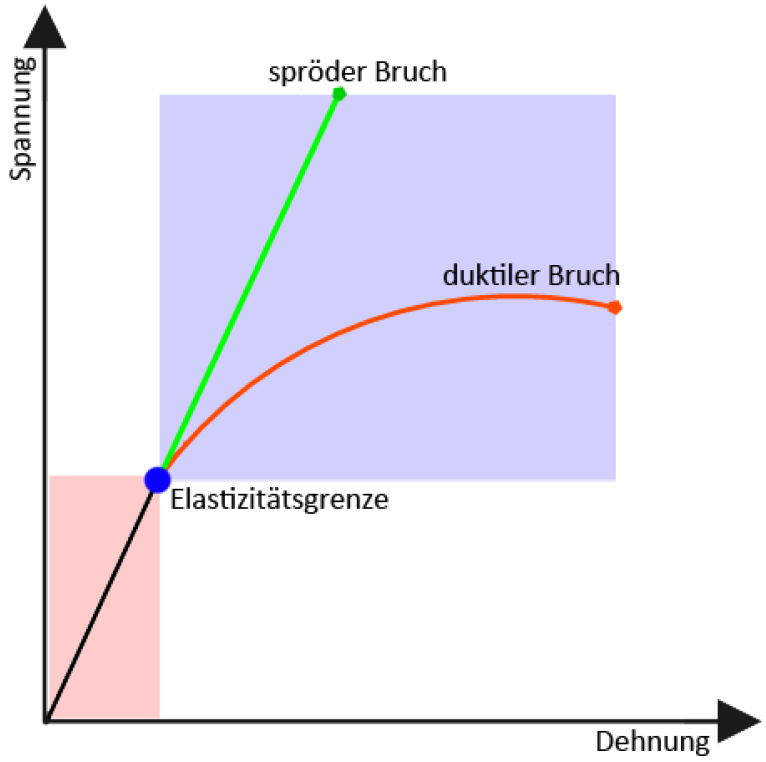


Abbildung 6: Bereiche des Spannungs-Dehnungs-Diagramm

In Abbildung 6 ist ein Spannungs-Dehnungs-Diagramm zu sehen, welches in zwei Bereiche unterteilt wurde. Im Ursprung des Koordinatensystems ist das Material im verformungsfreien Zustand. Sobald eine Belastung

[6] Ein Material gilt als duktil, wenn es sich bei Belastung erst plastisch verformt bevor es bricht.

des Objektes stattfindet, beschreibt der rot gekennzeichnete linear-elastische Bereich das Materialverhalten. In diesem Bereich findet eine elastische Verformung statt. Ein Objekt, welches lediglich elastisch verformt wird, kehrt nach Ende der Belastung in den ursprünglichen, verformungsfreien Zustand zurück. Nach Überschreiten der Elastizitätsgrenze folgt die plastische Verformung. Diese Verformung ist nicht mehr vollständig umkehrbar. Dieser Bereich ist blau gekennzeichnet.

Im Diagramm ist ein spröder und ein duktiler Bruch, nach Erreichen der Elastizitätsgrenze, eingezeichnet. Beim spröden Bruch findet kaum eine Dehnung des Materials statt, bevor es zerbricht, wohingegen beim duktilen Bruch eine extreme Dehnung eintreten kann. Durch diese Dehnung des Objekts wird das Material so sehr beansprucht, dass es nicht immer in den ursprünglichen Zustand zurückkehren kann. [8]

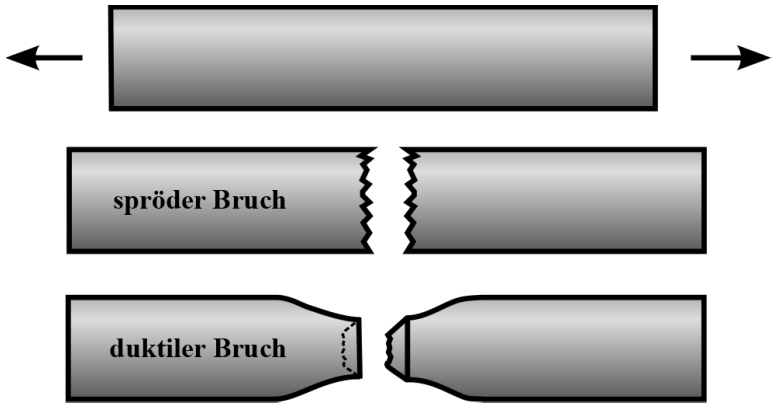


Abbildung 7: Abstraktion von sprödem und duktilem Bruch[24]

Die für die Unterscheidung der Verformung notwendigen Parameter sind in Tabelle 1 zu sehen. In Abschnitt 4.1.1 werden die Werte λ,μ,ϕ und ψ und in Abschnitt 4.1.1 der Wert τ für die Berechnung der Kräfte und Tensoren benötigt.

- Die Lamé-Konstanten beschreiben den Zusammenhang zwischen Deformation und Belastung innerhalb eines Körpers. Je höher diese Werte sind, umso höher muss die Belastung sein, um den Körper zu verformen. μ bestimmt hierbei die Starrheit des Materials, λ kontrolliert den Widerstand gegen Änderungen im Volumen,

- Die Dämpfungskonstanten ϕ und ψ definieren, wie schnell eine elastische Materialverformung zurückgeht, wenn die Krafteinwirkung auf das Objekt verschwindet. Die Parameter beschreiben, wie schnell ein Material die entstehende kinetische Energie wieder abführt.

- Die Materialhaltbarkeit τ legt den Schwellwert für die Belastbarkeit eines Materials fest.

		Glas	*Eisen*	*Blei*	*Keramik*
Lamé-Konstanten	λ	0.419	0.759	-	0.381
	μ	0.578	1.474	0.593	0.575
Dämpfungs-Konstanten	ϕ	1.04	18.98	-	4.79
	ψ	1.44	36.85	14.84	7.18
Materialhaltbarkeit	τ	6.01	24.82	11.88	2.48

Tabelle 1: Vergleich der Materialparameter (Auszug aus der Materialbibliothek in [16])

Ein spröder Bruch tritt bei Materialien mit geringer Duktilität auf und erfolgt ohne oder nur mit geringer plastischer Verformung. Beispielhaft für einen solchen Bruch ist das Zerbrechen von Glas oder Keramik. In Tabelle 1 ist deutlich zu erkennen, dass alle angegebenen Parameter deutlich unter denen von Eisen liegen. Eisen verformt sich im großen Maße plastisch bevor es bricht und hat somit eine sichtbar große duktile Verformung. In Abbildung 7 ist deutlich zu erkennen, dass sich das Objekt beim spröden Bruch lediglich unmittelbar an der Bruchkante verändert, wohingegen beim duktilen Bruch der Bereich in der Nähe des Bruchs ebenfalls abgewandelt wird.

3.4 Objektstruktur

Um aus einer Punktemenge ein Objekt zu erstellen, sind in der Computergraphik mehrere Verfahren bekannt. In dieser Ausarbeitung werden die Strukturen nach Voronoi und Delaunay genutzt.

Die räumliche Struktur der Voronoi-Diagramme wird von vielen Wissenschaftlern in der Informatik geschätzt, da sie sehr simpel und gleichzeitig sehr funktional ist. Eine Voronoi-Region (Teilraum des Objekts) enthält alle Punkte im Raum, deren Abstand zu einem charakteristischen Punkt am kleinsten sind. Da die Verarbeitung von Dreiecks-Netzen (Tetraeder im dreidimensionalen Raum) einfacher ist als von Polygon-Netzen, wird bei der Entwicklung von Algorithmen jedoch auch oft auf die „verwandte“ Delaunay-Triangulierung zurück gegriffen. Der Grund dafür liegt vor allem an der konstanten Anzahl von Massenpunkten und benachbarten Zellen bei der Delaunay-Darstellung. Diese Darstellung ist der duale Graph des

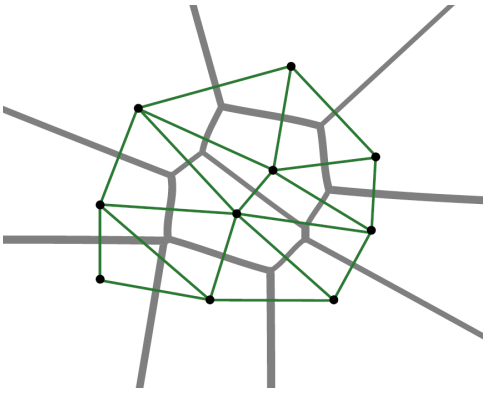


Abbildung 8: Zusammenhang zwischen Voronoi-Diagramm (grau) und Delaunay-Triangulierung (grün)[3]

Voronoi-Diagramms und wird ebenfalls von der in Abschnitt 5.2 auf Seite 35 vorgestellten Software genutzt. Beide Strukturen können voneinander abgeleitet werden. Für weitere Informationen über beide Verfahren und deren Implementation bietet sich die Arbeit „Voronoi diagrams - a survey of a fundamental geometric data structure“ [1] von Franz Aurenhammer an. Eine detaillierte Berechnung einer dreidimensionalen Voronoi-Struktur findet sich zudem in [11].

Die Darstellung der Polygon- oder Tetraedernetze wird in der Computergraphik oft auch durch den Begriff Mesh beschrieben.

3.5 Kollision

Die Kollision ist der Ursprung jeder Bruchsimulation. Nur wenn tatsächlich eine Kollision zwischen zwei Objekten stattfindet kann ein Bruch entstehen. Für diesen Zweck müssen theoretisch alle sich in der virtuellen Welt befindenden Objekte miteinander verglichen werden. Die Komplexität des Vergleichs dieser Objekte auf eine Kollision hängt hierbei sehr stark von deren Geometrie ab. Bei konkaven Objekten sind die Berechnungen beispielsweise deutlich umfangreicher als bei konvexen Objekten. Um die Performanz zu verbessern, wird die Kollisionserkennung in modernen Anwendungen meist in zwei Phasen aufgeteilt. In der ersten Phase, der so genannten „Broadphase“, wird überprüft, ob eine Kollision der Objekte überhaupt möglich ist. Hierfür werden die Objekte meist durch eine vereinfachte Geometrie angenähert. Im zweiten Schritt, der „Narrowphase“, wird dann die eigentliche Geometrie des Objekts genutzt. Durch diese Methode kann erheblich viel Rechenzeit eingespart werden. Die in Abschnitt 5.1 vorgestellte Physik- hat diese beiden Phasen ebenfalls implementiert.

Es existieren noch einige weitere Möglichkeiten der Optimierung. Das „Spatial Hashing“ beispielsweise teilt den gesamten Raum in gleich große und gleichförmige Zellen auf und trägt alle Objekte darin ein. Objekte der selben Zelle werden dann zur detaillierteren Kollisionsabfrage genutzt. Millington zeigt in seinem Buch „Game Physics - Development“ [13] in Kapitel 12 einige Möglichkeiten auf, wie eine performante Kollisionsabfrage implementiert werden kann.

3.6 Zeitschritt

Sich bewegende oder sich verformende Körper werden immer schrittweise simuliert. In jedem Zeitschritt (engl. timestep) werden Änderungen der Geschwindigkeit und Position oder erfolgte Kollisionen berechnet. Bei einer Simulation, welche in Echtzeit erfolgen soll, muss dieser Schritt relativ klein gewählt werden. Damit der Betrachter eine flüssige Bewegung, einer solchen Simulation wahrnimmt, sind mindestens 25 bis 30 Hertz[7] notwendig. Ein

[7] Vorgänge/Bilder pro Sekunde

Zeitschritt dauert dann $1/30 = 0.033$ Sekunden. Die in dieser Ausarbeitung genutzte BULLET Physik- verwendet zur Simulation einen Zeitschritt von 0.016s (60 Hertz).

4 Modelle der implementierten Verfahren

Die realistische Animation eines Bruchs ist schwierig, da für eine überzeugende Animation die physikalischen Eigenschaften eines Gegenstandes verstanden werden müssen. Das Objekt darf nicht lediglich als geometrische Gestalt betrachtet werden, sondern sollte die materiellen Eigenschaften wie Masse, Elastizität und Impuls besitzen. Die in diesem Abschnitt vorgestellten Verfahren wurden auf Grund ihrer verschiedenen Techniken ausgewählt. Die physikalischen Eigenschaften der Objekte werden auf unterschiedliche Art und Weisen simuliert. Die Verfahren bieten gute Möglichkeiten einen Vergleich auf ihre physikalische Plausibilität durchzuführen.

4.1 Verfahren auf Grundlage der Kontinuumsmechanik

Die Verfahren „Real-Time Simulation of Deformation and Fracture of Stiff Materials“ von Müller et al. [14] und „Real-Time Simulation of Brittle Fracture using Modal Analysis“ von Glondu et al. basieren auf dem von O'Brien und Hodgins [18, 17] vorgestellten Ansatz. Aus diesem Grund werden in diesem Abschnitt zunächst die Gemeinsamkeiten der Verfahren herausgestellt, und danach eine genaue Differenzierung der Verfahren durchgeführt. Das im Anschluss vorgestellte Verfahren, welches im Rahmen dieser Ausarbeitung entwickelt wurde, basiert ebenfalls auf den im Folgenden erläuterten Aspekten.

Die hier vorgestellten Ansätze ermitteln über eine Analyse der Spannungen eines Objekts einen möglichen Bruch und dessen Ausgangsposition. Die Effekte, die ein solcher Bruch mit sich bringt, können hierbei auffallend verschieden sein. Die Änderung der Gestalt eines Objekts, seine Materialeigenschaften oder vorkonfigurierte Bedingungen ermöglichen eine große Vielfalt an möglichen Brüchen.

Der allgemeine Bruchsimulations-Algorithmus der oben genannten Verfahren sieht wie folgt aus[18]:

1. Nach jedem Zeitschritt berechnet das System die internen Belastungen für jeden Massenpunkt.

2. An jedem Massenpunkt werden die berechneten Kräfte genutzt, um einen beschreibenden Tensor zu ermitteln.

3. Die Tensoren werden mit einem Parameter verglichen, welcher die Belastbarkeit des Materials angibt. Jedes Material hat hierbei einen individuellen Wert.

4. Sobald die Grenze der Belastbarkeit überschritten wurde, wird das Objekt an dieser Stelle gebrochen.

4.1.1 Berechnung der wirkenden Belastungen und Kräfte

Brüche entstehen aufgrund von internem Stress, welcher durch Materialdeformation entsteht. Für die richtige Darstellung von Brüchen wird eine Deformationsmethode benötigt, welche die Größe und Orientierung des internen Stresses liefert. Die Deformation/Verformung wird durch unterschiedliche Tensoren definiert, welche das Verhalten des entsprechenden Materials berücksichtigen.

Belastungs- und Belastungsratentensor Um die lokale Deformation eines Materials zu messen wird der Belastungstensor ϵ nach Green genutzt. Dieser wird dargestellt durch eine symmetrische 3×3 Matrix. Der Tensor beschreibt, wie sich die Weltkoordinaten $p[i]$ zu den Materialkoordinaten $u[i]$ verändert haben und ist definiert durch

$$\epsilon_{ij} = (\frac{\partial p}{\partial u_i} \cdot \frac{\partial p}{\partial u_j}) - \delta_{ij} \tag{6}$$

wobei δ_{ij} das Kronecker delta darstellt:

$$\delta_{ij} = \begin{cases} 1 & , i = j \\ 0 & , i \neq j \end{cases}. \tag{7}$$

Dieser Belastungstensor berücksichtigt nur die möglicherweise auftretenden Deformationen, nicht jedoch Transformationen des zu Grunde liegenden Objektes. Wenn beispielsweise lediglich eine Rotation des Objektes statt findet, verschwindet der Tensor ($\epsilon_{ij} = 0$).

Zusätzlich zum Belastungstensor wird der Belastungsratentensor ν bestimmt. Dieser Tensor misst die Veränderung der auftretenden Belastung und kann aus Gleichung 6 abgeleitet werden. Er wird definiert als:

$$\nu_{ij} = (\frac{\partial p}{\partial u_i} \cdot \frac{\partial v}{\partial u_j}) + (\frac{\partial v}{\partial u_i} \cdot \frac{\partial x}{\partial u_j}) \tag{8}$$

wobei v der Geschwindigkeit des Punktes und somit $\dot{p}$ entspricht.

Die Faktoren in den Formeln 6 und 8 dienen als Maß für die Deformation, aus der sich die Belastung der Elemente berechnen lässt. Sie sind festgelegt durch:

$$\frac{\partial p}{\partial u_i} = P\beta\delta_i$$

und

$$\frac{\partial v}{\partial u_i} = V\beta\delta_i$$

Hierbei ist δ_i das Kronecker-Delta als Vektor mit $[\delta_{i1}\delta_{i2}\delta_{i3}0]^T$dargestellt. P ist eine 3×4 Matrix und beinhaltet die Positionen der vier Eckpunkte ($p[i]$, Abbildung 4) des entsprechenden Elements in Weltkoordinaten. In V sind analog die dazugehörigen Geschwindigkeiten $v[i]$ gespeichert. Der Faktor β ist eine 4×4 Matrix und wird aus den Materialkoordinaten berechnet. Hierzu werden die Werte aus $u[i]$ eingetragen und die inverse Matrix gebildet.

$$\beta = \begin{pmatrix} u[0]_x & u[1]_x & u[2]_x & u[3]_x \\ u[0]_y & u[1]_y & u[2]_y & u[3]_y \\ u[0]_z & u[1]_z & u[2]_z & u[3]_z \\ 1 & 1 & 1 & 1 \end{pmatrix}^{-1}$$

Der Belastungstensor und der Belastungsratentensor enthalten die „Rohdaten“ für die Berechnung der internen Kräfte. Sie berücksichtigen jedoch nicht die Materialeigenschaften. Der im folgenden beschriebene Stresstensor kombiniert nun die Basisinformationen der Belastungstensoren mit den Eigenschaften des Materials.

Stresstensor Der Stresstensor σ ist eine symmetrische 3×3 Matrix, welche aus der Kombination von elastischem Stress $\sigma^{(\epsilon)}$ und Trägheitsstress $\sigma^{(\nu)}$ gebildet wird.

$$\sigma = \sigma^{(\epsilon)} + \sigma^{(\nu)} \tag{9}$$

Die erste Komponente, der elastische Stress, hat die Belastung ϵ als Grundlage und berechnet sich wie folgt:

$$\sigma_{ij}^{(\epsilon)} = \sum_{k=1}^{3} \lambda\epsilon_{kk}\delta_{ij} + 2\mu\epsilon_{ij}$$

λ und μ[8] sind die erste und zweite Lamé-Konstante des zu simulierenden Materials. Die Lamé-Konstanten beschreiben den Zusammenhang zwischen Deformation und Belastung innerhalb eines Körpers nach dem Hook'schen Gesetz[9]. Beide Werte haben die Einheit $\frac{N}{m^2}$. Konkrete Beispiele für Materialwerte der beiden Konstanten finden sich in Abschnitt 3.3.

Die zweite Komponente $\sigma^{(\nu)}$, der Trägheitsstress, berechnet sich sehr ähnlich und ist definiert als:

$$\sigma_{ij}^{(\nu)} = \sum_{k=1}^{3} \phi\nu_{kk}\delta_{ij} + 2\psi\nu_{ij}$$

[8] in der Literatur oft auch Schubmodul genannt

[9] Das Hook'sche Gesetz beschreibt das elastische Verhalten von Festkörpern.

Die zwei Werte ϕ und ψ bezeichnen die Dämpfungskonstanten. Sie definieren, wie schnell eine elastische Material-Verformung zurückgeht, wenn die Krafteinwirkung auf das Objekt verschwindet. Die Einheit für ϕ und ψ ist $\frac{N \cdot s}{m^2}$ und konkrete Beispielwerte befinden sich ebenfalls in Abschnitt 3.3.

Der Trägheitsstress$\sigma^{(\nu)}$ hat die Eigenschaft, dass nur interne Vibrationen abgeführt werden. Es werden keine Bewegungen gedämpft, die lokal und starr sind.

Innere Kraft Um die gesamte innere Kraft, welche ein Element auf einen seiner Massenpunkte ($i = \{0, 1, 2, 3\}$) ausübt, zu berechnen, werden die Dämpfungskraft $f_{[i]}^{(\nu)}$ und die elastische Kraft $f_{[i]}^{(\epsilon)}$miteinander addiert.

$$f_{[i]} = f_{[i]}^{(\epsilon)} + f_{[i]}^{(\nu)}$$

Der erste Summand, die elastische Kraft, ist der Teil der Kraft, der eine nicht-permanente Verformung des Objektes zur Folge hat. Diese Kraft wird in [17] definiert als die negative Ableitung der elastischen Potentialdichte η ([17] , Formel 9) nach $p[i]$, integriert über das Volumen des Elements. In diesem Anwendungsfall wird für einen Massenpunkt die elastische Kraft durch

$$f_{[i]}^{(\epsilon)} = -\frac{vol}{2} \sum_{j=1}^{4} p[j] \sum_{k=1}^{3} \sum_{l=1}^{3} \beta_{jl} \beta_{ik} \sigma_{kl}^{(\epsilon)} \tag{10}$$

mit[10]

$$vol = \frac{1}{6}(u[1] - u[0]) \cdot [(u[2] - u[0]) \times (u[3] - u[0])]$$

berechnet. Vergleichbar zu der elastischen Kraft berechnet sich die Dämpfungskraft für einen Massenpunkt. Hier wird jedoch der Trägheitsstress $\sigma^{(v)}$statt $\sigma^{(\epsilon)}$ als Grundlage genutzt.

$$f_{[i]}^{(\nu)} = -\frac{vol}{2} \sum_{j=1}^{4} p[j] \sum_{k=1}^{3} \sum_{l=1}^{3} \beta_{jl} \beta_{ik} \sigma_{kl}^{(\nu)} \tag{11}$$

Die exakte Herleitung der Formeln 10 und 11 ist nachzulesen in [17] ab Seite 139.

Nachdem die Kräfte, welche auf einen Massenpunkt wirken, berechnet sind, werden diese in Zug- (σ^+) und Druckkräfte (σ^-) aufgeteilt.

$$\sigma^+ = \sum_{i=1}^{3} \max(0, v^i(\sigma)) m(\hat{n}^i(\sigma)) \tag{12}$$

[10] Berechnung des Volumens eines Tetraeders durch: $V = \frac{1}{6}(\overrightarrow{AB} \circ \left(\overrightarrow{AC} \times \overrightarrow{AD}\right))$

$$\sigma^- = \sum_{i=1}^{3} \min(0, v^i(\sigma)) m(\hat{n}^i(\sigma)) \tag{13}$$

mit

$$m(a) = \begin{cases} aa^T/|a| & , a \neq 0 \\ 0 & , a = 0 \end{cases}.$$

$m(a)$ ist vereinfacht ausgedrückt die Matrixdarstellung eines Vektors, welche $|a|$ als Eigenwert und a als Eigenvektor hat[11]. Der in den Formeln 12 und 13 verwendete Wert $m(\hat{n}^i(\sigma))$ entspricht dann der Matrixdarstellung des i-ten Eigenvektors mit v^i als dazugehörigen Eigenwert. Da σ eine 3×3 Matrix ist, existieren drei Eigenwert/Eigenvektor-Paare ($i \in \{1, 2, 3\}$). In σ^+ werden durch die Berechnung die positiven Eigenwerte mit entsprechenden Vektoren gespeichert, und in σ^- die Negativen.

Ähnlich zur Formel der elastischen Kraft (siehe Formel 10) wird die positiv ($f^+_{[i]}$) und negativ ($f^-_{[i]}$) wirkende Kraft berechnet.

$$f^+_{[i]} = -\frac{vol}{2} \sum_{j=1}^{4} p[j] \sum_{k=1}^{3} \sum_{l=1}^{3} \beta_{jl} \beta_{ik} \sigma^+_{kl} \tag{14}$$

$$f^-_{[i]} = -\frac{vol}{2} \sum_{j=1}^{4} p[j] \sum_{k=1}^{3} \sum_{l=1}^{3} \beta_{jl} \beta_{ik} \sigma^-_{kl} \tag{15}$$

Separationstensor Die in den Formeln 14 und 15 berechneten Werte werden nun dafür genutzt um den Separationstensor zu ermitteln. Hierfür werden für einen Massenpunkt alle auf ihn wirkenden Kräfte berücksichtigt.

$$\varsigma = \frac{1}{2} \left(-m(f^+) + \sum_{f \in \{f^+\}} m(f) + m(f^-) - \sum_{f \in \{f^-\}} m(f) \right) \tag{16}$$

$\{f^+\}$ und $\{f^-\}$ werden von O'Brien und Hodgins [17] als Menge von Zug- und Druckkräften eingeführt. Sie beinhalten die Kräfte aller am Punkt anliegenden Elemente. f^+ ist definiert als die Summe über $\{f^+\}$ und bezeichnet die gesamte, nicht ausgeglichene, Zugkraft.

f^- bildet analog die Summe über $\{f^-\}$ und beschreibt die gesamte, nicht ausgeglichene, Druckkraft, welche auf einen Massenpunkt wirkt.

Anhand des Separationstensors ς lässt sich nun berechnen, ob eine Kraft ausreichend groß ist, um einen Bruch des Materials zu initiieren. Dafür wird

[11] Ein Eigenvektor $\vec{x}$ einer Matrix ist ein vom Nullvektor verschiedener Vektor, dessen Richtung durch Multiplikation mit der Matrix nicht verändert wird. Ein Eigenvektor wird also nur gestreckt. Der Streckungsfaktor λ heißt Eigenwert der Matrix. [20]

der höchste Eigenwert von ς ermittelt ($w_{\max(\varsigma)}$) und mit der Belastbarkeit eines Materials (τ)[12] verglichen.

$$w_{\max(\varsigma)} \geq \tau$$

Überschreitet dieser den Schwellwert, so muss ein Bruch des Objektes vollzogen werden. Der zu diesem Eigenwert korrespondierende Eigenvektor legt die Richtung des Bruchs fest: $\vec{v}_{\max(\varsigma)}$.

4.1.2 Modell

Die Objekte, die bei den hier vorgestellten Verfahren zerbrechen, werden durch dreidimensionale Tetraeder-Meshs repräsentiert. Das Objekt besteht dementsprechend aus einer Vielzahl von Elementen und Massenpunkten. Diese Darstellung ermöglicht eine flexible und performante Berechnung. Die einzelnen Teil-Elemente sind über gemeinsame Knoten miteinander verbunden. Die verwendete Objekt-Struktur ist durch die Delaunay-Triangulierung entstanden.

4.1.3 Verfahren nach Müller et al.

In ihrem Ansatz „Real-Time Simulation of Deformation and Fracture of Stiff Materials“ [14] stellen Müller et al. eine Methode für die Simulation von Brüchen und Deformationen in Echtzeit vor. Die präsentierte Technik ermöglicht die Deformation von dehnbaren Materialien, wie beispielsweise weichen Metallen, und die Bruchsimulation von spröden Materialien, wie Keramik oder Glas. In dieser Ausarbeitung wird lediglich der Teil der Bruchsimulation betrachtet.

Bruch Wie bereits in der allgemeinen Vorgehensweise beschrieben beginnt eine Bruchsimulation in diesem Fall immer mit der Berechnung der auf die Massenpunkte wirkenden Kräfte. Im Ansatz von Müller et al. werden die in Abschnitt 4.1.1 berechneten Tensoren nur ermittelt, wenn tatsächlich eine Kollision des Objekts vorliegt.

Sollte ein Objekt mit einem anderen kollidiert und die entsprechenden Tensoren berechnet sein, erfolgt der Reihe nach der Vergleich der an einem Punkt i wirkenden Kräfte mit der Materialhaltbarkeit.

$$w_{\max(\varsigma_i)} \geq \tau$$

$w_{\max(\varsigma_i)}$ gibt hierbei den höchsten Eigenwert des für den betrachteten Massenpunkt berechneten Separationstensors ς an (Details in Abschnitt 4.1.1 auf der vorherigen Seite). Der Schwellwert τ legt die Belastbarkeit des gegebenen Materials fest. Je niedriger dieser Wert ist, umso schneller zerbricht

[12] passende Werte für τ werden in Abschnitt 3.3 vorgestellt

das Objekt. In Tabelle 1 auf Seite 12 sind passende Materialparameter aufgezeigt.

