

Implementierung der Ductile-Fracture-Bruchsimulation in der Unity Engine

Implementation of Ductile Fracture within the Unity Game Engine

Paul Froelich

Bachelor-Abschlussarbeit

Betreuer: Prof. Dr. Christoph Lürig

Trier, 16.09.2020

Vorwort

Kurzfassung

Die Simulation von Deformierungen und Frakturen von Körpern finden weitflächig Anwendung, beispielsweise in Analyseprogrammen zur Prognose von Bruchstellen bei Bestandteilen von Motoren oder in 3D-Animationsfilmen. Doch durch den zunehmenden Realismus im Bereich der Videospiele steigt auch das Interesse an realistischen Zerstörungsanimationen und der Anwendung von realistischen Zerstörungen als zentrale Spielmechanik.

Das Ziel dieser Bachelor-Abschlussarbeit ist es, einen einfachen Algorithmus zur Berechnung von spröder und duktiler Frakturenpropagierung aus dem Jahre 1998 in der Spieleengine *Unity* zu implementieren und zu beurteilen, wie realistisch die Verwendung eines solchen Algorithmus in einem Videospiel ist. Hierbei wird besonders die Performanz des Algorithmus betrachtet. Die Arbeit richtet sich demnach vor allem an Entwickler von Softwaresystemen mit weichen Echtzeitanforderungen (zum Beispiel Videospiele).

Abstract

The same in English (option).

Inhaltsverzeichnis

1 Einleitung und Problemstellung.....	1
1.1 Motivation.....	1
1.2 Historischer Überblick.....	2
1.2.1 Deformierungsmodelle.....	2
1.3 Motivation zur Entwicklung des Algorithmus.....	2
1.4 Aufbau des Verfahrens.....	3
2 Der theoretische Hintergrund.....	3
2.1 Das kontinuierliche Modell.....	3
2.1.1 Die Begriffe “Kontinuum”, “spröde”, “duktil”, “plastisch” und “elastisch”..	4
2.1.2 Das Maß der Belastung.....	5
2.1.3 Duktile Fraktur.....	8
2.2 Das diskrete Modell.....	9
2.3 Die Kraftdekomposition.....	13
2.4 Die Frakturerkennung.....	14
2.5 Das Remeshing.....	14
2.6 Ein Kerninteraktionsschleifendurchlauf.....	18
3 Die Implementierung in Unity.....	20
3.1 Die Planung.....	20
3.1.1 Die Architektur.....	21
3.1.2 Das Testsystem.....	21
3.2 Ablauf der Implementierung.....	22
3.2.1 Das Node-Prefab.....	22
3.2.2 Das Editorskript FractureTool.....	22
3.2.3 Die Klasse PositionChangeListener und die Schnittstelle IPositionChange-	
Listener.....	23
3.2.4 Die Klasse TetrahedronBuilder.....	23
3.2.5 Die Klasse RelationManager.....	23
3.2.6 Die Klasse FractureSimulator.....	24
3.2.7 Die Klasse Tetrahedron.....	24
3.2.8 Die Klasse Node.....	26
3.2.9 Die Klasse MathUtility.....	27
3.3 Mögliche Erweiterungen.....	27
4 Diskussion der Ergebnisse.....	27
4.1 Eine grobe Komplexitätsanalyse.....	28
4.2 Experimentelle Daten und Resultate.....	29
4.3 Veränderungsvorschläge.....	29
5 Ausblick.....	29

Abbildungsverzeichnis

Duktile Brüche (links) und spröde Brüche (rechts) unter verschiedenen Belastungsarten (ForMatEng03).	5
fünf Tetraeder, die einen Würfel bilden.....	10
Inkonsistentes Netz (links) & konsistentes Netz (rechts) aus zwei Tetraedern.....	11
Abbildung: Fallbeispiele für die Schnittmenge der Frakturfläche und eines Tetraeders und die jeweils resultierenden neuen Tetraeder.....	15
Abbildung: Remeshing eines Nachbarn beim 2-Schnittpunkt-Fall. Oben bei 1 gemeinsamen geschnittenen Kante, Unten bei 2 gemeinsamen geschnittenen Kanten.....	17
Abbildung: Ablauf eines Frames.....	19

Tabellenverzeichnis

1 Einleitung und Problemstellung

In dieser Bachelor-Abschlussarbeit geht es um die Implementierung eines Algorithmus zur Berechnung von Frakturen in Körpern aus sprödem oder duktilem Material in der Spieleengine *Unity*. Der erste Teil des Algorithmus wurde im Jahre 1999 von den Autoren James F. O'Brien und Jessica K. Hodgins veröffentlicht und behandelte die Frakturpropagation spröder Materialien. Der zweite Teil wurde im Jahre 2001 von denselben Autoren und Adam W. Bargteil präsentiert und ergänzte den Algorithmus um duktile Materialien.

Ziel der Implementierung ist die Analyse des Algorithmus für seine Tauglichkeit in Videospielen. Hierbei wird besonders auf die Performanz der Implementierung des Algorithmus geachtet. Im Zuge dessen werden Verbesserungen des Algorithmus für den Einsatz in Videospielen vorgeschlagen. Ferner wird der Algorithmus in einen historischen Kontext gesetzt und es werden andere und neuere Algorithmen betrachtet und mit dem Algorithmus verglichen.

1.1 Motivation

Die Deformierung und Zerstörung von Objekten in Videospielen als eine zentrale Gameplay-Mechanik ist eine junge und kaum erschlossene Disziplin (*is it though?*). Doch die algorithmischen Möglichkeiten zur Berechnung von Frakturen existieren schon seit 1988 und die wissenschaftlichen Fachgebiete, welche die mathematische Grundlage dieser Algorithmen liefern, reichen bis ins Jahr 1958 zurück.

Diese Algorithmen waren vor allem für die Animation von Körpern in 3D-animierten Filmen zuständig. Sie waren also nicht dafür bestimmt, in einem Echtzeitsystem ausgeführt zu werden. Dies wird besonders klar bei der Betrachtung der Performanzangaben des Algorithmus: die Simulation von einer Sekunde einer zerbrechenden Glasscheibe dauerte auf einem SGI O2 (eine Low-End Unix-Workstation von 1996) mit einem 195MHz-Prozessor im Durchschnitt 273 Minuten (BritFrac99).

An dieser Stelle sind die Algorithmen definitiv von Softwaresystemen mit hohen Sicherheitsanforderungen (beispielsweise Frakturprognosen von Materialien in Motoren) abzugrenzen, denn die Ergebnisse der Simulationen sind nicht auf physische Genauigkeit ausgelegt, sondern darauf, zufriedenstellende Ergebnisse für Unterhaltungsfilme zu liefern.

Weiterhin wird die Implementierung des Algorithmus in der Unity-Spieleengine beschrieben, die im Rahmen dieser Abschlussarbeit durchgeführt wurde. Hierbei betrachten wir die Planung, die Durchführung und die möglichen Erweiterungen und Verbesserungen der Implementierung. Abschließend werden die Ergebnisse des Experiments diskutiert und ein Ausblick auf die Zukunft der Simulation von Deformierung und Fraktur von Körpern, besonders im Rahmen der Unterhaltungsindustrie, geliefert.

1.2 Historischer Überblick

Physikbasierte Modelle für computergenerierte Grafiken und Animationen gewannen Mitte der 1980er Jahre an Popularität aufgrund ihrer Vorteile gegenüber kinematischer Animation (DefModels88).

Konventionelle Animation ist kinematisch; Sie wird beschrieben durch die Bewegung ihrer geometrischen Einzelteile zu jedem Zeitschritt, möglicherweise ergänzt durch Interpolationen, um einen weicheren Übergang zu ermöglichen. Physikbasierte, dynamische Animation funktioniert durch Kraftanwendungen auf Körper, sodass die resultierende Position des Körpers aus numerischen Verfahren gewonnen werden kann (DefModels88).

Die Vorteile der dynamischen Animation gegenüber der kinematischen Animation bestehen darin, dass sie realitätsnäher ist und es dadurch für den Entwickler der Animationen einfacher ist, realistische und damit zufriedenstellende Animationen umzusetzen. Aufgrund dieser physikbasierten Modelle wurden daraufhin Techniken entwickelt, welche auch die Animation von nicht-starren Körpern ermöglichten (DefModels88).

1.2.1 Deformierungsmodelle

1988 publizierten Demetri Terzopoulos und Kurt Fleischer die wissenschaftlichen Arbeiten *Deformable Models* und *Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture*, welche die Funktionsweise eines Modells zur Deformierung von elastischen, viskoelastischen und plastischen Körpern aufzeigen. Das Konzept des Modells folgt, genauso wie der in dieser Arbeit implementierte Algorithmus, der Unterteilung des Systems in ein kontinuierliches und ein diskretes Modell. Das kontinuierliche Modell basiert auf Energiefunktionen – Funktionen, die den Status eines Körpers als Eingabe bekommen und die resultierende Energie der Deformierung als Ausgabe formulieren. Die Diskretisierung des kontinuierlichen Modells geschieht über die Finite-Differenzen-Methode (kurz FDM). Dieser Algorithmus lieferte vielversprechende Ergebnisse und eine Basis für weitere Ansätze.

Alan Norton und seine Kolleginnen und Kollegen Greg Turk, Bob Bacon, John Gerth und Paula Sweeney lieferten 1991 eine Technik zur Simulation solider Objekte, welche unter größeren Belastungen zerbrechen. Sie realisierten dies durch ein sog. Spring-Mass-System (dt. Feder-Masse-System). Hierbei wurden stets die Distanzen zwischen zwei Massepunkten des zu simulierenden Objektes gemessen, wobei die Verbindung zwischen eben jenen Massepunkten gekappt wurde, sobald die Distanz einen Grenzwert überschritt. Um ungewollt zusammenhängende Einzelbruchteile zu vermeiden, wurden benachbarte Massepunktverbindungen ebenfalls gelöst.

1.3 Motivation zur Entwicklung des Algorithmus

Die bereits beschriebenen Deformierungsmodelle lieferten eine sinnvolle Basis für die Weiterentwicklung von Modellen, welche verschiedenste Eigenschaften von Materialien berücksichtigen konnten. Allerdings waren die genannten Modelle dadurch limitiert, dass sie keinerlei Informationen liefern konnten, wo die sog. Frakturfläche einer Fraktur lag oder in welche Richtung ihre Normale zeigte. Des Weiteren stellten die Modelle keine Remeshing-Vorschriften

ten der gebrochenen Körper zur Verfügung. Die zerbrechenden Körper konnten also nur entlang der Grenzen ihrer Bestandteile zerbrechen, welche selbst unzerbrechlich waren.

Der GMABF-Algorithmus löst dies durch ein Finite-Elemente-Modell, Differenzgleichungen zur Beschreibung des zusammenhängenden Verhaltens und einer Remeshing-Vorschrift, die bei einem Bruch das zugrundeliegende Gitter kleiner aufteilt.

1.4 Aufbau des Verfahrens

Der GMABF-Algorithmus basiert auf der Finite-Elemente-Methode (kurz FEM) und der Kontinuumsmechanik. Die **FEM** ist eine bekannte und bewährte Methode zur Diskretisierung von Volumen, die darauf basiert, das zu diskretisierende Volumen in geometrische Elemente, in diesem Fall Tetraeder, zu unterteilen. Die **Kontinuumsmechanik** ist ein Fachgebiet aus der Mechanik und beschäftigt sich mit der Bewegung von deformierbaren Körpern aufgrund von äußeren Belastungen. Sie betrachtet das Material eines Volumens nicht als eine Menge von Partikeln / Teilchen / Molekülen / Atomen, sondern als eine kontinuierliche Menge, die dann mit der FEM diskretisiert wird. In den folgenden Kapiteln werden die beiden Bestandteile gründlicher beleuchtet.

2 Der theoretische Hintergrund

Das der Arbeit zugrundeliegende GMABF-Modell besteht aus 4 Teilen: dem kontinuierlichen Modell, welches auf der Kontinuumsmechanik basiert, dem diskreten Modell, welches die Finite-Elemente-Methode verwendet, der Kraftdekomposition / Frakturerkennung und zuletzt der Remeshing-Vorschrift.

Zuerst wird der Algorithmus zur Berechnung von Frakturpropagation in spröden Materialien beleuchtet und erklärt. Im Rahmen dessen werden das kontinuierliche und das diskrete Modell grundlegend erklärt und die Remeshing-Vorschrift vollständig erläutert.

Darauf folgt die Erklärung der Erweiterung des Algorithmus um duktile Materialien, welches ausschließlich das kontinuierliche Modell des vorher beschriebenen Algorithmus für spröde Materialien erweitert und das diskrete Modell, sowie die Remeshing-Vorschrift nicht beeinflusst.

Zuletzt wird die Remeshing-Vorschrift erläutert.

2.1 Das kontinuierliche Modell

Das Konzept des kontinuierlichen Modells ist es, an jedem räumlichen Punkt eines Körpers Daten über die Belastung an dieser Stelle zu liefern. Genauer bedeutet dies, dass die internen Belastungen des Körpers in Form von tensilen (umgangssprachlich äquivalent zu „spannen-

den“ oder „ziehenden“) und kompressiven Kräften durch das Modell zur Verfügung stehen. Man kann so an jedem Punkt innerhalb des Körpervolumens die vorliegenden kompressiven und tensilen Kräfte bestimmen.

