FH Technikum Wien

MASTER THESIS

zur Erlangung des akademischen Grades
„Master of Science in Engineering“
im Studiengang Game Engineering und Simulation

Implementierung und Vergleich parallelisierter Pfadplanungsalgorithmen für die CPU und GPU

ausgeführt von Stefan Denner, BSc
A-2500 Baden, Hofackergasse 8

1. Begutachter: Dipl.-Ing. Dr. Gerd Hesina
2. Begutachter: Dipl.-Ing. (FH) Dr. Dietmar Schreiner

Baden, 22. August 2011

FACHHOCHSCHULE
TECHNIKUM WIEN

Eidesstattliche Erklärung

„Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.“

Ort, Datum	Unterschrift

Kurzfassung

In den letzten Jahren wurden viele Fortschritte im Bereich der künstlichen Intelligenzen (KI) erzielt. Dabei ist die Pfadplanung durch eine virtuelle Spielwelt ein wichtiger und oft verwendeter Aspekt der künstlichen Intelligenzen. Die größtenteils serielle Architektur der CPUs (Central Processing Unit) ist nicht für die gleichzeitige Simulation von einer großen Anzahl voneinander unabhängiger Agenten geeignet. Im Gegensatz hierzu kann die Technologie der Grafikprozessoren mit der parallel arbeitenden Architektur der GPUs (Graphics Processing Unit) sehr gut für diese Zwecke genutzt werden.

In der folgenden Arbeit wird die Parallelität von GPUs und auch Multi Core CPUs genutzt um parallelisierte Pfadplanungsalgorithmen zu entwickeln, darunter der „Greedy Best First Search“ (kurz: BFS), der Dijkstra und der A*-Algorithmus (gesprochen A-Stern). Zusätzlich zu einer objektorientierten Version dieser Algorithmen für die CPU in C++ werden mehrere Versionen in unterschiedlichen Technologien für die GPU implementiert. Die erste Technologie ist CUDA (Compute Unified Device Architecture) der Nvidia Corporation, die zweite ist OpenCL (Open Computing Language), ein offener Computing Standard, der von der Khronos Group mit Unterstützung mehrerer Firmen veröffentlicht wurde, darunter auch Größen wie Intel, Nvidia und AMD.

Bevor eine Umsetzung aber stattfindet und präsentiert wird, werden ähnliche Arbeiten zu diesem Thema vorgestellt und deren Ergebnisse betrachtet. Außerdem wird die allgemeine Architektur der GPU vorgestellt und mit der CPU Architektur verglichen um die vorhandenen Vor- und Nachteile der einzelnen Plattformen hervorzuheben. Zudem werden die Grundlagen des GPU-Computing behandelt, welche gleichermaßen in CUDA sowie in OpenCL gefunden werden können. Des Weiteren werden die schon zuvor erwähnten Pfadplanungsalgorithmen, „Greedy Best First Search“, Dijkstra und A* vorgestellt.

Das Ziel dieser Arbeit ist die Entwicklung eines Frameworks in C++, welches einer konfigurierbaren Anzahl an Agenten eine Pfadplanung durchführen lässt und diese auf dem Bildschirm darstellt. Die virtuelle Umgebung, durch welche die Agenten navigieren, wird ebenso konfigurierbar sein, um die Dynamik der Pfadplanung gewährleisten zu können. Die einzelnen Implementierungen werden verglichen indem Testläufe durchgeführt werden, welche auf einem Testsystem mit einer Nvidia Grafikkarte und einem Multi Core CPU von Intel ausgeführt werden. Es werden die Laufzeiten mit unterschiedlichsten Konfigurationen des ausgewählten Algorithmus, der Agentenanzahl, der Ausmaße der virtuellen Umwelt und der gewählten Technologie gemessen und die Ergebnisse im Laufe dieser Arbeit präsentiert.

Schlagwörter: GPU, GPGPU, CUDA, OpenCL, OpenMP, DirectX, Shader, Nvidia, KI, Künstliche Intelligenz, Pfadfindung, Pfadplanung, A*, Dijkstra, Greedy Best First Search

Abstract

In the last years many advances have been achieved in the field of artificial intelligence (AI). Thereby the path planning in a virtual environment is one of the most important and most used aspects of artificial intelligence. The largely serial architecture of the CPUs (Central Processing Unit) is not suitable for the simulation of a large amount of autonomous agents. On the contrary the technology of graphics processors with their parallel working architecture of the GPUs (Graphics Processing Unit) can be used very well for this purpose.

In the following work the parallelism of GPUs and also multi core CPUs will be used for implementing a parallelized path planning algorithm, including the “Greedy Best First Search” (short: BFS), the Dijkstra and the A*-algorithm (spelled a-star). In addition of an object oriented version of the algorithms for the CPU in C++ several versions in different technologies for the GPU will be implemented. The first technology is CUDA (Compute Unified Device Architecture) of the Nvidia Corporation and the second one is OpenCL (Open Computing Language), an open computing standard published by the Khronos Group with support of several companies, including popular ones like Intel, Nvidia and AMD.

Prior to the presentation of the implementations, related work to this topic will be presented and their results will be shown. Furthermore the general architecture of the GPU will be presented and compared with the CPU architecture to highlight the existing advantages and disadvantages of the single platforms. Also the principles of the GPU computing will be presented, which can be found similarly in CUDA as well as in OpenCL. In addition the path planning algorithms mentioned above, “Greedy Best First Search”, Dijkstra and A* will be presented.

The goal of this work will be a framework developed in C++ which will perform a path planning for a configurable count of agents that will be shown on screen. The virtual environment in which the agents have to navigate will also be configurable to ensure the dynamic style of the path planning. The single implementations will be compared by performing test runs on a testing system utilized with an Nvidia graphics card and a multi core CPU by Intel. The runtime performances of the different configurations of the used algorithm, the agent count, the size of the virtual environment and the chosen technology will be measured and the results will be presented later in this work.

Keywords: GPU, GPGPU, CUDA, OpenCL, OpenMP, DirectX, Shader, Nvidia, AI, artificial intelligence, path finding, path planning, A*, Dijkstra, Greedy Best First Search

Danksagung

Ich danke meiner Familie für ihre Geduld, für ihr Vertrauen und ihre bedingungslose Unterstützung in dieser schweren und langen Zeit des Schreibens dieser Arbeit.

Außerdem möchte ich all meinen Studienkollegen danken, da sie mir immer eine Quelle für Motivation und Inspiration baten.

Überdies möchte ich der Fachhochschule Technikum Wien für die Bereitstellung der Software für die Entwicklung des in dieser Arbeit implementierten Frameworks danken, darunter eine Kopie eines Windows 7 Professional Betriebssystems und eine Lizenz für Microsoft Visual Studio 2008 Professional.

Zudem möchte ich mich bei meinem Diplomarbeitsbetreuer Dipl.-Ing. Dr. Gerd Hesina für seine Unterstützung an dieser Arbeit bedanken. Seine Vorschläge und seine Kritik waren mir immer eine große Hilfe.

Zu guter Letzt möchte ich mich bei meinen Freunden bedanken, welche ich in den letzten Monaten aufgrund dieser Arbeit des Öfteren vernachlässigen musste.

Inhaltsverzeichnis

1 Einleitung

Der Kern eines jeden modernen Computersystems ist eine CPU (Central Processing Unit). Diese führt Operationen aus und wird von jeder Programmiersprache genutzt um die Befehle, die von Softwareentwickler programmiert werden, umzusetzen und Berechnungen durchzuführen. Dies ist der normale Weg der Programmierung, der schon seit Dekaden verfolgt wird und auch verständlich ist, denn die CPU ist eine große und komplexe Recheneinheit, die viele Befehle beherrscht und viele Datenformate unterstützt. Die CPU hat keine Spezialisierung und ist somit ein Alleskönner, die für die Aufgabe als zentrale Steuereinheit eines ganzen Systems sehr gut geeignet ist.

Im Gegensatz hierzu ist die GPU (Graphics Processing Unit) ein Chip, der sich auf der Grafikkarte eines Systems befindet. Die Grafikkarte ist eine hochspezialisierte Recheneinheit, die jede Art von Grafikberechnungen, welche in der Regel hoch parallel ablaufen, sehr schnell durchführen kann. Dies gelingt der GPU durch eine sehr hohe Anzahl an Rechenkernen. Eine moderne GPU hat mehrere hundert davon, welche wiederum mehrere hunderte Threads parallel ausführen können.

Die GPU entspricht der SIMD (Single Instruction – Multiple Data) Architektur, wo eine Instruktion auf mehrere Daten zugleich bzw. parallel ausgeführt werden kann. Eine GPU kann somit dieselbe Aufgabe parallel mit unterschiedlichen Daten durchführen. Die CPU hingegen entspricht der SISD (Single Instruction – Single Data) Architektur, die der Von-Neumann-Architektur entspricht, von der sich das Prinzip der Funktionsweise eines Computers bzw. einer CPU ableiten lässt. Hierbei wird eine Instruktion immer auf bestimmte Daten durchgeführt. Wenn dies auf mehreren Daten erfolgen soll, muss programmiertechnisch eine Schleife über diese Daten implementiert werden, welche die Operationen aber wieder nur hintereinander ausführt.

Natürlich gibt es heutzutage auch Multi-Core CPUs, welche mehrere Threads zugleich ausführen können. Aber im Vergleich zu einer GPU mit ihren hunderten SIMD Kernen scheint die CPU mit ihrer geringen Anzahl von vier oder vielleicht sogar acht Kernen eindeutig zu unterliegen. Dies ist aber wieder nur bedingt richtig.

Ein guter Vergleich einer CPU und einer GPU kann mit folgender Analogie gut dargestellt werden: Die CPU entspricht einem Menschen, der groß und komplex ist und als Individuum alleine sehr viel leisten kann. Eine GPU hingegen entspricht einem Ameisenstaat. Jeder einzelne der hunderten parallel laufenden Threads in den hunderten GPU Kernen entspricht einer Ameise. Ein Thread ist alleine, genauso wie eine Ameise, sehr einfach, klein und im Vergleich zu einem Menschen nicht sehr leistungsfähig. Aber wenn viele hunderte oder sogar tausende Ameisen in einem Staat zusammenkommen, wo sie alle dieselben wenigen Aktionen zugleich ausführen, können sie Leistungen vollbringen die sogar Menschen in Staunen versetzt.

Im Grunde wurde die GPU nur für Grafikaufgaben entwickelt und konnte lange Zeit auch nichts anderes. Mit der Entwicklung der Shader für programmierbare Aufgaben für den Grafikprozessor wurde noch weiter gedacht und die GPU auch für nicht grafikrelevante Aufgaben bereitgestellt. Mit Hilfe der zurzeit vorhandenen Hardware und der verfügbaren Software für das General Purpose Computing auf der GPU können schon alle möglichen Berechnungen implementiert werden, wobei sich durch die parallele Architektur vor allem parallele Algorithmen sehr gut eignen.

Es ist natürlich auch möglich, serielle Berechnungen auf der GPU auszuführen. Dies würde aber keinen Vorteil gegenüber der CPU schaffen. Das Ziel für die Verwendung der GPU statt der CPU ist ja eine Steigerung der Laufzeitperformanz aufgrund der Verlegung von Berechnungen von der CPU auf die GPU. Dies wird wiederum nur durch eine hoch parallele Verwendung der GPU erreicht.

1.1 Problemstellung

Ziel dieser Arbeit ist es Pfadplanungsalgorithmen für die GPU zu parallelisieren, sodass diese einer äquivalenten parallelisierten Version für die CPU im Punkto Laufzeitperformanz überliegen. Die Algorithmen, die für diese Implementierung verwendet werden, werden detailliert im Kapitel 4.2 „Pfadplanungsalgorithmen“ vorgestellt.

Als Programmiersprache wird C++ verwendet, um die CPU Version der Pfadplanungsalgorithmen zu entwickeln. Für die Parallelität der CPU Version wird OpenMP [1] eingesetzt, ein einfaches Framework um Code in mehrere Threads ausführen zu können, darauf ankommend, wie viele CPU-Kerne im System zur Verfügung stehen. Für die GPU Versionen der Algorithmen werden zwei verschiedene Technologien verwendet. Die erste Version wird mittels der CUDA (Compute Unified Device Architecture) Technologie von Nvidia [2] implementiert, die zweite unter Verwendung von OpenCL (Open Computing Language) der Khronos Group [3].

Das im Laufe dieser Arbeit entwickelnde Framework wird eine Pfadplanung für eine konfigurierbare Anzahl an Agenten durchführen. Welcher Algorithmus zur Pfadplanung verwendet werden soll, wird konfigurierbar sein sowie auch die virtuelle Umwelt, durch die die Agenten navigieren. Dies soll die Dynamik der Pfadplanung gewährleisten und deren Anpassungsfähigkeit auf unterschiedliche Situationen aufzeigen. In mehreren Laufzeittests werden auf einem Testsystem die einzelnen Versionen ausgeführt und die Laufzeiten programmiertechnisch protokolliert. Die Ergebnisse dieser Testläufe werden präsentiert und die Laufzeitunterschiede zwischen der CPU und der GPU Versionen sowie, falls vorhanden bzw. beobachtbar, zwischen den CUDA und OpenCL Versionen analysiert.

1.2 Literaturüberblick

Es existieren zurzeit einige Arbeiten die sich mit dem Thema der Einsetzung der GPU für KI spezifische Aufgaben beschäftigt. Dabei wurde des Öfteren das Hauptaugenmerk auf die Pfadplanung von Agenten durch eine virtuelle Umwelt gelegt. Die Autoren dieser Arbeiten verwendeten hierfür unterschiedliche Herangehensweisen und beschäftigten sich mit unterschiedlichen Aspekten der Pfadplanung bzw. der Agentensteuerung. Schriften über dieses Thema, welche in dieser Arbeit referenziert werden, sind die beiden Paper von Bleiweiss [4] und [5] sowie von Fischer et al. [6] und Shopf et al. [7]. Diese werden im Kapitel 2 „Analyse der Vorarbeiten“ genauer betrachtet und analysiert. Dabei wurden nur CPU und GPU Implementierungen von Algorithmen und Umsetzungen in unterschiedlichsten Technologien miteinander verglichen, aber nie die Technologien selbst.

Im Gegensatz hierzu gibt es nur wenige Arbeiten bezüglich des Vergleichs von Technologien für das GPU-Computing. Eine Arbeit, die sich mit dem Vergleich der CUDA (Compute Unified Device Architecture) Technologie von Nvidia [2] und der OpenCL (Open Computing Language) Technologie der Khronos Group [3] beschäftigt, ist ein Paper von Karimi et al. [8]. In diesem wird ein Zufallszahlengenerator basierend auf dem „Mersenne-Twister“ für die GPU implementiert und hierbei zwei annähernd identische Kernel in CUDA und OpenCL erstellt. Der OpenCL Kernel wurde so umgesetzt, dass dieser sowohl mit den Nvidia Tools als auch mit denen von ATI kompiliert. Vergleiche der Laufzeiten bestätigen einen Vorteil gegenüber CUDA, basierend auf der Tatsache dass CUDA zu einhundert Prozent für Nvidia GPUs entwickelt wurde und somit gegen OpenCL auf dieser Plattform nur Vorteile besitzt. Andererseits ist der OpenCL Kernel auch auf ATI GPUs lauffähig, welche genauso weit verbreitet sind wie die GPUs von Nvidia.

1.3 Struktur der Arbeit

Im Kapitel 2 „Analyse der Vorarbeiten“ werden, wie schon im vorherigen Abschnitt erwähnt, interessante Arbeiten anderer Autoren im Gebiet der GPU beschleunigten Pfadfindung vorgestellt. Dabei werden die unterschiedlichen verfolgten Ansätze dieser Arbeiten aufgezeigt und die dargelegten Prinzipien analysiert. Zum Schluss dieses Kapitels werden deren Ergebnisse erläutert und miteinander verglichen. Die daraus resultieren Rückschlüsse und Folgerungen der Autoren werden im weiteren Verlauf dieser Arbeit berücksichtigt und für die weiterführende Zielfindung in Betracht gezogen.

Im Kapitel 3 „Die Graphics Processing Unit“ wird die Hardwarearchitektur von Grafikprozessoren vorgestellt. Dabei wird auch ein Vergleich mit der Prozessorarchitektur von CPUs durchgeführt, woraus die Vor- und Nachteile bzw. die Stärken und Schwächen der einzelnen Komponenten ersichtlich werden. Zudem werden die grundlegenden Prinzipien des GPU-Computing, welche sowohl bei CUDA als auch bei OpenCL Verwendung finden, vorgestellt und erläutert.

Im Kapitel 4 „Pfadplanung und Pfadfindung“ werden die grundlegenden Prinzipien der Pfadplanung erläutert sowie dessen Einsatzgebiete betrachtet. Es wird definiert was ein Graph ist und warum dieser für die Pfadplanung essentiell ist. Es werden mehrere Algorithmen zur Findung eines Weges zwischen zwei Punkten mit Hilfe dieses Graphen vorgestellt, darunter der Algorithmus „Best First Search“, der Dijkstra Algorithmus und der A*-Algorithmus (gesprochen A-Stern). Es werden Details über die Algorithmen preisgegeben sowie deren Eigenschaften zum Schluss des Kapitels verglichen.

Im Kapitel 5 „Das Framework“ wird das im Laufe dieser Arbeit entwickelte Framework vorgestellt. Dabei werden die einzelnen Implementierungen der Pfadplanung sowie der verwendeten Datenstrukturen bzw. Listen vorgestellt. Die Implementierungen der Pfadplanung in den einzelnen Technologien, also die objektorientierte C++ Version und die beiden GPU Versionen entwickelt mit CUDA C und OpenCL C, werden vorgestellt. Zum Schluss werden die Eigenheiten dieser Implementierungen aufgezeigt sowie die programmiertechnischen Unterschiede und Gemeinsamkeiten erläutert.

Im Kapitel 6 „Laufzeittests der Implementierungen“ werden die im Framework entwickelten Pfadplanungsalgorithmen auf einem Testsystem ausgeführt, welches detailliert in diesem Kapitel vorgestellt wird. Es werden die Testergebnisse in Form von Diagrammen und Tabellen präsentiert. Dabei wird zuerst ein Vergleich zwischen den einzelnen Implementierungen der Pfadplanungsalgorithmen gegeben und versucht, die Aussagen zu den einzelnen Algorithmen aus Kapitel 4 „Pfadplanung und Pfadfindung“ zu bestätigen. Zweitens werden die GPU-Versionen, implementiert in CUDA und OpenCL, und die CPU-Implementierung in C++ miteinander verglichen und deren Laufzeiten präsentiert und in späterer Folge auch analysiert.

Im letzten Kapitel „Diskussion“ wird ein Fazit aus den Ergebnissen der Laufzeittests gegeben und Folgerungen daraus geschlossen. Hierfür werden nochmals die Vor- und Nachteile der einzelnen Plattformen und APIs herangezogen und eine finale Conclusio mit Hilfe der Laufzeitergebnisse gestellt. Außerdem wird versucht einen Vorausblick in die Zukunft dieser Technologien und Umsetzungsideen zu geben. Überdies wird auch erläutert, was in dieser Arbeit nicht umgesetzt, betrachtet und analysiert wurde und welche Themen und Aspekte für die Zukunft noch interessant wären.

2 Analyse der Vorarbeiten

2.1 Überblick

Die Pfadplanung sowie andere KI spezifische Methoden und Algorithmen zur Problemlösung bestimmter Aufgaben sind schon sehr gut erforscht. Die Ergebnisse dieser Forschung führten zu weltbekannten Algorithmen, welche schon während des Informatikstudiums gelehrt werden und teilweise seit Jahrzehnten bekannt sind. Hierbei handelt es sich vor allem um den Pfadplanungsalgorithmus A* (gesprochen A-Stern) und dem Dijkstra-Algorithmus.

Avi Bleiweiss hat als einer der ersten in seiner Arbeit [4] diese beiden Algorithmen für die GPU parallelisiert. Hierbei verwendete er die CUDA Technologie von Nvidia [2] und verglich seine Ergebnisse mit einer optimierten CPU Version, programmiert in C++. Dabei konnte er mit der hoch parallelen GPU Implementierung bei einer hohen Anzahl an Agenten einen Performanzgewinn gegenüber der CPU Implementierung erreichen.

In einer weiteren Arbeit hat Bleiweiss [5] nicht wie zuvor die Pfadplanung selbst für die GPU implementiert, sondern die Kollisionsvermeidung der Agenten mit anderen dynamischen Objekten behandelt. Hierfür hat er das von Van Den Berg et al. [9] vorgestellte Theorem der „Reciprocal Velocity Obstacles“ (RVO) erfolgreich für die GPU mittels CUDA umgesetzt. Dabei sind die Agenten in der Lage, einander auszuweichen und sogar stehen zu bleiben, falls ihr Weg von anderen Agenten bzw. dynamischen Objekten versperrt wird und ein Ausweichen und Umgehen nicht möglich ist.

Des Weiteren wurde in einer Arbeit von Fischer et al. [6] ein ganz anderer Ansatz verfolgt. Diese Arbeit baut auf dem “Boundary Value Problem“ (BVP) von Dapper et al. [10] auf. Dabei liegt hier der Schwerpunkt mehr auf der Parametrierung der Agenten, welche eine Vielzahl von unterschiedlichen Agentenverhalten erzeugen kann. Hierfür wurde ein BVP-Planer implementiert, der zur Pfadplanung eine Erweiterung der Laplace-Gleichung benutzt, welche die numerische Lösung des BVP repräsentiert. Diese Gleichung ist parametriert und kann eben durch diese unterschiedliche Lösungen eines Pfades für einen Agenten erzeugen.

Ein wiederum anderer Ansatz ist in der Arbeit von Shopf et al. [7] zu finden. Diese basiert auf der Methode der „Continuum Crowd“ welche von Treuille et al. [11] definiert wurde. Des Weiteren baut sie auf der Arbeit von Jeong und Whitaker [12] auf, welche eine parallele Methode für eine approximierte Lösung der Eikonal-Gleichung bietet, welche für die Kostenfunktion der Wegplanung der „Continuum Crowd“ dient. Zur Umsetzung wurde mittels der „High Level Shading Language“ (HLSL) ein „Compute Shader“ entwickelt, der den Algorithmus für die GPU implementiert.

In den nun folgenden Abschnitten werden die oben erwähnten Arbeiten genauer erläutert und die Prinzipien und Herangehensweisen vorgestellt. Zum Schluss werden die einzelnen Arbeiten verglichen und die Gemeinsamkeiten und Unterschiede analysiert und bewertet.

2.2 GPU beschleunigte Pfadfindung

Die ersten Umsetzungen von Algorithmen zur Pfadplanung wie der A* oder der Dijkstra-Algorithmus für die GPU sind in einer Arbeit von Avi Bleiweiss [4] zu finden. Er hat diese Algorithmen für die GPU mit Hilfe der CUDA Technologie von Nvidia [2] implementiert und sie zu Testzwecken auf einer „Nvidia GTX 280“ Grafikkarte ausgeführt. Dabei wurden mehrere Tests mit unterschiedlicher Anzahl an Agenten auf dieser GPU ausgeführt und mit der Laufzeit einer äquivalenten C++ Implementierung mit zwei Threads auf einer Dual Core CPU verglichen.

Beim A*-Algorithmus führte bei diesen Tests die GPU-Implementierung gegenüber der C++ Implementierung bei einer hohen Anzahl an Agenten zu einer ca. achtzehnfachen Performanzsteigerung [4]. Dieser erfolgreiche Einsatz der GPU für KI-spezifische Aufgaben ist vor allem aufgrund der hohen Agentenanzahl in den Testläufen zurückzuführen. Bei der geringeren Anzahl von 64 Agenten konnte Bleiweiss beobachten, dass die CPU Implementierung sogar schneller war als die für die GPU, was auf die höheren Taktraten der CPU gegenüber der GPU zurückzuführen ist. Somit ist der Vorteil nur bei einer großen Anzahl an Agenten gegeben.

2.3 Reciprocal Velocity Obstacles

In einer weiteren Arbeit von Bleiweiss [5] hat er die Erkenntnisse aus seiner vorigen Arbeit [4] weiter entwickelt. Sein Hauptaugenmerk liegt nun weniger auf die Pfadplanungsalgorithmen sondern mehr auf die autonome Reaktion der Agenten auf eine sich dynamisch verändernde Umwelt. Dies bedeutet, dass er die einmal stattfindende Pfadplanung, welche er in der ersten Arbeit mit der GPU beschleunigt hat, nun in den Hintergrund stellt. Stattdessen versuchte er den Agenten eine Intelligenz zur Kollisionsvermeidung mit anderen dynamischen Objekten zu geben. Dabei soll der Pfad, welcher zuvor berechnet wurde, eingehalten werden und der Agent diesen nur für Ausweichmanöver verlassen.