Falls der Schwellwert überschritten wird, beginnt die eigentliche Simulation des Bruchs.

Zunächst wird aus den gegebenen Werten eine Schnittebene α ermittelt. Diese Ebene (auch „fracture plane“ genannt) basiert auf dem zum Eigenwert $w_{\max(\varsigma_i)}$ korrespondierenden Eigenvektor $\vec{v}_{\max(\varsigma_i)}$ und der Position p_i des Massenpunktes. Die Ebenengleichung der *fracture plane* sieht in Hessescher Normalform wie folgt aus:

$$\vec{x} \cdot \frac{\vec{v}_{\max(\varsigma_i)}}{\mid \vec{v}_{\max(\varsigma_i)} \mid} = \mid \vec{p_i} \mid, \tag{17}$$

mit $|\vec{p_i}| \geq 0$ als Abstand der Ebene zum Koordinatenursprung und $\frac{\vec{v}_{\max(\varsigma_i)}}{|\vec{v}_{\max(\varsigma_i)}|}$ als normierten Eigenvektor, welcher als Normale auf der Ebene steht. Die definierte Ebene besteht dann theoretisch aus genau den Punkten im Raum, dessen Ortsvektoren $\vec{x}$ die gegebene Gleichung 17 erfüllen.

Müller et al. nutzen die erstellte *fracture plane* nun, um ausgewählte Elemente voneinander zu trennen. Im Gegensatz zu O'Brien und Hodgins [17] beschränken sie sich dabei nicht nur auf die unmittelbar am beeinträchtigten Massenpunkt anliegenden Elemente. In ihrem Ansatz [14] werden alle Elemente, welche sich in einer festgelegten Umgebung befinden, berücksichtigt. Je nach Größe des Eigenwerts $w_{\max(\varsigma_i)}$ und den Materialeigenschaften wird ein Radius r_{frac} ermittelt, der die zu beeinflussenden Nachbarelemente bestimmt. Jedes Element e_j, welches einen Massenpunkt x_i besitzt, der im Umkreis von r_{frac} liegt, wird in die folgenden Berechnungen mit einbezogen. Je höher die Belastung $(w_{\max(\varsigma_i)})$ ist, umso mehr Elemente werden gegebenenfalls voneinander getrennt. In Abbildung 9 ist ein solcher Radius eingezeichnet, und die für das weitere Bruchverhalten beeinflussten Elemente des Objekts markiert. C definiert in der Abbildung den *crackpoint*, also den Massenpunkt, bei dem die Materialhaltbarkeit überschritten wurde.

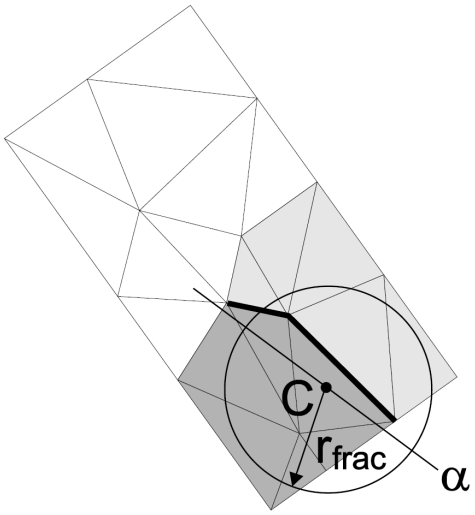


Abbildung 9: 2D-Ansicht auf ein Tetraeder-Mesh. Dreiecke werden abhängig ihrer Ausrichtung zur Schnittebene α markiert.[14]

Alle Elemente, die sich innerhalb des ermittelten Radius befinden, werden nun mit der *fracture plane* verglichen und mit + oder − markiert. Die Zuordnung zu diesen beiden Markierungen erfolgt durch den Vergleich des Schwerpunktes (engl. balance point) $\vec{b_j}$ des Elements mit dem Normalenvektor von α. Falls das Skalar der beiden Vektoren positiv ist, liegt das Element vorwiegend vor der Schnittebene, und es erfolgt eine Markierung mit einem + . Im anderen Fall liegt das Element zum Großteil hinter α und wird mit − gekennzeichnet.

$$\vec{v}_{\max(\varsigma_i)} \cdot \vec{b_j} \geq 0 \ : \ +$$

$$\vec{v}_{\max(\varsigma_i)} \cdot \vec{b_j} < 0 \ : \ -$$

In Abbildung 9 auf der vorherigen Seite sind die mit + markierten Tetraeder in dunkelgrau und die mit − markierten in hellgrau eingefärbt.

Der nächste Schritt des Bruch-Algorithmus ist die Trennung der Elemente mit unterschiedlichen Vorzeichen. Die Verbindungen der betroffenen Tetraeder werden gelöst. Die breitere schwarze Linie in Abbildung 9 kennzeichnet den Kontakt der in diesem Beispiel zu trennenden Dreiecks-Elemente. Sie definiert gleichzeitig die entstehende Bruchkante.

Da die zusammenhängenden Elemente gemeinsame Massenpunkte haben, wird die Topologie des Meshs verändert. Jeder Massenpunkt, der sich auf der Bruchkante befindet wird inklusive der beinhaltenden Informationen dupliziert und dem hinter der Bruchebene (Markierung: −) liegenden Element zugewiesen. Um eine Diskontinuität im Material zu vermeiden, muss nun an der Bruchkante ein so genanntes lokales „Remeshing“ durchgeführt werden. Die Elemente hinter der *fracture plane,* welche den neuen Massenpunkt zugewiesen bekommen haben müssen neu berechnet werden. Die exakte Neuberechnung der Verbindungen und Tetraeder ist im entsprechenden Implementierungsschritt (Paragraph: 6.2.3, Funktion: 19) beschrieben.

Da die Belastung innerhalb des Objekts in den meisten Fällen an mehreren Massenpunkten die Materialhaltbarkeit überschreitet, wird der soeben erläuterte Bruchvorgang mehrfach ausgeführt. Alle entsprechenden Massenpunkte lösen einen eigenen Bruch aus und zerteilen das Objekt in mehrere nicht mehr zusammenhängende Element-Gruppen.

4.1.4 Verfahren nach Glondu et al.

Glondu, Marchal und Dumont stellen in ihrer Veröffentlichung „Real-Time Simulation of Brittle Fracture Using Modal Analysis.“ [7] von 2013 einen Ansatz vor, der ebenfalls eine echtzeitfähige Simulation von Brüchen spröder Körper betrachtet.

Das vorgestellte Verfahren kann in zwei Schritte unterteilt werden:

1. Initiierung des Bruchs aufbauend auf der Analyse der Eigenwerte

2. Durchführung eines auf Energie basierenden Bruch-Algorithmus

Die Besonderheit dieses Algorithmus ist die Berechnung und Simulation einer „damped deformation wave“ , einer Art Druckwelle, welche sich innerhalb eines Objekts ausbreiten kann. Diese „Welle“ teilt die Elemente und löst den Bruch aus.

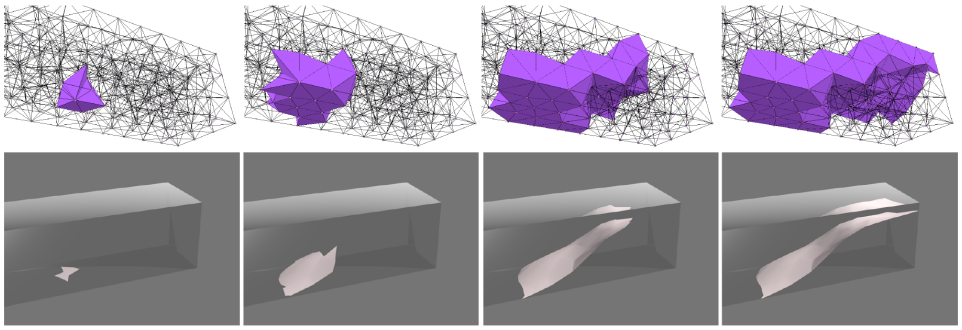


Abbildung 10: Entwicklung eines Bruchs innerhalb eines Tetraeder-Mesh *Oben:* Elemente die von dem *fracture surface* geschnitten wurden *Unten:* Das zum oberen Mesh korrespondierende *fracture surface [7]*

Bruch Bei jedem Kontakt zweier Objekte wird eine Analyse der wirkenden Kräfte durchgeführt.

Sollte der auf ein Element wirkende Separationstensor ς dabei den erläuterten Materialhaltbarkeits-Wert τ übersteigen, wird eine Bruchsimulation initiiert. Um die Inhomogenität des Materials zu simulieren, werden fehlerhafte, schwache Elemente in die Objekte eingefügt. Die Brüche werden dann an den schwachen Elementen e_0 gestartet, die sich am nächsten zu dem Element befinden, welches den Bruch veranlasst hat. Der Bruch wird genau am Schwerpunkt b_0 dieser Elemente begonnen. Die Richtung des Bruchs wird durch das „fracture surface“ S vorgegeben. S ist eine Fläche, die durch drei orthogonal zueinander stehende Vektoren und der Position von b_0 definiert ist. Der auf der Fläche stehende Vektor ist genau wie in der von Müller et al. vorgestellten Darstellung der zum Eigenwert $w_{\max(\varsigma_i)}$ korrespondierende Eigenvektor $\vec{v}_{\max(\varsigma_i)}$.

Das berechnete *fracture surface* wird nun dafür genutzt, die umliegenden Elemente des Meshs zu „besuchen“. Ausgehend vom initialen Element e_0 werden alle Nachbarelemente überprüft, ob sie von S geschnitten werden. Dies ist der Fall, wenn einer der vier Massenpunkte p_i vor und ein anderer hinter dem *fracture surface* liegt. Diese Vorgehensweise wird nun durch das Objekt fortgeführt. Diejenigen Elemente, welche von S geschnitten werden, initiieren eine weitere Untersuchung ihrer Nachbarelemente. Die Reihenfolge der ausgehend vom Element e_0 beschrittenen Tetraeder basiert hierbei auf dem aus der Informatik bekannten Verfahren der Breitensuche.

Um im späteren Verlauf den eigentlichen Bruch durchzuführen, wird bei der Überprüfung der Elemente das „set of crossed elements“[13] ξ erstellt. Dieses *set* speichert alle Elemente, die vom *fracture surface* geschnitten wurden. Sollte sich ein Element bereits in diesem *set* befinden, so wird es nicht erneut untersucht. Bei der Aufnahme des Elements in das *set* werden für jeden Knoten zwei Informationen abgespeichert. Zum einen, durch welches *fracture surface* das Element geteilt wurde und zum anderen, auf welcher Seite es sich befindet. Der Eintrag für einen Knoten liegt dann in der Form $(h_{s_i}, 0)$ oder $(h_{s_i}, 1)$ vor. h_{s_i} bezeichnet hierbei das aktuelle *fracture surface* und 0 bzw. 1 definieren, ob sich der Knoten vor (1) oder hinter (0) der Schnittebene befindet.

In Abbildung 10 auf der vorherigen Seite ist die Verbreitung eines Bruchs durch ein Objekt abgebildet. Die oberen Bilder zeigen den Inhalt von ξ und somit alle geschnittenen Elemente. Von links nach rechts steigt die Anzahl der dem *set* zugewiesenen Elemente.

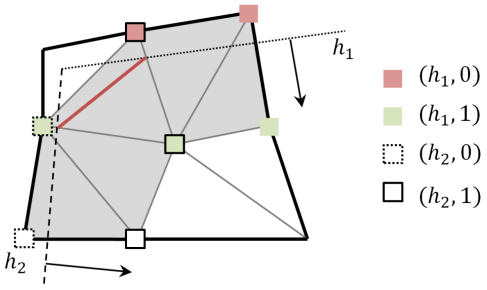


Abbildung 11: Zuordnung der Knoten zu den entsprechenden Fragmenten[7]

Um die Ausbreitung eines Bruchs zu unterbrechen, haben Glondu et al. ein Energie-basiertes Abbruchkriterium entwickelt. Hierfür wird zunächst der Energiebetrag, der für die Ausbreitung innerhalb des Objekts notwendig ist, stückweise angenähert. Glondu et al. [7] geben diese Kraft als „fracture energy“ E_f an und definieren sie wie folgt:

$$E_f = \sum_{e \in \xi} A_e \cdot G_c, \tag{18}$$

wobei A_e das Gebiet des *fracture surface* jenes ist, welches das Element e durchkreuzt. G_c ist ein Parameter des Materials, der den Bruchwiderstand bzw. die Risszähigkeit angibt. Je kleiner dieser Wert ist umso größer ist der „Widerstand“ des Materials gegenüber der Verbreitung eines Bruchs.

In ähnlicher Weise wird die verfügbare Energie E_s berechnet.

$$E_s = \sum_{e \in \xi} A_e \cdot \gamma \cdot \eta_e, \tag{19}$$

mit η_e als spezifische Verzerrungsenergie von Element e und γ als konstanten Faktor.

Die Energie E_s wird als Schwellwert genutzt, um gegebenenfalls die Fortsetzung eines Bruchs zu unterbinden. Wenn ein neues Element e_j überprüft

[13] Das *set* ist eine Speicherform in der Informatik.

wird, muss die Gleichung $E_f + A_{e_j} \cdot G_c < E_s + A_{e_j}\gamma \cdot \eta_{e_j}$ erfüllt sein, damit der Bruch sich fortsetzt. Die linke Seite setzt sich zusammen aus der bisher genutzten Energie und der für das aktuell untersuchte Element notwendigen Energie. Die rechte Seite der Gleichung zeigt die bisher berechnete, verfügbare Energie, addiert mit der Energie des untersuchten Elements.

Nachdem durch die Verbreitung des Bruchs ein *set* aus zu teilenden Elementen entstanden ist, können durch die eingetragenen Werte $(h_{s_i}, 0)$ bzw. $(h_{s_i}, 1)$ die potentiellen Bruchstücke (oder auch Fragmente) berechnet werden. Zwei Knoten gehören dem selben Fragment an, falls sie den selben Eintrag haben, oder mindestens einer der beiden Knoten nicht markiert ist, aber zum selben Element gehört. In Abbildung 11 auf der vorherigen Seite ist dieser Prozess zu sehen. In dem gezeigten Beispiel wurde das Objekt durch die beiden *fracture surfaces* h_1 und h_2 geschnitten. Die grau markierten Elemente befinden sich im *set of crossed elements*. Durch die vorher abgespeicherten Einträge der Knoten können diese unterschiedlichen Bruchstücken zugewiesen werden. Ausgehend von einem Knoten werden alle Knoten der Reihe nach untersucht und dem richtigen Fragment zugeordnet. Abbildung 12 zeigt die drei generierten Fragmente mit den korrespondierenden Knoten in *gelb, blau* und *rot.*

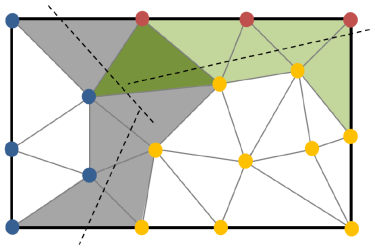


Abbildung 12: Durch die Zuweisung ermittelte Fragmente[7]

Nach dieser Zuweisung der Knoten wird das Resultat, also die eigentlichen Bruchstücke, generiert.

Erweiterung Für die Modellierung der Bruchkanten nutzen Glondu et al. ein „Remeshing"-Verfahren, welches auf dem beschriebenen Schneiden des Objekts aufbaut. Zu diesem Zweck wird für jede Kante, die durch das *fracture surface* geschnitten wird, der Schnittpunkt zwischen Ebene und Kante berechnet und gespeichert. Alle ermittelten Punkte, die dem selben Fragment angehören, werden dann zur Modellierung der sichtbaren Bruchkante genutzt. In Abbildung 12 wird diese Bruchkante durch die gestrichelte Linie gekennzeichnet.

4.1.5 Eigenes Verfahren

Die Idee für den hier erläuterten Ansatz entstand nach der Implementation des Verfahrens nach Müller et al. und der weiteren Recherche in der Arbeit „Physically-based and Real-time Simulation of Brittle Fracture for Interactive Applications" [6]. In dieser Arbeit wurden Benutzer-Tests durchgeführt, bei denen mehrere Probanden zwischen vorbereiteten Bruchmus-

tern entscheiden mussten. Auf den am häufigsten ausgewählten Mustern war deutlich zu sehen, dass die Brüche in einer Beziehung zu einander stehen. In dieser Arbeit wurden des Weiteren mechanische Tests durchgeführt, wie sich Brüche in der Realität verhalten. Sollte ein Bruch sich noch nicht durch das komplette Objekt vollzogen haben, so ist die Wahrscheinlichkeit für eine Fortsetzung dieses Bruches bei erneuter Belastung sehr hoch. Im Ausschnitt der Fotografie in Abbildung 13 ist zu erkennen, dass entstandene Brüche in einer Beziehung zueinander stehen bzw. sich verzweigen.

Aufgrund seiner im Folgenden erläuterten Technik wird das Verfahren in dieser Ausarbeitung als „secondary-Splits"-Verfahren bezeichnet.

Abbildung 13: Sich verzweigende Risse[6]

Bruch Der selbst entwickelte Ansatz basiert ebenfalls auf den aus der Kontinuumsmechanik bekannten Berechnungen der Belastungen. Nach einer Kollision wird zunächst die größte wirkende Kraft ermittelt. Dieser Parameter ist der höchste Eigenwert des Separationstensors ς. Er wird zur einheitlichen Benennung in diesem Verfahren w_{init} ($w_{init} = w_{\max(\varsigma)}$) genannt. Dieser Eigenwert hat einen zugehörigen Punkt P_{init} der gleichzeitig der Ausgangspunkt des möglichen ersten Bruchs ist. Zunächst erfolgt jedoch der Vergleich der wirkenden Kraft mit der definierten Materialhaltbarkeit.

$$w_{init} \geq \tau$$

Falls der Schwellwert überschritten wird, wird ein initialer Bruch des Objekts durchgeführt. Dieser erste Bruch basiert auf dem zum Eigenwert w_{init} korrespondierenden Eigenvektor $\vec{v}_{init}$. Dieser Vektor steht senkrecht auf der durch das Objekt führenden *fracture plane* α_{init} .

Der Schwerpunkt B (engl. balance point) der einzelnen Elemente des Objekts wird nun auf seine Lage gegenüber α_{init} überprüft. Hierzu wird der Vektor der Stecke zwischen B und P_{init} mit dem Vektor $\vec{v}_{init}$ verglichen. Sollte das Skalar der beiden Vektoren positiv sein, so ist der entsprechende Schwerpunkt vor der Ebene α_{init}, im anderen Fall hinter ihr.

$$\overrightarrow{P_{init}B} \cdot \vec{v} \geq 0 \ : \ +$$

$$\overrightarrow{P_{init}B} \cdot \vec{v} < 0 \ : \ -$$

Alle Elemente, die bei der Überprüfung vor der Ebene lagen, werden nun zu einem Bruchstück zusammengefasst. Aus den hinter der Ebene liegenden ergibt sich ein zweites. Während des Unterteilens des ursprünglichen Objekts werden alle Massenpunkte, die sich an dieser Bruchkante befinden, als *„cracknodes“* N_{crack} markiert.

Nachdem der initiale Schnitt durchgeführt wurde, werden die folgenden Schnitte nur noch an einem dieser markierten Massenpunkte beginnen. Der Parameter, der die Anzahl der sekundären Schnitte festlegt, lautet c_s. Diese Anzahl der nun folgenden Schnitte kann entweder durch den Anwender festgelegt, oder aus der initialen Belastung w_{init} abgeleitet werden. Eine Ableitung aus w_{init} wäre physikalisch plausibler, wird aber in dieser Ausarbeitung zur besseren Stabilität durch einen benutzerdefinierten Wert ersetzt.

Die folgenden Schritte werden c_s-mal ausgeführt. Zunächst wird der mit dem höchsten Eigenwert und als N_{crack} markierte Massenpunkt gesucht. Dieser Punkt P_{new} wird als neuer potentieller Ausgangspunkt für einen weiteren Schnitt genutzt. Ein neuer Bruch darf nur entstehen wenn die folgende Formel erfüllt ist:

$$w_{new} \geq \frac{\tau}{f}, \tag{20}$$

mit w_{new}, als Eigenwert von Massenpunkt P_{new} und $f \geq 1$ als Parameter, der die Zerbrechlichkeit (engl. fragility) des Materials wiedergibt. $\frac{\tau}{f}$ hat die Wirkung, dass bei den sekundären Schnitten die nötige Belastung, welche zu einem Bruch führt, niedriger als beim ersten Bruch ist. Sollte die Gleichung 20 erfüllt sein, wird ein weiterer Bruch initiiert. Die für den Bruch notwendige *fracture plane* α_{new} ergibt sich aus der Kombination des Vektors, der auf der initialen Schnittebene α_{init} steht, und dem zu P_{new} korrespondierenden Eigenvektor $\vec{v}_{new}$. Die Summe dieser beiden Vektoren ergibt den Vektor, der senkrecht auf α_{new} steht. Analog zu dem oben gezeigten Verfahren werden nun die Elemente des Objekts durch α_{new} aufgeteilt und die geteilten Massenpunkte als N_{crack} markiert.

4.2 Verfahren nach Smith et al.

Physikalisch korrekte Animationen können unter Umständen bei großen Datenmengen die Berechnungen nicht in Echtzeit durchführen. Der Grund dafür ist vor allem, dass echte Bewegungen und eine realitätsnahe Zersplitterung der Objekte extrem zeitintensiv sind. Eine solche Genauigkeit ist bei vielen Animationszwecken jedoch nicht unbedingt erforderlich und kann daher oft vernachlässigt werden, ohne große visuelle Nachteile zu haben. Durch lediglich „physik-gestützte“ Animationstechniken können realistisch aussehende Brüche mit weitaus weniger Arbeitsaufwand und trotzdem der notwendigen visuellen Präzision erstellt werden. Im Gegensatz zu den Verfahren aus

der Kontinuumsmechanik werden bei diesen Techniken manche physikalische Eigenschaften außer Acht gelassen.

Smith, Wittkin und Baraffz [21] haben 2001 ihren Ansatz „Fast and Controllable Simulation of the Shattering of Brittle Objects“ vorgestellt. Ihr System kombinierte Methoden der Starrkörpersimulation mit einem einschränkungsbasierten Modell, um das Zerbrechen von beliebigen Polyedern zu animieren. Für eine potentielle Bruchinitiierung wurden wirkende Kräfte und festgelegte Bedingungen miteinander verglichen.

4.2.1 Modell

Im Ansatz nach [21] ist die initiale Beschreibung eines Modells eine Oberfläche, welche durch Punkte und Dreiecke beschrieben wird. Durch das Hinzufügen von virtuellen Punkten und einer Triangulierung wird diesem Objekt ein Körper zugeordnet. Die Objekte werden nun durch eine Reihe von Elementen dargestellt. Jedes dieser Elemente definiert in seinem Schwerpunkt, die aktuelle Position des Elements, seine Masse und Dichte.

Zwei Elementen, welche über eine gemeinsame Oberfläche verbunden sind, wird in dem vorgestellten Ansatz eine lineare Bedingung (engl. constraint) zugeordnet. Diese Bedingung wird als abstandserhaltend definiert und führt dazu, dass benachbarte Elemente ihre Positionen zueinander nicht verändern. In Abbildung 14 sind zwei benachbarte Elemente zu sehen, welche über diese starre oder auch unelastische Bedingung (engl. rigid constraint) verbunden sind.

Diese Bedingung, welche zwischen den beiden Massenpunkten der Elemente liegt, soll die physikalische Stärke des Bandes zwischen den Mikro-Fragmenten des Ursprungsobjekt wiedergeben. Das Maß, welches dieser Verbindungsstärke entspricht, ist die *connection strength* S. Sie richtet sich nach der Größe der geteilten Fläche. Je größer das so genannte „shared face“ ist, umso stärker ist der Zusammenhalt zwischen den beiden Elementen. Der für diesen Zweck ermittelte Wert S ist hierbei proportional zur gemeinsame Oberfläche.

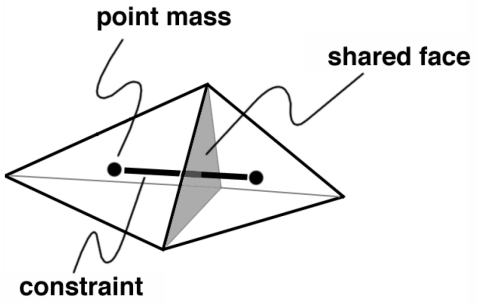


Abbildung 14: Zwei benachbarte Elemente mit linearer Bedingung[21]

Brüche entstehen bevorzugt an Kerben oder schwachen Stellen eines Objekts. In der auf [21] aufbauenden Arbeit [10] zeigen Kücükyilmaz und Özgüc, wie Unstetigkeiten in die initialen Objekte eingebaut werden können, um sie realistischer zerbrechen zu lassen. Hierfür wird der Zusammenhalt des Objektes bei der Generierung durch Rauschfunktionen oder virtuelle Spalten

verändert. Die dadurch vermiedene Gleichförmigkeit innerhalb des Objektes führt zu realistischeren Brüchen. Der Modellierer eines Objektes hat zusätzlich die Möglichkeit, bestimmte Constraints zu verstärken, um Bereiche während eines Bruchs intakt zu lassen, oder sie erst bei starkem Druck brechen zu lassen.

4.2.2 Bruchbedingung

Bei der Berechnung des Bruchs findet keine Deformation der Objekte statt. Entweder das Objekt zerbricht komplett, oder es findet keine äußerlich sichtbare Veränderung statt. Um einen Bruch zu initiieren, muss eine gewisse Kraft auf die Oberfläche oder innerhalb des Objektes wirken. Ein Bruch zwischen zwei Elementen erfolgt, sobald der vorgegebene Grenzwert der *connection strength* überschritten wurde. Dieser Schwellwert hängt wie oben beschrieben von der Geometrie der zusammenhängenden Elemente ab. Zusätzlich können benutzerspezifische Funktionen und Materialparameter diesen Wert anpassen.

Für die Berechnung der Kraft, welche auf einen Partikel wirkt und mit S verglichen wird, müssen einige Schritte durchgeführt werden. Die für jeden Massenpunkt vorgestellten Werte p, m und f werden für diese Berechnungen gebündelt und als Einheit betrachtet. P ist demnach ein Vektor, der alle Positionen der Schwerpunkte beinhaltet, der Vektor F beinhaltet die entsprechenden Kräfte und M ist eine Matrix, welche die Masse der Elemente in der Diagonalen speichert.

Um das Verhalten zweier Massenpunkte zueinander zu definieren, wird ihnen eine Funktion zugeordnet. Diese „behaviour function“ [25] C gibt benutzerdefinierte Bedingungen wieder. In diesem Fall ist die Bedingung der Zusammenhalt der Massenpunkte über einen vordefinierten Abstand:

$$C_i(p_a, p_b) = \parallel p_a - p_b \parallel - d_i, \tag{21}$$

wobei $p_a$und p_b die Positionen der Massenpunkte der beiden verbundenen Elemente sind und d_i der einzuhaltende Abstand.

Die in Formel 21 gezeigte *behaviour function* ist so aufgebaut, dass das Ergebnis 0 wird, wenn die Vorgaben („*things are happy*“ [25]) erfüllt sind. Die hier gezeigte Konfiguration entspricht einer Feder, welche sich im Ruhezustand befindet und keine Belastungen ausgleichen muss. Aus der *behaviour function* soll nun die „constraint force“ ermittelt werden, welche die wirkenden Kräfte widerspiegelt und den Vergleichswert für das Bruchkriterium angibt. Die Berechnung basiert auf folgendem Schema:

1. Es wird angenommen, dass die Ausgangswerte der *behaviour function* für Position und Geschwindigkeit gültig sind:$C(P) = 0, \dot{C}(P) = 0$.

2. Ein *constraint force vector* $\hat{F}$ soll nach Addition zu F (gesamte Kraft) gewährleisten, dass auch $\ddot{C}(P) = 0$ ist.

Es wird also eine Kraft (*constrain force vector* $\hat{F}$) gesucht, welche auf das Objekt einwirkt und die abstandserhaltende Bedingung aufrecht erhält.

