Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Philipp Kayser

Entwicklung und Integration einer GPU-Unterstützung für MARS

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Philipp Kayser

Entwicklung und Integration einer GPU-Unterstützung für MARS

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Clemen
Zweitgutachter: Prof. Dr. Wolfgang Fohl

Eingereicht am: 12. Juni 2018

Philipp Kayser

Thema der Arbeit
Entwicklung und Integration einer GPU-Unterstützung für MARS

Stichworte
Multi-Agenten Simulation, MARS, Kollisionserkennung, GPU, GPGPU

Kurzzusammenfassung
Immer größere Szenarien sollen in Multi-Agenten Simulationen abgebildet werden. Dadurch entstehen immer härtere Anforderungen an die Performanz der Ausführungsumgebung. Das MARS System ist darauf ausgelegt diese großen Szenarien simulieren zu können. Bisher stellt die Kollisionserkennung ein stark begrenzendes Element in MARS dar. Auf dieses Problem wird mit dieser Arbeit eingegangen, indem eine auf Multi-Agenten Simulationen ausgelegte Kollisionserkennung auf GPU-Basis entwickelt und in MARS integriert wird.
Philipp Kayser

Title of the paper
Development and integration of an GPU-Assistance for MARS

Keywords
Multi-Agent Simulation, MARS, Collision detection, GPU, GPGPU

Abstract
Ever larger scenarios are to be mapped in multi-agent simulations. This results in ever tougher demands on the performance of the execution environment. The MARS system is designed to simulate these large scenarios. Until now, collision detection has been a very limiting element in MARS. This work addresses this problem by developing a GPU-based collision detection designed for multi-agent simulations and integrating it into MARS.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **4**
2.1 MARS . . . 4
2.2 GPU-Programmierung . . . 6
2.3 Related Work . . . 11
2.3.1 Bounding Volume Hierarchies . . . 11
2.3.2 Spatial Subdivision . . . 13
2.3.3 Weitere Ansätze . . . 15

3 Analyse **17**
3.1 Annahmen . . . 17
3.2 Anforderungen . . . 17
3.2.1 Environment mit Kollisionserkennung auf GPU-Basis . . . 18
3.2.2 Kollisionserkennung im Multi-Agent-Simulation Kontext . . . 18
3.2.3 Erweiterte Integration . . . 19
3.3 Schnittstellen und relevante Komponenten für die Integration . . . 20

4 Konzept **23**
4.1 Integration . . . 23
4.2 Environments mit GPU-Basierter Kollisionserkennung . . . 24
4.2.1 Single Layer 3D Grid-Environment . . . 24
4.2.2 Multi Layer 2D Grid-Environment . . . 26

5 Realisierung **31**
5.1 Single Layer 3D Grid-Environment . . . 31
5.2 Multi Layer 2D Grid-Environment . . . 33
5.2.1 Reorder und Cellarrays . . . 34
5.2.2 Sortieren und Erstellen der Kollisionssublisten . . . 37
5.2.3 Permutationen und Kollisionsprüfungen . . . 38
5.2.4 Aufbereitung und Aufruf der Delegaten . . . 39
5.3 Radix-Sort und Scan . . . 40

6 Integration **43**
6.1 Async Environment Interface . . . 43
6.1.1 Synchron zu asynchron Wrapper . . . 46
6.2 Async Agents . . . 47

6.3 Beispielmodell Antelopes and Lions 48
6.4 Ausführungsumgebung 50
6.4.1 Portierung der OpenCL Library zu .NET Core 51
6.4.2 Notwendige Anpassungen bei der Docker Umgebung 52

7 Experimente **54**
7.1 Vergleichstests der Environments 54
7.1.1 Elemente mit konstanter Größe 55
7.1.2 Elemente mit konstanter Größenvariation 55
7.1.3 Elemente mit variierender Größenvariation 59
7.2 Antelopes and Lions 60

8 Diskussion **65**
8.1 Analyse der Experimente 65
8.2 Gegenüberstellung von Hypothesen und Ergebnissen 67

9 Ausblick **68**
9.1 Permutationsbildung 68
9.2 Speichermanagement 69
9.3 Verteilung 70

10 Fazit **71**

Tabellenverzeichnis

4.1 Zuordnung der Elemente in Home- und Phantomcells aus Abbildung 4.1 . . . 25

5.1 Nötige Kollisionsprüfungen im eindimensionalen Gridfeld 36
5.2 Nötige Kollisionsprüfungen im hierarchischem Gridfeld. Der Prefix N_ steht für native Zellen und I_ für importierte aus den unteren Ebenen. Prüfungen müssen nur von den nativen Zellen ausgehend durchgeführt werden 36

6.1 Auflistung der Attribute des Lion Agenten 49
6.2 Auflistung der Attribute des Antilope Agenten 50
6.3 Auflistung der Attribute des Grass Agenten 50

Abbildungsverzeichnis

2.1 Architekturvergleich zwischen GPU und CPU [32]. Links die CPU mit vier Kernen(ALUs), großen Cache und großer Kontrolleinheit. Rechts die GPU, die mit einem zentralem DRAM und mehren Arbeitsgruppen (Compute Units) bestehend aus jeweils einer Kontrolleinheit einem Cache und mehreren ALUs 6
2.2 OpenCL Memory Architektur [22] . . . 8
2.3 Ein Beispiel für BVH mit mehreren Ebenen . . . 12
2.4 Beispiel für eine Spatiale Subdivision mit einer Gridebene . . . 14
2.5 Bild einer ungleichen Verteilung von Objekten und die Anpassung des Gridfeldes entsprechend der Objektdichte [43] . . . 15

3.1 Struktur der Basic Agents [3] . . . 21

4.1 Beispiel eines Gridfeldes, auf Tabelle 4.1 sind Zuordnung der Elemente und die benötigten Kollisionsprüfungen dargestellt . . . 25
4.2 Darstellung eines Szenarios mit explorierenden Agenten in einem hierarchischen Gridfeld . . . 27
4.3 Darstellung eines hierarchischen Gridfeldes mit Bestimmung der Kollisionsprüfungen . . . 29

5.1 Einfügen der Objekte in das Cellid Array [24] . . . 32
5.2 Beispiel für das Erstellen der Kollisionssublisten auf der GPU . . . 33
5.3 Aufteilung der Elemente in den Cellarrays . . . 37
5.4 Radix-Sort Beispiel von Preiss[34] . . . 41
5.5 Eigene Implementierung der OpenCL Präfixsumme nach Blelloch [8] . . . 42

6.1 Abhängigkeitsgraph für das async Agents Paket . . . 48

7.1 Vergleich der Kollisionserkennung der verschiedenen Environments bei Kreisobjekten mit einem Radius von 2 . . . 56
7.2 Vergleich der Kollisionserkennung der verschiedenen Environments bei Kreisobjekten mit einem variierendem Radius zwischen 2 und 10 . . . 57
7.3 Vergleich der Kollisionserkennung der verschiedenen Environments bei Kreisobjekten mit einem variierendem Radius zwischen 2 und 30 . . . 58
7.4 Vergleich der Kollisionserkennung der verschiedenen Environments bei einer konstanten Anzahl an Elementen und einer steigenden Größenvarianz . . . 59

7.5 Vergleich der verschiedenen Environments beim Antelopes und Lions Modell. Gemessen wurde jeweils die Zeit, die MARS Life für die Ausführung eines Ticks benötigt hat 61
7.6 Direkter Vergleich der Environments beim Antelopes und Lionsmodell. In diesem Graph wird die Zeit verglichen, die durchschnittlich zur Abhandlung des Environment Commits benötigt wurde 62
7.7 Vergleich der Ausführungszeit eines Simulationsticks zwischen dem Antelopes and Lions Modell mit den asynchronen Environment und dem Wolves and Sheep Modell mit dem synchronen Gridenvironment 63

Listings

2.1 Beispiel für einen OpenCL Kernel 10

1 Einleitung

Multi-Agent-Simulation (MAS) werden heutzutage in vielen Domänen verwendet. Dazu gehört unteren anderem die Umweltforschung und Biologie.

Viele der Anwendungsbereiche wie zum Beispiel die Sozialökologie benötigen eine große Menge an Agenten, die miteinander agieren können, um möglichst viele Erkenntnisse aus der Simulation ziehen zu können. Betrachtet aus Sicht der Informatik werden viele Ressourcen benötigt, um eine Simulationen in diesen Anwendungsbereichen durchführen zu können. Das führt dazu, dass bei einer Ausführungsumgebung für MAS viel Wert auf Performanz gelegt wird. Ein wichtiger Aspekt ist dabei die Kollisionserkennung, die in der MAS vielseitige Anwendung findet. Dadurch, dass die benötigte Anzahl an Kollisionsprüfungen stark mit der Anzahl der Agenten steigt, kann sie schnell den Flaschenhals der Ausführungsumgebung darstellen. Diese Arbeit befasst sich damit eine Kollisionserkennung auf Graphic Processing Unit (GPU)-Basis zu entwickeln, die speziell auf die besonderen Anforderungen einer MAS optimiert ist. Neben dem Entwurf und Konzeption wird ebenfalls die vollständige Integration der Kollisionserkennung in das Simulations-Framework MARS Life beschrieben.

Es ist wichtig, die Verwendung und Bedeutung bestimmter Begriffe zu klären, wenn über Simulationssysteme und Modellierung gesprochen wird. Das übergeordnete Thema wird als Agent-based Computing [44] bezeichnet und beschreibt das Agenten-basierte System im Allgemeinen. Dies beinhaltet alles Agenten bezogene und geht von der Automatisierung bis hin zur Simulation von Evakuierungsszenarien. Im Allgemeinen wird das Agent-based Computing als ein Weg verstanden, reale Probleme in den Informatik Kontext zu übersetzen. Agenten werden als ein Lösungsansatz für verschiedene Probleme angesehen, die von Softwareentwicklung bis hin zur konzeptionellen Modellierung reichen.

Eine spezielle Ausprägung der Modellierung von Agenten basierten Simulationen ist das individual-based-modeling, das von Grimm und Railsbek [15] ins Leben gerufen wurde. Dort werden die Agenten als eigenständig handelnde Individuen betrachtet und damit deutlich weniger abstrakt als im Konzept des Agenten-based Computing, bei dem ein Agent alles Mögliche darstellen könnte, das autonom und unabhängig in seiner Umgebung arbeitet.

Die Multi Agent Research and Simulation (MARS) Group der HAW Hamburg hat mit MARS Life ein System zu Modellierung und Ausführung von Multi-Agenten Simulationen entwickelt. Es ist darauf ausgelegt Individual-Based Models zu simulieren und legt den Fokus auf Simulationen mit einer hohen Agentenanzahl.

Diese Arbeit wurde im Rahmen der MARS Group durchgeführt, mit dem Ziel das MARS Life System durch den Einsatz der GPU zu optimieren.

Sobald von stark parallelen Problemen gesprochen wird, rückt schnell die GPU in den Fokus. Dessen Architektur ist auf das Lösen von parallelen Aufgabenstellungen ausgelegt, anders als die Central Processing Unit (CPU) welche sich auf sequentielle Problemstellungen fokussiert. Damit ist die GPU in der Lage gut parallelisierbare Aufgaben schneller zu berechnen [32].

Das die GPU und MAS im Ansatz harmonieren wurde bereits in einigen Arbeiten gezeigt [21][35][18]. Allerdings handelt es sich dabei um voll integrierte Ansätze, bei denen das gesamte Simulationsmodell auf die GPU portiert wurde. Dadurch ist es zwar möglich die Ausführungszeit der Modelle stark zu reduzieren, jedoch wird für die Anpassung oder das Erstellen eines Modells spezielle Kenntnis über GPU-Programmierung benötigt.

MARS Life verfolgt das Ziel auch dem Domainexperten, eine Plattform zum Entwickeln des eigenen Modells zu bieten. Im Kontext des Modells hat dieser fundierte Fachkenntnisse, jedoch hat er oft keinen Bezug zur Informatik. Deswegen sollten keine Kenntnisse über GPU-Programmierung vorausgesetzt werden. Um eine Lösung schaffen zu können, die für alle Nutzer des MARS Frameworks verwendbar ist, wird der Fokus auf ein Teilproblem gelegt, das in vielen Szenarien auftritt, die Kollisionen. Das ermöglicht es, die GPU in einer Teilkomponente zu kapseln und damit eine nahtlose Integration vorzunehmen.

Kollisionen stellen ein großes Problem bei MAS dar, welches viel Performanz benötigt [17]. Über sie wird ein Großteil der Interaktion zwischen den Agenten realisiert, dazu zählt unter Anderem das direkte Kollidieren unter Agenten, als auch das Auffinden anderer Agenten in der Nähe. Das Kollisionspotential steigt in der Regel linear zu der Agentenanzahl, weswegen es bei Szenarien mit vielen Agenten zu Performanzeinbußen kommen kann. In dem Kontext kann die GPU gut eingesetzt werden, um dieses Problem anzugehen [27]. Dazu gibt Arbeiten zur Kollisionsvermeidung [17] im MAS Kontext und verschiedene Ansätze zur GPU-basierten Kollisionserkennung im Allgemeinen [11][42].

Im Kontext von MARS Life ist das Problem der Kollisionserkennung ebenfalls bereits bekannt. Christian Hüning hat im Rahmen seiner Masterarbeit Performanzmessungen mit MARS Life vorgenommen, wobei sich gezeigt hat, dass die Kollisionserkennung in einigen Szenarien mehr als ein Drittel der Simulationszeit einnimmt[19].

Dieses Problem soll mit dieser Arbeit angegangen werden. Dazu werden die Methoden der GPU-basierten Kollisionserkennung und MAS zusammengelegt. Damit soll eine Kollisionserkennung entwickelt werden, welche speziell auf die Anforderungen einer MAS zugeschnitten ist. Um den Domainexperten dieses Tool anbieten zu können, soll es nahtlos in das MARS Life framework integriert werden, sodass keine weiteren Kenntnisse seitens der Modellierer erforderlich sind. Weiterhin soll durch ein Beispielmodell entworfen werden, um einen Vergleich der Kollisionserkennungen im Simulationskontext zu ermöglichen.

Nachfolgend wird ein Überblick über den Aufbau der Arbeit gegeben. Zu Beginn wird in den Grundlagen auf die MARS Group und das MARS Life Framework eingegangen. Um einen besseren Einstieg zu erlauben, wird danach der Aufbau und die Programmierung der GPU erläutert. Einen Überblick über andere Arbeiten zum Thema Kollisionserkennung bietet das Kapitel Related Work. Nachdem die Grundlagen beschrieben wurden, stellt die Analyse die Ziele der Arbeit und die damit verbundenen Anforderungen vor. Im Rahmen dessen wird ein Blick auf die betroffenen Komponenten des MARS Life Frameworks geworfen. Der Analyse folgt das Konzept, welches sowohl auf die Kollisionserkennung als auch auf die Integration eingeht. In der Realisierung wird erläutert, wie das Konzept im Rahmen des MARS Life Systems umgesetzt wurde, um anschließend im Abschnitt Integration genauer darauf einzugehen, wie die Komponente ins System eingegliedert wird. Dabei wird auch das Antelopes and Lions Modell vorgestellt, welches speziell entworfen wurde, um die neue Kollisionserkennung direkt mit den vorherigen Ansätzen im Rahmen eines Modells, vergleichen zu können. Die Experimente werden darauf folgend beschrieben und zeigen sowohl einen direkten Vergleich der Komponenten als auch deren Performanz im Antelopes and Lions Modell.

Deren Ergebnisse werden im Kapitel Diskussion kritisch betrachtet und in den Vergleich zu den in der Analyse aufgestellten Hypothesen gezogen. Zuletzt bilden der Ausblick und das Fazit den Abschluss der Arbeit.

2 Grundlagen

In diesem Kapitel werden die Komponenten vorgestellt, welche die Basis für diese Arbeit darstellen. Zu Beginn wird die MARS Group und das MARS Life System vorgestellt. Nach der Beschreibung des Kontextes und der Umgebung wird ein Blick auf die Struktur und Funktionsweise von GPU-Programmen geworfen. Um einen besseren Überblick zu geben, wird dort ebenfalls auf die Architekturunterschiede zwischen CPU und GPU eingegangen. Zum Abschluss der Grundlagen gibt Abschnitt 2.3 einen Überblick über bisherige Arbeiten zu dem Thema.

2.1 MARS

Diese Arbeit findet im Rahmen der Multi Agent Research and Simulation (MARS) Group der HAW Hamburg statt. Das zentrale Element der MARS Group stellt das MARS Life Framework dar [20]. Damit wurde eine Plattform geschaffen, über die MAS modelliert und simuliert werden können.

MARS Life besteht jedoch nicht aus einem einzelnen Programm, sondern stellt vielmehr einen Service zur Simulation und Modellierung dar, welcher aus vielen Teilkomponenten besteht. In dem Rahmen verfolgt es den Ansatz des Modeling and Simulation as a Service (MSaaS) [7], bei dem sowohl Modellierung als auch das Simulieren als Service zur Verfügung gestellt werden. Von MSaaS Systemen wird erwartet, dass sie eine einfache Benutzerschnittstelle liefern, sodass auch die Domainexperten ohne technischen Bezug damit arbeiten können [2]. Dafür wird unter Anderem eine Websuite angeboten, über die von überall auf die angebotenen Dienste des MARS Life Systems zugegriffen werden kann. Der Funktionsumfang umfasst sowohl die Ausführung als auch die Parametrisierung der Modelle und die Betrachtung der Ergebnisse.

Über die letzten Jahre ist das MARS Life System immer weiter gewachsen. Neben dem MARS Life Core, welcher für die Ausführung der Simulationen zuständig ist, wurden diverse weitere Komponenten entwickelt, um die Arbeit mit dem MARS Life Framework zu verbessern.

Dazu zählt eine 3D Visualisierung, die im Jahre 2017 von Jan Dalski vorgestellt wurde [10], welche es erlaubt den Verlauf der Simulation in einer 3D Ansicht zu betrachten. Zudem kommt das MARS Meta Modell (MMM) [14], welches eine Abstraktion eines Modelles in

Form einer Grundstruktur für eine Simulation darstellt. Nach Erstellung kann das MMM zu einem ausführbaren Modell im MARS Life Kontext transformiert werden. Für die Erstellung eines MMM wurde eine eigene Domain Specific Language(DSL) entworfen, die MARS DSL [14].

Allgemein wurde ein starker Fokus darauf gelegt, dem Domainexperten selbst das Modellieren zu ermöglichen. Die neue Kollisionserkennung soll mit diesem Ansatz nicht brechen und deswegen ist das Ziel diese nahtlos in das MARS Life System zu integrieren. Aus dem Grund wird nachfolgend der MARS Life Core genauer betrachtet, um mögliche Schnittstellen aufzuzeigen und notwendige Integrationsschritte zu zeigen.

Der MARS Life Core ist für das Ausführen der Modelle zuständig. Dieser umfasst eine Vielzahl an Komponenten, von denen zwei näher betrachtet werden sollen. Dabei handelt es sich dabei um das Environment, welches für alles was die räumliche Position des Agenten betrifft, verantwortlich ist und die Basic Agents, welche eine Basisimplementierung für Agenten darstellen.

Das Environment ist im MARS Life Kontext überwacht und evaluiert jede Bewegung der Agenten. Weiterhin bietet es den Agenten die Möglichkeit die Umgebung zu erkunden und so andere Agenten zu finden. Im MARS Kontext wird dabei vom Explorieren gesprochen, dies wird genutzt um mit anderen Agenten zu interagieren.

Sowohl die Evaluierung der Bewegung als auch das Explorieren laufen auf eine Kollisionserkennung hinaus. Die Evaluierung muss dem Agenten mitteilen, wenn er kollidiert und um ein Gebiet zu explorieren wird eine Kollisionsprüfung auf diesem Gebiet durchgeführt. Damit übernimmt das Environment im MARS Kontext alle Aufgaben, die eine Kollisionserkennung benötigen. Dadurch stellt sie den Punkt dar, an dem die neue GPU-Basierte Kollisionserkennung eingegliedert werden muss, um eine nahtlose Integration zu gewährleisten. Es soll folge dem ein neues Environment mit einer GPU-Basierten Kollisionserkennung entworfen werden.

Die zweite Komponente, auf die eingegangen wird, sind die Basic Agents. Mit den Basic Agents wird eine Basisimplementierung für einen Agenten angeboten, die einige der komplexeren Operationen kapselt und über eine reduzierte Schnittstelle anbietet. Sie dienen dazu dem Modellentwickler eine vereinfachte Verwendung des MARS Systems zu ermöglichen. Dazu zählt die Interaktion mit dem Environment und die Ausgabe der Ergebniswerte. Diese vereinfachte Schnittstelle muss dem Modellentwickler ebenfalls mit dem neuen Environment mit GPU-basierter Kollisionserkennung zur Verfügung stehen.

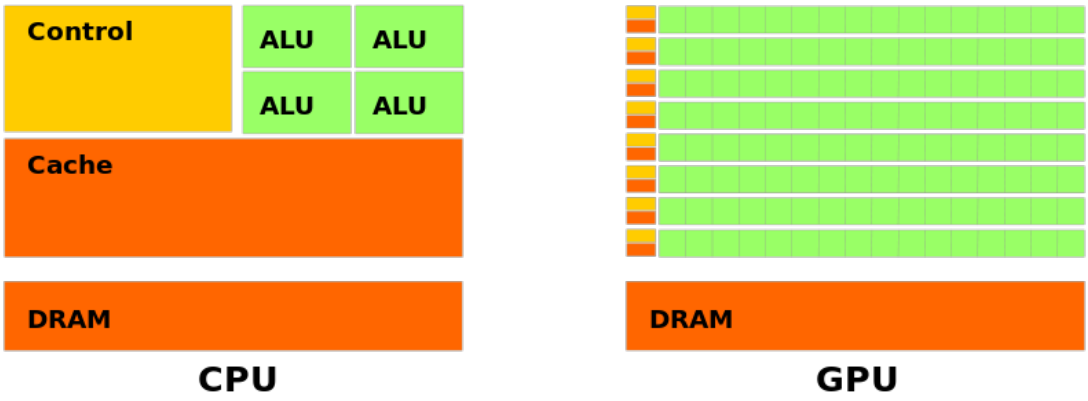

Abbildung 2.1: Architekturvergleich zwischen GPU und CPU [32]. Links die CPU mit vier Kernen(ALUs), großen Cache und großer Kontrolleinheit. Rechts die GPU, die mit einem zentralem DRAM und mehren Arbeitsgruppen (Compute Units) bestehend aus jeweils einer Kontrolleinheit einem Cache und mehreren ALUs

Zusammengefasst muss die neue Kollisionserkennung im Rahmen des Environments in MARS Life integriert werden. Weiterhin soll die Basisimplementierung eines Agenten, welche aktuell über die Basic Agents Komponente angeboten wird auch weiterhin verwendbar sein.

Nachdem in diesem Kapitel das MARS Life Framework und für die Integration relevanten Komponenten vorgestellt wurden, geht das nächste Kapitel auf die Grundlagen der GPU Programmierung ein, um einen Überblick über die Möglichkeiten der GPU zu geben und gleichzeitig die damit verbundenen Einschränkungen aufzuzeigen.

2.2 GPU-Programmierung

Zum Einstieg wird in diesem Kapitel beschrieben, welche Vorteile die GPU hat und wie diese bestmöglich genutzt werden können. Als Informationsbasis für den gesamten Abschnitt dient das Buch OpenCL in Action von Scarpino [36]. Die GPU zeigt ihre Stärke bei stark parallelisierbaren Aufgaben. Das folgt daraus, dass die GPU ursprünglich darauf ausgelegt war Bildberechnungen auszuführen, die größtenteils auf Matrixberechnungen basieren. Ein Monitor hat mehrere Millionen Pixel, für die jeweils die Matrixberechnungen durchgeführt werden müssen. Matrixberechnungen zeichnen sich dadurch aus, dass sie sehr gut parallel berechnet werden können. Um dieser Aufgabe gerecht werden zu können besitzt die GPU eine Architektur die auf parallele Berechnungen ausgelegt ist. Dazu haben sie oft mehrere hundert Rechenkerne, die jedoch jeweils im direkten Vergleich zu einen CPU Kern, langsamer sind.

Abbildung 2.1 zeigt die Architekturunterschiede zwischen GPU und CPU. Die CPU hat einen großen Cache Speicher, eine komplexe Kontrolleinheit und wenige aber schnelle Kerne. Cache und Kontrolleinheit dienen dazu, die Rechenkerne bestmöglich auszulasten und deren Durchsatz zu optimieren.
Bei Betrachtung der GPU fällt auf, dass der Cache deutlich kleiner ausfällt und sich auf mehrere Gruppen von Rechenkernen, die sogenannten Compute Units (CUs), aufgeteilt. Auch die Kontrolleinheit fällt kleiner aus und verteilt sich auf die CUs. Die GPU Architektur sieht es vor, dass die Rechenkerne Gruppenweise über die CUs angesteuert werden. Das heißt, dass jeder Rechenauftrag, der zur GPU gesendet wird, immer von mehreren Rechenkernen parallel berechnet wird.

Eine CPU ist, aufgrund der optimierten, schnellen Kerne bei sequentiellen Berechnungen schneller als eine GPU. Bei parallelen Aufgaben erzielt die GPU jedoch, durch die höhere Anzahl an Kernen und der parallelen Adressierungsmöglichkeiten, geringere Ausführungszeiten [28].