Die folgenden Abschnitte stellen die Informationen bereit, die für die Umsetzung eines solchen Modells benötigt werden. Zuerst wird ein grundlegendes Verständnis für die Begriffe *Kontinuum*, *duktil*, *spröde*, *plastisch* und *elastisch* geschaffen, daraufhin wird das Maß für Belastung (engl. „strain“) und Belastungsänderung (engl. „strain rate“) definiert und erläutert.

Des Weiteren wird der Bezug zu den physikalischen Eigenschaften eines Materials geschaffen und in das Modell eingepflegt, sodass ein Maß für die gesamten inneren Kräfte definiert wird. Zuletzt wird dann die Erweiterung um duktile Materialien beschrieben.

2.1.1 Die Begriffe „Kontinuum“, „spröde“, „duktil“, „plastisch“ und „elastisch“

Ein Kontinuum ist beispielsweise die Menge der reellen Zahlen; Es gibt unendlich viele von ihnen und zwischen zwei verschiedenen reellen Zahlen existiert immer eine andere reelle Zahl. Bei Betrachtung des mathematischen Konzepts eines dreidimensionalen Raumes, repräsentiert durch ein kartesisches Koordinatensystem, so ist ein solcher Raum auch ein Kontinuum, denn zwischen zwei verschiedenen Orten in diesem Raum gibt es immer einen weiteren Ort. Die Definition eines Kontinuums kann auf das Material eines Volumens erweitert werden. Um es mathematisch genau zu definieren (ContMech94):

Es existiere ein Volumen V_0 . Wenn die Dichte, der Schwung, die Energie und die Belastung an jedem Ort im Volumen V_0 definiert ist und wenn jede dieser Eigenschaften durch eine kontinuierliche Funktion mit räumlichen Koordinaten in V_0 definiert werden kann, dann ist das Material im Volumen V_0 kontinuierlich (ContMech94).

Für die folgenden Erklärungen reicht es aus, sich einen geometrischen Körper mit einem kontinuierlichen Material so vorzustellen, dass an jedem Punkt innerhalb des Körpervolumens die Belastung des Körpers messbar ist.

Die Notwendigkeit, die folgenden Begriffe in ihrem physischen Kontext erneut zu erläutern, liegt an ihrer Diskrepanz zur umgangssprachlichen Nutzung. Um ihre Bedeutung zu verstehen, muss zuerst verstanden werden, wie Deformierungen und Brüche auf einer grundlegenden physikalischen Ebene funktionieren.

Ein Material verformt sich unter Belastung, bis es bricht. Diese Verformung ist erst elastischer Natur und hat dann eine plastische Phase. Wenn die Belastung des Materials in der elastischen Verformungsphase aufhört, dann kehrt das Material in seine Ursprungsform zurück, als wäre die Verformung nicht aufgetreten. Bricht die Belastung in der plastischen Phase der Verformung ab, dann kommt es allerdings zu bleibenden, irreversiblen Verformungen des Materials. Der Übergang zwischen elastischer und plastischer Phase wird Elastizitätsgrenze oder Streckgrenze genannt (Werkstoff11).

Ein sprödes Material ist ein Material, was sehr direkt nach der Elastizitätsgrenze bricht, also keine oder nur kaum plastische Verformung aufweist. Ein duktile Material dagegen zeigt eine deutliche plastische Verformung, bevor es bricht.

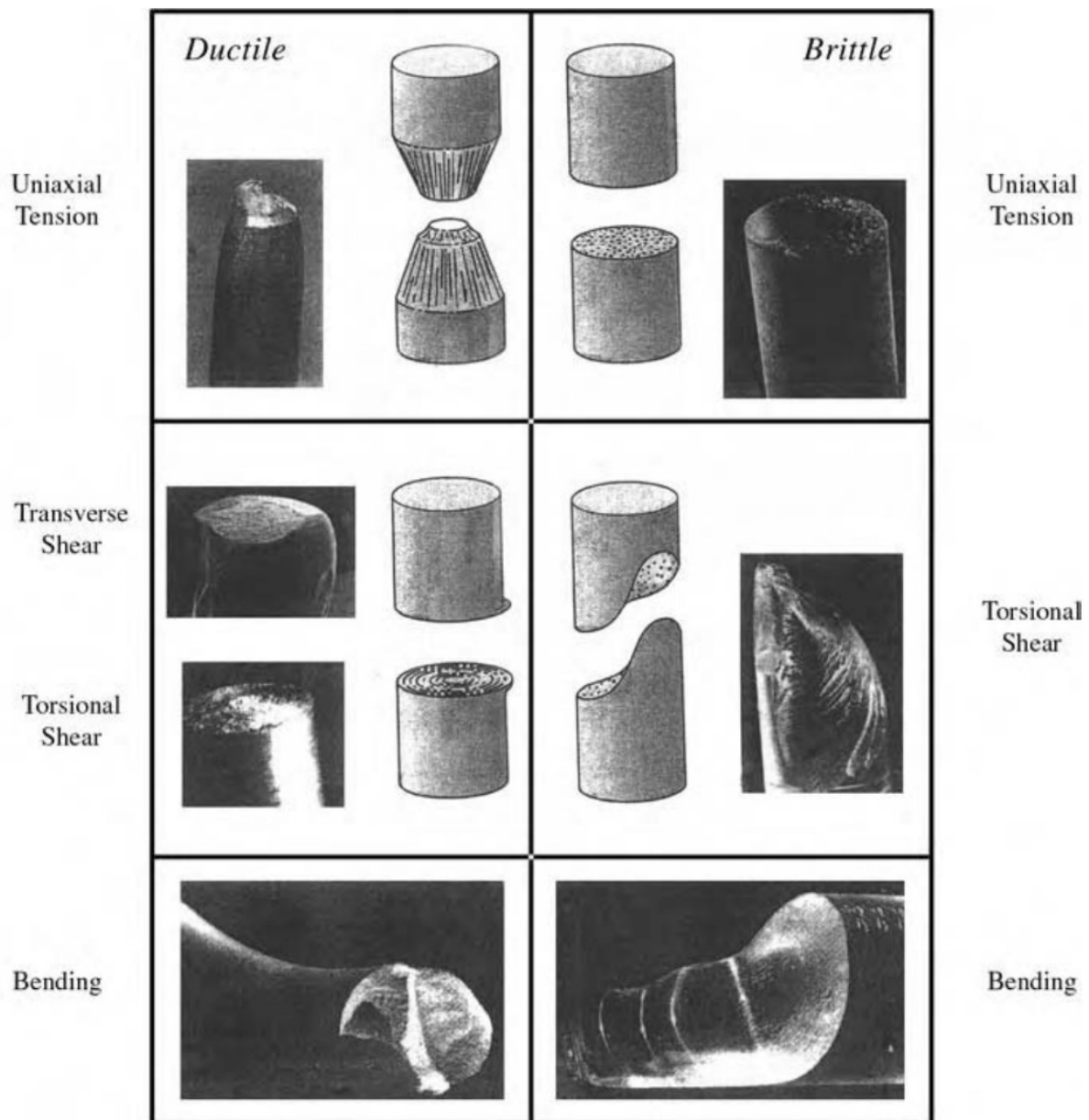


Abbildung 1: Duktile Brüche (links) und spröde Brüche (rechts) unter verschiedenen Belastungsarten (ForMatEng03)

2.1.2 Das Maß der Belastung

Bevor Belastung gemessen werden kann, müssen wir festlegen, welche Daten wir von dem zu messenden Objekt lesen können. Wir können die Belastung eines Objektes an einem bestimmten Punkt feststellen, wenn wir die **globale Position**, die **Materialposition** und die **globale Geschwindigkeit** des Punktes kennen.

- Die globale Position ist durch einen dreidimensionalen Ortsvektor beschrieben, der im globalen Koordinatensystem auf den Punkt zeigt.

- Die Materialposition ist ein dreidimensionaler Ortsvektor, welcher allerdings vom Ursprung des sog. Model-Koordinatensystems auf den Punkt zeigt, *nicht* vom globalen Koordinatensystem. Das Model-Koordinatensystem ist ein lokales Koordinatensystem, repräsentiert durch die Position eines Ursprungs (im Folgenden gekennzeichnet mit dem Vektor \vec{u} , der vom globalen Koordinatenursprung zum Ursprung des Model-Koordinatensystems zeigt) und die Rotation der Achsen eben jenes Koordinatensystems. In der Regel existiert ein Model-Koordinatensystem pro Objekt in der Simulation. Es dient dazu, die Position und die Rotation des Objektes zu beschreiben.
- Die globale Geschwindigkeit des Punktes ist die Änderung der globalen Position des Punktes relativ zur Zeit.

Definiert sei eine Funktion $x(\vec{a}) = \vec{a} + \vec{u}$, wobei \vec{u} die Position des zu messenden Objekts im globalen Koordinatensystem bezeichnet und \vec{a} ein beliebiger Vektor in Materialkoordinaten des zu messenden Objektes ist. \vec{u} bezeichnet also den Ortsvektor des Model-Koordinatensystems im globalen Koordinatensystem. Damit ist $x(\vec{a})$ eine Funktion, die von Materialkoordinaten in globale Koordinaten umrechnet.

Um nun die Belastung des Objektes an einem bestimmten Punkt \vec{a} zu berechnen, benötigen wir den Greenschen Verzerrungstensor, der im Folgenden mit ϵ bezeichnet wird. ϵ ist eine dreidimensionale, quadratische und symmetrische Matrix. In Tensornotation wird er folgendermaßen beschrieben:

$$\epsilon_{ij} = \left(\frac{\partial x}{\partial a_i} \cdot \frac{\partial x}{\partial a_j} \right) - \delta_{ij}$$

Wobei δ_{ij} das Kronecker-Delta bezeichnet. Es ist definiert als

$$\delta_{ij} = \begin{cases} 1: & i = j \\ 0: & i \neq j \end{cases}$$

Die Tensornotation liefert eine Beschreibung der Matrix-Komponente in der Zeile i und der Spalte j . In Matrixnotation sähe der Verzerrungstensor dann also folgendermaßen aus:

$$\begin{bmatrix} \left(\frac{\partial x}{\partial a_1} \cdot \frac{\partial x}{\partial a_1} \right) - 1 & \left(\frac{\partial x}{\partial a_1} \cdot \frac{\partial x}{\partial a_2} \right) & \left(\frac{\partial x}{\partial a_1} \cdot \frac{\partial x}{\partial a_3} \right) \\ \left(\frac{\partial x}{\partial a_1} \cdot \frac{\partial x}{\partial a_2} \right) & \left(\frac{\partial x}{\partial a_2} \cdot \frac{\partial x}{\partial a_2} \right) - 1 & \left(\frac{\partial x}{\partial a_2} \cdot \frac{\partial x}{\partial a_3} \right) \\ \left(\frac{\partial x}{\partial a_1} \cdot \frac{\partial x}{\partial a_3} \right) & \left(\frac{\partial x}{\partial a_2} \cdot \frac{\partial x}{\partial a_3} \right) & \left(\frac{\partial x}{\partial a_3} \cdot \frac{\partial x}{\partial a_3} \right) - 1 \end{bmatrix}$$

Der Greensche Verzerrungstensor reagiert nur auf Deformierungen. Wird das zu messende Objekt lediglich durch Starrkörpertransformationen verschoben oder gedreht, so schlägt das Maß nicht an.

Weiterhin wird der Verzerrungsratentensor benötigt, welcher ebenfalls eine dreidimensionale, quadratische und symmetrische Matrix ist. Er beschreibt die Veränderung der Deformierung über die Zeit. Er ist in Tensornotation definiert als:

$$v_{ij} = \left(\frac{\partial x}{\partial a_i} \cdot \frac{\partial \dot{x}}{\partial a_j} \right) + \left(\frac{\partial \dot{x}}{\partial a_i} \cdot \frac{\partial x}{\partial a_j} \right)$$

\dot{x} bezeichnet hierbei die Ableitung von x über die Zeit. Die Matrixnotation des Verzerrungsratentensors ist umfangreich und wird an dieser Stelle aus Platzgründen vermieden.

Das Modell stellt dem Entwickler damit ein Maß der Deformierung und ein Maß der Deformierungsrate eines beliebigen Punktes innerhalb des zu bemessenden Körpers zur Verfügung. Das Ziel des theoretischen Modells ist allerdings nicht nur die Deformierung und die dadurch entstehenden internen Kräfte zu messen, sondern auch zu erkennen, wann eine Deformierung zu einem Bruch führt und wie dieser Bruch durch das Volumen des Objektes propagiert. Diese Eigenschaften eines Volumens sind abhängig von seinem Material. Ist ein zu simulierendes Objekt aus Glas, so wird es sich kaum deformieren, bevor es bricht. Diese Eigenschaft wird als spröde (engl. „brittle“) bezeichnet und ist zurückzuführen auf einige physikalische Eigenschaften von Glas, zum Beispiel seine Tenazität / Zähigkeit, seine Dichte, seine Lamé-Konstanten und seine Dämpfungsparameter.

