C:\Users\Manfred Stueber\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Logo_Hauptcampus_Informatik.eps

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, 10.9.2020

Vorwort

Ein Vorwort ist nicht unbedingt nötig. Falls Sie ein Vorwort schreiben, so ist dies der Platz, um z.B. die Firma vorzustellen, in der diese Arbeit entstanden ist, oder einigen Leuten zu danken, die in irgendeiner Form positiv zur Entstehung dieser Arbeit beigetragen haben. Auf keinen Fall sollten Sie im Vorwort die Aufgabenstellung näher erläutern oder vertieft auf technische Sachverhalte eingehen.

Kurzfassung

Die Simulation von Deformierungen und Frakturen von Körpern finden weitflächig Anwendung, beispielsweise in Analyseprogrammen zur Prognose von Bruchstellen bei Bestandsteilen 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, aber es werden auch Urteile zur Willkür der Frakturen gefällt und inwiefern dies zu einer wünschenswerten und sinnvollen Gameplay-Mechanik beiträgt. 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](#_Toc468972122)

[2 Weitere Kapitel 2](#_Toc468972123)

[2.1 Abschnitt 3](#_Toc468972124)

[2.1.1 Unterabschnitt 3](#_Toc468972125)

[2.2 Abbildungen und Tabellen 3](#_Toc468972126)

[2.3 Mathematische Formel 4](#_Toc468972127)

[2.4 Sätze, Lemmas und Definitionen 5](#_Toc468972128)

[2.5 Fußnoten 5](#_Toc468972129)

[2.6 Literaturverweise 5](#_Toc468972130)

[3 Beispiel-Kapitel 6](#_Toc468972131)

[3.1 Warum existieren unterschiedliche Konsistenzmodelle? 6](#_Toc468972132)

[3.2 Klassifizierung eines Konsistenzmodells 6](#_Toc468972133)

[3.3 Linearisierbarkeit (atomic consistency) 7](#_Toc468972134)

[4 Zusammenfassung und Ausblick 9](#_Toc468972135)

[Literatur 10](#_Toc468972136)

[Index 11](#_Toc468972137)

[Glossar 12](#_Toc468972138)

[Erklärung der Kandidatin / des Kandidaten 13](#_Toc468972139)

Abbildungsverzeichnis

[Abbildung 2.1 Bezeichnung der Abbildung 4](#_Toc468972369)

Tabellenverzeichnis

[Tabelle 5.1 Bezeichnung der Tabelle 4](#_Toc349735145)

[Tabelle 6.1 Linearisierbarkeit ist erfüllt 7](#_Toc349735146)

[Tabelle 6.2 Linearisierbarkeit ist verletzt, sequentielle Konsistenz ist erfüllt 7](#_Toc349735147)

[Tabelle 6.3 Linearisierbarkeit und Konsistenz sind verletzt 8](#_Toc349735148)

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 weichen Echtzeitsystemen. Hierbei wird besonders auf die Performanz der Implementierung des Algorithmus geachtet und seine Komplexität festgestellt. 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.

## 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) im Durchschnitt 273 Minuten.

// folgender Paragraph eventuell fehl am Platz

Außerdem 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.

Das der Arbeit zugrundeliegende GMABF-Modell besteht aus 3 Teilen: dem kontinuierlichen Modell, welches auf der Kontinuumsmechanik basiert, dem diskreten Modell, welches die Finite-Elemente-Methode verwendet 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. Danach wird die Remeshing-Vorschrift erläutert.

# Weitere Kapitel

Die Gliederung hängt natürlich vom Thema und von der Lösungsstrategie ab. Als nützliche Anhaltspunkte können die Entwicklungsstufen oder -schritte z.B. der Softwareentwicklung betrachtet werden. Nützliche Gesichtspunkte erhält und erkennt man, wenn man sich

* in die Rolle des Lesers oder
* in die Rolle des Entwicklers, der die Arbeit z.B. fortsetzen, ergänzen oder pflegen soll,

versetzt. In der Regel wird vorausgesetzt, dass die Leser einen fachlichen Hintergrund haben - z.B. Informatik studiert haben. D.h. nur in besonderen, abgesprochenen Fällen schreibt man in populärer Sprache, so dass auch Nicht-Fachleute die Ausarbeitung prinzipiell lesen und verstehen können.