Lange Zeit war es so, dass die GPU eine sehr spezifische Schnittstelle hatte, welche direkt auf Bildbearbeitung und Matrixberechnungen ausgelegt ist. GPU-Programmierung ist damit nur dann möglich, wenn das Problem so umgestellt wird, dass es in die entsprechenden Strukturen passt und mit den Matrixoperationen gelöst werden kann. Dies änderte sich als im Jahr 2007 das General Purpose GPU (GPGPU) Framework CUDA [31] erschienen ist, auf das im Jahr 2009 OpenCL[37] folgte. OpenCL und CUDA sind Frameworks welche es erlauben die GPU zu programmieren, als wäre sie ein Multiprozessor. CUDA wurde von Nvidia entwickelt und wird zurzeit nur von Nvidia Grafikkarten unterstützt. Eine herstellerunabhängige Schnittstelle stellt OpenCL dar, welches von Apple entwickelt wurde und aktuell von der Khronos Group verwaltet wird. Neben Grafikkarten von AMD und Nvidia unterstützt OpenCL ebenfalls diverse CPUs. Auch wenn die Frameworks von verschiedenen Herstellern entwickelt wurden, haben sie eine nahezu identische Struktur. Es ist möglich Architekturen und Programme, die auf CUDA Basis entwickelt wurden, auf OpenCL zu portieren und umgekehrt. Zu beachten ist, dass CUDA mehr Abstraktionsschichten bietet und über eine größere Community verfügt als OpenCL. Aufgrund dessen, dass im Projekt Hardwareunabhängigkeit gefordert ist, wurde OpenCL für die Entwicklung verwendet. Trotz des sehr ähnlichen architektonischen Aufbaus und der Portabilität, verwenden OpenCL und CUDA unterschiedliche Notationen. So wird beispielsweise ein Rechenkern in CUDA als Thread bezeichnet und bei OpenCL als Work-Item. Nachfolgend wird für die Erläuterungen ausschließlich die OpenCL Notation verwendet.

Ein OpenCL Programm besteht immer aus zwei Teilen. Der Host stellt den verwaltenden Part einer OpenCL Applikation dar. Zu seinen Aufgaben gehört es Speicherübertragungen vorzunehmen, Kernelprogramme zu kompilieren und Command Queues zu verwalten. Sämtlicher

Datenaustausch zwischen GPU und Arbeitsspeicher wird explizit durch den Host durchgeführt. Das umfasst sowohl die Übertragung von Kernelargumenten als auch das Schreiben und Lesen von Buffern. Die Command Queues stellen die Kommunikationsschnittstelle zwischen Host und Device (GPU) dar, über diese können Methoden aufgerufen werden oder Bufferübertragungen initiiert werden. Nativ unterstützt der Host die Programmiersprachen C und C++.

Der Kernel bildet den zweiten Teil einer OpenCL Applikation. Er ist der exekutive Part und wird direkt auf dem Device, der GPU, ausgeführt. Der Kernel beinhaltet eine Anzahl an Kernelmethoden, welche vom Host aus aufgerufen werden können. Programmiert werden kann auf Kernelebene in OpenCL C, einer C basierten Programmiersprache, die keine Rekursion erlaubt, dafür aber einige Sonderfunktionen enthält, die ein besseres Mapping auf die OpenCL Architektur erlauben.

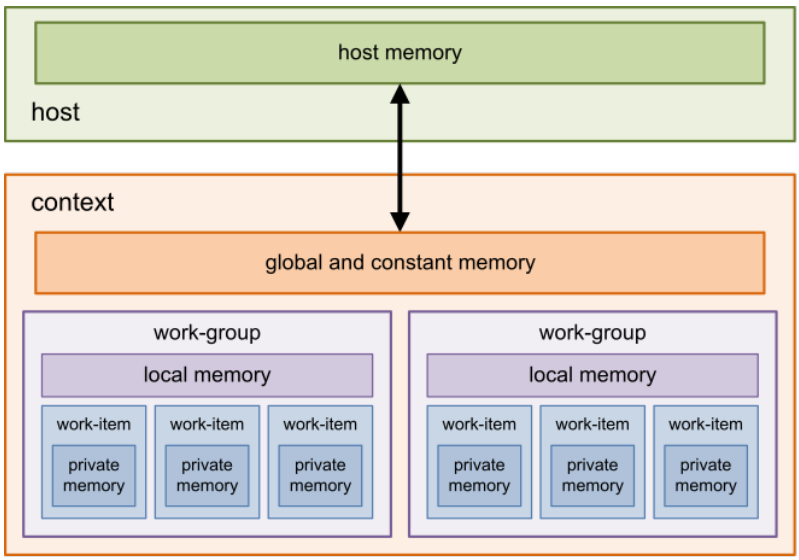

Abbildung 2.2: OpenCL Memory Architektur [22]

Einen ersten Einblick, was bei der Erstellung einer performanten OpenCL Applikation beachtet werden muss, gibt die Memory Architektur, welche in Abbildung 2.2 dargestellt ist. Die GPU besitzt mehrere Speicherebenen, die unterschiedlich schnell sind und auf die nur unter bestimmten Bedingungen zugegriffen werden kann. Direkt vom Host erreichbar ist nur der Global Memory. Dieser ist vergleichsweise langsam, kann aber von allen CUs erreicht werden. Eine GPU hat mehrere CUs, welche in OpenCL als Workgroups bezeichnet werden. Jede Workgroup beinhaltet mehrere GPU Cores. Die Speicherebene hinter dem Global Memory ist der Local Memory, welchen sich die Threads (Work-Items) einer Workgroup teilen. Dieser

ist deutlich schneller, kann aber nicht direkt vom Host beschrieben werden. Auf den Local Memory einer Workgroup können deren Work-Items zugreifen. Zuletzt folgt noch der private Memory welchen jedes Work-Item selbst verwaltet.

Wenn ein Programm gut auf der GPU skalieren soll, muss es in unabhängige Aufgaben unterteilt werden, die von einer Workgroup auf dem Local Memory bearbeitet werden kann. Dies funktioniert im Idealfall so, dass die Threads in der Workgroup zu Programmstart die Daten vom Global Memory auf den Local Memory übertragen, die Berechnungen dort durchführen und zuletzt die berechneten Ergebnisse wieder zurück in den Global Memory schreiben, von dem der Host sie auslesen kann.

```
// Quadrieren der Arraywerte in C
for(int i = 0;  i < size ; i++)
{
  result[i] = input[i] * input[i];
}
-----------------------------------------------------
// Host Teil des OpenCL Codes
...
// Zuweisung von Buffer zu Kernelargument
Cl.SetKernelArg(ckSquare, 0, ptrSize, cl_InputMem);
Cl.SetKernelArg(ckSquare, 1, ptrSize, cl_OutputMem);
Cl.SetKernelArg(ckSquare, 2, ptrSize, cl_Cellids);
// Starten des Kernels
Cl.EnqueueNDRangeKernel(cqCommandQueue, ckSquare, 1, null,
                        workGroupSizePtr, localWorkGroupSizePtr,
                        0, null, out clevent);

-----------------------------------------------------
// Quadrieren im Kernel mit OpenCL C
__kernel void square( __global float* input,
                      __global float* output,
                      const unsigned int count)
{
  // Ermitteln des zu bearbeitenden Arrayslots
  int i = get_global_id(0);
  // Abbruchbedingung pruefen und Operation durchfuehren
  if(i < count)
    output[i] = input[i] * input[i];
}
```

Listing 2.1: Beispiel für einen OpenCL Kernel

Der obige Codeausschnitt zeigt ein Minimalbeispiel eines Kernels. Er verdeutlicht, dass in diesem Kontext anders vorgegangen werden muss, als beim sequentiellen Programmieren. Anstatt über ein Array zu iterieren, wird eine Kernelmethode vielfach aufgerufen und enthält jeweils nur die Berechnungen für einen Iterationsschritt. Der *EnqueueNDRangeKernel* Aufruf des Hosts gibt dabei die Größe des Arrays an, wodurch anschließend entsprechend viele Work-Items auf der GPU gestartet werden. Mit dem Befehl werden die Parameter zur GPU übertragen und die Ausführung des Kernels wird gestartet. Es wird also jeder Iterationsschritt von einem einzelnen Thread durchgeführt, anstatt jeden Schritt sequentiell zu durchlaufen.

Nach dem ersten Einblick in die GPU Programmierung wird im nachfolgenden Abschnitt ein Überblick über andere Arbeiten zum Thema GPU-Basierte Kollisionserkennung gegeben.

2.3 Related Work

Für das MARS Life Framework soll eine neue GPU-basierte Kollisionserkennung entworfen und im Rahmen eines neuen Environments integriert werden. Dazu werden in diesem Kapitel andere Arbeiten zur GPU-basierten Kollisionserkennung vorgestellt, um den Stand der Technik herauszustellen. Zur Integration in das MARS Life Framework können keine verwandten Arbeiten vorgestellt werden, da diese stark auf die MARS Life Architektur angepasst ist.

Nachfolgend werden in diesem Abschnitt einige Ansätze zur GPU basierten Kollisionserkennung vorgestellt. Es wird die Kollisionserkennung dabei in zwei Phasen aufgeteilt. Das ist zum einen die *Broadphase* und zum anderen die *Narrowphase*. In der *Broadphase* werden mit schnellen Tests, im Vorwege möglichst viele Kollisionen ausgeschlossen. Als *Narrowphase* wird die konkrete Kollisionsprüfung zwischen den Objekten bezeichnet, die in der *Broadphase* nicht gefiltert werden konnten. Da für die MAS in vielen Szenarien keine detailreichen Agentenformen notwendig sind, liegt der Fokus auf der *Broadphase*.

Die Recherche hat ergeben, dass sich der Großteil der Arbeiten zur *Broadphase* in zwei verschiedene Schemata aufteilen lässt, mit denen sich eine Kollisionserkennung auf der GPU umsetzen lässt. Zum einen sind das Bounding Volume Hierachies (BVH) und zum anderen die Spatial Subdivision.

Diese beiden Vorgehen werden nachfolgend vorgestellt und verglichen, um anschließend den für die Umsetzung gewählten Ansatz im Detail zu erläutern. Im Anschluss daran, werden weitere Arbeiten präsentiert, die nicht direkt einem der beiden Schemata zugeordnet werden können. Ansätze zur Verteilung werden daraufhin beleuchtet um zuletzt auf die Besonderheiten einer Kollisionserkennung im MAS Kontext einzugehen.

2.3.1 Bounding Volume Hierarchies

BVH sind ein Ansatz zur Kollisionserkennung, der auch bei Algorithmen für die CPU eingesetzt wird.

Abbildung 2.3 zeigt die Visualisierung einer einfachen Bounding Volume Hierarchie (BVH). Im ersten Schritt werden alle Objekte mit einem Bounding Volume (BV) versehen. Ein BV ist eine geometrische Form, welche das Objekt vollständig umschließt. Oft werden hier sehr simple Formen wie ein Quader oder eine Kugel verwendet, da diese mit geringem Rechenaufwand auf Kollisionen geprüft werden können. Das BV für ein Objekt wird so gewählt, dass es die gesamte

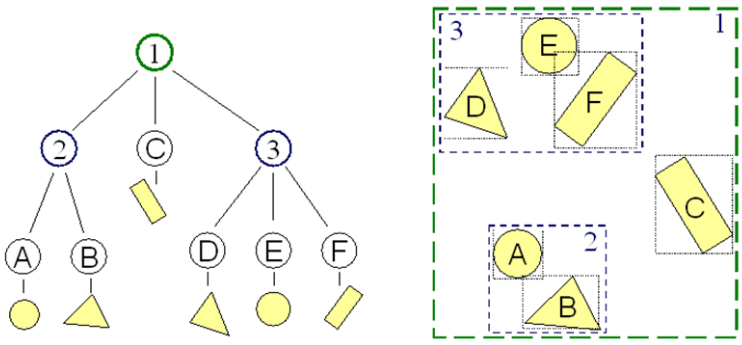

Abbildung 2.3: Ein Beispiel für BVH mit mehreren Ebenen

Form des Objekts umschließt, auf Abbildung 2.3 werden sie durch gepunktete Linien dargestellt. Der Vorteil, welcher daraus entsteht, ist der, dass nicht direkt die aufwendige Kollisionsprüfung mit den komplexen Formen der Objekte vorgenommen werden muss, sondern es kann erst durch einfache Checks geprüft werden, ob die BV der Objekte sich überschneiden. Ist dies nicht der Fall, können die Objekte nicht kollidieren und es muss keine konkrete Kollisionsprüfung zwischen den beiden Objekten durchgeführt werden. Im nächsten Schritt werden Hierarchieebenen hinzugefügt, in denen jeweils Gruppen von Objekten der unterliegenden Ebene in ein größeres BV eingebettet werden. Angenommen es soll geprüft werden, ob Objekt *C* und *E* in Abbildung 2.3 kollidieren. Es befinden sich beide Objekte in Ebene *1*, deswegen wird als nächstes geprüft, ob das BV von *2*, in dem sich Objekt *E* befindet mit dem BV von Objekt *C* kollidiert. Da dies nicht der Fall ist, würde er Algorithmus bereits in diesem Schritt mit dem Ergebnis abbrechen, das die beiden Objekte nicht kollidieren.

Lauterbach et al. haben im Jahr 2010 [23] ein Paper vorgestellt in dem sie den BVH Ansatz erfolgreich auf der GPU umsetzen. Der Fokus wurde dabei auf das Workbalancing der GPU Cores gelegt. Dazu wurde für jeden Core eine eigene Arbeitswarteschlange verwaltet. Sie wird dazu verwendet, um zu überwachen, ob der Core noch ausreichend Arbeitspakete ausstehend hat. Sobald ein Grenzwert an leerlaufenden Cores überschritten wurde, wird unterbrochen, um die Arbeitspakete neu zu verteilen. Anschließend nehmen alle ihre Arbeit wieder auf. Insgesamt haben sie eine performante GPU basierte Lösung erschaffen, bei der jedoch keine Verteilung auf mehrere GPUs vorgesehen ist.

Einen weiteren Ansatz hat Tang et al. im Jahre 2011 [39] mit einer streambasierten Variante der BVH vorgestellt. Die BVH Updates werden sukzessive auf die GPU übertragen, wodurch der Speicheroverhead verringert wird. Ebenfalls verwalten sie, zusätzlich zu der BVH, einen

Bounding Volume Traverse Tree (BVTT) in dem sie speichern, wie sie in den letzten Zyklen durch die BVH iteriert sind. Dies erlaubt ihnen weitere Bounding Volume Prüfungen zu vermeiden indem sie bestimmte Zweige der BVH ausschließen können. Insgesamt nutzen sie eine sehr komplexe Architektur und erzielen damit sehr gute Ergebnisse bei Nutzung einer GPU. Im Paper wird hier direkt Bezug auf die Variante von Lauterbach et al. genommen, wobei im Leistungssteigerung von Faktor 2 erreicht wird. Weitergeführt wurde dieser Ansatz zu einer Stoffsimulation [40], für die sie im nächsten Schritt eine GPU-basierte Kollisionsabwicklung entworfen haben [41]. Eine Verteilung auf mehrere GPUs ist hier ebenfalls nicht vorgesehen.

Mit einer Weiterentwicklung des BVTT haben sich Chitalu et al. in 2018 [9] beschäftigt. Durch ein optimiertes paralleles durchlaufen der BVTTs, sowie eine rein GPU basiertes Workload Balancing konnten sie eine Geschwindigkeitssteigerung von bis zu Faktor 7.1 gegenüber dem Streams Modell von Tang et al. [39] erreichen. Auch bei diesem Ansatz ist keine Verteilung angedacht.

2.3.2 Spatial Subdivision

Ein weiterer Ansatz zur parallelen Berechnung einer Kollisionserkennung ist die Spatial Subdivision. Er baut auf dem Prinzip der BVH auf, jedoch wird hier anstatt einer Baumstruktur ein Gridfeld mit Bounding Volumes verwendet.

Dabei wird die Umgebung in ein statisches, uniformes Gridfeld unterteilt, wie es auf Abbildung 2.4 dargestellt ist. Die Größe der Gridfelder wird so gewählt, dass sie minimal größer sind als jedes, sich in der Simulation befindende Objekt, wie auch bei einer BVH.

Abbildung 2.4 zeigt dies in einem zweidimensionalen Raum. Durch die Dimensionierung kann ein Objekt maximal vier Gridfelder schneiden, wie zum Beispiel Objekt *2* in der Abbildung. Zu Beginn wird für jedes Objekt ermittelt, welche Gridfelder es schneidet. Bei einer Kollisionsprüfung zwischen zwei Objekten kann dadurch initial geprüft werden, ob sie mindestens ein gleiches Gridfeld schneiden. Falls dies nicht der Fall ist, kann eine Kollision zwischen den beiden Objekten ausgeschlossen werden. Diese Vorauswahl ist sehr performant, da keine Kollisionsprüfung von BVs benötigt wird, sondern nur ein Vergleich der Gridfelder.

Pabst et al. haben 2010 [33] eine GPU basierte Umsetzung der Spatial Subdivison vorgestellt. Sie haben, um Kollisionsprüfungen zu vermeiden, zwischen Homecells und Phantomcells getrennt, wobei die Homecell eines Objekts die Gridzelle ist, in der sich der Hauptteil des Objekts befindet. Phantomcells sind alle anderen vom Objekt geschnittenen Gridfelder. Die Zuordnung von Objekt 2 in Abbildung 2.4, ergibt hiernach 8 als Homecell, 5,6 und 9 sind

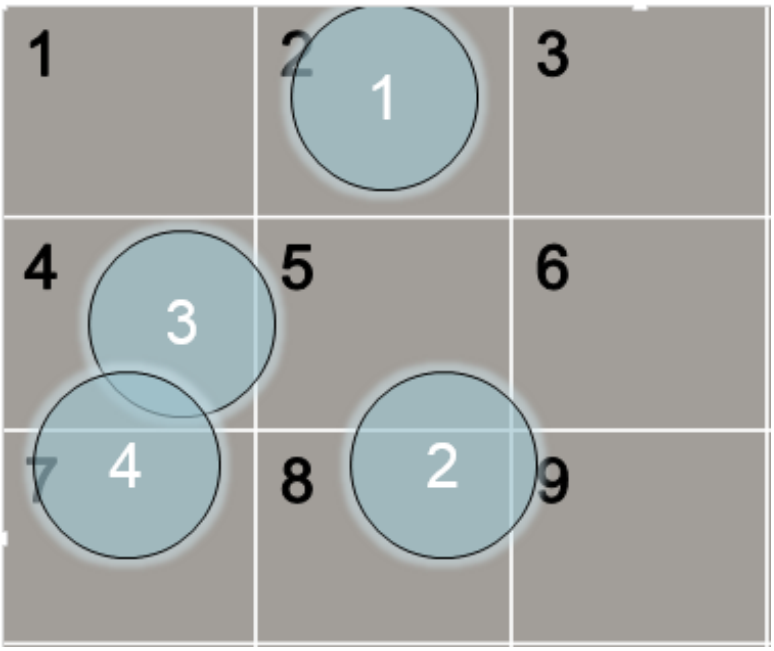

Abbildung 2.4: Beispiel für eine Spatiale Subdivision mit einer Gridebene

die Phantomcells. Mit dieser Trennung müssen nur noch Prüfungen durchgeführt werden falls sich Home- und Homecell oder Home- und Phantomcell überschneiden. Eine Prüfung zwischen Phantom- und Phantomcell ist in der Regel nicht notwendig, da sie sich aufgrund der Griddimensionierung nicht schneiden können. Ein weiterer Vorteil dieser Umsetzung ist, das die Last auf mehrere GPUs verteilt werden kann. Die im Paper vorgestellten Tests zeigen einen Performancegewinn von ca. 100 % bei vier GPUs/CPUs gegenüber einer GPU/CPU.

Diesen Ansatz haben Fan et al. im Jahre 2011 [13] erweitert, indem sie ein hierarchisches Gridfeld verwendet haben. Das Gridfeld nutzt zwei Ebenen, die so aufgeteilt sind, dass die Felder der unteren Ebene die halbe Kantenlänge haben. Somit füllen im zwei dimensionalem Raum je vier Felder der unteren Ebene ein Feld der Oberen aus. Effektiv rechnen sie mit einer Ebene, indem sie beide Ebenen basierend auf dem Vorkommen der Elemente zu einer Ebene zusammenführen. Durch die mehreren Ebenen sind sie in der Lage besser mit einer ungleichen Verteilung der Agenten umgehen zu können.

In 2014 haben Wong et al. [43] eine Erweiterung des Algorithmus vorgestellt. Sie haben anders als Fan et al. das hierarchische Gridfeld nicht auf zwei Ebenen begrenzt, sondern Arbeiten mit einer dynamischen Anzahl an Ebenen. Diese werden dazu verwendet, die Anzahl

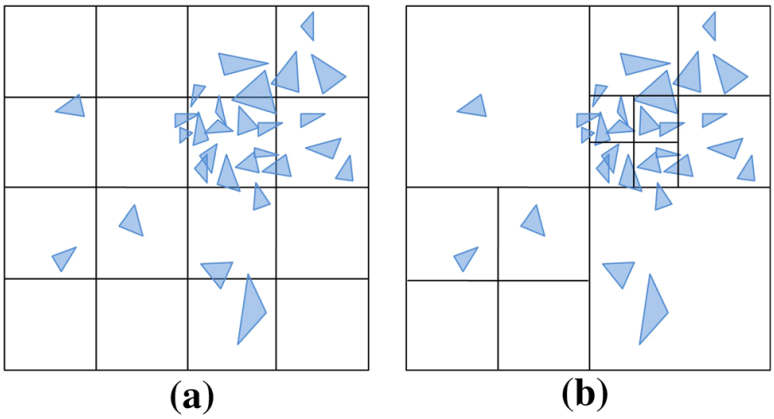

Abbildung 2.5: Bild einer ungleichen Verteilung von Objekten und die Anpassung des Gridfeldes entsprechend der Objektdichte [43]

der benötigten Kollisionschecks bei einer ungleichen Verteilung des Objekte zu verringern. Abbildung 2.5 zeigt die Unterteilung des Gridfeldes entsprechend des Objektaufkommens. Dafür wird im ersten Schritt eine Baumstruktur erstellt, in sie eintragen, wie viele Elemente sich jeweils in den Zellen befinden. Anschließend wird für jede Ebene die Anzahl der benötigten Kollisionsprüfungen berechnet. Dabei beginnen sie einmal auf der obersten Ebene und reduzieren so lange die Ebenen, bis sich die Anzahl der Kollisionsprüfungen nicht mehr verringert, um in nächsten Schritt das gleiche Verfahren beginnend bei der untersten Ebene durchzuführen. Damit können sie die Anzahl an benötigten Prüfungen gut reduzieren.

2.3.3 Weitere Ansätze

Neben den ausführlich vorgestellten Schemata gibt es einige weitere. Verschiedene Arbeiten beschäftigen sich mit einer Kollisionserkennung die auf Ray Tracing basiert [26][25]. Diese Ansätze sind jedoch für die *Narrowphase* ausgelegt und werden aufgrund dessen, dass der Fokus auf die *Breadphase* gelegt wurde nicht detaillierter erläutert. Alves at al. [4] hat in seiner Arbeit ein Verfahren vorgestellt, welches die Kapazitäten von Field Programmable Gate Arrays (FPGA) nutzt um die *Narrowphase* der Kollisionserkennung zu beschleunigen. Wiederum betrifft dies die *Narrowphase*, welche in dieser Arbeit aufgrund des Fokus auf MAS nicht genauer betrachtet wird.

Du et al. haben im Jahre 2017 ein Schema zur Verteilung der Kollisionserkennung auf einen GPU-Cluster vorgestellt [11]. Dafür unterteilen sie das Environment in identische unabhängige Blöcke, welche an die verschiedenen Nodes verteilt werden. Die Blockgröße hängt von dem verfügbaren Speicher auf der GPU ab. Jeder Node berechnet die Kollisionen für seinen zuge-

wiesenen Block. Diesen Ansatz haben sie auf dem TSUBAME Supercomputer [16] erfolgreich testen können.

3 Analyse

Nachdem die Grundlagen erläutert verwandte Arbeiten vorgestellt wurden, wird in diesem Kapitel die Analyse vorgenommen. Dazu werden die Annahmen aufgestellt, die als Grundlage für diese Arbeit genommen wurden. Anschließend werden im Abschnitt 3.2 die Anforderungen aus den Annahmen extrahiert. Zum Abschluss des Kapitels 3.3 werden die für Environment relevanten Komponenten analysiert und erläutert.

3.1 Annahmen

Die folgenden Annahmen stellen die Basis für diese Arbeit dar.

- **Ein Environment, welches die Kollisionserkennung auf der GPU berechnet, ist performanter als eins das alle Berechnungen auf der CPU durchführt:** Wie in Abschnitt 2.1 beschrieben, stellt das Environment eine zentrale Komponente des MARS Life Frameworks dar. Aus Abschnitt 2.2 geht hervor, dass die GPU bei parallelen Problemen Vorteile gegenüber der CPU bietet. Es ist zu prüfen, ob diese im Rahmen eines Environments genutzt werden können.
- **Die GPU optimierten Funktionalitäten können so angeboten werden, dass die Modellentwickler kein Spezialwissen benötigen:** Das MARS Life Framework wird von vielen Modellentwicklern aus unterschiedlichen Branchen verwendet. Folge dem kann nicht davon ausgegangen werden, dass Modellentwickler mit der GPU-Programmierung vertraut sind. Das Ziel ist es, das optimierte Environment allen zugänglich zu machen.