Um die *behaviour function* abzuleiten, wird die Jakobi- oder auch Ableitungsmatrix[14] zur Hilfe genommen. Die erste und zweite Ableitung von $C(P)$ sind dadurch definiert als:

$$\dot{C}(P) = J\dot{P},$$

und

$$\ddot{C}(P) = \dot{J}\dot{P} + J\ddot{P} \tag{22}$$

mit der partiellen Ableitung nach Jakobi $J = \frac{\partial C}{\partial P}$.

Aus dem zweiten Newtonschen Bewegungsgesetz (siehe Formel 1 auf Seite 7) folgt $\ddot{P} = M^{-1}F$, mit $\ddot{P}$ als Beschleunigung der Schwerpunkte und M^{-1}als Inverse der Massenmatrix. Durch Ersetzen von $\ddot{P}$ folgt aus Formel 22 :

$$\ddot{C}(P) = \dot{J}\dot{P} + JM^{-1}(F + \hat{F}), \tag{23}$$

mit $\hat{F}$ als gesuchtem *constraint force vector.*

Da, wie in 2 beschrieben, $\ddot{C} = 0$ sein muss entsteht aus der Umformung von 23:

$$JM^{-1}\hat{F} = -\dot{J}\dot{P} - JM^{-1}F \tag{24}$$

Um die Ausgewogenheit innerhalb des Objektes beizubehalten, muss sichergestellt sein, dass die *constraint forces* keine tatsächliche Änderung des Objektes durchführen, sondern lediglich als Vergleichswert dienen. Alle Vektoren, die diese Voraussetzung erfüllen, werden von [10] als:

$$\hat{F} = J^T\lambda \tag{25}$$

beschrieben. λ ist hierbei ein Vektor mit der selben Dimension wie C, mit Lagrange-Multiplikatoren [15] als Komponenten. Diese Komponenten definieren, wie groß der Einfluss einer bestimmten Bedingung auf den berechneten *constraint force vector* ist. Eine beispielhafte Darstellung und Berechnung der Lagrange-Multiplikatoren findet sich in der Arbeit „Dynamiksimulation in der Computergraphik“ [4] von Jan Bender ab Seite 22.

[14] Die Jacobi-Matrix (auch Funktionalmatrix) dient zur näherungsweisen Berechnung mehrdimensionaler Funktionen in der Mathematik.http://www.formel-sammlung.de/ld-Jacobi-Matrix-938.html

[15] Die Idee hinter dem Verfahren der Lagrange-Multiplikatoren ist recht einfach: Man erweitert die Zielfunktion f, indem man jede Nebenbedingung mit einer zusätzlichen Variablen multipliziert (Bezeichnung: $\lambda_1, \lambda_2, ...$) und diese auf die Zielfunktion addiert. Das Maximum bzw. Minimum der Zielfunktion unter den Nebenbedingungen kann gefunden werden, indem das Maximum bzw. Minimum dieser neuen Funktion bestimmt wird.[9]

Abschließend folgt aus den Formeln 24 und 25 :

$$JM^{-1}J^T\lambda = -\dot{J}\dot{P} - JM^{-1}F \tag{26}$$

Da alle weiteren Variablen der Formel 26 bekannt sind, kann nun λ berechnet werden. Durch Einsetzen von λ in Formel 25 kann nun schließlich der gesuchte Vektor $\hat{F}$ ermittelt werden. Dieser Vektor erfüllt nun alle aufgestellten Bedingungen.

Falls diese *constraint force* nun größer ist als die Stärke der Verbindung ($\hat{F} \geq S$), wird diese Verbindung entfernt.

Im anderen Fall ($\hat{F} < S$) wird die bestehende Verbindung um den Wert der *constraint force* geschwächt: $S' = S - \hat{F}$.

4.2.3 Erweiterungen

Zusätzlich zu der Haupt-Bruchbedingung wurden von Smith, Witkin und Baraff in [25] und [21] weitere Modifikationen vorgestellt, welche ein realistischeres Bruchbild erzeugen.

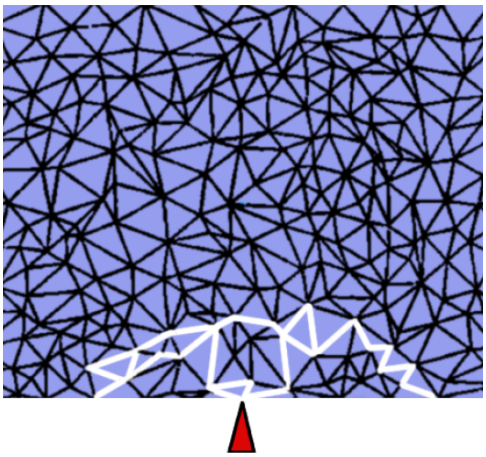


Abbildung 15: Mesh-Objekt mit initiiertem Bruch und Bruch-Wachstum[21]

In [25]werden zwei zusätzliche Parameter eingeführt. k_s, als Konstante, welche die Starrheit (engl. stiffness) des Materials angibt und k_d als Variable zur Beschreibung der Dämpfung. Diese beiden Werte, werden nun bei Bedarf in der Berechnung des *constraint force vectors* berücksichtigt. In Abschnitt 3.2 von [21] wird hierzu passend auch die physikalische Ausführbarkeit der beschriebenen Lösung diskutiert. Smith, Witkin und Baraff führen in ihrer Ausarbeitung an, dass bei gleicher Krafteinwirkung auf ein Objekt die Verbindungen bei einer Dehnbeanspruchung wesentlich schneller brechen als bei einer Stauchung.

Ein weiteres physikalisches Phänomen, dass von Smith, Witkin und Baraff in ihren Modifikationen berücksichtigt wird, ist das Wachstum von Rissen. Bei spröden Körpern ist die Energie, welche aufgebracht werden muss, um einen neuen Bruch zu initiieren, sehr viel niedriger, wenn sich bereits ein Bruch/Riss in der Nähe befindet. Dieses Verhalten ist der Grund dafür, dass beispielsweise Keramik in vorzugsweise große Stücke und nicht in eine Unmenge von Einzelteilen zerbricht. In ihrem vorgestellten Ansatz wird diese Gegebenheit in den Bruchprozess eingebunden. Sobald eine bestehende Verbindung zwischen zwei Elementen entfernt wird bzw. zerbricht, werden alle

nahe liegenden Verbindungen um einen Faktor geschwächt. Dies führt dazu, dass anliegende Elemente im weiteren Verlauf der Simulation schneller brechen, als weiter distanzierte. Wenn die *connection strength* durch die Reduzierung bereits sehr niedrig liegt, kann durch diesen Vorgang auch direkt ein weiterer Bruch entstehen. Um welchen Faktor die umliegenden Verbindungen reduziert werden sollten, liegt an den Eigenschaften des zerbrechenden Objekts. In [21] werden Funktionen (*crack growth function*) vorgestellt, welche die bestehende Stärke der Verbindung um einen Faktor ≤ 2 bzw. ≤ 1000 reduzieren. Das folgende Beispiel zeigt, wie die *connection strength* S auf diese Art und Weise geschwächt wird:

$$S_{new} = S_{old}(1.0 - \sin(2\theta + \frac{\pi}{2})),$$

wobei $\theta (\in [-\frac{\pi}{2} \ldots \frac{\pi}{2}])$ der Winkel zwischen der gebrochenen und der hier betrachteten, benachbarten Verbindung ist.

Der Wert der Reduzierung wird demnach von einem konstanten Faktor und der Ausrichtung der Verbindungen zueinander bestimmt. In Abbildung 15 ist zu sehen, wie sich ein ausgelöster Bruch auf die umliegenden Elemente auswirken kann. Die weiß markierten Verbindungen entsprechen dem berechneten Bruchmuster.

Die vorgestellte Vorgehensweise vermeidet die Konstruktion zu vieler neuer Brüche und gibt den an einem Bruch anliegenden Elementen die Möglichkeit, diesen fortzuführen. Realistische Bruchkanten sind infolgedessen das Ergebnis.

5 Physik- und Modellgenerierung

Als Grundlage für die Implementierung der vorgestellten Verfahren wurde die Physik- Bullet[16] genutzt. Diese Physik- bietet dem Nutzer die Möglichkeit, verschiedene Komponenten zu verwenden, um eine Physik-Simulation zu entwickeln. Die Gründe für die Nutzung dieser sind die guten Weiterentwicklungs-Möglichkeiten und die lediglich optionale Nutzung der bereitgestellten Funktionen. Dies ermöglicht eine zielgerichtete Implementierung ohne die Nachteile einer überladenen Anwendung.

5.1 Bullet Physics

Bullet Physics ist eine professionelle „open source“ Physik-, welche die Simulation von starren („rigid bodys“) und deformierbaren („soft bodys“) Objekten ermöglicht. Sie bietet eine Kollisionserkennung und unterschiedliche Einschränkungen[17] für die Objektsteuerung an. Es wurden bereits mehrere Plug-Ins für Maya, Blender oder Cinema 4D veröffentlicht.[5]

[16] Bullet Physics Library, http://bulletphysics.org/wordpress/
[17] engl. constraints

Die für diese Ausarbeitung wichtigen Komponenten der Architektur sind in Abbildung 16 grün eingefärbt und werden in diesem Abschnitt kurz vorgestellt. Zusätzlich wird auf einige weitere BULLET-spezifische Eigenschaften eingegangen.

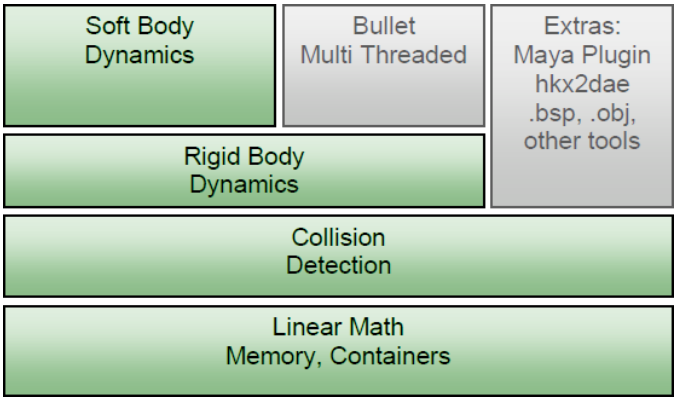


Abbildung 16: Komponenten der BULLET PHYSICS-[5]

Das Herzstück von BULLET PHYSICS ist die *btDiscreteDynamicsWorld*-Klasse, in welcher sämtliche Datenstrukturen und Berechnungen zusammenführen. Der Entwickler hat hier die Möglichkeit die gewünschte Kollisionsmethode, Objekte und deren Verhalten festzulegen.

Kollision Der Nutzer hat in der BULLET PHYSICS- mehrere Optionen eine Kollisionsüberprüfung für seine virtuelle Welt und die darin befindlichen Objekte zu implementieren.

Es ist dabei wichtig, den Objekten, welche kollidieren könnten, ein *btCollisionObject* zuzuweisen. Dieses *btCollisionObject* repräsentiert das Objekt aus der virtuellen Welt bei anstehenden Kollisionen. Jedem *btCollisionObject* wird eine zur Form des Objekts passende Gestalt zugeordnet, das so genannte *btCollisionShape*. Es besteht je nach Anforderung aus einer sehr simplen Form, wie der einer Kugel, oder aus relativ komplexer Geometrie. Der Entwickler sollte die Geometrie des *btCollisionShape* jedoch möglichst simpel halten, um eine schnelle Kollisionserkennung zu ermöglichen. In dem Manual für die BULLET PHYSICS-Enigne ([5]) ist auf Seite 18 ein Leitfaden zur Identifizierung des passenden „Collision Shapes“ zu finden. In unserer Ausarbeitung wird vor allem die konvexe Hülle (*btConvexHullShape*) zur Repräsentation genutzt.

Für die Kollisionserkennung an sich stellt BULLET die Möglichkeit zur Verfügung, die Kollisionsüberprüfung in zwei Phasen zu unterteilen. Zum einen die „Broad Phase“, in der überprüft wird, welche Objekte sich überhaupt überlagern könnten. Und die „Narrow Phase“, in der die Objekte auf mögliche Schnittpunkte untersucht werden. Diese Unterteilung macht vor

allem bei großen virtuellen Welten und komplexen Objekten Sinn, da Rechenleistung eingespart werden kann. In BULLET-PHYSICS geschieht diese Unterteilung durch die Klassen *btAxisSweep3* (Alternativ: *btDbvtBroadphase* und *bt32BitAxisSweep3*) für die „Broad Phase“ und den *btDispatcher* für die „Narrow Phase“.

Soft Bodys In der Physik- BULLET hat der Nutzer die Möglichkeit, seine Objekte als so genannte „Soft Bodys“ (Klasse: *btSoftBody*) darzustellen. Im Gegensatz du den „Rigid Bodys“ werden diese nicht durch eine einzige Koordinate mit zusätzlichen Objekt-Infos dargestellt, sondern durch mehrere Massenpunkte und deren Verbindungen. Diese Punkte werden in BULLET als *Nodes* bezeichnet und haben die Form und Parameter des in Abschnitt 3.1 vorgestellten Massenpunktes. Sie stellen zusammen mit den Tetraedern (in BULLET*: Tetras*) die in Abschnitt 3.2 aufgezeigte Unterteilung der Elemente dar. Diese Darstellung erlaubt es dem Benutzer in BULLET, die „Soft Bodys“ zu deformieren. Bereits in BULLET implementiert sind Funktionen zur Darstellung der Verformung von Tüchern, Seilen oder anderen elastischen Objekten. Die Verformung dieser Objekte konnte als Grundlage für die Berechnung der wirkenden Kräfte genutzt werden.

Jeder „Soft Body“ wird zusätzlich zu den Knoten,Verbindungen und Tetraedern über Konfigurations- und Material-Parameter definiert, welche das Verhalten bei Kollisionen festlegen. Der Koeffizient *Material::m_kVST* beschreibt beispielsweise, wie steif sich das Volumen des Objektes verhält.

Für die Kollisionserkennung dieser Objekte werden die Elemente der Soft Bodys zu größeren Einheiten zusammengefasst. Die Überprüfung jedes Tetraeders würde einen großen Performance-Verlust zur Folge haben. Um Rechenleistung einzusparen, werden von Bullet so genannte *Cluster* genutzt. Ein *Cluster* stellt eine Gruppe von Massenpunkten und Elementen der selben Region dar. Die Methode *generateClusters()* führt eine automatische Berechnung von konvexen, verformbaren Clustern durch. Jedem dieser Cluster kann ein *btCollisionObject* zugewiesen werden. Im Ergebnisse-Abschnitt dieser Ausarbeitung sind diese Cluster beispielsweise in Abbildung 39 auf Seite 75 zu sehen.

Rigid Bodys *btRigidBody* ist die Hauptklasse der „Rigid-Body“-Objekte. Es werden drei Typen dieser Objekt-Klasse unterschieden:

1. Statische Objekte, welche meist zur Repräsentation von sehr großen Einheiten, wie zum Beispiel eines Geländes (engl. terrain) genutzt werden. Sie haben eine Masse von Null und können sich nicht bewegen.

2. Dynamische Objekte mit positiver Masse, deren Position und Ausrichtung in jedem Zeitschritt aktualisiert werden.

3. Kinematische Objekte, welche vollständig durch den Nutzer kontrolliert werden müssen.

Lediglich die ersten beiden Typen, der „Rigid-Bodys“ sind in den Implementierungen dieser Ausarbeitung relevant.

Die Position eines *btRigidBody* wird über die *world transform*-Variable festgelegt. Durch *setWorldTransform* kann einem Objekt in der virtuellen Welt eine Position zugeordnet werden. Diese Position bezieht sich immer auf das Zentrum des Objektes(*center of mass)*.

Math Für die in Abschnitt 4.1.1 vorgestellten Berechnungen benötigt die Implementierung bestimmte Datentypen und Rechenmethoden. Die von BULLET gegebenen Klassen *btSoftBodyInternals* und *Vectormath::Aos* stellen für die Darstellung der Kräfte und Tensoren, Vektoren und Matrizen zur Verfügung. Sowohl die Berechnung von Kreuz-/Skalarprodukt, als auch die Berechnung der Eigenwerte und Inverse einer Matrix, sind darüber hinaus hilfreiche, nutzbare Methoden. Da für die Ermittlung der Kräfte jedoch zusätzliche komplexe Vorgehensweisen benötigt werden, wurde die Sammlung an mathematischen Berechnungen um einige Methoden erweitert.

Constraints Constraints sind definierte Einschränkungen, die beispielsweise von Objekten eingehalten werden müssen. Wenn diese Bedingung nicht eingehalten wird, führt dies zu Änderungen im Objektverhalten. In BULLET sind verschiedene Constraints implementiert. Alle Constraints dieser Physikagieren zwischen zwei Rigid Bodys und werden von der Klasse *btTypedConstraint* abgeleitet. In dieser Ausarbeitung wird der *btGeneric6DofConstraint* genutzt. Dieser Constraint kann durch seine sechs Freiheitsgrade (engl. degrees of freedom) konfiguriert werden. Durch Einschränkung der Translation (Werte 1-3) bzw. Rotation (Werte 4-6) kann so das Verhalten zweier Objekte zueinander bestimmt werden.[5]

Timestep Die Voreinstellung von BULLET ist eine intern festgelegte Bildfrequenz (engl. frame rate) von 60 Hertz. Anwendungen, die auf Bullet aufbauen, haben die Möglichkeit, diesen Zeitschritt zu verändern oder sogar variabel zu gestalten.

Eine Besonderheit von Physik-Simulationen ist es, dass die *frame rate* der eigentlichen Applikation meist von jener der physikalischen Simulation divergiert. Simulation und Animation der Anwendung werden hierbei voneinander getrennt. Wenn beispielsweise die Frequenz der Animation niedriger ist als diejenige der Simulation, so können mehrere Physikberechnungen pro Schritt stattfinden.

Um bei großen Zeitschritten keine störenden ruckartigen Bewegungen zu haben, bietet BULLET die Möglichkeit der Interpolation der Objekte. Hierfür wird mit der Methode *predictMotion()* die Bewegung vorhergesagt.

Draw Für die Anzeige der durchgeführten physikalischen Simulation wurde der interne „Debug-Renderer“ genutzt. Da der ästhetische Aspekt nicht im Vordergrund dieser Ausarbeitung stand, sind die Möglichkeiten der internen Methoden völlig ausreichend. Das Interface, welches für das Zeichnen der virtuellen Welt genutzt wurde, ist *btIDebugDraw*. *btIDebugDraw* basiert auf einer älteren Version von OpenGL. Wenn der „Renderer“ durch *setDebugDrawer* aktiviert wurde, kann die virtuelle Welt ihre Objekte visuell darstellen.

Simulation von Brüchen Im Forum[18] der hier beschriebenen BULLET-Physik- wurden bisher wenige Versuche vorgestellt eine Bruchsimulation durchzuführen. Die Demonstration anhand einer Voronoi-Struktur zeigte die bisher besten Resultate. Die Unterstützung der Tetraeder-Struktur oder konkaven Objekten war hierbei allerdings nicht gegeben. Verfahren aus der Literatur und eine physikalische Plausibilität blieben bisher unberücksichtigt.

5.2 NETGEN

Für die Erstellung der in dieser Ausarbeitung genutzten dreidimensionalen Meshs wurde das Program NETGEN[19] genutzt. Mit dieser Software können sehr viele Formate vorliegender Geometrien geladen werden, um daraus eine Tetraeder-Darstellung zu erschaffen. Durch die Exportfunktion ist es möglich, die reinen Koordinaten- und Elementdaten des Objektes zu erhalten. Da BULLET eine konkrete Struktur der Objekt-Dateien vorgibt, mussten jedoch Anpassungen vorgenommen werden. Eine Datei, mit denen die Physikletztendlich die Objekte erstellt, ist beispielhaft in Listing 1 zu sehen. Die Dateien haben den Typ: C/C++ Inline File (.inl) .

```
static const char* getNodes() { return(
" 68 3 0 0 \n"
"  1 0.000000 0.000000 0.250000 \n"
"  2 0.250000 0.000000 0.000000 \n"
...
"  66 0.165848 0.769164 0.109753\n"
"  67 0.869761 0.746497 0.127286\n");}

static const char* getElements() { return(
" 101 4 0        \n"
"  1 11 55 6 38  \n"
"  2 13 27 66 61 \n"
...
"  99 11 55 38 60\n"
"  100 9 42 5 63 \n");}
```

Listing 1: gekürzte Rohdaten-Datei eines Objekts

[18] http://www.bulletphysics.org/Bullet/phpBB3/
[19] NETGEN 5.1: http://www.hpfem.jku.at/netgen/

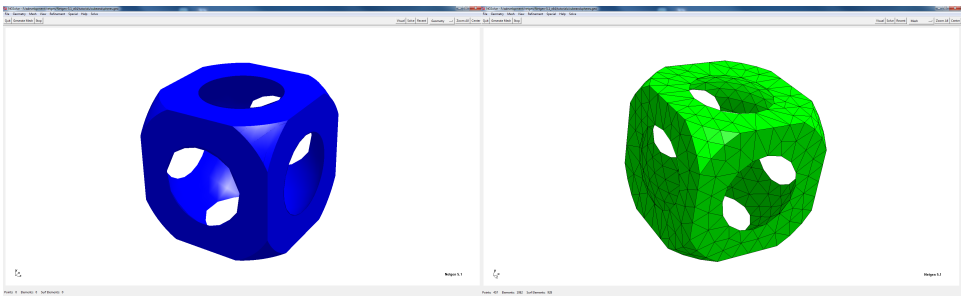


Abbildung 17: Netgen 5.1 - links: Geometrie, rechts: Mesh

6 Implementierung

In diesem Abschnitt wird die Implementierung der vier erläuterten Verfahren vorgestellt. Die in Abschnitt 5.1 erläuterten Strukturen und Klassen der BULLET Physik- werden hierbei als bekannt vorausgesetzt. In Abbildung 18 ist ein gekürzter Ablaufplan der hier vorgestellten Methoden zu sehen.

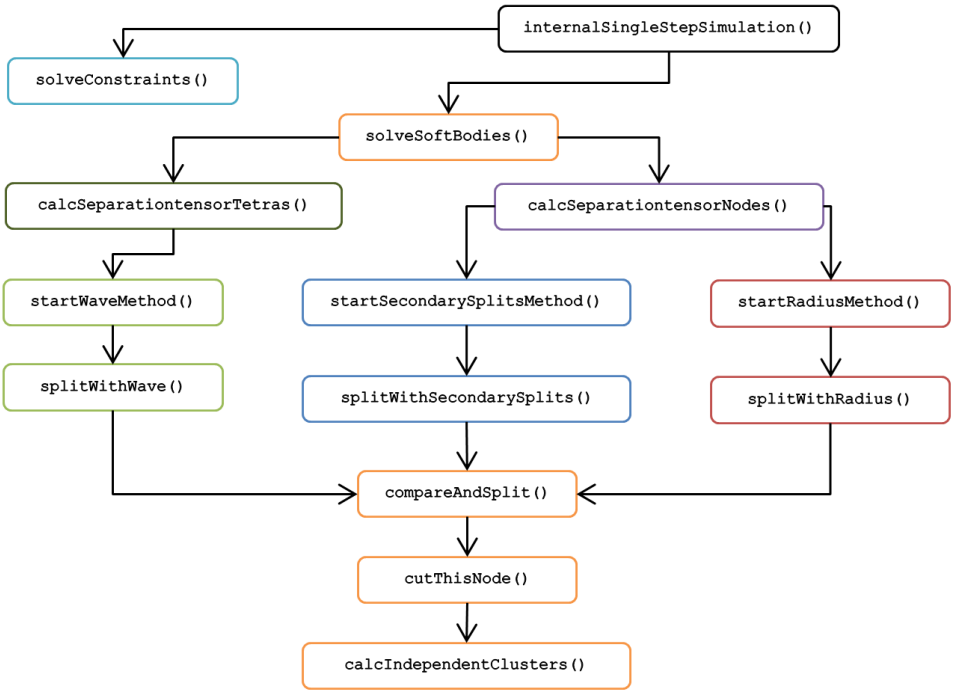


Abbildung 18: Gekürzte Übersicht der relevanten Methoden und Abläufe der Implementation.

6.1 Verfahren nach Smith et al.

6.1.1 Modell und Parameter

Zur Implementation des von Smith et al. beschriebenen Verfahren (siehe Abschnitt 4.2) wurden die Objekte in dieser Ausarbeitung zusätzlich zu der Tetraeder-Struktur durch Voronoi-Elemente angenähert. Wie bereits in Abschnitt 3.4 auf Seite 12 beschrieben, sind sich die in NETGEN generierte Darstellung der Objekte als Mesh und die Voronoi-Struktur sehr ähnlich. Eine Voronoi-Darstellung kann daher aus den bestehenden Knoten der Tetraeder-Meshs relativ einfach berechnet werden. Der Grund für die Nutzung der Voronoi-Diagramme als weitere Objekt-Struktur liegt in ihrer Performanz. Im Ergebnisse-Abschnitt (7 auf Seite 59) wird dieser Aspekt näher erläutert.