Die äußere Gestaltung der Ausarbeitung hinsichtlich Abschnittformate, Abbildungen, mathematische Formeln usw. wird in Kapitel 0 kurz dargestellt.

Bausteine

Der Text wird in bis zu drei Ebenen gegliedert:

1. Kapitel
2. Unterkapitel
3. Unterunterkapitel

## Abschnitt

Text der Gliederungsebene 2.

### Unterabschnitt

Text der Gliederungsebene 3. Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text

Beispiel für Quelltext:

Process 1:

Acquire();

a := 1;

Release();

...

Acquire();

if (b == 0)

{

c := 3;

d := a;

}

Release();

Prozess2:

Acquire();

b := 1;

Release();

...

Acquire();

if ( a == 0)

{

c := 5;

d := b;

}

Release();

Größere Code-Fragmente sollten im Anhang eingefügt werden.

## Abbildungen und Tabellen

Abbildung und Tabellen werden zentriert eingefügt. Grundsätzlich sollen sie erst dann erscheinen, nach dem sie im Text angesprochen wurden (siehe Abbildung 2.1). Abbildungen und Tabellen (siehe Tabelle 2.1) können im (fließenden) Text, am Seitenanfang, am Seitenende oder auch gesammelt auf einer nachfolgenden Seite oder auch ganz am Ende der Ausarbeitung erscheinen. Letzteres sollte man nur dann wählen, wenn die Bilder günstig zusammen zu betrachten sind und die Ausarbeitung nicht zu lang ist (< 20 Seiten).

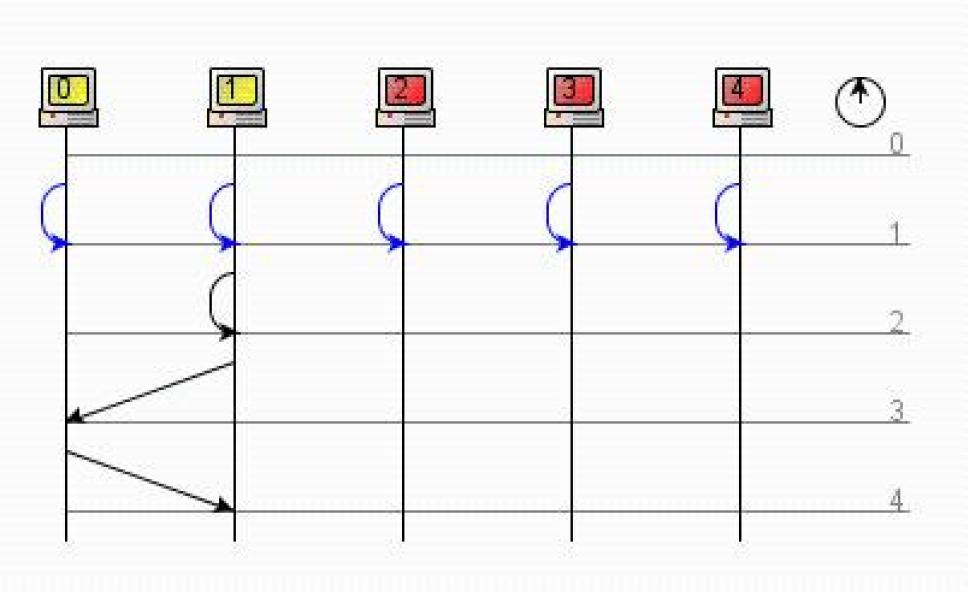


Abbildung 2.1 Bezeichnung der Abbildung

|  |  |
| --- | --- |
| **Prozesse** | **Zeit 🡪** |
|  |  |
|  |  |
|  |  |
|  |  |

Tabelle 2.1 Bezeichnung der Tabelle

## Mathematische Formel

Mathematische Formeln bzw. Formulierungen können sowohl im laufenden Text (z.B.) oder abgesetzt und zentriert im Text erscheinen. Gleichungen sollten für Referenzierungen nummeriert werden (siehe Formel 2.1).



Formel 2.1

Entscheidungsformel:



Formel 2.2

Matrix:



Formel 2.3

Vektor:



Formel 2.4

## Sätze, Lemmas und Definitionen

Sätze , Lemmas, Definitionen, Beweise, Beispiele können auch mit dem Formeleditor erstellt werden.