3.2 Anforderungen

Die Anforderungen lassen sich aufteilen in allgemeine Anforderungen bezüglich des GPU-Basierten Environments und Anforderungen an die Integration dessen.

3.2.1 Environment mit Kollisionserkennung auf GPU-Basis

Es soll die Performance des Environments durch den Einsatz der GPU verbessert werden. Dafür sollten möglichst viele der Environmentsfunktionen in die GPU-Basierte Kollisionserkennung integriert werden. Nicht außer Acht gelassen werden darf die Parallelisierung der Aufgaben, um einen möglichst großen Performancegewinn zu erreichen. Dabei ist zu beachten, dass das Environment, um vollständig kompatibel zu sein, alle Funktionen eines Environments im MARS Life Kontext anbieten muss. Diese umfassen folgende Punkte:

- Dynamische Agentenanzahl
- Bewegen der Agenten
- Evaluieren der Bewegung, prüfen auf Kollisionen
- Explorieren der Umgebung des Agenten

Die Hauptaufgabe des Environments ist das Verwalten von Agenten und das Überwachen derer Positionen. Deswegen ist es notwendig, dass dynamisch Agenten dem Environment hinzugefügt werden können. Beim Hinzufügen muss eine Prüfung stattfinden, ob der Agent mit anderen Elementen kollidiert. Weiterhin soll sämtliche Bewegung der Agenten verwaltet und organisiert werden. Immer wenn ein Agent seine Position verändern möchte, muss das Environment dies evaluieren und anschließend sowohl die resultierende Position als auch mögliche Kollisionen mit anderen Agenten als Ergebnis liefern. Eine weitere Aufgabe des Environments umfasst das Explorieren der Agenten. Agenten können mit anderen Agenten aus ihrer räumlichen Umgebung zu interagieren. Um diese zu finden, führt der Agent ein Explore über das Environment aus, welches andere Agenten in dem gewünschten Umkreis sucht und den Agenten mitteilt. Zu beachten ist, dass ein Filtern nach verschiedenen Agententypen möglich sein muss. Auch sollen unterschiedliche Kollisionstypen unterstützt werden. Der Kollisionstyp eines Agenten bestimmt ob und mit welchen anderen Agenten er kollidieren kann. Aktuell wird dort in MARS Life zwischen Geistern und massiven Agenten unterschieden. Geist Agenten kollidieren nicht mit anderen Agenten im Gegensatz zu den massiven Agenten.

Die Anforderungen sind so umzusetzen, dass die Vorteile der GPU-Basierten Kollisionserkennung bestmöglich ausgenutzt werden.

3.2.2 Kollisionserkennung im Multi-Agent-Simulation Kontext

Nach den Anforderungen für ein Environment im MARS Kontext wird nun auf die speziellen Anforderungen an eine Kollisionserkennung in einer MAS eingegangen.

Bei den MAS die mit MARS Life durchgeführt werden, reichen in vielen Szenarien simple Formen um die Ausmaße der Agenten darzustellen. Die Kollisionserkennung wird in den meisten Fällen dafür verwendet, um mit dem Kollisionspartner zu interagieren. Zum Beispiel frisst ein Schaf das Gras, mit dem es kollidiert. Zudem ist MARS Life darauf ausgelegt, Simulationen mit einer hohen Anzahl von Agenten durchzuführen. Beim Großteil der Simulationen sorgt eine detailgetreue Darstellung der Agenten nicht für aussagekräftigere Simulationsergebnisse. Deswegen wird der Fokus darauf gesetzt die Simulationsdurchläufe möglichst schnell zu berechnen. Dafür wird auf simple Formen für die Kollisionsberechnungen gesetzt, um mit mehr Agenten simulieren zu können und insgesamt geringe Ausführungszeiten zu erreichen.

Eine weitere Eigenheit der MAS ist, dass Kollisionen nicht der Ausnahmefall sind. Agenten erkunden oft in jedem Tick ihre Umgebung. Dadurch dass das Explorieren so gelöst wird, dass ein Objekt mit den Ausmaßen des zu explorierenden Bereichs auf Kollisionen mit Agenten geprüft wird, entstehen in jedem Tick viele gewollte Kollisionen. Dabei ist zu beachten, das die Explorationsradien/Sichtradius in der Regel deutlich größer sind als die Agenten, wodurch wiederum eine höhere Chance auf Kollisionen entsteht. Das sind alles Prüfungen die zusätzlich zu den klassischen Kollisionsprüfungen der Agenten untereinander durchgeführt werden müssen. Ein wichtiger Punkt ist hierbei, dass die Kollisionen, welche durch das Explorieren entstehen, nicht umgangen werden sollen.

Beim Explorieren ist ebenfalls zu beachten, dass ein Agent in der Regel seine unmittelbare Umgebung erkundet. Die Folge daraus ist, dass viele der Agenten auf ihrem eigenen Standort mit einen deutlich größeren Radius explorieren. Somit wird die Vergleichsweise kleine Form des Agenten von dem großen Explorationsobjekt umfasst. Für Ansätze wie von Wong et al. [43] die ein adaptives Gridfeld verwenden kann dies sehr unvorteilhaft sein, da das Gridfeld auf Basis des großen Explorationsobjektes gewählt werden müsste.

Weiterhin ist wichtig, dass große Szenarien abgebildet werden können. Speziell beim MARS Life Framework, welches sich auf das Simulieren großer Szenarien spezialisiert hat. Das ausgewählte Schema sollte dahingehend ausgelegt sein.

3.2.3 Erweiterte Integration

In diesem Abschnitt wird beschrieben, auf welche Punkte bei der Integration des Environments in MARS Life geachtet werden muss. Wie in Abschnitt 2.2 beschrieben wurde, unterscheidet sich die GPU-Programmierung in vielen Punkten von dem Programmieren in einer objektorientierten Hochsprache wie C#, welche für die Modellentwicklung verwendet wird. Es soll dem Modellentwickler weiterhin möglich seine Modelle zu entwerfen, ohne sich mit der GPU-Programmierung auseinander zu setzen. Dafür muss die Kollisionserkennung so entwor-

fen und integriert werden, dass keine GPU Kenntnisse für die Verwendung notwendig sind. Ebenfalls ist es vollständig in die bestehende Struktur zu integrieren, sodass Modelle, die das neue Environment verwenden, weiterhin alle anderen bestehenden Komponenten von MARS Life nutzen können. Dabei ist zu Beachten, dass die Abstraktionsschichten, die oberhalb des Environments angeordnet sind weiterhin für das Environment mit GPU-Basierter Kollisionserkennung verfügbar sein muss. Dazu wird im nächsten Abschnitt darauf eingegangen welche Komponenten, neben dem Environment, durch die neue Kollisionserkennung betroffen sind.

3.3 Schnittstellen und relevante Komponenten für die Integration

Es soll eine nahtlose Integration der Kollisionserkennung im Rahmen eines Environments vorgenommen werden. Dazu werden in diesem Abschnitt die Komponenten des MARS Life Systems vorgestellt, die direkt mit dem Environment interagieren und entsprechend bei der Integration berücksichtigt werden müssen.

MARS Life bietet bereits mehrere Environments die auf unterschiedliche Anwendungszwecke zugeschnitten sind. Eine Gruppe bilden zwei Environments die auf kartesischen Koordinaten basieren. In ihnen können sich die Agenten frei bewegen, ohne Konkretisierung durch ein Gridfeld oder Einschränkungen in Bezug auf die Agentenform. Sie werden über das *IESC* Interface als allgemeine Schnittstelle zur Verfügung gestellt. Zusätzlich gibt es noch zwei weitere Umsetzungen eines Environments, die nicht kartesisch arbeiten. Dabei handelt es sich um das GeoGridEnvironment, welches mit Gps Koordinaten nutzt und das GridEnvironment, bei dem die Umgebung in ein festes Gridfeld eingeteilt wird. Beide haben aufgrund der unterschiedlichen Vorgehensweise eine eigene Schnittstelle über die sie im MARS Life Framework erreichbar sind.

Um die Modellierung eines Modells zu erleichtern, bietet das MARS Life System mit den Basic Agents eine Grundimplementierung eines Agenten, welche von Jan Dalski entworfen und umgesetzt wurde [3]. Diese stellen eine weitere Abstraktionsschicht über dem Environment dar, welche dem Modellierer die Umsetzung seines Modells erleichtert. Abbildung 3.1 zeigt den Aufbau des basic Agents.

Der Basis Agent ist nach dem Sense Reason Act (SRA) Pattern konstruiert worden. Zu Beginn eines Simulationsticks wertet der Agent seine Umgebung aus (Sense). Als Nächstes folgt der Reason Schritt, in dem Hauptlogik des Agenten, wie Interaktion mit anderen Agenten stattfindet. Die letzte Aktion des Agenten ist die Bewegung, welche bei SRA als Act beschrieben wird. Dort bewegt der Agent sich zu seinem nächsten Ziel. Um die Durchführung der einzelnen

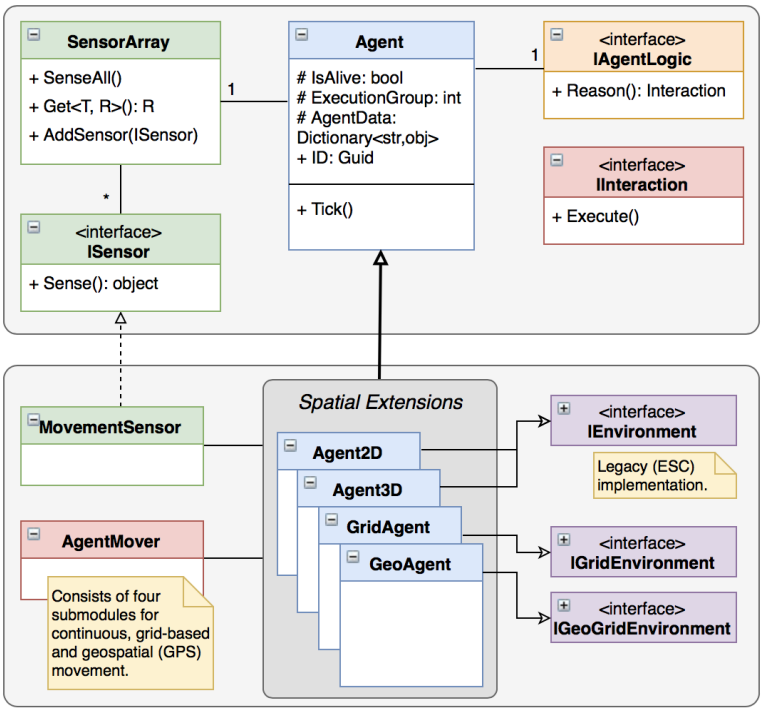

Abbildung 3.1: Struktur der Basic Agents [3]

Phasen zu erleichtern, gibt es den AgentMover, welcher die Bewegung über das Environment abstrahiert. Neben dem Mover wird auch ein Sensorpaket bereitgestellt, was die Wahrnehmung der Umgebung im Sense Schritt erleichtert. Diese können generisch angelegt werden basierend auf dem aktuellen Szenario. Ein MovementSensor wird jedoch für jeden Agenten pauschal angelegt. Über diesen kann der Agent prüfen ob er während seiner letzten Bewegung kollidiert ist. Eine detaillierte Beschreibung des SRA Patterns und dessen Implementierung im Rahmen der Basic Agents, bietet der entsprechende Projektbericht von Jan Dalski [3]. Dort befinden sich auch weitere Informationen über das Mover- und Sensorkonzept, welches in diesem Rahmen angelegt umgesetzt wurde. Für jedes Environment im MARS Life System wurde in basic Agent nach dem SRA Pattern umgesetzt.

Auf Abbildung 3.1 wird das IESC Interface als Legacy Code bezeichnet. Dieses wird aktuell aufgrund der, im Vergleich zu den anderen Environments, schlechten Performanz nicht in den Modellen verwendet. Allerdings ist es auch das einzige Environment, welches in der Lage ist Kollisionen zwischen Agenten unter Beachtung deren Form zu bestimmen. Sowohl das GeoGridEnvironment als auch das GridEnvironment Arbeiten mit einem unterliegenden Gridfeld. Die Gridfelder haben keine konkrete Größe, sondern könne als einheitliche Flächen betrachtet werden. Ein Agent kann sich nur auf einem Gridfeld befinden und sich von Gridfeld zu Gridfeld bewegen. Es gibt keine Schritte zwischen den Feldern. Dementsprechend kann keine direkte Kollision ermittelt werden, sondern nur ob sich mehrere Agenten auf demselben Gridfeld befinden. Ebenso haben sie keine Form oder Ausdehnung, es gibt bei den Environments keine unterschiedlich großen Agenten. Aus Performanz-gründen wurden, trotz der Einschränkungen, in den letzten Modellen die GridEnvironments verwendet.

Ein weiterer wichtiger Aspekt ist das alle Environments die Anfragen der Agenten synchron und damit direkt beantworten. Viele dieser Anfragen benötigen eine Kollisionsprüfung, welche dann direkt ausgeführt werden muss. Das führt zu einem hohen aufkommen an sequentiellen Anfragen. Dort müssen Anpassungen vorgenommen werden, da die GPU nur bei vielen parallelen Prüfungen ihre Vorteile ausspielen kann.

4 Konzept

Das Konzept wird in zwei Abschnitte unterteilt. Der Erste geht auf die Integration in das MARS Life Framework ein. Dabei werden die bestehenden Environment-Schnittstellen betrachtet und erläutert. Anschließend wird im zweiten Abschnitt die Konzeption des GPU-Environments betrachtet.

4.1 Integration

In diesem Absatz wird beschrieben, wie die GPU-Basierte Kollisionserkennung als Environment in das MARS Life Framework integriert wird.

Es soll ein neues kartesisches Environment angeboten werden, welches ausreichend Performanz hat, um in großen Modellen verwendet zu werden. Ziel ist es damit eine Alternative zu den Grid-Environments zu schaffen, die in der Lage ist, mit Agenten zu arbeiten, die eine Ausdehnung und Form haben. Es wird zu dem Zweck auf das IESC Interface als Basis Schnittstelle zum GPU-Environment gesetzt, da es bereits in das MARS Life Framework integriert ist. Zu beachten ist jedoch, dass das eine GPU-Basierte Kollisionserkennung nur bei stark parallelisierten Anfragen performant arbeiten kann.

Das IESC Interface sieht eine synchrone Verarbeitung der Anfragen vor, wodurch bei jeder Anfrage die Kollisionsprüfung direkt erfolgen muss, um den Agenten das Ergebnis liefern zu können. Eine solche Architektur erlaubt wenig Parallelisierung, da keine Anfragen gesammelt und somit parallel verarbeitet werden können. Aus diesem Grund wird eine neue asynchrone Schnittstelle entworfen, die sich in das Schema des MARS Life Environments integrieren lässt. Es sollen jedoch weiterhin alle Funktionen angeboten werden, die über das IESC Interface verfügbar waren. Das SRA Pattern des Basic Agents soll weiterhin angeboten werden, sodass für die Modellentwickler keine große Umstellung nötig ist. Dafür wird ein asyc Agent Paket entworfen, welches die AgentMover, Sensoren und Agents für das SRA Pattern im asynchronen Kontext anbietet.

4.2 Environments mit GPU-Basierter Kollisionserkennung

Aus den Analyse geht hervor, dass die Kollisionserkennung am Besten über ein Environment in MARS Life umgesetzt wird. Da es aktuell kein Environment gibt, welches mit kartesischen Koordinaten arbeitet und eine ausreichend gute Performance hat um größere Szenarien zu simulieren, stellt dies den Angriffspunkt für die Kollisionserkennung auf GPU-Basis dar.

In Kapitel 2.3 wurden die beiden verbreiteten Schemata BVH und Spatial Subdivision zur GPU-Basierten Kollisionserkennung vorgestellt. Aufgrund der deutlich besseren Möglichkeiten zur Verteilung und die damit verbundene Möglichkeit größere Szenarien simulieren zu können, ist die Entscheidung auf die Spatiale Subdivision gefallen. Zu beachten ist jedoch, dass die Spatiale Subdivision deutlich langsamer wird, sobald eine starke Größenvarianz unter den Agenten herrscht. Ein solches Szenario ist bei MAS nicht unüblich, weswegen dieses Problem konkret durch eine Erweiterung der Spatialen Subdivision angegangen wird.

Bei der Umsetzung und Konzeption wurde so vorgegangen, dass im ersten Schritt eine Kollisionserkennung mit Spatialer Subdivision auf einem eindimensionalen statischen Gridfeld nach dem Schema von Pabst et al. [33] entworfen wurde. Diese arbeitet im drei dreidimensionalen Raum, um keine Einschränkungen bezüglich der Szenarien zu erzeugen. Daraus ist Konzept für das Single Layer 3D Grid-Environment (SLGE) entstanden, welches im Abschnitt 4.2.1 weiter erläutert wird.

Dieser Ansatz wurde im nächsten Schritt erweitert mit dem Ziel besser mit der Größenvarianz und den gewollten Kollisionen umgehen zu können. Dafür wurde ein angepasstes Konzept entworfen, das auf mehrere Gridebenen setzt. Für den Großteil der Modelle, die bisher im MARS Life Kontext simuliert werden, ist ein Environment welches im zweidimensionalen Raum arbeitet ausreichend. Aus dem Grund wird die angepasste Kollisionserkennung ebenfalls im zweidimensionalen Raum arbeiten. Dadurch wird sowohl der Speicherbedarf als auch die Ausführungszeit reduziert. Das daraus entstandene Konzept wird im Abschnitt 4.2.2 vorgestellt.

4.2.1 Single Layer 3D Grid-Environment

In diesem Abschnitt wird das Konzept des SLGE erläutert. Das SLGE setzt auf das Schema von Pabst et al. [33]. Dabei wird das Environment in ein uniformes Gridfeld eingeteilt. Die Größe der Felder wird durch die Elemente im Environment bestimmt. Um den Algorithmus nutzen zu können, muss ein Gridfeld größer als das größte Element im Environment sein.

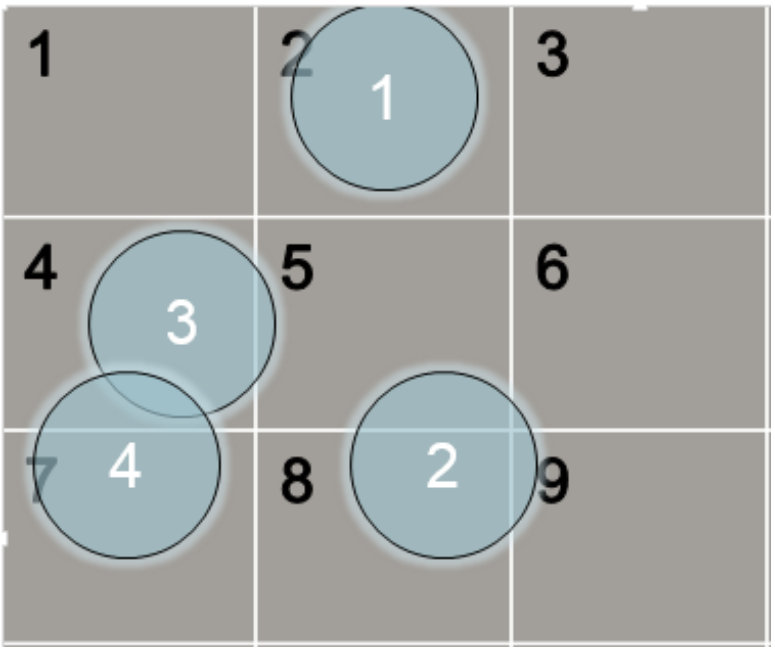

Abbildung 4.1: Beispiel eines Gridfeldes, auf Tabelle 4.1 sind Zuordnung der Elemente und die benötigten Kollisionsprüfungen dargestellt

Elementnummer	Homecells	Phantomcells	Kollisionsprüfung
ID 1	2	-	-
ID 2	8	5, 6, 9	-
ID 3	4	5	Obj 4
ID 4	7	4	Obj 3

Tabelle 4.1: Zuordnung der Elemente in Home- und Phantomcells aus Abbildung 4.1

Anschließend werden jedem Element seine geschnittenen Felder zugeordnet. Hierbei wir zwischen Home- und Phantomcell unterschieden. Die Homecell stellt die Zelle dar, in dem sich der Hauptteil des Elements befindet. Als Phantomcells eines Elements werden alle übrigen geschnittenen Zellen bezeichnet. Zu beachten ist, dass jedes Element mit einem runden Bounding Volume (BV) ummantelt wird. Auf Basis der Gridfelder kann eine Vorauswahl getroffen werden, welche Elemente auf Kollision geprüft werden müssen. Ausgeschlossen werden können pauschal alle Elemente, die keine gleichen Gridfelder schneiden. Zusätzlich können noch die Elemente ausgeschlossen werden, bei denen sich jeweils nur eine Phantomcell überschneidet. Aufgrund des BV und der Gridgröße ist es nicht möglich, dass die Elemente in dem Fall kollidieren.

Abbildung 4.1 und Tabelle 4.1 zeigen die Aufteilung der Elemente anhand eines Beispiels. Zu beachten ist, dass Element *2* und *3* nicht geprüft werden müssen, da sie nur Zelle 5 als gemeinsame Zelle haben und es sich für beide Elemente nur um eine Phantomcell handelt. Nach diesem Schema wird die Kollisionserkennung im SLGE umgesetzt.

Zusätzlich zur reinen Kollisionserkennung muss auch das Explorieren, sowie die verschiedenen Kollisionstypen umgesetzt werden. Aufgrund der hohen Anzahl an Agenten und des stark begrenzten Speichers der GPU, wurde bei dem Design auf eine speicher-optimierte Lösung hingearbeitet. Jeder der Kollisionstypen hat bestimmte Sonderregeln, die im Falle einer Kollision mit anderen Elementen beachtet werden müssen. Auf GPU Ebene werden diese Objekteigenschaften bitcodiert abgelegt. Damit können sie performant geprüft werden und verbrauchen nur eine geringe Menge Speicher.

Das Explorieren wird gelöst, indem ein neues Element erzeugt wird, welches die Ausmaße des zu explorierenden Bereiches hat. Dieses Element wird als „Exploreelement“ gekennzeichnet und ebenfalls über die GPU auf Kollisionen geprüft. Alle Elemente, die mit dem „Exploreelement“ kollidiert sind, werden als gefundene Elemente in dem Bereich zurückgegeben. Die Auflösung, wer mit wem kollidieren kann, wird durch die Auswertung der Kollisionstypen vorgenommen.

Weiterhin wird zwischen neu hinzugefügten und sich bereits im Environment befindlichen Agenten unterschieden. So kann ein individuelles Verhalten bei der Kollision während des Hinzufügens verwendet werden.

4.2.2 Multi Layer 2D Grid-Environment

Die Anpassung der Kollisionserkennung an den MAS Kontext wird über das Multi Layer 2D Grid-Environment (MLGE) umgesetzt und integriert. Wie das SLGE setzt das MLGE auf das Verfahren der Spatialen Subdivision. Jedoch wird ein hierarchisches Gridfeld genutzt, mit dem Ziel effektiver mit einer starken Größenvariation umgehen zu können. Auch Wong et al. [43] haben bei ihrer Kollisionserkennung auf ein hierarchisches Grid gesetzt. Jedoch werden die unterschiedlichen Hierarchieebenen anders um- und eingesetzt. Wong et al. verwenden das hierarchische Gridfeld nicht mit dem Ziel ein auf unterschiedliche Elementgrößen gut reagieren zu können, sondern um effektiver bei einer ungleichen Verteilung der Agenten zu sein. Das hat den Hintergrund, dass die dort entwickelte Kollisionserkennung auf die Prüfung von Triangle-Meshes ausgelegt ist. Diese werden verwendet um damit komplexe flexible Elemente wie Kleider und ähnliches darzustellen. Sie bestehen aus einer Vielzahl an Dreiecken, die so verbunden sind, dass sie das gewünschte Element darstellen. Allerdings haben diese Dreiecke alle eine sehr ähnliche Größe, weswegen bei ihrer Arbeit der Fokus nicht darauf lag, mit

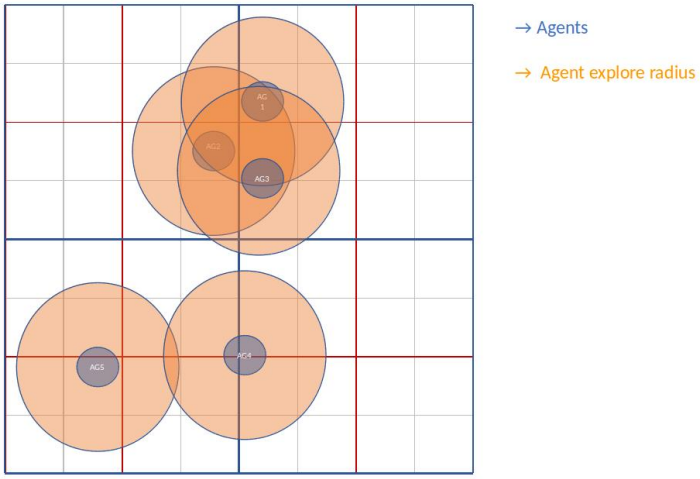

Abbildung 4.2: Darstellung eines Szenarios mit explorierenden Agenten in einem hierarchischen Gridfeld

unterschiedlichen Größen gut arbeiten zu können.