Diese Kollisionsvermeidung baut auf dem „Reciprocal Velocity Obstacle“ (RVO) Theorem von Van Den Berg et al. [9] auf, welche eine weiterentwickelte Form des „Velocity Obstacle“ Theorems von Fiorini und Shiller [13] ist. Mit den RVOs sind die Agenten in der Lage autonom Kollisionen mit anderen dynamischen Objekten zu verhindern und dabei plausible Ausweichmanöver durchzuführen. Dabei ist zu beachten, so wie bei LaValle [14] erwähnt, dass jeder sich bewegende Agent für die anderen Agenten ein dynamisches Hindernis darstellt. So weichen die Agenten einander aus und bleiben auch stehen, wenn für sie aufgrund von Anhäufungen anderer Agenten kein Weiterkommen besteht.

Dies ist vor allem in der aus dieser Arbeit resultierenden Demo eines Evakuierungsszenarios aus einem einstöckigen Bürogebäude sehr gut zu beobachten, bei der Agenten an den Ausgängen Anhäufungen erzeugen und das Weiterkommen der nachfolgenden Agenten verhindern [5]. Dieses Szenario ist in Abbildung 1 zu sehen, in dem 500 Agenten versuchen aus einem einstöckigen Bürokomplex mit nur zwei engen Ausgängen zu fliehen.

Abbildung 1: Screenshots aus der Evakuierungssimulation von Bleiweiss [5].

Für die Implementierung verwendete Bleiweiss wie zuvor CUDA [2] und verglich diese Implementierung mit einer äquivalenten Implementierung in C++. Das Ergebnis dieser Arbeit führte, so wie in der vorhergehenden Arbeit [4], wiederum zu einer Leistungssteigerung gegenüber der CPU Implementierung. Dabei ist aber wiederum die hohe Anzahl an Agenten ausschlaggebend für den hohen Performanzunterschied gegenüber der CPU Implementierung, welche dieses Mal auf 16 Threads aufgeteilt und auf einer 3.0 GHz Quad Core CPU ausgeführt wurde.

2.4 Boundary Value Problem

Eine weitere Autorengruppe bestehend aus Leonardo G. Fischer, Renato Silveira und Luciana Nedel haben sich mit diesem Thema beschäftigt und in ihrer Arbeit [6] einen ganz anderen Ansatz verfolgt. Dabei bauen sie auf die Arbeit von Dapper et al. [10] auf, in der das „Boundary Value Problem“ (BVP) behandelt wird und in deren Rahmen eine C++ Implementierung eines BVP-Planers durchgeführt wurde. Dieser BVP-Planer benutzt zur Pfadplanung eine Erweiterung der Laplace-Gleichung, welche die numerische Lösung des BVP repräsentiert und eine Familie von Potenzialfeldfunktionen produziert, die frei von lokalen Minima sind. Diese Gleichung zu sehen in (1) lautet wie folgt:

$$\nabla^2 p(r) + \epsilon v . \nabla p(r) = 0 \qquad (1)$$

Wobei v ein beliebig ausgerichteter Einheitsvektor und ϵ ein Skalarwert ist.

Diese serielle C++ Implementierung dieses BVP-Planers wurde in der Arbeit von Fischer et al. [6] für die GPU parallelisiert. Hier wurde, so wie bei Bleiweiss in seinen Arbeiten [4] und [5] CUDA verwendet. Dabei haben sie ihren Schwerpunkt weniger auf die Pfadplanung gelegt sondern mehr auf die Parametrierung der einzelnen Agenten. Dadurch können Verhaltensänderungen der einzelnen Agenten erzeugt werden, welche durch die einfache Änderung der Werte der Variablen v und ϵ in der Gleichung in (1) erreicht werden. Dies ist in Abbildung 2 zu erkennen, welche auf einer Abbildung von Fischer et al. [6] basiert, wo Pfade von Agenten mit folgenden Werten für ϵ und v gerechnet wurden:

(a) $\epsilon = 0$

(b) $\epsilon = -1.0$ und $v = (1,0)$

(c) $\epsilon = -1.0$ und $v = (1, \sin(0.6t))$

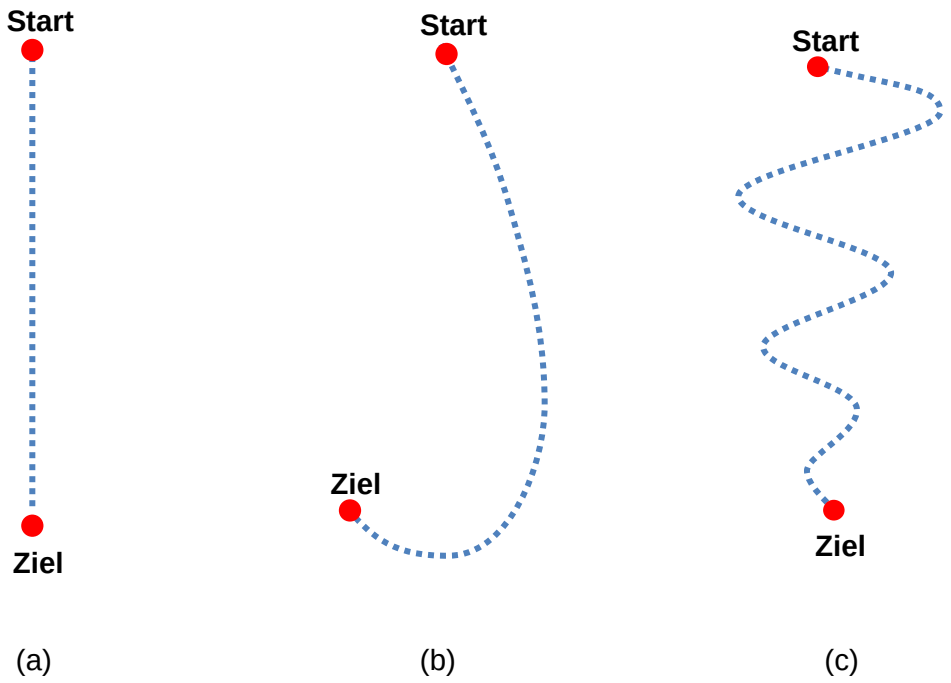


Abbildung 2: Verschiedenste Pfade eines Agenten mit Verwendung der Formel in (1).

Es wurde die C++ Implementierung des BVP-Planers aus der Arbeit von Dapper et al. [10] übernommen und mit einer selbst erstellten parallelisierten GPU Version verglichen. Diese GPU Version erzielte, vor allem da die CPU Version von Dapper et al. [10] komplett single-threaded ist, eine sechsundfünfzigfache Performanzsteigerung [6]. So konnten Szenarien mit einer großen Anzahl an Agenten in Echtzeit simuliert werden, was zuvor mit der CPU Implementierung nicht möglich gewesen wäre.

2.5 Continuum Crowds

Eine komplett andere Herangehensweise an die Verarbeitung von einer großen Anzahl an Agenten ist die von Treuille et al. [11] vorgestellte Methode der „Continuum Crowds“. Bei dieser entstehen sehr weiche und flüssige Bewegungen, die charakteristisch für eine große Menschenmenge sind. Eine Implementierung dieser Methode für die GPU wurde von Shopf et al. [7] durchgeführt. Hierbei werden bekannte numerische Methoden aus der Optik und der Physik verwendet, um Bewegungsplanung in ein Optimierungsproblem zu übersetzen. Es werden nicht die Agenten einzeln betrachtet und verarbeitet, sondern die Berechnung bezieht sich auf einzelne Teilräume mit Agenten. Dabei wird die virtuelle Umwelt als Kostenfunktion formuliert. Diese bezieht sich auf bestimmte Faktoren für beliebige Punkte in der Umwelt. Hierzu gehören die erreichbare Geschwindigkeit, basierend auf der Beschaffenheit und der Steigung des Terrains, sowie der Faktor der Vermeidung, z.B. durch hohe Agentendichte oder große Hindernisse [7]. Diese Kostenfunktion berechnet die Reisezeit, auch „Potential“ (φ) genannt. Dieses Potenzial wird auf die Art und Weise berechnet, sodass es der Eikonal-Gleichung in (2) entspricht.

$$\left|\left|\nabla\varphi(x)\right|\right| = F(x) \tag{2}$$

Wobei $F(x)$ als die positive Kostenfunktion evaluiert durch die Richtung des Gradienten $\nabla\varphi(x)$ zu verstehen ist.

Um diese Gleichung lösen zu können benötigt es aber einer Approximation. Ein leicht zu verstehender Algorithmus zur Annäherung dieser Eikonal-Gleichung ist die „Fast Marching Method“ von Tsitsiklis [15]. Dieser ist hoch seriell und somit nicht für eine parallelisierte Berechnung auf der GPU geeignet. Es gibt aber einen Algorithmus von Joeng und Whitaker [12], die „Fast Iterative Method“, die die Lösung der Eikonal-Gleichung mit Hilfe eines parallelen Algorithmus approximiert. Dieser „Löser“ der Eikonal-Gleichung wurde für die GPU von Shopf et al. [7] im Rahmen einer TechDemo von AMD implementiert. Diese trägt den Namen „March of the Froblins“ und simuliert eine virtuelle Umgebung bestehend aus Bergen und Tälern, in denen so genannte „Froblins“, eine Mischung aus Frosch und Goblin, in großen Massen umherwandern und Futter in Form von Pilzen aufsammeln. Ein Screenshot aus dieser TechDemo kann in Abbildung 3 gesehen werden.

Zur Umsetzung dieser Demo wurde als Technologie die „High Level Shading Language“ (HLSL) verwendet. Der dadurch entstandene „Compute Shader“ wurde auf einer AMD Grafikkarte (ATI Radeon HD 4870) ausgeführt. Dabei ist aber zu erwähnen, dass HLSL eine für Grafikaufgaben konzipierte Sprache ist und für diese Implementierung quasi „zweckentfremdet“ wurde. Der Vorteil dieser Auswahl liegt hierbei in der Kompatibilität der Sprache HLSL mit den einzelnen Grafikchip-Herstellern, denn sowohl Nvidia als auch AMD GPUs sind dank ihrer DirectX-Unterstützung in der Lage diesen Code auszuführen.

Abbildung 3: AMD TechDemo „March of the Froblins“ von Shopf et al. [7].

2.6 Vergleich der Methoden

In den vorhergehenden Abschnitten wurden die einzelnen Methoden zur Umsetzung von künstlicher Intelligenz auf der GPU und die dabei entstandenen Arbeiten vorgestellt. Dabei gingen die Autoren jeweils anders vor und hatten unterschiedlichste Herangehensweisen und Ideen. Sie konzentrierten sich alle auf unterschiedliche Aspekte der KI. Dadurch sind viele unterschiedliche Ergebnisse entstanden, welche nur hie und da auf denselben Grundprinzipien aufbauen.

Ein Grundprinzip, welches von mehreren Autoren aufgegriffen wurde, ist die lokale Kollisionsvermeidung der Agenten gegenüber anderer Agenten mit dem von Fiorini et al. [13] konzipierten „Velocity Obstacle“ Theorem. Bleiweiss verwendet diese in seiner zweiten Arbeit [5] und benutzt eine Erweiterung dieser von Van Den Berg et al. [9], genannt „Reciprocal Velocity Obstacle“ Theorem. In der Arbeit von Shopf et al. [7], in der die AMD TechDemo „March of the Froblins“ entstanden ist, wurde auch das „Velocity Obstacle“ Theorem für die lokale Kollisionsvermeidung der „Froblins“ verwendet. Dabei wurde aber im Gegensatz zur Arbeit von Bleiweiss ein zentralisierter Ansatz zur dynamischen Pfadplanung, namens „Continuum Crowd“ von Treuille et al. [11], verwendet.

So können eben zwei grundlegende Gedanken bzw. Herangehensweisen zur Simulation einer großen Anzahl an Agenten in einer virtuellen Spielszene ausgemacht werden. Dabei handelt es sich um folgende:

1. Dezentralisiert, Berechnung per Agent
2. Zentralisiert, Berechnung per Teilraum der virtuellen Umgebung

Der dezentralisierte Ansatz, so wie er bei Bleiweiss Verwendung findet, ist im ersten Augenblick der naheliegendste und verständlichste. Dabei werden die Agenten einzeln berechnet und besitzen ihr eigenes Wissen über die Umgebung und der umliegenden Agenten sowie ihre eigene Wahrnehmung. Im Prinzip funktioniert die menschliche Navigation, vor allem in großen Menschenmengen, auf dieselbe Art und Weise, bei der eine Kommunikation zwischen den einzelnen Individuen der Gruppe nicht zwingend notwendig ist für deren Interaktion.

Im Gegensatz dazu steht der zentralisierte Ansatz. Dabei wird die Simulation der Agenten von einer zentral liegenden Logik übernommen, welche im Grunde globales Wissen besitzt und dieses an die Agenten gezwungenermaßen weitergibt. Die Methode der „Continuum Crowd“ ist hierbei ein eindeutiger Vertreter dieses Ansatzes. Verwendung findet diese bei Shopf et al. [7] in der Demo „March of the Froblins“. Dabei entstehen durch diesen Ansatz flüssige Gruppenbewegungen und sehr gute Ausweichmanöver, da die Situationen von einer zentralen Stelle aus sehr gut und mit viel Übersicht bewertet werden können anstatt aus einer vergleichsweise kurzen Sicht pro Agent. Der Nachteil ist aber, so wie auch bei Shopf et al. [7] erwähnt, dass es den Anschein erweckt, dass die Agenten Entscheidungen treffen können aufgrund von Wissen, welches sie gar nicht in der Realität haben können. So kann ein Agent einen anderen Pfad wählen, basierend auf einer Situation die außerhalb der vermeintlichen Wahrnehmung stattfindet. Genau dies ist der Nachteil von zentralisierten Lösungen.

Ein weiterer zentralisierter Ansatz wird von Fischer et al. [6] verfolgt, wo ein BVP-Planer basierend auf der Arbeit von Dapper et al. [10] für die GPU implementiert wurde. Dieser BVP-Planer führt seine Berechnungen auch zentralisiert aus. Dabei wurde aber der Schwerpunkt mehr auf ein parametrierbares Agentenverhalten gelegt um die Simulation bzw. die Pfadfindung der Agenten so unabhängig voneinander wie möglich durchzuführen. Dabei wurde versucht den Agenten eine eigene Wahrnehmung ihrer Umwelt zu geben sowie lokales Wissen zu erzeugen und dieses in der zentralisierten Berechnung miteinzubeziehen. So konnten die zuvor erwähnten Nachteile des ungewollten globalen Wissens der Agenten vermieden werden.

Die einzige Arbeit, die sich explizit nur mit der Parallelisierung der Pfadplanungsalgorithmen A* und Dijkstra beschäftigt ist die erste Arbeit von Bleiweiss [4]. Dort werden weniger die dynamischen Aspekte der Agentenbewegung behandelt. Viel mehr liegt hier der Schwerpunkt auf der reinen Berechnung der Algorithmen und der Beobachtung der Performanzunterschiede zwischen einer GPU Implementierung mittels CUDA und einer äquivalenten CPU Implementierung in C++.

Dieser Ansatz wird nun für diese Arbeit übernommen und eigene Implementierungen von Pfadplanungsalgorithmen vorgenommen. Zusätzlich zum Vergleich der Algorithmen selbst werden die CPU und GPU Versionen sowie die GPU Versionen untereinander verglichen. Zuvor wird aber im nächsten Abschnitt die Hardwarearchitektur der GPU erläutert und mit der der CPU verglichen, um die Ergebnisse der zuvor erwähnten Arbeiten noch näher verständlich zu machen und den Grund für die Parallelität der Grafikhardware aufzuzeigen. Zudem werden die Prinzipien vorgestellt, mit denen das nicht grafikrelevante Programmieren auf der GPU durch die Technologien CUDA und OpenCL ermöglicht wird.

Außerdem werden im Kapitel „Pfadplanung und Pfadfindung“ die zu implementierenden Pfadplanungsalgorithmen, darunter der „Greedy Best First Search“ (kurz BFS), der Dijkstra und der A*-Algorithmus (gesprochen A-Stern), vorgestellt. Es werden die Grundlagen der Pfadplanung und der hierfür verwendeten Datenstrukturen erläutert sowie die Funktionsweisen und Abläufe der Algorithmen in Form von Pseudocodes und Text präsentiert.

3 Die Graphics Processing Unit

Die Graphics Processing Unit (kurz: GPU) ist ein Mikrochip, welcher auf so genannte Grafikkarten eingebaut ist. Diese sind wiederum Steckkarten mit einem bestimmten Interface für die im inneren des Computers befindliche Hauptplatine, dem „Mainboard“. Auch Versionen, die sich direkt auf dieser Hauptplatine befinden, sind im Officebereich zu finden. Diese sind aber nicht für die hohen Ansprüche der 3D Computergrafik konzipiert und technisch größtenteils nur für die zweidimensionale Darstellung des Desktops, von Dokumenten, Webseiten, o.Ä. ausgelegt und somit für die in dieser Arbeit dienlichen Zwecke nicht geeignet.

Der Ausdruck GPU kam zuerst im Jahr 1999 auf und blieb bis heute erhalten, als die Nvidia Corporation ihren ersten Mikrochip mit 3D Vertex-Processing veröffentlichte, die Nvidia GeForce 256 [16]. Dieser war in der Lage, Punkte im dreidimensionalen Raum, so genannte Vertices, hardwarebeschleunigt zu verarbeiten. Die Vorgängerkarten waren indes nur in der Lage „Rasterization“ durchzuführen, also das Umrechnen von 3D Daten in für den Bildschirm darstellbare Bildpunkte, so genannte Pixel. Von diesen älteren Karten stammt auch der Ausdruck „Pixel Spitter“, zu Deutsch „Bildpunkt Spucker“, welcher die einfache Aufgabe dieser Karten im Vergleich zu den heutzutage produzierten modernen und komplexen GPUs sehr gut umschreibt.

Es wurden immer weitere Aufgaben in den Grafikchips verwirklicht und beschleunigt. Zu Beginn waren dies, wie eben erwähnt, die „Rasterization“ bis hin zum Vertex-Processing. Im Laufe der Zeit kamen Aufgaben hinzu, welche in einer Schritt für Schritt Abfolge auf der Hardware hintereinander ausgeführt werden und zu einem gerenderten Bild führen. Diese Abfolge nennt sich „Fixed-Function Pipeline“ und beschreibt eine fix vorgeschriebene Reihenfolge von Funktionen der GPU. Die einzelnen voneinander getrennten Teile bzw. Funktionen dieser Pipeline werden „Stages“ genannt.

Die Entwicklung dieser Pipeline begann von hinten, beim letzten Schritt der „Rasterization“. Mit der Zeit wurden immer mehr Funktionen davor hinzugefügt, um diese in Hardware zu beschleunigen, um den für das Real-time Rendering kritischen Zeitfaktor zu verkürzen [16]. Diese Pipeline wurde weiterentwickelt und einzelne Stages wie z.B. das Vertex-Processing und Teile der „Rasterization“ wurden programmierbar gemacht, um mehr Kontrolle und Freiheit über diese zu erlangen und den Grafikprogrammierern die Möglichkeit zu geben eigene Algorithmen und Effekte zu implementieren und auf der GPU auszuführen.

Eine genauere Vorstellung dieser Pipeline sowie der einzelnen Stages wird im nächsten Kapitel gegeben und die programmierbaren Elemente sowie diese, die fix in der Pipeline integriert sind, werden erläutert und beschrieben.

3.1 Die Rendering Pipeline

Die Rendering Pipeline beschreibt den Ablauf von Aufgaben bzw. Stages zum Rendern eines zweidimensionalen Bildes aus dreidimensionalen Welt- und Kameradaten. Die Theorie dieser Stages sowie deren Konzeption werden detailliert in Akenine-Möller et al. [16] beschrieben und würden in diesem Detailgrad den Umfang dieser Arbeit sprengen. Deswegen wird im folgenden Abschnitt nur oberflächlich der hardware- und programmiertechnische Aufbau dieser Pipeline vorgestellt. Eine grafische Repräsentation hierfür ist in Abbildung 4 zu sehen. Dabei ist zu erkennen, dass diese Pipeline in acht Stages unterteilt ist. Sie entspricht dem Shader Model 4.0, welche mit der DirectX 10 API implementiert und zusammen mit Windows Vista im Jahr 2007 veröffentlicht wurde. Die Stages sind mit unterschiedlichen Farben gekennzeichnet, darauf ankommend, wie sehr diese programmierbar oder konfigurierbar sind [16].

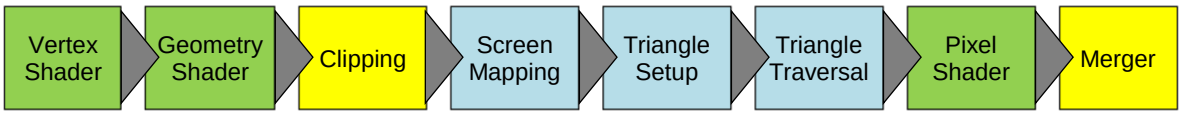


Abbildung 4: Darstellung der Rendering Pipeline der GPU (Shader Model 4.0) [16].

Die blauen Stages sind fix und können weder programmiert noch konfiguriert werden. Dabei handelt es sich um Teile der schon im letzten Abschnitt erwähnten „Rasterization“. Die erste Stage, das Screen Mapping, transformiert die Vertices der 3D-Szene in Bildschirmkoordinaten um. Danach wird die nächste Stage, das „Triangle Setup“ durchgeführt, welche die Dreiecke für den nächsten Schritt vorbereitet. Die „Triangle Traversal“ Stage weist die Bildschirmkoordinaten und Dreiecksflächen der zuvor durchgeführten Stages den Pixeln des Ausgabefensters zu und führt hierfür auch Interpolationen durch. Diese Stages folgen immer demselben Prinzip und sind somit fix in dieser Pipeline integriert.

Die gelben Stages sind im Gegensatz zu den blauen konfigurierbar. Dadurch können unterschiedliche Ergebnisse des Renderprozesses erzielt werden, darauf ankommend wie diese Stages konfiguriert werden. Die erste konfigurierbare Stage ist die, die für das so genannte „Clipping“ zuständlich ist. „Clipping“ ist ein Prozess, bei dem die Vertices und Dreiecke außerhalb des View-Frustums, also der Bildschirmabgrenzung, „weggeschnitten“ werden. Dabei werden Vertices von Geometrie, die sich nur zum Teil außerhalb dieses Frustums befindet, umgerechnet und auch neue Vertices hinzugefügt. Dieses View-Frustum ist in dieser Stage konfigurierbar und kann nicht nur Geometrie am Rand des Ausgabefensters „wegschneiden“, sondern auch Objekte die zu nahe oder zu weit weg sind für die nächsten Stages ausschließen (Stichwort: Near und Far Clipping Plane).

Die letzte Stage der Pipeline, der „Merger“, der aus den gerenderten Objekten das finale Bild produziert, ist auch konfigurierbar. Dieser arbeitet mit dem z-Buffer, auch Depth-Buffer genannt, der die Tiefenwerte der Pixel, also wie weit weg sie sich vom Betrachter befinden, abspeichert und nur dann überschrieben werden, wenn das danach gerenderte Pixel näher beim Betrachter ist. Zusätzlich gibt es den Stencil-Buffer, der beliebige Werte speichern kann. Diese Puffer können sehr zahlreich und unterschiedlich konfiguriert werden um Effekte erzielen zu können wie Spiegelungen, transparente Flächen, und noch vieles mehr.

Die grünen Stages sind komplett programmierbar und sind somit „leer“ bzw. „blank“. Die Funktionalität dieser Stages muss erst in Form von Shadern implementiert werden, welche den Entwicklern im Grunde alle Möglichkeiten zur Implementierung eigener Algorithmen bzw. Effekten bieten. In der Vergangenheit implementierte die Fixed-Function Pipeline diese Stages auf sehr einfache Weise. Dies reichte aber mit der Zeit nicht aus und beschränkte die Möglichkeiten der GPU und die Qualität der gerenderten Bilder, welche nur durch die Shader den heute zu erkennenden hohen Standard erreichen konnten.

Die erste programmierbare Shader Stage ist der so genannte Vertex Shader (VS). Dieser erhält Daten in Form von Vertices als Punkte in der 3D-Welt, die durch Dreiecke verbunden eine darstellbare „Triangle-Mesh“ ergeben. Solch eine „Triangle-Mesh“ definiert sich durch diese Vertices und zusätzliche Informationen darüber, wie die einzelnen Vertices zu Dreiecken verbunden sind. Diese Informationen stehen dem Vertex Shader aber nicht zur Verfügung, denn dieser erhält nur Vertices, welche mehrere Daten besitzen, darunter aber mindestens einen Positions- und einen Farbwert. Der Vertex Shader führt keine grafischen Aufgaben durch, sondern ist vielmehr eine Stage zum Manipulieren und Bearbeiten der Vertices einer „Triangle-Mesh“, bevor diese zu den nächsten Stages der Pipeline weitergereicht wird. Die zahlreichen Effekte, die durch den Vertex Shader implementiert werden können, sowie auch grafische Darstellungen hierfür, können bei Akenine-Möller et al. [16] detailliert nachgelesen werden.