Um also Brüche zu simulieren und nicht nur die Deformierung, müssen die physikalischen Eigenschaften des Materials, aus dem das zu simulierende Objekt besteht, berücksichtigt werden. Dies geschieht folgendermaßen:

Es wird ein Tensor der Belastung definiert. Dieser Tensor σ kombiniert die oben definierten Tensoren zur Messung von Deformierung und Deformierungsrate mit den Eigenschaften des Materials des zu simulieren Körpers. Das Ergebnis ist ein Maß für die gesamte innere Belastung eines Körpers. Der Tensor besteht aus zwei Einzelteilen, der elastischen Belastung durch Deformierung (σ^e) und der viskosen Belastung durch die Deformierungsrate (σ^v).

$$\sigma = \sigma^e + \sigma^v$$

Die genaue mathematische Definition von den beiden Teilkomponenten der gesamten inneren Belastung folgen. Die elastische Belastung (σ^e) bezieht den Greenschen Verzerrungstensor ein und verrechnet die sogenannten Lamé-Konstanten des Materials mit dem Tensor. Die zwei Lamé-Konstanten repräsentieren die Elastizitätsmaße von Festkörpern und werden dargestellt durch einen physikalischen Druck (Kraft pro Fläche, N/m²). Ihre konventionelle Beschreibung ist gegeben durch λ und μ . Mit diesen Informationen kann nun σ^e in Tensornotation definiert werden:

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

Auch hier liefert die Tensornotation wieder eine Beschreibung der Matrix σ^e in der Zeile i und in der Spalte j . Analog wird die viskose Belastung durch die Deformationsrate (σ^v) definiert, welche, anstatt der Lamé-Konstanten, welche ein Maß für Elastizität ist, zwei andere namenlose Konstanten ϕ und φ verwendet, die dämpfende Eigenschaften des Materials repräsentieren. Sie werden ebenfalls als ein physikalischer Druck (N/m²) dargestellt. Die Tensornotation von σ^v ist somit:

$$\sigma_j^{(v)} = \sum_{k=1}^3 \phi v_{kk} \delta_{ij} + 2\psi v_{ij}$$

Nun fehlt nur noch die Definition eines einzigen Maßes, um das kontinuierliche Modell bezüglich spröder Frakturpropagation abzuschließen. Die Tenazität / Zähigkeit eines Materials wird durch eine Druckkonstante (N/m²) repräsentiert. Sie wird in den folgenden Beschreibungen aufgegriffen und ihre Relevanz wird deutlich, denn sie ist essenziell, um zu erkennen, ab welchem Ausmaß innerer Kräfte sich ein Bruch aufzutut.

2.1.3 Dukile Fraktur

Um das kontinuierliche Modell auf duktile Frakturen zu erweitern, ist nur eine kleine Änderung notwendig. Bei der Berechnung der elastischen Belastung σ^e wird der Greensche Verzerrungstensor ϵ durch die elastische Verzerrung ϵ^e ersetzt, welche im Folgenden definiert wird. Des Weiteren wird eine Routine eingebaut, welche die plastische Verzerrung aktualisiert. Dazu im Folgenden ebenfalls mehr.

Es wird mit der Neudefinition von Deformierung und Verzerrung begonnen. Betrachtet man die Greensche Verzerrung, so kann man sie aufteilen in zwei Komponenten: elastische und plastische Verzerrung.

$$\epsilon = \epsilon^p + \epsilon^e$$

Diese Aufteilung ermöglicht dem Entwickler die Separation der beiden verschiedenen Deformierungsstadien eines duktilen Materials und somit die Simulation eben jener. Die Berechnung der gesamten Verzerrung ϵ ist bereits durch den Teil des spröden Modells gegeben. Im Folgenden wird definiert, wie die plastische Verzerrung ϵ^p zu berechnen ist. Durch die Subtraktion der plastischen Verzerrung von der gesamten Verzerrung ergibt sich dann schließlich die elastische Verzerrung ϵ^e .

Im Rahmen der Berechnungen werden zwei Materialkonstanten definiert, γ_1 und γ_2 . γ_1 bezeichnet die elastische Grenze des Materials, also ab welchem Punkt plastische Deformierung auftritt. γ_2 bezeichnet die plastische Grenze des Materials, also ab welchem Punkt Frakturen auftreten. Ein sprödes Material hätte somit die Materialkonstanten $\gamma_1=0,0$ und $\gamma_2=0,0$.

Es ist wichtig zu erkennen, dass die plastische Verzerrung zu Anfang der Simulation nicht existent ist (es sei denn sie wird explizit gesetzt). Somit wird im ersten Durchgang der Kerninteraktionsschleife (zum Zeitpunkt t_0) mit der Null-Matrix als plastische Verzerrung begonnen. Der nächste Durchgang (zum Zeitpunkt t_1) berechnet die Differenz der plastischen Verzerrung vom ersten Zeitpunkt zum zweiten Zeitpunkt der Simulation, genannt $\Delta \epsilon^p$. Die neue plastische Verzerrung ergibt sich dann aus folgender Berechnung:

$$\epsilon^p := (\epsilon^p + \Delta \epsilon^p) \min \left(1, \frac{\gamma_2}{\|\epsilon^p + \Delta \epsilon^p\|} \right)$$

Die Differenz der plastischen Verzerrung wird aus der elastischen Verzerrung berechnet. Dafür wird die Ableitung der elastischen Verzerrung ϵ^e , genannt $\epsilon^{'}$, benötigt:

$$\epsilon^{'} = \epsilon - \frac{\text{Trace}(\epsilon)}{3} I$$

Wobei I die Einheitsmatrix bezeichnet und $\text{Trace}(\epsilon)$ die Spur der elastischen Verzerrung beschreibt, also die Summe der Elemente der Hauptdiagonale. Ausformuliert geschieht Folgendes: Der Durchschnitt der Hauptdiagonale von ϵ wird gebildet und ersetzt die Elemente der Hauptdiagonale einer Einheitsmatrix. Dann wird diese Matrix von ϵ subtrahiert. Dadurch werden durch die Deformierungen gegebenenfalls entstehende Vergrößerungen des Volumens verhindert. Im Folgenden definieren wir das Von-Mises-Kriterium, welches, falls es zutrifft, angibt, dass plastische Verformungen auftreten:

$$\gamma_1 < \|\epsilon'\|$$

Schließlich definieren wir die Berechnung von $\Delta \epsilon^p$, damit ϵ^p berechnet werden kann:

$$\Delta \epsilon^p = \frac{\|\epsilon'\| - \gamma_1}{\|\epsilon'\|} \cdot \epsilon'$$

Somit können nun sowohl die gesamte Verzerrung ϵ , als auch die plastische Verzerrung ϵ^p berechnet werden. Und durch Umformulierung der Zusammensetzungsformel $\epsilon = \epsilon^p + \epsilon'$ kann auch ϵ' berechnet werden. Damit sind alle notwendigen mathematischen Maße und Formeln zur Berechnung dieser Maße gegeben, um elastische und plastische Verformung zu simulieren.

2.2 Das diskrete Modell

Das oben beschriebene kontinuierliche Modell liefert alle mathematischen Funktionen, um die verschiedenen Eigenschaften eines geometrischen Körpers aus einem bestimmten Material zu berechnen. Doch um diese Berechnungen sinnvoll auf einen geometrischen Körper in einer Computer-Simulation durchzuführen, bedarf es einer sog. Diskretisierung. Hierbei wird der zu simulierende Körper auf eine gewisse Art und Weise in Teile unterteilt oder mit einem Gitter bedeckt, welches die Grenzen des Materials absteckt, je nachdem welche Diskretisierungsmethode gewählt wurde.

In diesem Projekt wurde die sog. Finite-Elemente-Methode, kurz FEM, gewählt. Sie besagt, dass der zu simulierende Bereich des Materials in kleinere Subbereiche, Elemente genannt, unterteilt werden soll. Das Material innerhalb des Volumens dieser Elemente wird durch sogenannte Shape- oder Base-Functions (dt. Form- oder Basisfunktionen) beschrieben. Diese Funktionen hängen jeweils von genau einem Knotenpunkt dieses Elementes ab. Die FEM kann mit verschiedenen Arten von Elementformen umgesetzt werden, zum Beispiel mit Quadraten, Würfeln, oder – wie im Falle dieses Projektes – mit Tetraedern.

Um nun einen dreidimensionalen Körper simulieren zu können, muss er also diskretisiert werden. Das bedeutet, dass der Raum, den der Körper einnimmt, durch Tetraeder definiert wird. Alle diese Tetraeder sind über ihre Eckpunkte mit ihren Nachbartetraedern verbunden. Existiert also zum Beispiel ein Würfel als zu simulierender Körper, so kann dieser Würfel auf verschiedene Arten in Tetraeder unterteilt werden. Eine Beispielunterteilung wird in Abbildung 2 gezeigt. Wenn es sich um komplexere dreidimensionale Körper handelt als die handelsüblichen Primitive (Sphären, Würfel, Zylinder, usw.), so wird die Unterteilung in Tetraeder komplizierter.

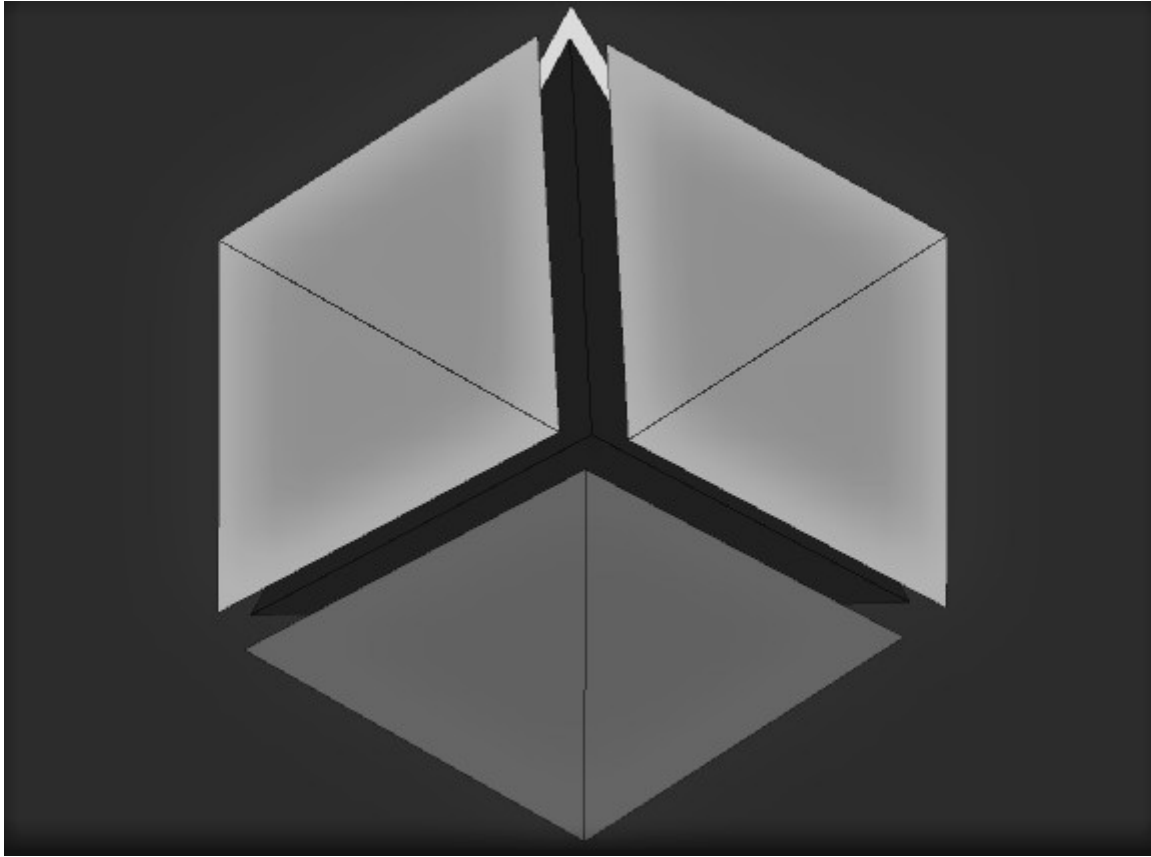


Abbildung 2: fünf Tetraeder, die einen Würfel bilden

Hier kommt die Natur dreidimensionaler Modelle in Videospielen der Diskretisierung allerdings zugute, denn sie sind meistens mit Oberflächen aus Dreiecken realisiert. So können sich bei der Erstellung der Tetraeder diese Dreiecke zunutze gemacht werden, denn Tetraeder haben als Flächen ausschließlich Dreiecke.

Eine wichtige Eigenschaft muss das entstandene Netz aus Tetraedern allerdings besitzen: Es muss konsistent sein. Das bedeutet konkret, dass alle Kanten, die zwei Tetraeder teilen, auch dieselben Knotenpunkte als Start- und Endpunkte besitzen müssen. Dies ist noch einmal grafisch in Abbildung 3 festgehalten. Diese Bedingung ist wichtig für die Remeshing-Vorschrift des Algorithmus, denn es vereinfacht die Behandlung der Tetraederverbindungen zueinander ungemein, wenn von dieser Eigenschaft ausgegangen werden kann.