## Fußnoten

In einer Fußnote können ergänzende Informationen[[1]](#footnote-1) angegeben werden. Außerdem kann eine Fußnote auch Links enthalten. Wird in der Arbeit eine Software (zum Beispiel Java-API[[2]](#footnote-2)) eingesetzt, so kann die Quelle, die diese Software zur Verfügung stellt in der Fußnote angegeben werden.

## Literaturverweise

Auf benutzte Literatur wird im Text verwiesen und kann im Literaturverzeichnis automatisch übernommen werden. Dazu wird in der Menüleiste unter „Verweise“ auf „Zitat einfügen“ geklickt. Dort können neue Quellen angegeben werden. Als Formatvorlage wird „ISO 690“ vorgeschlagen. Dazu gibt es zwei Möglichkeiten: Anzeige des Autors bzw. der Autoren und Erscheinungsjahres (Beispiel: (Coulouris, Dollimore, & Kindberg, 2002)) bzw. Anzeige einer Ziffer. Wird die Anzeige des Autors/der Autoren mit Jahreszahl gewählt und es gibt vom selben Autor mehrere Veröffentlichungen aus dem angegeben Jahr die aufgezählt werden sollen, so soll die Unterscheidung der Quellen durch Hinzufügen einer Nummerierung mit kleinen Buchstaben unterschieden werden, wie beispielsweise (Mustermann, 2013a, S. 43) und (Mustermann, 2013b, S. 89).

Weitere Formatvorlagen als die von Word standardmäßig zur Verfügung gestellten finden Sie unter (CodePlex, 2013) zu finden. Ebenso können Sie die Stile auch selbst anpassen.

# Beispiel-Kapitel

In diesem Kapitel wird beschrieben, warum es unterschiedliche Konsistenzmodelle gibt. Außerdem werden die Unterschiede zwischen strengen Konsistenzmodellen (Linearisierbarkeit, sequentielle Konsistenz und schwachen Konsistenzmodellen (schwache Konsistenz, Freigabekonsistenz) erläutert. Es wird geklärt, was Strenge und Kosten (billig, teuer) in Zusammenhang mit Konsistenzmodellen bedeuten.

## Warum existieren unterschiedliche Konsistenzmodelle?

Laut (Malte, 1997) sind mit der Replikation von Daten immer zwei gegensätzliche Ziele verbunden: die Erhöhung der Verfügbarkeit und die Sicherung der Konsistenz der Daten. Die Form der Konsistenzsicherung bestimmt dabei, inwiefern das eine Kriterium erfüllt und das andere dementsprechend nicht erfüllt ist (Trade-off zwischen Verfügbarkeit und der Konsistenz der Daten). Stark konsistente Daten sind stabil, das heißt, falls mehrere Kopien der Daten existieren, dürfen keine Abweichungen auftreten. Die Verfügbarkeit der Daten ist hier jedoch stark eingeschränkt. Je schwächer die Konsistenz wird, desto mehr Abweichungen können zwischen verschiedenen Kopien einer Datei auftreten, wobei die Konsistenz nur an bestimmten Synchronisationspunkten gewährleistet wird. Dafür steigt aber die Verfügbarkeit der Daten, weil sie sich leichter replizieren lassen.

Nach (Mosberger, 1993) kann die Performanzsteigerung der schwächeren Konsistenzmodelle wegen der Optimierung (Pufferung, Code-Scheduling, Pipelines) 10-40 Prozent betragen. Wenn man bedenkt, dass mit der Nutzung der vorhandenen Synchronisierungsmechanismen schwächere Konsistenzmodelle den Anforderungen der strengen Konsistenz genügen, stellt sich der höhere programmiertechnische Aufwand bei der Implementierung der schwächeren Konsistenzmodelle als ihr einziges Manko dar.

In (Cheriton, 1985) ist beschrieben, wie man sich Formen von DSM vorstellen könnte, für die ein beachtliches Maß an Inkonsistenz akzeptabel wäre. Beispielsweise könnte DSM verwendet werden, um die Auslastung von Computern in einem Netzwerk zu speichern, so dass Clients für die Ausführung ihrer Applikationen die am wenigsten ausgelasteten Computer auswählen können. Weil die Informationen dieser Art innerhalb kürzester Zeit ungenau werden können (und durch die Verwendung der veralteten Daten keine großen Nachteile entstehen können), wäre es vergebliche Mühe, sie ständig für alle Computer im System konsistent zu halten (Coulouris, Dollimore, & Kindberg, 2002). Die meisten Applikationen stellen jedoch strengere Konsistenzanforderungen.