Am gegenüberliegenden Ende der Pipeline befindet sich der Pixel Shader (PS). Dieser erhält von der vorhergehenden Stage, dem „Triangle Traversal“, die Bildpunkte der in der Szene befindlichen Dreiecke, dessen Vertices der Vertex Shader zuvor bearbeitet hat. Im Pixel Shader können die Farbwerte dieser Bildpunkte modifiziert werden, z.B. mit Hilfe von Texturen, Lichtdaten sowie den Normalen und Texturkoordinaten der Vertices, deren Daten in der Regel zuvor im Vertex Shader berechnet oder zumindest in die weiteren Stages weitergereicht wurden. Dadurch können, zusätzlich zu texturierten Flächen, auch Effekte entwickelt werden wie das bekannte „Bump Mapping“ bzw. das „Normal Mapping“ sowie das „Parallax Mapping“. Diese Effekte erzeugen sehr performant, im Zusammenspiel mit Licht- und Höhendaten in Form von Texturen, zusätzliche Details in den texturierten Oberflächen der Szene, welche sonst nur durch eine extrem starke Erhöhung der Anzahl an Vertices erreicht werden können.

Seit dem Shader Model 4.0, welches detaillierter im nächsten Kapitel erläutert wird, gibt es eine zusätzliche optionale Shader Stage, den Geometry Shader. Diese Shader Stage befindet sich, so wie in der Darstellung der Rendering Pipeline in Abbildung 4 zu erkennen, direkt nach der Vertex Shader Stage. Als Eingangswerte bzw. Input erhält diese Shader Stage ein Objekt bestehend aus Vertices z.B. in Form eines Dreiecks, eines Liniensegments oder einfach nur eines einzelnen Punktes. Diese Formen können in Abbildung 5 von Akenine-Möller et al. [16] gesehen werden.

Abbildung 5: Formen für Input- und Output-Geometrie für den Geometry Shader [16].

Der Rückgabewert des Geometry Shaders muss auch einer dieser Formen entsprechen, wobei die Input-Geometrie nicht mit der Output-Geometrie übereinstimmen muss. In der Stage selbst können, im Gegensatz zum Vertex Shader, neue Geometrie bzw. Vertices erzeugt und schon vorhandene entfernt werden. Es ist aber auch möglich einfach nur Vertices zu manipulieren, so wie es auch im Vertex Shader möglich ist. Somit kann der Geometry Shader als eine Erweiterung des Vertex Shaders angesehen werden.

Eine weitere Neuerung des Shader Models 4.0 ist das so genannte Prinzip des „Stream Output“. Anstatt dass Daten vom Vertex Shader bzw. optional vom Geometry Shader zu den nachfolgenden Stages der Pipeline weiter gereicht werden wie bisher, können diese auch als Datenstrom, eben als „Stream“, der Applikation zurückgegeben werden, z.B. in Form eines Arrays. Die nachfolgenden Stages, welche die „Rasterization“ durchführen, können komplett abgeschaltet werden und die GPU als ein „nicht-grafischer Stream-Prozessor“ verwendet werden [16].

In dieser Tatsache liegt der Ursprung des General Purpose Computing auf der GPU. Der auszuführende Shader-Code kann einfach, anstatt Vertices oder Geometrie zu berechnen und spezifische Operationen für das Rendering auszuführen, einfach nicht-grafikrelevante Aufgaben implementieren. Anstatt dass die Pipeline nach der Berechnung fortgesetzt wird, wird diese einfach abgebrochen und die berechneten Werte mittels „Output Stream“ zurück zur Applikation geschrieben, wo sie dann beliebig weiter verarbeitet werden können.

Die Entstehung der Shader bzw. auch der einzelnen Shader Models, bis zum für das GPU-Computing wichtige Shader Model 4.0, werden im nächsten Kapitel genauer erläutert. Es werden zudem auch die Programmiersprachen, mit denen diese Shader unter den unterschiedlichsten APIs und Plattformen entwickelt werden können, vorgestellt.

3.2 Programmable Shaders und Shader Models

Programmierbare Shader kamen zuerst mit dem Release von Microsofts DirectX 8 im Jahr 2000 auf [17]. Davor unterstützen GPUs nur die schon im vorherigen Abschnitt erwähnte Fixed-Function Pipeline. Unter DirectX 8 wurde die Rendering Pipeline programmierbar, indem die fix vorgegebenen Befehlsätze der Fixed-Function Pipeline frei kombinierbar wurden. Die ersten wirklich programmierbaren Vertex- und Pixel-Shading Stages kamen erst mit dem Release von DirectX 9 und der dazu kompatiblen GPUs auf, welche das Shader Model 2.0 unterstützten bzw. implementierten.

Dieses erste, wirklich programmierbare Shader Model 2.0 beinhaltete einige Restriktionen für den Grafikprogrammierer, vor allem bezüglich der Anzahl an Instruktionen pro Shader-Stage und der Anzahl an verwendbaren Texturen. Außerdem gibt es im Shader Model 2.0 keine „Flow-Control“, sprich programmatische Anweisungen wie if- oder case-Statements sowie auch Schleifen werden nicht unterstützt. Genauere Informationen über die Spezifikationen der einzelnen Shader Models sowie deren Unterschiede und Fähigkeiten sind in Tabelle 1 von Akenine-Möller et al. [16] zu finden.

	Shader Model 2.0	Shader Model 3.0	Shader Model 4.0
Release	DirectX 9.0, 2002	DirectX 9.0c, 2004	DirectX 10.0, 2007
VS max. Instruktionen	256	≥ 512 (1)	4096
VS max. Steps	65536	65536	∞
PS max. Instruktionen	≥ 96 (2)	≥ 512 (1)	≥ 65536 (1)
PS max. Steps	≥ 96 (2)	65536	∞
Temp. Register	≥ 12 (1)	32	4096
VS konstante Register	≥ 256 (1)	≥ 256 (1)	14 x 4096 (3)
PS konstante Register	32	224	14 x 4096 (3)
Dyn. Flow-Control	✗	✓	✓
VS Texturen	✗	4 (4)	128 x 512 (5)
PS Texturen	16	16	128 x 512 (5)
Integer Support	✗	✗	✓
VS Input Register	16	16	16
Interpolationsregister	8 (6)	10	16 / 32 (7)
PS Output Register	4	4	8

(1) Mindestanforderung (mehr können verwendet werden wenn vorhanden)

(2) Mindestanforderung von 32 Textur- und 64 Arithmetikinstruktionen

(3) 14 Konstantenpuffer, die je bis zu 4096 Konstanten beinhalten können

(4) Shader Model 3.0 Hardware hat meist begrenzte Formate und keine Filterung für Vertex Shader Texturen

(5) Bis zu 128 Textur-Arrays mit je bis zu 512 Texturen

(6) Nicht inkludierend zwei Farbinterpolatoren mit limitierter Präzision und Reichweite

(7) Vertex Shader Output sind 16 Interpolatoren, welche der Geometry Shader auf 32 erhöhen kann

Tabelle 1: Vergleich der einzelnen Shader Models von Akenine-Möller et al. [16].

Als Programmiersprache für diese Shader stehen, je nach Plattform und API, unterschiedliche Sprachen zur Verfügung. Unter DirectX ist dies die „High Level Shading Language“ (HLSL). Für das frei zugängliche und nach DirectX beliebteste Grafikframework OpenGL [18] gibt es auch eine Shading Language namens GLSL (OpenGL Shading Language). Nvidia hat, parallel zur Entwicklung von Microsofts HLSL, ihre eigene, plattformspezifische Variante namens Cg (C for Graphics) entwickelt. All diese Shader Sprachen sind sich sehr ähnlich und unterscheiden sich nur in ihrer Syntax und in der Namensgebung derer Funktionen. Sie leiten sich alle von der bekannten Programmiersprache „C“ ab und besitzen zusätzliche, grafikspezifische Datentypen wie z.B. „float3“ oder „float4“ für Vektoren und „float3x3“ oder „float4x4“ für Matrizen (HLSL). Auch grafikspezifische Funktionen wie die Berechnung des Skalarprodukts und Kreuzprodukts von Vektoren sowie die Multiplikation von Matrizen wird performant unterstützt [16].

Der Shader-Code wird zur Laufzeit kompiliert und in eine maschinenunabhängige Assemblersprache, die so genannte „Intermediate Language“ (IL), konvertiert. Diese wird in einem weiteren Schritt, in der Regel vom Grafikkartentreiber, in den eigentlichen Maschinencode für die GPU übersetzt. Dieses Arrangement erlaubt die Kompatibilität des Shader Codes für unterschiedliche Hardwareimplementierungen zu gewährleisten [16].

Seit dem Shader Model 4.0 basieren alle Shader Stages auf demselben Aufbau, den so genannten „Common-Shader Core“. Dieser ist als Diagramm in Abbildung 6 zu erkennen, welches auf einer Abbildung aus der MSDN-Library von Microsoft [17] basiert. Dieses zeigt die einzelnen Komponenten der Shader Stages in diesem „Common-Shader Core“ sowie den durch Pfeile repräsentierten Programmablauf und Datenfluss zwischen diesen.

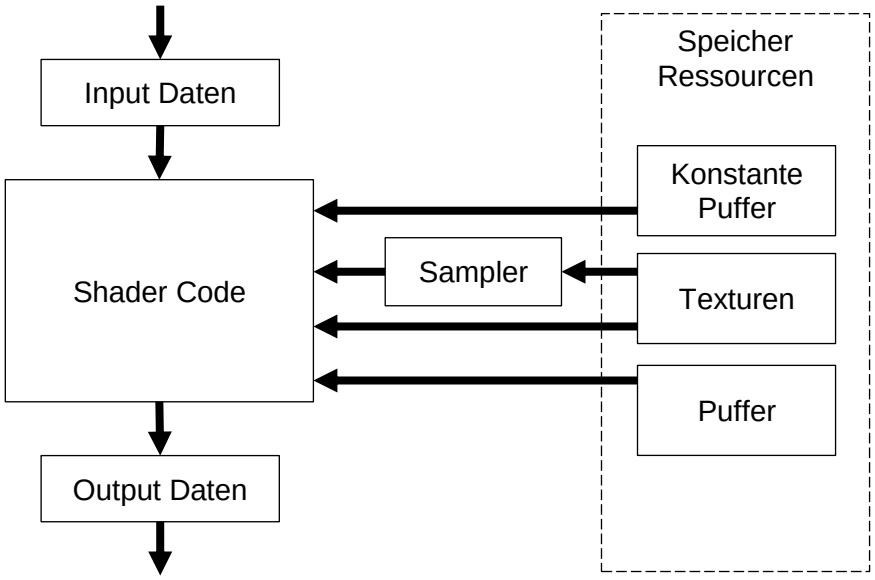


Abbildung 6: Datenfluss einer Shader Stage im „Common-Shader Core“ [17].

Bei den Komponenten im Diagramm des „Common-Shader Core“ in Abbildung 6 handelt es sich um folgende:

1. Input Daten: Der Shader erhält Eingangsdaten von der vorherigen Stage
2. Output Daten: Der Shader schickt Ausgangsdaten zur nächsten Stage
3. Shader Code: Der Shader kann Daten von den Speicher Ressourcen lesen, arithmetische Operationen ausführen und den Programmverlauf mit if-Statements und Schleifen steuern (Flow Control)
4. Sampler: Definieren wie Texturen mittels „Texture Sampling" gefiltert und gelesen werden sollen
5. Texturen: Texturen können gefiltert durch die Sampler vom Speicher gelesen werden oder per Element bzw. Texel (Bildpunkt)
6. Puffer: Können wie Texturen vom Speicher gelesen werden, aber im Unterschied zu diesen nur per Element
7. Konstante Puffer: Wie die Puffer, aber optimiert für konstante Shader-Variablen und häufige Zugriffe seitens der CPU

Hardwaretechnisch wird dieser „Common-Shader Core“ in Form eines Verbundes vieler kleiner Recheneinheiten, so genannten ALUs (Arithmetisch-logische Einheiten), auf der GPU verwirklicht. Diese sind in der Lage die programmierten Aufgaben der Shader Stages für jedes Inputelement gleichzeitig auszuführen. So kann jeder Vertex, der in den Vertex Shader gelangt, sowie auch jeder Pixel im Pixel Shader und Geometrieobjekt im Geometry Shader, parallel von mehreren SIMD-Prozessoren auf die gleiche Weise bearbeitet werden. Da dies auf Hardwareebene geschieht, ist die GPU in Bereich des 3D-Renderings im Gegensatz zur CPU um ein großes Stück performanter und schneller.

Dieser Verbund von unabhängigen und baugleichen SIMD-Prozessoren in der GPU nennt sich „Unified Shader“-Architektur [16]. Diese Architektur implementiert ausgezeichnet das „Unified Shader“-Model, welches dem Shader Model 4.0 sowie auch dem zuvor schon erwähnten „Common-Shader Core“ entspricht. Die Unterscheidung zwischen den Shader Stages ist in der Hardware, im Gegensatz zu älteren GPUs, nicht mehr gegeben. Alle Recheneinheiten können sowohl Aufgaben für den Vertex Shader, den Geometry Shader als auch für den Pixel Shader, ausführen. So kann die Last der in den Shader Stages auftretenden Berechnungen gleichmäßig auf die Recheneinheiten aufgeteilt werden und ein so genanntes „Workload Balancing“ erfolgen [16].

Genauere Details zur Hardwarearchitektur von GPUs und der „Unified Shader“-Architektur können im nächsten Kapitel gefunden werden. Außerdem wird folgend auch ein Vergleich der Architektur und der Rechenleistung von GPUs und CPUs gegeben.

3.3 Vergleich von GPU und CPU

Die Entwicklung im Bereich der Mikrochips und Mikroprozessoren war in den letzten Jahren und Jahrzehnten enorm. Keine Entwicklung in der Geschichte der Menschheit war über einen vergleichbar langen Zeitraum so anhaltend konstant hoch. Sie unterliegt dem so genannten „Mooreschen Gesetz“, im Englischen auch als „Moore’s Law“ bekannt, welches 1965 von Gordon E. Moore definiert wurde [19]. Dieses besagt, dass sich die Komplexität sowie auch die Dichte von integrierten Schaltkreisen auf einem Mikrochip pro Flächeneinheit in regelmäßigen Abständen verdoppeln. Diese Annahme, auch wenn sie schon Jahrzehnte alt ist, war korrekt und hat heute noch bestand. In der Realität geschieht solch eine regelmäßige Verdoppelung wie bei Moore beschrieben ca. alle 20 Monate.

Eine CPU besteht, so wie auch bei Glaskowsky [20] beschrieben, zum größten Teil aus Schaltkreisen mit verborgener Logik zur Steigerung der Performanz bei linearen bzw. single-threaded Programmausführungen. Diese bauen zum größten Teil auf Spekulationen auf, welche nach bestimmten Regeln Performanzsteigerungen erzielen können. Hierzu zählen das Caching, welches Daten, die wahrscheinlich später wieder gebraucht werden, in Caches speichert, um schneller darauf zugreifen zu können. Auch die „Branch Prediction“, also das Vorhersagen von Sprüngen im Code wie bei if-Anweisungen und Schleifen, sowie „Out-Of-Order-Execution“, also das Ausführen von Instruktionen in anderer Reihenfolge um z.B. Speicherlatenzen entgegenzuwirken, sind Beispiele für spekulative Chiplogiken auf einer CPU [20].

Eine GPU hingegen besitzt solche spekulative Logiken nicht. Der größte Teil eines Grafikchips besteht aus ALUs (Arithmetisch-logische Einheiten). Dies bedeutet, dass der größte Teil der Transistoren auf dem Chip auch wirklich für Berechnungen verwendet wird. Eine bildliche Darstellung der GPU- und CPU-Architektur sowie deren Mengenaufteilung der Transistoren auf die einzelnen Aufgaben ist in Abbildung 7 zu erkennen, welche auf einer Abbildung aus dem „CUDA C Programming Guide“ von Nvidia basiert [2].

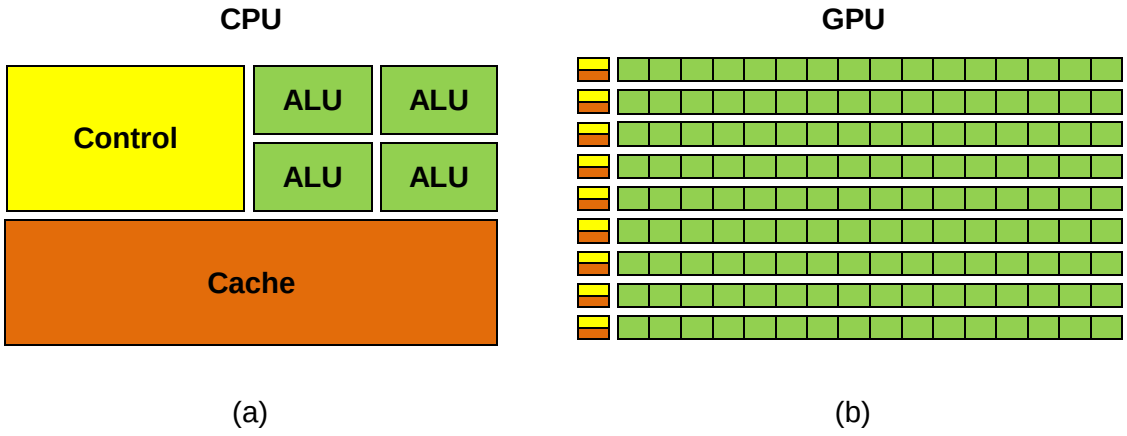


Abbildung 7: Darstellung der Architektur einer (a) CPU und (b) GPU [2].

Wie in Abbildung 7 zu erkennen ist, verwendet die CPU (a) für deren Kontrolleinheit (Control), farblich mit Gelb markiert, sowie für den internen Cache, in orange, einen sehr großen Teil des Chipsatzes. Dies ist aufgrund der zuvor schon erwähnten spekulativen Chiplogiken zurückzuführen, die die CPU implementiert. Die CPU in Abbildung 7 (a) besitzt vier Recheneinheiten (ALUs), zu erkennen an den grünen Markierungen, und ist somit ein Vier-Kern bzw. Quad-Core Prozessor.

Die GPU (b) hingegen besitzt hunderte Kerne, welche parallel bzw. multi-threaded arbeiten und hierfür optimiert sind. Jede Reihe dieses Diagramms in (b) entspricht einem Streaming-Multiprozessor auf der GPU, welcher jeweils ein Cluster von Kernen verwaltet. Solch ein Streaming-Multiprozessor besitzt jeweils eine Kontrolleinheit, in Gelb dargestellt, sowie auch einen gemeinsamen Cache, dargestellt in orange. Diese Einheiten sind aber im Vergleich zu den Kontrolleinheiten und Caches der CPU in (a) kleiner und somit auch weniger komplex.

Das Hauptaugenmerk bei einer GPU liegt im Verarbeiten und Berechnen von vielen Daten gleichzeitig, mit wenig Aufwand und auch so schnell wie möglich. Deswegen besitzen Grafikchips in der Regel mehr Rechenleistung als eine CPU. Dieser Trend, welcher sich vor allem in den letzten Jahren bemerkbar machte, ist in Abbildung 8 in Form mehrerer Graphen dargestellt, welche verschiedenste GPUs von Nvidia und CPUs von Intel, welche in den letzten Jahren entwickelt und hergestellt wurden, miteinander vergleicht.

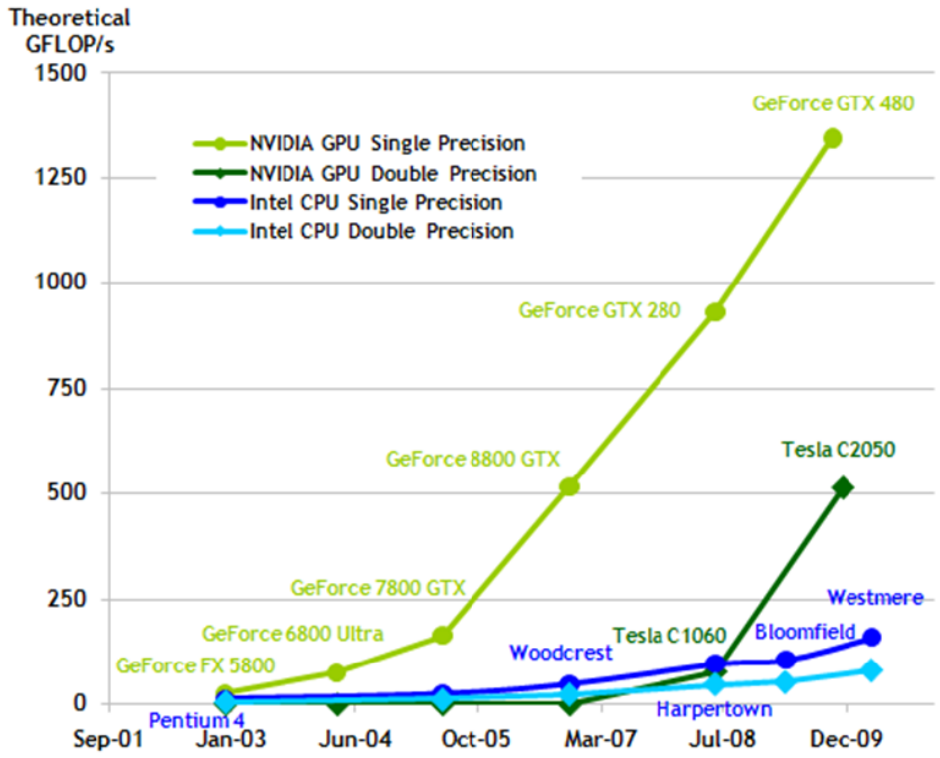


Abbildung 8: Vergleich der Rechenleistung von GPUs und CPUs in GFLOPS [2].

Der Graph in Abbildung 8 zeigt die theoretische Rechenleistung verschiedener GPUs und CPUs in GFLOPS (Giga-FLOPS), also tausend Gleitkommaoperationen pro Sekunde. Es ist eindeutig zu erkennen, dass Grafikchips die CPUs, vor allem bei „Single Precision“, also „float“ Gleitkommaberechnungen, weit überholt haben. Dies wurde vor allem durch die gestiegene Anzahl an Streaming-Prozessoren auf den Grafikchips erreicht.

Zusätzlich zur Kernanzahl spielt aber auch die Bandbreite des Speichers zum Chip eine große Rolle in der Performanz der GPUs und CPUs. Wie in Abbildung 9 gut zu erkennen ist, ist die Bandbreite und somit auch der Durchsatz der Speicher für GPUs um einiges höher. Dieser Speicherdurchsatz von GPUs und CPUs wird in diesem Diagramm in Gigabyte pro Sekunde (GB/s) gemessen. Dies wird vor allem durch den Verbau von schnelleren Speichereinheiten im Vergleich zum Arbeitsspeicher für die CPU auf den Grafikkarten erreicht. Dieser Speicher ist aber auch teurer, weswegen Grafikkarten im Vergleich weniger Speicher besitzen als ein PC an Arbeitsspeicher für dessen CPU.

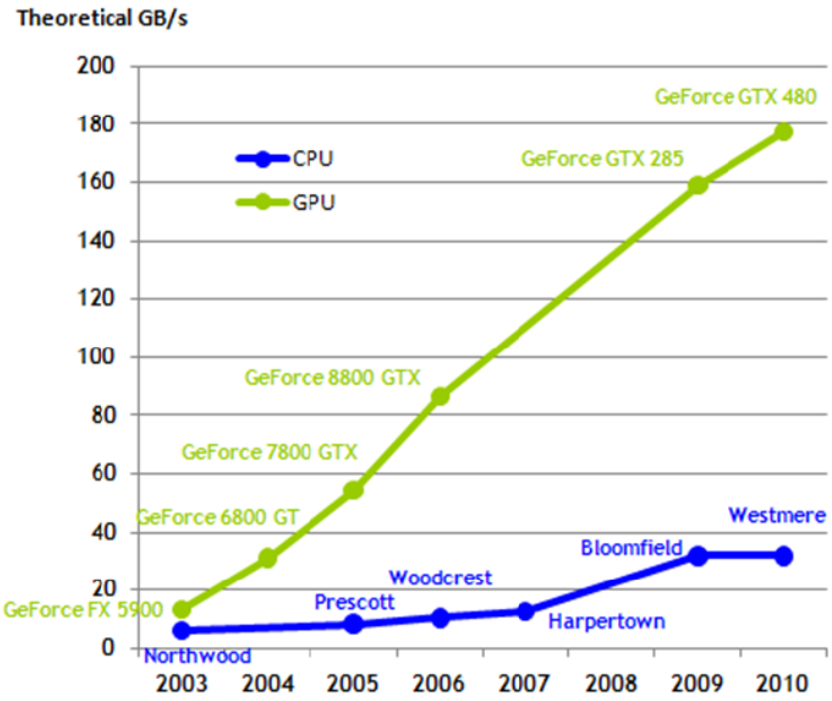


Abbildung 9: Vergleich der Memory-Bandbreite von GPUs und CPUs in GB/s [2].