Wong et al. gehen so vor, dass sie initial prüfen, wie viele Kollisionsprüfungen auf jeder Hierarchieebene pro Quadranten nötig sind. Auf Basis dieser Daten werden jeweils die Hierarchieebenen bestimmt, die zu der geringsten Gesamtanzahl an Kollisionsprüfungen führen. Dieses Verfahren funktioniert, wie sie in Ihrem Paper gezeigt haben, sehr gut für Triangle-Meshes. Allerdings sind Kollisionen dort der Ausnahmefall und es wird stets versucht diese aufzulösen. Für den Anwendungsfall der MAS würde das reduzieren auf eine Ebene mit unterschiedlicher Granularität zu einer deutlich höheren Zahl an Kollisionsprüfungen führen. Das folgt daraus, dass beispielsweise das Explorieren oft in einem großen Radius um die Agenten herum erfolgt. Würde mit einer Ebene gearbeitet werden, müsste das Gridfeld sich an das Explorieren anpassen und hätte damit eine sehr grobe Auflösung. Auf Abbildung 4.2 wird das Problem anhand eines Beispiels dargestellt. Jeder Agent für sich hat eine relativ kleine Form und einen großen Explorationsradius, dadurch wird das Gridfeld anhand der Exploration dimensioniert. Daraus folgt, dass sich sehr viele Agenten im gleichen Gridfeld befinden können, für die dann jeweils eine Detailprüfung vorgenommen werden muss. Um diesem Problem entgegenzugehen, wurde ein neues Schema für die Umsetzung des hierarchischen Gridfeldes entwickelt.

Damit auch bei Elementen unterschiedlicher Größe im selben Bereich die Anzahl der Kollisionsprüfungen effektiv reduziert werden kann, werden mehrere Gridebenen parallel verwaltet, anstatt auf eine Ebene zu reduzieren. So können alle Kollisionsprüfungen für die Elemente jeweils in der Ebene bestimmt werden, in der die Elemente basierend auf ihrer Größe am besten hineinpassen, diese wird als native Ebene bezeichnet. Bezogen auf das obere Beispiel in Abbildung 4.2 heißt das, dass die Kollisionschecks für Explores auf der oberen Gridebene bestimmt werden und die Agenten untereinander ihre Prüfungen auf der unteren Ebene vornehmen können. Dadurch können auf der Ebene der Agenten, mit Hilfe des feinmaschigen Gridfeldes, viele Checks ausgeschlossen werden.

Nachfolgend wird das angepasste Verfahren im Detail vorgestellt. Dazu wird zuerst auf die Zuordnung der Elemente zu den Gridebenen eingegangen, um im Anschluss zu beleuchten wie nach dem neuen Schema die erforderlichen Detailprüfungen bestimmt werden.

Zuordnung der Elemente

Dieser Abschnitt beschreibt, wie das hierarchische Gridfeld aufgebaut ist und nach welchen Kriterien den Elementen ihre Ebene zugeordnet wird. Das hierarchische Gridfeld wird im 2D Raum erstellt. Die Größeneinteilung wird wie bei einem Quadtree vorgenommen. Damit hat ein Feld ein Viertel der Größe des darüber liegenden Levels, wodurch genau vier Felder ein Feld der oberen Ebene Ausfüllen. Abbildung 4.2 ein Grid mit dieser Aufteilung.

Um den Ansatz der Phantomcells von Pabst et al. weiter verfolgen zu können, wird jedes Element mit einem kreisförmigen BV versehen. Eine weitere davon ausgehende Anforderung ist, dass ein Gridfeld größer sein muss, als alle Elemente, die es beherbergt.

Für jedes Element wird die tiefste und damit kleinste Ebene gesucht, für die diese Anforderung noch erfüllt ist. Diese wird als native Ebene des Elements markiert und das Element wird dieser Ebene zugeordnet. Danach wird jedes Element, welches sich auf einer der unteren Ebenen befindet, auf allen darüber liegenden eingetragen.
Als Beispiel wird ein hierarchisches Gridfeld mit fünf Ebenen betrachtet, bei dem Ebene 1 die gröbste Ebene darstellt und Ebene 5 entsprechend die kleinsten Gridfelder hat. Ein Element, welches von seinen Ausmaßen gerade noch in ein Gridfeld von Ebene 4 passt und bei Ebene 5 größer als ein Gridfeld ist, würde sich in Ebene 4 als natives Element eintragen. Zusätzlich würde es bei den Ebenen 3,2 und 1 eingetragen werden, da es dort ebenfalls in die Gridfelder passt. Bei den oberen Ebenen wird beim Element vermerkt, dass es nicht die native Ebene des Elementes ist.
Abbildung 4.3 zeigt die Aufteilung der Elemente in den jeweiligen Ebenen.

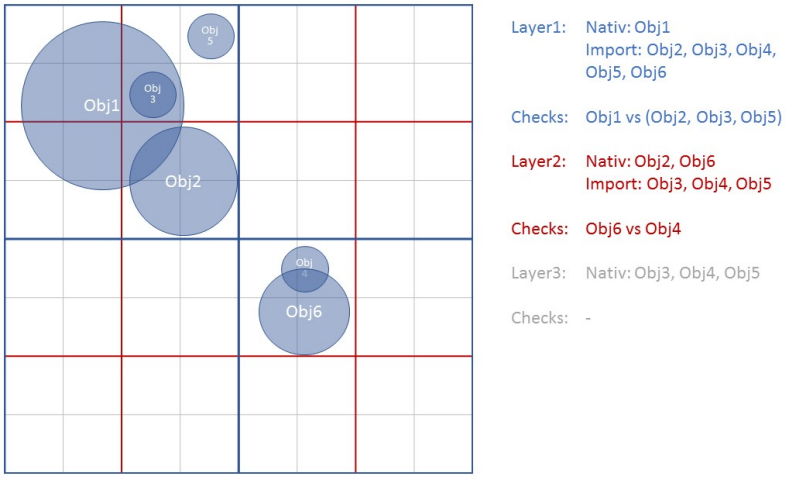

Abbildung 4.3: Darstellung eines hierarchischen Gridfeldes mit Bestimmung der Kollisionsprüfungen

Damit ist jedes Element in den Ebenen eingetragen, in denen die Gridfelder kleiner sind, als die Ausmaße des Elements.

Bestimmen der Kollisionsprüfungen

In diesem Abschnitt wird beschrieben, wie auf Basis des hierarchischen Gridfeldes bestimmt wird, welche Detailprüfungen durchgeführt werden müssen.

Auch wenn sich die Elemente auf mehreren Ebenen eintragen, soll die Anzahl der Kollisionsprüfungen insgesamt verringert werden.

Dabei wird so vorgegangen, dass die nativen Elemente einer Ebene den aktiven Teil des Prüfens übernehmen. Dazu wird von jedem nativen Element ausgehend geprüft, ob es weitere Elemente auf der Ebene gibt, die ein identisches Gridfeld schneiden. Für diese Elemente wird nach dem Schema von Pabst et al. bestimmt, ob eine Detailprüfung vorgenommen werden muss. Dieser Vorgang wird für alle Ebenen wiederholt. Auf jeder Ebene wird nur von den dort nativen Elementen ausgehend geprüft, wodurch das doppelte Prüfen eines Elementenpaars verhindert wird.

Dadurch, dass sich die Elemente ebenfalls auf den oberen Ebenen eintragen, sind alle für die Ebene relevanten Elemente vorhanden. Jede Ebene kann deswegen komplett unabhängig von den anderen Ebenen betrachtet werden.

Die Abbildung 4.3 zeigt ein Beispiel mit drei Ebenen. Auf der rechten Seite des Bildes ist die Aufteilung der Elemente auf die einzelnen Ebenen (Layer) aufgezeigt. Weiterhin ist angegeben, welche Elemente in der Ebene als nativ markiert sind und bei welchen es sich um importierte Elemente und damit Elemente einer unteren Ebene handelt. Zusätzlich dazu sind die Detailprüfungen (Checks) eingetragen, welche auf dieser Ebene vorgenommen werden müssen. Nach dem neuen Schema ergibt das für dieses Beispiel insgesamt vier Detailprüfungen. Bei einem statischen Gridfeld mit einer Ebene könnte die Kollision zwischen Objekt 2,3 und 5 nicht ausgeschlossen werden. Damit müssten sieben Detailchecks durchgeführt werden.

Damit ist der konzeptionelle Teil zu dem Environment und der Kollisionserkennung abgeschlossen und es wird im nächsten Kapitel auf die Umsetzung des Konzeptes eingegangen.

5 Realisierung

In diesem Kapitel wird die Umsetzung der Kollisionserkennung im Rahmen des Environments detailliert erläutert. Dabei wird zuerst das SLGE beschrieben, um im Anschluss dessen Erweiterung, das MLGE vorzustellen. Zuletzt wird noch kurz auf den Radix-Sort eingegangen, der für beide Verfahren benötigt wurde.

5.1 Single Layer 3D Grid-Environment

Das SLGE mit dem statischen Gridfeld wurde als erstes umgesetzt und stellt die Basis für das MLGE dar. Dieses Kapitel geht auf die Implementierung des SLGEs ein. Zu Beginn werden zwei Arrays angelegt, das Cellid und das Celldata Array. Das Cellid Array beinhaltet die gehashten Gridzellen IDs und das Celldata enthält die Daten über das Objekt, welches sich in der Zelle befindet. Es wird im Celldata Array neben der Objekt-ID auch der Kollisionstyp sowie die Aktionsart (Hinzufügen, Bewegen oder Explorieren) gesichert. Beide Arrays haben eine identische Größe und werden immer parallel verändert, sodass zu jeder Zeit über den Index die Elemente zugeordnet werden können.
Im ersten Schritt der Spatialen Subdivison wird eine Kernelmethode aufgerufen, welche die Formen und Ids der Objekte als Eingangsparameter hat. Basierend auf den Formen wird berechnet, welche Gridfelder geschnitten werden, um das Cellid und Celldata Array entsprechend zu füllen. In Abbildung 5.1 ist die Erstellung des Cellid Arrays visualisiert. Es wird ein dreidimensionales Szenario betrachtet, jedes Objekt kann daher bis zu 8 Gridfelder schneiden. Um das Array nicht dynamisch ändern zu müssen, werden immer 8 Felder je Objekt vorgesehen. Dies hat zusätzlich den Vorteil, dass die Aufteilung des Arrays vorher festgelegt werden kann. Für eine erleichterte Weiterverarbeitung werden die Homecells beim Hinzufügen gebündelt an den Anfang des Arrays gelegt. Anschließend werden beide Arrays mit Hilfe des Radix-Sorts nach der Cellid sortiert. Aufgrund der Reihenfolge, die vor dem Sortieren hergestellt wurde, befinden sich bei gleicher Cellid die Homecells vor den Phantomcells.

Der nächste Kernel durchsucht das Cellid Array auf aufeinander folgende identische Werte. Diese sind repräsentativ für mehrere Objekte, die das gleiche Gridfeld schneiden. Um dies

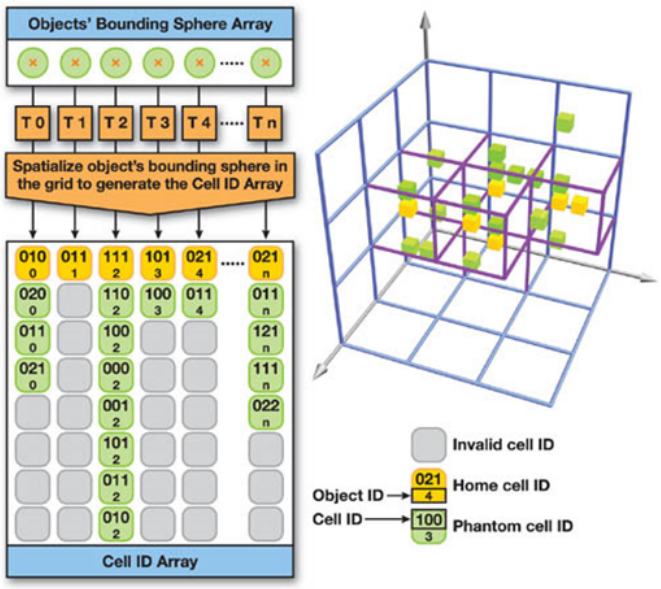

Abbildung 5.1: Einfügen der Objekte in das Cellid Array [24]

performant auf der GPU ausführen zu können, wird das Cellid Array in mehrere Abschnitte unterteilt, welche den GPU Threads zugewiesen werden. Jeder Thread hat einen fest zugewiesenen Abschnitt, den er prüft. Damit Zahlenfolgen, die einen Abschnitt überschreiten, korrekt erkannt werden können, prüft jeder Thread bei einer aktiven Folge über seine Abschnittsgrenze hinaus, bis er das Ende der Folge erreicht hat. Auf Abbildung 5.2 ist ein Beispiel für die Aufteilung der Threads dargestellt. Dort muss *Thread 1* über seine Grenze hinauslaufen, um die vollständige Folge erfassen zu können. Um zu verhindern, dass mehrere Threads dieselbe Zahlenfolge erkennen, falls diese über die Abschnittsgrenzen hinaus geht, prüft jeder Thread zuerst, ob er sich zu Beginn seines Abschnitts bereits innerhalb einer Folge befindet und überspringt diese gegebenenfalls. Im Beispiel auf Abbildung 5.2 würde *Thread 2* den ersten Wert überspringen, da dieser bereits von *Thread 1* geprüft wird. Als Ergebnis wird gesichert, wo die wiederholten Werte starten und wie viele identische Werte folgen. Die daraus entstehenden Listen werden als Kollisionssublisten bezeichnet. Zusätzlich wird die Anzahl an Homecells hinterlegt, welche sich aufgrund der Sortierung immer am Anfang einer Folge befinden.

Der nächste Schritt ermittelt zwischen welchen Objekten Detailprüfungen vorgenommen werden müssen. Geprüft muss, nach Pabst et al.[33] Home- gegen Homecell und Home- gegen Phantomcell. Ein entsprechender Kernel durchläuft die vorher ermittelten Folgen und erstellt, falls eine Detailprüfung notwendig ist, ein Tupel mit den Ids der jeweiligen Objekte. Auch hier wird die Arbeit unterteilt um sie auf mehreren Threads durchführen zu können.

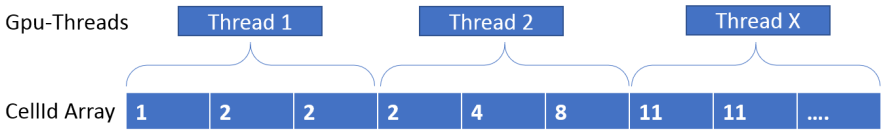

Abbildung 5.2: Beispiel für das Erstellen der Kollisionssublisten auf der GPU

Für die zu prüfenden Tupel werden die konkreten Formdaten geladen und zusammen mit den Objektdaten an die GPU gesendet. Die Detailprüfungen können aufgrund der unabhängigen Tupel aus dem vorherigen Schritt auf mehrere GPUs verteilt werden. Das Ergebnis des Checks ist eine Liste mit kollidierenden Objekten, in der sich noch Duplikate befinden können. Diese filtert anschließend der C# Teil auf dem Host und somit die CPU. Als Ergebnis wird für jeden Agenten eine Liste mit den Objekten erstellt, die mit ihm kollidieren. Mit diesem Schritt ist die Kollisionserkennung abgeschlossen und die Ergebnisse werden den Agenten über Delegaten mitgeteilt.

5.2 Multi Layer 2D Grid-Environment

Nachdem das Konzept der hierarchischen Kollisionserkennung im letzten Abschnitt vorstellt wurde, wird nun auf die Umsetzung vom Multi Layer 2D Grid-Environment (MLGE) eingegangen.

Für die Kollisionsprüfungen auf den einzelnen Ebenen wird angepasste Variante des Algorithmus von Pabst et al. [33] verwendet. Neben der Home- und Phantomcell Logik, erhalten die Zellen ein neues weiteres Attribut, über das zwischen den nativen und nicht nativen Zellen unterschieden wird. Es werden die nicht nativen Zellen im Folgenden als importierte Zellen bezeichnet.

Verwaltet werden die Agenten in Cellid- und Celldata-Arrays. Die Cellids stellen hierbei einen Hashwert dar, der eindeutig identifizierend für jede Zelle des Gridfeldes ist. Dieser wird im 2D Raum über $Zellhash(i) = Zelle.X(i) + Zelle.Y(i) * Grid.xDimension$ gebildet. Wird beim Start der Kollisionserkennung festgestellt, dass das Environment so groß ist, das diese Berechnung zu einem Overflow führt, wird stattdessen folgende Formel angewandt: $Zellhash(i) = (Zelle.X(i) * 7 + Zelle.Y(i) * 163)\%Integer.MaxValue$.

In das Cellid Array wird wieder nach dem Schema von Pabst et al. [33] befüllt, allerdings mit 4 statt 8 Feldern aufgrund des 2D Raumes. Weiterhin wird jede Manipulation der Reihenfolge am Cellid Array und auch auf dem Celldata Array mit den Daten zur aktuellen Zelle durchgeführt.

Die Cellid und Celldata Arrays werden für jeden Layer erstellt und getrennt verwaltet, somit gibt es keine Ebenen übergreifende Interaktionen zwischen den Cellarrays.

Im nächsten Schritt werden sie auf Basis des Cellid Arrays sortiert. Dafür wird ein Key-Value Sortieralgorithmus verwendet, bei dem die Cellids als Key fungieren und die Celldata als Value verwendet werden. Nach dem Sortieren sind die Elemente, welche sich in der gleichen Gridzelle befinden, direkt nebeneinander, aufgrund des identischen Cellid-Wertes. Deswegen muss nun das Cellid Array auf aufeinander folgende identische Werte geprüft werden, um zu erkennen, wenn sich mehrere Elemente in der gleichen Gridzelle befinden. Der Abschnitt 5.2.2 geht auf diesen Punkt im Detail ein.

Das Ergebnis dieser Prüfung ist eine Menge an Teillisten des Cellids Arrays, die jeweils eine Folge gleicher Cellids beschreiben. Für die Teillisten werden im darauf folgenden Schritt die Permutationen gebildet, da jedes Element der Liste mit den jeweils anderen kollidieren könnte. Am Ende des Schrittes wird eine Liste mit Tupeln aus je zwei Elementen der Teillisten gebildet, welche im letzten Schritt auf Kollision geprüft werden müssen. Dieses Vorgehen wird im Abschnitt 5.2.3 erläutert.

Die Environment-Schnittstelle bietet eine Variante der Explore-Funktion, bei der nur nach bestimmten Agententypen gesucht wird. Im Rahmen des MLGE wurde der Typenfilter ebenfalls auf die GPU ausgelagert. Dafür wurden die bisher freien zwölf Bit in der Celldata Datenstruktur verwendet. Über diesen Filter können die zwölf häufigsten Agententypen direkt auf der GPU gefiltert werden. Die Auswahl erfolgt basierend auf der Häufigkeit. Alle weiteren Agententypen filtert weiterhin CPU.

5.2.1 Reorder und Cellarrays

Dieser Abschnitt beschreibt das Vorgehen in den ersten beiden Schritten der Kollisionserkennung. Zu Beginn wird der OpenCL Kernel Reorder gestartet, dieser bekommt als Parameter die Formen der Agenten und die zugehörigen Objektids. Dort werden die Agenten basierend auf ihrer nativen Ebene umsortiert. In einem OpenCL Kernel ist es nicht möglich mehrdimensionale Arrays zu verwalten oder dynamisch Speicher zu erhalten. Da die Zuordnung auf die jeweiligen Ebenen in einem Kernel vorgenommen werden soll, muss daher das mehrdimensionale Array für die Ebenen auf eine eindimensionales abgebildet werden. Damit auch der Fall abgedeckt ist, dass sich alle Agenten auf derselben Ebene befinden muss pro Ebene ausreichend Speicher für alle Agenten vorgesehen sein. Es werden deswegen die Agenten der Ebene null an den Anfang des Arrays gelegt, die der Ebene eins haben einen Offset, der Anzahl der Agenten entspricht, der Offset der zweiten Ebene ist zweimal die Anzahl der Agenten und so weiter. Zusätzlich

wird noch für jede Ebene gespeichert, wie viele Agenten ihr zugeordnet wurden.

Aus den Daten werden im nächsten Schritt die Cellarrays erstellt. Aufgrund dessen, dass jetzt bestimmt ist, wie viele Agenten sich jeweils auf den Ebenen befinden, können die Cellarrays in der korrekten Größe angelegt werden. Anders als beim Reorder Kernel, der nur einmal aufgerufen wird, wird nun ein Kernel pro Ebene gestartet. Das folgt daraus, dass die Ebenen unabhängig voneinander betrachtet werden. Durch die Reduzierung auf ein zweidimensionales Gridfeld je Ebene, benötigt jeder Agent vier statt den acht Feldern im SLGE. Weiterhin wird die Phantom- und Homecell Logik von Pabst et al. angewandt, um die Anzahl der Kollisionsprüfungen zu reduzieren. Das Celldata Array hat den Datentyp „Long“ und ist damit in der Lage 64 Bit pro Feld zu Speichern. Die erste Hälfte des Arrays wird durch die ObjektId des zugehörigen Agenten belegt und auf den übrigen 32 Bit werden folgende Flags hinterlegt:

- Homecell
- Phantomcell
- Explore
- Nativ
- Unused
- Ghost
- Weitere freie Bits können für einen GPU-Seitigen Agenten-Typenfilter verwendet werden

Explore gibt an ob es sich um ein Explorieren eines Agenten handelt. Darüber werden andere Agenten in seinem Sichtfeld aufgefunden. Das hat zur Folge, dass das Exlore Objekt zwar mit anderen Agenten kollidieren kann und sie somit "findet“, jedoch anderen Agenten keine Kollisionsmeldung mit einem Exploreobjekt erhalten dürfen. Ebenso sollen Explore Objekte nicht mit anderen Explores kollidieren.
Nativ wird gesetzt, wenn es sich um die native Ebene des Agenten handelt. Für den Fall, dass ein Agent weniger als vier Gridfelder schneidet, werden die Übrigen mit Unused markiert und erhalten als Cellid den maximalen Integer-wert, damit sie beim Sortieren am Ende des Arrays landen.
Das Flag Ghost repräsentiert eine Besonderheit des MARS Life Frameworks. Einem Agenten können verschiedene Kollisionsverhalten zugewiesen werden. Ghost bedeutet, dass der Agent nicht mit anderen Agenten kollidiert, er kann jedoch durch ein Explore aufgefunden werden.

Zelle 1	Zelle 2	Kollisionsprüfung?
Home	Home	Ja
Home	Phantom	Ja
Phantom	Phantom	Nein

Tabelle 5.1: Nötige Kollisionsprüfungen im eindimensionalen Gridfeld

Zelle 1	Zelle 2	Kollisionsprüfung?
N_Home	N_Home	Ja
N_Home	N_Phantom	Ja
N_Home	I_Home	Ja
N_Home	I_Phantom	Ja
N_Phantom	N_Phantom	Nein
N_Phantom	I_Home	Ja
N_Phantom	I_Phantom	Nein

Tabelle 5.2: Nötige Kollisionsprüfungen im hierarchischem Gridfeld. Der Prefix N_ steht für native Zellen und I_ für importierte aus den unteren Ebenen. Prüfungen müssen nur von den nativen Zellen ausgehend durchgeführt werden

Das Schema der Cellarrays wurde auch beim SLGE verwendet, jedoch wurde da nur mit einer Ebene gearbeitet. Dadurch, dass mehrere Ebenen verwendet werden, muss die Logik der nativen- und importierten Agenten/Elemente mit abgebildet werden. Es hat jede Ebene damit vier unterschiedliche Arten von Zellen, die alle für das Bestimmen der benötigten Kollisionsprüfungen relevant sind. Zum einen sind das die nativen Home- und Phantomcells, dazu kommen noch die importierten Home- Phantomcells aus den unteren Ebenen.

Die Tabelle 5.2 zeigt, dass durch die Ebene deutlich mehr Fälle unterschieden werden müssen, als im eindimensionalen Fall. Zusätzlich ist es wichtig, dass die Agenten bereits beim Einordnen in die Cellarrays in einer bestimmten Reihenfolge abgelegt werden. Das folgt daraus, dass beim Erstellen der Kollisionslisten nach der Tabelle 5.2 die Kollisionstupel bestimmt wird. Es müssen die Agenten deswegen in der folgenden Reihenfolge im Cellarray liegen:

1. Native Homecells
2. Native Phantomcells
3. Imported Homecells für alle unteren Ebenen
4. Imported Phantomcells der unteren Ebenen

Zu beachten ist, dass atomare Aktionen zu vermeiden sind. Diese verlangsamen den Kernel stark, da bei jeder atomaren Aktion alle Kernel synchronisiert werden müssen, wie auch Elteir et al. [12] gezeigt haben. Deswegen wird jeden Agenten die Position im Cellarray rechnerisch

Native Homecells	Native Phantomcells	Importierte Homecells			Importierte Phantomcells		
Native Homecells	Native Phantomcells	Importierte Homecells von Unterebene 1	Importierte Homecells von Unterebene 2	.....	Importierte Phantomcells von Unterebene 1	Importierte Phantomcells von Unterebene 2	.....