Listing 2 zeigt die Initiierung eines Objekts. Um ein Voronoi-Objekt zu erstellen, wird für jeden Massenpunkt die Methode c*alculateVoronoiPoints()* aufgerufen. In dieser Methode wird die Distanz zwischen dem Punkt i und allen weiteren Massenpunkten berechnet und überprüft, ob dieser Abstand geringer ist als der bisher gespeicherte Abstand. Alle Punkte, die nach der Iteration den geringsten Abstand zu einem Massenpunkt haben, ergeben das Voronoi-Element. Für jedes entstandene Element wird anschließend eine konvexe Hülle berechnet. Aus jeder konvexen Hülle wird ein Kollisionsobjekt und ein Starrkörper *(btRigidbody)* abgeleitet und in der virtuellen Welt platziert. Falls keine Voronoi-Repräsentation des Objekts gewünscht ist, wird für jedes gegebene Tetraeder-Element eine konvexe Hülle berechnet. *m_ toughness* beschreibt hier die Materialhaltbarkeit.

```
void Init_ConstraintObject()
{
  btAlignedObjectArray<Vector3> points = Object::getNodes();
  btAlignedObjectArray<Vector3> vertices = Object::getTetraeder();
  btConvexHullComputer* cHC = new btConvexHullComputer();


  for (int i = 0; i < points.size(); i++)
  {
    if(voronoimode)
        vertices = calculateVoronoiPoints(i);
    cHC->compute(vertices);
    btCollisionShape* s = new btConvexHullShape(cHC->vertices);
    btRigidBody* shardBody = new btRigidBody(s.info);
  }
  m_toughness = getFractureParameter().getValue();
}
```

Listing 2: Berechnung der Voronoi-Darstellung

6.1.2 Berechnung der abstandserhaltenden Bedingung

Für alle zusammenhängenden *btRigidbodys* wird im Anschluss initial ihre abstandserhaltende Bedingung (engl. constraint) berechnet (siehe Listing 3). Hierzu werden alle Verbindungen über gemeinsame Flächen und Knoten (engl. manifold *:* Anschlussstück) berechnet und deren Kontakte untersucht. Jedes „Verbindungsstück“ hat genau zwei Objekte, die sich berühren (*body0* und *body1*). Aus deren Kontaktpunkt in inversen Weltkoordinaten wird dann die eigentliche Bedingung berechnet. Die Stärke des Bundes zwischen den beiden Elementen wird aus einem Material-Faktor und den beiden Massen der *btRigidBodys* berechnet. Die Masse wird aus Volumen und Dichte der Körper abgeleitet. Die berechnete lineare Bedingung wird im weiteren Verlauf der Simulation als Schwellwert für einen möglichen Bruch genutzt. Damit den beiden verbundenen Rigid Bodys keine Veränderung zueinander möglich ist, werden mit *dof6->setLimit* alle sechs möglichen Freiheitsgerade „gesperrt“. Die in der Orginal-Ausarbeitung von Smith et al. beschriebene *behaviour-function* wird in der hier vorgestellten Implementierung nur angenähert. Sie konnte durch die von BULLET vorgegebene Struktur nicht eins zu eins übernommen werden.

```
void Init_Constraints()
{
  calculateManifolds();

  for (int i = 0; i<getNumManifolds(); i++)
  {
    btManifold* manifold = getManifoldByIndexInternal(i);
    if (!manifold->getNumContacts())
        continue;
    btRigidBody* body0 = manifold->getBody0();
    btRigidBody* body1 = manifold->getBody1();

    btTransform tr = manifold->getContactPoint();
    tr->getWorldTransform().inverse();

    btGeneric6DofConstraint* dof6 =
              new btGeneric6DofConstraint(*body0, *body1, tr);

    float totalMass = body0->getMass() + body1->getMass();
    dof6->setBreakingImpulseThreshold(m_toughness*totalMass);

    for (int j = 0; j<6; j++)
      dof6->setLimit(j, 0, 0);

    getDynamicsWorld()->addConstraint(dof6);
  }
}
```

Listing 3: Berechnung der abstandserhaltenden Bedingung

6.1.3 Simulation des Bruchs

Die von *Bullet* bereitgestellte Methode *internalSingleStepSimulation(btScalar timeStep)* führt in jedem Zeitschritt die für die Simulation notwendigen Berechnungen durch. In Listing 4 ist zu sehen, dass zunächst die möglichen Kollisionspaare berechnet werden und danach deren Kollision durchgeführt wird. In *processCollision()* wird u.a. der weitere Bewegungsverlauf der Objekte berechnet. Für jede Kollision wird zusätzlich eine Markierung (engl. flag) für später aufgerufene Methoden gesetzt. Nachdem die Kollisionserkennung abgeschlossen ist, werden mögliche Brüche initiiert. Die im Verfahren von Smith, Witkin und Baraff aufgestellten abstanderhaltenden Bedingungen, werden in der Methode *solveConstraints()* (Listing 5) aufgelöst. Die drei weiteren in dieser Ausarbeitung implementierten Verfahren werden in der Methode *solveSoftBodies()* aufgerufen.

```
void internalSingleStepSimulation(btScalar timeStep)
{
  btCollisionWorld::performDiscreteCollisionDetection();
  btCollisionWorld::computeOverlappingPairs();
  dispatchAllCollisionPairs();

  for ( int i=0;i<m_collisionPairs.size();i++)
  {
    processCollision();
    setCollisionFlags();
  }
  solveConstraints();
  solveSoftBodies();
}
```

Listing 4: Berechnung der Kollisionen und Initiierung der potentiellen Brüche

In der Methode *solveConstraints()* wird über alle bestehenden Constraints iteriert und der aktuell wirkende Impuls mit dem in Listing 3 berechneten Schwellwert verglichen. Sollte der Schwellwert überschritten werden, so wird die Verbindung der beiden Elemente entfernt und die umliegenden Elemente durch einen Koeffizienten geschwächt. Sollte der Schwellwert nicht übertroffen werden so wird das entsprechende Constraint selbst geschwächt. In der Implementierung des Verfahrens nach Smith et al. wurde hier der von BULLET gelieferte Impuls als Vergleichswert genutzt, da ein „echter“ Belastungswert nicht berechnet werden konnte. Der Grund dafür lag in der Struktur der Rigid Bodys und wird in den Resultaten dieser Ausarbeitung näher betrachtet.

```
void solveConstraints()
{
  for (j=0;j<m_ConstraintPool.size();j++)
  {
    btSolverConstraint& sConstr = m_ConstraintPool[j];
    btTypedConstraint* constr = sConstr.m_originalContact;
```

```
    constr->SetAppliedImpulse(sConstr.m_appliedImpulse);

    //compare impulse with threshold
    if (sConstr.m_appliedImpulse
                    >=constr->getBreakingImpulseThreshold())
    {
      //delete this constraint
      constr->setEnabled(false);

      btRigidBody* BodyA = &constr->getRigidBodyA();
      btRigidBody* BodyB = &constr->getRigidBodyB();

      for (int a=0;a<numPoolConstraints;a++)
      {
        btSolverConstraint& csConstr = m_ConstraintPool[a];
        btTypedConstraint* cconstr = csConstr.m_originalContact;

        btRigidBody* cBodyA = &cconstr->getRigidBodyA();
        btRigidBody* cBodyB = &cconstr->getRigidBodyB();

        if (BodyA == compareBodyA || BodyB == compareBodyA
            || BodyB == compareBodyB || BodyA == compareBodyB )
        {
          weakness_coefficient = 0.975;
         //degrade neighbours
          cconstr->setBreakingImpulseThreshold(cconstr->
            getBreakingImpulseThreshold()*weakness_coefficient);
        }
      }
    }
    else
    {
      constr->setBreakingImpulseThreshold(constr->
        getBreakingImpulseThreshold()-sConstr.m_appliedImpulse);
    }
  }

}
```

Listing 5: Berechnung der wirkenden Kraft und Vergleich des Impuls mit dem Schwellwert

6.2 Verfahren auf Grundlage der Kontinuumsmechanik

6.2.1 Modelle und Parameter

Jedes in dieser Ausarbeitung implementierte Objekt hat ein Mesh als Grundlage. Die durch *Netgen* ausgegebenen Dateien werden durch die von *Bullet* bereitgestellte Funktion *CreateFromTetGenData()* eingelesen und aus den entsprechenden Knoten und Elementen das *btSoftBody* erstellt. Dieses *btSoftBody* wird nun mit beliebiger Ausrichtung in einer vorgegebenen Höhe

in der virtuellen Welt platziert. Des Weiteren werden notwendige Parameter zur Kollisionserkennung angegeben.

```
static void Init_Cube()
{
  btSoftBody* psb=btSoftBodyHelpers::CreateFromTetGenData(
                  TetraCube::getElements(),TetraCube::getNodes());

  getSoftDynamicsWorld()->addSoftBody(psb);
  psb->rotate(randomize(btQuaternion rq));
  psb->translate(Vector3(0,5,0));
  psb->setVolumeMass(1);

  psb->generateClusters(8);
  psb->m_cfg.collisions = btSoftBody::fCollision::CL_RS;
}
```

Listing 6: Initialisierung eines „Soft Body“

Für die Bruchsimulation sind zusätzlich zur Struktur des Objekts die Material-Parameter entscheidend. Diese werden zusammen mit den jeweiligen Koeffizienten für aktuelles Objekt (*objectID*), aktives Verfahren (*fractureMode*) und maximale Anzahl an Bruchinitiationen (*fractureInits*) in einer Matrix gespeichert. Die Initialisierung dieser Matrix ist in 7 zu sehen. Die Bedeutungen zu den hier aufgeführten griechischen Symbolen findet sich in Abschnitt 3.3 .

```
Matrix3 paramMatrix;

paramMatrix.setValue( materialID,   phi,        psi,
                      tau,          lambda,     mu,
                      fractureMode, objectID,   fractureInits );
```

Listing 7: Initialisierung der Materialparameter

6.2.2 Berechnung der wirkenden Kräfte

Wie bereits in der Beschreibung der Implementierung des Verfahrens von Smith et al. (Abschnitt 6.1) angedeutet, wird nach Berechnung der Kollisionen ein möglicher Bruch initiiert. In den Verfahren, die auf Mesh-Objekten basieren, geschieht dies durch die Methode *solveSoftBodies()*.

Die Methode *solveSoftBodies()* löst bei einer Kollision die Berechnung der nun auftretenden Kräfte aus und startet die Vorgänge des aktuell ausgewählten Verfahrens. Die Berechnungen der wirkenden Kräfte sind für alle Verfahren hierbei gleich. Eine Unterscheidung in *calcSeparationtensorNodes()* und *calcSeparationtensorTetras()* muss dennoch gemacht werden. Die Methode von Glondu et al. zieht den Vergleich zwischen wirkenden Kräfte und Schwellwert auf Element- anstatt auf Massenpunktebene.

Nach der Berechnung der Kräfte werden diese Werte an die gewählte Bruch-Methode übergeben. Es folgt die potentielle Durchführung des Bruchs

durch *startRadiusMethod()*, *startWaveMethod()* oder *startSecondarySplits-Method()*. Abschließend werden in dieser Methode die neuen Kollisionsobjekte berechnet (Erläuterungen im Abschnitt 6.2.4) und die Soft Bodys aktualisiert.

```
void solveSoftBodies()
{
  for ( int i=0;i<m_softBodies.size();i++)
  {
    btAlignedObjectArray<Matrix4>    compareValues;
    btSoftBody* psb=(btSoftBody*)m_softBodies[i];
    if (psb->getCollisionFlags())
    {
      //Fracturemode 0: radius method (Mueller et al.)
      if(psb->getFractureMode()== 0)
      {
        compareValues = calcSeparationtensorNodes();
        startRadiusMethod(compareValues);
      }

      //Fracturemode 1: wave method (Glondu et al.)
      else if(psb->getFractureMode()== 1)
      {
        compareValues = calcSeparationtensorTetras();
        startWaveMethod(compareValues);
      }

      //Fracturemode 2: Secondary-Splits Method (own)
      else if(psb->getFractureMode()== 2)
      {
        compareValues = calcSeparationtensorNodes();
        startSecondarySplitsMethod(compareValues);
      }
    }
  }
  calcIndependentClusters();
  updateSoftBodies( );
}
```

Listing 8: Initiierung der Kräfteberechnungen und des ausgewählten Verfahren

Für die *Radius-* und *secondary-Splits-Methode* wird für den Vergleich zum Materialschwellwert der Separationstensor berechnet. Dieser Tensor wird aus den auf die einzelnen Tetraeder wirkenden Kräfte ermittelt. Diese Berechnung ist in der Methode *calculateForces()* in Listing 11 auf Seite 44 zu sehen. Negative und positive Kräfte werden addiert und im Array *varsigma* gespeichert. Nachdem die Kräfte für jedes Element berechnet wurden, müssen diese zur weiteren Vorgehensweise auf die Massenpunkte übertragen werden. Zu diesem Zweck werden alle Kräfte der an einem Massenpunkt anliegenden Elemente summiert und in in der Matrix *NodeForce* gespeichert. Nachdem die ermittelte Kraft durch die Anzahl der anliegenden Elemente dividiert wurde, werden aus der entstandenen Matrix die Eigenwerte und Vektoren berech-

net. Hierzu wird das von *Bullet* bereitgestellte Struct *btEigen* genutzt. Die Eigenwerte und Vektoren werden anschließend im Array *cnodes* gespeichert. Als letzter Schritt wird das Array *cnodes* nach dem höchsten Eigenwert der Massenpunkte sortiert. Der Punkt der danach an erster Stelle im Array steht, hat also die höchste vorhandene Belastung.

```
btAlignedObjectArray<Matrix4> calcSeparationtensorNodes()
{
  Matrix4 positiveForces;
  Matrix4 negativeForces;
  btAlignedObjectArray<Matrix4> varsigma;
  btAlignedObjectArray<Matrix4> cnodes;

  for (int TetIndex = 0; TetIndex < m_tetras.size();TetIndex++)
  {
    calculateForces (TetIndex,&negativeForces,&positiveForces);
    varsigma[TetIndex] = negativeForces + positiveForces;
  }
  for (int NodeID = 0; NodeID<m_nodes.size();NodeID++)
  {
    Node* mnode=&m_nodes[NodeID];
    Matrix4 NodeForce;
    int appendedTetraCount = 0;

    for (int tID = 0; tID < m_tetras.size();tID++)
    {
      Tetra& mtetra = m_tetras[tID];
      for (int internID =0; internID <4;internID++ )
      {
        //if same node
        if (mtetra.m_n[internID] == mnode)
        {
          NodeForce += varsigma[tID];
          appendedTetraCount++;
        }
      }
    }
    NodeForce = NodeForce/appendedTetraCount;

    //calculating eigenvalues
    btEigen EigenStruct;
    cnodes[NodeID] = EigenStruct.eigen_decomposition(NodeForce);
    cnodes.quicksort();
  }
  return cnodes;
}
```

Listing 9: Berechnung des Separationstensors mit Massenpunkten als Basis

Der einzige Unterschied zwischen den verwendeten Methoden *calcSeparationtensorNodes()* und *calcSeparationtensorTetras()* liegt darin, dass die Eigenwerte in der erstgenannten Methode für Knoten und nicht für Elemente berechnet werden. In beiden Methoden wird das Array *cnodes/ctetras* für

die nachfolgende Bruchsimulation an die Methode *solveSoftBodies()* zurückgegben.

```
btAlignedObjectArray<Matrix4> calcSeparationtensorTetras()
{
  Matrix4 positiveForces;
  Matrix4 negativeForces;
  btAlignedObjectArray<Matrix4>  varsigma;
  btAlignedObjectArray<Matrix4> ctetras;

  for (int TetIndex = 0; TetIndex < m_tetras.size();TetIndex++)
  {
    calculateForces (TetIndex,&negativeForces,&positiveForces);
    varsigma[TetIndex] = negativeForces + positiveForces;
  }

  for (int tID = 0; tID < m_tetras.size();tID++)
  {
    Tetra& mtetra = m_tetras[tID];

    //calculating eigenvalues
    btEigen EigenStruct;
    ctetras[tID] = EigenStruct.eigen_decomposition(varsigma[tID]);
    ctetras.quicksort();
  }
  return ctetras;
}
```

Listing 10: Berechnung des Separationstensors mit Elementen als Basis

Die in beiden Methoden aufgerufene Methode *calculateForces()* bildet das physikalische Herzstück der drei in diesem Abschnitt erläuterten Verfahren. In dieser Methode werden sämtliche Tensoren und wirkenden Kräfte ermittelt. Da diese Methode sehr umfassend ist, wird sie im Folgenden in mehrere Programmauflistungen aufgeteilt.

Am Anfang dieser Methode werden zunächst die verwendeten Parameter und Informationen der Eckpunkte geladen. P enthält die Position der vier Massenpunkte des Tetraeders in Weltkoordinaten, M deren Position in lokalen Koordinaten und V die Geschwindigkeiten dieser Punkte. *beta* gibt die inverse Matrix von M an. Für die Berechnung des Belastungstensors wird diese Matrix zunächst mit den Weltkoordinaten der Punkte multipliziert. Das Ergebnis wird intern in drei Vektoren gespeichert (*DeltaXDeltaUxyz*). Die für die Berechnung des Belastungsratentensors notwendige Matrix *DeltaDotXDeltaU* berechnet sich analog. Die hier umgesetzten Formeln sind im Abschnitt 4.1.1 genauer erläutert.

```
void calculateForces(int TetIndex,
                     Matrix4* negativeForces,
                     Matrix4* positiveForces )
{
  Tetra& my_tetra = m_tetras[TetIndex];
```

```
float phi = getParamMatrix().getRow(0).getY();
float psi = getParamMatrix().getRow(0).getZ();
float mu = getParamMatrix().getRow(1).getZ();
float lambda = getParamMatrix().getRow(1).getY();

//3x4 matrices
//containing the information of four vertices
Transform3 P = my_tetra.getNodePositions();
Transform3 V = my_tetra.getNodeVelocities();
Transform3 M = my_tetra.getNodeMaterialPositions();

Matrix4 beta = Matrix4(M,1);
beta = beta.inverse();

Transform3 DeltaXDeltaU = multiplyMatrices(&P,&Beta);
Vector3 DeltaXDeltaUxyz [3];
DeltaXDeltaUxyz [0] = multiply(&DeltaXDeltaU,
                               &Vector4(1.0,0.0,0.0,0.0));
DeltaXDeltaUxyz [1] = multiply(&DeltaXDeltaU,
                               &Vector4(0.0,1.0,0.0,0.0));
DeltaXDeltaUxyz [2] = multiply(&DeltaXDeltaU,
                               &Vector4(0.0,0.0,1.0,0.0));

Transform3 DeltaDotXDeltaU = multiplyMatrices(&V,&Beta);
Vector3 DeltaDotXDeltaUxyz [3];
DeltaDotXDeltaUxyz [0] = multiply(&DeltaDotXDeltaU,
                               &Vector4(1.0,0.0,0.0,0.0));
DeltaDotXDeltaUxyz [1] = multiply(&DeltaDotXDeltaU,
                               &Vector4(0.0,1.0,0.0,0.0));
DeltaDotXDeltaUxyz [2] = multiply(&DeltaDotXDeltaU,
                               &Vector4(0.0,0.0,1.0,0.0));
```

Listing 11: Parameter und Knoteninformationen werden bestimmt

Der Belastungstensor *epsilon*, der die lokale Deformation des Materials misst, wird aus den Skalaren (Skalarfunktion: *dot()*) von *DeltaXDeltaUxyz* gebildet. Des Weiteren wird *nu*, der Belastungsratentensor, berechnet. Er beschreibt die Änderung der Belastung und ist die Ableitung von *epsilon* über die Zeit.

```
//strain tensor
Matrix3 epsilon;
for (int i= 0; i<3; i++)
{
  for (int j= 0; j<3; j++)
  {
    if (i==j)
      epsilon.setElem(i,j,dot(DeltaXDeltaUxyz[i],
                              DeltaXDeltaUxyz[j]) - 1);
    else
      epsilon.setElem(i,j,dot(DeltaXDeltaUxyz[i],
                              DeltaXDeltaUxyz[j]) - 0);
  }
}
```

```
//strain-rate tensor
Matrix3 nu;
for (int i= 0; i<3; i++)
{
  for (int j= 0; j<3; j++)
  {
    nu.setElem(i,j,dot( DeltaXDeltaUxyz[i], DeltaDotXDeltaUxyz[j])
               + dot( DeltaDotXDeltaUxyz[i], DeltaXDeltaUxyz[j]));
  }
}
```

Listing 12: Bestimmung von Belastungs- und Belastungsratentensor

Aus diesen beiden Tensoren wird im nächsten Schritt der Stresstensor *sigma* ermittelt. *sigma* ist die Summe der beiden Tensoren *sigmaepsilon* und *sigmanu*. Zunächst wird der elastische Stress *sigmaepsilon* berechnet. *mu* und *lambda* beschreiben hier die erste und zweite Lamé-Konstante. Die in Listing 13 umgesetzte Formel findet sich in Abschnitt 4.1.1 auf Seite 16 wieder. Der Trägheitsstress *sigmanu* berechnet sich in diesem Fall sinngemäß aufgrund der Belastungsrate *nu*. *phi* und *psi* definieren hier, wie schnell eine elastische Verformung zurückgeht, nachdem die Krafteinwirkung auf das Objekt verschwunden ist.

```
//elastic stress
Matrix3 sigmaepsilon;
float sevalue;
for (int i= 0; i<3; i++)
{
  for (int j= 0; j<3; j++)
  {
    sevalue= 2*mu*epsilon.getElem(i,j);

    if (i==j)
      for (int k=0; k<3; k++)
        sevalue += lambda*epsilon.getElem(k,k);

    sigmaepsilon.setElem(i,j,sevalue);
  }
}

//viscous stress
Matrix3 sigmanu;
float snvalue;
for (int i= 0; i<3; i++)
{
  for (int j= 0; j<3; j++)
  {
    snvalue= 2*psi*nu.getElem(i,j);

    if (i==j)
      for (int k=0; k<3; k++)
        snvalue += phi*nu.getElem(k,k);
```

```
      sigmanu.setElem(i,j,snvalue);
    }
  }

  //stresstensor
  Matrix3 sigma = sigmaepsilon + sigmanu;
```

Listing 13: Bestimmung des Stresstensors

Aus *sigmanu* und *sigmaepsilon* kann nun die gesamte innere Kraft ermittelt werden. Hierfür werden die elastische Kraft *forceElastic* und die Dämpfungskraft *forceDamping* durch die Hilfsvariablen *fevalue/fevector* bzw. *fdvalue/fdvector* berechnet. Die Summe dieser beiden Kräfte ergibt die gesamte innere Kraft, welche ein Element auf einen Knoten/Massenpunkt ausübt. Diese Kraft wird in Listing 14 durch die Variable *nodeForce* angegeben.

```
Vector3 forceElastic [4];
float fevalue;
Vector3 fevector;

Vector3 forceDamping [4];
float fdvalue;
Vector3 fdvector;
for (int i= 0; i<4; i++)
{
  for (int j= 0; j<4; j++)
  {
    for (int k= 0; k<3; k++)
    {
      for (int l= 0; l<3; l++)
      {
        fevalue += beta.getElem(j,l)*beta.getElem(i,k)
                                    *sigmaepsilon.getElem(k,l);
        fdvalue += beta.getElem(j,l)*beta.getElem(i,k)
                                    *sigmanu.getElem(k,l);
      }
    }
    fevector += P.getColumn[j]*fevalue;
    fdvector += P.getColumn[j]*fdvalue;
  }
  forceElastic[i] = -vol2/2*(fevector);
  forceDamping[i] = -vol2/2*(fdvector);
}

//total inner force
Vector3 nodeForce [4];
for (int i =0; i<4;i++)
{
  nodeForce[i] = forceElastic [i] + forceDamping [i];
}
```

Listing 14: Bestimmung des inneren Kraft

In Listing 15 ist der letzte Schritt der für die Bruchsimulation notwendigen Kraftberechnung aufgeführt. Zunächst werden die Eigenwerte der ermittelten Krafteinwirkung (*nodeForce*) bestimmt. Danach folgt eine Aufteilung der ermittelten Eigenwerte zu *sigmaPositive* bzw. *sigmaNegative*. Diese beiden Matrizen bezeichnen den Zugkraft- bzw. Druckkrafttensor aus den Formeln 12 und 13. *eVectorMatrix* ist die Matrix-Darstellung des zum Eigenwert korrespondierenden Eigenvektors.

Durch die Hilfsvariablen *nfpvalue/nfpvector* bzw. *nfnvalue/nfnvector* werden dann die Formeln 14 und 15 umgesetzt und abschließend deren Resultate den an die Methode *calculateForces()* übergebenen Matrizen *negativeForces* und *positiveForces* zugewiesen.

```
//calculating eigenvalues
btEigen EigenStruct;
Matrix4 eigenResult = EigenStruct.eigen_decomposition(sigma);

float evalue;
Matrix3 eVectorMatrix;
Matrix3 sigmaPositive;
Matrix3 sigmaNegative;
for (int i =0; i<3;i++)
{
  evalue = eigenResult.getEigenValue(i);
  eVectorMatrix = prepareMatrix(evalue.getCorrespondingVector());
  if (evalue>0)
    sigmaPositive += evalue*eVectorMatrix;
  else
    sigmaNegative += evalue*eVectorMatrix;
}
//tetrahedra volume
float vol = (dot( M.getColumn[3]-M.getColumn[0],
              (cross((M.getColumn[1]-M.getColumn[0]),
              (M.getColumn[2]-M.getColumn[0])))))/6;

float nfpvalue;
Vector3 nfpvector;

float nfnvalue;
Vector3 nfnvector;

for (int i = 0;i<4;i++)
{
  for (int j = 0;j<4;j++)
  {
    for (int k= 0; k<3; k++)
    {
      for (int l= 0; l<3; l++)
      {
        nfpvalue += beta.getElem(j,l)*beta.getElem(i,k)
                              *sigmaPositive.getElem(k,l);
        nfnvalue += beta.getElem(j,l)*beta.getElem(i,k)
                              *sigmaNegative.getElem(k,l);
```

```
        }
      }
      nfpvector += P.getColumn(j)*nfpvalue;
      nfnvector += P.getColumn(j)*nfnvalue;
    }
    negativeForces->setCol(i,Vector4(-vol/2*(nfnvector),1));
    positiveForces->setCol(i,Vector4(-vol/2*(nfpvector),1));
  }


//end of calculateForces()
}
```

Listing 15: Bestimmung der positiven und negativen Kraft

6.2.3 Simulation des Bruchs

Radius-Methode Die in Listing 8 auf Seite 42 bereits erwähnte Methode *startRadiusMethod()* bekommt die in Listing 9 auf Seite 43 berechneten Eigenwerte und Eigenvektoren des Separationstensors als Array übergeben. Die Anzahl der möglichen Schnitte durch das Objekt wird durch den Parameter *maxsplit* durch den Benutzer vorgegeben. Der Standard-Wert in der implementierten Demo ist hierbei 3. Da *compareValues* nach der Größe der Eigenwerte sortiert wurde, findet nun der Reihe nach ein Vergleich zwischen dem höchsten Eigenwert und der Materialhaltbarkeit *tau* statt. Sollte der Schwellwert überschritten werden, wird der Bruch durch die Methode *splitWithRadius()* gestartet.

```
void startRadiusMethod(btAlignedObjectArray<Matrix4> compareValues)
{
  //toughness
  float tau = getParamMatrix().getRow(1).getX();

  int maxsplit = getParamMatrix().getColumn(2).getZ();
  for(int split=0;split<maxsplit;++split)
  {
    float m_eigenvalue = compareValues[split].
                                        getHighestEigenValue();
    if (m_eigenvalue > tau)
    {
      Vector3 m_eigenvector = compareValues[split].
                        getCorrespondingVector(m_eigenvalue);

      splitWithRadius(compareValues[split].getNodeID(),
                                      m_eigenvector,eigenvalue);
    }
  }
}
```

Listing 16: Initialisierung des Radius-Verfahren

Diese Methode erhält die ID des Massenpunktes, der aufgeteilt werden soll, den höchsten Eigenwert und den dazu korrespondierenden Eigenvektor als Eingabeparameter. Der Eigenwert dient in dieser Methode als Radius und spiegelt damit die Auswirkungen der Kraft auf die in der Nähe liegenden Elemente wieder. Jedes Element, dass sich innerhalb dieses Radius befindet, wird durch die boolsche Variable *tetraIsNear* markiert. Sollte sich das Tetraeder in der Nähe des Knotens befinden, so wird es auf seine Orientierung zur in Abschnitt *4.1.3* beschriebenen Schnittebene überprüft. Hierfür wird der Schwerpunkt *balancePoint* des Tetraeders ermittelt und dessen Skalar zu dem übergebenen Eigenvektor v berechnet. Im Array *tetraOrientation* wird die entsprechende Orientierung durch den Eintrag *„-1“* (hinter der Ebene) bzw. *„1“* (vor der Ebene) gespeichert.

```
void splitWithRadius( int NodeID, Vector3 &v, float eigenvalue )
{
  btScalar mRadius = eigenvalue;
  Node* m_node=&m_nodes[NodeID];
  btAlignedObjectArray<int> tetraOrientation;

  for(int i=0;i<m_tetras.size();++i)
  {
    bool tetraIsNear = false;
    Tetra& m_tetra=m_tetras[i];

    for (int j =0; j<4; j++)
    {
      Node* c_node = m_tetra.m_n[j];

      if((c_node->m_x - m_node>m_x).length() < mRadius)
        tetraIsNear = true;
    }
    if (tetraIsNear)
    {
      for (int j =0; j<4; j++)
      {
        Node* c_node = m_tetra.m_n[j];
        Vector3 balancePoint += 0.25*c_node;
      }
      float dotProductbalancePoint = dot(balancePoint,v);

      if (dotProductbalancePoint < 0)
        tetraOrientation[i] = -1;
      else
        tetraOrientation[i] = 1;
    }
  }
  compareAndSplit(tetraOrientation);
}
```

Listing 17: Untersuchung der Massenpunkte auf einen notwendigen Schnitt

Falls nun ein Massenpunkt an Tetraedern liegt, welche einen unterschiedlichen Eintrag in diesem Array haben, wird er geschnitten. In der Methode *compareAndSplit()* werden für diesen Zweck boolsche Variablen als „flag“ genutzt. Sollten beide „flags“ gesetzt sein, wird ein Bruch initiiert. Dieser „Split“ wird in der Methode *cutThisNode()* durchgeführt.

```
void compareAndSplit(btAlignedObjectArray<int> tetraOrientation)
{
  for (int nID = 0; nID<ncount; nID++)
  {
    bool positiveSide = false;
    bool negativeSide = false;

    for (int tID = 0; tID< tetraOrientation.size(); tID++)
    {
      Tetra& mTet = m_tetras[tID];

      //if tetrahedra contains the actual node
      if (mTet.m_n[0] == nID || mTet.m_n[1] == nID ||
                mTet.m_n[2] == nID || mTet.m_n[3] == nID)
      {
        if (tetraOrientation[tID] == 1)
          positiveSide = true;
        else if (tetraOrientation[tetraederID] == -1)
          negativeSide = true;
      }
      if (positiveSide && negativeSide)
        cutThisNode(nID, tetraOrientation);break;
    }
  }
}
```