Warum CPUs weiterhin mehr für Berechnungen und das Computing verwendet werden als GPUs hat zum größten Teil geschichtliche Gründe. Die Compiler, also die Programme zum Erstellen von ausführbaren Dateien aus Quellcode, entwickelten sich, so wie auch bei Glaskowsky [20] beschrieben, parallel mit den CPUs mit. Sie machen, so wie die Programmiersprachen, die Entwicklung von Programmen für die CPU mit der Zeit immer einfacher und verbergen die komplexe Architektur der CPUs komplett. Außerdem können Compiler von sich heraus Optimierungen durchführen und viele Probleme, die Software

Developer in derer Entwicklungsarbeit stören würden, abnehmen. Zum Unterschied ist die Entwicklung für die GPU sehr jung und komplexer im Vergleich zur aus historischen Gründen gewohnten linearen bzw. single-threaded Programmierung für die CPUs. Außerdem ist die Verbreitung von GPUs, die für das GPU-Computing in der Lage sind, noch sehr gering, wodurch die Kompatibilität mit unterschiedlichen Computersystemen nicht gewährleistet werden kann.

Die Entwicklung der GPU Development Technologien schreitet dank zahlreich beteiligter Firmen wie Nvidia und Microsoft sehr stark voran. Die Komplexität der GPU-Hardware wird immer mehr vom Programmierer wegabstrahiert, um die Hemmschwelle vom alten, gewohnten single-threaded CPU-Systems zum neuen Computing-Systems mit parallel arbeitenden Threads auf einer GPGPU zu erleichtern.

Dabei ist zu erkennen, dass bei all den zurzeit erhältlichen APIs und Technologien für das GPU-Computing, egal von welchem Unternehmen und für welche Plattformen entwickelt, dieselben grundlegenden Prinzipien zu finden sind. Bei den gängigsten Entwicklungen in diesem Bereich handelt es sich, wie schon des Öfteren in dieser Arbeit erwähnt, um CUDA [2] und OpenCL [3], aber auch um DirectCompute [17]. Die grundlegenden Prinzipien dieser APIs werden nun in den folgenden Abschnitten vorgestellt und genauer erläutert.

3.4 Das GPU-Computing

Beim GPU-Computing handelt es sich, wie schon in den letzten Abschnitten angedeutet, um eine Ausführung von nicht grafikrelevantem Code auf einer GPU. Hierfür wird die hoch parallele Hardwarearchitektur der GPU, bestehend aus mehreren Stream-Prozessoren optimiert für grafikrelevante Aufgaben, verwendet, um allgemeinen Code mit Hilfe einer GPU-Computing API zu parallelisieren und auszuführen.

Die APIs und Technologien, welche für das GPU-Computing verwendet werden können, bieten eine Schnittstelle zu einer oder auch mehreren im System vorhandenen Grafik- bzw. Computing-Karten und kann Code, so genannte „Kernels“, auf diesen ausführen. Dabei wird der im Kernel beschriebene Code mehrmals parallel auf der GPU ausgeführt, nur auf unterschiedliche Daten bzw. Speicheradressen, sodass Berechnungen zugleich auf unterschiedliche Datensätze durchgeführt werden können. Dies entspricht der SIMD-Architektur (Single Instruction – Multiple Data) der GPUs.

Die Daten, welche die Kernels für deren Berechnungen verwenden, müssen sich auf dem Speicher der Grafik- bzw. Computing-Karte befinden. Daten, welche sich im Arbeitsspeicher des Systems befinden und von der GPU für deren Berechnungen verwendet werden sollen, müssen erst in den globalen Speicher der Grafik- bzw. Computing-Karte, mit Hilfe der verwendeten API, kopiert werden. Die von der GPU berechneten Werte, welche nach der Berechnung auch in den globalen Grafikkartespeicher geschrieben werden, werden

dann wieder mit Hilfe der API zurück in den Arbeitsspeicher des Systems kopiert und können nun von der CPU bzw. vom Programm weiter verwendet werden. Dies führt schlussendlich zu einer heterogenen Architektur, in der sich Codeabschnitte, welche sich gut parallelisieren lassen und dadurch auch einen Performanzschub erhalten, auf der GPU ausgeführt werden, abwechselnd mit normalen, seriell arbeitenden Programmcode, ausgeführt auf der CPU (siehe „CUDA C Programming Guide“ [2]).

Um diese Trennung zwischen den beiden Codesegmenten auf der CPU und GPU sowie deren Speicherbereiche, Arbeits- und Grafikkartenspeicher zu gewährleisten, wird in jeder GPU-Computing API das Prinzip von „Host“ und „Device“ implementiert. Dieses wird nun im nächsten Abschnitt präsentiert und genauer erläutert.

3.4.1 Prinzip von Host und Device

Bei einem „Host“ handelt es sich um ein Computersystem, welches eine CPU besitzt und ein oder mehrere „Devices“, welche die APIs ausfindig machen und auf ihnen Kernels ausführen können. Im Allgemeinen kann gesagt werden, dass es sich beim „Host“ um einen PC handelt. Das „Device“ hingegen ist eine GPU, von denen mehrere im System, also im „Host“, vorhanden sein können (Stichwort: Multi-GPU).

Beide Teile, der „Host“ und das „Device“, haben voneinander getrennte Speicherbereiche. Der Arbeitsspeicher des „Host“ wird der Einfachheit halber „Host-Memory“ bezeichnet. Die Speicherbereiche des „Device“, also der Arbeitsspeicher und die On-Chip Speicher der Grafik- bzw. Computing-Karte, werden zusammengefasst als „Device-Memory“ bezeichnet.

Der „Host“ kann Daten in den Speicher des „Device“ sowohl lesen als auch schreiben und kann Berechnungen in Form von Kernels auf dem „Device“ starten lassen, alles mit Hilfe der GPU-Computing API. Das „Device“ hingegen kann nur in den eigenen Speicherbereichen lesen und auch nur in gewisse schreiben. Diese unterschiedlichen Speicherbereiche des „Device“ sowie deren Eigenschaften wie Lese- und Schreibrechte sowie Zugriffszeiten bzw. Latenzen werden nun in Tabelle 2 erläutert, zusammengestellt aus Informationen aus dem „CUDA C Programming Guide“ [2] und der OpenCL Spezifikation [3]:

Benennung		Speicherzugriff		Speicher-latenz	Definition
CUDA	OpenCL	read	write		
local	private	✔	✔	schnellste	Variablen des Kernels (On-Chip, in Registern)
constant	constant	✔	✘	schnell	Konstante Variablen (On-Chip), nur lesen
shared	local	✔ (1)	✔ (1)	schnell	Speicher für Block bzw. Work-Group (On-Chip)
global	global	✔	✔	langsam	Globaler Arbeitsspeicher (RAM) des „Device“

(1) Nur innerhalb des Thread-Blocks (CUDA) bzw. der Work-Group (OpenCL) erreichbar (siehe Kapitel 3.4.2 für Details!)

Tabelle 2: Speicherbereiche des „Device“ bei CUDA [2] und OpenCL [3].

Zusätzlich zu den Speicherbereichen aus Tabelle 2 kann bei GPUs bzw. bei Grafikkarten noch der so genannte „Texturspeicher“ unterschieden werden. Dieser befindet sich im globalen Speicherbereich, im RAM, verfügt aber über eine höhere Bandbreite beim Zugriff auf diesen als ein normal allokierter globaler Speicher. Außerdem werden Zugriffe auf Texturen, im Gegensatz zu anderen Leseoperationen im globalen RAM-Speicher, von der Grafikkarte gecached, was wiederum zu Performanzsteigerungen führen kann [2].

Innerhalb von Kernels, welche auf dem „Device“ ausgeführt werden, können diese Speicherbereiche sowie Pointer auf diese frei verwendet werden. Der auf dem „Device“ ausgeführte Code wird der Einfachheit halber „Device-Code“ genannt. Darauf ankommend welche API verwendet wird, ist dieser „Device-Code“ in einer eigenen, meist C-ähnlichen Programmiersprache geschrieben. Im Fall von CUDA ist dies die CUDA C Erweiterung mit eigener Syntax und Schlüsselwörter [2]. Bei OpenCL handelt es sich beim „Device-Code“ um OpenCL C [3], welcher auch eine Erweiterung des C-Standards darstellt. Nur bei DirectCompute von Microsoft wird der „Device-Code“, so wie die Shader bei DirectX, direkt in HLSL geschrieben und auch ausgeführt [17].

Der „Device-Code“ steht im direkten Kontrast zum „Host-Code“, welcher z.B. in C++ geschrieben und auf der CPU ausgeführt wird. Es ist kein direkter Zugriff vom „Host-Code“ auf die Variablen auf dem „Device“ möglich. Nur über API-Funktionen und Klassen bzw. eigene Datentypen wird dies von CUDA bzw. OpenCL ermöglicht und implementiert.

3.4.2 Prinzip von Threads, Blocks und Grids

Da ein Kernel auf der GPU mehrmals parallel ausgeführt wird, muss beim Starten einer Berechnung auf der GPU mit Hilfe der API im Vorhinein bestimmt werden, wie viele Threads, also wie oft der Kernel, gestartet werden soll. Solch ein „Thread“ bei CUDA bzw. „Work-Item“ bei OpenCL befindet sich zusammen mit anderen „Threads“ bzw. “Work-Items“ in einer Gruppe, bei CUDA genannt „Block“ bzw. bei OpenCL „Work-Group“.

Die Threads bzw. Work-Items dieser Blocks bzw. Work-Groups werden innerhalb eines Streaming-Multiprozessors der GPU parallel berechnet. Der Multiprozessor verfügt aber nur über beschränkte Ressourcen. So ist z.B. die maximale Anzahl der Register pro Block bzw. pro Work-Group oder die maximale Anzahl an Threads bzw. Work-Items pro Prozessor definiert und müssen mit Hilfe der API abgefragt und beachtet werden.

Solch ein Block bzw. solch eine Work-Group aus Threads bzw. Work-Items teilen sich einen gemeinsamen Speicherbereich auf dem Streaming-Multiprozessor auf dem sie ausgeführt werden, nämlich die „Shared-Memory“ bei CUDA bzw. „Local-Memory“ bei OpenCL. Nur Threads bzw. Work-Items innerhalb eines Blocks bzw. einer Work-Group können auf derselben „Shared-Memory“ bzw. „Local-Memory“ zugreifen.

Alle Blocks bzw. Work-Groups zusammen entsprechen dem gesamten Ausmaß der durchzuführenden Arbeit, also die Anzahl der auszuführenden Threads bzw. Work-Items für die Berechnung. Bei CUDA wird dieses Ausmaß „Grid“ genannt und entspricht der Anzahl der auszuführenden Blocks, welche alle dieselben Ausmaße besitzen. Bei OpenCL hingegen entspricht dies der „Work-Dimension“ bzw. der so genannten „NDRange“, welche der gesamten Anzahl an allen auszuführenden Work-Items entspricht.

Sowohl Blocks bzw. Work-Groups als auch Threads bzw. Work-Items können, so wie in Abbildung 10 ersichtlich, in mehreren Dimensionen innerhalb eines Grids bzw. einer NDRange und eines Blocks bzw. einer Work-Group verwaltet werden. So kann, wie in Abbildung 10 dargestellt, ein Grid bzw. eine NDRange bestehend aus 2x3 Blocks bzw. Work-Groups definiert werden, indem die Threads bzw. Work-Items ebenfalls zweidimensional angeordnet sind und eine eigene ID besitzen, errechnet unter der Zuhilfenahme von Informationen wie der gesamten Anzahl an Threads bzw. Work-Items und die Ausmaße der Blocks bzw. Work-Groups sowie des Grids bzw. der NDRange selbst.

Sowohl die Blocks bzw. Work-Groups als auch das Grid bzw. die NDRange können in bis zu drei Dimensionen verwaltet werden. Mit Hilfe dieser Aufteilung können Elemente von Daten wie zweidimensionalen oder auch dreidimensionalen Arrays einem gewissen Thread zugeschrieben werden, welcher dann genau auf diesen Daten die im Kernel programmierten Operationen ausführt, und zwar parallel zu den anderen Threads bzw. Work-Items in diesem Block bzw. in dieser Work-Group.

Dabei werden nicht alle Blocks bzw. Work-Groups parallel auf der GPU abgearbeitet, sondern mehrere Blocks bzw. Work-Groups zugleich, basierend auf der Anzahl der auf der GPU vorhandenen Streaming-Multiprozessoren. Jeder dieser Streaming-Multiprozessoren bearbeitet einen Block bzw. eine Work-Group. Uso mehr Streaming-Multiprozessoren eine GPU besitzt, umso mehr Blocks bzw. Work-Groups können parallel berechnet werden. Details hierzu können im „CUDA C Programming Guide“ [2] nachgelesen werden.

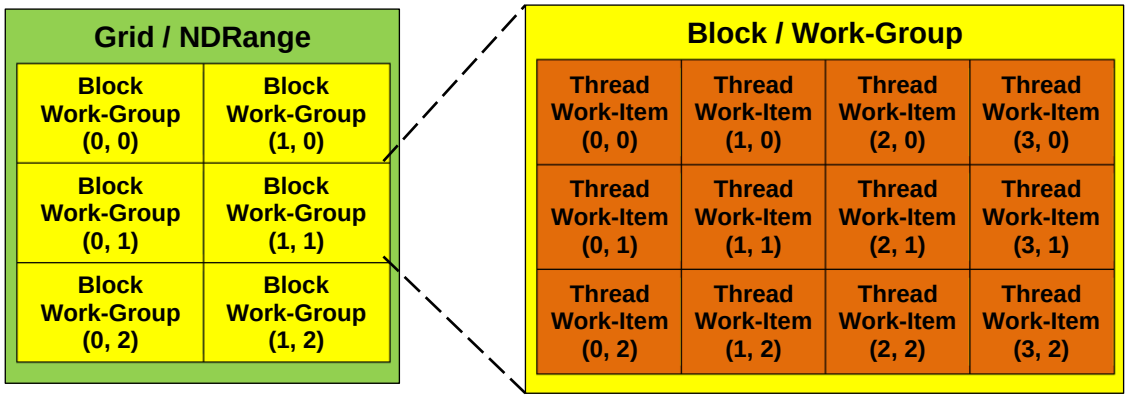


Abbildung 10: Ein Grid/NDRange von Blocks/Work-Groups aus Threads/Work-Items.

4 Pfadplanung und Pfadfindung

Bei der Pfadplanung bzw. Pfadfindung handelt es sich um eine der grundlegendsten Funktionen einer künstlichen Intelligenz. Deren Ursprung liegt in der Robotik und wurde auch für die Computerspieleindustrie sehr erfolgreich und ausgiebig adaptiert. Dabei liegt die Hauptaufgabe darin einen Pfad, oder aber auch mehrere Pfade, zwischen zwei Punkten in einer virtuellen Umwelt zu finden, egal ob diese Umwelt zwei- oder dreidimensional ist. Das Ziel der Pfadplanung ist es eine für das Spiel nachvollziehbare Navigation von computergesteuerten Charakteren, so genannte NPCs (Non-Player Character) zu erhalten. Dabei soll die Illusion von Intelligenz seitens der NPCs im Spiel erzeugt werden, welche mehr oder weniger in der Lage sind in dieser Umwelt erfolgreich und nachvollziehbar zu navigieren.

In der Regel sind die Daten, mit denen die virtuelle Umwelt beschrieben wird, für solch eine Pfadplanung nicht geeignet um sie in Echtzeit durchzuführen. Der Detailgrad der Umwelt ist, vor allem in modernen 3D Spieletiteln, zu groß um sie in einer angemessenen Zeit für eine oft durchgeführte Pfadplanung verwenden zu können. Es benötigt einer vereinfachten Repräsentation der Umwelt, welche am besten vorberechnet wird, um die bestmögliche Performanz gewährleisten zu können. Diese Repräsentation soll explizit nur der KI und der Pfadplanung dienen und exakt für diese Zwecke optimiert sein. Für diese Aufgabe ist ein so genannter Graph am besten geeignet. Ein Graph entspricht einer Datenstruktur, die sehr gut für die Zwecke einer Agentennavigation geeignet ist. Er ist der Grundstein für jede Pfadplanung, wobei zu erwähnen ist, dass sich die später vorgestellten Algorithmen auf solch einem Graphen aufbauen und am effizientesten mit diesem arbeiten.

Informationen über Graphen und der zur Repräsentation geeigneten Datenstrukturen werden im nächsten Abschnitt präsentiert und genauer erläutert.

4.1 Graphen

Ein Graph besteht aus so genannten Knoten und Kanten. Ein Knoten repräsentiert eine Stelle bzw. eine erreichbare Position in der virtuellen Umwelt. Knoten im Graph können als Orientierungspunkte für die KI verstanden werden. Die Knoten sind verbunden mit so genannten Kanten. Eine Kante entspricht einer direkten Verbindung zwischen zwei Knoten. Diese Verbindung hat in der Regel eine Gewichtung, sodass jeder Kante eine gewisse „Attraktivität“ zugeschrieben werden kann. So kann z.B. der Abstand zum nächsten Knoten über diese Kante zur Gewichtung herangezogen werden sowie auch der Grad der zu überwindenden Steigung von Knoten A nach Knoten B. Es handelt sich dabei um eine Kostenfunktion, welche die Pfadplanung zur Berechnung heranzieht. Ein sehr anschauliches Beispiel eines Graphen ist in Abbildung 11 zu sehen.

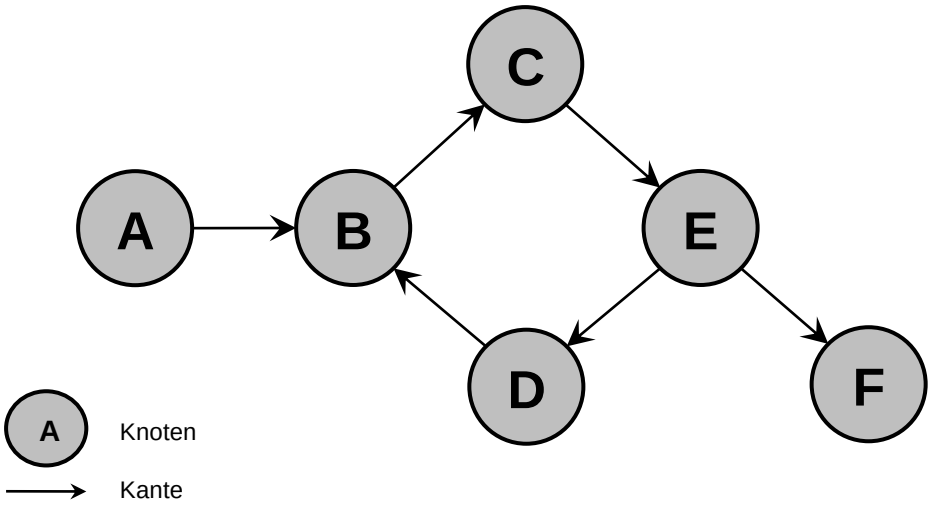


Abbildung 11: Ein Beispiel eines gerichteten Graphen mit sechs Knoten und Kanten.

Der Graph in Abbildung 11 ist relativ klein und besteht aus einer Menge von sechs Knoten $N = \{A, B, C, D, E, F\}$ und einer Menge von sechs Kanten zwischen den einzelnen Knoten $E = \{A \rightarrow B, B \rightarrow C, C \rightarrow E, E \rightarrow D, D \rightarrow B, E \rightarrow F\}$. Dieser Graph ist gerichtet, was bedeutet, dass die Kanten nur in eine Richtung passierbar sind, wie durch die Pfeile in Abbildung 11 zu erkennen ist. So ist es möglich von A nach B zu gelangen, aber nicht von B zurück nach A. Im Gegensatz hierzu sind die Knoten in einem ungerichteten Graphen immer in beide Richtungen erreichbar, sodass zu einer Kante von A nach B automatisch auch ein Kante von B nach A existiert.

Somit kann ein Graph formell als eine Menge von Knoten und Kanten definiert werden, zu sehen in der Formel in (3), welche so auch bei Bleiweiss [4] zu finden ist:

$$G = \{N, E\} \tag{3}$$

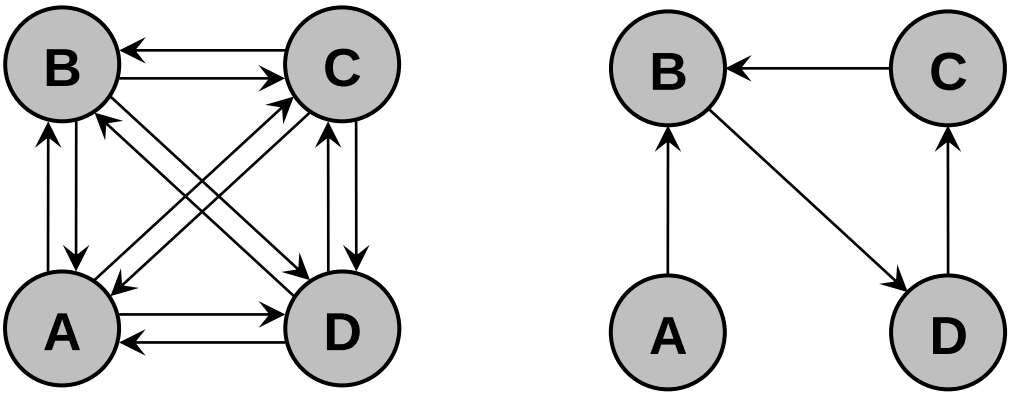
Wobei G ein Graph ist, definiert durch eine Menge bestehend aus den Mengen der Knoten N und der Kanten E.

Das Verhältnis zwischen der Anzahl an Elementen in N und E, mit anderen Worten das Verhältnis der Anzahl der Kanten zur Anzahl der Knoten im Graphen, ist eine besondere Bedeutung zuzuschreiben Da ein Knoten maximal nur mit jedem anderen Knoten im Graphen in beide Richtungen verbunden sein kann, und nicht mehr, ist die Anzahl der Kanten im Verhältnis zur Anzahl der Knoten beschränkt. Folgende Formel in (4) lässt sich aufgrund dieses Umstandes für die Beschränkung aufzeigen.

$$|E| \leq |N|^2 - |N| \tag{4}$$

Wobei $|E|$ der Anzahl der Kanten und $|N|$ der Anzahl der Knoten im Graphen entspricht.

Die Anzahl der Kanten $|E|$ kann nur kleiner oder gleich der quadrierten Anzahl an Knoten $|N|^2$ minus der Anzahl der Knoten selbst $|N|$ sein. Wenn $|E| = |N|^2 - |N|$ entspricht, dann sind alle Knoten mit allen anderen Knoten im Graphen in beide Richtungen verbunden. Dies geschieht in der Regel aber sehr selten. Für den Fall, dass der Wert von $|E|$ sehr nah an diesem Ergebnis herankommt, wird der Graph als „dicht“ (engl. *dense*) bezeichnet. Im anderen Fall, wenn $|E|$ sehr viel kleiner ist als $|N|^2 - |N|$, dann wird der Graph als „karg“ (engl. *sparse*) bezeichnet [4]. Beispiele für einen dichten und einen kargen Graphen mit jeweils vier Knoten, aber einmal „dicht“ mit allen möglichen zwölf Kanten ($12 = 4^2 - 4$) und einmal „karg“ mit nur vier Kanten, sind in Abbildung 12 dargestellt.



(a) (b)

Abbildung 12: (a) ein dichter Graph und (b) ein karger Graph mit jeweils vier Knoten.

Die Unterscheidung von „dichten“ und „kargen“ Graphen ist wichtig für die Auswahl der geeigneten Datenstruktur zum Speichern eines Graphen. Im Grunde existieren zwei Datenstrukturen, um die Topologie eines Graphen darzustellen:

1. Die Adjazenzmatrix
2. Die Adjazenzliste

Ersteres ist ein zweidimensionales Array bzw. eine Matrix, in der die Zeilen der Startknoten und die Spalten der Endknoten einer Kante entsprechen. In Abbildung 13 ist dies besser zu erkennen. Dort werden die Adjazenzmatrizen (a) und (b) der korrespondierenden Graphen (a) und (b) aus Abbildung 12 grafisch dargestellt. In den Zellen, in denen eine 1 steht, führt eine Kante von dem Knoten, der links in der Zeile steht, zu den Knoten, der oben in der Spalte steht. So können einfache und schnelle Nachbarschaftsabfragen durchgeführt und Kanten schnell gefunden werden. Überdies können statt einer 1 auch andere Werte, die einer Gewichtung der Kante entsprechen, gespeichert werden, wobei eine 0 weiterhin als Fehlen einer Kante zu verstehen ist.