Abbildung 5.3: Aufteilung der Elemente in den Cellarrays

ermittelt, anstatt über ein atomares Inkrementieren einen Zähler für die Arrayposition zu erhöhen.

Durch den Reorder-Kernel haben wir die Information, wie viele Agenten jede Ebene hat und somit auch die Anzahl an importierten Agenten. Auf der Basis werden die Arrayindizes der Cellarray bestimmt. Abbildung 5.3 zeigt, wie das Cellarray strukturiert wird. Die Struktur bringt den Vorteil, dass die Elemente sich nach dem Sortieren in der richtigen Reihenfolge befinden, um später mit Tabelle 5.2 die Kollisionssublisten erstellen zu können. Es werden die Kollisionssublisten im nachfolgenden Abschnitt detailliert erläutert.

5.2.2 Sortieren und Erstellen der Kollisionssublisten

Nachdem die Cellarrays erstellt wurden, müssen diese nun sortiert werden. Hierfür wird ein key-value Sort verwendet, welcher sowohl die Key- als auch die Value-Werte basierend auf dem Key sortiert. Aufgrund dessen, dass jede Ebene eigene Cellarrays hat, muss der Sort für jede Ebene durchgeführt werden.

Der nächste Schritt im Verfahren, ist das Erstellen von Kollisionssublisten. Dabei wird das sortierte Cellid-Array durchlaufen und es wird für jede Folge von gleichen Werten eine Subliste erstellt. Diese Sublisten sind Strukturen mit den folgenden Feldern:

- Startindex: Der Index im Cellid-Array bei dem die Subliste beginnt
- NHome: Anzahl der nativen Homecells in der Subliste
- NPhant: Anzahl der nativen Phantomcells in der Subliste
- IHome: Anzahl der importierten Homecells in der Subliste
- ObjCnt: Gesamtzahl der Elemente in der Subliste

Für das Erstellen der Sublisten, ist es notwendig, dass die Elemente vor dem Sortieren in der korrekten Reihenfolge, wie im Abschnitt 5.2.1 in die CellArrays geschrieben werden. Der verwendete Sort garantiert, dass die Reihenfolge der Values bei gleichen Keys nicht verändert

wird. Das führt dazu, dass native Zellen mit der gleichen Cellid immer vor den importierten Zellen stehen, aufgrund der Anordnung vor dem Sortieren. So kann beim Erstellen der Kollisionssublisten das Cellid Array nach aufeinander folgenden gleichen Cellids durchsucht werden. Durch die getrennten Ebenen kann hier ein stark an den Vorgang vom SLGE angelehnt werden. Aufgrund dessen, dass sich die Elemente bereits in der Reihenfolge: *NHome*, *NPhant*, *IHome*, *NPhant* abgelegt sind, muss nur die Anzahl hochgezählt werden, bis sich der Zelltyp ändert. Die Kollisionssublisten stellen die Basis dar, auf der im nächsten Schritt ermittelt für welche Elemente Detailprüfungen vorgenommen werden müssen. Um die Kollisionsprüfungen mit wenig Aufwand parallel berechnen zu können, werden aus den Kollisionssublisten Kollisionstupel gebildet, die jeweils zwei Elemente enthalten, die auf Kollision geprüft werden müssen. Das Vorgehen zum Erstellen der Kollisionstupel wird im nächsten Abschnitt beschrieben.

5.2.3 Permutationen und Kollisionsprüfungen

Mit den Kollisionssublisten haben wir die Information wer sich alles in gleichen Gridzellen aufhält. Theoretisch können Kollisionen zwischen Agenten, welche die gleiche Gridzelle schneiden, nicht ausgeschlossen werden. Die Folge daraus ist, dass jeder gegen jeden geprüft werden muss. Allerdings haben wir im Kapitel 5 festgestellt, dass die Prüfungen nur von den nativen Zellen ausgehen müssen und aus dem Abschnitt 5.2.1 geht hervor, dass Phantomcell gegen Phantomcell nicht geprüft werden muss. Konkret bedeutet das für die Kollisionslisten, dass die nativen Homecells gegen alle anderen geprüft werden müssen und danach die nativen Phantomcells gegen die importierten Homecells. Der folgende *C* Codeabschnitt zeigt die aktuelle Implementierung für das Bilden der Kollisionstupel.

```
...
int actIdx;
for(int i = threadIdx; i < boundary; i +=
   dConst.numThreadsPerBlock ){
 CollisionSublist colLst = narrowCheckSublist[i];
 // Order in CellidArray :
 // | nHome | nPhant | iHome | iPhant |
 int nHomeEnd = colLst.startIdx + colLst.nHome;
 int nPhantEnd = colLst.startIdx + colLst.nHome + colLst.nPhant;
 int iHomeEnd = colLst.startIdx + colLst.nHome + colLst.nPhant
    + colLst.iHome;
 // home collisions
 for(int j = colLst.startIdx; j < nHomeEnd; j++){
  for(int k = j+1; k < colLst.startIdx + colLst.objCnt; k++){
```

```
    actIdx = atomic_inc(tupleIdx);
    collisionCheckTuples[actIdx].obj1Idx = j;
    collisionCheckTuples[actIdx].obj2Idx = k;
   }
  }

  // Add phantom - import collisions
  for(int j = nHomeEnd ; j < nPhantEnd; j++){
   for(int k = nPhantEnd ; k < iHomeEnd; k++){
    actIdx = atomic_inc(tupleIdx);
    collisionCheckTuples[actIdx].obj1Idx = j;
    collisionCheckTuples[actIdx].obj2Idx = k;
   }
  }
}
...
```

Pro Kollisionssubliste wird ein GPU-Thread verwendet. Das ist eine performante Lösung, solange die Kollisionssublisten relativ klein sind. Dies ist der Fall, wenn das hierarchische Grid so gewählt ist, dass es gut zu den Agentengrößen passt. Es kann das Verfahren jedoch so erweitert werden, dass bei Erstellen der Sublisten direkt geprüft wird, ob die Liste eine zu definierende Maximalgröße überschreitet und in dem Fall die betroffene Liste separat speichert. Für diese großen Sublisten kann dann ein einzelner Kernel gestartet werden, der parallel die Permutationen dafür berechnet. Eine Möglichkeit wäre es die Permutationen für jedes Element in einem eigenen Thread zu berechnen.

Im letzten Schritt werden für jedes Kollisionstupel die Formen geladen, um auf der GPU die Detailprüfung durchzuführen. Zu beachten ist, dass nur Kreise als Agentenformen verwendet werden. Das kann jedoch erweitert werden, sodass die Kreise als Bounding Volume dienen. Für die Prüfung der konkreten Formen müsste dann einer weiterer Kernel entworfen werden, der diese Prüfung durchführt. Zusätzlich zur Formprüfung wird in dem Schritt auch auf Basis der Flags im Celldata Array gefiltert, sodass als Ergebnis nur valide Kollisionen ausgegeben werden.

5.2.4 Aufbereitung und Aufruf der Delegaten

Das Ergebnis des Kernels stellt eine Kollisionsliste mit Objektid Paaren dar, die jeweils kollidiert sind. Es muss nun jedem Agenten seine Kollisionsgegner mitgeteilt werden. Zu dem Zweck

wird jedem Agenten ein Set an Objektids zugeordnet. Dieses wird mithilfe der Kollisionsliste mit allen Kollisionsgegnern des jeweiligen Agenten gefüllt. Sobald die Sets erstellt sind, werden die Delegaten aufgerufen. Damit wird den Agenten mitgeteilt, ob und mit wem sie kollidiert sind.

5.3 Radix-Sort und Scan

Das Sortieren auf der GPU ist kein neues Problem, deswegen sollte hierfür eine externe Library verwendet werden. In der Recherche hat sich herausgestellt, dass keine der aktuell verfügbaren Librarys einen Sortieralgorithmus bietet, der die Anforderungen bezüglich Anzahl und Komplexität der Elemente erfüllt. Es wurde versucht den Radix-Sort einer Bibliothek entsprechend anzupassen. Das ist jedoch aufgrund der sehr komplexen Codestruktur sowie der teils mangelhaften Dokumentation, verworfen worden. Für maximale Flexibilität wurde eine eigene Implementierung des Radix-Sorts erstellt. Dadurch konnten Anpassungen an die benötigten Verwaltungsstrukturen vorgenommen werden, um eine bessere Zusammenarbeit mit der Spatialen Subdivision zu ermöglichen. Nachfolgend wird die Umsetzung detaillierter beschrieben.

Beim Radix-Sort handelt es sich um einen Algorithmus aus der Familie der zählenden Sortieralgorithmen, auch *Counting-Sorts* genannt. Zum Verdeutlichen der Funktionsweise wird das Sortieren eines Arrays mit Zahlenwerten genommen. Von jedem Wert werden zunächst nur die unteren Bits betrachtet. Für alle möglichen Werte der betrachteten Bits wird ein Bucket in Form eines Counters angelegt. Anschließend werden die Zahlenwerte des Arrays in die jeweils passenden Buckets einsortiert. In Abbildung 5.4 wird ein Beispiel für die Sortierung eines Datenarrays mit Hilfe des Radix-Sorts dargestellt. Dabei beschreibt *data* die Eingangswerte, *counts* sind die Buckets für die einzelnen Zahlenwerte und *offsets* ist die Präfixsumme des *counts* Arrays. Im ersten Schritt werden nur die Einerstellen ausgewertet, somit wird bei 31, 41 nur die 1 betrachtet, bei 58 entsprechend die 8. Auf Basis der Einerstellen werden die entsprechenden Counter im *counts* Array erhöht. Nachdem alle Eingangswerte im *counts* Array eingetragen wurden, wird das *offsets* Array gebildet. Es stellt die Präfixsumme von *counts* dar und beinhaltet dadurch die Offsets für die einzelnen Werte des *counts* Arrays. Berechnet werden die Offsets nach folgender Formel:

$$offsets(x) = offsets(x-1) + counts(x-1)$$

Beginnend mit bei $x = 1$ und mit $offset(0) = 0$. Im nachfolgenden Schritt wird das *counts* Array von Position 0 beginnend durchgegangen, bis ein Index erreicht wird, an dem der Wert

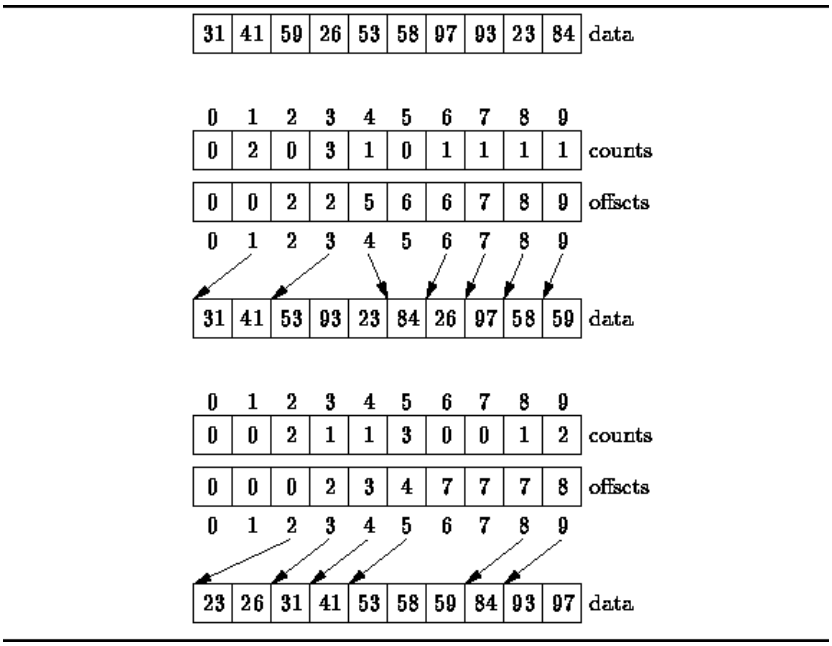

Abbildung 5.4: Radix-Sort Beispiel von Preiss[34]

größer als 0 ist. Dies ist an Position 1 der Fall, dann werden die entsprechenden Werte beginnend bei dem im *offsets* stehenden Index, in diesem Fall 0, in das Ergebnis-Array geschrieben. Der nächste relevante Counter ist die 3, dieser steht auf 3, aufgrund der 3 Zahlen die auf 3 enden im Datenarray. Bei *offsets* steht an der Position eine 2, damit werden die entsprechenden Zahlen beginnend bei Index 2 in das Ergebnis-Array geschrieben. Nach diesem Schema wird das gesamte Array durchgegangen. Im Ergebnis-Array sind die Werte nun entsprechend ihrer Einerstelle sortiert. Das Schema wird nun erneut für die Zehnerstelle durchgeführt. Der Radix-Sort ist danach abgeschlossen und das Datenarray ist sortiert.

Damit die GPU ihr Potenzial ausschöpfen kann, muss der Vorgang maximal parallelisiert werden, wofür sich der Ansatz von Grand [24] eignet. Dort werden die Workgroups der GPU zusätzlich noch in mehrere Threadgroups unterteilt, um die Aufgaben effektiver verteilen zu können. Um die Eingangswerte ohne zusätzliche Synchronisation ihren Buckets zuordnen zu können, wird pro Threadgroup für jede betrachtete Bitkombination ein Bucket erstellt. Die Anzahl der Buckets beträgt somit:

$$AnzahlBuckets = AnzahlBits * AnzahlWorkgroups * AnzahlThreadgroups$$

```
inline void PrefixLocal(__local uint* inout, int p_length, int numThreads){
//   if(get_local_id(0) == 0) return;
    __private uint glocalID = get_local_id(0);
    __private int inc = 2;

    // reduce
    while(inc <= p_length){

        for(int i = ((inc>>1) - 1) + (glocalID * inc)  ; (i + inc) < p_length ; i+= numThreads*inc){
            inout[i + (inc>>1)] = inout[i] + inout[i + (inc>>1)];
        }
        inc = inc <<1;
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    // Downsweep
    inout[p_length-1] = 0;
    barrier(CLK_LOCAL_MEM_FENCE);

    while(inc >=2){
        for (int i = ((inc>>1) - 1) + (glocalID * inc)  ; (i + (inc>>1)) <= p_length ; i+= numThreads*inc)
        {
            uint tmp = inout[i + (inc >>1)];
            inout[i + (inc >>1)] = inout[i] + inout[i + (inc >>1 )];
            inout[i] = tmp;
        }
        inc = inc>>1;
        barrier(CLK_LOCAL_MEM_FENCE);
    }
}
```

Abbildung 5.5: Eigene Implementierung der OpenCL Präfixsumme nach Blelloch [8]

Das Array der Eingangsdaten wird gleichmäßig auf die Threadgroups verteilt und in die entsprechenden Buckets einsortiert. Um maximale Performance zu erreichen werden alle Operationen nach Möglichkeit außerhalb der Schleife durchgeführt. Dadurch müssen in jedem Schleifendurchlauf noch fünf Additionen und zwei Multiplikationen durchgeführt werden. Durch die Logik der inneren For-Schleife werden die Threads innerhalb einer Threadgroup synchronisiert, wodurch kein globaler Mutex oder ein atomares Inkrement benötigt wird. Um die Elemente in der korrekten Reihenfolge entnehmen zu können, wird eine Präfixsumme über die Buckets mit identischem Bitwert gebildet. Die Präfixsumme enthält die Offsets für die Buckets. Mit diesen ist es möglich die Buckets im nächsten Schritt parallel auszulesen, um das Ergebnisarray zu füllen.

Abbildung 5.5 zeigt den entwickelten OpenCL Code zum Berechnen der Präfixsumme innerhalb einer Workgroup. Er wurde nach dem Schema des Blelloch-Scans [8] entwickelt und angepasst, sodass die Präfixsumme von einer Workgroup parallel berechnet wird.
Im nächsten Schritt werden die Werte aus den Buckets genommen und basierend auf den Präfixwerten in das Ergebnisarray geschrieben. Wie bei der herkömmlichen Präfixsumme wird der Vorgang so oft wiederholt, bis die hochwertigsten Bits erreicht wurden.

6 Integration

Die Integration der Kollisionserkennung durch die neuen Environments muss über mehrere Ebenen geschehen. Als direkte Schnittstelle dient das Async Environment Interface, welches beide entworfenen Environments implementieren. Um die Umstellung auf das asynchrone Arbeiten für den Modellentwickler zu vereinfachen, wurde ebenfalls das SRA Pattern für alle Environments, die das Async Environment Interface unterstützen, umgesetzt. Aufgrund der asynchronen Funktionsweise, unterscheidet sich das Verhalten der Asyc Agents leicht von dem der Basic Agents. Deswegen wurde mit dem Antelopes and Lions Modell ein Beispiel für die Umsetzung eines Modells mit den asynchronen Environments geschaffen, an dem sich die Modellentwickler orientieren können. Neben direkten Integration in MARS Life mussten noch einige Änderungen bei der Ausführungsumgebung von MARS Life vorgenommen werden. Der MARS Life Core ist in der Programmiersprache C# geschrieben und wird mit .NET Core ausgeführt. .NET Core ist eine quelloffene Software-Plattform für die Ausführung von .NET Code, die auch Unix basierte Systeme wie Linux oder OSX unterstützt. Um die GPU Environments direkt in MARS Life integrieren zu können, müssen sie in C# geschrieben werden und unter .NET Core ausführbar sein. Nativ wird von OpenCL jedoch nur C++ unterstützt, weswegen hier mit einer Wrapper Library gearbeitet werden muss. Die Details der Umsetzung werden im Abschnitt 6.4.1 beschrieben. Der .NET Core Code von MARS Life wird wiederum in einem Dockercontainer ausgeführt. Daraus folgt, dass die GPU im Docker Container verfügbar sein muss, damit diese im nächsten Schritt auch in .NET Core genutzt werden kann.

6.1 Async Environment Interface

Um die Environments auf GPU Basis zu integrieren wurde ein neues Interface entworfen. MARS Life hat bisher mit dem IESC Interface nur eine Schnittstelle für Environments mit Formen und Kollisionserkennung.

Dieses ist aufgrund des synchronen Aufbaus nicht für die Verwendung mit einem GPU Environment geeignet. Deswegen wurde eine neues Interface entwickelt, welches auf eine asynchrone Abarbeitung der Anfragen ausgelegt ist. Die Neuentwicklung des Interfaces bot die Möglichkeit einen weiteren Aspekt anzupassen, der bisher für die Verteilung Nachteilhaft

war. Dabei handelt es sich um die Verwaltung der spatialen Daten der Agenten. Ebenso wurde das Verhalten des Environments bei Kollisionen angepasst.

Spatiale Daten im MARS Life Kontext enthalten neben dem Identifikator des Agenten auch dessen Form, Ausrichtung und Position. Beim IESC Interface wurden bei jeder Anfrage die vollständigen spatialen Daten in Form eines ISpatialEntity Objekts an das Environment übergeben. Es wurden die Daten als Parameter per Referenz weitergegeben und anschließend vom Environment manipuliert. Die Move Methode wird beispielsweise so durchgeführt, dass das Environment auf Basis der spatialen Daten eine Kollisionsprüfung an der Zielposition vornimmt und anschließend den Positionswert der übergeben spatialen Daten entsprechend manipuliert. Um die Kollisionsprüfung mit den anderen Objekten des Environments durchführen zu können muss es zusätzlich sich die spatialen Daten aller Objekte speichern. Das führt dazu, dass die spatialen Daten der Agenten doppelt vorgehalten werden müssen. Zum einen muss der Agent seine spatialen Daten wissen, um Anfragen an das Environment durchführen zu können und zum anderen muss das Environment die spatialen Daten aller Agenten speichern um auf Kollisionen prüfen zu können. Ein weiterer Nachteil von diesen Aufbau ist, dass die spatialen Daten an verschiedenen Stellen manipuliert werden, was zu ungewollten Nebenläufigkeitseffekten führen kann.

Mit der neuen asynchronen Schnittstelle wird dafür eine anderes Schema verwendet. Anstatt das jeder Agent seine spatialen Daten verwaltet, werden diese nun zentral im Environment gehalten. Zu Beginn einer Simulation melden sich die Agenten mit ihrer Startposition und Form beim Environment an. Ab diesem Zeitpunkt werden die spatialen Daten nur noch im Environment vorgehalten. Sämtliche Bewegungs- und Explorationsanfragen werden nur noch über den Identifikator des Agenten durchgeführt. Wenn ein Agent spatialen Daten benötigt, muss er diese über seinen Identifikator beim Environment anfragen. Diese Änderung hat den Vorteil, dass die spatialen nur noch an einer zentralen Stelle vorgehalten werden. Aufgrund des stark verteilten Aufbaus von MARS müssen dadurch weniger Daten über das Netzwerk gespiegelt werden. Ebenfalls zu beachten ist hierbei, dass keine ungewollten Seiteneffekte auftreten können, da nur Kopien der spatialen Daten vom Environment herausgegeben werden.

Ein weiterer wichtiger Aspekt ist die Umsetzung der asynchronen Aufrufe. Dafür bietet C# mehrere Möglichkeiten.

Asyc und Await ist eine davon. Dabei wird beim Aufruf der Methode mittels Async ein neuer Thread gestartet. Await erlaubt es anschließend auf die Beendigung des Threads zu warten. Im Kontext des Environments würde das bedeuten, dass die Environment-Methoden von eigenen Threads aufgerufen werden, auf dessen erfolgreiche Durchführung anschließend jeder Agent

mit Await wartet. Zu beachten ist dabei, dass die Auswertung der Methodenaufrufe gebündelt am Ende eines Ticks stattfindet, mit der Folge, dass jeder Aufruf blockiert werden muss, bis die Prüfung erfolgt ist. Für jeden blockierten Thread muss der Kontext gesichert werden, wodurch eine große Menge an Speicher benötigt wird. Deswegen eignet sich dieser Ansatz nicht für diesen Verwendungszweck. Bei den hohen Agentenzahlen, die in MARS simuliert werden, benötigt dieser Ansatz zu viele Ressourcen.

Eine weitere Möglichkeit für die asynchrone Umsetzung sind Events. Dabei könnte im Environment ein Event erstellt werden, welches dann gefeuert wird, wenn das Environment seine Prüfungen abgeschlossen hat. In dem Fall würde dann jeder Agent ein unabhängiges Event bekommen, um sich anschließend vom Environment seine Ergebnisse abzuholen. Dies wäre problematisch, da nach dem Event alle Agenten gleichzeitig aktiv werden und ihre Ergebnisse beim Environment anfragen würden. Ein weiteres Problem entsteht bei Aktionen, die nicht direkt einem Agenten zugeordnet sind, wie zum Beispiel das Explorieren. Hier müsste eine neue Methodik entworfen werden, um die Ergebnisse des Aufrufs abzufragen, da es keinen Agenten Identifikator gibt, der dafür verwendet werden kann. Der Vorteil bei Events, ist, dass deutlich weniger Speicher benötigt wird, da keine Thread Kontexte gespeichert werden müssen.

Die letzte Methodik zum Vergleich herangezogen wird, sind Delegaten. Ein Delegat stellt einen Verweis auf eine Methode mit einer bestimmten Parameterliste dar. Im Kontext des Environments, kann dies so verwendet werden, dass jede Methode als Argument einen Delegaten erwartet, über den das Ergebnis geliefert werden kann. Das Environment kann, nachdem es die Ergebnisse für alle Aktionen bestimmt hat, den Agenten über die Delegaten das ihr Ergebnis mitteilen. Sie können zentral über das Environment aufgerufen werden, welches dafür eine flexible Anzahl an Threads verwenden kann. Dabei ist zu beachten, dass die Agenten unabhängigen Methoden kein Problem mehr darstellen, da die Ergebnisse direkt über die Delegaten zurückgegeben werden und nicht über eine Anfrage an das Environment.

Die Wahl für die Umsetzung ist auf Delegaten gefallen. Das Schema passt am Besten zu dem gegebenen Anwendungsfall und lässt sich Aufgrund der zentralen Verwaltung am besten skalieren.

Zuletzt wurde mit der neuen Schnittstelle das Verhalten des Environments bei Kollisionen angepasst. Für die neuen Environments wurde entschieden, dass jede Bewegung eines Agenten durchgeführt wird, auch wenn der Agent dadurch mit einem Anderen kollidiert. Das Environment hat nicht die Aufgabe Kollisionen selbst aufzulösen, da der Umgang mit Kollisionen sehr stark von dem Szenario abhängt. Der Agent wird speziell für jedes Modell entworfen und

kann somit am besten Entscheiden, was im Fall einer Kollision zu tun ist. In den synchronen Environments wurde dies zum Teil anders umgesetzt. Die Environments haben den Agenten bei einer Kollision auf seine vorherige Position zurückgesetzt. Allerdings ist dieses Verfahren bei einem asynchronen Environment sehr problematisch.

Dadurch dass alle Bewegungen gleichzeitig durchgeführt werden, könnte im selben Moment ein anderer Agent auf die vorherige Position des kollidierten Agenten gelaufen sein, wodurch das zurücksetzen zu einer weiteren Kollision führen würde. Bei vielen Agenten in einem System kann es sein, dass die Agenten in einem Tick um mehrere Positionen zurückgesetzt werden müssen. Das würde zum einen dazu führen, bei jeder Kollision eine neue Kollisionsprüfung gestartet werden muss, weil die Agenten durch das zurücksetzen eine andere Position haben. Zum Anderen ist das Zurücksetzen des Agenten in vielen Modellen nicht das gewünschte Verhalten. Wird zum Beispiel ein Räuber-Beute Modell betrachtet, bei dem ein Räuber mit seiner Beute kollidiert, soll dieser nicht auf die vorherige Position zurückgesetzt werden, sondern die Beute fressen, mit der er kollidiert ist. Deswegen ist die Entscheidung darauf gefallen die Kollisionsbehandlung den Agenten zu überlassen.