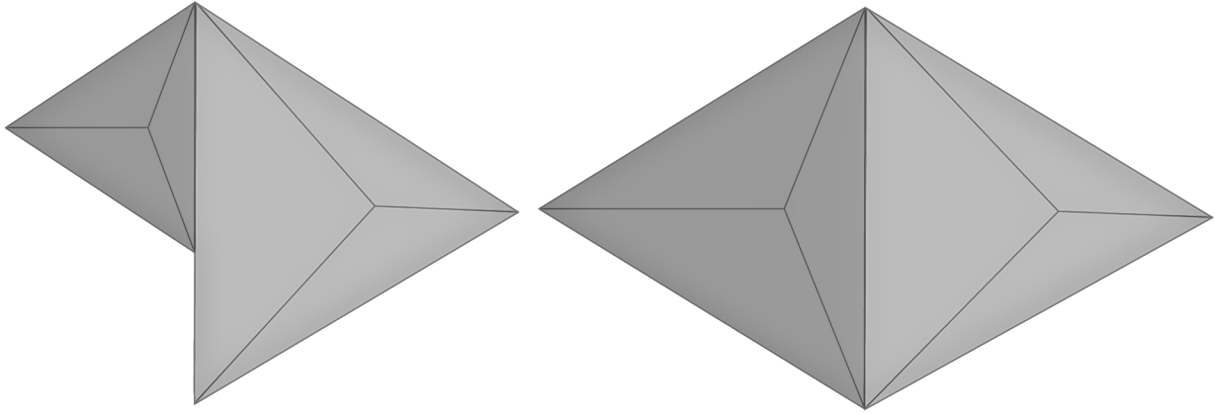


Abbildung 3: Inkonsistentes Netz (links) & konsistentes Netz (rechts) aus zwei Tetraedern.

Im Folgenden wird das diskrete Modell erläutert, insbesondere, wie es dem Entwickler mithilfe der Diskretisierung durch die Tetraeder und durch die Formeln, die aus dem kontinuierlichen Modell stammen, gelingen kann, interne Belastungen von dreidimensionalen Körpern approximieren zu können und so Frakturen simulieren zu können.

Jedes Tetraeder, im Folgenden auch Element genannt, in das das zu simulierende Objekt diskretisiert wurde, besitzt 4 Eckpunkte, im Folgenden auch Knotenpunkte genannt. Aneinanderliegende Tetraeder teilen sich diese Eckpunkte, wie auf der rechten Seite von Abbildung 3 zu sehen ist. Jeder dieser Knotenpunkte hat eine Position in der Welt, eine Position relativ zum Objekt und eine Geschwindigkeit. Jede dieser 3 Eigenschaften wird über einen dreidimensionalen Vektor dargestellt. Die Weltposition wird mit p bezeichnet, die Position relativ zum Gesamtobjekt (sog. Materialkoordinaten) wird mit m bezeichnet und die Geschwindigkeit mit v . Die 4 Knotenpunkte eines Elements werden über ihre Indizes 1-4 angesprochen. Möchte man also die Geschwindigkeit des 3. Knotenpunktes, so schreibt man v_3 .

Baryzentrische Koordinaten ermöglichen die Definition der linearen Form-/Basisfunktionen. Um die baryzentrischen Koordinaten $b = [b_1, b_2, b_3, b_4]^T$ eines beliebigen Punktes u , spezifiziert in Materialkoordinaten, zu bestimmen, wird folgende Definition von baryzentrischen Koordinaten durch die Materialkoordinaten genutzt:

$$\begin{bmatrix} u \\ 1 \end{bmatrix} = \begin{bmatrix} m_{[1]} & m_{[2]} & m_{[3]} & m_{[4]} \\ 1 & 1 & 1 & 1 \end{bmatrix} b$$

Die Invertierung dieser Formel bringt

$$b = \begin{bmatrix} m_{[1]} & m_{[2]} & m_{[3]} & m_{[4]} \\ 1 & 1 & 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} u \\ 1 \end{bmatrix}$$

Der Term

$$\begin{bmatrix} m_{[1]} & m_{[2]} & m_{[3]} & m_{[4]} \\ 1 & 1 & 1 & 1 \end{bmatrix}^{-1}$$

wird substituiert auf den Bezeichner β . Weiterhin lassen sich baryzentrische Koordinaten durch folgende Gleichungen dafür verwenden, die Weltposition und die Geschwindigkeit zu interpolieren:

$$\begin{bmatrix} x \\ 1 \end{bmatrix} = \begin{bmatrix} p_{[1]} & p_{[2]} & p_{[3]} & p_{[4]} \\ 1 & 1 & 1 & 1 \end{bmatrix} b$$

$$\begin{bmatrix} \dot{x} \\ 1 \end{bmatrix} = \begin{bmatrix} v_{[1]} & v_{[2]} & v_{[3]} & v_{[4]} \\ 1 & 1 & 1 & 1 \end{bmatrix} b$$

Durch die oben beschriebene Invertierung lassen sich Funktionen ableiten, welche für die Interpolation sorgen:

$$x(u) = P \beta \begin{bmatrix} u \\ 1 \end{bmatrix}$$

$$\dot{x}(u) = V \beta \begin{bmatrix} u \\ 1 \end{bmatrix}$$

Wobei $P = [p_{[1]} p_{[2]} p_{[3]} p_{[4]}]$ und $V = [v_{[1]} v_{[2]} v_{[3]} v_{[4]}]$.

Bei Betrachtung von Formel (2) und (3) werden die Komponenten $\frac{\partial x}{\partial a_i}$ und $\frac{\partial \dot{x}}{\partial a_i}$ ersichtlich, die partiellen Ableitungen der Position und Geschwindigkeit nach a . Diese Komponenten kann man mithilfe einer Umwandlung der Interpolationsfunktionen berechnen. Dazu benötigt man zusätzlich einen Vektor, der das Verhalten des Kronecker Deltas auf drei Dimensionen erweitert:

$$\delta_i = [\delta_{i1} \delta_{i2} \delta_{i3} 0]^T$$

$$\frac{\partial x}{\partial a_i} = P \beta \delta_i$$

$$\frac{\partial \dot{x}}{\partial a_i} = V \beta \delta_i$$

Nun sind dem Entwickler alle notwendigen Mittel gegeben, um jeden Punkt in jedem Tetraeder zu interpolieren, sei es die Weltposition, die Position relativ zum Objekt oder die Geschwindigkeit. Des Weiteren kann der Entwickler also nun auch zum Beispiel den Green-schen Verzerrungstensor eines jeden Tetraeders berechnen und somit die Verzerrung des gesamten Objektes als eine Kombination der Verzerrungen der Tetraeder darstellen. Dasselbe gilt auch für den Verzerrungsratentensor. Dadurch ist nun auch die Berechnung der elastischen Belastung σ^e und die viskose Belastung durch die Deformationsrate σ^v möglich, was letztlich dazu führt, dass der Tensor der Belastung σ berechenbar ist.

2.3 Die Kraftdekomposition

Das Prinzip der Frakturerkennung folgt einer simplen Bedingung: Das System besteht aus einem Netz aus Tetraedern, die über Knotenpunkte miteinander verbunden sind. Wenn sich das Netz verformt, verformen sich die Tetraeder. Dem Entwickler ist es möglich, die Kraft der Verformung eines Tetraeders in Form des Tensors der Belastung σ zu berechnen. Diese Kraft wirkt das Tetraeder gleichmäßig auf seine 4 Knotenpunkte aus. Wenn ein Knotenpunkt einen gewissen Schwellwert, der im Folgenden erläutert wird, überschreitet, dann tritt eine Fraktur auf. Um diesen Schwellwert mit den Kräften zu vergleichen, fehlt uns allerdings noch eine Methode, die Kräfte in ihre tensilen (spannenden) und kompressiven Bestandteile zu zersetzen.

Wird eine Eigenwertdekomposition auf den Belastungstensor σ angewendet, so erhält man drei reelle Eigenwerte $v^i(\sigma)$ und ihre korrespondierenden Eigenvektoren $\hat{n}^i(\sigma)$. Wenn die Eigenwerte positiv sind, dann liegt eine tensile Belastung vor, im negativen Falle eine kompressive Belastung.

Um die tensilen und kompressiven *Belastungen* berechnen zu können, benötigt man einerseits die Eigenwerte und andererseits eine spezielle Matrix, die auf folgende Weise aus den korrespondierenden Eigenvektoren gebildet wird:

$$m(\vec{a}) = \begin{cases} a a^T / |a| & : a \neq 0 \\ 0 & : a = 0 \end{cases}$$

Die Matrix $m(\vec{a})$ besitzt $|a|$ als Eigenwert und a als Eigenvektor. Die anderen beiden Eigenwerte dieser Matrix sind gleich null. Diese Matrix wird folgendermaßen mit den Eigenwerten verrechnet, um die Belastungen zu berechnen:

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

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

Mithilfe dieser beiden Komponenten σ^+ und σ^- werden zwei weitere Funktionen definiert, die einerseits den tensilen und andererseits den kompressiven Bestandteil der inneren *Belastungskräfte* eines Tetraeders berechnen. Dieser gesamte Vorgang wird im Folgenden als Kraftdekomposition bezeichnet. Die beiden Funktionen sind folgendermaßen definiert, wobei vol das Volumen des Tetraeders bezeichnet:

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

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

Damit ist die Kraftdekomposition abgeschlossen. Der Entwickler hat nun alle Belastungskräfte, aufgeteilt in tensile und kompressive Kräfte, für jedes Tetraeder zur Verfügung. Um nun diese Kräfte zu analysieren und festzustellen, ob sie gewisse Schwellwerte über-

schreiten, fassen wir die Kräfte folgendermaßen zusammen: Jeder Knotenpunkt hängt an mindestens einem Tetraeder. Jedes Tetraeder wirkt seine tensilen und kompressiven Belastungskräfte gleichmäßig auf alle 4 seiner Knotenpunkte aus. Somit wird jeder Knotenpunkt eine Menge besitzen, die alle tensilen Kräfte beinhaltet (bezeichnet mit $\{f^+\}$) und ebenso eine Menge für die kompressiven Kräfte (bezeichnet mit $\{f^-\}$). Die Kardinalität beider Mengen ist gleich der Anzahl der Tetraeder, an denen der Knotenpunkt hängt. Des Weiteren ist eine sogenannte unbalancierte tensile Ladung f^+ und ebenso eine unbalancierte kompressive Ladung f^- für jeden Knotenpunkt vorhanden. Diese sind die Summen über die vorher angesprochenen Mengen.

2.4 Die Frakturerkennung

Alle vorher beschriebenen Formeln und Rechenvorschriften resultieren nun in einem einzigen Sachverhalt: Wenn die Kräfte eines Knotenpunktes einen gewissen Wert überschreiten, dann tritt eine Fraktur an genau diesem Knotenpunkt auf. Im Folgenden wird geklärt, was dies für ein Schwellwert ist und wie er mit den Kräften verglichen werden muss.

Der Schwellwert ist die Zähigkeit (oder auch die Tenazität) des Materials, aus dem das zu simulierende Objekt besteht. Dieser Wert gibt an, wie widerstandsfähig ein Material gegen Rissausbreitung ist und wird in N / m^2 gemessen. Bei Glas beträgt dieser Wert zum Beispiel etwa $10.140 \text{ N} / \text{m}^2$.

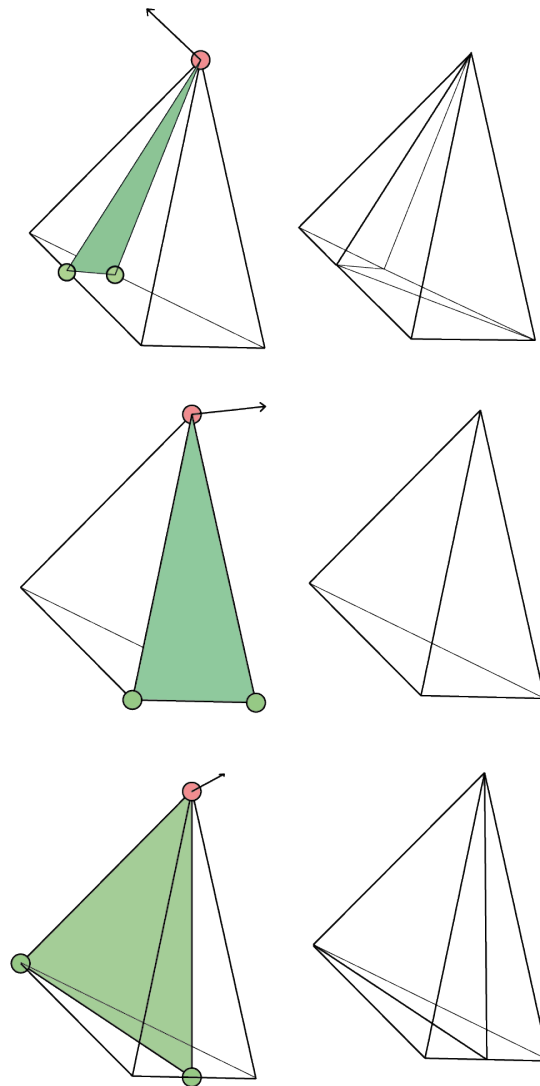
Der Wert, der diesen Schwellwert überschreiten muss, damit eine Fraktur auftritt, ist der größte Eigenwert des sogenannten Separationstensors. Der Tensor ist wie folgt definiert:

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

Ist also der größte Eigenwert von ς größer als die Zähigkeitskonstante des Materials, dann tritt eine Fraktur an diesem Knotenpunkt auf. Wie die Fraktur aussieht und was sie für Auswirkungen auf das Netz aus Tetraedern hat, wird im folgenden Kapitel erklärt.