## Klassifizierung eines Konsistenzmodells

Die zentrale Frage, die für die Klassifizierung (streng oder schwach) eines Konsistenzmodells von Bedeutung ist (Coulouris, Dollimore, & Kindberg, 2002): wenn ein Lesezugriff auf eine Speicherposition erfolgt, welche Werte von Schreibzugriffen auf diese Position sollen dann dem Lesevorgang bereitgestellt werden? Die Antwort für das schwächste Konsistenzmodell lautet: von jedem Schreibvorgang, der vor dem Lesen erfolgt ist, oder in der ”nahen“ Zukunft, innerhalb des definierten Betrachtungsraums, erfolgten wird. Also irgendein Wert, der vor oder nach dem Lesen geschrieben wurde.

Für das strengste Konsistenzmodell, Linearisierbarkeit (atomic consistency), stehen alle geschriebenen Werte allen Prozessoren sofort zur Verfügung: eine Lese- Operation gibt den aktuellsten Wert zurück, der geschrieben wurde, bevor das Lesen stattfand. Diese Definition ist aber in zweierlei Hinsicht problematisch. Erstens treten weder Schreib- noch Lese-Operationen zu genau einem Zeitpunkt auf, deshalb ist die Bedeutung von ”aktuellsten“ nicht immer klar. Zweitens ist es nicht immer möglich, genau festzustellen, ob ein Ereignis vor einem anderen stattgefunden hat, da es Begrenzungen dafür gibt, wie genau Uhren in einem verteilten System synchronisiert werden können.

Nachfolgend werden einige Konsistenzmodelle absteigend nach ihrer Strenge vorgestellt. Zuvor müssen wir allerdings klären, wie die Lese- und Schreibe-Operationen in dieser Ausarbeitung dargestellt werden.

Sei  eine Speicherposition, dann können Instanzen dieser Operationen wie folgt ausgedrückt werden:

* - eine Lese-Operation, die den Wert a von der Position x liest.
* - eine Schreib-Operation, die den Wert b an der Position x speichert.

## Linearisierbarkeit (atomic consistency)

Die Linearisierbarkeit im Zusammenhang mit DSM kann wie folgt definiert werden:

* Die verzahnte Operationsabfolge findet so statt: wenn  in der Folge vorkommt, dann ist die letzte Schreib-Operation, die vor ihr in der verzahnten Abfolge auftritt,  , oder es tritt keine Schreib-Operation vor ihr auf und  ist der Anfangswert von . Das bedeutet, dass eine Variable nur durch eine Schreib-Operation geändert werden kann.
* Die Reihenfolge der Operationen in der Verzahnung ist konsistent zu den Echtzeiten, zu denen die Operationen bei der tatsächlichen Ausführung aufgetreten sind.

|  |  |
| --- | --- |
| **Prozesse** | **Zeit 🡪** |
|  |  |
|  |  |

Tabelle 3.1 Linearisierbarkeit ist erfüllt

Die Bedeutung dieser Definition kann an folgendem Beispiel (Tabelle 3.1) nachvollzogen werden. Es sei angenommen, dass alle Werte mit 0 vorinitialisiert sind.

Hier sind beide Bedingungen erfüllt, da die Lese-Operationen den zuletzt geschriebenen Wert zurückliefern. Interessanter ist es, zu sehen, wann die Linearisierbarkeit verletzt ist.

|  |  |
| --- | --- |
| **Prozesse** | **Zeit 🡪** |
|  |  |
|  |  |

Tabelle 3.2 Linearisierbarkeit ist verletzt, sequentielle Konsistenz ist erfüllt

In diesem Beispiel (Tabelle 3.2) ist die Echtzeit-Anforderung verletzt, da der Prozess  immer noch den alten Wert liest, obwohl er von Prozess  bereits geändert wurde. Diese Ausführung wäre aber sequentiell konsistent (siehe kommender Abschnitt), da es eine Verzahnung der Operationen gibt, die diese Werte liefern könnte (, , , ). Würde man beide Lese-Operationen des 2. Prozesses vertauschen, wie in der Tabelle 3.3 dargestellt, so wäre keine sinnvolle Verzahnung mehr möglich.