Listing 18: Vergleich der Element-Orientierung

Jeder Massenpunkt, der als Parameter in diese Methode übergeben wird, wird aufgeteilt. Für diese Aufteilung wird zunächst ein weiterer Knoten *nNode* erstellt. Dieser Knoten erhält die gleichen Positions- und Geschwindigkeitsinformationen wie der zu „splittende“ Massenpunkt *pNode* und wird dem *m_nodes* Array angehängt. *nPlus* und *nMinus* enthalten die ID dieser beiden Punkte. Die Aufteilung des Massenpunkts ist notwendig, da im weiteren Verlauf der Methode die Elemente, je nach Ausrichtung, nur einem der Knoten angehängt werden dürfen. Die vor der Schnittebene liegenden Elemente werden nun *pNode* zugewiesen, und alle anderen dem Knoten *nNode*. Bevor diese Aufteilung statt finden kann, werden jedoch alle bestehenden Verbindungen zum Knoten *pNode* entfernt. Danach erfolgt die Überprüfung der Knoten und die entsprechende Zuweisung. Hierfür werden die in *tetraOrientation* gespeicherten Ausrichtungen der Elemente genutzt. Nachdem das Element dem richtigen Knoten zugewiesen wurde, wird der Knoten mit den bereits bestehenden Massenpunkten des Elements verbunden.

```
void cutThisNode(int nID, btAlignedObjectArray<int>
                                                tetraOrientation)
{
  int nPlus = nID;
  Node* pNode = &m_nodes[nPlus];
  pNode->m_cracknode = true;

  appendNode(pNode->m_x, pNode->m_im);
  m_nodes[m_nodes.size()-1].m_v=v0;

  int nMinus = m_nodes.size()-1;
  Node* nNode = &m_nodes[nMinus];

  //delete existing connections of nPlus
  for(int lID=0;lID<m_links.size();++lID)
  {
    int id[]={m_links[lID].m_n[0]-&m_nodes[0]),
              m_links[lID].m_n[1]-&m_nodes[0])};

    if (id[0]== nPlus || id[1]== nPlus )
    {
      btSwap(m_links[lID],m_links[m_links.size()-1]);
      m_links.pop_back();
      --lID;
    }
  }
  for (int j = 0; j< tetraOrientation.size(); j++)
  {
    Tetra& mTet = m_tetras[j];
    for (int k = 0; k < 4; k++)
    {
      if (mTet.m_n[k] == nID)
      {
        if (tetraOrientation[j] == 1)
          mTet.m_n[k]=pNode;
        else
          mTet.m_n[k]=nNode;

        //connect the new node with the three further nodes
        refineLinks();
      }
    }
  }


}
```

Listing 19: Aufteilung der Elemente zu den entsprechenden Massenpunkten

Wave-Methode Die gezeigte Methode *startWaveMethod()* bekommt die berechneten Eigenwerte und Eigenvektoren im Array *compareValues* übergeben. Im Gegensatz zu der in Listing 16 auf Seite 49 vorgestellten Methode

startRadiusMethod() befinden sich im Array *compareValues* Elemente/Tetraeder und keine Knoten. Sollte der Schwellwert *tau* durch den höchsten Eigenwert des Elements *(m_ eigenvalue)* überschritten werden, wird zunächst ein schwaches Element in der Nähe dieses Elements gesucht. Im Durchschnitt wurde für dieses Verfahren bei der Erstellung des Objekts in etwa jedes zehnte Element als „weak element“ deklariert. Wenn ein schwaches Element in der Nähe des ausgehenden Objekts gefunden wurde, wird dessen ID der Methode *splitWithWave()* übergeben. Zusätzliche Übergabe-Parameter sind ein Array zur Speicherung und Eigenwert bzw. Eigenvektor zur Berechnung der Schnittebene. Nachdem in *splitWithWave()* (siehe Listing 21 auf der nächsten Seite) diejenigen Tetraeder ermittelt wurden, welche potentiell geschnitten werden, wird die in Listing 18 bereits gezeigte Methode *compareAndSplit()* zur eigentlichen Aufteilung der Elemente genutzt.

```
void startWaveMethod(btAlignedObjectArray<Matrix4> compareValues)
{
  //toughness
  float tau = getParamMatrix().getRow(1).getX();

  int maxsplit = getParamMatrix().getColumn(2).getZ();
  for(int split=0;split<maxsplit;++split)
  {
    float m_eigenvalue = compareValues[split].
                                        getHighestEigenValue();
    if (m_eigenvalue > tau)
    {
      Vector3 m_eigenvector = compareValues[split].
                          getCorrespondingVector(m_eigenvalue);
      int tID = compareValues[split].getTetraID();
      Tetra& mTet = m_tetras[tID];

      //Searching for weak elements
      bool foundWeakElement = false;
      for (int cTedID = 0;cTedID< m_tetras.size(); cTedID++)
    {
        Tetra& cTet = m_tetras[cTedID];
        if (foundWeakElement)
          break;

        for (int i = 0; i<4;i++)
        {
          for (int j = 0; j<4;j++)
          {
            if (cTet.m_n[i] == mTet.m_n[j])
            {
              if (cTet.m_weak)
              {
              tID = cTedID;
              foundWeakElement = true;
                break;
              }
```

```
            }
          }
        }
      }

      btAlignedObjectArray<int> mTetras;
      mTetras.resize(m_tetras_size(),-1);
      splitWithWave(&mTetras, tID, m_eigenvector, m_eigenvalue);
      compareAndSplit(mTetras);
    }
  }
}
```

Listing 20: Initialisierung des Druckwellen-Verfahren

Nach der Methode von Glondu et al. werden ausgehend vom initialen Element die benachbarten Elemente per Breitensuche-Verfahren untersucht. Hierfür werden die von der Schnittebene geteilten Nachbar-Elemente in das Array *TetrasToHandle* eingetragen. Dieses Array wird nun der Reihe nach abgearbeitet. Die so entstehende „Welle“ endet, wenn keine Tetraeder mehr abgehandelt werden müssen oder die berechnete Energie nicht mehr ausreicht. In *alreadyUsedEnergy* wird durch einen Materialparameter und dem Tetraeder-Volumen die bisher verbrauchte Energie berechnet. Die verfügbare Energie *totalAvailableEnergy* wird durch das Tetraeder-Volumen, einem Koeffizienten zur Anpassung der Werte, und der aktuellen Belastung des Tetraeders wiedergegeben. Im Laufe der Abhandlung der Tetraeder werden die beiden Energie-Faktoren durch die beschriebenen Parameter des jeweiligen Tetraeder verändert. Sollte *alreadyUsedEnergy* den Wert *totalAvailableEnergy* übersteigen, wird die While-Schleife unterbrochen. Innerhalb dieser Schleife werden die Elemente auch darauf überprüft, ob sie von der Schnittebene durchkreuzt werden. Hierfür werden die vier Punkte des Tetraeders mit dem Massenpunkt des initialen Elements verglichen. Sollte die Überprüfung ergeben, dass das Element geschnitten werden muss, so wird es in *TetrasToHandle* eingefügt.

```
void splitWithWave(btAlignedObjectArray<int> *TetrasResults,
                   int TetraID, Vector3 &v, float eigenvalue )
{
  btAlignedObjectArray<int> TetrasToHandle;

  //first element to handle: given TetraID
  TetrasToHandle.push_back(TetraID);
  Tetra& firstTetra=m_tetras[TetraID];
  Vector3 mMasspoint =  0.25*(firstTetra.m_n[0]->m_x +
                              firstTetra.m_n[1]->m_x +
                              firstTetra.m_n[2]->m_x +
                              firstTetra.m_n[3]->m_x);

  btScalar fractureToughness = getFractureToughness();
  btScalar alignmentCoefficient = getAlignmentFactor();
```

```
  btScalar alreadyUsedEnergy= firstTetra.getVol()*
                                              fractureToughness;

  btScalar totalAvailableEnergy=firstTetra.getVolume()*
                    alignmentCoefficient*firstTetra.m_stress;

  while (TetrasToHandle.size() > 0 &&
                     (alreadyUsedEnergy<=totalAvailableEnergy))
  {
    //first element of the queue
    int tID = TetrasToHandle[0];
    Tetra& actualTetra=m_tetras[tID];

    alreadyUsedEnergy+= actualTetra.getVol()*fractureToughness;
    totalAvailableEnergy += actualTetra.getVolume()*
                    alignmentCoefficient*actualTetra.m_stress;

    for(int i=0;i<m_tetras.size();i++)
    {
      Tetra& compareT=m_tetras[i];
      bool positiveSide = false;
      bool negativeSide = false;

      for (int k =0; k<4; k++)
      {
        Vector3 orientation = compareT - mMasspoint;
        float dotProductbalancePoint = dot(orientation,v);

        if (dotProductbalancePoint >= 0)
          positiveSide = true;
        else
          negativeSide = true;}
      }
      if (positiveSide &&negativeSide)
      {
        TetrasResults[i] = 1;
        TetrasToHandle.push_back(i);
      }
    }

  //update the queue
  btSwap(TetrasToHandle[0],TetrasToHandle[TetrasToHandle.size()-1]);
  TetrasToHandle.pop_back();
  }
}
```

Listing 21: Untersuchung der Elemente durch die Druckwelle

secondary-Splits-Methode Durch die Methode *startSecondarySplitsMethod()* wird das selbst erarbeitete Verfahren gestartet. Sie wird in der Funktion *solveSoftBodies()* aufgerufen (Listing 8 auf Seite 42). Der Vergleich des

Eigenwerts aus *compareValues* mit dem Schwellwert *tau* ist wie in den bereits erläuterten Verfahren wieder die Bedingung für einen Bruch.

In diesem Verfahren richtet sich die Anzahl der durchgeführten Schnitte durch das Objekt nicht nach *maxSplit*, wie beispielsweise in Listing 20 auf Seite 53, sondern nach dem Faktor *maxSecSplits*. Der erste Knoten (*initNode*) wird mit der entsprechenden *fracture plane,* gebildet aus dem Eigenvektor des höchsten Eigenwerts, geteilt. Als nächstes werden die folgenden Brüche (*secondary splits*) nur noch an Massenpunkten initiiert, die vorher durch *m_cracknode=true* markiert wurden. Als *cracknode* werden alle Knoten behandelt, die sich bereits an einer Bruchkante befinden. Am Anfang der Methode *cutThisNode()* wird diese boolsche Variable für jeden Knoten gesetzt. Die Initiierung eines neuen Bruchs erfolgt nur, wenn die an diesem Punkt wirkende Belastung den gegebenen Schwellwert überschreitet. Dieser Schwellwert ist eine dezimierte Materialhaltbarkeit *tau*. Der Parameter der diesen Wert verkleinert ist *mfragility* (auf deutsch: Zerbrechlichkeit).

Die neue Schnittebene *newPlainOrientation* berechnet sich durch die Summe der Eigenvektoren des initialen und des aktuellen Knoten.

```
void startSecondarySplitsMethod(btAlignedObjectArray
                                        <Matrix4> compareValues)
{
  //toughness
  float tau = getParamMatrix().getRow(1).getX();

  //user-defined value
  int maxSecSplits = getParamMatrix().getColumn(2).getZ();
  float mfragility = 2.0;
  int secondSplits = 0;

  for(int split=0;split<compareValues.size();++split)
  {
    //node with highest stress
    int initNodeID = compareValues[0].getNodeID();
    Node* initNode = &m_nodes[initNodeID];
    float initNodeEigenvalue = compareValues[0].
                                        getHighestEigenValue();
    Vector3 initNodeEigenvector = getCorrespondingVector(
                                            initNodeEigenvalue);

    if (initNodeEigenvalue <= tau || secondSplits > maxSecSplits)
      break;

    if (initNodeEigenvalue > tau && split == 0)
    {
      splitWithSecondarySplits(initNodeID,initNodeEigenvector);
      split++;
    }

    int thisNodeID = compareValues[split].getNodeID();
    Node* thisNode = &m_nodes[thisNodeID];
```

```
    float thisNodeEigenvalue = compareValues[split].
                                        getHighestEigenValue();
    Vector3 thisNodeEigenvector = getCorrespondingVector(
                                                thisNodeEigenvalue);
    if (!thisNode->m_cracknode)
      continue;

    if ( thisNodeEigenvalue > tau/mfragility)
    {
      Vector3 newPlainOrientation = Vector3(thisNodeEigenvector+
                                                initNodeEigenvector);
      splitWithSecondarySplits(thisNodeID, newPlainOrientation);
      secondSplits++;
    }
  }
}
```

Listing 22: Initialisierung des secondary-Splits-Verfahren

In der Methode *splitWithSecondarySplits(),* welche die Knoten-ID und die Schnittebene als Parameter erhält, wird die Orientierung aller Tetraeder zum initialen Knoten ermittelt. Anschließend wird mit den erstellten Werten die Methode *compareAndSplit()* (siehe Listing 18 auf Seite 51) aufgerufen und das Objekt geteilt.

```
void splitWithSecondarySplits( int NodeID, Vector3 &v)
{
  Node* m_node=&m_nodes[NodeID];
  btAlignedObjectArray<int> tetraOrientation;

  for(int i=0;i<m_tetras.size();++i)
  {
    Tetra& m_tetra=m_tetras[i];
      for (int j =0; j<4; j++)
      {
        Node* c_node = m_tetra.m_n[j];
        Vector3 balancePoint += 0.25*c_node;
      }
      float dotProductbalancePoint = dot(balancePoint,v);

      if (dotProductbalancePoint < 0)
         tetraOrientation[i] = -1;
      else
        tetraOrientation[i] = 1;

  }
  compareAndSplit(tetraOrientation);
}
```

Listing 23: Untersuchung der Massenpunkte auf einen notwendigen Schnitt

6.2.4 Generierung neuer Kollisionsobjekte

Nach der Berechnung der Brüche der drei vorgestellten Verfahren müssen die Objekte für die weitere Kollisionserkennung zu neuen Teilobjekten zusammengefasst werden. Hierzu wird in der Methode *solveSoftBodies()* für jeden Soft Body die Methode *calcIndependentClusters()* aufgerufen. In dieser Methode werden zunächst mit Hilfe der Methode *SearchAppendedNodes()* alle zusammenhängenden Elemente gesucht und danach dem entsprechenden Cluster zugewiesen. Aus den so entstehenden Clustern werden in *initializeClusters()* und *updateClusters()* konvexe Hüllen berechnet, die im Verlauf der Simulation zur weiteren Kollision benutzt werden.

```
void calcIndependentClusters()
{
  btAlignedObjectArray<int> appNodeIds;
  appNodeIds.resize(m_nodes.size(),-1);

  int ClusterID = 0;
  for (int i = 0; i<appNodeIds.size();i++)
  {
    if (appNodeIds[i] == -1)
    {
      SearchAppendedNodes(&appNodeIds, i, ClusterID);
      ClusterID++;
    }
  }
  for(int i=0;i<ClusterID;i++)
  {
    for(int j=0;j<m_nodes.size();j++)
    {
      if (appNodeIds[j]== i)
        m_clusters[i]->m_nodes.push_back(&m_nodes[j]);
    }
  }
  initializeClusters();
  updateClusters();
}
```

Listing 24: Neuberechnung der Kollisionsobjekte

In der Methode *SearchAppendedNodes()* wird ausgehend von einem Knoten über alle Verbindungen iteriert, und die so verbundenen Knoten ermittelt. Alle bereits untersuchten Knoten werden für diesen Zweck mit der aktuellen *ClusterID* markiert.

```
void SearchAppendedNodes(btAlignedObjectArray<int> *NodeIds,
                                        int NodeID, int ClusterID)
{
  if (NodeIds[NodeID] == ClusterID)
    return;
  else
  {
```

```
    NodeIds[NodeID] = ClusterID;
    for (int i = 0; i<m_links.size(); i++)
    {
      //contains this link the searched node
      if (m_links[i].m_n[0] == NodeID)
        SearchAppendedNodes(&NodeIds, m_links[i].m_n[1], ClusterID);
      if (m_links[i].m_n[1] == NodeID)
        SearchAppendedNodes(&NodeIds, m_links[i].m_n[0], ClusterID);
    }
    return;
  }
}
```

Listing 25: Ermittlung der zusammenhängenden Knoten

7 Ergebnisse

Nachdem im vorherigen Abschnitt die Details der implementierten Verfahren erläutert wurden, werden in diesem Abschnitt die Ergebnisse der Untersuchung vorgestellt.

Alle Berechnungen wurden auf einem Notebook mit einem Intel Core i5 Prozessor (Taktrate: 2.50GHz) und 4 Gigabyte RAM durchgeführt. Die Untersuchungen der Verfahren fokussieren sich vor allem auf die Performanz und physikalische Plausibilität der Brüche. Die ermittelten Performanz-Werte der Simulation wurden durch den von BULLET bereitgestellten *profileIterator* berechnet. Weitere Aspekte werden im Anschluss zusätzlich in gekürzter Form aufgezeigt.

Zur Demonstration der vier Verfahren wurde eine Anwendung entwickelt, die es ermöglicht die beschriebenen Methoden in Echtzeit zu vergleichen. Hierbei hat der Anwender die Möglichkeit, eine parametrisierte Bruchsimulation zu starten. In Abbildung 19 ist diese Anwendung zu sehen. Die möglichen Konfigurationsparameter sind hierbei:

- Bruchsimulation
- Objekt
- Material und Materialhaltbarkeit τ
- Anzahl der maximalen Bruchiterationen
- Auswahl einer Belastungsansicht

Für die Präsentation der Ergebnisse wurden vier Materialien und insgesamt zehn Objekte in die Verfahren integriert. Das ausgewählte Objekt wird in der Anwendung mit zufälliger Ausrichtung aus einer Höhe von ”5” mit einer Schwerkraft von 10” fallen gelassen. Diese Ausgangsposition wäre in der

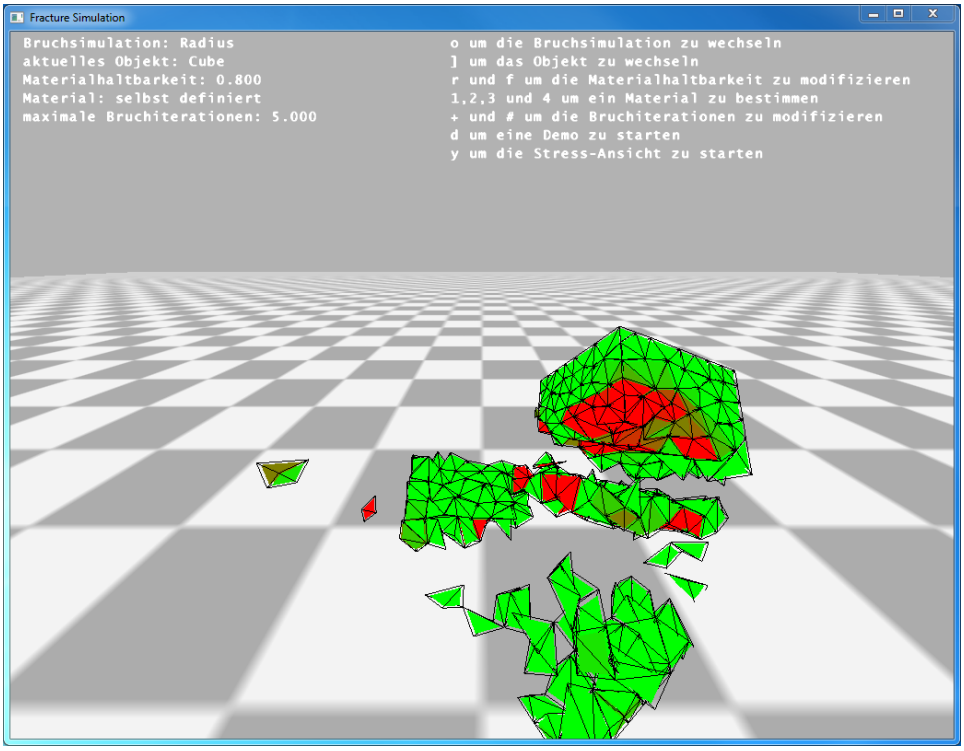


Abbildung 19: Anwendung zur Analyse der vier Verfahren.

Realität vergleichbar mit dem Fall einer Blumenvase aus einem Meter Höhe. Abbildung 20 zeigt beispielhaft eine Bruchfolge in sechs verschiedenen Zeitabschnitten. Diese Art der Darstellung findet sich in den Ergebnissen vermehrt, da sie eine gute Präsentation der simulierten Brüche ermöglicht. Im Anhang dieser Ausarbeitung befinden sich zusätzliche Resultate der durchgeführten Tests.

Eine Auflistung der durch NETGEN (Abschnitt 5.2) erstellten und in dieser Ausarbeitung verwendeten Modelle findet sich in Tabelle 2.

7.1 physikalische Plausibilität

Es ist schwierig, anhand ihrer Bruchstücke zu sagen, ob eine Bruchsimulation gültig bzw. realistisch ist. Wann ein Bruch korrekt ausgeführt wird, kann nicht gemessen werden. In der Literatur wird daher oft die physikalische Plausibilität für die Bewertung genutzt. Es werden dabei folgende Simulationsschritte zur Beurteilung unterschieden:

1. Bruchinitiierung
2. Bruchausbreitung und Resultate des Bruchs

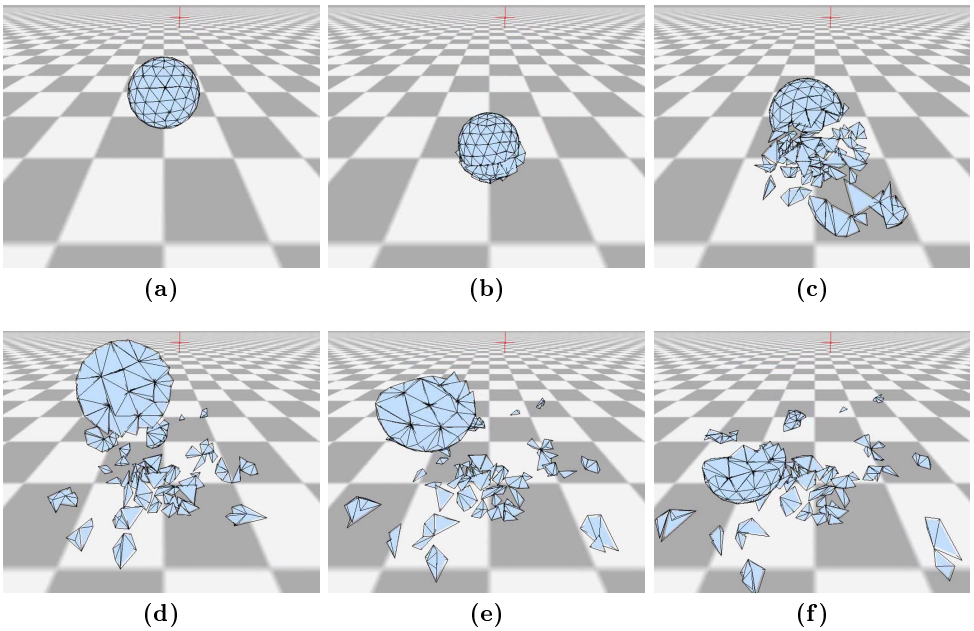


Abbildung 20: Bruch eines *Sphere-Objekts*. Simulation: *Radius,* Material: *Glas* mit $\tau = 6.01$, max. Bruchiterationen: *22*

Die Bruchinitiierung gibt an, ob und an welcher Position das Objekt bricht. Unter der Bruchausbreitung wird zusammengefasst, wie das Objekt geteilt wird. Sie berücksichtigt die Struktur des genutzten Objekts. Darüber hinaus fällt auch die Veränderung des ursprünglichen Objekts unter diesen Punkt.

7.1.1 Bruchinitiierung

Da die Bruchsimulation das Verhalten eines realen Objekts nachbilden soll, müssen die physikalischen Eigenschaften des Objekts berücksichtigt werden. Für diesen Zweck nutzen die analysierten Verfahren unterschiedliche Ansätze.

Basierend auf der Kontinuumsmechanik Für die Verfahren nach Müller et al. und Glondu et al. spielt hierbei eine hohe Genauigkeit in den Berechnungen eine wichtige Rolle. Die Bruchsimulation sollte seinem realen Vorbild so ähnlich wie möglich sein. Diese beiden Verfahren aus der Literatur und das selbst entwickelte Verfahren nutzen für dieses Ziel die auf der Kontinuumsmechanik basierte Berechnung der Belastung. Die beiden Hauptkomponenten, wie in Abschnitt 4.1.1 vorgestellt, sind dabei die Verformung des Objekts und die Änderung der aktuellen Geschwindigkeit. Die Verformung von Objekten ist durch die Objekte der *btSoftBodys* aus BULLET gegeben gewesen und ermöglichte eine darauf aufbauende Stressberechnung. Um die Plausibilität der berechneten Werte zu überprüfen, wurden die Belastungen

	Tetraeder	Knoten
Cube w.h.	357	182
Fichera	477	163
Sphere	345	136
Cube	1580	400
Cone	406	143

	Elemente	Constraints
Voronoi-Cube	76	346
Voronoi-Cuboid	143	652
Cuboid	513	6840
Cube w.h.(moderate)	99	591
Cube w.h.(fine)	357	3567

Tabelle 2: Eigenschaften der verwendeten Modelle

visuell in die Objekte eingetragen. Die daraus resultierende Belastungsansicht ist in Abbildung 21 und später folgenden Abbildungen zu sehen. Die grün eingefärbten Tetraeder sind dabei nicht bzw. wenig belastet, während die rot eingefärbten Tetraeder extrem stark belastet sind. Die Farben für die Belastungen zwischen diesen Werten werden interpoliert dargestellt. Als Referenzwert für die maximale Belastung wird die entsprechende Materialhaltbarkeit τ genutzt.

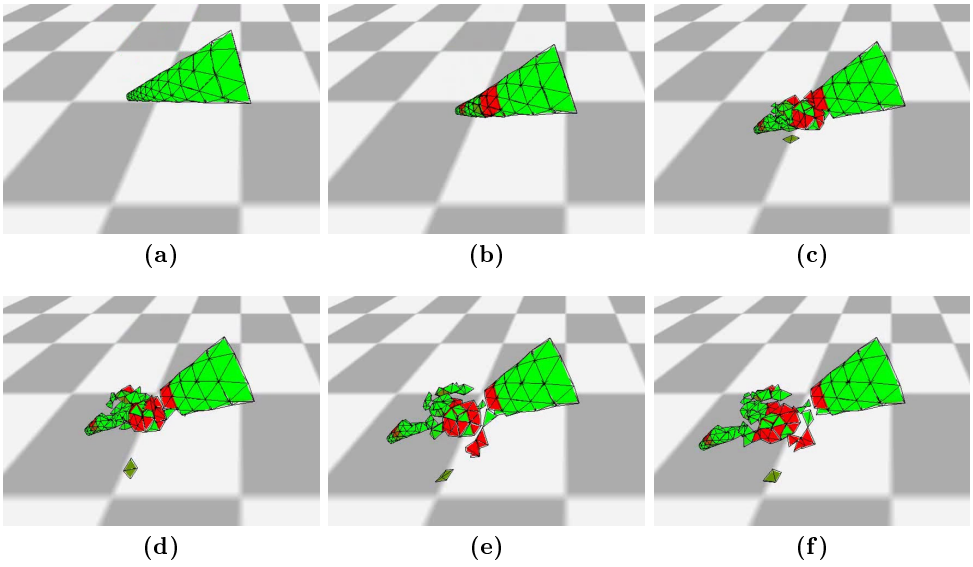
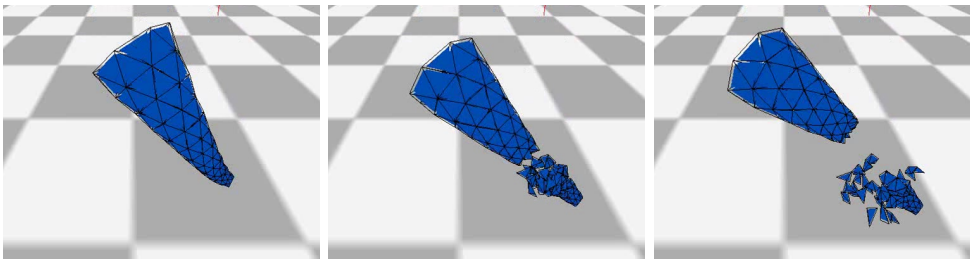


Abbildung 21: *Belastungsansicht*: Bruch eines *Cone-Objekts*. Simulation: *Radius,* Material: *Glas* mit $\tau = 6.01$, max. Bruchiterationen: *10*

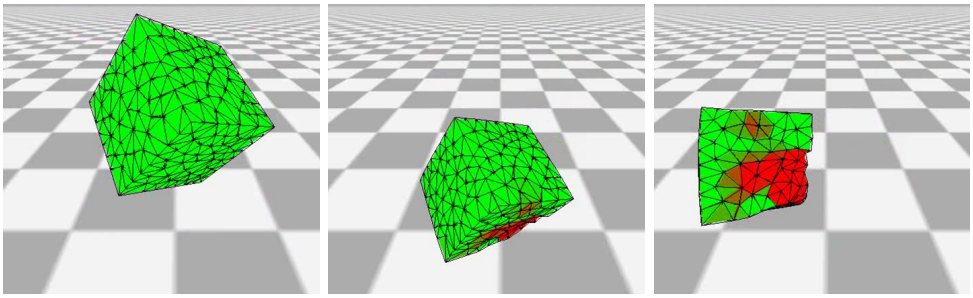
Die Ergebnisse dieser Darstellung lassen einige Rückschlüsse auf die berechneten Belastungen zu. Es wird deutlich, dass die Kräfte meist auf meh-



(a) (b) (c)

Abbildung 22: Bruch eines *Cone-Objekts*. Simulation: *secondary-Splits,* Material: *selbst definiert* mit $\tau = 0.5$, max. Bruchiterationen: *10*

rere zusammenhängende Tetraeder gleichzeitig wirken. Belastungen treten selten nur vereinzelt auf. Der Ursprung einer Belastung ist in den Abbildungen 21(b) und 22 zu sehen. Die Verformung der Spitze des *Cone-Objekts* ist der Auslöser für den Bruch. Abbildung 23 zeigt eine Simulation mit einer sehr hohen Materialhaltbarkeit. Die Folge ist, dass sich das Objekt -den Belastungen entsprechend- verformt, jedoch nicht bricht. Dieses Ergebnis ist auch auf die gegebenen Materialeigenschaften von Eisen zurückzuführen. Eisen hat, wie in den Materialeigenschaften in Abschnitt 3.3 auf Seite 10 erläutert eine sehr große duktile Verformung.