2.5 Das Remeshing

Das Remeshing tritt auf, wenn ein Knotenpunkt im Netz der Tetraeder einer solchen Belastung ausgesetzt ist, dass die Frakturerkennungsbedingung wahr wird. Vermutlich ist die Remeshing-Vorschrift der umfangreichste und komplizierteste Bestandteil in der Implementierung des gesamten Algorithmus, denn sie deckt viele Sonderfälle ab und behandelt viele verschiedene Daten der Tetraeder und Knotenpunkte. Das Prinzip ist folgendermaßen: Eine Fraktur tritt in einem überlasteten Knotenpunkt auf. Die Fraktur besitzt eine sogenannte Frakturfläche. Diese Fläche ist definiert durch den korrespondierenden Eigenvektor des maximalen Eigenwerts des Separationstensors als Normale der Fläche. Die Frakturfläche läuft durch den überlasteten Knotenpunkt. Alle Tetraeder, die von dieser Frakturfläche geschnitten werden, werden in kleinere Tetraeder aufgeteilt. Um die Konsistenz aufrechtzuerhalten, werden außerdem die Nachbarn der aufgeteilten Tetraeder an die nun kleineren Kanten angepasst. Einige (nicht alle) Beispielszenarien sind in der Abbildung 4 festgehalten, um die Vorschrift anschaulich darzustellen.



4. Abbildung: Fallbeispiele für die Schnittmenge der Frakturfläche und eines Tetraeders und die jeweils resultierenden neuen Tetraeder

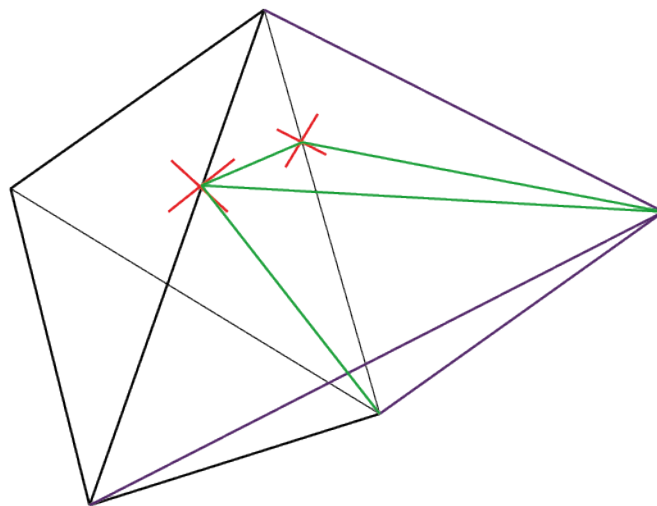
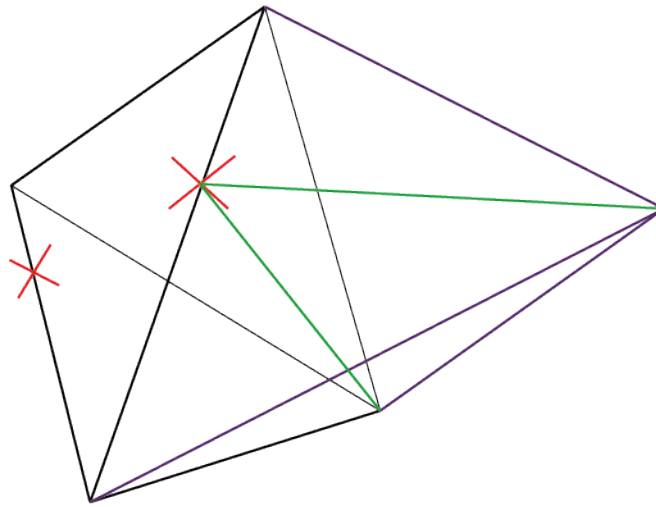
Im Folgenden wird nun die Vorgehensweise der Remeshing-Vorschrift beschrieben. Zuerst wird eine Schnittberechnung der Frakturfläche mit jedem einzelnen Tetraeder durchgeführt, welches an dem Knotenpunkt hängt, welcher die Belastungsgrenze überschritten hat. Die Schnittberechnung bezieht sich auf 3 bestimmte Kanten des Tetraeders und die Frakturfläche. Diese 3 Kanten sind die Kanten des Tetraeders, die nicht mit dem Knotenpunkt verbunden sind, welcher die Belastungsgrenze überschreitet, denn die 3 Kanten, die mit diesem Knotenpunkt verbunden sind, schneiden die Frakturfläche entweder, indem sie komplett in der Fläche liegen, oder sie schneiden sie gar nicht. Das bedeutet konkret, dass keine sinnvollen Informationen aus den Schnittberechnungen der 3 Kanten gezogen werden könnten. Die Ergebnisse dieser Berechnung sind also Punkte auf den 3 Kanten des Tetraeders, welche nicht mit dem brechenden Knotenpunkt verbunden sind oder gar nichts, wenn die Frakturfläche das Tetraeder nicht schneidet. Somit geschieht hier bereits die erste Fallunterscheidung, die im Laufe des Remeshings relevant wird: Entweder, es existieren Schnittpunkte, dann muss Remeshing geschehen, oder es existieren keine Schnittpunkte, dann ist auch kein Remeshing notwendig.

Nun gibt es noch mehrere Fallunterscheidungen für den Fall, dass eine Schnittmenge zwischen dem Tetraeder und der Frakturfläche existiert. Der erste und komplizierteste, aber auch am häufigsten auftretende Fall ist der in der Abbildung 4 ebenfalls als erstes aufgelistete Fall. Die Frakturfläche schneidet das Tetraeder so, dass zwei Schnittpunkte entstehen. Beide Schnittpunkte liegen nicht auf einem Knotenpunkt. Da der Knotenpunkt, dessen Belastungswert überschritten wurde, immer auf der Frakturfläche liegt, wird er als nicht relevant für die Fallunterscheidung betrachtet. Falls also nun zwei Schnittpunkte vorliegen, welche nicht identisch mit der Position einer der Knotenpunkte sind, wird folgendermaßen Remeshing betrieben: Das Tetraeder wird entlang der Schnittfläche aufgespalten in zwei Körper. Der eine Körper ist wieder ein Tetraeder, denn er hat wieder 4 Knotenpunkte. Der andere Körper ist ein Polyeder mit 5 Knotenpunkten. Nun gilt es, das Polyeder auch in 2 Tetraeder aufzuteilen. Dies kann recht einfach vorgenommen werden, indem es entlang einer Kante halbiert wird. Somit ist das Tetraeder in 3 kleinere Tetraeder aufgebrochen worden. Um die in Kapitelabschnitt 2.2 und Abbildung 3 beschriebene Konsistenz des Tetraedernetzes aufrechtzuerhalten, ist es nun notwendig, die Kanten des alten Tetraeders zu betrachten, welche von der Frakturfläche geschnitten wurden. In einer Schleife wird über alle Nachbarn des alten Tetraeders iteriert, um herauszufinden, ob es Nachbarn gibt, die sich die geschnittenen Kanten mit dem alten Tetraeder geteilt haben. Wenn dies der Fall ist, dann muss auch für diesen Nachbarn Remeshing betrieben werden. Hierbei gibt es wiederum 2 mögliche Fälle. Der Nachbar teilt sich nur eine der beiden geschnittenen Kanten mit dem alten Tetraeder, oder der Nachbar teilt sich beide geschnittenen Kanten mit dem Tetraeder. Falls sich der Nachbar nur eine geschnittene Kante mit dem alten Tetraeder teilt, dann wird er einfach in 2 weitere Tetraeder aufgebrochen. Dies ist in Abbildung 5 zu erkennen. Falls der Nachbar beide geschnittenen Kanten besitzt, so muss er in ein Tetraeder und ein Polyeder aufgeteilt werden. Das Polyeder wird darauf auch wieder in 2 Tetraeder unterteilt. Dieses komplizierte Verhalten wird in Abbildung 5 deutlich gemacht.

Wie zuvor allerdings beschrieben, gibt es auch andere Intersektionsarten. Also wird nun im Folgenden der 2. Fall betrachtet: Die Schnittpunkte liegen beide exakt auf zwei Knotenpunkten. Dieser Fall ist in der zweiten Zeile in der Abbildung 4 zu sehen. Diese Intersektionsart wird als koplanar bezeichnet, denn die Frakturfläche ist parallel zu einer der Seiten des Tetraeders. Wie man ebenfalls der Abbildung in der zweiten Reihe und zweiten Spalte entnehmen kann, geschieht hier kein Remeshing, denn keine neuen Tetraeder entstehen.

Zuletzt existiert noch der 3. Fall, wo nur ein Schnittpunkt genau auf einem Knotenpunkt liegt. Diese Unterscheidung zum 1. Fall ist notwendig, denn es ändert die Art und Weise, wie das Tetraeder in weitere kleinere Tetraeder aufgebrochen wird. Dieser 3. Fall ist in der dritten Zeile der Abbildung 4 aufgezeigt. Bei diesem Fall kann es ebenfalls dazu kommen, dass kein Remeshing stattfinden muss: Wenn nur ein Schnittpunkt existiert und dieser genau auf einem Knotenpunkt liegt. Das bedeutet, dass die Frakturfläche parallel zu einer der Kanten verläuft, welche mit den Knotenpunkt verbunden sind, dessen Belastungsgrenze überschritten wurde. Allerdings kann es auch dazu kommen, dass zwei Schnittpunkte existieren, wobei nur einer der beiden auf einem Knotenpunkt liegt. Hierbei lässt sich das Tetraeder direkt in 2 kleinere Tetraeder unterteilen und nicht, wie im 1. Fall, in ein Tetraeder und 1 Polyeder. Das Nachbarremeshing geschieht analog zum Nachbarremeshing im 1. Fall.

✗ = Schnittpunkt

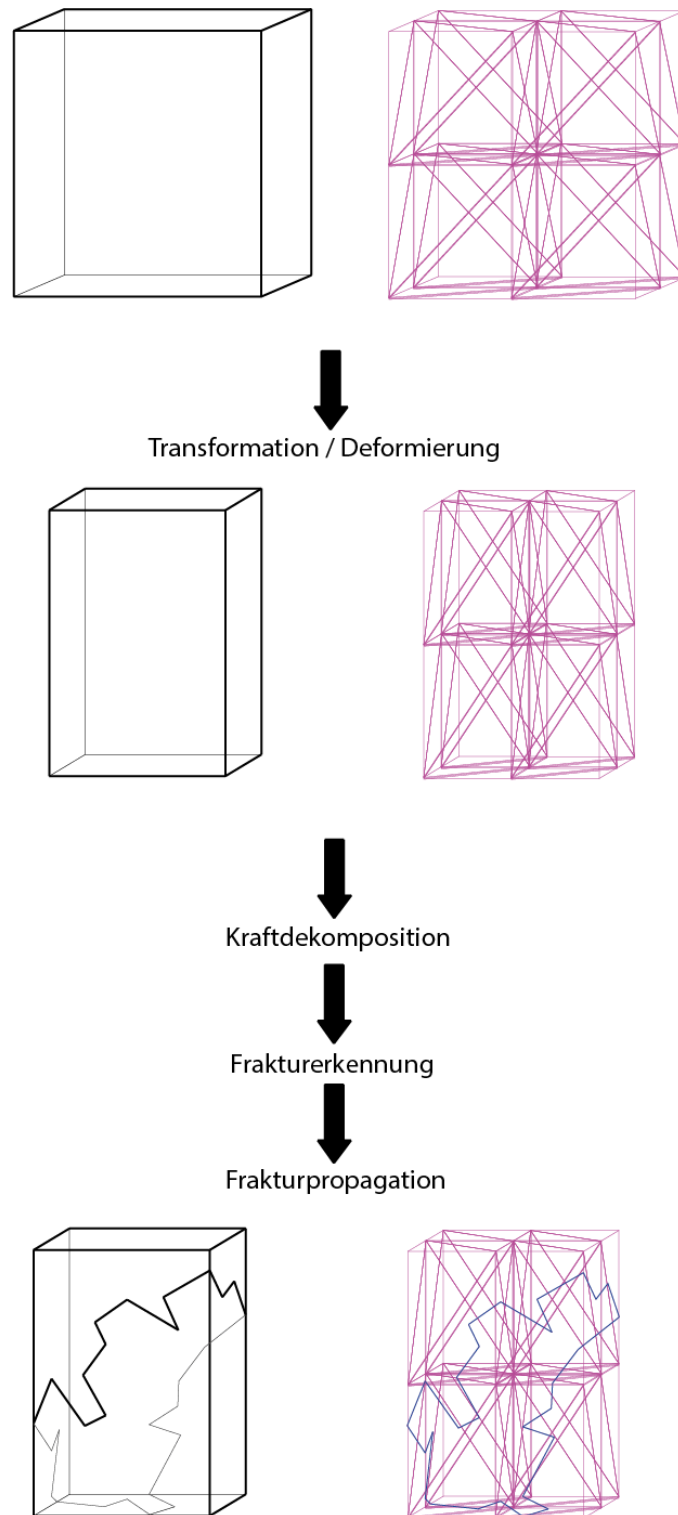


5. Abbildung: Remeshing eines Nachbarn beim 2-Schnittpunkt-Fall. Oben bei 1 gemeinsamen geschnittenen Kante, Unten bei 2 gemeinsamen geschnittenen Kanten.