Der Nachteil der Adjazenzmatrix ist aber schnell erkennbar, wenn die Matrix (b) des „kargen“ Graphen aus Abbildung 12 betrachtet wird. Diese besteht zum größten Teil nur aus dem Wert 0. Der Speicherverbrauch entspricht dem Quadrat der Anzahl der Knoten, in O-Notation $O(N^2)$. Wenn viele Kanten zu den Knoten existieren, so wie in der Matrix des „dichten“ Graphen (a), dann entsteht kein Nachteil. Die Repräsentation ist einfach und intuitiv. Im anderen Fall, wie beim „kargen“ Graphen (b), verbraucht diese Repräsentation genauso viel Speicher für wenig Kanten wie für viele, was im Grunde nicht sehr vorteilhaft ist und bei großen „kargen“ Graphen schnell zu einem verschwenderischen Speicherverbrauch führt [4].

	A	B	C	D
A	**1**	**1**	**1**	**1**
B	**1**	**1**	**1**	**1**
C	**1**	**1**	**1**	**1**
D	**1**	**1**	**1**	**1**

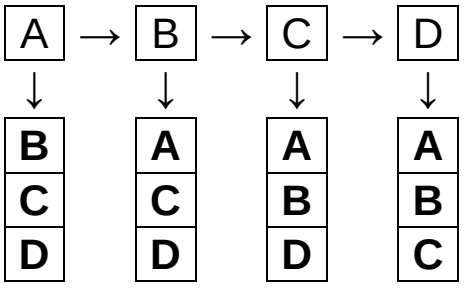
(a)

	A	B	C	D
A	0	**1**	0	0
B	0	0	0	**1**
C	0	**1**	0	0
D	0	0	**1**	0

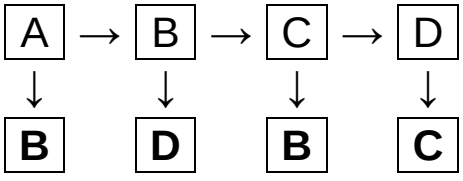
(b)

Abbildung 13: Adjazenzmatrizen (a) und (b) der Graphen (a) und (b) aus Abbildung 12.

Die zweite Repräsentation, die Adjazenzliste, ist in der Hinsicht des Speicherverbrauches um einiges kompakter. Die Adjazenzliste beinhaltet die Knoten in einer Liste und speichert sich zu jedem Knoten wiederum eine Liste mit dessen Nachbarn, also all die Knoten die durch diese Kante erreicht werden können. Diese Datenstruktur ist in Abbildung 14 für die Graphen (a) und (b) aus Abbildung 12 grafisch dargestellt. Der Speicherverbrauch besteht hierbei aus der Anzahl der Knoten plus der Anzahl der Kanten, in O-Notation $O(N + E)$. Somit braucht der “karge“ Graph (b) automatisch weniger Speicher, da er weniger Kanten besitzt als der „dichte“ Graph (a). Der Gewinn an Speicherplatz geht aber einher mit einer geringeren Effizienz beim Finden von Kanten und Erkennen von Verbindungen zwischen zwei Knoten [4].



(a)



(b)

Abbildung 14: Adjazenzlisten (a) und (b) der Graphen (a) und (b) aus Abbildung 12.

Nachdem die Datenstrukturen und Prinzipien eines Graphen erläutert wurden, werden nun im nächsten Kapitel Algorithmen vorgestellt, mit denen der Graph traversiert werden kann um Pfade zwischen zwei Knoten zu finden. Diese Algorithmen sind im Grunde miteinander verwandt und ähneln sich auch sehr, bis auf ein paar spezifische Eigenheiten und Unterschiede, welche nun in den folgenden Abschnitten genauer erläutert werden.

4.2 Pfadplanungsalgorithmen

4.2.1 Grundlagen

Alle Pfadplanungsalgorithmen folgen demselben Prinzip und durchsuchen die Knoten eines Graphen mit Hilfe der zuvor erwähnten Kanten. Knoten können nur über die Kanten die sie verbinden erreicht bzw. gefunden werden. Der Graph selbst entspricht, so wie auch bei LaValle [14] definiert, einem Zustandsraum, wo jeder Knoten im Graphen einem bestimmten Planungszustand des zu simulierenden Agenten entspricht. Die Kanten entsprechen den Übergängen zu diesen Zuständen, wobei diese durch die zuvor erwähnten Gewichtungen parametriert sind. Solch ein Planungszustand beinhaltet im Fall der Pfadplanung meist die Position des Agenten, kann aber auch Informationen über die Geschwindigkeit und Orientierung des Agenten beinhalten.

Die Gewichtungen der Kanten spielen für das Verhalten des Algorithmus und der Qualität der errechneten Pfade eine große Rolle. Die Art und Weise wie der Algorithmus die Gewichtungen der Kanten verarbeitet und bewertet beeinflusst den resultierenden Pfad und die daraus entstehenden „Entscheidungen“ der Agenten sehr. Darin liegt das größte Potenzial zum Feintuning des Algorithmus, um das vermeintlich „intelligenteste“, aber vor allem das nachvollziehbarste Verhalten von Agenten erreichen zu können.

Der Vorgang der Pfadplanung entspricht einer Traversierung eines Graphen. Dabei soll ein bestimmter Endknoten bzw. Endzustand erreicht werden, ausgehend von einem Startknoten bzw. Startzustand des Agenten. Dies entspricht der Definition nach der Basis eines so genannten Planungsproblems [4]. Zur genaueren Formulierung solch eines Planungsproblems sind die zuvor erwähnten Planungszustände eine sehr wichtige Komponente. Ein Planungsproblem kann, je nach Definition seiner Planungszustände, unterschiedliche Motivationen der Pfadsuche simulieren. So kann je nach Spielmechanik und Simulationsgrund ein anderer Ergebnispfad erzeugt werden. Zum Beispiel kann der kürzeste Pfad zwischen zwei Positionen in der Welt gesucht sein, wobei hier die Abstände zwischen den Punkten in die Gewichtung der Kanten einfließen. Oder der Agent versucht aus dem Sichtbereich eines oder mehrerer „feindlicher“ Spieler zu gelangen, wobei die Kanten zu den Knoten, die vom „Feind“ eingesehen werden können, sehr hoch gewichtet und somit „unattraktiv“ für den Agenten werden.

Im Grunde basiert jeder der später präsentierten Pfadplanungsalgorithmen auf demselben grundlegenden Aufbau oder auch Grundgerüst, welches in Abbildung 15 in Pseudocode gezeigt wird. Dieser Code entstand aus einem Pseudocode Beispiel von Bleiweiss [4] und zeigt sehr gut die einzelnen Grundzüge einer Traversierung eines Graphen zur Pfadsuche.

Die erste Eigenschaft eines Pfadplanungsalgorithmus ist die Verwendung einer so genannten „Priotity Queue“, zu sehen im Pseudocode in Abbildung 15 in Zeile 1 namens `Queue`. Diese entspricht einer Prioritätenliste, welche nach einer bestimmten Bewertungsfunktion für die beinhaltenden Knoten sortiert ist. Diese Bewertungsfunktion `f` ist in Abbildung 15 in Zeile 8 zu sehen und gibt einen Knoten `nachbar` zurück. Diese Funktion definiert das Verhalten des Algorithmus und ist unter den Algorithmen unterschiedlich.

Die Prioritätenliste besitzt zwei Operationen. Die erste namens `Queue.Insert()`, zu sehen im Pseudocode in Zeile 11, fügt einen Knoten entsprechend der Bewertungsfunktion `f` in die Liste ein. Eine weitere Operation ist die Rückgabe mit einer gleichzeitigen Entfernung des Knotens mit der besten Bewertungsfunktion aus der Prioritätenliste namens `Queue.Extract()`, zu sehen im Pseudocode in Zeile 4.

Die in der Liste befindlichen Knoten werden so lange abgearbeitet, bis die Liste leer wird (siehe `while`-Schleife in Zeile 3) oder ein Knoten gefunden wurde, der dem Zielknoten entspricht (siehe Zeilen 5 und 6). Im Fall wo die Liste leer wird, wurde kein Pfad gefunden und die Suche wird beendet, zu sehen im Pseudocode in Zeile 14. Der Algorithmus untersucht, wenn er einen Knoten aus der Prioritätenliste extrahiert, jeden über dessen Kanten zu erreichenden Nachbarn und bewertet diesen mit der Hilfe der zuvor erwähnten Bewertungsfunktion. Daraufhin fügt er auch den Nachbarknoten der Prioritätenliste hinzu.

```
1:    Queue.Insert(startknoten)
2:    startknoten als besucht markieren
3:    while Queue nicht leer do
4:       knoten ← Queue.Extract()
5:       if knoten ist gleich zielknoten then
6:          return PFAD_GEFUNDEN
7:       foreach kante von knoten do
8:          nachbar ← f(knoten,kante)
9:          if nachbar nicht besucht then
10:            Markiere nachbar als besucht
11:            Queue.Insert(nachbar)
12:         else
13:            Behandle schon besuchten Knoten nachbar
14:   return KEIN_PFAD_GEFUNDEN
```

Abbildung 15: Pseudocode des Grundgerüstes eines Pfadplanungsalgorithmus.

Des Weiteren ist es wichtig, dass Knoten, die schon einmal vom Algorithmus bearbeitet wurden, entsprechend markiert werden, damit schon besuchte Knoten speziell behandelt werden können, zu sehen in Zeile 13 des Pseudocodes in Abbildung 15. Knoten, die für eine Traversierung durchsucht werden, besitzen deswegen folgende „Markierungen“, wie auch bei Bleiweiss [4] erwähnt:

1. Unbesucht (*unvisited*)
2. Tot (*dead*)
3. Lebend (*alive*)

Knoten sind zu Beginn als „unbesucht“ markiert und werden, wenn sie der Algorithmus als Nachbar eines anderen Knoten findet, als „lebend“ markiert und in die „Priority Queue“ eingefügt. Der Knoten, welcher der Algorithmus aufgrund der Bewertungsfunktion wieder aus der „Priority Queue“ extrahiert, um dessen unbesuchte Nachbarn zu bearbeiten, wird als „tot“ markiert, da er weder in der „Priority Queue“ noch als unbesuchter Nachbar eines „lebenden“ Knoten gefunden werden kann.

4.2.2 Greedy Best First Search

Der „Greedy Best First Search“ Algorithmus (kurz: BFS) wurde von Judea Pearl, einem israelischen Informatiker und Philosoph, im Jahr 1984 in seinem Buch über Heuristiken [21] veröffentlicht. Der Algorithmus kommt von seinem Aufbau her dem in im letzten Kapitel vorgestellten Grundgerüst eines Pfadplanungsalgorithmus in Abbildung 15 am nächsten. Alle anderen Algorithmen, die noch in den folgenden Kapiteln vorgestellt werden, entfernen sich immer mehr von dieser Vorlage.

Ein Pseudocode des Algorithmus ist in Abbildung 16 zu erkennen. Die Prioritätenliste heißt in diesem Code `Open` und beinhaltet, wie der Name schon andeutet, alle Knoten, die noch „offen“ sind für Veränderungen. Der Startknoten wird, so wie auch im Pseudocode des Grundgerüsts, als erstes in die Prioritätenliste eingefügt und ist somit der erste „besuchte“ oder auch „offene“ Knoten. Die Operationen für das Einfügen und Extrahieren von Knoten mit der besten Bewertung besitzt diese Liste genauso wie im letzten Pseudocode-Beispiel.

Zusätzlich zur Open-Liste gibt es auch eine zweite Liste namens `Closed`, welche die Knoten beinhaltet, die sich nicht mehr ändern können, da die Nachbarn abgearbeitet wurden. Dies entspricht der im letzten Kapitel erwähnten Markierung von Knoten als „tot“. Die Closed-Liste ist keine „Priority Queue“, sondern eher als ein assoziativer Container zu verstehen. Diese Liste muss in der Lage sein, schon eingefügte Knoten schnell wieder finden zu können. Die Operation, zusätzlich zum Einfügen in die Liste, die am häufigsten gebraucht wird ist die zur Abfrage, ob sich ein Knoten schon in der Closed-Liste befindet, zu sehen im Pseudocode in Abbildung 16, Zeile 12.

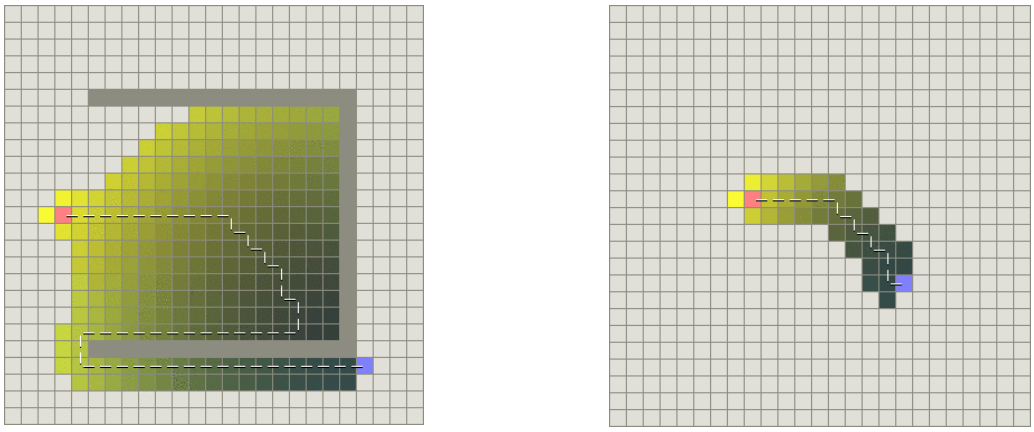
```
1:    f = []                  // Bewertungen für Knoten für Queue
2:    vorgängerVon = []       // Speichert Vorgänger von Knoten
3:    Open.Insert(startknoten)
4:    Closed = []
5:    while Open nicht leer do
6:       knoten ← Open.Extract()
7:       Closed.Insert(knoten)
8:       if knoten ist gleich zielknoten then
9:          return KonstruierePfad(vorgängerVon)
10:      foreach nachbar von knoten do
11:        bewertung ← Heuristik(zielknoten,nachbar)
12:        if nachbar nicht in Closed then
13:           f[nachbar] ← bewertung
14:           vorgängerVon[nachbar] ← knoten
15:           Open.Insert(nachbar)
16:        else if bewertung < f[nachbar] then
17:           f[nachbar] ← bewertung
18:           vorgängerVon[nachbar] ← knoten
19:   return KEIN_PFAD_GEFUNDEN
```

Abbildung 16: Pseudocode des „Greedy Best First Search“ Algorithmus von Pearl [21].

Die Bewertungsfunktion des BFS-Algorithmus entspricht einer zuvor definierten Heuristik, die einen Wert aus dem Zielknoten und dem gerade betrachteten Nachbarknoten errechnet. Dies ist im Pseudocode in Zeile 11 zu sehen. In der Regel entspricht dieser Heuristik einer Abstandabschätzung zwischen dem Ziel und dem Knoten, basierend auf den in den Knoten gespeicherten Weltkoordinaten. Aus diesem Grund „giert“ dieser Algorithmus (daher auch der Name „greedy“, englisch für gieren oder gierig) immer weiter in Richtung Ziel und wählt mit der Bewertung der Heuristik als nächsten Knoten immer jenen aus, der am nächsten am Ziel ist.

Durch das Betrachten von zusätzlichen Informationen und Verwendung von Wissen der Welt, die durch eine Heuristik bewertet wird, zählt dieser Algorithmus zu den informierten Suchalgorithmen. Das durch die Heuristik bestimmte Suchmuster scheint anfänglich sehr logisch und erfolgversprechend zu sein, hat aber eine große Schwäche. Wenn nämlich zwischen dem direkten Weg zum Ziel, z.B. über die Luftlinie, ein Hindernis ist, sodass dieses nur über Umwege über andere, vom Ziel weiter entfernte Knoten, erreicht werden kann, werden trotzdem die Knoten näher beim Ziel bevorzugt, auch wenn es von dort aus keine Kanten zum Ziel gibt. Eine Darstellung dieses Verhaltens ist in Abbildung 17 (a) zu erkennen, wo die einzelnen Felder den Knoten des Graphen entsprechen und die Nachbarn eines Knotens die vier umgebenden horizontalen und vertikalen Felder sind. Der errechnete Pfad, dargestellt mit einer grauen, gestrichelten Linie, ist eindeutig nicht optimal und entspricht nicht wie gewünscht dem kürzesten Pfad zwischen dem Start, dem roten

Feld, und dem Ziel, dem blauen Feld. Die gelb bis grau verlaufenden Felder entsprechen den besuchten Knoten des Algorithmus. Umso grauer bzw. dunkler ein Knoten ist, desto besser ist sein heuristischer Wert basierend auf den berechneten Restabstand zum Zielknoten. Somit erzeugt der BFS-Algorithmus in komplizierteren Situationen wie in (a), wo geographische Nähe zum Ziel nicht mit einer direkten Verbindung zu jenem einhergehen muss, ein schlechtes, nicht optimales Ergebnis eines Pfades. Im Gegensetz hierzu ist das Szenario in (b) sehr leicht und schnell zu lösen, unter Betrachtung der wenigen, immer in die Richtung des Ziels „gierenden“ besuchten Knoten.



(a) (b)

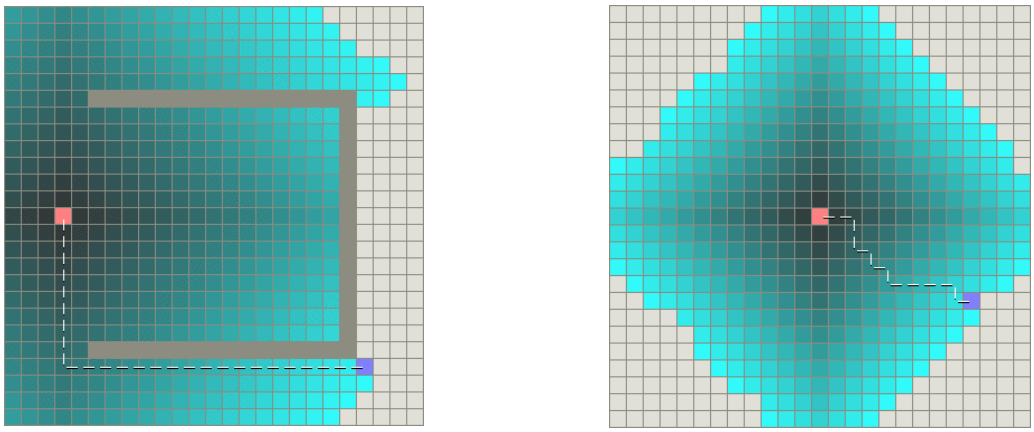
Abbildung 17: (a) Worst Case und (b) Best Case Szenario für den BFS-Algorithmus.

Die Listen im Pseudocode in Zeile 1 und 2 namens `f` und `vorgängerVon` speichern die heuristischen Werte der Knoten für die Prioritätsliste `Open` sowie die Vorgänger der Knoten. Letzteres ist wichtig, um den gefundenen Pfad mit Hilfe der Vorgänger, die während der Suche für die Knoten gesetzt werden, den finalen Pfad zurückgeben zu können. Dies geschieht, wenn der momentan gewählte Knoten dem Zielknoten entspricht, was im Pseudocode in den Zeilen 8 und 9 zu sehen ist. Mit dem Aufruf der Funktion `KonstruierePfad` wird der Pfad mit Hilfe der Liste `vorgängerVon` konstruiert und als Ergebnis dieser Suche zurückgegeben.

Die Behandlung schon besuchter Knoten, welche im Pseudocode in Abbildung 15 vorkommt, läuft beim BFS-Algorithmus folgendermaßen ab: Der Nachbarknoten, der schon einmal von einem anderen Knoten aus gefunden wurde, wird wieder betrachtet und bewertet. Wenn aber dieses Mal die Bewertung besser ausfällt als beim letzten Mal, also die neue Bewertung kleiner ist als der alte in `f` gespeicherte Wert für diesen Knoten, dann wird die neue Bewertung zu diesem Knoten gespeichert und der gerade besuchte Knoten als dessen Vorgänger gemerkt. Dies geschieht im Pseudocode in Abbildung 16 im `if-else`-Block von Zeile 12 bis 18.

4.2.3 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus wurde nach seinem Erfinder, dem niederländischen Informatiker Edsger W. Dijkstra, benannt. Veröffentlicht wurde sein Algorithmus zur Pfadfindung in einem Graphen in Numerische Mathematik 1 im Jahr 1959 [22]. Dieser Algorithmus ist im Gegensatz zum „Greedy Best First Search“ Algorithmus aus dem vorigen Kapitel in der Lage, immer den besten Weg zwischen zwei Knoten zu finden. Dies ist in Abbildung 18 (a) und (b) zu sehen, welche dieselben Szenarien zeigen wie die in Abbildung 17. Es ist zu erkennen, dass der Dijkstra-Algorithmus, im Gegensatz zum BFS-Algorithmus in der Lage ist, im Szenario (a) einen eindeutig kürzeren und somit auch besseren Pfad zu finden. Im Gegenzug ist aber beim Dijkstra-Algorithmus in Abbildung 18 zu erkennen, dass dieser viel mehr Knoten besucht als der BFS-Algorithmus in Abbildung 17, erkennbar an den Türkis eingefärbten Feldern. Vor allem im Szenario (b) ist dies sehr stark erkennbar.



(a) (b)

Abbildung 18: Darstellung des Suchverhaltens des Dijkstra-Algorithmus.

Der Grund, warum der Dijkstra-Algorithmus immer den kürzesten Pfad zwischen zwei Knoten findet, liegt in seiner Bewertungsfunktion. Die „Priority Queue“ gibt beim Aufruf von `Queue.Extract()`, zu sehen im Pseudocode in Abbildung 19 in Zeile 10, immer den Knoten zurück, der sich am nächsten beim Startknoten befindet. Somit verhält sich dieser Algorithmus genau anders herum als der BFS-Algorithmus, der den Knoten extrahiert, der sich am nächsten beim Ziel befindet. Somit ist das Verhalten des Dijkstra-Algorithmus ausgehend von seiner Bewertungsfunktion, am besten in Abbildung 18 (b) zu erkennen. Er breitet sich nämlich kreisförmig vom Startknoten, dem rot eingefärbten Feld, aus und durchsucht die umliegenden Felder bzw. Knoten, bis der Zielknoten, erkennbar als blaues Feld, gefunden wird.

Da der Dijkstra-Algorithmus nicht mit Hilfe einer Heuristik arbeitet wie der BFS-Algorithmus, sondern nur die Kosten der Kanten zu den einzelnen Knoten im Graphen zur Bewertung heranzieht, zählt er zu den uninformierten Suchalgorithmen.

```
1:    // Kante = {kosten, vonKnoten, zuKnoten}
2:    f = []                    // Bewertung für Knoten für Queue
3:    vorgängerVon = []         // Speichert Vorgänger von Knoten
4:    foreach knoten in Graph do
5:       f[knoten] ← unendlich
6:       vorgängerVon[knoten] ← unbekannt
7:       Queue.Insert(knoten)
8:    f[startknoten] ← 0
9:    while Queue nicht leer do
10:      knoten ← Queue.Extract()
11:      if f[knoten] ist unendlich then
12:         break
13:      if knoten ist gleich zielknoten then
14:         return KonstruierePfad(vorgängerVon)
15:      foreach kante von knoten do
16:       nachbar ← kante.zuKnoten
17:       kosten ← f[knoten] + kante.kosten
18:       if kosten < f[nachbar] then
19:          f[nachbar] ← kosten
20:          vorgängerVon[nachbar] ← knoten
21:   return KEIN_PFAD_GEFUNDEN
```

Abbildung 19: Pseudocode des Algorithmus von Dijkstra [22].

Der gesamte Algorithmus ist in Form eines Pseudocodebeispiels in Abbildung 19 zu erkennen. Eine Tatsache, die sehr auffällig ist im Gegensatz zum BFS-Algorithmus, ist die `foreach`-Schleife von Zeile 4 bis 7 vor Beginn des Algorithmus in der `while`-Schleife ab Zeile 9. In dieser `foreach`-Schleife werden die einzelnen Variablen für die Suche für jeden Knoten im Graphen initialisiert.

Es wird jedem Knoten im Graphen eine Startbewertung `unendlich` zugewiesen, zu sehen in Zeile 5. Nur die Kosten vom Startknoten werden mit 0 initialisiert, zu sehen in Zeile 8. Die dafür verwendete Liste namens `f` entspricht der gleichnamigen Liste des BFS-Algorithmus, zu finden im Pseudocode in Abbildung 16. Im Unterschied zum BFS-Algorithmus speichert der Dijkstra in dieser Liste nicht den heuristischen Wert zu Knoten sondern die aufzuwendenden Kosten zum Erreichen eines Knotens ausgehend vom Start.

Zusätzlich werden zu jedem Knoten im Graphen dessen Vorgänger in einer Liste namens `vorgängerVon` gespeichert, so wie auch schon beim BFS-Algorithmus. Diese wird auch in der `foreach`-Schleife initialisiert, indem jedem Knoten im Graphen der Vorgänger dem Wert `unbekannt` zugewiesen wird.