6.1.1 Synchron zu asynchron Wrapper

Die GPU-Environments sind die asynchronen Environments. Jedoch wurde entschieden die bestehenden, getesteten Environments ebenfalls im asynchronen Kontext verwendbar zu machen. Dadurch können die GPU-Environments besser evaluiert werden. Des Weiteren erlaubt dies bessere Vergleichstests der Environments.

Zu diesem Zweck wurde eine Wrapper Klasse geschrieben, welche die synchronen Environments auf das asynchrone Interface adaptiert. Dabei wurde stark darauf geachtet, den durch den Wrapper entstehenden Mehraufwand bei den Berechnungen möglichst gering zu halten.

Die synchronen Environments werten alle Aufrufe direkt aus. Um trotzdem ein asynchrones Verhalten abzubilden, werden alle Aufrufe innerhalb eines Ticks zwischengespeichert. Mit dem Commit werden diese dann alle an das synchrone Environment weitergeleitet. Zusätzlich zu den Listen werden noch die spatialen Daten der Agenten in der Wrapper Klasse gespeichert, da die synchronen Environments dies nicht tun.

Das heißt, dass für jeden Methodenaufruf folgender Mehraufwand gegenüber der standardkonformen Verwendung der synchronen Environments entsteht:

1. Speichern der Methodenparameter in einer Liste
2. Auslesen des Parameter aus der Liste beim Commit
3. Aufruf des Delegaten um den Rückgabewert dem Agenten mitzuteilen

Wobei der dritte Punkt für ein asynchrones Environment unvermeidbar ist. Alle zusätzlichen Aktionen haben eine geringe Komplexität, sodass sie um Vergleich zur Kollisionserkennung vernachlässigbar sind.

6.2 Async Agents

Damit die Modellentwickler weiterhin nach dem Sense Reason Act (SRA) Pattern arbeiten können, wurde eine asynchrone Variante der Basic Agents entworfen. Nach außen besitzt, er die gleiche Schnittstelle wie der Basic Agent, die intern auf das Async Environment Interface umgebrochen wird. Aufgrund dessen, dass das Act in jedem Tick die letzte Aktion des Agenten darstellt, kann die asynchrone Ausführung der Bewegung ohne Änderung des Schemas umgesetzt werden. Das Commit des Environments, bei dem alle Move- und Exploreanfragen ausgewertet werden, wird im Posttick und somit immer kurz nach dem Agententick berechnet. Dieser wird aufgerufen, nachdem der Tick aller Agenten durchgeführt wurde. Es wird somit die Bewegungsaktion, welche die letzte Aktion im SRA Pattern ist, im Posttick evaluiert. Die Auswertung erfolgt über den Movementsensor, dessen Delegat dem Environment bei der Bewegung übergeben wird. Über ihn kann sich der Agent im darauffolgenden Tick seine neue Position abholen.

Der einzige Unterschied aus Sicht des Modellentwicklers zu den synchronen Agenten besteht bei dem Sense Schritt des SRA Patterns. Im Environments-Kontext kann das Sense ein Explorieren darstellen, welches gemeinsam mit dem Move im Posttick berechnet wird. SRA gibt jedoch Sense als ersten Schritt des Ablaufs an. Um weiterhin nur ein Commit der Environments durchführen zu müssen, muss das Explorieren gemeinsam mit dem Move am Ende des vorherigen Ticks durchgeführt werden. Dadurch entsteht der folgende Ablauf:

- Haupttick
  - Sense: Auslesen der Move und Explore Sensoren
  - Reason: Hauptaktion des Agenten auf Basis der Sense Werte
  - Vorbereiten des Sense: Explore Aufruf
  - Act: Aktion des Agenten, kann eine Bewegungsaktion sein
- Posttick
  - Environment Commit: Berechnen der Kollisionen und Auswerten der Explores

Act und Sense folgen direkt aufeinander, deswegen hat der Agent auch die gleiche Informationsbasis vor den Aktionen. Der Agent kennt sein Ziel und kann somit auch gleichzeitig mit

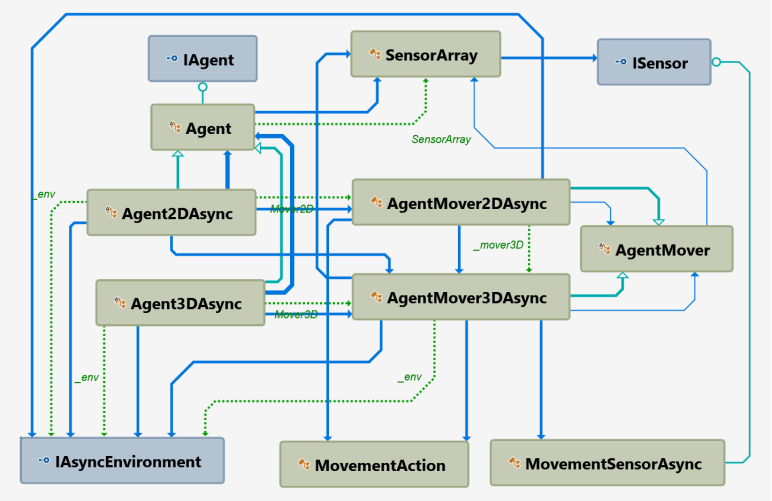

Abbildung 6.1: Abhängigkeitsgraph für das async Agents Paket

dem Move das Explore an seiner Zielposition aufrufen. Das Ergebnis kann dann weiterhin im Sense Schritt zu Beginn des Ticks über den Sensor abgefragt werden.

Aufgrund des angepassten Verhaltens bei Kollisionen stellt die im Act festgelegte Position auch immer die Position dar, in der sich der Agent im nächsten Tick befindet. Damit steht die neue Position des Agenten bereits vor dem Sense fest und es kann das entsprechende Explore aufgerufen werden.

Das daraus resultierende Async Agent Paket mit allen Klassen wird auf Abbildung 6.1 gezeigt. Es bietet alle Funktionen für das asynchrone Environment, die das Basic Agent Paket für das synchrone Environment bietet. Zu beachten ist, dass es auch die *Agent* Basisklasse implementiert, wodurch die Async Agents ohne weitere Anpassungen mit den übrigen Komponenten von MARS Life interagieren können.

6.3 Beispielmodell Antelopes and Lions

Mit dem asynchronen Environment wird eine neue Environmentart in MARS Life eingeführt. Um ein Beispiel für dessen Verwendung zu haben, wurde ein Demonstrationsmodell entwickelt. Die Entscheidung ist auf ein simples Räuber-Beute Modell gefallen, welches es, in ähnlicher Ausführung, bereits als Beispiel für das synchrone Environment gibt.

Das Modell hat drei unterschiedliche Agentenarten mit den entsprechenden Eigenschaften:

Attribut	Beschreibung
Sichtradius	Sichtweite des Agenten
Form	Ausdehnung des Agenten
Energy	Aktueller Energiewert des Agenten
Hunger	Hunger des Agenten
Speed	Bewegungsgeschwindigkeit des Agenten

Tabelle 6.1: Auflistung der Attribute des Lion Agenten

Löwen

Die Löwen können sich frei in der Umwelt bewegen und sind dabei auf der Suche nach Antilopen. Sobald sie beim Explorieren eine Antilope entdecken und ihr Hunger nicht gedeckt ist, laufen sie direkt zur Antilope. Haben sie keinen Hunger, oder ist keine Antilope in Sichtweite bewegen sie sich zufällig. Tabelle 6.1 zeigt die Attribute, die ein Löwenagent besitzt. Der Energiewert des Löwen kann als Lebensenergie gesehen werden. Er hat einen festen Startwert und sinkt über die Zeit. Wenn der Löwe frisst, steigt sein Energiewert um den Nahrungswert der gefressenen Antilope. Fällt der Energiewert auf 0, stirbt der Löwe. Über den Hunger wird festgelegt, ob sich der Löwe auf die Jagt macht. Nach erfolgreicher Jagt ist der Hunger gestillt und steigt danach wieder über die Zeit.

Antilopen

Bei den Antilopen handelt es sich um dynamische Agenten, die sich durch die Umwelt bewegen. Dabei Explorieren sie in jedem Tick ihre Umgebung auf der Suche nach Grass, welches sie als Nahrung benötigen. Wenn sie Grass in ihrer Nähe gefunden haben und der Hunger groß genug ist, bewegen sie sich direkt dort hin, ansonsten laufen in eine zufällige Richtung. Der Antilopenagent ist mit auf Tabelle 6.2 gezeigten Attributen versehen. Energie hat bei der Antilope das gleiche Verhalten wie beim Löwen. Über den FoodValue wird der Nahrungswert der Antilope angegeben. Dieser bestimmt wie stark der Energiewert des Löwen steigt, wenn er eine Antilope frisst.

Grass

Die Grass-Agenten sind statisch und werden in festen Intervallen reproduziert. Sie erhalten pro Tick einen leicht erhöhten Nahrungswert, bis sie von den Antilopen gefressen werden. Abgesehen davon verhalten sie sich passiv. Auf Tabelle 6.3 sind die Attribute des Grassagenten zu finden.

Attribut	Beschreibung
Sichtradius	Sichtweite des Agenten
Form	Ausdehnung des Agenten
Energy	Aktueller Energiewert des Agenten
Hunger	Hunger des Agenten
Speed	Bewegungsgeschwindigkeit des Agenten
FoodValue	Bestimmt den Nahrungswert des Agenten

Tabelle 6.2: Auflistung der Attribute des Antilope Agenten

Attribut	Beschreibung
Form	Ausdehnung des Agenten
FoodValue	Bestimmt den Nahrungswert des Agenten

Tabelle 6.3: Auflistung der Attribute des Grass Agenten

Das Modell besitzt ein zentrales Environment, in dem sich alle Agenten befinden. Dieses erstellt zu Beginn der Simulation eine feste Anzahl an Löwen, Antilopen und Grass. Im Laufe der Simulation bewegen sich die Löwen und Antilopen frei im Environment.

Zusammengefasst jagen Löwen Antilopen, die wiederum Grass fressen, welches regelmäßig nachwächst. Für die Umsetzung des Modells wird sowohl das noch asynchrone Environment-Interface verwendet, als auch das neue Async Agent Paket. Es wurde bewusst ein Modell gewählt, welches in ähnlicher Form im synchronen Kontext existiert, sodass die Entwickler, beim Vergleich der beiden Modelle, die Unterschiede zwischen der asynchronen und synchronen Vorgehensweise besser nachvollziehen können.

Ein weiterer Vorteil des neuen Modells ist, dass es durch async Agents und dem entworfenen Wrapper mit allen Environments durchgeführt werden kann. Das Antelopes und Lions Modell stellt damit eine gute Möglichkeit dar, die kartesischen Environments zu vergleichen.

6.4 Ausführungsumgebung

Um OpenCL Code im MARS Kontext ausführen zu können, mussten einige Anpassungen an der Ausführungsumgebung vorgenommen werden. Der Kern von MARS Life ist in C# geschrieben. In diesen muss das Environment integriert werden, da es viel mit den Agenten interagiert, welche sich ebenfalls dort befinden. Eine alternative ist das Environment als eigenständigen Microservice umzusetzen. Jedoch folgt daraus, dass die Kommunikation zwischen Agenten und Environment über das Netzwerk geschehen müsste. Aufgrund der engen Verzahnung von

Agent und Environment hätte das eine starke Belastung des Netzwerks zur Folge. Deswegen ist das Environment direkt als Komponente in MARS Life integriert.

Diese direkte Integration hat zur Folge, dass im Kontext von MARS Life die GPU erreichbar sein muss. Um zu verdeutlichen, was dafür benötigt wird, wird anschließend kurz die Architektur von MARS erläutert. MARS ist in einer Mircoservice Architektur [30] aufgebaut. Es gibt eine Vielzahl an Services die sich um den MARS Life Core ranken. Jeder Service wird in einem eigenen Docker Container gestartet. Docker selbst ist ein System welches es erlaubt Container zu starten [29]. Bei den Docker Containern handelt es sich um leichtgewichtige virtuelle Maschinen auf Linux Basis. Zum Organisieren der Microservices in der MARS Cloud wird Kubernetes [6] verwendet. Kubernetes erlaubt es Docker Container über mehrere Nodes zu verwalten, es ist eine weitere Abstraktionsschicht über Docker.

Für diese Arbeit relevant ist aufgrund der direkten Integration nur MARS Life Core. Auf die Ebenen Docker und .NET Core wird in den nachfolgenden Kapiteln genauer eingegangen. Der GPU-Support von Kubernetes befindet sich aktuell noch im Alpha Stadium, weswegen dort noch keine Schritte vorgenommen wurden. Allerdings ist der Code der Environments so aufgebaut, dass er mit einem beliebigen OpenCL Device funktioniert. Sobald im Kubernetes Cluster ein OpenCL Device verfügbar ist, können die GPU Environments dort verwendet werden.

Um den C# Code unter Linux ausführen zu können, setzt MARS Life auf .NET Core. Dabei handelt es sich um die Plattformunabhängige Plattform für .NET Code und damit auch C#. Zum aktuellen Zeitpunkt ist .NET Core noch in einem relativ frühen Stadium, weswegen nur eine geringe Anzahl an Bibliotheken zur Verfügung steht. Zum Beispiel existiert noch keine .NET Core kompatible Bibliothek für OpenCL, im Abschnitt 6.4.1 wird darauf im Detail eingegangen.

6.4.1 Portierung der OpenCL Library zu .NET Core

Zu beginn der Entwicklung des GPU-Environments, wurde nativer .NET Code verwendet, ohne .NET Core. Unter Linux wurde auf Mono gesetzt [1]. Mono bietet .NET Unterstützung im vollen Umfang. Alle .NET Bibliotheken sind grundsätzlich unter Mono nutzbar. Allerdings hat sich herausgestellt, dass die Ausführung des Codes unter Mono zu starken Einbußen seitens der Performance führt. Eine Ausführung des gleichen Codes unter nativem Windows .NET ist um ein vielfaches schneller.

Deswegen wurde vor einiger Zeit entschieden nicht weiter auf Mono zu setzen und den Linux Support einzustellen. Mit dem Release von .NET Core wurde der Support wieder aufgenommen. Es hat sich herausgestellt, dass die Ausführung darüber deutlich stabiler und performanter ist als unter Mono. Daraufhin wurde MARS Life core auf .NET Core umgestellt, um erneut

Plattformunabhängigkeit zu erreichen. Anders als Mono wird unter .NET Core nur ein Teil der Standardbibliotheken unterstützt, was dazu führt, dass einige externe Bibliotheken nicht direkt portierbar sind.

Für eine direkte Integration des GPU-Environments in den MARS Life Core ist es deswegen notwendig, dass der OpenCL Code unter .NET Core ausgeführt werden kann. Dafür wurden mehrere OpenCL Bibliotheken betrachtet und deren Kompatibilität zu .NET Core geprüft. Dabei hat sich herausgestellt, das die bisher verwendete Bibliothek OpenCl.NET [5] sich nicht ohne großen Aufwand auf .NET Core portieren lässt. Ein weiterer Kandidat war CUDAfy [38], welches sowohl CUDA als auch OpenCL Integration bietet. Jedoch wären bei einer vollständigen Portierung von CUDAfy Änderungen an mehr als 2000 Codezeilen nötig, weswegen auf eine vollständige Portierung verzichtet wurde. CUDAfy nutzt für die Hardwarenahe Umsetzung des OpenCL Parts die Cloo Bibliothek. Eine Analyse von Cloo hat ergeben, dass es zu großen Teilen .NET Core kompatibel ist. Von allen betrachteten Bibliotheken, ist Cloo die einzige, von der mehr als 90% ohne Anpassungen in .NET Core verwendet werden können. Die übrigen Teile wurden im Rahmen der Arbeit auf .NET Core portiert, sodass sie für die Entwicklung des GPU-Environments verwendet werden konnte. Um nachfolgende Projekte zu erleichtern, wurde

6.4.2 Notwendige Anpassungen bei der Docker Umgebung

Neben .NET Core stellt Docker eine weitere Hürde dar, die überwunden werden muss, um OpenCL Code in der MARS Life Umgebung auszuführen. Docker Container sind leichtgewichtige Virtuelle Maschinen, die den Kernel des Hostsystems nutzen. Es teilen sich die Docker Container auf einem Host die nativen Ressourcen. Dadurch ist es grundsätzlich möglich in einem Docker Container die GPU zu erreichen. Jedoch muss dem Container der Zugriff auf das GPU-Device explizit erlaubt werden. Weiterhin ist zu beachten, dass das Betriebssystem in dem Container die gleichen Treiber für die GPU nutzen muss, wie das Hostsystem. Ist dies gegeben, kann im Docker Container die GPU verwendet werden.

Um MARS Life in Docker mit GPU-Unterstützung auszuführen, wurde ein entsprechender Container erzeugt, bei dem die korrekten GPU Treiber und .NET Core vorinstalliert sind. Dafür wurde ein Dockerfile erstellt, welches den Aufbau des Containers beschreibt.

```
FROM ubuntu:16.04

ENV CUDA_RUN https://developer.nvidia.com/.....

RUN apt-get update
```

```
RUN apt-get install -q -y wget
RUN apt-get install -q -y build-essential
RUN apt-get install -q -y module-init-tools

RUN cd /opt && \
wget $CUDA_RUN && \
mv cuda_9.0.176_384.81_linux-run cuda_9.0.176_384.81_linux.run
   && \
chmod +x *.run && \
mkdir nvidia_installers && \
./cuda_9.0.176_384.81_linux.run
   -extract=‘pwd‘/nvidia_installers && \
cd nvidia_installers && \
./NVIDIA-Linux-x86_64-384.81.run -s -N --no-kernel-module

RUN cd /opt/nvidia_installers && \
./cuda-linux.9.0.176-22781540.run -noprompt
RUN sh -c ’echo "deb [arch=amd64]
   https://apt-mo.trafficmanager.net/repos/dotnet-release/
   xenial main" > /etc/apt/sources.list.d/dotnetdev.list’
RUN apt-get install -q -y apt-transport-https
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80
   --recv-keys 417A0893
RUN apt-get update
RUN apt-get install -q -y dotnet-dev-1.0.4
# Ensure the CUDA libs and binaries are in the correct
   environment variables
ENV LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda-9.0/lib64
ENV PATH=$PATH:/usr/local/cuda-9.0/bin
```

Der obige Code stellt den Inhalt des Dockerfiles dar, das einen Container mit CUDA Treibern und einer .NET Core Installation erstellt. Aufgrund der einheitlichen Treiberstruktur der NVIDIA Grafikkartentreiber, kann der gleiche Treiber für einen Großteil der NVIDIA Grafikkarten verwendet werden. Für den Support anderer Grafikkarten müsste ein neuer Container mit den entsprechenden Treibern erstellt werden. Der OpenCL Code der GPU-Environments muss hierfür nicht angepasst werden, da OpenCL nativ herstellerunabhängig ist.

7 Experimente

Um das neue SLGE und MLGE zu evaluieren, wurden einige Experimente vorgenommen. Im Rahmen dessen, werden die neuen Environments, mit der GPU-Basierten Kollisionserkennung, mit den bestehenden verglichen. Allerdings wird der Fokus auf die kartesischen Environments gelegt, um vergleichbare Ergebnisse hervorzurufen. Es wurde ein direkter Vergleich der Kollisionserkennung in unterschiedlichen Szenarien durchgeführt. Zusätzlich wurden einige Tests mit dem Antelopes and Lions Modell durchgeführt, um den Nutzen in einem konkreten Szenario zu zeigen. Dabei wird für die synchronen Environments der entworfene Wrapper genutzt, sodass der gleiche Modellcode für alle Environmentvarianten verwendet werden kann.

7.1 Vergleichstests der Environments

Die Tests für den direkten Performancevergleich der Environments werden unter einer Windowsumgebung vorgenommen. Dafür wird ein PC mit einer Intel Core I7 4770K CPU, 16GB DDR3 RAM und einer AMD Radeon R9 290 GPU verwendet. Um einen möglichst aussagekräftigen Vergleich der Komponenten zu erhalten, wird ausschließlich die Kollisionserkennung betrachtet. In dem Rahmen wurden mehrere Tests entworfen, um die einzelnen Stärken und Schwächen der Environments zu zeigen. Als Testobjekte werden Kreise genutzt, da diese auch in vielen Modellen, die das MARS Life Framework nutzen, verwendet werden. Für die kommenden Tests wird jeweils die Environment Methode „Add“ verwendet. Diese fügt das Element dem Environment hinzu und prüft ob es beim Hinzufügen kollidiert ist. Dabei wird die Position der Agenten zufällig bestimmt. Damit die Ergebnisse vergleichbar sind, wird für alle Tests ein Pseudozufallszahlengenerator mit identischem Startwert verwendet. Weiterhin ist die Größe des Environments zu beachten. Abhängig von der Anzahl an Elementen wird die Environmentgröße skaliert, um eine möglichst konstante Kollisionswahrscheinlichkeit zu erreichen. Bei den kommenden Tests wurde das Environment so skaliert, dass zwischen 10 und 20 Prozent der Agenten kollidieren.

Neben den beiden GPU Environments werden in den Tests auch die Werte von zwei CPU Environments, *Distributed ESC* und *Tree ESC*, aufgenommen. Die *Distributed ESC* nutzt intern

eine BVH Implementierung und kann verteilt verwendet werden. Den Kern der *Tree ESC* stellt ein Octree dar, welcher gleichzeitig auch der Namensgeber dieser Environmentsvariante ist.

Für diese Environments wurden insgesamt vier unterschiedliche Testszenarien entworfen:

1. Konstante Größe, steigende Anzahl an Elementen
2. Leicht variierende Größe, steigende Anzahl an Elementen
3. Stärker variierende Größe, steigende Anzahl an Elementen
4. Konstante Anzahl an Elementen, steigende Varianz der Elementgröße

Nachfolgend werden diese der Reihenfolge nach detailliert analysiert.

7.1.1 Elemente mit konstanter Größe

Für das erste Testszenario wurden Elemente mit identischer Ausdehnung verwendet und das Verhalten bei Erhöhung der Elementanzahl bei den einzelnen Environments beobachtet. Durch die identische Form aller Elemente, sollte diese Variante besonders gut auf den GPU Environments skalieren, da die Vorauswahl der Kollisionsprüfungen auf Gridbasis in dem Fall sehr effektiv ist.

Abbildung 7.1 zeigt den Graphen zu diesem Testlauf. Als Grundlage für den Graph dienen mehrere Messungen beginnend bei 50 Elementen bis hin zu 1.500.000 Elementen, die dem Environment hinzugefügt werden. Diese stellen die X-Achse im Graphen dar. Aufgrund der großen Spanne an Werten wurde eine doppelt logarithmische Darstellung gewählt. Es hat sich gezeigt, dass Break-Even-Point, ab dem ein GPU Environment schneller ist bei ca. 1000 Elementen liegt. Von dort an setzen sich SLGE und MLGE deutlich ab mit einem Fünftel der Rechendauer bei 5000 Elementen, ab 10000 ein Achtel und bei 1.500.000 Elementen sind sie um einen Faktor 11 schneller als die schnellste CPU Variante. Ein Unterschied zwischen den beiden GPU Environments ist hier nicht zu erkennen. Aufgrund der identischen Elementform verhalten sich die Environments identisch. Das MLGE erhöht die Anzahl der Ebenen nur, solange es auch Elemente gibt, die den jeweiligen Ebenen zugeordnet werden können. In diesem Fall gehören alle Elemente derselben Ebene an, wodurch auch nur eine Ebene erstellt wird.