2.6 Ein Kerninteraktionsschleifendurchlauf

Um nun die letzten Teilkapitel zusammenzufassen und alle beschriebenen Bestandteile des Algorithmus noch einmal abstrakt aufzuzeigen und in Relation zu setzen, wird im Folgenden anhand der Abbildung 6 der Ablauf des gesamten Algorithmus innerhalb eines Durchlaufs der Kerninteraktionsschleife des Videospiels (oder der weichen Echtzeitsimulation) beschrieben.

Die Abbildung zeigt das zu simulierende Objekt in der linken Spalte und die Tetraeder, welche das Objekt diskretisieren, in der rechten Spalte. Wichtig zu erkennen ist, dass der Algorithmus erst läuft, wenn alle geometrischen Transformationen des Spieleobjekte in diesem Frame geschehen sind. Zu Anfang des Algorithmus betrachtet der Algorithmus jedes Tetraeder und seine Knotenpunkte und berechnet die Kraftdekomposition jedes Tetraeders. Daraus ergeben sich die tensilen und kompressiven Belastungskräfte der Tetraeder. Mithilfe dieser wird die Frakturerkennung durchgeführt, die hier nun nicht pro Tetraeder, sondern pro Knotenpunkt läuft. Finden sich Knotenpunkte, welche den Schwellwert der Belastung überschreiben, so resultiert dies in Remeshing, in der Abbildung als “Frakturpropagation” bezeichnet. Hierbei werden die Tetraeder anhand der Frakturfläche in kleinere Tetraeder aufgeteilt, sodass sich der Bruch aufteilen kann und sich das Objekt an der Stelle der Fraktur spalten kann.



6. Abbildung: Ablauf eines Frames

3 Die Implementierung in Unity

Die Implementierung des in dieser Arbeit beschriebenen Algorithmus zur Frakturerkennung und -propagation von sowohl spröden als auch duktilen Materialien geschah in der Unity3D-Spieleengine. Diese Engine ist weitläufig bekannt und weist alle erdenklichen Bestandteile einer Spieleengine auf, die einem die Implementierung des Algorithmus erleichtern. So entstand zum Beispiel kein Mehraufwand, weil zuerst ein Renderer entwickelt werden musste. Des Weiteren bietet die Engine dem Entity-Component-System (kurz ECS) konforme Game-Objects, welche in ihrer minimalen Form nur einen Punkt im Raum beschreiben, oder in einer komplexeren Ausführung eines der Tetraeder, die im Kapitel 2.2 besprochen wurden. Letztlich bietet die Unity-Engine eine äußerst nützliche Funktionalität zum schnellen Testen: das Starten des Spiels im Editor. Dadurch wird der Algorithmus im selben Fenster aktiv, in welchem gerade noch die Game-Objects gesetzt und manipuliert wurden.

Die Implementierung des Algorithmus folgt streng dem „Entity-Component-System“-Muster insofern, dass alle Objekte oder komplexeren Entitäten des Algorithmus als Game-Objects realisiert sind. Der Algorithmus setzt beispielsweise voraus, dass das zu simulierende Objekt aus Tetraedern besteht, die über Knotenpunkte miteinander verbunden sind. So sind die Tetraeder selbst als Game-Objects umgesetzt und die Knotenpunkte, die die Tetraeder verbinden, ebenfalls. Diese intuitive Herangehensweise ermöglichte eine schnelle Testbarkeit, denn Game-Objects in Unity sind zur Laufzeit eines Spiels recht einfach zu manipulieren. So kann ein Nutzer des Algorithmus etwa ein Mesh an Tetraedern und Knotenpunkten erstellen, das Spiel innerhalb der Unity-Engine starten und zur Laufzeit des Spiels die Knotenpunkte mit der Maus verschieben und somit Deformierungen des Meshes erschaffen.

Die Algorithmus-Funktionalität wurde folgendermaßen an die Game-Objects angehängt: In Unity existiert das Konzept von Skripten. Ein Skript in Unity ist eine Komponente eines Game-Objects, implementiert durch eine C#-Klasse. Diese Quellcodedatei enthält mehrere Callback-Funktionen, die bei Start des Spiels (zum Beispiel innerhalb des Editors) oder während das Spiel läuft von der Engine aufgerufen werden können (insofern sie implementiert sind). So existiert die `start()`-Methode, welche bei Beginn des Spiels einmal aufgerufen wird und die `update()`-Methode, die einmal pro Durchlauf der Kerninteraktionsschleife des Spiels aufgerufen wird. Da das Skript ansonsten eine reguläre Klasse im objektorientierten Sinne ist, lassen sich einfach weitere Methoden und Attribute definieren.

Im Folgenden wird der Vorgang der Implementierung beschrieben. Es wird beleuchtet, in welcher Form die Diskretisierung erreicht wurde, also inwiefern das diskrete Modell umgesetzt wurde. Daraufhin wird die Berechnung der verschiedenen Maße von Deformierung, Belastung und interner Kräfte beschrieben. Nicht zuletzt werden Inhalte aufgelistet, die weiterhin implementiert werden könnten. Abschließend wird kurz erläutert, wie das System in ein existierendes Unity-Spieleprojekt eingeplegt werden kann.

3.1 Die Planung

Der Beginn der Bearbeitung dieser Abschlussarbeit bestand aus einer intensiven Recherche an Materialien, die den Einstieg in die sehr physikalische Thematik erleichtern konnten. Hierbei

wurde vor allem auf die vom Algorithmus referenzierte Literatur zurückgegriffen, um ein generelles Verständnis für die physischen Abläufe zu erlangen. Die im Algorithmus-Paper verwendete Notation, die sogenannte Tensor-Notation, musste nachvollzogen und verstanden werden, äußerte sich dann aber als sehr hilfreich bei der Formulierung der verschiedenen Daten. Nachdem der Algorithmus verstanden und in einen physikalischen Kontext eingebunden war, konnte mit der Planung der Software begonnen werden. Es stand bereits zu Anfang fest, dass die Implementierung in der Unity-Engine stattfinden soll und so umgesetzt werden sollte, dass sie von verschiedensten Unity-Spielprojekten eingebunden werden konnte.

3.1.1 Die Architektur

Die Idee der Architektur war folgende: Das zu simulierende Objekt würde als ein Game-Object realisiert sein. Es solle ein Kind-Game-Object besitzen, welches den expliziten Namen "FEM_Mesh" tragen müsse. Dieses Kind-Game-Object wäre das Wurzelement für die Datenhaltung und Funktionalität des Algorithmus. Dieses "FEM_Mesh"-Game-Object besäße eine C#-Skript-Komponente, die den Algorithmus umsetze. Die Daten des Algorithmus - die Tetraeder und Knotenpunkte - wären als Kind-Game-Objects des "FEM_Mesh"-Game-Objects realisiert und würden ihre Daten und Funktionen innerhalb einer C#-Klasse kapseln, die sie als Skript-Komponente angehängen bekämen. Hierbei wird kurz das Konzept eines Unity-Prefabs erläutert. Prefabs erlauben in Unity, ein Game-Object mit all seinen Komponenten und Einstellungen als eine Datei abzuspeichern. Dies ermöglicht, dieses Game-Object zu einem beliebigen Zeitpunkt im Spiel in beliebiger Anzahl, mit beliebiger Position und Ausrichtung zu instanziierten. Es solle ein Prefab für einen Knotenpunkt existieren, denn solch einer solle im Quellcode instanziiert werden können. Diese Architektur sollte dem Nutzer des Algorithmus ermöglichen, den Algorithmus einfach in ein eventuell bereits existierendes Spielprojekt einzubinden, indem man dem zu simulierenden Game-Object ein weiteres Kind-Game-Object anhängen würde, welches die beschriebene Funktionalität implementierte.

In Falle der entwickelten Testfälle wurde auf das Eltern-Game-Object verzichtet und einfach direkt ein "FEM_Mesh"-Game-Object kreiert.

3.1.2 Das Testsystem

Damit war eine Architektur gegeben. Nun fehlte noch das Konzept eines Workflows, welches das sinnvolle und breite Testen der zu entwickelnden Funktionalität ermöglichen würde. Hierfür würde ein Editorskript reichen, dass die spontane Erstellung von Tetraeder-Meshes mit ihren Knotenpunkten ermöglichen sollte. Des Weiteren sollte das Tetraeder-Mesh nicht einfach nur eine im Quellcode vorhandene Datenstruktur sein, sondern sichtbar gezeichnet werden, damit eventuelle Inkonsistenzen, Fehler oder Artefakte sofort erkannt würden. Damit war die Grundlage für die Implementierung des Algorithmus in Unity gegeben.

Anhand dessen wurden im Laufe der Implementierung dem Entwickler sinnvoll erscheinende Tests umgesetzt. Für jeden zu testenden Aspekt des Projekts wurde eine neue Unity-Szene angelegt. Eine Unity-Szene ist ein Konzept aus der Unity-Engine, welches (unter anderem) alle Positionen, Rotationen und Skalierungen jedes Game-Objects in der Spielwelt in einer Datei speichert. Ein Projekt kann mehrere Szenen haben. Die eigentliche Verwendung von Szenen basiert auf dem Konzept von Levels in Videospielen, aber kann in diesem Kontext auch hervorragend zum Testen verwendet werden.

3.2 Ablauf der Implementierung

Zu Beginn der Entwicklungsphase nach der Planungsphase wurde zuerst das Workflow-Konzept umgesetzt. Dies bedeutet konkret, dass zuerst das in Kapitelabschnitt 3.1.1 erwähnte Prefab der Knotenpunkte entwickelt wurde und darauf das in Kapitelabschnitt 3.1.2 erwähnte Editorskript entwickelt wurde.

// wie gings weiter?

3.2.1 Das Node-Prefab

Das entwickelte Knotenpunkt-Prefab enthält verschiedene Komponenten:

- zuerst die obligatorische “Transform”-Komponente, welche die Position, Rotation und Skalierung des Objektes speichert,
- einen Sphärenrenderer, welcher eine kleine Kugel an der Stelle des Knotenpunktes rendert,
- eine Skriptkomponente namens `PositionChangeListener`, welche in einem folgenden Abschnitt erläutert wird und
- eine Skriptkomponente namens `Node`, welche die Funktionalität und Datenhaltung des Frakturalgorithmus bezüglich der Knotenpunkte des Tetraedernetzes beinhaltet. Hierzu ebenfalls mehr im folgenden Paragraphen.

Diese Knotenpunkte können vom Nutzer beliebig oft in der Szene instanziiert werden und beliebig positioniert werden.

3.2.2 Das Editorskript `FractureTool`

Ziel dieses Editorskripts `FractureTool` war es,

- dem Benutzer des im Rahmen dieses Projekts entwickelten Systems zu ermöglichen, Tetraedernetze im Editor von Unity intuitiv erstellen zu können,
- das erstellte Netz direkt mit dem Algorithmus zu versehen und
- eine ordentliche, automatische Game-Object-Hierarchie für das Tetraeder-Netz anzulegen und zu verwalten.

Das Editorskript ist eine C#-Klasse, die von der von Unity gegebenen Klasse `EditorWindow` ableitet. Sie implementiert

- eine Funktion `OpenWindow()`, welche ein Fenster öffnet,
- die Callback-Funktion `OnGUI()`, welche in dem Fenster, das von `OpenWindow()` geöffnet wird, einige Buttons bereitstellt, die die Erstellung von Tetraedern ermöglichen. Der erste Button generiert ein Tetraeder aus den Knotenpunkten, die der Nutzer in der Szene gerade selektiert hat. Hierbei ist wichtig, dass exakt 4 Knotenpunkte und nichts sonst ausgewählt ist, sonst wird kein Tetraeder generiert. Der zweite Button aktualisiert die Relationen aller Knotenpunkte und Tetraeder zueinander (dazu später genauere Informationen). Dies geschieht beim Starten des Spiels zwar automatisch, aber ist zum expliziten Testen und Debuggen eine sinnvolle Funktionalität, um bereits vor Start der Anwendung zu sehen, wie die Relationen der Daten zueinander aussehen. und
- eine Callback-Funktion `OnHierarchyChange()` zur Aufrechterhaltung der Game-Object-Hierarchie. Diese Funktion wird aufgerufen, wenn sich, sowohl im Editiermodus der Engine, als auch im Spielmodus, etwas in der Game-Object-Hierarchie der Szene ändert. Die Funktion stellt sicher, dass die Tetraeder- und Knotenpunkt-Hierarchien korrekt bleiben.