Außerdem werden alle Knoten von Anfang an der „Priority Queue“ zugefügt und im Laufe des Algorithmus, wenn sie extrahiert werden, wieder aus dieser entfernt. Somit wird die Closed-Liste, so wie sie im BFS-Algorithmus zu finden ist, nicht benötigt, da der Dijkstra-Algorithmus implizit einen schon besuchten Knoten nicht noch einmal finden kann, da er immer den Knoten zuerst extrahiert, der am nächsten beim Startknoten ist. Es kann somit nicht noch ein Knoten mit einer geringeren Distanz zum Start extrahiert werden als in vorhergehenden Durchläufen. Der Abstand zum Startknoten erhöht sich immer stetig mit der fortschreitenden Suche und alle Knoten, die extrahiert werden, wurden schon einmal als Nachbar eines anderen, zuvor extrahierten Knotens überprüft und bewertet. Durch diese Tatsache können einige Vereinfachungen im Algorithmus erzielt werden, wie eben die zuvor erwähnte Weglassung der Closed-Liste.

Der Dijkstra-Algorithmus bewertet, so wie der BFS-Algorithmus, nachdem ein Knoten extrahiert wurde, dessen Nachbarn. Die Bewertungsfunktion des Dijkstra-Algorithmus ist in Zeile 17 zu erkennen. Hierbei werden die Kosten des gerade extrahierten Knotens `f[knoten]` mit den Kosten der Kante, die zum gerade überprüften Nachbarn führt, `kante.kosten` addiert. Dies ist der errechnete Wert des gerade überprüften Nachbarn ausgehend vom Knoten `knoten`. Falls ein anderer Vorgänger dieses Nachbarknotens zuvor schon einen Weg zu diesem gefunden hatte, dieser aber schlechter ist, dann werden einfach die neuen Kosten und der neue Vorgänger des gerade überprüften Nachbarn in den Listen `f` und `vorgängerVon` gespeichert. Falls dies nicht der Fall ist, passiert gar nichts und die in vorigen Durchläufen zugewiesenen Werte werden beibehalten.

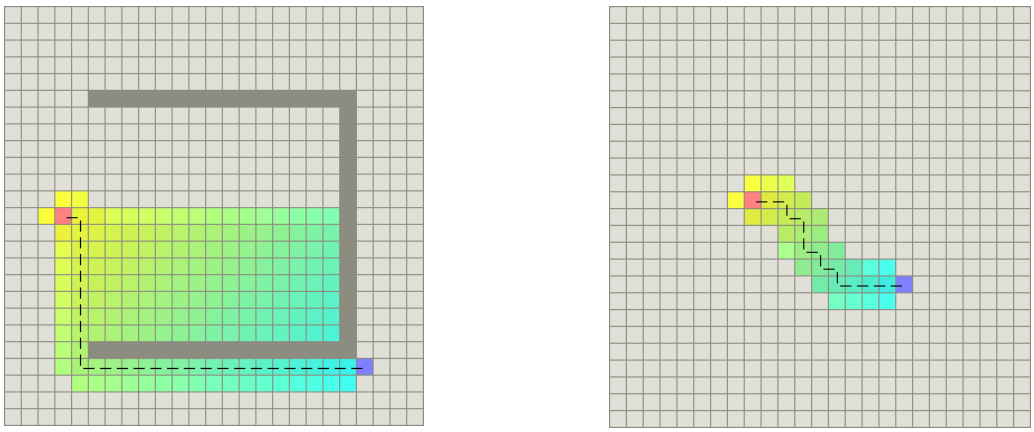
Die Abbruchbedingung ist wie zuvor beim BFS-Algorithmus dieselbe: Falls der Zielknoten gefunden wird, zu sehen im Pseudocode in der `if`-Abfrage in Zeile 13, wird mit der Funktion `KonstruierePfad` der endgültige Pfad mit Hilfe der gespeicherten Vorgänger konstruiert und als Ergebnis zurückgegeben.

Eine neue, zusätzliche Abbruchbedingung, zu sehen in Zeile 11 im Pseudocode, entsteht durch die Tatsache, dass die „Priority Queue“ von Anfang an alle Knoten mit schon zuvor initialisierten Werten dieser in der Liste `f` mit `unendlich` beinhaltet. Wenn der Wert des gerade extrahierten Knotens diesem Initialisierungswert entspricht, dann existieren keine Knoten mehr in der „Priority Queue“ mit einem geringeren Wert als `unendlich`. Dies impliziert, dass alle übrigen Knoten in der „Priority Queue“ nicht als Nachbarn eines schon zuvor extrahierten Knotens entdeckt wurden, denn dann hätten sie einen zulässigen Wert zwischen 0 und `unendlich`. Dadurch sind alle übrigen Knoten, darunter auch der Zielknoten, nicht vom Start aus erreichbar und der Algorithmus kann sofort abgebrochen werden, weil kein Pfad zwischen Start und Ziel existiert.

4.2.4 A*-Algorithmus

Der A*-Algorithmus (gesprochen A-Stern) wurde im Jahr 1968 von P. E. Hart, N. J. Nilsson und B. Raphael in einem Paper [23] veröffentlicht und ist bis dato einer der beliebtesten und meist eingesetzten Pfadplanungsalgorithmen. Der Grund hierfür ist in Abbildung 20 in den Szenarien (a) und (b) zu erkennen, die den Szenarien des BFS-Algorithmus in Abbildung 17 und des Dijkstra-Algorithmus in Abbildung 18 entsprechen. Es ist zu erkennen, dass der A*-Algorithmus, genauso wie der Dijkstra-Algorithmus und im Gegensatz zum BFS-Algorithmus, für das Szenario (a) einen optimalen Pfad findet. Dabei durchsucht er viel weniger Knoten als der Dijkstra-Algorithmus, erkennbar an den farbigen Feldern in Abbildung 20 bzw. Abbildung 18. Im Szenario (b) hingegen hat der A*-Algorithmus die Stärke des BFS-Algorithmus und „giert“ regelrecht Richtung Ziel, was der Dijkstra-Algorithmus, zu sehen in Abbildung 18 (b), gar nicht tut und ungleich mehr Knoten bzw. Felder durchsucht als der A* oder BFS, bevor er den Zielknoten findet.

Der Grund für das bessere Suchverhalten des A*-Algorithmus beruht auf der Verwendung und Mischung der Stärken des BFS und des Dijkstra-Algorithmus. Der A*-Algorithmus benutzt eine Heuristik, genauso wie der BFS-Algorithmus, um den Abstand zum Ziel abzuschätzen. Genauso aber verwendet dieser die schon aufgewendeten Kosten vom Startknoten aus zum gerade besuchten Knoten als Teil seiner Bewertungsfunktion. Durch diese Mischung entsteht ein hybrider Pfadplanungsalgorithmus, welcher im Gegensatz zum BFS-Algorithmus immer einen optimalen Pfad findet und dabei weniger Knoten besucht als der Dijkstra-Algorithmus, der auch immer optimale Pfade findet, dafür aber eine längere Zeit benötigt, weil er in der Regel mehr Knoten durchsucht als der A*. Somit verhält sich der A*-Algorithmus im Szenario (b) ca. so wie der BFS-Algorithmus. Im Szenario (a) hingegen ist eindeutig die Mischung von Heuristik und Kostenschätzung vom Startknoten aus zu erkennen.



(a) (b)

Abbildung 20: Darstellung des Suchverhaltens des A*-Algorithmus.

Das rote Feld entspricht wieder dem Startknoten und das blaue dem Zielknoten. Die besuchten Knoten haben eine Farbe verlaufend zwischen Gelb und Türkis. Umso gelber ein Feld ist, umso geringer und somit auch besser ist der Wert der aufgebrachten Wegkosten vom Start aus zu diesem Knoten. Umso türkiser das Feld wird, umso besser ist der heuristische Wert dieses Feldes und der Abstand zum Ziel. Umso mehr sich der Algorithmus dem Zielknoten nähert, umso türkiser werden die Felder und umso weniger gelb sind sie, da sich die Suche immer weiter vom Startknoten entfernt. So wählt der Algorithmus beim Extrahieren der Knoten aus der „Priority Queue“ immer den Knoten mit dem besten Gleichgewicht zwischen Nähe zum Ziel und Nähe zum Start.

Das Verhalten des A*-Algorithmus kann im Pseudocode in Abbildung 21 erkannt werden. Der Aufbau basiert immer noch auf dem Grundgerüst, auf dem auch die zuvor behandelten Algorithmen aufbauen, und besteht im Grunde aus einer Mischung von BFS und Dijkstra-Algorithmus. Erstens besitzt dieser Algorithmus eine Liste namens `kostenVonStartZu` zum Speichern der aufzuwendenden Kosten vom Start aus zu einem bestimmten Knoten, so wie beim Dijkstra-Algorithmus für dessen Bewertungsfunktion gespeichert in `f`. Die Bewertungen der Knoten befinden sich beim A* ebenfalls in der Liste `f` und werden aus den Kosten der gerade verwendeten Kante zum Nachbar plus dem Abstand vom Start aus in `kostenVonStartZu` plus dem heuristischen Wert des Nachbars, also dem geschätzten Abstand zum Zielknoten, so wie beim BFS-Algorithmus, berechnet. Diese Operation ist im Pseudocode in Abbildung 21 in Zeile 20 zu sehen.

Der A*-Algorithmus besitzt, genauso wie der BFS, eine „Priority Queue“ namens `Open` und eine Liste namens `Closed`. Der Startknoten wird als erstes in die „Priority Queue“ eingefügt und in `f` der Wert des Startknotens, so wie beim Dijkstra-Algorithmus, auf 0 gesetzt. Ein Knoten, der aus der „Priority Queue“ mittels der Funktion `Open.Extract()` extrahiert wird, wird in die Closed-Liste eingefügt, so wie es auch beim Pseudocode des BFS in Abbildung 16, Zeile 7 getan wird.

Der A*-Algorithmus bewertet, so wie auch die vorhergehenden Algorithmen, die Nachbarn eines extrahierten Knotens. Dabei wird zuerst der Abstand vom Startknoten aus mit Hilfe des gerade extrahierten Knotens und den Kosten der benutzten Kante errechnet. Die ist in Zeile 15 im Pseudocode zu erkennen und ist ident mit der Operation des Dijkstra-Algorithmus, zu sehen in dessen Pseudocode in Abbildung 19, Zeile 17. Dieser Wert wird aber nun beim A*-Algorithmus für eine `if`–Abfrage verwendet, welche in den Zeilen 16, 17 und 18 in Abbildung 21 zu sehen ist. Diese fragt ab, ob der gerade betrachtete Nachbarknoten nicht schon in der Closed-Liste ist. Wenn doch, dann wurde noch kein endgültig kürzester Pfad zu diesem Knoten entdeckt und ein anderer Pfad könnte kürzer sein. Deswegen wird als nächstes auch betrachtet, ob die gerade berechneten Kosten zu diesem Knoten kleiner sind als ein vielleicht schon zuvor gespeicherter Wert dieses Knotens in der Liste `kostenVonStartZu`.

```
1:    // Kante = {kosten, vonKnoten, zuKnoten}
2:    kostenVonStartZu = []     // Abstände vom Startknoten aus
3:    f = []                    // Bewertung für Knoten für Queue
4:    vorgängerVon = []         // Speichert Vorgänger von Knoten
5:    f[startknoten] ← 0
6:    Open.Insert(startknoten)
7:    Closed = []
8:    while Open nicht leer do
9:       knoten ← Open.Extract()
10:      Closed.Insert(knoten)
11:      if knoten ist gleich zielknoten then
12:         return KonstruierePfad(vorgängerVon)
13:      foreach kante von knoten do
14:         nachbar ← kante.zuKnoten
15:         kosten ← kostenVonStartZu[knoten] + kante.kosten
16:         if nachbar nicht in Closed or
17:         kosten < kostenVonStartZu[nachbar] and
18:         nachbar nicht in Open then
19:            kostenVonStartZu[nachbar] ← kosten
20:            f[nachbar] ← kosten + Heuristik(nachbar,zielknoten)
21:            vorgänger[nachbar] ← knoten
22:            Open.Insert(nachbar)
23:   return KEIN_PFAD_GEFUNDEN
```

Abbildung 21: Pseudocode des A*-Algorithmus von Hart et al. [23].

Ist die `if`-Abfrage erfolgreich, dann wird die Bewertung, wie schon zuvor erwähnt aus Abstand zum Startknoten und Heuristik, berechnet und in die Liste `f` gespeichert, welche wieder für das Extrahieren von Knoten aus der „Priority Queue“ verwendet wird. Außerdem wird der Wert von `kosten` für diesen Knoten in `kostenVonStartZu` zugewiesen, da er ja aufgrund der zuvor erwähnten `if`-Abfrage kleiner und somit besser ist. Zudem wird der im Moment extrahierte Knoten als Vorgänger des gerade betrachteten Nachbarn in der Liste `vorgängerVon` gemerkt und der Nachbarknoten in die „Priority Queue“ eingefügt.

Die Abbruchbedingung ist, so wie beim BFS und Dijkstra-Algorithmus, wenn der gerade extrahierte Knoten dem Zielknoten entspricht. Dann wird wieder mit Hilfe der Funktion `KonstruierePfad` mittels der gespeicherten Vorgänger in der Liste `vorgängerVon` der Pfad konstruiert und als Ergebnis der Pfadplanung zurückgegeben. Dieser Vorgang ist im Pseudocode in Abbildung 21 in Zeile 11 und 12 zu sehen.

4.3 Zusammenfassung der Algorithmen

Im vorhergehenden Kapitel über Pfadplanungsalgorithmen wurden die drei Algorithmen „Greedy Best Frist Search“ (kurz: BFS), Dijkstra und A* vorgestellt. Deren Suchverhalten wurden erläutert und untereinander verglichen, sowie deren Eigenschaften und Prinzipien erläutert und vorgestellt.

Im nun folgenden Abschnitt werden diese Algorithmen übersichtlich zusammengefasst und deren Stärken und Schwächen bzw. Vor- und Nachteile nochmals kurz erläutert. Eine kleine Übersicht in tabellarischer Form ist in Tabelle 3 zu sehen, welche von einer Abbildung von Bleiweiss [4] inspiriert wurde. Diese ist in Spalten unterteilt, welche folgende Eigenschaften beschreiben, welche diese Algorithmen besitzen, dargestellt durch grüne Häkchen, oder auch nicht, dargestellt durch rote Kreuze:

1. Start: aufgewandte Kosten vom Start bis zum Knoten werden berücksichtigt
2. Ziel: geschätzter Abstand vom Knoten bis zum Ziel wird berücksichtigt
3. Heuristik: Der Algorithmus verwendet eine Heuristik
4. Informiert: Der Algorithmus zählt zu den informierten Suchalgorithmen
5. Optimal: Gibt an, ob der Algorithmus immer optimale Pfadergebnisse liefert
6. Laufzeit: Beschreibung des Laufzeitverhaltens der Algorithmen in Worten

	Start	Ziel	Heuristik	Informiert	Optimal	Laufzeit
BFS	✗	✓	✓	✓	✗	mittel
Dijkstra	✓	✗	✗	✗	✓	langsam
A*	✓	✓	✓	✓	✓ (1)	schnell

(1) Voraussetzung: eine akzeptable Heuristik

Tabelle 3: Übersicht über die Eigenschaften der Pfadplanungsalgorithmen.

Der BFS-Algorithmus verwendet den geschätzten Abstand zum Ziel als seine Bewertungsfunktion. Hierfür verwendet er eine Heuristik, die den restlichen Abstand zum Ziel schätzt. Zur Schätzung verwendet dieser Algorithmus die Weltkoordinaten der Knoten, wodurch sich dieser zu den informierten Suchalgorithmen zählen lässt. Beim BFS-Algorithmus ist zu erkennen, dass er nicht die aufgewandten Kosten vom Start aus berücksichtigt. Deswegen ist er der einzige Algorithmus, der auch nicht optimale Ergebnisse der Pfadplanung liefert. Dafür ist sein Laufzeitverhalten allgemein im mittleren Bereich zwischen Dijkstra und A* einzuordnen, wobei dies auf die Situation ankommt. Der BFS ist nämlich beim „Best-Case“-Szenario, zu sehen in Abbildung 17 (b), der schnellste dieser Algorithmen. Nur beim „Worst-Case“-Szenario in (a) verhält sich der BFS-Algorithmus im Unterschied zum A* langsamer bzw. schlechter, wie auch in den Darstellungen der Suchverhalten dieser Algorithmen in Abbildung 17 und Abbildung 20 gesehen werden kann.

Der Dijkstra-Algorithmus hingegen benutzt die aufgewandten Kosten vom Start aus für die Bewertungsfunktion und ist somit im Gegensatz zum BFS in der Lage, immer optimale Pfade zu berechnen. Dafür schätzt dieser Algorithmus nicht den Abstand zum Ziel mit Hilfe einer Heuristik ab, die ihn näher zum Ziel führen kann. Deswegen zählt dieser Algorithmus zu den uninformierten Suchalgorithmen, woraus ein Suchverhalten resultiert, dessen Laufzeitverhalten im Vergleich zum BFS oder A*-Algorithmus langsamer ist, da die Tendenz zum Traversieren von zusätzlichen Knoten im Graphen, die weit entfernt sind vom Ziel, sehr groß ist.

Zu guter Letzt ist der A*-Algorithmus eine Mischung aus BFS und Dijkstra-Algorithmus und verwendet sowohl die aufgewandten Kosten vom Start aus als auch eine Heuristik zum Abschätzen der restlichen Entfernung zum Ziel. Er ist somit auch ein informierter Suchalgorithmus der versucht, den Abstand zum Ziel zu verkürzen. Dabei benutzt dieser Algorithmus, wie zuvor schon erwähnt, gleichzeitig auch den Abstand vom Start aus für seine Bewertungsfunktion. So werden die Knoten, die sehr nahe beim Start sind und somit einen kurzen Pfad vom Start aus wahrscheinlich machen, auch wahrscheinlicher besucht. Er erzeugt immer optimale Pfadergebnisse, so wie der Dijkstra-Algorithmus, ist aber aufgrund der Heuristik schneller als dieser. Dabei ist aber wichtig zu beachten, dass eine nicht akzeptable Heuristik das Ergebnis nicht optimal werden lassen kann. Außerdem ist der A* in Situationen, in der der BFS-Algorithmus einen schlechten bzw. nicht optimalen Pfad zurückgibt, im Durchschnitt auch schneller als dieser. Somit ist A* im Vergleich zu Dijkstra und BFS der optimalste Pfadplanungsalgorithmus.

All diese Aussagen sind im Moment nur theoretische Aufstellungen und basieren aus Analysen und Erfahrungswerten der Algorithmen. Zum Beweis müssen diese Algorithmen erst implementiert und in mehreren Szenarien ausgeführt und verglichen werden. Dies wird im Laufe dieser Arbeit, zusätzlich zum Vergleich von CPU und GPU Implementierungen dieser Algorithmen in unterschiedlichen Technologien wie C++, CUDA und OpenCL, durchgeführt und die Ergebnisse mit den zuvor getätigten Aussagen verglichen. Bevor dies aber vonstattengeht und Testläufe durchgeführt werden, wird das Framework, mit deren Hilfe diese Tests durchgeführt und gemessen werden, im nächsten Kapitel detailliert erläutert und vorgestellt.

5 Das Framework

Das im Laufe dieser Arbeit entwickelte Framework zur Simulation und Darstellung von Pfadplanungsalgorithmen trägt den Namen „Masterthesis_A_Star“, dessen Vorstellung inklusive einer Bedienungsanleitung im Anhang A dieser Arbeit zu finden ist. Entwickelt wurde „Masterthesis_A_Star“ mit Microsoft Visual Studio 2008 unter Windows 7 64bit. Die für die Entwicklung des Frameworks verwendete Software sowie auch das Windows 7 Betriebssystem wurden von der Fachhochschule Technikum Wien bereitgestellt.

„Masterthesis_A_Star“ ist kompatibel mit Windows XP und aufwärts, 32bit und 64bit. Es stellt eine 3D Szene dar, in welche Agenten eine Pfadplanung zu einem zugewiesenen Ziel durchführen. Die Szene wird dargestellt mittels DirectX 9 von Microsoft sowie mit in HLSL geschriebenen Shadern der Version 2.0. Programmiert wurde dieses Framework in der Sprache C++. Als Ausgang diente ein leeres DXUT-Sample (DirectX Utility Toolkit) aus dem Microsoft DirectX SDK (Juni 2010) [17]. Es wurden auch Datenstrukturen der C++ STL (Standard Template Library) wie z.B. „vector“ oder „list“ sowie für Datei- und String-Operationen Stream-Objekte wie „ifstream“, „ofstream“ oder „stringstream“ verwendet.

Das Framework kann sowohl über die Programmzeile als auch über eine Konfigurations-datei „config.ini“ konfiguriert werden. Zum Parsen der Programmzeilenargumente bzw. der Konfigurationsdatei wurde von der Open Source C++ Bibliothek „Boost“ [24] das „Program Options“ Framework verwendet. Die Möglichkeiten zur Konfiguration des Frameworks sowie die Verwendung dieser inklusive eine Liste der verfügbaren Programmargumente ist in der Bedienungsanleitung im Anhang A in Tabelle 12 zu finden.

Die Pfadplanung ist in mehreren Technologien implementiert, welche über eben diese Konfigurationsmöglichkeiten eingestellt werden kann. Standardgemäß wird die CPU für die Berechnung der Pfadplanung verwendet, implementiert in C++. Aber es kann auch die GPU, also die Grafikkarte des Systems, für die Berechnung der Pfadplanung verwendet werden. Hierfür wurden zwei Technologien in das Framework integriert, nämlich CUDA [2] von Nvidia und der Open Source Standard für Computing namens OpenCL [3]. Details zu den Implementierungen der Pfadplanungsalgorithmen werden später im Kapitel 5.3 genauer erläutert und vorgestellt.

Zusätzlich zu den Algorithmen und der Technologie ist auch die Welt konfigurierbar, in der die Simulation durchgeführt wird. So können verschiedenste Szenarien in unterschiedlicher Größe und Agentenanzahl simuliert werden. Dabei können zufällige, quadratische Welten ohne Hindernisse erschaffen werden mit einer definierten Anzahl an Agenten, oder auch Szenarien aus so genannten „Szenario-Dateien“ geladen werden, welche einen genau vordefinierten Aufbau der Umwelt sowie die Ziele der einzelnen Agenten definiert. Informationen über die Verwendung sowie den Aufbau von „Szenario-Dateien“ sind in der Bedienungsanleitung im Anhang A „Szenario-Dateien“ zu finden.

Das wichtigste Feature dieses Frameworks ist aber das integrierte Benchmarking, mit dem die Laufzeiten der Algorithmen gemessen werden können. Dieses Benchmarking wird vom „Benchmark-Modus“ des Frameworks benutzt um die Laufzeiten zu protokollieren. Diese protokollierten Ergebnisse werden dann in eine CSV-Datei (Comma-Separated Values) gespeichert und später im Kapitel 6 vorgestellt und analysiert. Details zum „Benchmark-Modus“ sowie zum Export von CSV-Dateien werden ebenfalls in der Bedienungsanleitung des Frameworks im Anhang A „Der Benchmark-Modus“ genauer erläutert.

In den nächsten Abschnitten folgt nun eine Beschreibung der technischen Umsetzung der Pfadplanungsalgorithmen des Frameworks. Zuerst wird die Repräsentation des Graphen vorgestellt, sowie dessen Darstellung von Knoten und Nachbarn erläutert. Außerdem werden die im Framework verwendeten Heuristiken vorgestellt. Die Definition von Graph und Knoten kann im Kapitel 4 „Pfadplanung und Pfadfindung“ gefunden werden.

Zudem werden die Implementierungen der Pfadplanungsalgorithmen in den jeweiligen Technologien vorgestellt und beschrieben, für die CPU in C++ und OpenMP [1] und für die GPU mittels CUDA [2] und OpenCL [3]. Außerdem werden die eigenen Umsetzungen der für die Pfadplanung wichtigen Listen, darunter auch die Prioritätenliste „Priority-Queue“ und die „Closed-Liste“, wie sie schon bei den Pfadplanungsalgorithmen im Kapitel 4.2 beschrieben wurden, präsentiert und erläutert.

5.1 Repräsentation des Graphen

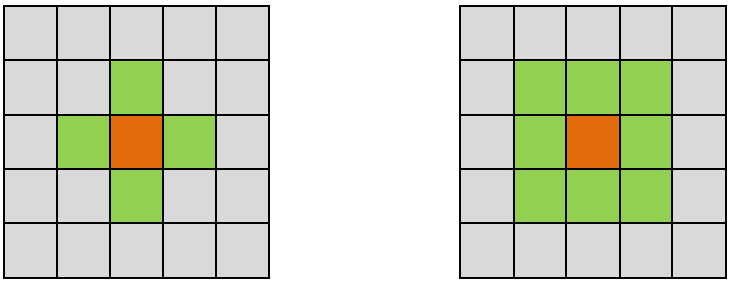
Der Graph für die Pfadplanungsalgorithmen des Frameworks entspricht einer zweidimensionalen Darstellung der zu simulierenden Umwelt in Form eines Rechtecks. Dieses besteht aus gleichmäßigen, quadratischen Flächen, wo jede dieser Flächen einen Knoten des Graphen repräsentiert und ein Raster bilden. Diese Repräsentation der Umwelt wurde in dieser Arbeit schon mal benutzt, und zwar für die bildliche Darstellung der Suchverhalten der Algorithmen in den Kapiteln 4.2.2 bis 4.2.4, in Abbildung 17, Abbildung 18 und Abbildung 20. Somit wird keine der im Kapitel 4.1 vorgestellten Repräsentationen für Graphen, wie eine Adjazenzmatrix oder eine Adjazenzliste, verwendet.