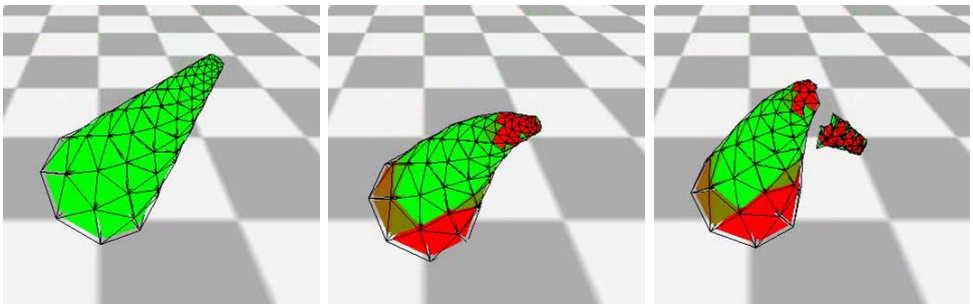


(a) (b) (c)

Abbildung 23: *Belastungsansicht*: *Cube-Objekt* bricht nicht und verformt sich stark. Simulation: *secondary-Splits,* Material: *Eisen* mit $\tau = 24.820$

Durch die Kombination der Elastizität der genutzten Soft-Bodys und der physiknahen Berechnungen kam es auch zu nicht zufriedenstellenden Ergebnissen. Das *Cone-Objekt* neigte in der Bruchsimulation dazu, häufig an seiner spitz zulaufenden Seite zu brechen. Der Grund dafür liegt darin, dass das entsprechende Tetraeder-Mesh an dieser Seite deutlich höher aufgelöst war. Diese Struktur hatte eine anschaulich höhere Elastizität in diesem Bereich zur Folge. Wenn dieses Objekt, wie in Abbildung 24 zu sehen, mit dem Boden kollidierte, krümmte sich die Spitze in den meisten Fällen. Diese Krümmung hatte häufig eine starke Objekt-Belastung mit Bruch zur Folge.

Dieses Phänomen zeigt die enorme Bedeutung der Objekt-Struktur für die Bruchsimulation. Es kann darüber hinaus als Ansatz für die Modellierung von schwachen Objektbereichen dienen.



(a) **(b)** **(c)**

Abbildung 24: *Belastungsansicht*: Bruch eines *Cone-Objekts*. Simulation: *Radius,* Material: *Glas* mit $\tau = 6.01$, max. Bruchiterationen: *10*

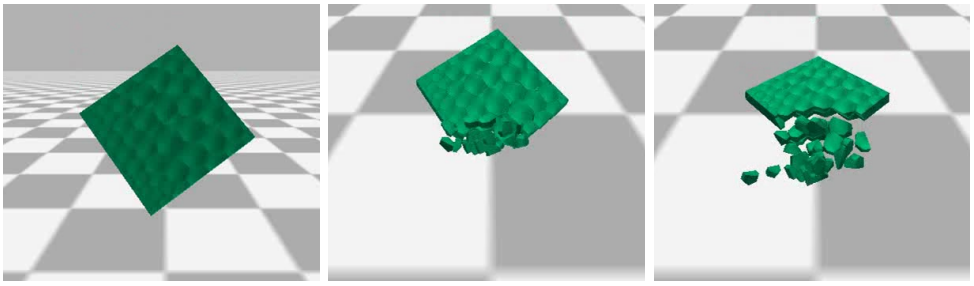
Die echten Werte, welche in der Belastungsansicht eingetragen sind, sind beispielhaft in Tabelle 3 zu sehen. Sie basieren auf der Eigenwert-Berechnung des Separationstensors ς. Im Laufe der Tests der implementierten Verfahren zeigte sich, dass die berechneten Eigenwerte nicht für jede Konfiguration gleich behandelt werden konnten. Die berechneten Werte unterschieden sich trotz relativ ähnlicher Ausgangsparameter (siehe Tabelle 1 auf Seite 12) zum Teil um den Faktor 1000. Da sich die Ergebnisse der Belastungen für die jeweiligen Konfigurationen gleichbleibend verhielten, konnte nach Anpassung der korrespondierenden Materialhaltbarkeit τ um eine Konstante die Vergleichbarkeit zu ς gesichert werden.

		Radius/secondary-Splits	*Wave*
Keramik	*Cube w.h.*	19	40
	Fichera	69	210
Glas	*Cube w.h.*	12	41
	Fichera	4	16
Eisen	*Cube w.h.*	6400	18800
	Fichera	2900	760

Tabelle 3: Maximum der Eigenwerte des Separationstensors ς bei unterschiedlichen Konfigurationen. .

Basierend auf Constraints Im Gegensatz zu den aufwändigen und exakten Berechnungen der drei weiteren implementierten Verfahren, nutzen Smith et al. eher eine Annäherung. Das Verfahren baut nicht, wie die bereits erläuterten Verfahren, auf Soft-Bodys sondern auf Rigid-Bodys, also

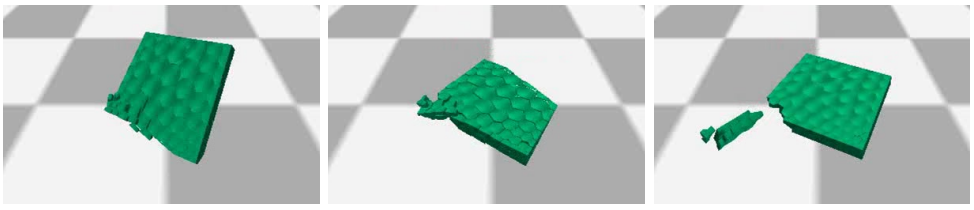
Starrkörpern auf. Auf Grund dieser „Starrheit“, kann die Verformung eines Objektes nicht als Basis der Berechnungen genutzt werden. Es bleibt aus der physikalischen Sicht daher nur die alternative Berechnung der sich ändernden Geschwindigkeit. Dieser so genannte Impuls wird als Vergleichswert zu den berechneten Constraints genutzt.



(a) (b) (c)

Abbildung 25: Bruch eines *Voronoi-Cuboid-Objekts* mit moderater Materialhaltbarkeit. Simulation: *Constraints,* Material: *selbst definiert* mit $\tau = 2.5$

Die Constraints, welche den „Zusammenhalt“ des Objekts gewährleisten, haben nur in kleinem Umfang eine physikalische Grundlage. Die Verbindungsstärke zwischen zwei Elementen richtet sich zum einen nach der Größe der gemeinsamen Fläche und zum anderen nach einem gegebenen Materialparameter. Die Berücksichtigung der Flächengröße spielt hierbei eine eher untergeordnete Rolle, da die verwendeten Tetraeder- bzw. Voronoi-Strukturen einen sehr uniformen Aufbau besitzen. Der Parameter der Material-Beschaffenheit hat jedoch einen großen Einfluss auf die durchgeführten Bruchsimulationen. Der Vergleich der Abbildungen 25 und 26 zeigt eine deutliche Veränderung des erzeugten Bruchbilds durch den Materialparameter. Je schwächer der Zusammenhalt der einzelnen Elemente ist, umso mehr kleine Bruchstücke entstehen.



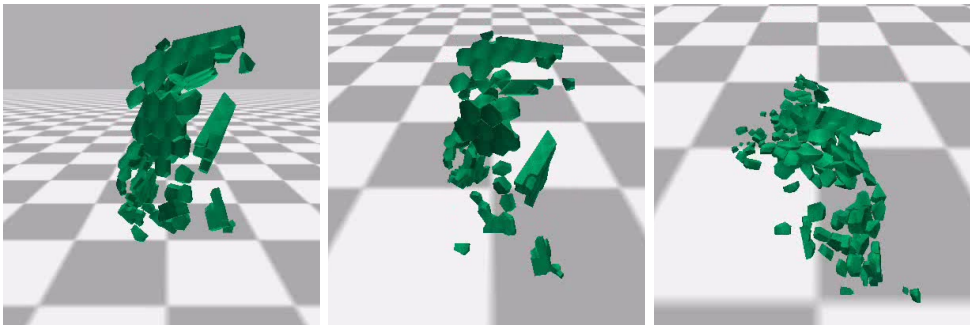
(a) (b) (c)

Abbildung 26: Bruch eines *Voronoi-Cuboid-Objekts* mit hoher Materialhaltbarkeit. Simulation: *Constraints,* Material: *selbst definiert* mit $\tau = 3.5$

Die Spanne dieses Materialparameters ist jedoch sehr klein und ermöglicht somit nur eine kleine Anzahl von verschiedenartigen Brüchen. Wenn der Materialwert $\tau \geq 4.0$ ist, zerbricht beispielsweise das *Voronoi-Cuboid-*

Objekt überhaupt nicht während es bei einem Wert $\tau \leq 2.2$ bereits in der Luft zerbricht. Das Problem dieses stellenweisen Bruchs liegt an dem nicht ausreichenden Zusammenhalt der Elemente des Objekts verglichen mit dem Impuls. Der Impuls, der auf die Teil-Elemente des Objekts wirkt, liegt in der Fallphase durch die Reibung der Elemente bei einem Wert > 0. Hierbei können bereits minimale Kräfte große Auswirkungen auf die Objekt-Struktur haben.

Diese Problematik wird in der originalen Ausarbeitung von Smith et al. durch die beschriebene *behaviour-function* umgangen. Diese Funktion, die die Belastung der Elemente beschreibt, ist im unbelasteten Zustand immer null. Sie wurde in dieser Ausarbeitung zwar angenähert, jedoch aufgrund der von BULLET gegebenen Struktur nicht unverändert übernommen. Die Ergebnisse eines bereits in der Luft zerfallenen Objekts findet sich in Abbildung 27 wieder.



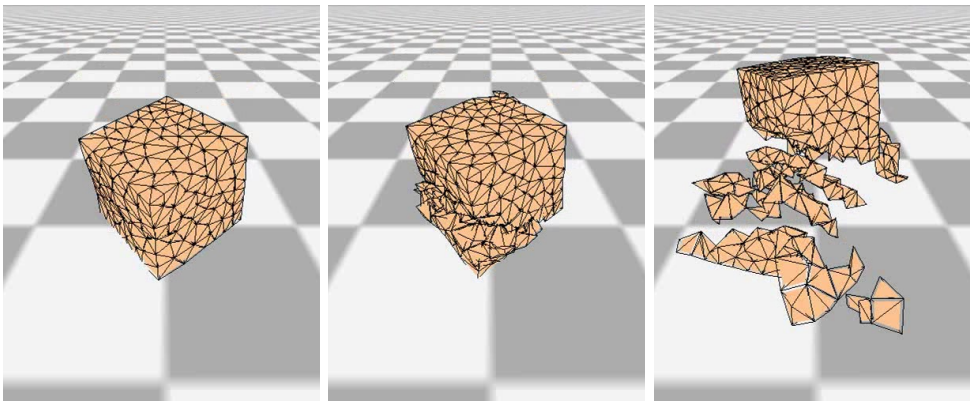
(a) (b) (c)

Abbildung 27: *Voronoi-Cuboid-Objekts* bricht, wegen niedriger Materialhaltbarkeit, bereits in der Luft. Simulation: *Constraints,* Material: *selbst definiert* mit $\tau = 1.3$

7.1.2 Bruchausbreitung und Resultate des Bruchs

Für die eigentliche Durchführung des Bruchs nach der Initiierung sind zwei Parameter der untersuchten Verfahren wichtig. Zum einen die Struktur der zugrunde liegenden Objekte, und zum anderen die eigentliche Technik der Ausbreitung eines Bruchs und deren Resultate.

Struktur Die Objekt-Struktur für die drei auf der Kontinuumsmechanik basierenden Verfahren ist in allen Fällen gleich. Sie nutzen eine Tetraeder-Mesh-Struktur. Dieses Struktur wurde in BULLET durch die erläuterten Soft Bodys umgesetzt. Der große Vorteil dieser Mesh-Struktur ist die mögliche Simulation jedes einzelnen Massenpunktes. Dieser Aufbau ist nicht nur für die Berechnung der für die Belastung entscheidenden Verformung immens wichtig, sondern ermöglicht auch eine realistischere Verbreitung der Brü-

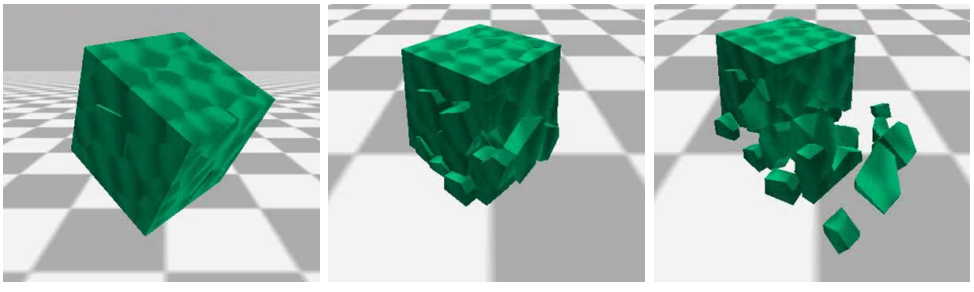


(a) (b) (c)

Abbildung 28: Bruch eines *Cube-Objekts*. Simulation: *secondary-Splits,* Material: *Keramik* mit $\tau = 2.48$, max. Bruchiterationen: *5*

che. Obwohl die genutzten Objekte durch teilweise über Tausende von Teil-Elementen angenähert werden, ist eine weitere Feingranularität in diesem Bereich durchaus denkbar. Die Struktur der über das Objekt verteilten Massenpunkte macht es darüber hinaus möglich, weitere Informationen zur Optimierung eines Bruchs innerhalb des Objekts abzuspeichern. In der entwickelten *secondary-Splits-Methode* werden beispielsweise die vorgestellten *cracknodes* vermerkt. Die genutzte Struktur ermöglicht die Einbindung von Objekten jeglicher Komplexität. Es werden sowohl konkave als auch „hohle“ Objekte unterstützt.

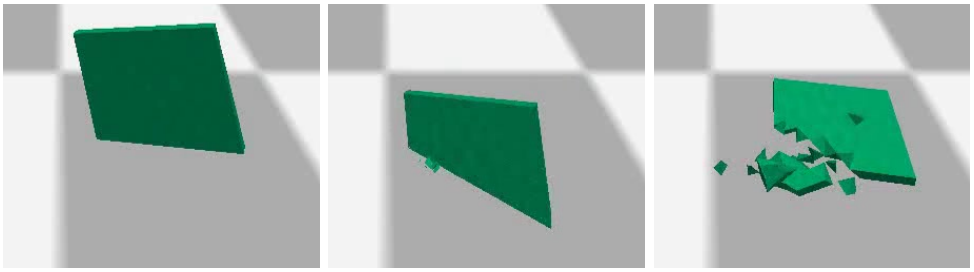
Durch die Konstruktion anhand der Delaunay-Triangulierung haben die Meshs die typische, gitterartige Struktur. Die dargestellten, spitzen Tetraeder verhindern eine schöne Form der generierten Bruchstücke. Um die entstehenden Formen bereits durch die Struktur des Ausgangsobjekts zu verbessern, bieten sich unterschiedliche Möglichkeiten an. Eine irreguläre Netz-Struktur, höher aufgelöste und „klüger“ modellierte Meshs werden hierfür in der Literatur verwendet.



(a) (b) (c)

Abbildung 29: Bruch eines *Voronoi-Cube-Objekts* mit niedriger Materialhaltbarkeit. Simulation: *Constraints,* Material: *selbst definiert* mit $\tau = 1.5$

Bei der *Constraint-Methode* beeinflusst die Struktur der Ausgangsobjekte die Brüche in noch stärkerem Ausmaß. Das Objekt wird durch mehrere Teil-Objekte zusammengestellt. Durch diese Aufteilung scheint das Gesamt-Objekt bereits vor der ersten Belastung zerbrochen. In der Literatur wird dieses Phänomen „prefractured parts“ genannt. Die vorgefertigten Brüche erlauben kein Abweichen von den bereits bestehenden Kanten der Teil-Objekte. Die durchgeführten Brüche erscheinen durch die Starrheit der Unterelemente zum Teil zusätzlich sehr statisch. Da sich die Teil-Objekte keine Massenpunkte teilen, sind die Constraints das Fundament, warum die Teile zusammenhalten. Diese Constraints können keine physikalischen Brüche ergeben, da jedes Constraint für sich nur an einem kleinen Punkt innerhalb des Objektes wirkt. Die Bedingungen, welche die einzelnen Teil-Objekte aneinander halten, haben keine Informationen über den Status der anderen Constraints. Um dieser Struktur einen stärkeren physikalischen Hintergrund zu geben, werden in der Literatur nicht uniforme Verbindungsstärken eingesetzt. Es werden für diesen Zweck schwache Elemente innerhalb der Struktur simuliert, indem deren Verbindungsstärke verändert wird. In [10] werden hierzu beispielsweise Rauschfunktionen oder die Simulation von „schwachen Adern“ eingesetzt.

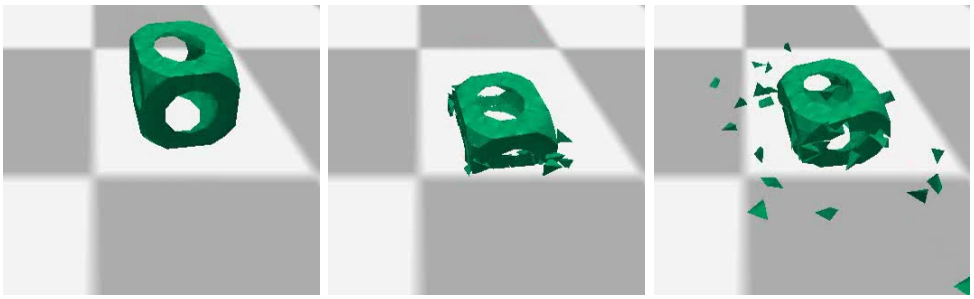


(a) **(b)** **(c)**

Abbildung 30: Bruch eines *Cuboid-Objekts*. Simulation: *Constraints,* Material: *selbst definiert* mit $\tau = 2.0$

Während der durchgeführten Implementierung des *Constraint-Verfahren* wurde aus Performanz-Gründen zusätzlich zu der eigentlich von Smith et al. eingesetzten Tetraeder-Struktur (siehe Abbildungen 30 und 31) die Voronoi-Struktur (Abb. 29) eingesetzt. Wie in Tabelle 2 zu sehen, konnte so die Anzahl der für die Objekte notwendigen Constraints um zum Teil das zehnfache reduziert werden.

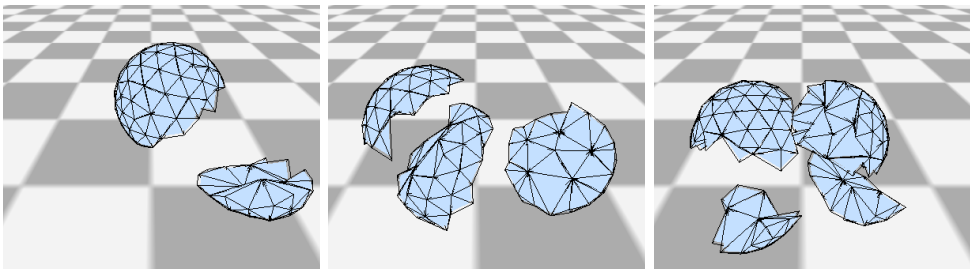
Technik/Resultate Die Techniken der Bruchausbreitung führen zu unterschiedlichen Resultaten in den Brüchen. In den Abbildungen 32 und 33 sind die Resultate der drei auf der Kontinuumsmechanik basierenden Verfahren aufgezeigt. Es sind die Ergebnisse eines Bruchs mit maximal 1 bzw. maximal 10 Bruchiterationen. Das selbst entwickelte Verfahren *secondary-*



(a) (b) (c)

Abbildung 31: Bruch eines *Cube with Holes-Objekts* mit feiner Auflösung (357 Elemente). Simulation: *Constraints,* Material: *selbst definiert* mit $\tau = 1.8$

Splits hat in diesen Beispielen auf Grund des initialen Bruchs jeweils eine Bruchiteration mehr vollzogen (2 bzw. 11).



(a) *Radius-Methode* (b) *Wave-Methode* (c) *Secondary-Splits-Methode*

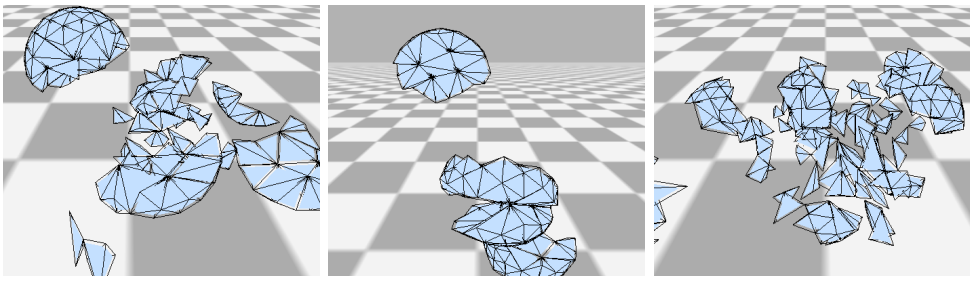
Abbildung 32: Resultate eines Bruchs des *Sphere-Objekts* mit unterschiedlichen Verfahren. Material: *Glas* mit $\tau = 6.01$, max. Bruchiterationen: *1*

Bei spröden Materialien bedeutet eine Belastung, die zu einem Bruch führt, in der Realität immer einen Durchbruch, also die komplette Teilung des Objekts. Alle hier vorgestellten Verfahren setzen diesen Ansatz erfolgreich in ihren Methoden um. Zur realistischen Verbreitung eines Bruchs innerhalb des Objekts bedienen sich die Verfahren, wie in ihren Erläuterungen in Abschnitt 4 beschrieben, unterschiedlicher physikalischer Phänomene. Im Folgenden werden die Techniken noch einmal kurz skizziert.

- *Radius-Verfahren:* Der Ausgangspunkt dieses Verfahrens ist der Massenpunkt, der am stärksten belastet wird. Der zu dieser Belastung passende Eigenvektor bildet die Schnittebene. Auf Grundlage der Belastungsstärke wird ein Umkreis zum belasteten Punkt festgelegt. Alle Punkte des Objekts, die sich in diesem Umkreis befinden, werden mit der Schnittebene verglichen und einer der beiden Hälften zugeordnet. Der physikalische Ansatz dieses Verfahrens ist der Zusammenhang zwischen dem gewählten Radius und der Größe der Belastung.

- *Wave-Verfahren:* Die Schnittebene wird durch den zur höchsten Belastung korrespondierenden Eigenvektor bestimmt. Ausgehend von dem am stärksten belasteten Element werden nun die Nachbarelemente auf ihre Ausrichtung zur Schnittebene überprüft. Nach dem Verfahren der Breitensuche bewegt sich so eine Art „Welle“ durch das Objekt. Jedes Element, welches untersucht wurde und mit dem Bruchkriterium übereinstimmt, wird abgespeichert. Das Stopkriterium für diese Welle ist hierbei durch eine Energie-Funktion gegeben. Diese Energie-Funktion berücksichtigt die Materialhaltbarkeit, Belastung und Struktur des Objekts.

- *Secondary-Splits-Verfahren:* Das Objekt wird initial am Massenpunkt mit der höchsten Belastung auf Basis des gegebenen Eigenvektors geschnitten. Nach diesem initialen Schnitt werden nur noch Massenpunkte für die Initiierung eines neuen Bruchs genutzt, welche sich an einer Bruchkante befinden. Die nun neu generierten Brüche haben eine Ausrichtung, die sowohl die Richtung des bestehenden Bruchs, als auch die Richtung der wirkenden Kräfte berücksichtigt.



(a) *Radius-Methode* **(b)** *Wave-Methode* **(c)** *Secondary-Splits-Methode*

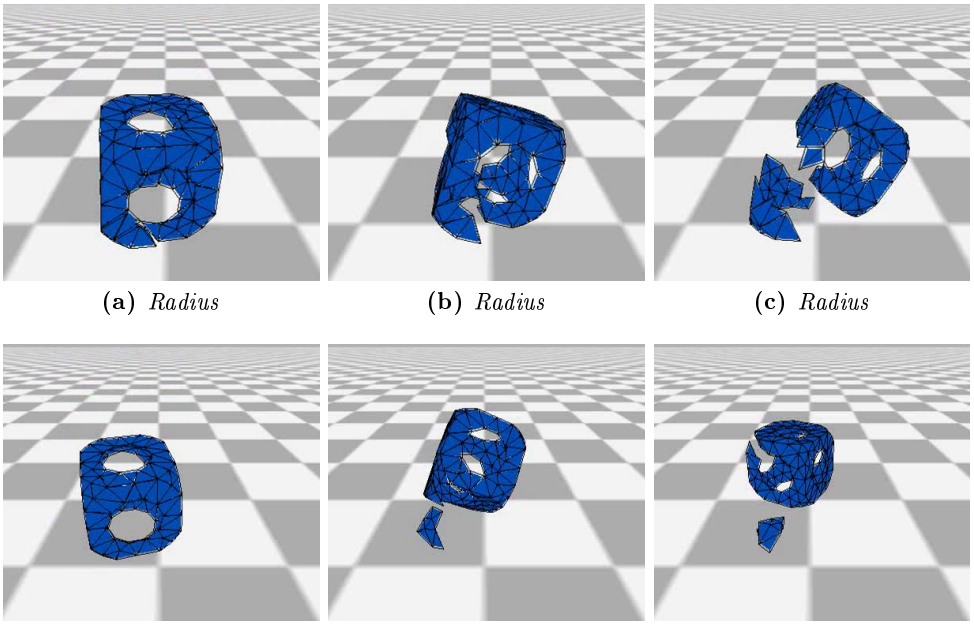
Abbildung 33: Resultate eines Bruchs des *Sphere-Objekts* mit unterschiedlichen Verfahren. Material: *Glas* mit $\tau = 6.01$, max. Bruchiterationen: *10*

Die Resultate dieser drei „physik-nahen“ Verfahren sind dabei in einigen Bereichen unterschiedlich. Bei der Betrachtung der Ergebnisse in den Abbildungen 32 und 33 fällt zunächst die unterschiedliche Anzahl der entstandenen Bruchstücke auf. Die Tabelle 4 im Performanz-Abschnitt zeigt die durchschnittlich generierten Bruchstücke aller Objekte.