3.2.3 Die Klasse `PositionChangeListener` und die Schnittstelle `IPositionChangeListener`

Die Klasse `PositionChangeListener` und die Schnittstelle `IPositionChangeListener` realisieren eine vereinfachte Version des Entwurfsmusters “Observer”. Da die Klasse `PositionChangeListener` als Skriptkomponente an dem Node-Prefab hängt und die Callback-Funktion `update()` implementiert, kann sie erkennen, sobald sich die Position eines Knotenpunktes verändert. An dieser Stelle ist wichtig, zu erwähnen, dass die Klasse, obwohl sie als Skriptkomponente an den Knotenpunkten hängt, auch im Editiermodus aktiv ist und nicht nur, wenn die Simulation aktiv wird. Dies ist über das Attribut `[ExecuteInEditMode]` realisiert. Wenn die Klasse eine Positionsveränderung registriert, dann kommuniziert sie dies an die Liste aller registrierten Beobachter (engl. “Observer/Listener”). Dies geschieht über die Schnittstelle `IPositionChangeListener`. Jede Klasse, die diese Schnittstelle implementiert, implementiert ebenfalls eine Funktion namens `updateListener(GameObject changed_node)`. Diese Funktion wird eben dann aufgerufen, wenn sich die Position eines Knotenpunktes verändert.

Der Zweck dieser Klasse und Schnittstelle besteht vor allem darin, dass die vom Benutzer des Systems erstellten Tetraeder sich mit einer gegebenen geometrischen Transformation der Knotenpunkte ebenfalls transformieren soll. Zieht man beispielsweise einen Knotenpunkt von einem beliebigen Tetraeder weiter von ihm weg, dann soll sich das Tetraeder entsprechend strecken und nicht einfach unberührt in seiner Position verweilen.

3.2.4 Die Klasse `TetrahedronBuilder`

Wird im Editor ein neues Tetraeder mithilfe des Editorskripts `FractureTool` erstellt, dann wird die Klasse `TetrahedronBuilder` genutzt, um die korrekte Geometrie des Tetraeders aufzubauen und das resultierende Game-Object des Tetraeders mit allen zugehörigen Komponenten zusammenzustellen. Die Klasse implementiert nur eine einzige öffentlich aufrufbare (public) Funktion namens `GenerateTetrahedron(GameObject[] nodes)`, welche das Tetraeder-Game-Objekt generiert und zurückgibt. Des Weiteren implementiert die Klasse drei weitere, private Hilfsfunktionen, die die Erstellung des Tetraeders unterstützen.

Die Funktion `GenerateTetrahedron` wird allerdings nicht nur vom Editorskript verwendet, sondern findet auch Anwendung, falls die laufende Simulation einen Bruch erkennt und Remeshing betreiben muss. Dieses Thema wird im Abschnitt der Klasse `Node` detaillierter erläutert.

Da die Funktionalität dieser Klasse sowohl im Editiermodus, als auch in der laufenden Simulation erforderlich ist, muss das Skript als Komponente an einem Game-Object hängen. Deshalb sollte es immer am “FEM_Mesh”-Elternobjekt hängen, welches in Kapitelabschnitt 3.1.1 beschrieben wurde.

3.2.5 Die Klasse `RelationManager`

Alle Tetraeder verwalten eine Liste ihrer Nachbartetraeder, also alle Tetraeder, mit denen sie sich mindestens einen Knotenpunkt teilen. Weiterhin verwalten sie zu jedem ihrer Nachbarn eine Liste mit Knotenpunkten, die sie sich mit eben jenem Nachbarn teilen. Außerdem kennt jedes Tetraeder alle 4 Knotenpunkte, die es definieren. Alle Knotenpunkte verwalten eine Liste der Tetraeder, an denen sie hängen.

Die Konsistenz dieser Daten muss gegeben sein, sobald die Knotenpunkte auf ihre Belastungen überprüft werden. Dieses Ziel verfolgt die Klasse `RelationManager` mit zwei essentiellen

Funktionen: `UpdateAttachedElements()` und `UpdateNeighbors()`. Die erste Funktion iteriert über alle Tetraeder, betrachtet jeweils die 4 Knotenpunkte der Tetraeder und trägt das jeweilige Tetraeder in die Liste aller Tetraeder ein, an denen der jeweilige Knotenpunkt hängt. Die zweite Funktion iteriert mit einer doppelten For-Schleife über alle Tetraeder und trägt jeweils die Nachbarn der Tetraeder in ihre Listen ein.

Die Funktionalität dieser Klasse wird nicht unbedingt einmal pro Frame aufgerufen, denn dies würde aufgrund der doppelten For-Schleife in der `UpdateNeighbors()`-Funktion schnell zu merklichen Effizienzeinbrüchen führen, besonders weil die Anzahl der Tetraeder im Laufe der Simulation im Falle von Frakturen stark ansteigen kann. Anstattdessen wird diese Funktion einmal am Anfang der Simulation aufgerufen, um die Relationen erstmalig zu definieren und dann jedes mal aufgerufen, wenn Remeshing stattfindet, um die neuen und alten Tetraeder mit ihren Nachbarn bekanntzumachen und die Datenhaltung der neuen Knotenpunkte zu aktualisieren. Außerdem kann diese Funktionalität ebenfalls explizit im Editor aufgerufen werden, wie im Kapitelabschnitt 3.2.2 beschrieben.

3.2.6 Die Klasse `FractureSimulator`

Der Ablauf des in diesem Projekt beschriebenen Algorithmus ist in Abbildung 6 festgehalten. Die höhere Logik des Algorithmus ist an sich also recht simpel. Diese Logik wird in der Klasse `FractureSimulator` umgesetzt. Da die Klasse in Form einer Skriptkomponente am "FEM_Mesh"-Elternobjekt hängt, kann sie die `FixedUpdate()`-Callback-Funktion implementieren. In dieser Funktion wird eben diese Logik ausgeführt.

Zuerst wird die Geschwindigkeit aller Knotenpunkte in Relation zu ihrer Position im vorherigen Frame berechnet. Daraufhin werden die internen Kräfte aller Tetraeder berechnet und auf ihre Knotenpunkte angewandt, indem die Funktion `UpdateInternalForcesOfNodes()` der Klasse `Tetrahedron` aufgerufen wird, welche die Kraftanalyse und -dekomposition durchführt und die resultierenden Kräfte in die Datenstruktur der Knotenpunkte des jeweiligen Tetraeders einträgt. Dann wird über alle Knotenpunkte iteriert und die Frakturbedingung, beschrieben in Kapitelabschnitt 2.4, für jeden Knotenpunkt abgefragt. Trifft die Frakturbedingung für einen Knoten zu, so wird Remeshing für diesen Knotenpunkt betrieben. Die Frakturerkennung und das Remeshing sind jedoch nicht in der Klasse `FractureSimulator`, sondern in der Klasse `Node` implementiert. Die Kraftdekomposition geschieht in der Klasse `Tetrahedron`.

Des Weiteren verwaltet die Klasse einige physische Beispielmateriale, wie Glas. Dies ist realisiert über eine private interne Klasse mit dem Namen `Material` und all den physischen Konstanten, die diesem Material angehören und die zur Berechnung der Belastungen und Kräfte notwendig sind.

3.2.7 Die Klasse `Tetrahedron`

Die Klasse `Tetrahedron` hängt an allen Tetraeder-Game-Objects als Skriptkomponente. Sie enthält die Datenstruktur aller Nachbarn, die in Abschnitt 3.2.5 beschrieben wurde, sowie eine Liste seiner 4 Knotenpunkte. Des Weiteren implementiert die Klasse eine Reihe an Funktionen, die jeweils die verschiedenen Formeln implementieren, welche im Kapitel 2 aufgeführt wurden:

- die Funktion `Beta()` berechnet die $\begin{bmatrix} m_{[1]} & m_{[2]} & m_{[3]} & m_{[4]} \\ 1 & 1 & 1 & 1 \end{bmatrix}^{-1}$ -Matrix aus dem Abschnitt 2.2, welche auf den Bezeichner β substituiert wurde. Diese Matrix enthält die Koeffizienten für die lineare Basisfunktion des Tetraeders. Die Funktion wird

aufgerufen, wenn das Tetraeder erstellt wird, oder wenn es neue Knotenpunkte zugewiesen bekommt.

- Die Funktion `Volume()` berechnet das Volumen des Tetraeders.
- Die Funktion `P()` berechnet die Matrix $P = [p_{[1]} p_{[2]} p_{[3]} p_{[4]}]$ aus Abschnitt 2.2.
- Die Funktion `V()` berechnet die Matrix $V = [v_{[1]} v_{[2]} v_{[3]} v_{[4]}]$ aus Abschnitt 2.2.
- Die Funktion `PartDerivOfXWithRespectToU()` berechnet den Term $\frac{\partial x}{\partial a_i} = P \beta \delta_i$ aus Abschnitt 2.2.
- Die Funktion `PartDerivOfXDotWithRespectToU()` berechnet den Term $\frac{\partial \dot{x}}{\partial a_i} = V \beta \delta_i$ aus Abschnitt 2.2.
- Die Funktion `GreenStrain()` berechnet den Tensor $\epsilon_{ij} = \left(\frac{\partial x}{\partial a_i} \cdot \frac{\partial x}{\partial a_j} \right) - \delta_{ij}$ aus Abschnitt 2.1.2.
- Die Funktion `StrainRate()` berechnet die Matrix $v_{ij} = \left(\frac{\partial x}{\partial a_i} \cdot \frac{\partial \dot{x}}{\partial a_j} \right) + \left(\frac{\partial \dot{x}}{\partial a_i} \cdot \frac{\partial x}{\partial a_j} \right)$ aus Abschnitt 2.1.2.
- Die Funktion `ElasticStressDueToStrain()` berechnet die Matrix $\sigma_j^{(\epsilon)} = \sum_{k=1}^3 \lambda \epsilon_{kk} \delta_{ij} + 2 \mu \epsilon_{ij}$ aus Abschnitt 2.1.2.
- Die Funktion `ViscousStressDueToStrainRate()` berechnet die Matrix $\sigma_j^{(v)} = \sum_{k=1}^3 \phi v_{kk} \delta_{ij} + 2 \psi v_{ij}$ aus Abschnitt 2.1.2.
- Die Funktion `ElasticStrain()` berechnet die elastische Belastung ϵ^e aus Abschnitt 2.1.3, welches einfach die Reduktion von ϵ um ϵ^p ist.
- Die Funktion `ElasticStrainDeviation()` berechnet die Ableitung der elastischen Belastung $\epsilon' = \epsilon - \frac{\text{Trace}(\epsilon)}{3} I$ aus Abschnitt 2.1.3.
- Die Funktion `DeltaPlasticStrain()` berechnet den Term $\Delta \epsilon^p = \frac{\|\epsilon'\| - \gamma_1}{\|\epsilon'\|} \cdot \epsilon'$ aus Abschnitt 2.1.3.
- Die Funktion `PlasticStrain()` berechnet die plastische Belastung $\epsilon^p := (\epsilon^p + \Delta \epsilon^p) \min \left(1, \frac{\gamma_2}{\|\epsilon^p + \Delta \epsilon^p\|} \right)$ aus Abschnitt 2.1.3.

- Die Funktion `VonMisesYieldCriterion()` berechnet den Wahrheitswert für das Von-Mises-Kriterium beschrieben in Abschnitt 2.1.3.
- Die Funktion `TotalInternalStress()` berechnet die Summe aus $\sigma^{(\epsilon)}$ und $\sigma^{(v)}$.

Diese Funktionen sind alle Teil des kontinuierlichen und diskreten Modells, um die Kräfte zu berechnen, welche die Tetraeder auf ihre Knotenpunkte ausüben. Im Folgenden werden weitere Funktionen von der Klasse `Tetrahedron` aufgeführt, die sich um die Kraftdekomposition kümmern:

- Die Funktion `TensileComponentOfTotalInternalStress()` berechnet die Matrix $\sigma^+ = \sum_{i=1}^3 \max(0, v^i(\sigma)) m(\hat{n}^i(\sigma))$ aus Abschnitt 2.3.
- Die Funktion `CompressiveComponentOfTotalInternalStress()` berechnet die Matrix $\sigma^- = \sum_{i=1}^3 \min(0, v^i(\sigma)) m(\hat{n}^i(\sigma))$ aus Abschnitt 2.3.
- Mithilfe der obigen Funktionen kann der Entwickler nun die tensile Kraft eines Tetraeders auf seine Knotenpunkte mit der Funktion `TensileForce()` berechnen. Sie implementiert die Gleichung $f_{[i]}^+ = \frac{-vol}{2} \sum_{j=1}^4 p_{[j]} \sum_{k=1}^3 \sum_{l=1}^3 \beta_{jl} \beta_{ik} \sigma_{kl}^+$ aus Abschnitt 2.3.
- Ebenso kann die kompressive Kraft eines Tetraeders mit der Funktion `CompressiveForce()` berechnet werden. Sie implementiert die Gleichung $f_{[i]}^- = \frac{-vol}{2} \sum_{j=1}^4 p_{[j]} \sum_{k=1}^3 \sum_{l=1}^3 \beta_{jl} \beta_{ik} \sigma_{kl}^-$ aus Abschnitt 2.3.