Diese Darstellung des Graphen eignet sich sehr gut für eine zweidimensionale Pfadplanung und kann technisch gesehen als ein zweidimensionales Array vorgestellt werden, in dem die gespeicherten Elemente des Arrays die Knoten repräsentieren. Die Knoten selbst besitzen einen Wert für deren Position innerhalb dieses Arrays (`x` und `y`) und einen Wert für dessen Gewichtung (`weight`). Dabei entspricht die Gewichtung des Knotens dem Wert der Kante zu diesem Knoten, egal von welchem Nachbarn aus zu diesem gelangt werden soll. Es gibt somit keine Kanten, sondern die Knoten selbst besitzen eine Gewichtung, die die Kanten zu ihnen allgemein repräsentieren. Dieser Gewichtungswert entspricht in der Regel innerhalb der Applikation dem Wert 1, oder für nicht begehbare Felder, also für Hindernisse welche die Agenten umgehen müssen, den Wert 0.

5.1.1 Nachbarschaften im Graphen

Die zweidimensionale und rasterähnliche Repräsentation des Graphen hat den Vorteil, dass Kanten zu Nachbarknoten schnell und unkompliziert ausgemacht werden können. Die Nachbarn eines Knotens sind einfach die Elemente des Arrays, die an dem Element angrenzen, das den Knoten selbst repräsentiert. Dabei sind alle Knoten mit einem Gewichtungswert von 0 als „nicht begehbar“ zu verstehen und können somit auch nicht über andere Nachbarknoten erreicht werden. Sie stellen die Hindernisse der Simulation dar, die die Agenten umgehen müssen.

Des Weiteren kann die Definition von Nachbarschaft in solch einer zweidimensionalen Rasterdarstellung des Graphen nicht hundertprozentig bestimmt werden, da diagonal angrenzende Knoten entweder als Nachbarn anerkannt werden oder eben nicht. Dadurch können zwei unterschiedliche Nachbarschaftsansichten definiert werden, welche auch als „Von Neumann“ und „Moore“-Nachbarschaft bekannt sind. Diese beiden Nachbarschaften können in Abbildung 22 gesehen werden, wobei die orangenen Felder den Knoten selbst repräsentieren und die Grünen die Nachbarn.



(a) (b)

Abbildung 22: (a) „Von Neumann“- und (b) „Moore“-Nachbarschaft in einem Raster.

Bei der „Von Neumann“-Nachbarschaft, zu sehen in Abbildung 22 (a), werden nur die vier umliegenden Felder oben, unten, links und rechts als Nachbarn angesehen. Bei der „Moore“-Nachbarschaft in (b) hingegen werden zusätzlich auch die vier diagonalen Knoten als Nachbarn angesehen und sind somit in Summe acht. Dabei ist zu beachten, dass die diagonalen Nachbarn bei der „Moore“-Nachbarschaft weiter weg vom Knoten in der Mitte sind und somit auch der Weg zu diesem länger wird. Da die Felder des Graphen alle gleich große Quadrate sind, kann die Entfernung zu den diagonalen Nachbarn einfach durch die Berechnung der Diagonale des Quadrates bestimmt werden. Dies geschieht indem der Gewichtungswert, der im Grunde immer die Entfernung darstellt, mit der Quadratwurzel von 2 ($\sqrt{2} \approx 1.4142136$) multipliziert wird.

5.1.2 Heuristiken

Für die einzelnen im vorherigen Abschnitt erwähnten Nachbarschaftsrepräsentationen „Von Neumann“ und „Moore“ werden jeweils unterschiedliche Heuristiken zur Bewertung von Knoten zum Zielknoten hin verwendet. Dabei verändern sich nicht nur die Berechnung der Nachbarn eines Knoten und die Bewegungen der Agenten, sondern genau deswegen auch die Heuristik zur Bewertung von Knoten der beiden Pfadplanungsalgorithmen BFS und A* (siehe Kapitel 4.2.2 bzw. 4.2.4).

Bei der „Von Neumann“-Nachbarschaft, in der die Agenten nicht in der Lage sind sich diagonal zwischen den Feldern des Rasters zu bewegen, wird mit Hilfe der so genannten „Manhattan Distanz“ die Felder bewertet. Diese definiert sich wie folgt, zu sehen in (5):

$$h = |n.x - n'.x| + |n.y - n'.y| \tag{5}$$

Wobei h der „Manhattan Distanz“ zwischen den beiden Knoten n und n' mit deren Indexwerten für x und y entspricht.

Diese Heuristik berechnet die „Manhattan Distanz“ zwischen zwei Knoten des Graphen, indem der absolute Abstandswert der Knoten für die x- und y-Achse berechnet und addiert wird. Dies entspricht dann der geschätzten Anzahl an restlichen Bewegungen zu diesen Knoten, vorausgesetzt es sind keine Hindernisse auf dem Weg oder Umwege nötig.

Im Unterschied dazu sind bei der „Moore“-Nachbarschaft diagonale Bewegungen der Agenten zwischen den Feldern des Rasters bzw. den Knoten des Graphen möglich. Somit wird auch eine andere Heuristik benötigt, die dies auch, im Gegensatz zur „Manhattan Distanz“, berücksichtigt. Für diesen Fall wird die so genannte „diagonale Distanz“ der Knoten berechnet, welche aus mehreren Teilberechnungen besteht. Zuerst wird die Anzahl der benötigten diagonalen Schritte berechnet, bevor nur mehr gerade Schritte zum Ziel ohne diagonale Bewegungen, wie bei der „Manhattan Distanz“, benötigt werden. Dies wird wie folgt in (6) berechnet:

$$h' = \min(|n.x - n'.x|, |n.y - n'.y|) \tag{6}$$

Wobei h' der Anzahl der diagonalen Schritte entspricht, welches wiederum dem Minimum min des Abstands der Knoten n und n' in x- oder y-Richtung entspricht.

Dieser Wert, sowie die „Manhattan Distanz“ der Knoten wie in (5) berechnet, wird nun weiterverwendet, um die „diagonale Distanz“ der Knoten wie folgt in (7) zu berechnen:

$$h'' = \sqrt{2}\, h' + (h - 2h') \tag{7}$$

Wobei h' dem berechneten Wert aus (6) und h der „Manhattan Distanz“ aus (5) entspricht.

In dieser Formel wird die Anzahl der benötigten diagonalen Schritte h' mit $\sqrt{2}$ multipliziert, da die diagonalen Schritte, so wie bei der „Moore“-Nachbarschaft erläutert, der Diagonalen der quadratischen Felder entsprechen, welche der Seitenlänge des Quadrats multipliziert mit der Quadratwurzel von 2 entsprechen. Die übriggebliebenen, geraden Schritte der „Manhattan Distanz“ ergeben sich dann durch die Subtraktion der doppelten Anzahl an diagonalen Schritten ($h - 2h'$), da jeder diagonale Schritt immer durch genau zwei gerade Schritte repräsentiert werden kann. Mit dieser Heuristik kann der Abstand zwischen Knoten sehr genau berechnet werden, wenn Agenten sich sowohl gerade als auch diagonal im Raster, welches den Graphen repräsentiert, bewegen können.

Eine akzeptable und zur Situation passende Heuristik ist wichtig für ein gutes Laufzeitverhalten eines Pfadplanungsalgorithmus wie z.B. dem BFS oder dem A* (siehe Kapitel 4.2.2 bzw. 4.2.4). Auch die verwendeten Datenstrukturen, welche für die Listen der Pfadplanung verwendet werden, tragen einen großen Anteil am Laufzeitverhalten des Algorithmus bei. Diese Listen und deren Operationen werden nun im nächsten Kapitel erläutert und deren Implementierungen im Framework präsentiert.

5.2 Datenstrukturen der Listen

Bei der ersten Liste bzw. Datenstruktur, welche von allen in dieser Arbeit im Kapitel 4.2 vorgestellten Algorithmen BFS, Dijkstra und A* benutzt werden, ist die so genannte Prioritätenliste, oder auch „Priority Queue“. Diese beinhaltet die noch zu expandierenden Knoten des Graphen und besitzt, so wie detaillierter im Kapitel 4.2 beschrieben, eine Operation zum Einfügen von Knoten in die Liste und eine Operation zum Extrahieren des Knotens aus dieser Liste mit dem besten Wert, berechnet von einer Bewertungsfunktion.

Durch diese beiden für die Prioritätenliste charakteristischen Operationen kann eine bestimmte, in der Informatik bekannte Datenstruktur ausgemacht werden, die diese Operationen sehr gut und performant umsetzt. Dabei handelt es sich um einen so genannten binären Heap, der auch bei der Implementierung von Pfadplanungsalgorithmen bei Bleiweiss [4] Verwendung findet.

Die zweite Datenstruktur zum Verwalten von Listen, ebenfalls zu finden in den Pseudocodes im Kapitel 4.2, wie z.B. die Closed-Liste und andere assoziative Listen wie `kostenVonStartZu` oder `f`, werden im Framework in Form eines „hashed Array“ implementiert. Dieses ist in der Lage, Werte zu einem bestimmten Graphen-Knoten zu speichern und auch schnell wieder zu finden.

Diese beiden für die Pfadplanungsalgorithmen wichtigen Datenstrukturen, die „Priority Queue“ und das „hashed Array“ werden in den nun folgenden Abschnitten erläutert und deren Nutzen und Vorteile im Rahmen der Pfadplanung beschrieben und analysiert.

5.2.1 Priority Queue

Die Prioritätenliste bzw. „Priority Queue“ wurde innerhalb des Frameworks als ein so genannter „binärer Min-Heap“ implementiert. Diese Datenstruktur ist im Grunde ein „Binärbaum“. „Binärbaum“ bedeutet hierbei, dass der Heap Knoten besitzt, welche jeweils maximal zwei Kinderknoten besitzen, die wiederum bis zu zwei Kinderknoten besitzen und so weiter. Eine bildliche Darstellung eines Heaps in solch einer Binärbaum-Struktur kann in Abbildung 23 (a) gesehen werden.

Der Heap ist aber nicht nur ein einfacher Binärbaum, sondern besitzt eine Eigenschaft die diese Datenstruktur für die Verwendung in einer „Priority Queue“ prädestiniert. Basierend auf der Tatsache, dass es sich bei dem zu verwendenden Heap um einen so genannten „Min-Heap“ handelt, definiert sich dieser in der Eigenschaft, dass alle Kinderknoten eines Knotens einen größeren Wert besitzen als der Knoten selbst. Somit besitzen alle Elternknoten kleinere Werte als deren bis zu zwei Kinderknoten. Der Nutzen dieser Eigenschaft für die „Priority Queue“ bzw. für den Pfadplanungsalgorithmus wird später bei der Erläuterung derer Operationen „Insert“ und „Extract“, zu sehen als Pseudocode in Abbildung 24 bzw. Abbildung 25, erläutert und ersichtlich.

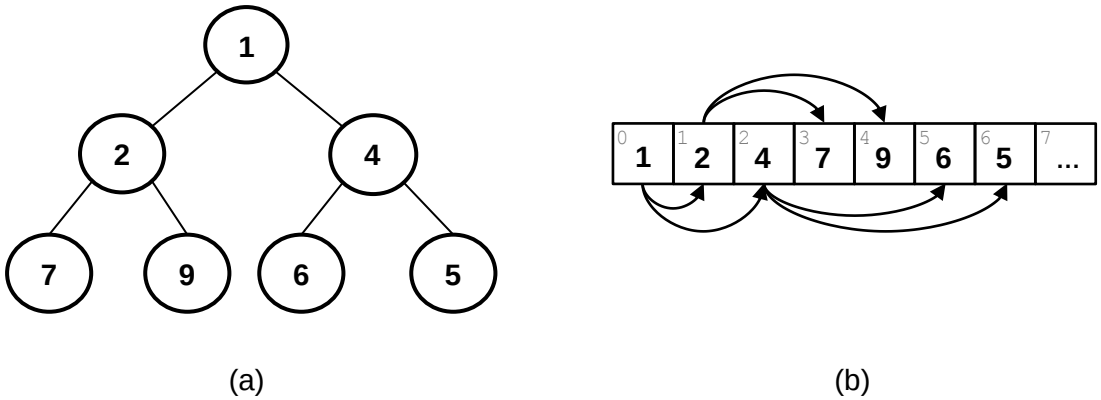
Unter dem „Wert“ eines Knotens, nach dem der Heap geordnet wird, ist hierbei der durch die Bewertungsfunktion des Algorithmus berechnete Wert für den Knoten zu verstehen. Diese werden innerhalb des Frameworks in einem Array gespeichert und auch verwaltet, sodass die Eigenschaft des Heaps immer gewährleistet wird. Dies ist innerhalb eines Arrays möglich, ohne Referenzen auf Eltern- bzw. Kinderknoten, unter Zuhilfenahme einfacher arithmetischer Funktionen auf die Indexwerte des Arrays zum Finden der Kinder eines Knotens. Bei diesen Funktionen handelt es sich, angenommen das erste Element im Array besitzt den Index 0, um folgende Funktionen (8) und (9) für den ersten (linken) bzw. zweiten (rechten) Kinderknoten:

$$left = 2i + 1 \tag{8}$$

$$right = 2i + 2 \tag{9}$$

Wobei i dem Index eines Knotens im Array der „Priority Queue“ und $left$ bzw. $right$ den jeweiligen Indices der Kinderknoten des Knotens bei Index i entsprechen.

So kann jeder Kinderknoten eines Knotens gefunden werden, indem der Index dieses Knotens mit 2 multipliziert wird ($2i$). Dadurch erhält man den Index genau vor den beiden Kinderknoten, deren Indices durch die Addition mit 1 bzw. 2 errechnet werden können. Eine Darstellung solch eines „Min-Heaps“ als Array, inklusive Pfeile zur Darstellung der Beziehung der Knoten zu derer Kinderknoten, kann in Abbildung 23 (b) gesehen werden, der dem Heap in Binärbaum-Struktur in Abbildung 23 (a) entspricht.



(a) (b)

Abbildung 23: Binärer Min-Heap dargestellt als Baumstruktur (a) und als Array (b).

Mit der Repräsentation des Heaps als ein Array und der in den Formeln (8) und (9) zu findenden Berechnung der Indices von Kinderknoten, kann die „Priority Queue“ bzw. der „Min-Heap“ sehr gut und performant, sowohl für die CPU als auch die GPU, implementiert werden. Dabei stehen die beiden Operationen „Insert“ und „Extract“ im Mittelpunkt und sind ausschlaggebend für eine performante Implementierung. Die Art und Weise wie diese im Framework umgesetzt sind, kann in den Pseudocodes der Funktionen „Insert“ in Abbildung 24 und „Extract“ in Abbildung 25 gesehen werden.

Das Array der „Priority Queue“, welches die zu expandierenden Knoten des Graphen beinhaltet, wird in den Pseudocodes der Funktionen „Insert“ und „Extract“ in Abbildung 24 und Abbildung 25 `pqArray` genannt. Dieses hält und verwaltet den Heap, so wie zuvor beschrieben und in Abbildung 23 (b) bildlich dargestellt. Die einzelnen Elemente im Array werden mittels nummerische Indexwerte adressiert. Bei den Elementen selbst handelt es sich um Knoten des Graphen, welche in den Pseudocodes vom Typ `Node` sind.

Das zweite Array `f`, welches in den Pseudocodes in Abbildung 24 und Abbildung 25 zu finden ist, beinhaltet hingegen die Werte der von der Bewertungsfunktion des Algorithmus bewerteten Knoten, im Framework vom Typ `float`. Die Elemente der „Priority Queue“ werden mit Hilfe der zu den jeweiligen Knoten gespeicherten Werte in `f` nach der zuvor erwähnten „Min-Heap“-Eigenschaft (Kinderknoten haben größeren Wert als Elternknoten) sortiert.

Dieses Array `f` ist so umgesetzt, dass es assoziativ ist und mit Hilfe von Knoten-Objekten vom Typs `Node`, welche in der „Priority Queue“ zu finden sind, darauf zugegriffen wird. Dies kann auch im Pseudocode in Abbildung 24 in Zeile 9 gesehen werden, wo der `float`-Wert `kosten` dem Array `f` für den `Node` namens `knoten` zugewiesen wird (`f[knoten] ← kosten`). Informationen zur Implementierung solch einer assoziativen Liste kann im Kapitel 5.2.2 „Hashed Array“ nachgelesen werden.

```
1:    // pqArray[]: Priority Queue mit Knoten in Heap-Order nach f[]
2:    // f[]: Bewertung für Knoten in Queue
3:    function Insert(Node knoten, float kosten)
4:       int i ← (pqArray.size)++
5:       while i > 0 and f[pqArray[i>>1]] > kosten do
6:          pqArray[i] ← pqArray[i>>1]
7:          i >>= 1
8:       pqArray[i] ← knoten
9:       f[knoten] ← kosten
10:   end
```

Abbildung 24: Pseudocode der Funktion „Insert“ der Priority Queue.

Die erste Funktion der Prioritätenliste namens „Insert“, die mit den beiden Listen `pqArray` und `f` arbeitet, ist im Pseudocode in Abbildung 24 zu sehen. Diese Einfüge-Operation in die „Priority Queue“ ist so auch im Framework umgesetzt und ist inspiriert von der Heap-Implementierung vorgestellt von Bleiweiss in seiner Arbeit [4] mit CUDA.

Diese fügt einen Knoten (`Node knoten`) in die „Priority Queue“, gespeichert in `pqArray`, ein und verwendet den Wert `kosten` zur Sortierung dieses Knotens in `pqArray` zur Gewährleistung der „Min-Heap“-Eigenschaft. Das Einfügen in die Liste geschieht zeitgleich mit dessen Sortierung und wird in der `while`-Schleife im Pseudocode von Zeile 5 bis 7 durchgeführt. Dabei wird in der Schleife solange mittels Verringerung des Index `i` durch Bit-Shifting nach rechts (`i>>1`), beginnend beim letzten Element, nach einem Elternelement in `pqArray` gesucht, dessen Wert nicht mehr größer ist als der für den einzufügenden Knoten in `kosten`, oder der Index `i` nicht mehr weiter geshiftet werden kann, weil dieser schon bis zum ersten Element mit dem Index 0 nach rechts bit-geshiftet wurde. Tritt einer dieser beiden Situationen ein, dann wird der Knoten `knoten` in die „Priority Queue“ `pqArray` beim momentanen Index `i` eingefügt und der Wert zu diesem Knoten in `kosten` in `f` gespeichert, zu sehen im Pseudocode in den Zeilen 8 und 9.

```
1:    // pqArray[]: Priority Queue mit Knoten in Heap-Order nach f[]
2:    // f[]: Bewertung für Knoten in Queue
3:    function Extract()
4:       Node knoten ← unbekannt
5:       if pqArray.size >= 1 then
6:          knoten ← pqArray[0]
7:          pqArray[0] ← pqArray[--(pqArray.size)]
8:          Heapify()
9:       return knoten
10:   end
```

Abbildung 25: Pseudocode der Funktion „Extract“ der Priority Queue.

Die zweite sehr häufig verwendete Operation der „Priority Queue“ zum Extrahieren des Knotens aus dieser mit dem besten Wert namens „Extract“ kann im Pseudocode in Abbildung 25 gesehen werden und wurde auf diese Art auch bei Bleiweiss [4] für seine Heap-Implementierung in CUDA verwendet. Sie nutzt, genauso wie die Funktion „Insert“, die Eigenschaften des Heaps und wird im Pseudocode in Abbildung 25 dargestellt.

Bei der Operation „Extract“ wird der erste Knoten im Index 0 der „Priority Queue“, der aufgrund der „Min-Heap“-Eigenschaft automatisch auch der mit dem kleinsten bzw. besten Wert in `f` ist, aus der Liste `pqArray` entfernt und stattdessen in der `Node`-Variable `knoten` gespeichert, so wie in Zeile 6 im Pseudocode zu sehen. An die Stelle des nun fehlenden ersten Knotens wird, so wie in Zeile 7 im Pseudocode in Abbildung 25 zu sehen, der hinterste bzw. letzte Knoten von `pqArray` eingefügt. Durch diese Umsortierung ist die Eigenschaft des Heaps natürlich nicht mehr gegeben und muss mit der Funktion `Heapify` wieder in eine korrekte Ordnung, auch „Heap-Order“ genannt, gebracht werden.

Die Funktion `Heapify` implementiert hierbei eine Heap-Sortierungsfunktion, die in der Lage ist, ein beliebiges ungeordnetes Array so umzusortieren, dass die Heap-Eigenschaft bzw. die so genannte „Heap-Order“ des Arrays wieder gegeben ist. Informationen zu dieser Funktion sowie zu dessen Implementierung innerhalb des Frameworks kann im Anhang B dieser Arbeit detailliert nachgelesen werden.

Die Implementierung der „Priority Queue“ als Heap, mit den in den Pseudocodes in Abbildung 24 und Abbildung 25 beschriebenen Operationen „Insert“ und „Extract“, hat, so wie auch bei Bleiweiss [4] beschrieben und auch gemessen, performancetechnisch einen Vorteil gegenüber anderer, naiver Implementierungen mit einer normal sortierten Liste. Bei der Implementierung mit Heap steigt die Laufzeit des Algorithmus im Durchschnitt logarithmisch mit der Anzahl an Elementen in der Liste an, in O-Notation ausgedrückt $O(logN)$, wobei N der Anzahl der Elemente in der „Priority Queue“ entspricht. Eine Implementierung mit einer normal sortierten Liste und ohne Heap hätte hingegen eine annähernd lineare Laufzeit $O(N)$, wie auch bei Bleiweiss [4] beschrieben und erläutert.

Aufgrund der häufigen Verwendung der „Priority Queue“ und seiner Operationen innerhalb der Algorithmen ist dieser Unterschied ausschlaggebend und hat nachweisbar einen sehr großen Einfluss auf das allgemeine Laufzeitverhalten der Algorithmen. Zusätzlich zur „Priority Queue“ werden in den Pfadplanungsalgorithmen auch andere Listen verwendet, wie z.B. das assoziative Array mit den Werten der Bewertungsfunktion für Knoten in `f`, zu sehen und auch verwendet in den Pseudocodes in Abbildung 24 und Abbildung 25. Auch deren Performanz ist ausschlaggebend für das Laufzeitverhalten der Algorithmen. Wie diese assoziativen Listen umgesetzt bzw. implementiert sind und wie sie einen Beitrag zu einem guten Laufzeitverhalten leisten, wird in dem nun folgenden Kapitel genauer erläutert und präsentiert.

5.2.2 Hashed Array

Alle Listen der Pfadplanungsalgorithmen des Frameworks, ausgenommen der „Priority Queue“, sind als so genanntes „hashed Array“ implementiert. Bei den Listen handelt es sich unter anderem um die Closed-Liste sowie um die Listen `kostenVonStartZu` und `f`, zu sehen in den Pseudocodes der Pfadplanungsalgorithmen im Kapitel 4.2 in Abbildung 16, Abbildung 19 und Abbildung 21.

Ein „hashed Array“ speichert Werte bezüglich Knoten, wobei ein Knoten genau einen Wert in einem Index des „hashed Array“ besitzt. Im Framework sind die Werte in der Regel Gleitkommazahlen vom Typ `float`, die Abstände oder Bewertungen wie heuristische Werte für Knoten beinhalten. Nur die Closed-Liste ist eine Sonderform und hält stattdessen boolesche Werte (`true` oder `false`) die bestimmen, ob ein Knoten schon in die Liste eingefügt wurde oder nicht, wie auch im Kapitel 5.3.1 genauer erläutert.

Das „hashed Array“ berechnet für jedes Element im Array einen so genannten „Hash“. Dieser ist in der Informatik als eine Art Schlüssel zu verstehen, der durch eine Umrechnung aus einem anderen Wert oder Objekt ermittelt wird. Diese Umrechnung geschieht mit einer Funktion, der so genannten „Hash-Funktion“, und ist für alle Elemente im Array gleich definiert. Der Schlüssel entspricht einem Index in diesem „hashed Array“, dem das Objekt oder der Wert zugewiesen wird. Dabei muss sichergestellt sein, dass jedes Objekt eindeutig mit dessen Schlüssel bzw. mit dessen Index im „hashed Array“ identifizierbar ist und keine Überschneidungen entstehen können, ausgelöst durch gleiche Ergebnisse der „Hash-Funktion“ für unterschiedliche Objekte bzw. Werte.