|  |  |
| --- | --- |
| **Prozesse** | **Zeit 🡪** |
|  |  |
|  |  |

Tabelle 3.3 Linearisierbarkeit und Konsistenz sind verletzt

In diesem Beispiel sind beide Bedingungen verletzt. Selbst wenn die Echtzeit, zu der die Operationen stattgefunden haben, ignoriert wird, gibt es keine Verzahnung einzelner Operationen, die der Definition entsprechen würde.

# Zusammenfassung und Ausblick

In diesem Kapitel soll die Arbeit noch einmal kurz zusammengefasst werden. Insbesondere sollen die wesentlichen Ergebnisse Ihrer Arbeit herausgehoben werden. Erfahrungen, die z.B. Benutzer mit der Mensch-Maschine-Schnittstelle gemacht haben oder Ergebnisse von Leistungsmessungen sollen an dieser Stelle präsentiert werden. Sie können in diesem Kapitel auch die Ergebnisse oder das Arbeitsumfeld Ihrer Arbeit kritisch bewerten. Wünschenswerte Erweiterungen sollen als Hinweise auf weiterführende Arbeiten erwähnt werden.

Literatur

Cheriton, D. R. (1985). Preliminary Thoughts on Problem-oriented Shared Memory: A Decentralized Approach to Distributed Systems. (S. University, Ed.) *Operating Systems Review, 19*(4), pp. 26-33.

CodePlex. (2013, 2). *BibWord*. (CodePlex) Retrieved 2 26, 2013, from http://bibword.codeplex.com/releases/view/15852

Coulouris, G., Dollimore, J., & Kindberg, T. (2002). *Verteilte Systeme: Konzepte und Design.* Addison-Wesley-Verlag.

Malte, P. (1997). (A. Grosse, & D. Kottmann, Editors) Retrieved from Architektur vernetzter Systeme. Seminar SS 1996 & WS 1996/97: Replikation in Mobil Computing: http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=/ira/1997/31

Mosberger, D. (1993). *Memory Consistency Models.* Technical Report, University of Arizona.

Index

A

Abbildung 5

Abschnitt 5

B

Beispiel 7

Beweis 7

D

Definition 7

E

Echtzeiten 9

F

Formel 6

Freigabekonsitenz 8

I

Inkonsitenz 8

K

Kapitel 5

Konsitenz 8

schwach 8

Konsitenzmodelle 8

L

Lemma 7

Linearisierbarkeit 9

Literaturverweise 7

M

Matrix 6

O

Operation

Lese 9

Schreib 9

Optimierung 8

Q

Quelltext 5

S

Satz 7

schwach 8

sequentiell

Konsitenz 8

streng 8

T

Tabelle 5

U

Unterkapitel 5

V

Vektor 7

Verfügbarkeit 8

Glossar

|  |  |
| --- | --- |
| DisASTer | DisASTer (Distributed Algorithms Simulation Terrain) A platform for the Implementation of Distributed Algorithms |
| DSM | Distributed Shared Memory |
| AC | Linearisierbarkeit (atomic consistency) |
| SC | Sequentielle Konsistenz (sequential consistency) |
| WC | Schwache Konsistenz (weak consistency) |
| RC | Freigabekonsistenz (release consistency) |

Erklärung der Kandidatin / des Kandidaten

* Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.
* Die Arbeit wurde als Gruppenarbeit angefertigt. Meine eigene Leistung ist im Kapitel „Verantwortliche“ zu Beginn der Dokumentation aufgeführt.

Diesen Teil habe ich selbstständig verfasst und keine anderen als die angegebenen   
 Quellen und Hilfsmittel verwendet.

Name der Mitverfasser: ....................................................................................................  
 ...........................................................................................................................................

|  |  |  |
| --- | --- | --- |
| Datum |  | Unterschrift der Kandidatin / des Kandidaten |

1. Informationen, die für die Arbeit zeitrangig sind, jedoch für den Leser interessant sein können [↑](#footnote-ref-1)
2. http://java.sun.com [↑](#footnote-ref-2)