Durch Abbildung 32 kann die unterschiedliche Anzahl auf niedrigerem Bruchiterationen-Niveau erläutert werden. Die entstandenen Ergebnisse sind auf die einzelne Bruch-Technik zurückzuführen. Bei der *Radius-Methode* wird das ursprüngliche Objekt in zwei Stücke unterteilt, da genau eine Schnittebene durch das Objekt führt und dieses separiert. Die in (b) gezeigt *Wave-Methode* separiert das Objekt hingegen in 3 Bruchstücke. Der Grund hierfür ist die durchgeführte „Energie-Welle“. Sowohl die Elemente oberhalb, als auch

die Elemente unterhalb der Schnittebene werden markiert. Beim eigentlichen Bruch wird diese Markierung mit den benachbarten Elementen verglichen und führt bei Ungleichheit zur Trennung. Das dritte Bild (*Secondary-Splits-Verfahren)* weist 4 Bruchstücke auf. Diese Bruchstücke entstehen durch den einen initialen/primären Bruch und einen durch beide Bruchstücke gehenden sekundären Bruch.

Die moderate Anzahl an Bruchstücken bei höherer Anzahl an Bruchiterationen liegt bei der *Wave-Methode* daran, dass neu entstehende Brüche nicht über Element- und Bruchgrenzen hinaus laufen dürfen. Abbildung 34 zeigt, warum dieses Kriterium physikalisch plausibel ist.



(a) *Radius* **(b)** *Radius* **(c)** *Radius*

(d) *Wave* **(e)** *Wave* **(f)** *Wave*

Abbildung 34: Bruch eines *Cube with Holes-Objekts.* Oben: Simulation: *Radius,* Unten: Simulation *Wave,* Material: *selbst definiert* mit $\tau = 12.0$, max. Bruchiterationen: *1*

In diesem Test wird ein Knoten (*Radius-Verfahren*) bzw. ein Element (*Wave-Verfahren*) zur Bruchinitiierung genutzt. Da die Belastung des Massenpunktes im *Radius-Verfahren*-Beispiel sehr hoch ist, wird ein großer Umkreis der Massenpunkte des Objekts zum Vergleich mit der berechneten Schnittebene genutzt. Die Folge ist der Bruch an zwei Stellen des Objekts. Zum einen an der hoch belasteten Stelle, und zum anderen an der gegenüberliegenden Seite des Objekts. Da die Brüche jedoch theoretisch nur durch das Objekt und nicht durch die Luft übertragen werden können, ist dieses Ergebnis unrealistisch. Die *Wave-Methode,* welche in der unteren Bildreihe abgebildet ist, zeigt hierbei eine realistischere Lösung. Der Bruch überträgt sich nur auf tatsächlich anliegende Elemente und bleibt somit lokal.

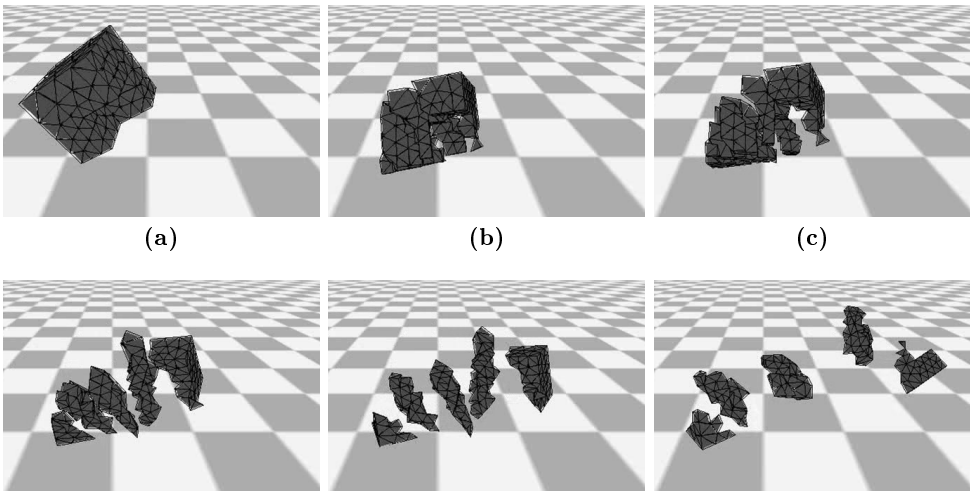


Abbildung 35: Bruch eines *Fichera-Objekts*. Simulation: *Waves,* Material: *Eisen* mit $\tau = 24.82$, max. Bruchiterationen: *3*

Beim weiteren Vergleich der Resulate der *Wave-Methode* mit den beiden anderen Verfahren fällt auf, dass es nicht nur deutlich weniger Bruchstücke sind, sondern diese auch eine andere Form haben. Die Bruchstücke sind häufig größer und untereinander ähnlich. Ein weiteres Beispiel zu diesem Umstand ist in Abbildung 35 zu sehen. Die Ausrichtung der Schnittebene ist durch einen ähnlichen Eigenvektor annähernd gleich. Eine nicht so uniforme Bildung der Brüche ist in der *Secondary-Splits-Methode* sichtbar. Durch die Kombination verschiedener Belastungsrichtungen der Massenpunkte entsteht ein Bruchmuster mit vielen unterschiedlich großen und unterschiedlich geformten Bruchstücken.

Ob die initiale Ausrichtung der Schnittebene der drei Verfahren physikalisch plausibel ist, lässt sich schwer sagen, da auch in der Realität die Richtung der Brüche nicht immer einheitlich verläuft. In der Literatur wird darauf hingewiesen, dass sich bildende Brüche im Normalfall orthogonal zur Richtung der größten Zugspannung bilden sollten. Dieser Fakt wird in der Belastungstensor-Berechnung aller drei Verfahren berücksichtigt.

Ein komplett anderes Bild zeigt sich in der Constraint-Methode. In der Implementierten Methode von Smith et al. ist keine wirkliche Ausbreitung des Bruchs vorgesehen. Lediglich eine ansatzweise, physikalische Lösung der Einflussnahme auf andere Elemente ist in diesem Verfahren vorgesehen. Sollte ein Constraint innerhalb eines Objekts entfernt werden, so werden alle benachbarten Constraints um einen bestimmten Teil der übrigen Energie geschwächt. Diese Relaxation auf andere Elemente führte jedoch in der Implementation oft zu instabilen Objekten. Die Folgen sind die selben, wie diejenigen des bereits im Abschnitt der Bruchinitiierung und Abbildung 27

erläuterten Probleme. Bereits kleine Impulse führten innerhalb des Objekts zu einer Massenreaktion und führten zum Bruch beinahe aller Constraints. Um diesen Effekt einzudämmen, mussten die Übertragungen durch einen Schwellwert kontrolliert werden.

7.1.3 Bewertung

Die Überprüfung der Verfahren ergab, dass jedes für sich eine physikalischen Ansatz besitzt. Die Intensität, mit der die physikalischen Eigenschaften eines Objekts berücksichtigt werden, divergieren dabei aber sehr stark.

Die drei auf der Kontinuumsmechanik basierenden Verfahren haben eine sehr exakte Berechnung der wirkenden Kräfte als Grundlage und besitzen dadurch ein physikalisch sehr plausibles Fundament für ihre Bruchsimulation. Die Nutzung dieser berechneten Belastungen wird durch unterschiedliche Techniken fortgeführt. Das *Wave-* und das *Radius-Verfahren* stützen dabei physikalisch plausibel ihre Brüche auf der an die Methoden übergebenen Belastungen. Das selbst entwickelte Verfahren nimmt diese Belastung ebenfalls als Ausgangspunkt. Zusätzlich dazu wird die Strukturveränderung nach dem ersten Bruch berücksichtigt. Die dadurch entstehenden schwachen Teile des Objekts werden nun für die weitere Bruchsimulation genutzt.

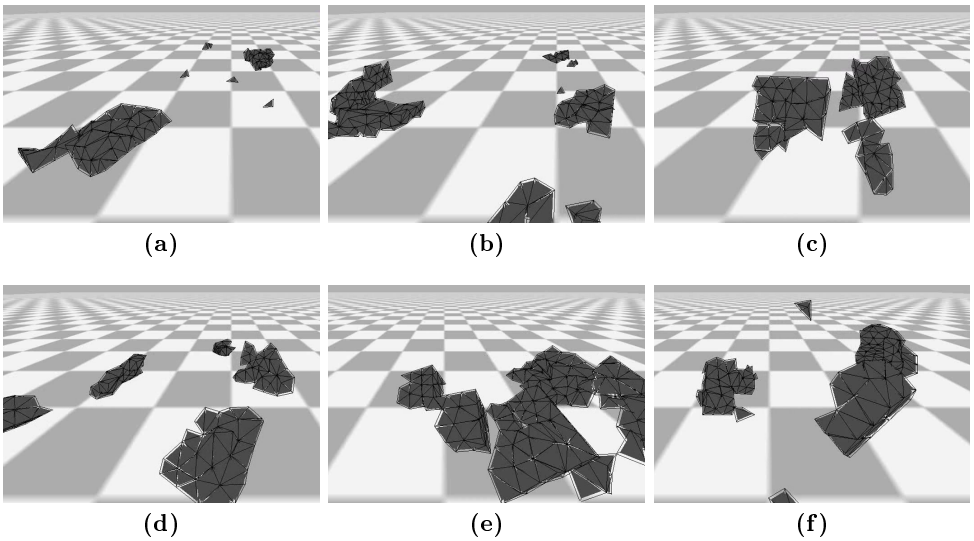


Abbildung 36: Bruch mehrerer *Fichera-Objekte* mit den selben Parametern. Simulation: *Waves,* Material: *Eisen* mit $\tau = 24.820$, max. Bruchiterationen: 4

Alle drei Verfahren konnten durch ihre Vielseitigkeit überzeugen. Im Gegensatz zu dem *Constraint-Modell* haben bei diesen Methoden die gegebenen Parameter und Konfigurationen einen sichtbaren Einfluss auf die Bruchsimulation. Je nachdem, mit welcher Orientierung ein Objekt mit dem Boden kol-

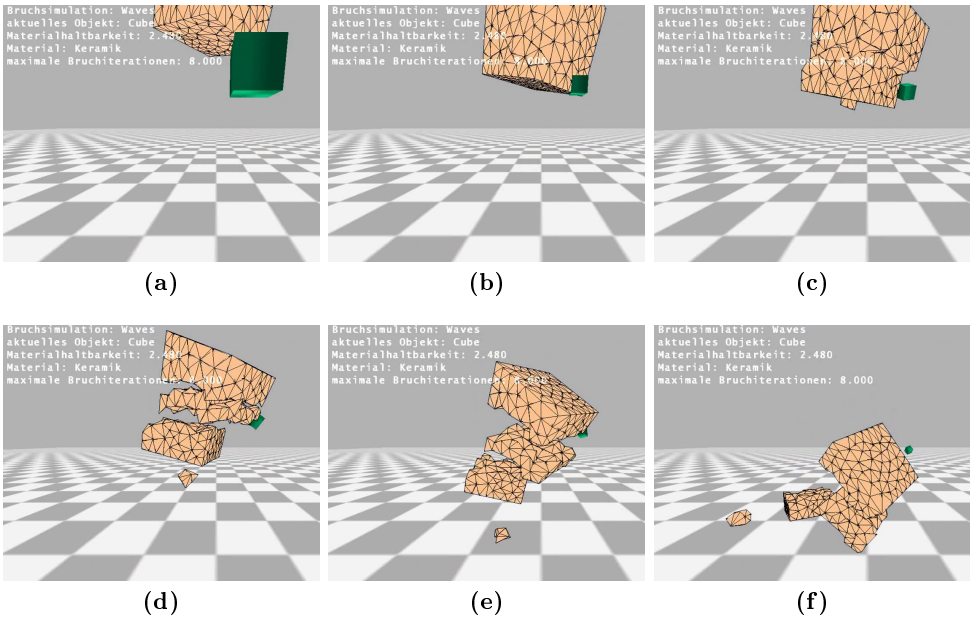
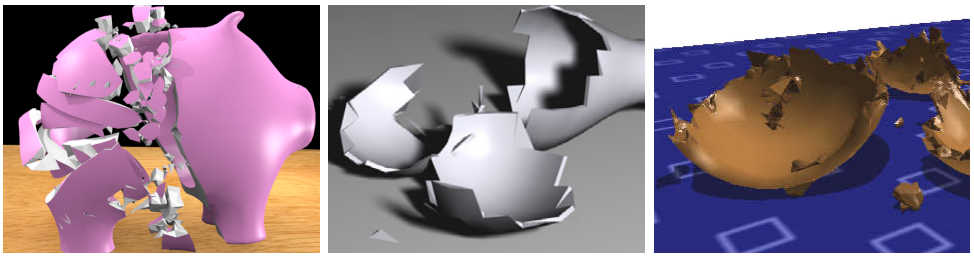


Abbildung 37: Bruch eines *Cube-Objekts* durch ein Geschoss. Simulation: *Waves,* Material: *Keramik* mit $\tau = 2.48$, max. Bruchiterationen: *8*

lidierte, entstanden physikalisch plausibel andere Belastungen und dementsprechend andere Brucherzeugnisse. Das Beispiel in Abbildung 36 zeigt sechs unterschiedliche Bruch-Resultate, die bei gleichen Parametereinstellungen entstanden sind. Wenn diese Parameter anders konfiguriert werden, ergeben sich dem Benutzer dieser Verfahren zusätzlich einige Möglichkeiten, um diverse Materialien zu nutzen und unterschiedliche Szenarien zu konstruieren. So sind die implementierten Verfahren nicht nur für fallende Objekte geeignet, sondern lassen sich variabel für andere Situationen verwenden. Ein Beispiel dafür ist in Abbildung 37 zu sehen. Alle drei Verfahren konnten in ihren Ergebnissen durch ihre realistischen Berechnungen überzeugen. Wenn die implementierten Methoden durch beispielsweise Remeshing-Verfahren oder finite Elemente optimiert würden, könnten sogar ästhetisch anspruchsvolle Brüche entstehen.

Das *Constraint-Verfahren* von Smith et al. bietet einen Ansatz, bei dem das Objekt durch zusammengesetzte starre Körper definiert ist. Die wesentliche Technik der Bruchsimulation basiert auf der Entfernung der initiierten Constraints. Da die Objekte durch ihre Teil-Elemente theoretisch schon gebrochen sind, muss hierbei nur der simulierte Zusammenhalt gelöst werden. Um einen komplexen Bruch oder einen Bruch durch ein komplettes Objekt zu vollziehen, ist diese Basis nicht ausreichend. Abbildung 25 und 29 lassen diesen Umstand ebenfalls erahnen. Ein Manko des umgesetzten Verfahren ist des Weiteren die geringe Möglichkeit der Anpassung. Lediglich die Verbindungsstärke des Objekts kann vorher festgelegt werden. Hinzu kommt,



(a) *Wave-Verfahren* nach Glondu et al.[7]

(b) *Radius-Verfahren* nach Müller et al.[14]

(c) *Constraint-Verfahren* nach Smith et al.[21]

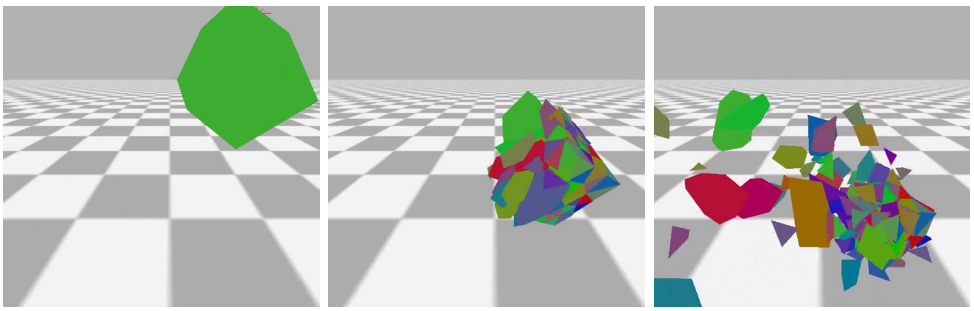
Abbildung 38: Bruchergebnisse aus den Original-Ausarbeitungen.

dass genau diese Variable aufgrund der beschriebenen Probleme nicht viel Spielraum zulässt. Massenhaft unterschiedliche Materialien können aus diesem Grund nicht simuliert werden. Das Verfahren konnte trotz seiner recht simplen Berechnungen und Technik sehr solide Brüche erzielen.

In der Literatur [6] werden die kontinuumsbasierten Methoden ebenfalls als physikalisch am plausibelsten bezeichnet. Die *Constraint-Methode* wird als simpel und effizient beschrieben.

Beispiele aus den Original-Arbeiten der drei implementierten Verfahren sind in Abbildung 38 zusammengefasst. Die in dieser Ausarbeitung erzielten Ergebnisse finden sich zum Teil in den abgebildeten Resultaten wieder. Die Problematik der Struktur der regulären Meshs ist beispielsweise auch hier zu erkennen. Da es sich um hohle Objekte und erkennbar aufwändiger gerenderte Ergebnisse handelt, ist ein visueller Vergleich schwierig. Die zu sehenden Bruchstücke sollten auf physikalischer Ebene jedoch sehr ähnlich sein.

7.2 Performanz



(a) **(b)** **(c)**

Abbildung 39: *Clustersansicht*: Bruch eines *Fichera-Objekts* mit 129 entstandenen Bruchstücken.

In Computerspielen oder anderen Anwendungen der virtuellen Realität spielt die Performanz der Simulation eine übergeordnete Rolle. Der Nutzer

darf nicht durch „ruckelnde“ Bilder oder langsame Animationen in seiner Immersion gestört werden. Eine Einhaltung der maximalen Rechenzeit ist sehr wichtig, um den Vorgang einer interaktiven Anwendung nicht auszubremsen. In dieser Ausarbeitung wurde die Anforderung aufgestellt, dass die Simulation der Brüche mindestens in Echtzeit ablaufen soll. Da pro Zeitschritt viele Prozesse gleichzeitig stattfinden können, müssen Simulationen häufig sogar schneller als in Echtzeit ablaufen.

7.2.1 Messungen/Daten

Eingabe und Bilderstellung haben in der entwickelten Anwendung nur einen marginalen Einfluss auf die Performanz und können in diesem Fall bei der Betrachtung vernachlässigt werden. Die entscheidenden Ausführungszeiten werden in diesem Vergleich durch Kollision und Bruchsimulation an sich bestimmt.

	Cube w.h.	*Sphere*	*Cube*
max. *3* Iterationen:			
Radius	9	6	6
Wave	6	4	6
secondary-Splits	9	10	12
max. *8* Iterationen:			
Radius	22	14	15
Wave	9	5	8
secondary-Splits	30	21	20
max. *15* Iterationen:			
Radius	38	35	55
Wave	10	6	11
secondary-Splits	42	50	70

Tabelle 4: Generierte Bruchstücke / Erstellte *Cluster*. Die eingetragenen Werte sind Durchschnittswerte von 10 durchgeführten Brüchen. Die Bruchsimulation wurde mit einer selbst definierten Materialhaltbarkeit von $\tau = 0.5$ durchgeführt.

Die Kollisionsüberprüfung bei Objekten, die aus vielen Teilelementen bestehen, in Echtzeit durchzuführen, kann bei vielen und komplexen/konkaven Objekten problematisch sein. Da sich die Massenpunkte innerhalb eines Objekts und somit auch die Form des Objekts ständig ändern, müssen auch die Kollisionsobjekte ständig angepasst werden. In dieser Ausarbeitung wurde die Kollisionsüberprüfung auf den in Abschnitt 5.1 vorgestellten Clustern aufgebaut. Diese Cluster sind die konvexe Hüllen von bestimmten Punktemengen. Im Fall der Bruchsimulation ist jedes Cluster nach dem vollzogenen Bruch ein Bruchstück. In Abbildung 39 ist ein Bruch mit abgebildeten Clus-

tern zu sehen. Die Anzahl der generierten Bruchstücke basiert vor allem auf der Größe der Belastung und der Anzahl der Bruchiterationen. In der Tabelle 4 sind Beispielwerte für drei unterschiedliche Objekte zu finden. Wenn die Anzahl der Bruchiterationen und somit auch der Bruchstücke zu sehr steigt, kann die Simulation nichtmehr in Echtzeit berechnet werden. Yilmaz et al. [10] zeigten in ihrer Ausarbeitung, dass durch die weitere Behandlung der Bruchstücke sehr große Einbußen der Performanz entstehen können. In ihrer Ausarbeitung implementierten sie ebenfalls das *Constraint-Verfahren* von Smith et al..

	Cube w.h.	*Fichera*	*Sphere*	*Cone*	*Cube*
max. *3* Iterationen:					
Radius	17	20	17	16	135
Wave	16	25	16	16	105
secondary-Splits	17	21	20	17	145
max. *8* Iterationen:					
Radius	27	28	28	26	270
Wave	19	28	16	19	120
secondary-Splits	30	30	28	26	250
max. *15* Iterationen:					
Radius	38	45	40	42	480
Wave	20	40	16	21	150
secondary-Splits	44	50	48	40	590

Tabelle 5: Zeit für die Berechnung und Durchführung eines Bruchs in Millisekunden (*ms*). Die eingetragenen Werte sind durch den BULLET *profileIterator* gemessene Durchschnittswerte. Die Bruchsimulation wurde mit einer selbst definierten Materialhaltbarkeit von $\tau = 0.5$ durchgeführt.

Die Tabelle 5 zeigt gemessene Werte für die fünf genutzten Objekte. Die eingetragenen Zeiten inkludieren die Kollisionsbehandlung und alle anderen für die Simulation notwendigen Berechnungen. Der Standardwert für einen Zeitschritt in der durchgeführten Simulation beträgt *16ms* und entspricht somit mehr als 60 Bildern pro Sekunde. Da der Mensch nur in etwa 25 bis 30 Bilder pro Sekunde verarbeitet, kann der Wert *35ms* in etwa als Grenzwert für die Echtzeitfähigkeit genutzt werden. Durch Beobachtungen der simulierten Anwendung wurde herausgefunden, dass sogar bei bis zu *45ms* pro Zeitschritt von einer durchgehend flüssigen Simulation gesprochen werden kann. Wenn das mit seiner großen Anzahl an Tetraedern sich abhebende *Cube-Objekt* vernachlässigt wird, fällt auf, dass fast alle Konfigurationen echtzeitfähig sind. Da acht Iterationen im Normalfall nicht überschritten werden, können die vier Objekte *Cube w.h., Fichera, Sphere* und *Cone* in den meisten Fällen von jedem der angegebenen Verfahren in Echtzeit zer-

brochen werden. In Tabelle 6 sind zusätzlich zu den Werten für einen Bruch die für die Initialisierung und normale Simulation zu sehen. Die oberen 5 Objekte sind die auf Constraints basierenden Objekte, die unteren fünf die Tetraeder-Meshs für die anderen drei implementierten Verfahren.

	Initialisierung	Simulation	Bruch
Constraint-Methode:			
Cube w.h. (moderate)	40	16	16
Cube w.h. (fine)	390	90	90
Cuboid	2700	200	200
Voronoi-Cuboid	90	16	16
Voronoi-Cube	115	16	16
Radius-Methode:			
Cube w.h. (fine)	22	16	18
Fichera	24	16	20
Sphere	22	16	17
Cube	48	16	68
Cone	22	16	16

Tabelle 6: Benötigte Zeit für die Initialisierung, Bewegungssimulation und Bruch eines Objekts in Millisekunden (*ms*). Die eingetragenen Werte sind durch den BULLET *profileIterator* gemessene Durchschnittswerte. Die Bruchsimulation wurde mit einer selbst definierten Materialhaltbarkeit von $\tau = 0.5$ durchgeführt.

Drei der für die *Constraint-Methode* genutzten Elemente können ebenfalls in Echtzeit simuliert werden. Der angegebene Wert von *16ms* ist der Standard-Wert der BULLET Physik- und wird womöglich von der Bruchsimulation deutlich unterschritten. Die in der Tabelle gezeigten Werte weisen darauf hin, dass die Performanz der Objekte an ihre Element-Anzahl gebunden ist. Die Elemente *Cuboid* und *Cube w.h. (fine)* haben mit 513 bzw. 357 Elementen deutlich mehr als beispielsweise das *Voronoi-Cuboid-Objekt* mit 143 Elementen (Tabelle 2 auf Seite 62). Aufgrund ihrer Einfachheit und dieser Performanz-Vorteile wurden die *Voronoi-Objekte* in diese Ausarbeitung aufgenommen. Die beiden Voronoi-Objekte und der *Cube w.h. (moderate)* mit der Tetraeder-Struktur, aber deutlich weniger Elementen, konnten in Echtzeit durchgeführt werden. *Cuboid* und *Cube w.h. (fine)* erreichen in den Tests eine zu hohe Berechnungsdauer, und ihre Simulation wird in der Anwendung nicht mehr als flüssig wahrgenommen. In Tabelle ist des Weiteren zu sehen, dass der eigentliche Bruch des Objekts bei diesen Verfahren keine Performanz-Nachteile ergibt. Die angesprochenen Objekte erzielen sowohl vor, während und nach dem Bruch die selben gemessenen Werte. Der Grund hierfür ist der geringe Aufwand des eigentlichen Bruchs. Es müssen lediglich einige Constraints entfernt werden. Dieser Umstand hat auf die gemessenen

Werte keinen Einfluss.

Die abgebildeten Zeiten für die Initialisierung in Tabelle 6 zeigen, dass der Berechnungsaufwand der Objekte für die *Constraint-Methode* anfangs deutlich höher ist. Diese längere Berechnungsdauer ist durch die initiale Constraint-Berechnung gegeben. Die Soft-Bodys für die anderen Simulationen müssen lediglich in die Szene geladen werden und benötigen keine weitere Behandlung.

Der von BULLET intern festgelegte Zeitschritt von *16ms* war für die durchgeführten Simulationen völlig ausreichend. Im Hinblick auf Anwendungen, die beispielsweise sich extrem schnell bewegende Objekte behandeln, sollte dieser Zeitschritt oder die Kollisionsbehandlung angepasst werden. In den Ergebnissen dieser Anwendung ist die mögliche Problematik in Abbildung 37 auf Seite 74 angedeutet. Ein Geschoss, mit welchem das Objekt in der Luft getroffen wird, scheint in das Objekt einzudringen, bevor dieses zerbricht. Mit einem kleineren Zeitschritt könnte dieses Problem beseitigt werden.

Um die Belastung der entstandenen Bruchstücke auf die Simulation in dieser Anwendung zu überprüfen, wurde ein *Cube-Objekt* in 0 bis 600 Teile zerbrochen. Die in Abbildung 40 auf der nächsten Seite gezeigten Werte berücksichtigen hierbei nur die normale Simulation der Szene ohne die Bruchberechnungen. Es ist deutlich zu sehen, dass ab einer Anzahl von 400 bis 500 Elementen der Aufwand der Berechnung eine Simulation in Echtzeit nicht mehr zulässt. Die Werte überschreiten in dieser Region die für die Echtzeitfähigkeit vorausgesetzten *35ms/45ms*.

7.2.2 Stabilität

Im Rahmen der Entwicklung der Verfahren ist eine Anwendung entstanden, in der viele unterschiedliche Konfigurationen möglich sind. Es wurde eine hohe Variabilität aufgezeigt. In Simulationen, die eine Einflussnahme des Benutzers durch Parameterveränderungen ermöglicht, ist die Stabilität der Anwendung ein nicht zu vernachlässigender Aspekt. Die hohe Performanz einer Simulation ist irrelevant, wenn sie zu einer Instabilität der Anwendung führt. Unter der Berücksichtigung der in Abschnitt 7.1 auf Seite 60 aufgezeigten Problematiken ist die Anwendung sehr zuverlässig und stabil.