7.1.2 Elemente mit konstanter Größenvariation

Nachdem die Performance der Environments bei Elementen identischer Größe betrachtet wurde, wird bei den nächsten Tests mit einer Größenvariation gearbeitet. Der erste Vergleich hierzu verwendet Elemente mit einer Größenvarianz von Faktor fünf. Konkret werden Kreise

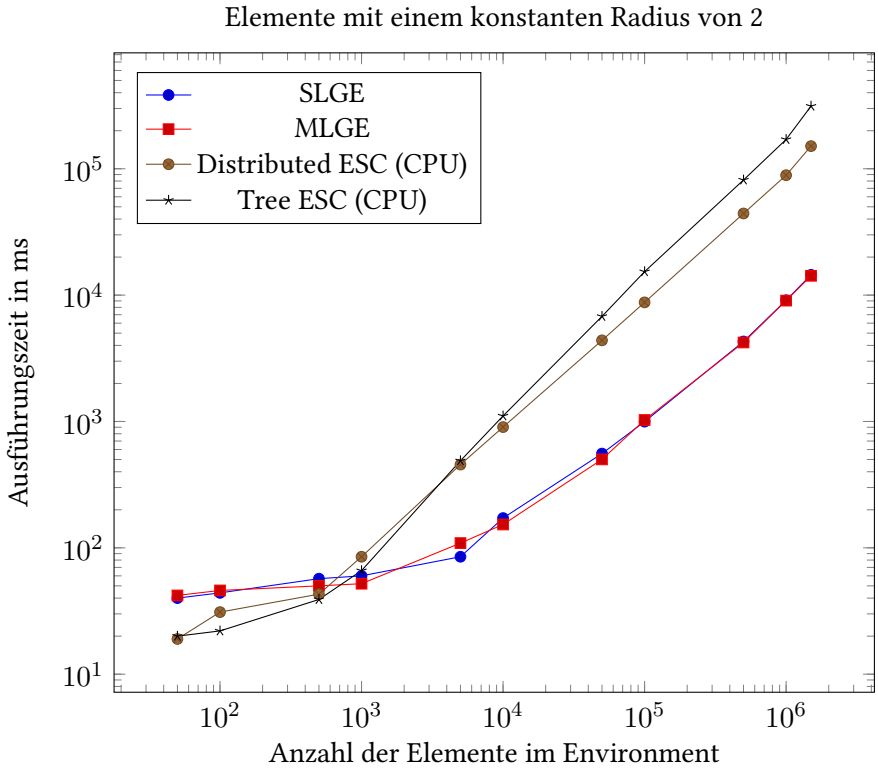

Abbildung 7.1: Vergleich der Kollisionserkennung der verschiedenen Environments bei Kreisobjekten mit einem Radius von 2

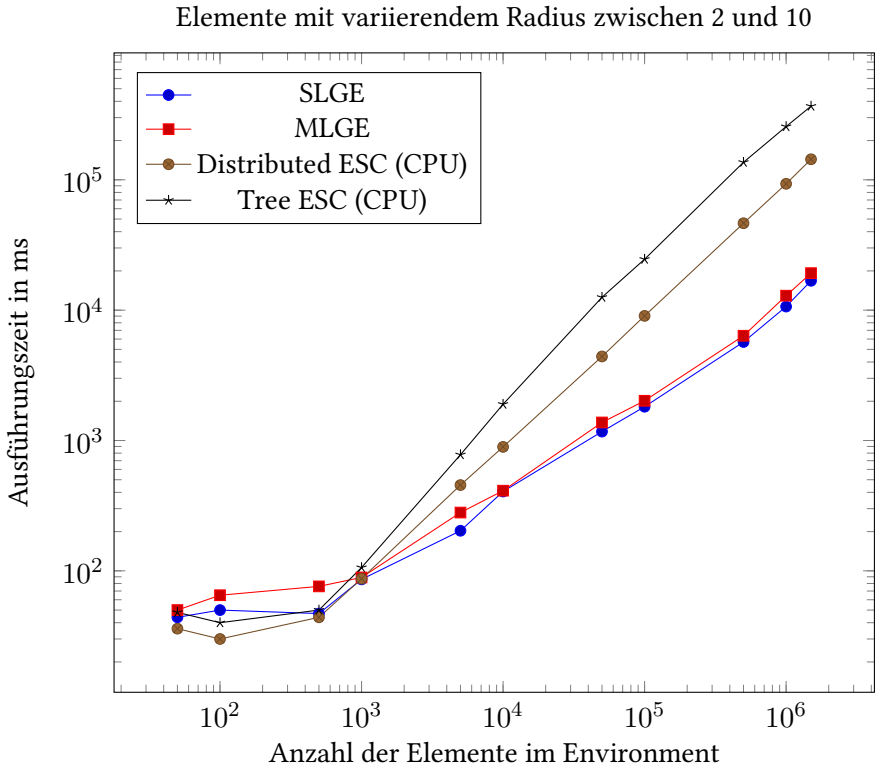

Abbildung 7.2: Vergleich der Kollisionserkennung der verschiedenen Environments bei Kreisobjekten mit einem variierendem Radius zwischen 2 und 10

mit zufälligen Radien zwischen zwei und zehn erzeugt. Wie beim vorherigen Test wird mit 50 Elementen begonnen und bis 1.500.000 Elemente hoch-skaliert. Diese werden dem Environment an zufälligen Positionen hinzugefügt, wodurch eine Kollisionsprüfung durchgeführt wird.

Abbildung 7.2 zeigt die Ergebnisse dieses Testdurchlaufes. Die beiden CPU Varianten verhalten sich ab 500 Elementen wie erwartet, sehr linear zur Elementanzahl. Dort kann gegenüber der Test mit identischer Form kaum ein Unterschied festgestellt werden. Bei den GPU Environments unterscheidet sich die Ausführungszeit ebenfalls nur gering. Sie sind im Schnitt ca. 10 bis 20 Prozent langsamer. Auffällig ist jedoch, dass das MLGE hier sogar etwas langsamer als das SLGE ist.

In dieser Reihe wurde noch ein zweiter Test vorgenommen. Der Aufbau ist bis auf die Größenvariation identisch zum vorherigen Testlauf. Diese umfasst hier Kreise mit Radien zwischen 2 und 100, folge dem einen Faktor 50.

Auf Abbildung 7.3 sind die Ergebnisse hierzu dargestellt. Bei den CPU Environments ist weiterhin kein Unterschied zu sehen.

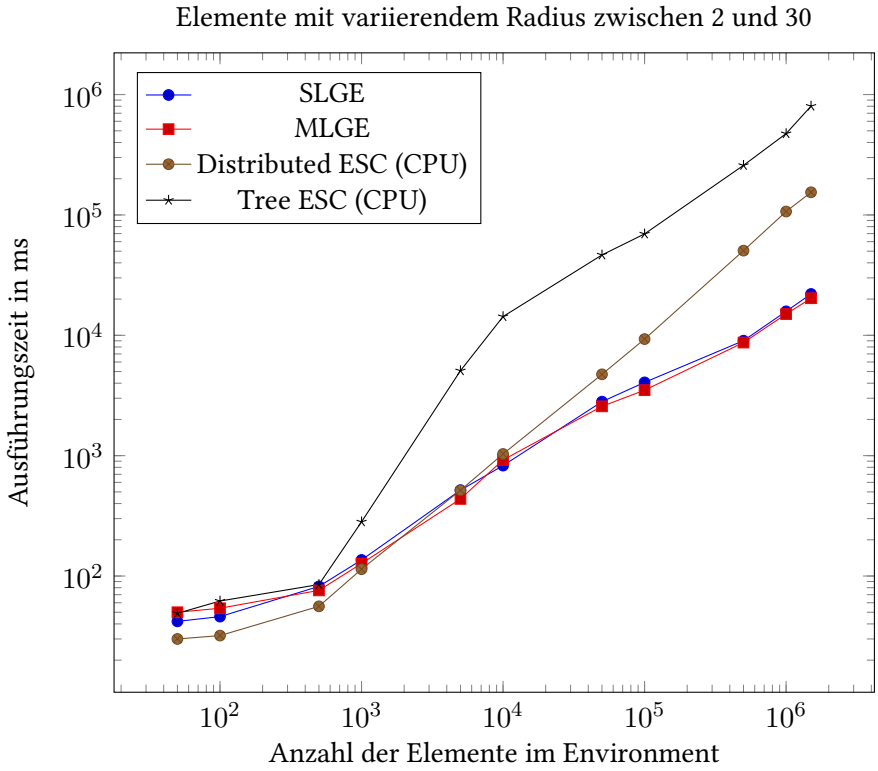

Abbildung 7.3: Vergleich der Kollisionserkennung der verschiedenen Environments bei Kreisobjekten mit einem variierendem Radius zwischen 2 und 30

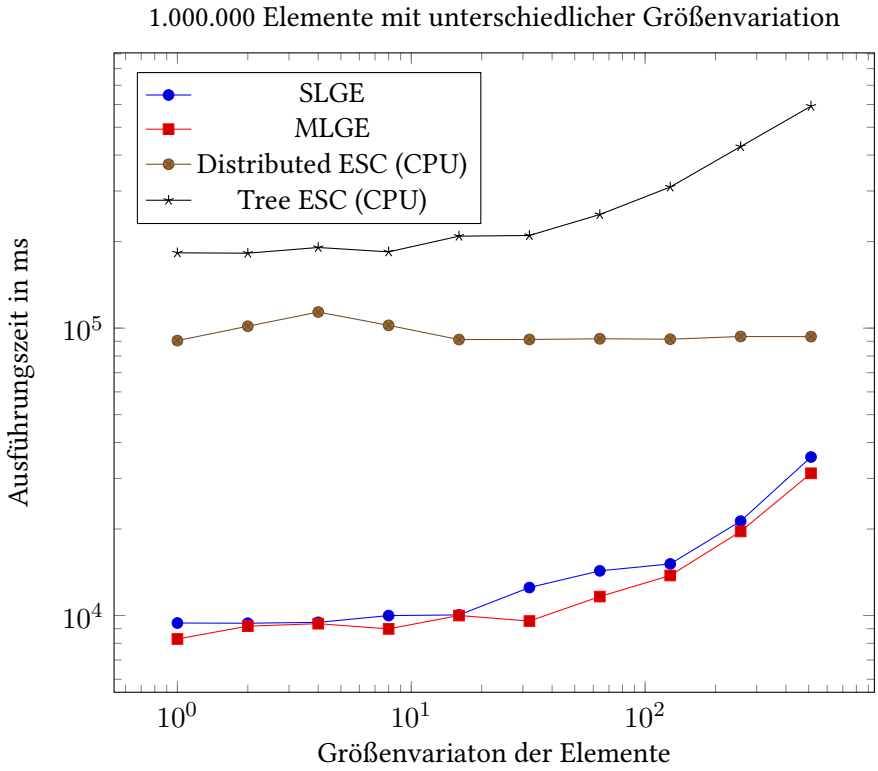

Abbildung 7.4: Vergleich der Kollisionserkennung der verschiedenen Environments bei einer konstanten Anzahl an Elementen und einer steigenden Größenvarianz

Eine genauere Betrachtung der GPU Environments zeigt, dass diese Performanz einbüßen gegenüber den Test mit geringeren Größenunterschieden. Dies führt dazu, dass der Break-Even-Point sich auf 5000 Elemente verlagert. Des Weiteren ist hier zu sehen, MLGE sich leicht vom SLGE absetzen kann.

7.1.3 Elemente mit variierender Größenvariation

Um den Effekt, den die Varianz der Formgrößen auf die GPU Environments hat, genauer analysieren zu können wurde ein weiterer Versuch vorgenommen. Dabei wird mit einer konstanten Agentenanzahl von einer Million Elementen gearbeitet. Die Größenvariation der Agenten wird erhöht, beginnend bei einem konstanten Radius von zwei bis zu einen Radius zwischen 2 und 512.

Auf Abbildung 7.4 ist sehr gut zu erkennen, dass das auf die CPU Environments kaum Einfluss hat. Bei der *Distributed ESC* kann entsteht nahezu keine Erhöhung der Ausführungszeit selbst bei einer sehr starken Varianz. Ebenso ist zu erkennen, dass die GPU Environments zwar mit

steigender Varianz langsamer werden, jedoch selbst größten getesteten Wert noch um einen guten Faktor vier schneller sind als die schnellste CPU Variante. Der direkte Vergleich zwischen SLGE und MLGE zeigt, dass das MLGE bei starker Varianz ca. 10 Prozent schneller ist als das SLGE.

Nachdem in diesem Abschnitt ein direkter Vergleich der Kollisionserkennung erfolgt ist, wird im nächsten Abschnitt die Performance der Environments im Kontext des Antelopes und Lions Modell getestet.

7.2 Antelopes and Lions

Um den Nutzen des GPU-Environments in einem konkreten Szenario zu zeigen, wurden weitere Tests mit dem Antelopes and Lions Modell vorgenommen. Für diese Tests wurde ein Testsystem mit einer Intel i5 3540 CPU, einer Nvidia GTX 760 GPU und 16GB Arbeitsspeicher verwendet. Zu dem Zweck ist eine lokale Instanz des MARS Life Frameworks gestartet worden, auf der die Tests ausgeführt wurden. Ebenfalls zu beachtet ist, dass alle für MARS Life notwendigen Komponenten inklusive Ergebnisdatenbank auf dem lokalen System werden. Dadurch wird ausgeschlossen, das andere Nutzer des Systems die Ergebnisse beeinträchtigen. Um besser mit den Environmentstests vergleichen zu können werden ebenfalls doppelt logarithmische Graphen verwendet.

Im ersten Test wurde das Antelopes and Lions Modell mit unterschiedlichen Environments ausgeführt. Dabei wurde die Zeit gemessen, die das System für einen Simulationstick benötigt. Für einen besseren Vergleich der Environments wurde ebenfalls aufgenommen, wie viel davon das Environment beansprucht hat. Die Aufteilung der Agenten ist jeweils in einer 12:2:1 (Grass : Antilopen : Löwen) erfolgt.

Auf Abbildung 7.5 sind die Ergebnisse des ersten Testdurchlaufs zu sehen. Die Ausführungszeiten stellen jeweils einen Mittelwert über 50 Simulationsticks dar. Es ist zu erkennen, dass die GPU Environments ab dem Testdurchlauf mit 15000 Agenten den CPU Environments überlegen sind, jedoch der Abstand deutlich geringer ist, als bei den direkten Vergleichstest aus Abschnitt 7.1; Bei der letzten Messung mit 300000 Agenten reduziert die Dauer eines Simulationsticks um 23 Prozent im Vergleich zur schnellsten CPU Lösung. Zwischen SLGE und MLGE zeigt der Graph kaum einen Unterschied.

Die Abbildung 7.6 zeigt einen direkten Vergleich der benötigten Rechenzeit für das Commit der Environments. Im Commit erfolgt sowohl die Exploration als auch die Bewegung der Agenten, weswegen eine Beobachtung der Rechenzeit dieser Funktion sich gut für den Vergleich der Environments eignet. Dort zeigt sich ein ähnliches Bild wie bei den direkten Vergleichstests

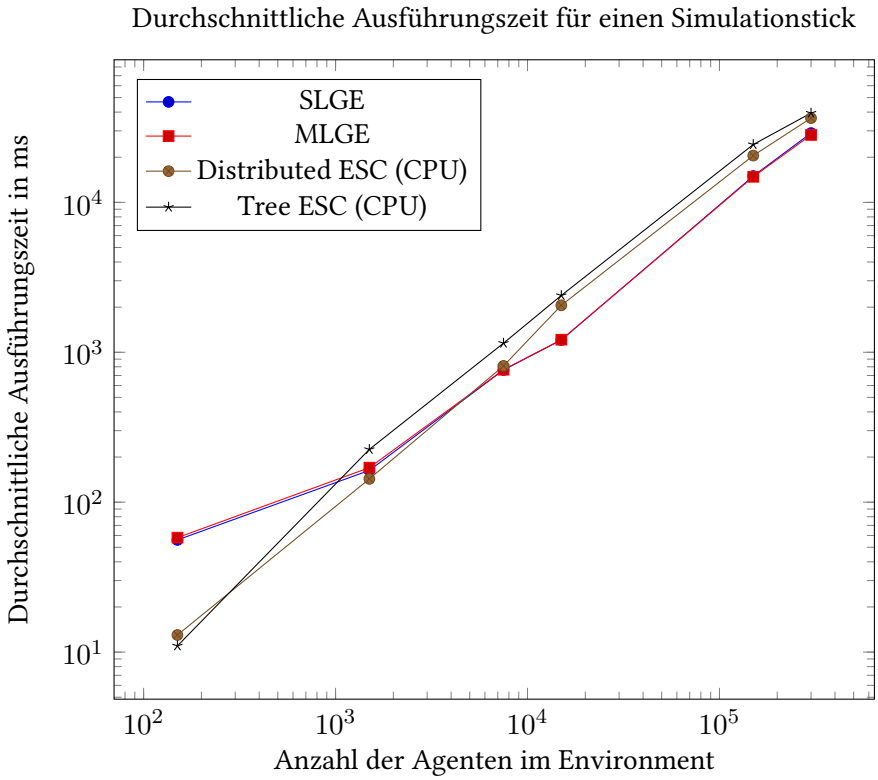

Abbildung 7.5: Vergleich der verschiedenen Environments beim Antelopes und Lions Modell. Gemessen wurde jeweils die Zeit, die MARS Life für die Ausführung eines Ticks benötigt hat

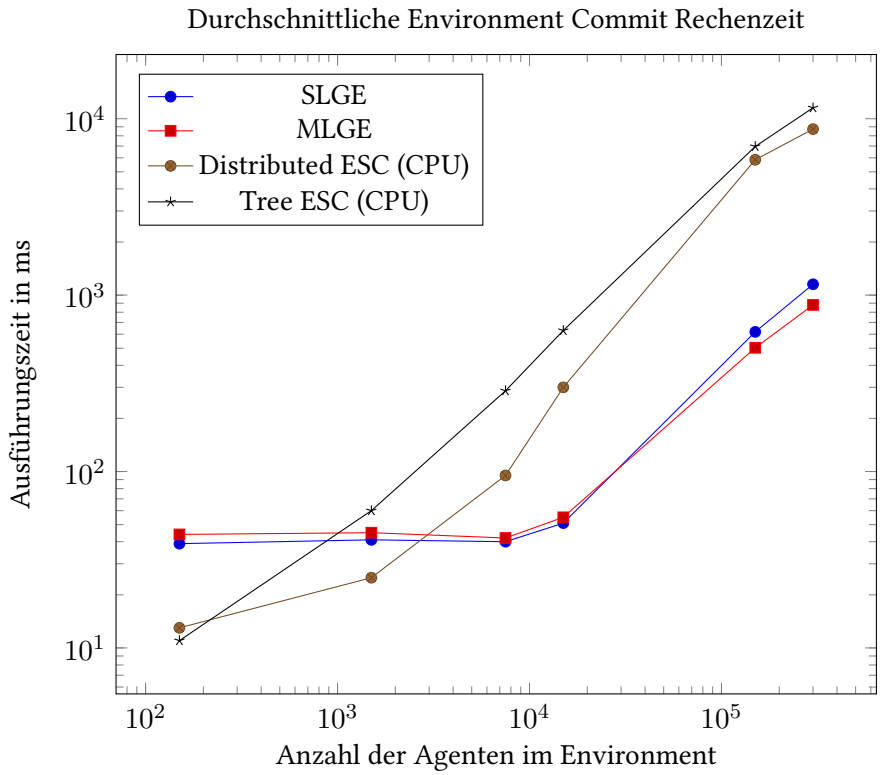

Abbildung 7.6: Direkter Vergleich der Environments beim Antelopes und Lionsmodell. In diesem Graph wird die Zeit verglichen, die durchschnittlich zur Abhandlung des Environment Commits benötigt wurde

Vergleich zwischen dem Antelopes and Lions und dem Wolves and Sheep Modell

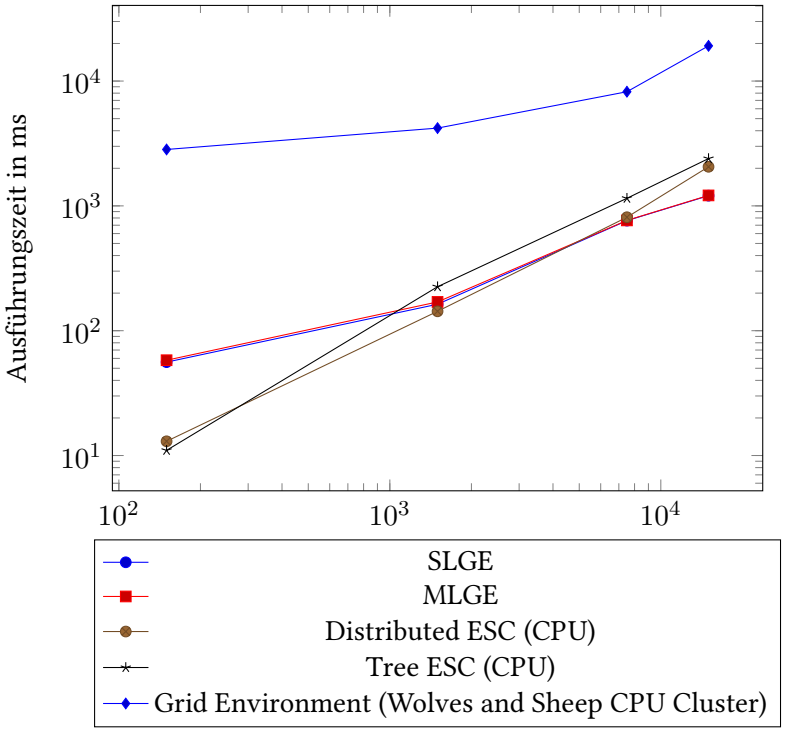

Abbildung 7.7: Vergleich der Ausführungszeit eines Simulationsticks zwischen dem Antelopes and Lions Modell mit den asynchronen Environment und dem Wolves and Sheep Modell mit dem synchronen Gridenvironment

aus Abschnitt 7.1. Bis 10000 Agenten hält sich die Ausführungszeit der GPU Environments nahezu konstant und von dort an steigt sie annähernd linear zur Anzahl der Agenten. Der Abstand zwischen CPU und GPU liegt, ähnlich wie in den Vergleichstests mit stärkerer Varianz, bei einem Faktor von 9,92. Zu beachten ist ebenfalls, dass das Environment bei 300000 Agenten nur 3,12 Prozent der Ausführungszeit des Simulationsticks einnimmt.

Um ein Verhältnis zu der Performanz von MARS Life vor dieser Masterarbeit zu haben wurde das Wolves and Sheep Modell, dessen Grundlogik der des Antelopes and Lions Modells ähnelt, auf dem Kubernetes Cluster (der aktuellen Ausführungsumgebung) ausgeführt und ebenfalls die durchschnittliche Dauer eines Simulationsticks bestimmt. Das Wolves and Sheep Modell arbeitet mit dem Gridenvironment und den synchronen Basic Agents. Aufgrund der eingeschränkten Ressourcen konnten keine Simulationen mit mehr als 15000 Agenten ausgeführt werden. Abbildung 7.7 zeigt den Graphen, der aus diesem Testablauf entstanden ist.

Zu erkennen ist das die aktuelle Umgebung mit dem Gridenvironment deutlich langsamer ist, als das Antelopes and Lions Modell. Aufgrund der geringen Agentenzahl liegen die GPU Environments noch gleich auf mit der *Distributed ESC*.

Eine kritische Betrachtung der Ergebnisse folgt im nächsten Kapitel.

8 Diskussion

In diesem Kapitel werden die Ergebnisse der Experimente analysiert und anschließend den Hypothesen gegenüber gestellt.

8.1 Analyse der Experimente

Zuerst wird auf die direkten Vergleichstests eingegangen. Das Verhalten der GPU- im Vergleich zu den CPU Environments entspricht grundsätzlich den Erwartungshaltungen. In den hohen Agentenzahlen setzen sich die GPU Environments, durch ihre parallele Architektur, deutlich ab und können die Aufgaben in einem Zehntel der Zeit lösen. Ebenfalls zu erwarten war, dass der Abstand zu den CPU Environments, bei starker Größenvarianz der Elemente sinkt. Folgt daraus, dass die Vorauswahl der Kollisionskandidaten auf Basis des unterliegenden Gridfeldes vorgenommen wird, dessen Effizienz bei steigender Varianz der Elementgröße deutlich sinkt. Die CPU Environments sind der Varianz gegenüber nahezu unempfindlich, was sich besonders deutlich bei der *Distributed ESC* zeigt, wie auf Abbildung 7.4 zu erkennen ist. Das ist darauf zurückzuführen, dass beide mit einer dynamischen Baumstruktur arbeiten, welche gut mit Varianz arbeiten kann. Zu beachten ist jedoch, dass die GPU Environments trotz der starken Varianz weiterhin um einen Faktor vier schneller sind.

Deutlich weniger erwartet, war jedoch der geringe Unterschied in den Ausführungszeiten von SLGE und MLGE bei erhöhter Varianz der Elementgröße. Durch das hierarchische Gridfeld ist das MLGE darauf konzipiert, besser mit erhöhter Varianz umgehen zu können als das SLGE. Dies zeigt sich jedoch mit ca. zehn Prozent verringerter Rechendauer nur geringfügig. Bei einer sehr geringen Varianz das war SLGE sogar minimal schneller. Das kann darauf zurückgeführt werden, dass in dem Fall der Mehraufwand zum Erstellen der Ebenen, den Vorteil durch die reduzierte Anzahl an Detailprüfungen übertrifft. Eine genauere Analyse der einzelnen Schritte der Kollisionserkennung hat ergeben, dass der limitierende Faktor auf der CPU Seite liegt. Es wird die Anzahl der Detailprüfungen durch das MLGE um 50 Prozent reduziert, jedoch stellen diese nur einen Teil der Ausführungszeit dar. Den größten Anteil, hat das Aufrufen der Delegaten, allerdings besteht dort wenig Potenzial zu Optimierung, da dies nur auf der

CPU geschehen kann und bereits parallel ausgeführt wird. Dies lässt sich darauf zurückführen, dass in den Tests nur mit Kreisen gearbeitet wird, bei denen eine Kollisionsprüfung nicht sehr komplex ist. Das führt dazu, dass die Operationen nach der Kollisionsprüfung das Ergebnis verzerren und der Unterschied zwischen MLGE und SLGE verschwimmt. Weiterhin muss hier beachtet werden, dass die Tests mit ca. 20 Prozent bei einer Million Agenten ein recht hohes Kollisionsaufkommen haben, wodurch dies noch verstärkt wird.