All diese Funktionen fassen den gesamten Prozess der Kraftanalyse und -dekomposition, beschrieben in dem vorhergehenden Kapitel, zusammen. Ruft man nun die Funktion `UpdateInternalForcesOfNodes()` auf, so wird über alle Knotenpunkte eines Tetraeders iteriert und die tensile sowie kompressive Kraft in die Datenstruktur der jeweiligen Knotenpunkte eingetragen.

3.2.8 Die Klasse `Node`

Die `Node`-Klasse behandelt die Datenhaltung eines Knotenpunktes, also die Liste der Tetraeder, an denen der Knotenpunkt hängt, und die gesamte Frakturerkennungslogik und Remeshing-Logik. So implementiert sie die Funktion `IsExceedingDeformationLimit()`, welche die Frakturerkennung, beschrieben in Kapitelabschnitt 2.4, implementiert. Diese Funktion wird für jeden Knotenpunkt in jedem Frame von der Klasse `FractureSimulator` aufgerufen. Sie bildet den Separationstensor ζ aus den ihr von den anhängenden Tetraedern übermittelten tensilen und kompressiven Kräften. Darauf folgt eine Eigenwertdekomposition des Separationstensors und ein Vergleich des größten Eigenwerts mit der Zähigkeit des simulierten Materials. Wenn ein Eigenwert diese Zähigkeit überschreitet, tritt ein Bruch auf und die Funktion gibt einen positiven Wahrheitswert zurück. Wenn kein Eigenwert größer als die Zähigkeit des Materials ist, tritt kein Bruch auf.

Sollte ein Bruch auftreten, so wird die Funktion `Remesh()` von der Klasse `FractureSimulator` aufgerufen. Bei dieser Funktion handelt es sich um die umfangreichste des gesamten

Systems. Sie deckt viele Einzelfälle ab und benötigt deshalb mehrere hundert Zeilen Code. Im Kapitel 2.5 wird die Funktionalität im Detail behandelt. Am Ende der Funktion werden verschiedene Datenstrukturen zurückgegeben, welche Referenzen auf die neuen und veralteten Tetraeder und Knotenpunkte enthalten, die es dann zu entfernen, beziehungsweise in die Datenstrukturen einzupflegen gilt.

3.2.9 Die Klasse MathUtility

Die Klasse MathUtility ist eine eigens entwickelte Klasse, die mathematische Hilfsfunktionen bereitstellt, wie etwa die Berechnung des Kronecker Deltas mit der Funktion `KroneckerDelta(int i, int j)`, oder `ProjectOnto(Vector<float> a, Vector<float> b)`, welche einen Vektor a auf einen Vektor b projiziert und den resultierenden Vektor zurückgibt.

3.3 Mögliche Erweiterungen

Das entwickelte Unityprojekt bietet die Möglichkeit, ein Objekt zu simulieren, das bereits ein konsistentes Netz aus Tetraedern als ein Kind-Game-Object besitzt. Diese Tetraeder haben weiterhin allerdings keine Collider, das System kann also nur auf interne Kräfte reagieren. Dieser Sachverhalt präsentiert also direkt zwei Möglichkeiten der Erweiterung:

- Eine automatische Generierung des Tetraedernetzes anhand des Rendering-Meshes des zu simulierenden Objektes. Hierfür gibt es verschiedene Algorithmen, welche häufig die Delaunay-Tetraedrisierung verwenden, siehe zum Beispiel **X**, **Y** und **Z**.
- Die Generierung von Collidern, welche dann auch die Evaluation von externen Kräften und deren Effekte auf die Knotenpunkte zulassen würden.

Ein weiterer denkbarer Ansatz wäre die Einführung eines Events, welches feuert, wenn sich ein Bruch auftut, um Spielen die Möglichkeit zu geben, auf den Bruch zu reagieren, zum Beispiel durch Geräuscheffekte oder visuelle Effekte.

Nicht zuletzt wäre auch eine Art der Abspaltungserkennung sinnvoll. Falls sich also Teile des Netzes komplett vom Netz abspalten, sollte dies erkannt werden und es sollte aus den Einzelteilen wieder eigene Game-Objects gemacht werden, um diese separat behandeln zu können. So könnten alle Einzelteile separate Rigidbody-Komponenten erhalten, um jedes Einzelteil einzeln physikalisch zu simulieren.

4 Diskussion der Ergebnisse

Im Folgenden werden die Performanzergebnisse und Komplexitäten des Algorithmus aufgezeigt und seine Eignung für Videospiele geprüft. Danach werden Ideen und Änderungsvorschläge präsentiert, welche die Eignung für Videospiele erhöhen könnten.

4.1 Eine grobe Komplexitätsanalyse

Zuerst wird die grobe Komplexität der höheren Logik des Systems betrachtet. Hierbei ist besonders die `FixedUpdate()`-Funktion der Klasse `FractureSimulator` relevant. Die Anzahl aller Knotenpunkte wird mit k bezeichnet, die Anzahl aller Tetraeder mit t . Die gegebenen Anweisungen werden mit der O-Notation versehen.

```
foreach (Node n in allnodes)
{
    n.velocity = (n.transform.position - n.old_world_position) / Time.deltaTime;
    n.old_world_position = n.transform.position;
}

foreach (Tetrahedron t in alltetrahedra)
{
    t.UpdateInternalForcesOfNodes(current.dilation, current.rigidity, current.phi,
    current.psi, current.elastic_limit, current.plastic_limit);
}
```

Die erste Schleife hat die Komplexität $O(n)$ und behandelt lediglich zwei fast atomare Zuweisungen pro Schleifendurchlauf. Die zweite Schleife hat die Komplexität $O(t)$ und behandelt die komplexe Berechnung der Kräfteanalyse und -dekomposition jedes Tetraeders, beschrieben in den Kapitelabschnitten 2.1 und 2.3. Auf dieser höheren Logik ist hier also ebenfalls eine lineare Komplexität vorhanden, wie in der ersten Schleife, allerdings ist die Berechnung der Kräfte verhältnismäßig aufwändig. Nun folgt die Frakturerkennung und das Remeshing:

```
int node_count = allnodes.Count;
for (int i = 0; i < node_count; i++)
{
    Node n = allnodes[i];
    if (n.IsExceedingDeformationLimit(current.toughness))
    {
        nodes_to_be_removed.Add(n);
        Tuple<List<Tetrahedron>, List<Tetrahedron>, List<Node>>
            updated_tets_and_nodes = n.Remesh(transform.position);

        foreach (Tetrahedron t in updated_tets_and_nodes.Item1)
        {
            alltetrahedra.Remove(t);
        }
    }
}
```

```

        t.nodes = null;
        t.node_transforms.Clear();
        Destroy(t.gameObject);
    }
    alltetrahedra.AddRange(updated_tets_and_nodes.Item2);
    allnodes.AddRange(updated_tets_and_nodes.Item3);
    Debug.Log("Crack occurs");

    relation_manager.UpdateRelations();

    foreach (Tetrahedron t in alltetrahedra)
    {
        t.Beta();
    }

    foreach(Tetrahedron t in updated_tets_and_nodes.Item2)
    {
        t.ResetElasticStrain();
        t.ResetPlasticStrain();
    }
}
n.ClearTensileAndCompressiveForces();
}

```

Die äußere For-Schleife hat eine Komplexität von $O(n)$. Die inneren Vorschleifen haben jeweils die Komplexität von $O(t)$. Somit hat die gesamte For-Schleife bereits eine Komplexität von $O(n * t)$. Hinzu kommt die Komplexität von `UpdateRelations()`, welche bei $O(n * t^2)$ liegt. Somit besitzt auch diese For-Schleife in einer tieferen Betrachtungsweise bereits eine Komplexität von $O(n * t^2)$. Die Komplexität ist in einem nicht-wünschenswerten Bereich für eine Anwendung mit weichen Echtzeitanforderungen.

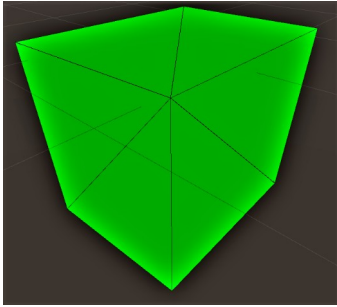
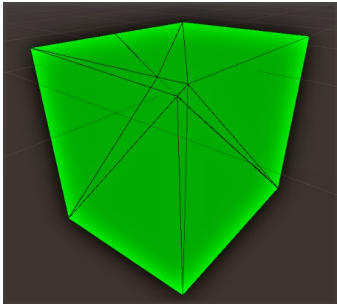
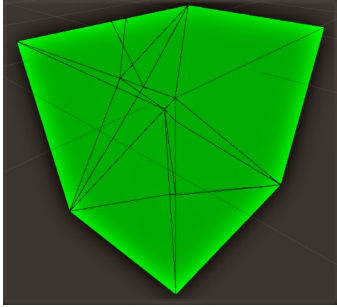
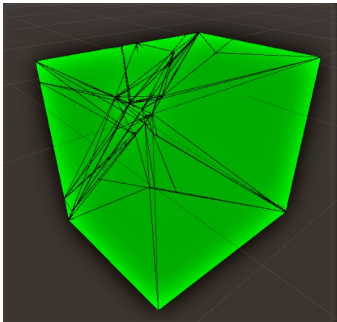
Ohne tiefere Komplexitäten der Funktionsaufrufe zu berechnen, liegt die Komplexität also bereits bei einem quadratischen Niveau. Betrachtet man nun die Tatsache, dass das Remeshing eines einzelnen Tetraeders im schlimmsten Fall zu drei weiteren Tetraedern führt (exklusive der Nachbarn, welche ebenfalls Remeshing erfahren), unterliegt die Anzahl der Knotenpunkte und insbesondere der Tetraeder einem explosiven Wachstum, sollten Brüche vorkommen.

Diese Komplexitätsdimensionen prophezeien also, dass diese Algorithmus für Videospiele nicht geeignet ist. In der Tat führen Praxisversuche unter dieser Premisse zu besonders langen Framedurchläufen pro Sekunde. Dazu mehr im folgenden Abschnitt.

4.2 Experimentelle Daten und Resultate

Im Rahmen dieser Arbeit wurde eine Fallstudie durchgeführt, welche den Algorithmus testet und die Performanz in Form von Frames pro Sekunde misst. Sie wurde mit einer AMD Ryzen 7 1800X CPU (3,6GHz auf 8 CPU-Kernen) und einer GTX 1060 GPU (Basistaktrate 1582 MHz, 6GB GDDR5 Speicher) durchgeführt.

Die Fallstudie ist ein Quader aus 5 Tetraedern und 8 Knotenpunkten. Im Folgenden wird eine kleine Tabelle betrachtet, die in den Zellen der ersten Spalte die Anzahl der Tetraeder und Knotenpunkte aufzählt und in den Zellen der zweiten Spalte die Frames pro Sekunde angeben. Jede Zeile gibt eine Frakturstufe des Netzes an.

5 Tetraeder, 8 Knotenpunkte	1800 FPS	
17 Tetraeder, 15 Knotenpunkte	1000 FPS	
65 Tetraeder, 44 Knotenpunkte	300 FPS	
386 Tetraeder, 137 Knotenpunkte	2 FPS	

4.3 Veränderungsvorschläge

5 Ausblick

In diesem Kapitel werden neuere Algorithmen und Systeme betrachtet, die zur Simulation von Frakturen oder ähnlichen physischen Situationen entwickelt wurden.

Literaturverzeichnis

BritFrac99: James F. O'Brien, Jessica K. Hodgins, Graphical Modeling and Animation of Brittle Fracture, ACM Inc. (1999)

DefModels88: Demetri Terzopoulos, Kurt Fleischer, Deformable Models, (1988)

ContMech94: Y. C. Fung, A First Course in Continuum Mechanics, Prentice-Hall (1994)

Werkstoffe11: Eckard Macherauch, Hans-Werner Zoch, Praktikum in Werkstoffkunde, Vieweg+Teubner Verlag (2011)

ForMatEng03: Peter Rhys Lewis, Ken Reynolds, Colin Gagg, Forensic Materials Engineering, CRC Press (2003)

Index

Glossar

Erklärung des Kandidaten

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel „Implementierung der Ductile-Fracture-Bruchsimulation in der Unity Engine“ selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt, keine anderen als die angegebenen Hilfsmittel verwendet und wörtliche sowie sinngemäße Zitate als solche gekennzeichnet habe.

Trier, den 16.09.2020

Paul Froelich