7.2.3 Bewertung

Die durchgeführten Messungen zeigten, dass die implementierten Verfahren in Abhängigkeit der genutzten Objekte den Anforderungen an die Performanz gerecht werden. Die erzeugten Bildraten sind für diese Objekte vollkommen echtzeitfähig. Es war zu beobachten, dass mit zunehmender Anzahl an Elementen die Bruchsimulation an ihre Grenzen geriet. Für eine in der Literatur als sehr komplex beschriebene Simulation sind die Ausführungszeiten

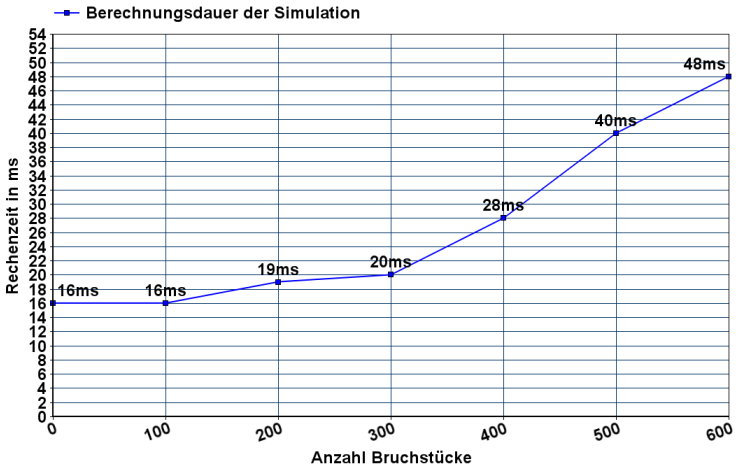


Abbildung 40: Berechnungsdauer der Simulation in Abhängigkeit von der Anzahl der Bruchstücke. Anzahl der Bruchstücke zwischen *0* und *600*. Dieser Test wurde mit dem *Cube-Objekt* durchgeführt.

bemerkenswert gering.

Der Vergleich der Performanz mit anderen Ergebnissen der Literatur ist auf Grund der unterschiedlichen Umstände -wie der Rechenkapazität- nicht ganz einfach. Sie geben dennoch einen sehr interessanten Einblick in die Entwicklung der Performanz der Bruchsimulationen.

Das vor 15 Jahren vorgestellte Verfahren von O'Brien und Hodgins[17] war zum damaligen Zeitpunkt das „state of the art"-Verfahren. Es brauchte für eine Sekunde Simulation 75 Minuten Berechnungszeit (Objekt mit 338 Knoten und 1109 Elementen). Dieser Zeitaufwand ist heutzutage eher mit der Bruchsimulation in einem Rendering-Programm wie BLENDER zu vergleichen und hat mit Echtzeitfähigkeit nichts zu tun.

Das in dieser Arbeit implementierte Verfahren von Smith et al.[21] zeigte in den Original-Ergebnissen, dass Modelle mit einer Anzahl von tausenden Constraints in wenigen Minuten berechnet werden konnten. In dieser Ausarbeitung wurde gezeigt, dass diese Simulation mittlerweile in weniger als einer Sekunde möglich ist.

Die Ausarbeitung von Müller et al. [14] hat in den Ergebnissen zu ihrem Verfahren bereits 2001 die Echtzeitfähigkeit einer Bruchsimulation vorgestellt. Die Berechnungen von Objekten mit 1000 bis 4000 Tetraedern wurden im Bereich der zweistelligen Millisekunden durchgeführt. Glondu et al. [7] zeigten 2012 in ihrem entwickelten Verfahren die echtzeitfähige Simulation von relativ komplexen Objekten. Die Berechnungszeit für ein Objekt mit etwa 5000 Knoten und 19000 Tetraedern wurde mit *16ms* angegeben. Durch

die stetig steigende Rechnerkapazitäten wird in Zukunft noch sehr viel mehr möglich sein.

Die möglichen Performanz-Optimierungen dieser Implementation und der Verfahren zur Bruchsimulation im Allgemeinen liegen vor allem in der Aufteilung der Berechnungen. Diese Aufteilung kann auf zwei verschiedene Art und Weisen passieren. Komplexe Berechnungen können in aufeinander folgende Zeitschritte aufgeteilt oder in „threads“ parallelisiert werden. Durch die aktuelle Verbreitung von Multi-Kern-Systemen ist diese Aufteilung sehr erfolgversprechend.

8 Fazit

Zu Anfang dieser Ausarbeitung wurden die in der Literatur bekannten Verfahren zur Bruchsimulation analysiert und auf ihren physikalischen Hintergrund überprüft. Auf dieser Basis wurden drei der Verfahren ausgewählt und implementiert. Ein eigenständig entwickeltes viertes Verfahren wurde zusätzlich implementiert. Aus diesen vier verschiedenen Ansätzen ist eine Anwendung entstanden, die einen Vergleich der vier Verfahren auf physikalischer und performanter Ebene ermöglicht. Die Ergebnisse sollten eine physikalisch plausible Simulation in Echtzeit darstellen.

Die Resultate der Untersuchung ergaben, dass sich die Verfahren in der Nutzung der physikalischen Eigenschaften zur Bruchinitiierung stark unterscheiden. Die Verfahren nach Glondu et al. , Müller et al. und das selbst entwickelte Verfahren orientierten sich sehr stark an die aus der Literatur der Kontinuumsmechanik bekannten Berechnungen. Die Verfahren führten zu sehr exakten und plausiblen Belastungen. Das Verfahren nach Smith et al. zeigte einen komplett anderen Ansatz. Die Objekte dieser Methode sind bereits in Bruchstücke aufgeteilt, welche durch eine gegebene Verbindungsstärke zusammengehalten werden.

In der eigentlichen Bruchsimulation finden sich diese Unterschiede ebenfalls wieder. Alle vier Verfahren nutzen physikalische Phänomene eines realen Bruchs für ihre Simulation, jedoch ergeben diese verschiedenen Techniken stark voneinander abweichende Ergebnisse. Während die auf der Kontinuumsmechanik basierenden Brüche physikalisch plausible Brüche als Resultate vorweisen konnten, waren die Brüche im vierten Ansatz sehr statisch und vorhersehbar. Die erstgenannten Verfahren konnten der Anforderung, ein wirklichkeitsgetreues Abbild der Natur zu entwerfen, auf physikalischer Ebene vollkommen entsprechen.

Die Resultate der Performanz-Analyse ergaben, dass alle vier Verfahren eine Bruchsimulation in Echtzeit durchführen können. Da auch Bruchsimulationen mit mehreren Iterationen in Echtzeit möglich sind, können in Kombination mit den Materialparametern sehr unterschiedliche Brüche erzeugt werden. Bei übermäßig vielen Bruchiterationen und großen Meshs stieß die

entwickelte Anwendung an ihre Grenzen.

Die Problematik des Performanz-Verlustes bei sehr vielen der von BULLET bereitgestellten Rigid-Bodys liegt daran, dass diese ursprünglich nicht für die Verwendung in einer solch großen Anzahl entwickelt worden sind. Die Klasse *btRigidBody* ist eigentlich zu mächtig, um lediglich als Teil eines größeren Objekts zu fungieren. Eine andere Datenstruktur für die *Constraint-Methode* bietet sich daher für Implementationen des von Smith et al. vorgestellten Verfahren an. Die Nutzung der *btSoftBody*-Klasse war für die durchgeführten Implementationen der drei anderen Verfahren eine gute Wahl. Die Struktur der Soft-Body-Objekte war als Basis für die Belastungsberechnungen aufgrund der Verformbarkeit der Elemente sehr hilfreich. Insgesamt war die BULLET Physik- eine gute Basis zur Implementation der gestellten Aufgabe. Jedoch wurde im Laufe der Umsetzung der unterschiedlichen Verfahren klar, dass sehr viele bestehende Klassen für beispielsweise Kollisionen und Constraints angepasst und mathematische Strukturen verändert werden müssen um das gewünschte Ziel zu erreichen.

Die entwickelte Anwendung stellt bereits die Möglichkeit der Bruchsimulation verschiedener Objekte in Echtzeit zur Verfügung. Aufbauend auf den Erkenntnissen der Implementierung und deren Resultaten sind einige Optimierungen und Erweiterungen denkbar. Die duktile Verformung vor einem Bruch könnte zum Beispiel intensiver behandelt werden. Eine andere Erweiterung wäre die Behandlung der entstandenen Bruchstücke als potentiell weiterhin brechende Elemente. Sehr interessant wäre auch zu sehen, wie die Simulation zweier kollidierender Elemente aussehen würde, wenn beide zerbrechen. Die erste Optimierung die ich vornehmen würde, wäre allerdings die Verbesserung der Bruchkanten durch in der Literatur vorgestellte Verfahren und die Darstellung des Bruchs mit einer echten Rendering-Software. Obwohl die Ergebnisse der implementierten Verfahren durchaus ansehnlich sind, könnte das Resultat durch die Vermeidung von spitz zulaufenden Tetraedern noch einmal deutlich verbessert werden.

Es gibt des Weiteren in der Literatur verschiedene Aspekte die den hier vorgestellten Ansatz erweitern und verbessern würden. Am wichtigsten wäre dabei die effizientere Programmierung der Bruch-Berechnungen. Die Optimierung für Multi-Kern-Systeme und parallele Threads sind hierbei die entscheidenden Ansätze. Denkbar wäre auch die Umstellung der Verfahren auf eine echte „Finite Elemente Methode“, um dem realen atomaren Aufbau eines Objekts näher zu kommen.

Der Trend, der sich auf der jüngsten SIGGRAPH-Konferenz wiederspiegelt, ist jedoch ein ganz anderer. Wang et al. stellten in ihrem Ansatz „Physics-Inspired Adaptive Fracture Refinement“ eine Methode vor, die niedrig aufgelöste Bruchstücke zu hoch aufgelösten umrechnet. Eine andere Idee zeigen O‘Brien et al. im Ansatz „Adaptive Tearing and Cracking of Thin Sheets“. Dieses Verfahren ermöglicht eine Umstrukturierung des zu Grunde liegenden Meshs genau an den Stellen wo es benötigt wird. Es basiert auf einer sich

selbst anpassenden Struktur.

Die Simulation von Brüchen erwies sich als sehr vielschichtiges Thema und gab überaus interessante Einblicke in die Funktionalität einer physikalischen Simulation. Ich fand es faszinierend, wie physikalische Gegebenheiten mit Mitteln der Computergraphik umgesetzt werden können. Nach Optimierung der Schwachstellen durch die aufgezeigten Methoden kann ich mir vorstellen, die entwickelte Implementation als Patch für die BULLET Physik-Engine anzubieten.

Literatur

[1] Franz Aurenhammer. Voronoi diagrams, a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, September 1991.

[2] Zhaosheng Bao, Jeong-Mo Hong, Joseph Teran, and Ronald Fedkiw. Fracturing rigid materials. *IEEE Trans. Vis. Comput. Graph.*, 13(2):370–378, 2007.

[3] Daniel Bartsch. Bv-algorithmen auf modernen grafikkarten parallelisierte erzeugung von voronoi-diagrammen mit hilfe des cuda-frameworks. Master's thesis, FH SWF Iserlohn, 2009.

[4] Jan Bender. *Dynamiksimulation in der Computergraphik*. Habilitation, Karlsruhe Institute of Technology (KIT), Germany, January 2014.

[5] Erwin Coumans. *Bullet 2.80 Physics SDK Manual*, 2012.

[6] Loeïz Glondu. *Physically-based and Real-time Simulation of Brittle Fracture for Interactive Applications*. These, École normale supérieure de Cachan - ENS Cachan, November 2012.

[7] Loeïz Glondu, Maud Marchal, and Georges Dumont. Real-time simulation of brittle fracture using modal analysis. *IEEE Trans. Vis. Comput. Graph.*, 19(2):201–209, 2013.

[8] David Körner. Computergraphische simulation dynamischer bruchbildung in starr-plastischen gemengen. Master's thesis, TECHNISCHE UNIVERSITAET DRESDEN, 2008.

[9] Robert Koschig. Das optimierungsverfahren mit lagrange-multiplikatoren. Technical report, www.massmatics.de, 2012.

[10] Ayse Kücükyilmaz and Bülent Özgüc. An animation system for fracturing of rigid objects. In *ISCIS*, volume 3733 of *Lecture Notes in Computer Science*, pages 688–697. Springer, 2005.

[11] Hugo Ledoux. Computing the 3d voronoi diagram robustly: An easy explanation. In *ISVD*, pages 117–129. IEEE Computer Society, 2007.

[12] LevelCapGaming. Bf4 levolution guide: How to trigger destruction events! (battlefield 4 launch gameplay/commentary), 2013.

[13] Ian Millington. *Game physics engine development*. The Morgan Kaufmann series in interactive 3D technology. Morgan Kaufmann Publishers, San Francisco, CA, 2007. Cd-rom d'accompagnement contenant le code source.

[14] Matthias Müller, Leonard McMillan, Julie Dorsey, and Robert Jagnow. Real-time simulation of deformation and fracture of stiff materials. In *Proceedings of the Eurographic Workshop on Computer Animation and Simulation*, pages 113–124, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[15] Alan Norton, Greg Turk, Bob Bacon, John Gerth, and Paula Sweeney. Animation of fracture by physical modeling. *Vis. Comput.*, 7(4):210–219, July 1991.

[16] James F. O'Brien. *Graphical Modeling and Animation of Fracture.* PhD thesis, Atlanta, GA, USA, 2000. AAI9978415.

[17] James F. O'Brien and Jessica K. Hodgins. Graphical modeling and animation of brittle fracture. In *SIGGRAPH*, pages 137–146, 1999.

[18] James F. O'Brien and Jessica K. Hodgins. Animating fracture. *Commun. ACM*, 43(7):68–75, 2000.

[19] Eric G. Parker and James F. O'Brien. Real-time deformation and fracture in a game environment. In Dieter W. Fellner and Stephen N. Spencer, editors, *Symposium on Computer Animation*, pages 165–175. ACM, 2009.

[20] Andreas Schneider. Eigenwerte und eigenvektoren.

[21] Jeffrey Smith, Andrew P. Witkin, and David Baraff. Fast and controllable simulation of the shattering of brittle objects. *Comput. Graph. Forum*, 20(2):81–90, 2001.

[22] Demetri Terzopoulos and Kurt Fleischer. Modeling inelastic deformation: Viscolelasticity, plasticity, fracture. *SIGGRAPH Comput. Graph.*, 22(4):269–278, June 1988.

[23] Battlefield Wiki. Destruction-feature, 2011-2013.

[24] Wikipedia. Duktilität, 2014.

[25] Andrew Witkin. Particle system dynamics. In *In ACM SIGGRAPH'92 Courses*, pages 1–12. ACM Press, 1992.

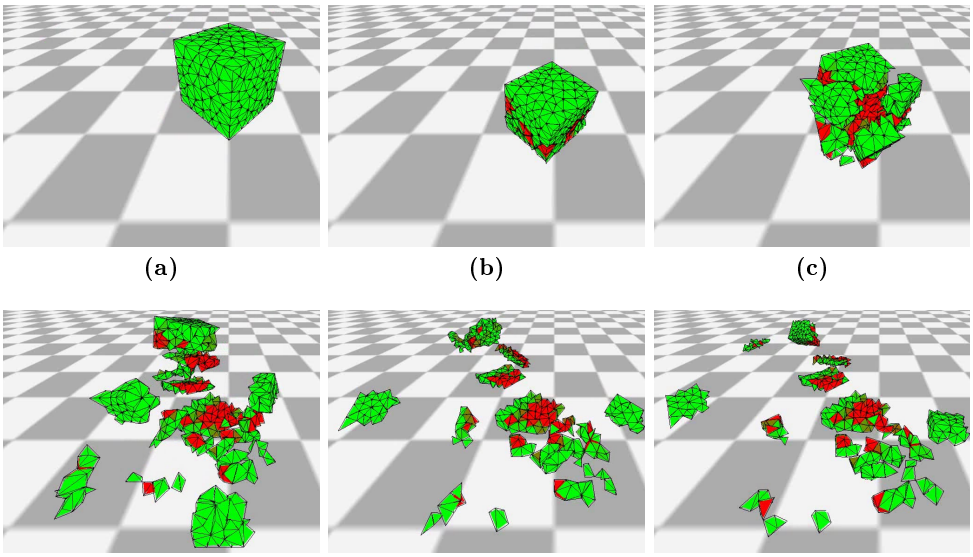
Abbildungsverzeichnis

Anhang

Abbildung 41: Zerstörung in Battlefield Bad Company [23]

Abbildung 42: Zerstörung in Battlefield Bad Company 2 [23]

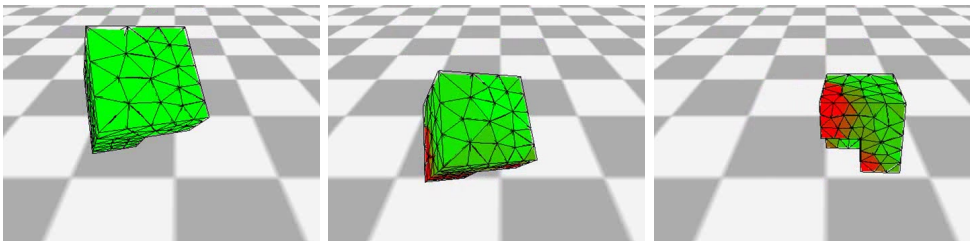
Abbildung 43: Zerstörung in Battlefield 3 [23]



(a) **(b)** **(c)**

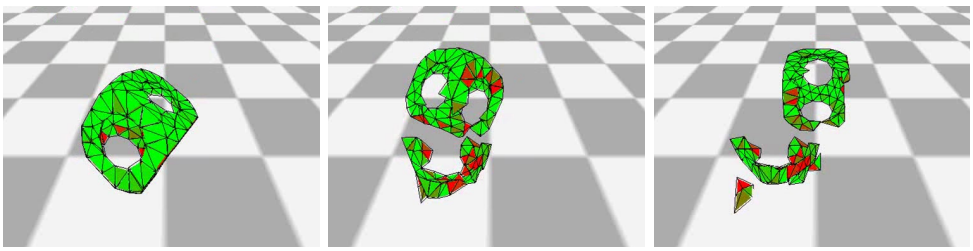
(d) **(e)** **(f)**

Abbildung 44: *Belastungsansicht*: Bruch eines *Cube-Objekts*. Simulation: *secondary-Splits,* Material: *Keramik* mit $\tau = 2.48$, max. Bruchiterationen: *4*



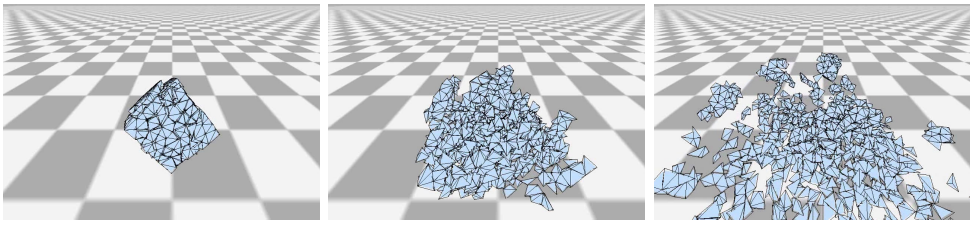
(a) **(b)** **(c)**

Abbildung 45: *Belastungsansicht*: *Fichera-Objekt* bricht nicht. Simulation: *secondary-Splits,* Material: *Eisen* mit $\tau = 24.820$



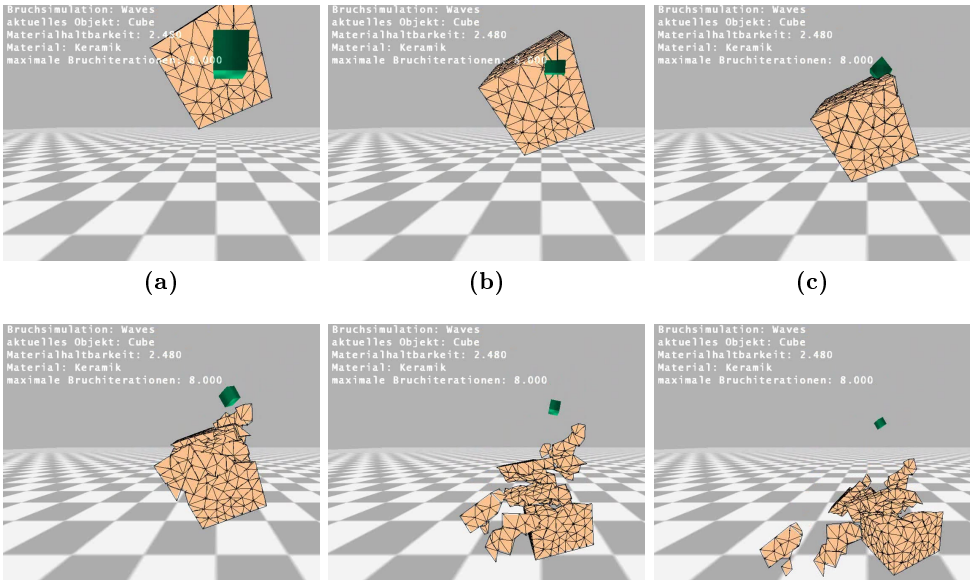
(a) **(b)** **(c)**

Abbildung 46: *Belastungsansicht*: Bruch eines *Cube with Holes-Objekts*. Simulation: *Waves,* Material: *Eisen* mit $\tau = 24.82$, max. Bruchiterationen: *3*



(a) **(b)** **(c)**

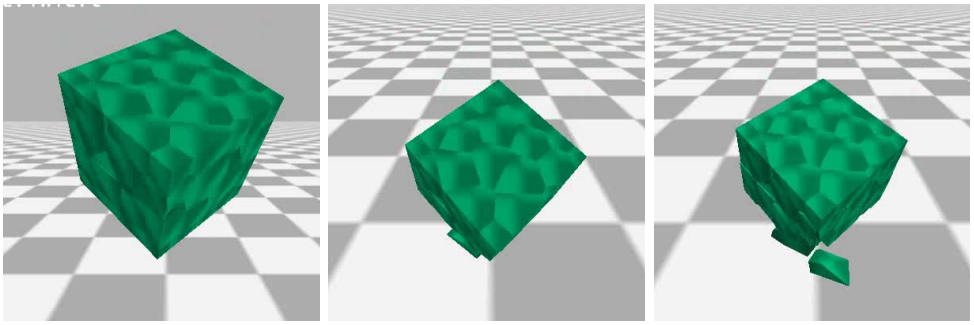
Abbildung 47: Bruch eines *Cube-Objekts* mit extrem hoher Bruchanzahl. Simulation: *secondary-Splits,* Material: *Glas* mit $\tau = 6.01$, max. Bruchiterationen: *150*



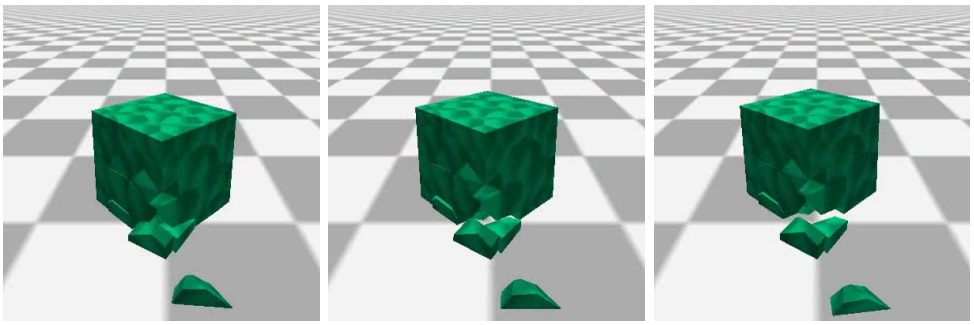
(a) **(b)** **(c)**

(d) **(e)** **(f)**

Abbildung 48: Bruch eines *Cube-Objekts* durch ein Geschoss. Simulation: *Waves,* Material: *Keramik* mit $\tau = 2.48$, max. Bruchiterationen: *8*

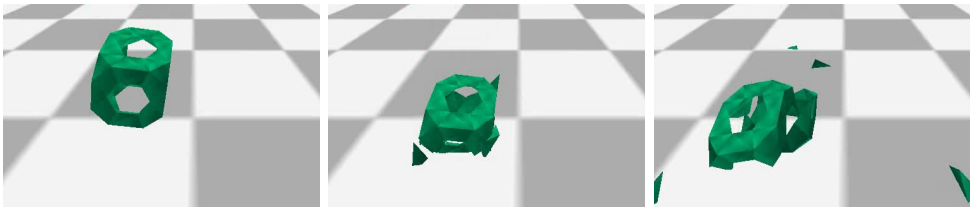


(a) (b) (c)



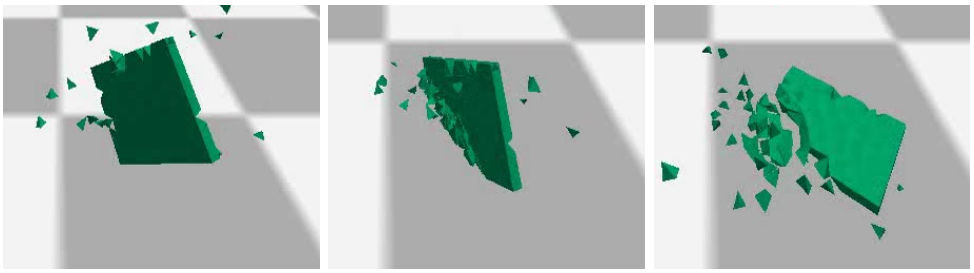
(d) (e) (f)

Abbildung 49: Bruch eines *Voronoi-Cube-Objekts* mit hoher Materialhaltbarkeit. Simulation: *Constraints,* Material: *selbst definiert* mit $\tau = 2.3$



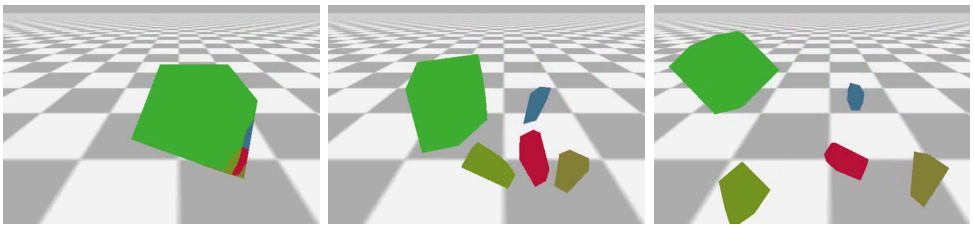
(a) (b) (c)

Abbildung 50: Bruch eines *Cube with Holes-Objekts* mit moderater Auflösung (99 Elemente). Simulation: *Constraints,* Material: *selbst definiert* mit $\tau = 1.8$



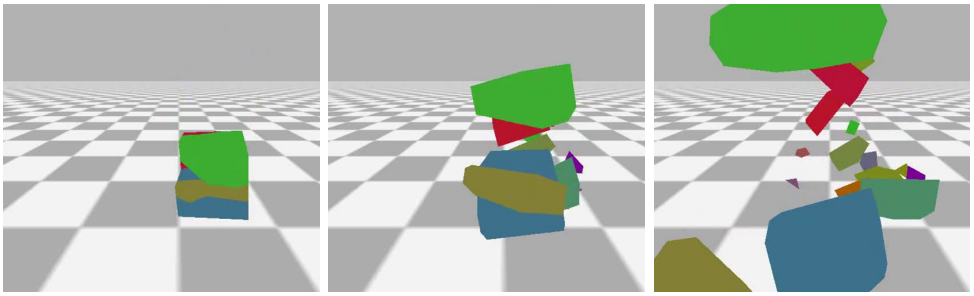
(a) (b) (c)

Abbildung 51: Bruch eines *Cuboid-Objekts* mit Artefakten. Simulation: *Constraints,* Material: *selbst definiert* mit $\tau = 1.4$



(a) (b) (c)

Abbildung 52: *Clustersansicht*: Bruch eines *Fichera-Objekts* mit 5 entstandenen Bruchstücken.



(a) (b) (c)

Abbildung 53: *Clustersansicht*: Bruch eines *Fichera-Objekts* mit 17 entstandenen Bruchstücken.