Die Tests mit dem Antelopes and Lions Modell haben gezeigt, dass die GPU Environments auch in einer Modellumgebung gute Performanz liefern. Es konnte die Ausführungszeit um mehr als 22 Prozent gesenkt werden. Wobei das Environment um mehr als das neun Fache beschleunigt werden konnte. In dem Zusammenhang hat sich auch gezeigt, dass das Environment nicht das einzige limitierende Element im MARS Life Framework ist. Das Environment nimmt bei mehr als 20000 Agenten gut ein Viertel der Gesamtausführungszeit, was durch den Einsatz der GPU auf unter vier Prozent gesenkt werden konnte.

Der Vergleich zwischen dem synchronen Räuber-Beute Modell Wolves and Sheep mit dem asynchronen Antelopes and Lions Modell zeigt, dass durch eine asynchrone Berechnung des Environments die Ausführungszeit deutlich verringert werden kann. Dieser Effekt konnte sowohl bei den CPU- als auch bei den GPU Environments festgestellt werden. Dabei ist zu Beachten, dass die Modelle nicht komplett identisch sind, weswegen die Ergebnisse nur eingeschränkt vergleichbar sind. Aufgrund der großen Unterschiede in den Ausführungszeiten ist es trotz dessen sehr wahrscheinlich, dass das asynchrone Environment auch bei identischer Logik schneller ist.

Betrachtet man die Experimente insgesamt, hat sich herausgestellt, dass die Environments mit der Kollisionserkennung auf GPU-Basis, dann eingesetzt werden sollten, sobald mit mehr als 10000 Elementen gearbeitet wird. Bei starker Varianz der Agentengröße werden die GPU-Environments bedingt durch die Architektur langsamer, aber der Varianztest hat gezeigt, dass sie auch dann noch schneller als die CPU Environments sind. Auch bei Elementen, deren Radius um Faktor 256 variiert waren die GPU-Environments um einen Faktor 3 schneller bei 100000 Elementen.

8.2 Gegenüberstellung von Hypothesen und Ergebnissen

Dieser Abschnitt enthält die Gegenüberstellung der Hypothesen aus der Analyse und den Ergebnissen der Experimente.

Ein Environment, welches die Kollisionserkennung auf der GPU berechnet, ist performanter als eins das alle Berechnungen auf der CPU durchführt
Durch die Experimente konnte diese bestätigt werden. Mit steigendem Agentenaufkommen haben die GPU Environments eine deutlich bessere Performance gezeigt. Selbst bei einer starken Varianz der Agentengröße sind sie, ab einer Agentenanzahl im zweistelligen Tausender-Bereich, in allen Tests schneller als die CPU Environments, auch wenn der Abstand mit steigender Varianz sinkt. Ein sehr ähnliches Bild hat sich auch bei der Anwendung mit dem Antelopes und Lions Modell gezeigt. Im direkten Vergleich konnten sich die GPU Environments deutlich von den CPU Environments absetzen. Bei hohen Agentenzahlen haben die Environments-Berechnungen nur noch 3,4 Prozent der Gesamtdauer eingenommen.

Die GPU optimierten Funktionalitäten können so angeboten werden, dass die Modellentwickler kein Spezialwissen benötigen
Diese Hypothese konnte ebenfalls bestätigt werden. Allerdings musste die Schnittstelle zum Environment angepasst werden. Bisher wurde das Environment ausschließlich in einen synchronen Kontext verwendet. Die synchrone Abarbeitung des Environment fordert implizit eine Serialisierung der Anfragen. Aufgrund der parallelen Architektur der GPU, kann sie ihr Potenzial nur bei parallelen Arbeiten ausschöpfen. Deswegen wurde eine asynchrone Schnittstelle geschaffen, die jedoch alle Funktionen der synchronen Schnittstelle bietet. Durch die Async Agents Abstraktion wird dem Modellentwickler eine Schnittstelle geboten, mit der er ,bis auf wenige Ausnahmen, so Arbeiten kann, wie mit dem synchronen Environment.

Da auch die bisherige Environments über die asynchrone Schnittstelle verwendbar sind, besteht die Möglichkeit, fortan sämtliche Modellentwicklung asynchron durchzuführen und damit auf das schnellere Environment zuzugreifen.

9 Ausblick

In diesem Kapitel wird ein Ausblick auf mögliche Erweiterungen im Kontext des Environments mit GPU-Basierter Kollisionserkennung gegeben. Dabei wird im ersten Abschnitt auf die Permutationsbildung eingegangen. Anschließend wird die dynamische Typfilterung betrachtet, welche bereits im MLGE vorgesehen ist, jedoch noch umgesetzt werden muss. Ein weiterer Punkt, indem Erweiterungen möglich sind, ist die Speicherverwaltung. MARS Life ist darauf ausgelegt große Szenarien zu simulieren und der Speicherplatz auf der GPU ist stark begrenzt. Im Abschnitt 9.2 werden einige Ansätze hierzu beleuchtet. Zuletzt wird die Verteilung der Last auf mehrere GPUs betrachtet. In dem Rahmen wird auch auf die Kubernetes Integration eingegangen.

9.1 Permutationsbildung

Beide Varianten des GPU-Environments nutzen Spatiale Subdivision zur Kollisionserkennung. Durch das Gridfeld in der Spatialen Subdivision lassen sich bereits viele Prüfungen im Voraus ausschließen. Sobald sich jedoch mehrere Elemente in demselben Gridfeld befinden, müssen dieses auf Kollision geprüft werden. Es kann eine Kollision zwischen Elementen in selbem Gridfeld nicht ausgeschlossen werden. Deswegen müssen in dem Fall alle Permutationen für die Menge der Elemente im selben Gridfeld gebildet werden, um jeweils eine Detailprüfung durchzuführen. Was zu einer langen Ausführungszeit führen kann. In der Regel befinden sich nur wenige Elemente im selben Feld, sodass die Permuationsbildung nicht viel Rechenleistung benötigt. Aus dem Grund wird in den GPU-Environments so verfahren, dass je ein GPU-Workitem die Permutationen für ein Gridfeld berechnet.

Um den Rechenaufwand genauer bestimmen zu können wird nachfolgend erläutert, woraus sich die Anzahl der benötigten Einzelprüfungen ermitteln lässt. Die Basis dazu stellt die Gaußsche Summenformel dar: $\frac{n*(n+1)}{2}$. Wobei n in diesem Fall Anzahl der Agenten 1 ist, da immer Tupel gebildet werden. Hierzu ein Beispiel:
Angenommen fünf Agenten *a, b, c, d* und *e* befinden sich auf demselben Gridfeld.
a muss gegen *b, c, d* und *e* auf Kollision prüfen.

b muss noch gegen *c*, *d* und *e* prüfen, da die Prüfung *a*-*b* bereits von *a* vorgenommen wurde.
c prüft gegen *d* und *e*.
Zuletzt muss *d* noch gegen *e* prüfen.
Insgesamt müssen $4(n-1)+3+2+1=10$ Prüfungen vorgenommen werden.
Eingesetzt in die Gaußsche Formel erhält man das gleiche Ergebnis:

$$\frac{4*(4+1)}{2}=10$$

Es steigt die Anzahl der Permutationen quadratisch.

Damit es in dem Fall nicht zu langen Rechenzeiten kommt, kann die Berechnung der Permutationen aufgeteilt werden. Beispielsweise könnte, abhängig von der Anzahl Agenten im selben Feld, ein eigener Kernel für die Permutationsbildung in dem Feld gestartet werden, welcher die Permutationen parallel berechnet. Die Anzahl der Agenten im Feld wird bereits bestimmt, um den benötigten Speicher reservieren zu können. An den Punkt kann eine Schwelle gesetzt werden, ab der das Gridefld markiert wird um im separaten Kernel berechnet zu werden. Aufgrund des Ansatzes mit den Phantomcells und beim MLGE mit den importierten Elementen, muss nicht für jede Kombination ein Tupel erstellt werden. Zu dem Zweck könnte eine angepasste Lösung der parallelen Permutationsbildung entworfen werden.

9.2 Speichermanagement

Das Problem des begrenzten Speichers der GPU wurde bereits in einigen Abschnitten der Arbeit erwähnt. Die GPU bietet nur eine sehr begrenzte Menge an Speicher und durch die Simulation großer Szenarien kann es passieren, dass dieser nicht ausreichend ist um das gesamte Environment abzubilden. Es gibt zwei Punkte bei der Kollisionserkennung, die in dem Kontext besonders kritisch betrachtet werden müssen. Das ist zum einen die Sortierung der Cellarrays und zum Anderen das Bilden der Kollisionstupel.

Bei den Cellarrays ist das Problem, dass alle Elemente sich zum Sortieren gleichzeitig auf der GPU befinden müssen. Sobald die Anzahl an Agenten so groß ist, dass der Speicherplatz nicht ausreicht, dass die Sortierung durchgeführt werden kann, ist es nicht mehr möglich dieses Schema zu verwenden. Um dies zu umgehen, müsste das Environment in unterschiedliche, komplett getrennte, Bereiche unterteilt werden. Darauf wird im Abschnitt 9.3 detaillierter eingegangen.

Ebenfalls problematisch ist das Bilden der Kollisionstupel. Wie im Abschnitt 9.1 erläutert wächst die Anzahl der benötigten Kollisionsprüfungen bei mehreren Agenten auf demselben Gridfeld quadratisch. Dadurch kann es bei großen Agentenanhäufungen und einer unpassenden Gridfeldgröße dazu kommen, dass mehr Kollisionstupel erstellt werden müssen als der GPU-Speicher verwalten kann. Um dies zu umgehen, müsste in dem Schritt die Bearbeitung in mehrere Durchläufe aufgeteilt werden, abhängig vom verfügbaren Speicher. Diese Durchläufe müssten neben dem Erstellen der Kollisionstupel auch die Kollisionsprüfung enthalten, da der Speicher erst freigegeben werden kann, nachdem die Kollisionsprüfung für die Tupel erfolgt ist.

9.3 Verteilung

Die Spatiale Subdivision bietet die Möglichkeit zur Verteilung an mehreren Positionen. Ohne zusätzlichen Aufwand könnten die Detailprüfungen der Elemente auf mehrere GPUs verteilt werden. Als Eingang für die Kollisionsprüfungen dient eine Liste an Tupeln, die geprüft werden müssen. Diese kann unterteilt und auf mehrere GPUs verteilt werden. Aufgrund der simplen Formen, die aktuell für die Agenten verwendet werden, benötigen die Detailprüfungen nur wenig Rechenzeit, weswegen sich die Verteilung hier erst bei komplexeren Formen bezahlt macht.

Für die Unterteilung des Environment kann der Ansatz von Du et al [11] verfolgt werden, der im Abschnitt 2.3 vorgestellt wurde. Zur Verwaltung würde sich dort eine Master/Slave Hierarchie eignen, in der der Master eine Unterteilung des Environments in mehrere Abschnitte übernimmt. Diese verteilt er an seine Slaves, welche anschließend, in ihrem Abschnitt, eine Kollisionsprüfung für alle Elemente vornehmen. Jeder Slave könnte dafür die Kollisionserkennungslogik des MLGE nutzen. Die Ergebnisse müssten abschließend vom Master zusammengetragen und ausgewertet werden. Um diesen Ansatz umzusetzen, muss zum einen ein Algorithmus zur Unterteilung des Environments entwickelt werden und zum anderen wird ein Schema zum Austausch der Daten zwischen Master und Slave benötigt.

10 Fazit

Zum Abschluss wird zuerst eine Kurzzusammenfassung der Masterarbeit gegeben, um zuletzt die Ergebnisse der Arbeit kritisch zu betrachten.

Das Ziel dieser Arbeit war es die Performanzprobleme des MARS Life Frameworks durch den Einsatz einer GPU-Basierten Kollisionserkennung zu beheben, ohne dabei die Modellentwickler zu beeinträchtigen.

Dafür wurde in den Grundlagen 2 ein Einblick das MARS Life Framework vorgestellt, auf dem diese Arbeit basiert. Speziell wurde auf das Environment eingegangen, welches zum einen die Aufgabe hat, die Bewegung der Agenten zu überwachen und zum anderen den Agenten das Explorieren ermöglicht. Dadurch, dass viele dieser Aufgaben eine Kollisionserkennung benötigen, ist die Entscheidung darauf gefallen, die GPU-Basierten Kollisionserkennung im Rahmen eines neuen Environments zu integrieren.

Dafür wurde im nächsten Abschnitt 2.2 in die Mechanik der GPU-Programmierung eingeführt. Im Anschluss daran wurden bestehende Ansätze zu Kollisionserkennung auf GPU Basis vorgestellt. Nach den Grundlagen wurden in der Analyse Anforderungen für das Environment sowie die Ziele der Arbeit in Form von Hypothesen festgelegt. Zum Abschluss des Kapitels wurde auf die für Integration des Environment relevanten Komponenten und Schnittstellen des MARS Life Frameworks eingegangen.

Das Konzept für die Umsetzung wurde im Anschluss vorgestellt. Dabei wurde zuerst im Abschnitt Integration 4.1 beschlossen, über welche Schnittstelle das GPU integriert wird. Zudem wird dort der Ansatz des asynchron arbeitenden Environments vorgestellt, welches benötigt wird, um die Parallelperformance der GPU nutzen zu können. Anschließend folgen die Konzepte für die beiden Environmentvarianten SLGEs und MLGEs. Das SLGE ist ein 3D Environment welches für die Kollisionserkennung Spatiale Subdivision auf einen statischen Gridfeld verwendet. Mit dem MLGE wurde das SLGE erweitert, indem auf ein hierarchisches Gridfeld gesetzt wird, um besser mit starker Größenvarianz zwischen den Agenten umgehen zu können. Zur Kompensation des erhöhten Speicherbedarfs wurde dabei auf den zwei Dimensionen Raum reduziert.

Nachdem die Konzepte vorgestellt worden folgt im Kapitel 5 die Realisierung, in dem neben den beiden GPU Environments auch der Radix-Sort vorgestellt wurde, welcher zur Umsetzung der Environments benötigt wurde. Im Kapitel 6 wurden die Komponenten erläutert, die für eine nahtlose MARS Life Integration notwendig sind. Dabei auf die Async Agents Abstraktion eingegangen und auf die benötigten Anpassungen, um die GPU im MARS Life Kontext verfügbar zu machen. Weiterhin wird dort das Antelopes and Lions Modell vorgestellt, welches für Vergleichstests zwischen den Environments genutzt werden kann.

Damit ist der Entwicklungsteil abgeschlossen und das entstandene Environment wird im Kapitel 7 in den Vergleich zum vorherigen Environments gestellt. Die Ergebnisse davon werden anschließend in der Diskussion kritisch betrachtet. Zuletzt wurden im Kapitel 9 Ansätze für weiterführende Projekte vorgestellt.

Unerwartete Komplikationen gab es beim Anpassen der Ausführungsumgebung, damit diese die Nutzung der GPU zu erlaubt. Dort musste unter Anderem eine OpenCL Bibliothek auf .NET Core portiert werden. Debugging von OpenCL Kernels hat sich auch als sehr problematisch herausgestellt, es musste viel Mehraufwand zu Evaluierung getrieben werden.

Insgesamt betrachtet, ist es gelungen die Performanzprobleme des MARS Life Systems durch den Einsatz der GPU-Basierten Kollisionserkennung deutlich zu verringern. Über die GPU Environments konnte die Kollisionserkennung nahtlos integriert werden. Dafür musste allerdings von einem synchronen auf ein asynchrones Environment gewechselt werden. Dessen Verwendung unterscheidet sich durch die entwickelte Abstraktionsschicht der Async Agents nur geringfügig vom vorherigen System. Damit ist auch für die Domainexperten keine Hürde entstanden und sie können das MARS Life Framework nahezu identisch verwenden.

Trotz der Komplikationen konnte durch die GPU-Basierte Kollisionserkennung eine Leistungssteigerung des Environments von bis zu Faktor zehn im direkten Vergleich erreicht werden. Auch im Testmodell Antelopes and Lions konnte eine verbesserte Performanz festgestellt werden. Bei der Verwendung einer GPU Kollisionserkennung konnte der Anteil des Environments an der Gesamtdauer eines Simulationsticks von über 25 Prozent auf weniger als vier Prozent gesenkt werden. Der Break-Even Point ab dem die GPU-Basierte Kollisionserkennung schneller ist, liegt abhängig von der Größenvarianz der Agenten zwischen 1000 und 10000 Elementen.

Im Vergleich der beiden GPU Environments, dem SLGE mit der standard-GPU-Basierten Kollisionserkennung und dem MLGE mit der auf MAS optimierten Kollisionserkennung konnte nur ein geringerer Performanzgewinn erzielt werden als erhofft. Die Anzahl der Detailprüfungen konnte zwar um mehr als 50 Prozent reduziert werden, jedoch hat sich nur eine Leistungssteigerung von ca. 10 Prozent abhängig von der Varianz eingestellt. Damit ist noch Potenzial vorhanden um den Agenten in zukünftigen Modellen komplexere Formen zu geben.

Abkürzungsverzeichnis

MAS Multi-Agent-Simulation . . . 1

SRA Sense Reason Act . . . 20

MSaaS Modeling and Simulation as a Service . . . 4

GPU Graphic Processing Unit . . . 1

CPU Central Processing Unit . . . 2

SLGE Single Layer 3D Grid-Environment . . . 24

MLGE Multi Layer 2D Grid-Environment . . . 26

BV Bounding Volume . . . 25

CU Compute Unit . . . 7

BVH Bounding Volume Hierarchie . . . 11

Literaturverzeichnis

[1] Mono a cross platform, open source .net framework. http://www.mono-project.com. Accessed: 2017-11-18.

[2] *Modeling & Simulation as a Service with the Massive Multi-Agent System MARS*. To appear in Proceedings of the 2016 Spring Simulation Multiconference, 2016.

[3] *The LIFE BasicAgents Package - A Generic SRA Agent Framework for the MARS Platform*, 2017.

[4] F. A. M. Alves, P. Jamieson, L. B. da Silva, R. S. Ferreira, and J. A. M. Nacif. Designing a collision detection accelerator on a heterogeneous cpu-fpga platform. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, Dec 2017.

[5] Ananth Balasubramaniam. Opencl.net. https://archive.codeplex.com/?p=openclnet nuget: https://www.nuget.org/packages/OpenCL.Net, 2013.

[6] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):10:70–10:93, January 2016.

[7] Erdal Cayirci. Modeling and simulation as a cloud service: A survey. In *Proceedings of the 2013 Winter Simulation Conference: Simulation: Making Decisions in a Complex World*, WSC '13, pages 389–400, Piscataway, NJ, USA, 2013. IEEE Press.

[8] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 666–675, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[9] Floyd M. Chitalu, Christophe Dubach, and Taku Komura. Bulk-synchronous parallel simultaneous bvh traversal for collision detection on gpus. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '18, pages 4:1–4:9, New York, NY, USA, 2018. ACM.

[10] Jan Dalski, Christian Huening, and Thomas Clemen. An output and 3d visualization concept for the msaas system mars. In *Proceedings of the 2017 Spring Simulation Multiconference*, ADS '17, Virginia Beach, Virginia, USA, 2017. Society for Computer Simulation International.

[11] Peng Du, Elvis S. Liu, and Toyotaro Suzumura. Parallel continuous collision detection for high-performance gpu cluster. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '17, pages 4:1–4:7, New York, NY, USA, 2017. ACM.

[12] M. Elteir, H. Lin, and W. C. Feng. Performance characterization and optimization of atomic operations on amd gpus. In *2011 IEEE International Conference on Cluster Computing*, pages 234–243, Sept 2011.

[13] Wenshan Fan, Bin Wang, Jean-Claude Paul, and Jia-Guang Sun. A hierarchical grid based framework for fast collision detection. *Comput. Graph. Forum*, 30:1451–1459, 2011.

[14] Daniel Glake, Julius Weyl, Carolin Dohmen, Christian Hüning, and Thomas Clemen. Modeling through model transformation with mars 2.0. In *Proceedings of the 2017 Spring Simulation Multiconference*, ADS '17, Virginia Beach, Virginia, USA, 2017. Society for Computer Simulation International.

[15] Volker Grimm and Steven F. Railsback. *Individual-based Modeling and Ecology:*. Princeton University Press, stu - student edition edition, 2005.

[16] GSIC. *TSUBAME 2.5 User's Guide*. Accessed: 2017-11-18.

[17] Stephen J. Guy, Ming C. Lin, and Dinesh Manocha. Modeling collision avoidance behavior for virtual humans. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 2 - Volume 2*, AAMAS '10, pages 575–582, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems.

[18] Emmanuel Hermellin and Fabien Michel. Gpu delegation: Toward a generic approach for developping mabs using gpu programming. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, AAMAS '16, pages 1249–1258, Richland, SC, 2016. International Foundation for Autonomous Agents and Multiagent Systems.

[19] Christian Huening. Analysis of performance and scalability of the cloud-based multi-agent system mars. Master's thesis, Hochschule für Angewandte Wissenschaften Hamburg, Berliner Tor 5, 20099 Hamburg, Germany, 6 2016.

[20] Christian Huening, Jason Wilmans, Nils Feyerabend, and Thomas Thiel-Clemen. Mars - a next-gen multi-agent simulation framework. In J Wittmann and D Maretis, editors, *Simulation in Umwelt- und Geowissenschaften, Workshop Osnabrück*. GI, Shaker, 2014.

[21] Klaus Kofler, Gregory Davis, and Sandra Gesing. SAMPO: an agent-based mosquito point model in OpenCL. In *ADS '14: Proceedings of the 2014 Symposium on Agent Directed Simulation*. Society for Computer Simulation International, April 2014.

[22] Björn König. OpenCL memory model . Available at https://de.wikipedia.org/wiki/Datei:OpenCL_Memory_model.svg, License : http://creativecommons.org/licenses/by-sa/3.0/de/legalcode, November 28, 2008.

[23] C Lauterbach, Q Mo, and D Manocha. gProximity: Hierarchical GPU based Operations for Collision and Distance Queries. *Computer Graphics Forum*, 2010.

[24] S. Le Grand. Broad-Phase collision detection with CUDA. *GPU Gems 3*, pages 697–721, 2008.

[25] François Lehericey, Valérie Gouranton, and Bruno Arnaldi. New iterative ray-traced collision detection algorithm for gpu architectures. In *Proceedings of the 19th ACM Symposium on Virtual Reality Software and Technology*, VRST '13, pages 215–218, New York, NY, USA, 2013. ACM.

[26] François Lehericey, Valérie Gouranton, and Bruno Arnaldi. Gpu ray-traced collision detection for cloth simulation. In *Proceedings of the 21st ACM Symposium on Virtual Reality Software and Technology*, VRST '15, pages 47–50, New York, NY, USA, 2015. ACM.

[27] Y. Liu and X. Zhang. Comparing two continuous collision detection algorithms on cpu and gpus. In *2016 International Conference on Virtual Reality and Visualization (ICVRV)*, pages 212–218, Sept 2016.

[28] Jiayuan Meng, Vitali A. Morozov, Kalyan Kumaran, Venkatram Vishwanath, and Thomas D. Uram. Grophecy: Gpu performance projection from cpu code skeletons. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 14:1–14:11, New York, NY, USA, 2011. ACM.

[29] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[30] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture.* O'Reilly Media, Inc., 1st edition, 2016.

[31] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[32] NVIDIA. CPU vs. GPU. Available at https://upload.wikimedia.org/wikipedia/commons/c/c6/Cpu-gpu.svg, accessed March 23, 2018.

[33] Simon Pabst, Artur Koch, and Wolfgang Straßer. Fast and scalable cpu/gpu collision detection for rigid and deformable surfaces. *Computer Graphics Forum*, 29(5):1605–1612, 2010.

[34] Bruno R. Preiss. *Data Structures and Algorithms with Object-oriented Design Patterns in C++.* John Wiley & Sons, Inc., New York, NY, USA, 1999.

[35] Paul Richmond. FLAME GPU technical report and user guide (CS-11-03). Technical report, Department of Computer Science, University of Sheffield, 2011.

[36] M. Scarpino. *OpenCL in Action: How to Accelerate Graphics and Computation.* Manning, 2012.

[37] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[38] Hybrid DSP Systems. Cudafy.net. https://archive.codeplex.com/?p=cudafy nuget: https://www.nuget.org/packages/CUDAfy.NET/, 2015.

[39] Min Tang, Dinesh Manocha, Jiang Lin, and Ruofeng Tong. Collision-streams. *Symposium on Interactive 3D Graphics and Games on - I3D '11*, 2011.

[40] Min Tang, Ruofeng Tong, Rahul Narain, Chang Meng, and Dinesh Manocha. A gpu-based streaming algorithm for high-resolution cloth simulation. *Computer Graphics Forum*, 32(7):21–30, 2013.

[41] Min Tang, Huamin Wang, Le Tang, Ruofeng Tong, and Dinesh Manocha. Cama: Contact-aware matrix assembly with unified collision handling for gpu-based cloth simulation. *Comput. Graph. Forum*, 35(2):511–521, May 2016.

[42] Jin Wang, Norman Rubin, Haicheng Wu, and Sudhakar Yalamanchili. Accelerating simulation of agent-based models on heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 108–119, New York, NY, USA, 2013. ACM.

[43] Tsz Ho Wong, Geoff Leach, and Fabio Zambetta. An adaptive octree grid for gpu-based collision detection of deformable objects. *Vis. Comput.*, 30(6-8):729–738, June 2014.

[44] Michael Woolridge and Michael J. Wooldridge. *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 12. Juni 2018 Philipp Kayser