Im Framework repräsentieren Knoten die einzelnen Indices eines „hashed Array“. Jedem Knoten kann, mit Hilfe dessen x- und y-Werts sowie mit Informationen über die Ausmaße der Welt, ein eindeutiger Schlüssel bzw. Index innerhalb des „hashed Array“ berechnet werden. Die im Framework verwendete „Hash-Funktion“ zur Berechnung des Index eines Knotens innerhalb des „hashed Array“ ist wie folgt in (10) definiert:

$$key = n.y \times width + n.x \tag{10}$$

Wobei n dem Knoten, dessen Index key im „hashed Array“ gefunden werden soll, und $width$ der Breite der Welt, aufgebaut als Raster, in Anzahl an Feldern entspricht.

Diese Funktion berechnet aus dem x- und y-Wert von Knoten, im Zusammenhang mit der Breite des Rasters in Feldern $width$, welches die Welt repräsentiert, einen eindeutig dem Knoten zuweisbaren Schlüssel key. Im Grunde konvertiert diese Funktion einen zweidimensionalen Index mit x- und y-Werten zu einem eindimensionalen Index für das „hashed Array“. Eine bildliche Darstellung dieser Konvertierung kann in Abbildung 26 (a) und (b) gesehen werden und lässt deren Verhalten sehr gut erkennen.

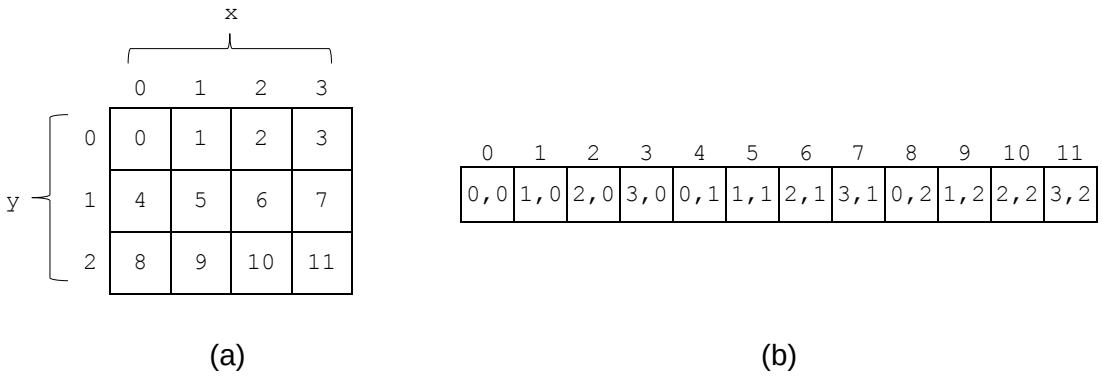


Abbildung 26: Darstellung der Funktionsweise der „Hash-Funktion“ aus (10).

Wie in Abbildung 26 gesehen werden kann, wird aus jedem Knoten mit dessen zweidimensionalen x- und y-Wert, zu erkennen in (a), ein Index für das in (b) dargestellte Array berechnet. In der zweidimensionalen Darstellung des Rasters in Abbildung 26 (a) sind innerhalb der Felder die berechneten Werte der „Hash-Funktion“ in (10) für die Knoten zu erkennen. Diese Werte entsprechen den Indexwerten des „hashed Array“, dargestellt in Abbildung 26 (b), deren Elemente wiederum den einzelnen Knoten zugewiesen werden können, identifiziert und dargestellt durch deren x- und y-Werte getrennt durch Kommata.

Solch ein „hashed Array“ mit seiner Konvertierungsfunktion bzw. „Hash-Funktion“ hat den großen Vorteil, dass jeder gespeicherte Wert für Knoten innerhalb des Arrays schnell durch die Berechnung des Schlüssels gefunden werden kann. Eine Suche im Array wird in dem Sinn nie benötigt. Jeder Zugriff geschieht durch die Berechnung des Index mit Hilfe des Knotens. Dadurch hat die Anzahl der beinhaltenden Knoten keinen Einfluss auf die Operationen des „hashed Array“ wie z.B. dem Einfügen, Löschen und Abfragen von Werten zu Knoten.

In dieser Eigenschaft liegt auch der große Vorteil des „hashed Array“. Es besitzt für alle Operationen eine konstante Laufzeit bei steigender Anzahl an Elementen, in O-Notation $O(1)$. Die Anzahl der schon eingefügten Werte hat somit keinen Einfluss auf deren Operationen, im Gegensatz zur im Kapitel 5.2.1 vorgestellten „Priority Queue“ mit ihrer logarithmischen Laufzeit $O(logN)$, wobei N der Anzahl der beinhaltenden Elemente entspricht. Auch der Speicherverbrauch ist konstant und entspricht immer der Anzahl der Felder im Raster bzw. der Knoten im Graphen.

Diese Implementierung mit einem „Hash“ ist, im Gegensatz zu den in der C++ STL zu findenden Containern zur Verwaltung von Elementen mit Schlüsseln wie die „Map“ oder das „Set“, sowohl für die CPU-Implementierung als auch für die GPU-Implementierung sehr gut geeignet. Sie wurden im Framework in C++, CUDA [2] und OpenCL [3] auch genau gleich umgesetzt, damit der Vergleich der Laufzeiten der Algorithmen sowie der Technologien so fair und nachweisbar wie möglich gestaltet werden kann.

5.3 Implementierungen der Pfadplanung

Nachdem in den vorhergehenden Kapiteln die Implementierung der Datenstrukturen und Listen für die Algorithmen behandelt wurde, werden nun in den folgenden Abschnitten die Umsetzung der Algorithmen bzw. die Prinzipien der Umsetzung der Pfadplanung in den einzelnen Technologien, C++ bzw. OpenMP [1], CUDA [2] und OpenCL [3] vorgestellt. Dabei wird zuerst die Umsetzung der Pfadplanung auf der CPU erläutert und danach die Umsetzung allgemein für die GPU, also mittels CUDA und OpenCL, behandelt. Dabei werden die zum Starten, Laden, Verwalten und schlussendlich auch zum Parallelisieren der Algorithmen verwendeten Prinzipien und Vorgehensweisen präsentiert und erläutert.

Die Implementierungen der Algorithmen selbst, welche den im Kapitel 4.2 vorgestellten Pseudocodes der Algorithmen entsprechen, werden in den nächsten Abschnitten nicht mehr näher erläutert und sind in den einzelnen Technologien auf die gleiche Art und Weise implementiert, um den Vergleich zwischen den einzelnen Technologien so fair wie möglich gestalten zu können.

5.3.1 CPU Implementierung

Die Implementierung für die CPU wurde mittels C++ umgesetzt. Für die Parallelität bzw. für die Aufteilung der Berechnung der Agenten auf die im System vorhandenen CPU-Kerne wird mittels der OpenMP API [1] erreicht. Die Liste der zu simulierenden Agenten wird innerhalb eines „Vectors“ der C++ STL verwaltet und besitzt wie ein Array einen Indexoperator für den Zugriff auf die Liste mittels Indexwerte. Dieser „Vector“ wird mit Hilfe von OpenMP und der „`#pragma omp for`“-Direktive [1] innerhalb einer `for`-Schleife, die über diesen „Vector“ iteriert, in Teilstücke aufgeteilt und diese dann parallel auf unterschiedlichen Threads auf der CPU abgearbeitet. Die Anzahl der zu startenden Threads wird von OpenMP mittels der API-Funktion „`omp_get_num_procs()`“ ermittelt und entspricht der Anzahl der im System vorhandenen CPU-Kerne.

Die Listen der Pfadplanung, also die „Priority Queue“, die Closed-Liste sowie andere Listen vom Typ „hashed Array“, sind innerhalb in C++ als Klassen implementiert, welche in den UML-Klassendiagrammen in Abbildung 27 und Abbildung 28 vorgestellt werden. Diese verwalten die Daten in dynamisch allokierten Arrays, welche zu Beginn der Algorithmen für jeden Agenten innerhalb des Konstruktors der Liste einmal allokiert und am Ende der Berechnung des Agenten innerhalb des Destruktors „DTOR“ wieder freigegeben werden. Die Größe der Arrays entspricht immer der Anzahl der vorhandenen Felder im Raster, das den Graphen repräsentiert. Diese Größe wird mittels der Breite und der Höhe der Welt berechnet, welche den Konstruktoren „CTOR“ der Listen als Übergabewerte `width` und `height` den Klassen „HashArray“, „ClosedList“ und „PriorityQueue“ übergeben werden, wie auch in den Klassendiagrammen in Abbildung 27 und Abbildung 28 zu erkennen.

Die beiden Klassen „HashArray“ und „ClosedList“ besitzen die im Kapitel 5.2.2 vorgestellten Eigenschaften eines „hashed Array“. Die in Formel (10) vorgestellte „Hash-Funktion“ wird innerhalb der Basisklasse von „HashArray“ und „ClosedList“ namens „Hashable“ in der Funktion „CreateKey“, zu sehen in Abbildung 27, implementiert. Diese bekommt den x- und y-Wert des zu berechnenden Knotens als eine Struktur vom Typ „int2“ übergeben, welche zwei Integer beinhaltet, genannt `x` und `y`. Die Breite der Welt, genannt `width`, wird bei der Konstruktion des „HashArray“ und der „ClosedList“ dem Konstruktor derer Basisklasse „Hashable“ übergeben. Dieser Wert wird, so wie auch in Formel (10) zu erkennen, für die Berechnung des „Hash-Wertes“ benötigt.

Die Klasse „HashArray“ besitzt zusätzlich zur Breite `width` und Höhe `height` noch einen zusätzlichen Übergabewert des Konstruktors namens `init`. Der Wert von `init` wird verwendet, um die Werte innerhalb des float-Arrays `m_Array` zu initialisieren. Dieser Wert ist in der Regel -1.0 und entspricht einem „leeren“ bzw. „nicht vorhandenen“ Inhalt für diesen Index in `m_Array`. Die Funktion „Contains“ prüft diesen Wert indem der Index des Knotens, repräsentiert durch die int2-Variable `node`, mit der „Hash-Funktion“ berechnet wird. Ist der Wert bei diesem Index -1.0, dann befindet sich kein Wert zu diesem Knoten in der Liste und die Funktion gibt `false` zurück, anderenfalls `true`.

Die beiden Funktionen „GetValueOf“ und „SetValueOf“ geben den Wert zurück bzw. setzen den Wert des Knotens `node` in der Liste `m_Array`, indem der Index des Arrays mittels der Funktion „CreateKey“ der Basisklasse „Hashable“ berechnet und dann für den Zugriff auf `m_Array` direkt verwendet wird.

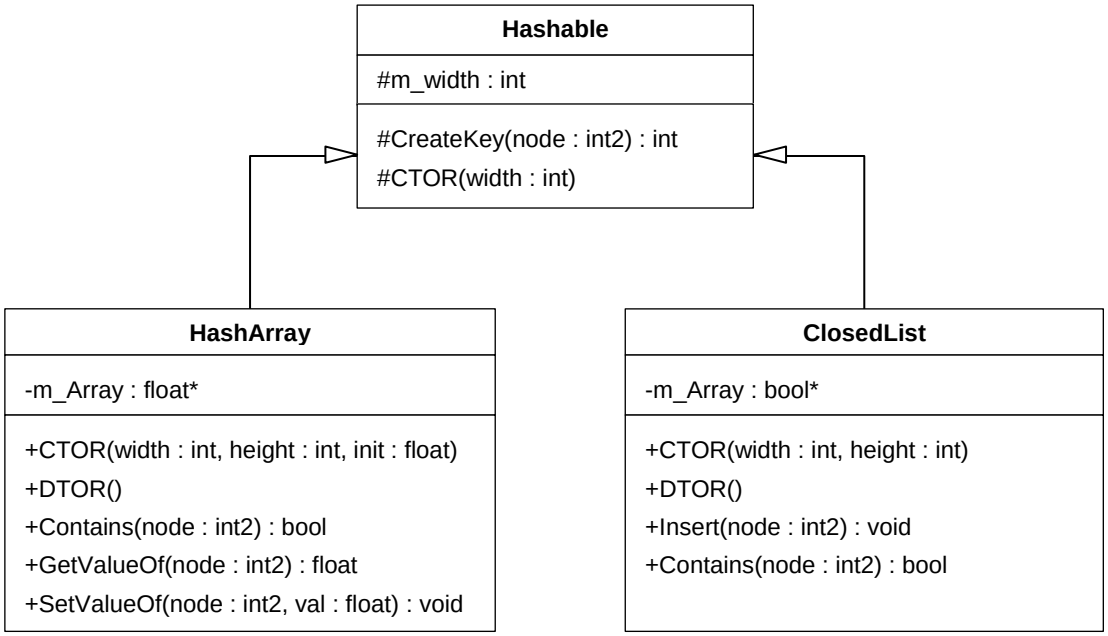


Abbildung 27: UML-Klassendiagramm der Klassen „HashArray“ und „ClosedList“.

Die Klasse „ClosedList“ besitzt so wie „HashArray“ ein Array zum Speichern der Werte namens `m_Array`, nur hält dieses boolesche Werte, welche anzeigen, ob sich ein Knoten schon in der Closed-Liste befindet (`true`) oder nicht (`false`). Die Werte im Array werden mit `false` initialisiert, da von Beginn an die Closed-Liste immer leer ist. Deswegen gibt es auch keinen Übergabeparameter `init` wie beim Konstruktor der Klasse „HashArray“. Die Funktion „Insert“ entspricht der Funktion „SetValueOf“ der Klasse „HashArray“ und setzt den Wert des Knotens bei dessen Index, berechnet mit der „Hash-Funktion“ „CreateKey“ der Basisklasse, auf `true`. Die Funktion „Contains“ hingegen prüft den Wert eines Knotens, in dem der boolesche Wert im Array zu diesem Knoten zurückgegeben wird. Dies entspricht der Implementierung nach der Funktion „GetValueOf“ der Klasse „HashArray“.

Die Klasse „Priority Queue“, zu sehen in Abbildung 28, besitzt ebenfalls einen Konstruktor „CTOR“, der die Breite und Höhe der Welt für die Initialisierung des Arrays `m_queue` verwendet, in dem der Heap, so wie in Kapitel 5.2.1 erläutert und `pqArray` genannt, mit Knoten vom Typ „int2“ verwaltet wird. Zusätzlich wird im Konstruktor „CTOR“ eine Referenz zu einem „HashArray“ übergeben, welches die Werte der zu den in `m_queue` eingefügten Knoten beinhaltet, genannt `m_fValues`. Dies ist die Liste, wie ebenfalls im Kapitel 5.2.1 erläutert, genannt `f`, welche die vom Algorithmus berechneten Bewertungen zu den Knoten beinhaltet, die in die „Priority Queue“ eingefügt werden. Des Weiteren entspricht die Klassenvariable `m_size`, zu sehen im Klassendiagramm in Abbildung 28, der Variable `pqArray.size` aus den Pseudocodes in Kapitel 5.2.1.

Die Funktionen „Push“ und Extract“ entsprechen den in Kapitel 5.2.1 in Pseudocodes vorgestellten Funktionen „Insert“ und „Extract“ der „Priority Queue“ und sind auch dieser Maßen implementiert. Die Funktion „MinHeapify“ entspricht hierbei der von „Extract“ aufgerufenen Heap-Sortierungsfunktion, im Pseudocode im Kapitel 5.2.1 „Heapify“ genannt, und wird detailliert im Anhang B dieser Arbeit erläutert und in Form von Pseudocode deren Implementierung im Framework präsentiert.

Die Funktion „Contains“ der Klasse „PriorityQueue“ überprüft das Vorhandensein eines Knotens innerhalb der „Priority Queue“ im Array `m_queue`, indem über dieses iteriert wird. Diese Operation besitzt bei der „Priority Queue“, im Gegensatz zu den „hashed Arrays“ wie „HashArray“ und „ClosedList“, welche eine konstante Laufzeit $O(1)$ beim Suchen von Elementen besitzen, eine lineare Laufzeit $O(N)$, wobei N der Anzahl der bereits eingefügten Knoten in der „Priority Queue“ `m_queue` entspricht, welche wiederum vom Wert der Klassenvariable `m_size` repräsentiert wird.

Des Weiteren überprüft die Funktion „NotEmpty“, ob die „Priority Queue“ nicht leer ist. Dies ist dann der Fall, wenn `m_size` dem Wert 0 entspricht. Dies tritt aber nur dann auf, wenn der Pfadplanungsalgorithmus den Zielknoten bei der Suche nicht finden kann.

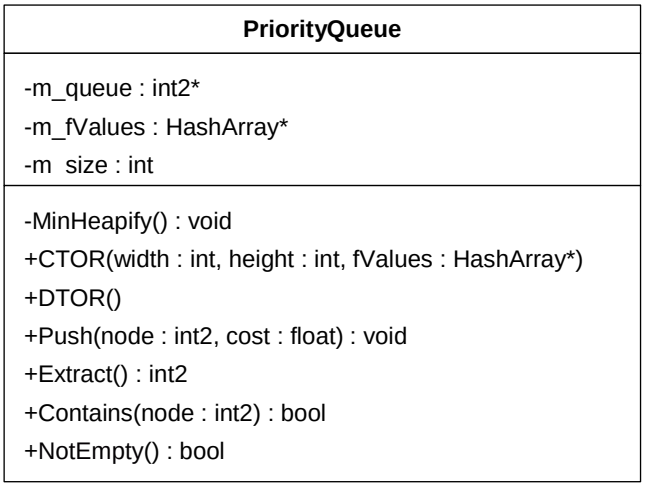


Abbildung 28: UML-Klassendiagramm der Klasse „PriorityQueue“.

5.3.2 GPU Implementierung

Die GPU-Implementierungen in CUDA [2] und OpenCL [3] unterscheiden sich zur CPU-Implementierung vor allem im C++ Code im Punkto Starten, Laden und Verwalten der Algorithmen. Die Berechnung der Algorithmen geschieht ausschließlich auf der GPU und wurden in CUDA C [2] bzw. OpenCL C [3] geschrieben. Die Parallelisierung geschieht mit Hilfe der Streaming-Multiprozessoren der GPU, wobei jeder Agent von einem Thread berechnet wird. Dies ist im Gegensatz zur CPU in so vielen Threads möglich, da die GPU, wie schon im Kapitel 3.3 erläutert, mehrere Tausend SIMD-Threads starten und verwalten kann, während die CPU im Vergleich nur wenige Hardware-Threads (z.B. vier bei einem „Intel Core2 Quad“ Prozessor) besitzt.

Die Algorithmen sind auf die gleiche Weise implementiert wie für die CPU in C++, vorgestellt im vorangehenden Kapitel 5.3.1. Für die GPU-Implementierung wurde aber keine objektorientierte Programmierung umgesetzt, sondern die Operationen der Listen wurden in Funktionen implementiert, welche auf der GPU ausgeführt werden und die zu operierenden Listen in Form von Pointern zu Arrays als Übergabewerte erhalten. Objektorientierte Programmierung wäre theoretisch in CUDA möglich gewesen, aber aus Kompatibilitätsgründen zu älteren CUDA-Versionen sowie zu OpenCL wurde dies ohne objektorientierter Programmierung umgesetzt.

Für das GPU-Computing müssen nicht nur seitens der GPU und des Device-Codes die Algorithmen implementiert werden, sondern im Host-Code bzw. in C++ und mit der GPU-Computing API einige Einstellungen und Berechnungen durchgeführt werden, bevor der erste Pfadplanungsalgorithmus auf der GPU berechnet werden kann. Die im Host-Code umgesetzten Berechnungen und durchgeführten Operationen werden nun in den folgenden Abschnitten erläutert und liefern einen tieferen Einblick in die Funktionsweise der CUDA Driver API [2] und der OpenCL API [3].

Die Welt als Textur

Die Repräsentation der Welt geschieht, wie auch bei der CPU-Implementierung, in einem zweidimensionalen Raster aus Knoten (siehe Kapitel 5.1). Diese liegt aber nicht, wie bei der CPU-Implementierung, im Arbeitsspeicher des Systems, sondern wird in den globalen RAM-Speicher der Grafikkarte geladen. Dabei wird die Repräsentation der Welt als eine Textur gespeichert. Dies hat gegenüber normal abgelegten Daten im globalen Speicher des Device den Vorteil, dass der Textur-Speicher, wie schon im Kapitel 3.4.1 erläutert und auch im „CUDA C Programming Guide“ [2] nachlesbar, über eine größere Bandbreite verfügt und überdies auch Zugriffe auf diesen vom Device gecached werden.

Die Repräsentation der Welt als eine zweidimensionale Textur bietet sich aufgrund der Rasterform der Welt und der Struktur der Knoten-Daten an. Da jeder Knoten einen Wert für `x`, `y`, und `weight` besitzt (siehe Kapitel 5.1 „Repräsentation des Graphen“), welche Ganzzahl bzw. Integer-Werte sind, können sie als Texel in der Textur dargestellt werden. Das Format der Textur bzw. der Texel entspricht dem RGBA-Format (Red, Green, Blue, Alpha), wobei der Rot-Wert dem `x`-Wert, der Grün-Wert dem `y`-Wert und der Blau-Wert der Gewichtung `weight` des Knotens entsprechen.

Der Alpha-Wert der Textur wird nicht verwendet. Dieser vierte Wert wird nur benötigt, da in OpenCL, wie auch in der OpenCL Spezifikation [3] nachlesbar, in den dreiteiligen RGB-Formaten keine 32-bit „Signed Integer“ gespeichert werden können, diese aber für die Knoten-Daten benötigt werden. Außerdem entspricht das RGBA-Format mit 32-bit „Signed Integer“ in OpenCL einer der minimalen Voraussetzungen für die Hardware zum Speichern von Texturen [3]. Zudem werden bei den Operationen zum Lesen von Texturen, sowohl in CUDA als auch OpenCL, immer entweder zwei oder vier Werte des Typs der Texturdaten (z.B. int2, int4, float2, float4, etc.) zurückgegeben, auch bei dreiteiligen RGB-Texturen.

Zusätzlich zum Datentyp der Textur muss auch die Größe der Welt bzw. die Größe der Textur zum Speichern der Welt betrachtet werden, damit die Kompatibilität zwischen unterschiedlicher Grafikhardware sowie auch zwischen CUDA und OpenCL gewährleistet werden kann. So werden die Ausmaße der Textur, also die Breite und Höhe in Texel, auf die nächst höhere Zweierpotenz (2, 4, 8, 16, 32, 64, 128, 256, 512,…) aufgerundet. So wird eine Welt, die die Breite von 20 und eine Höhe von 40 Feldern besitzt in eine Textur mit der Breite von 32 Texel und der Höhe 64 Texel gespeichert, da 32 die nächst höhere Zweierpotenz nach 20 und 64 die nächst höhere nach 40 ist.

Diese Größenänderung der Textur ist in OpenCL wichtig weil dort nur Texturen mit einer Seitengröße, die einer Zweierpotenz entspricht, erlaubt sind, so wie auch in der OpenCL Spezifikation [3] nachlesbar. Dies entspricht der klassischen Form zur Speicherung von Texturen in der Computergrafik und besitzt seitens der Grafikhardware, auch wenn es in CUDA nicht unbedingt nötig ist, die beste Kompatibilität und auch Performanz.

Device, Context und Platform Management

Da die Algorithmen auf einem Device ausgeführt werden, welches aber nicht zwingend im System vorhanden sein muss, muss mit Hilfe der APIs die Anzahl und Eigenschaften der für die Technologie vorhandenen Devices ermittelt werden. Hierfür gibt es sowohl bei CUDA als auch bei OpenCL Funktionen zum Abfragen und Verwalten von Devices, welche detailliert in den jeweiligen Spezifikationen [2] und [3] nachgelesen werden können. Diese „Device-Verwaltung“ bzw. dieses „Device Management“ wird in C++ Code mit der CUDA Driver API und der OpenCL API auf sehr ähnliche Weise realisiert.

Bei OpenCL gibt es noch zusätzlich das „Platform Management“, da es ja im Gegensatz zu CUDA, welche nur für die Nvidia-Plattform vorhanden ist, mehrere Plattformen, wie z.B. ATI oder Intel, unterstützt. Diese Zielplattform muss bei Verwendung der OpenCL API ausgewählt werden. Im Framework wird hierfür gezielt nach einer GPU oder einer Computing-Karte (wie z.B. die Nvidia Tesla-Karten) vom Typ „Nvidia CUDA“ oder „ATI Firestream“ gesucht. „Firestream“ ist hierbei die für ATI Grafikkarten implementierte GPU-Computing Lösung von AMD, welche mit OpenCL verwendet werden kann.

Zusätzlich zum „Device Management“ wird für das GPU-Computing mit der CUDA Driver API und der OpenCL API auch ein „Context Management“ benötigt, wobei unter dem „Context“ die Verbindung zwischen dem Host-Thread des Systems, der sich auf der CPU befindet, mit einem oder mehreren Devices verstanden werden kann. Unter diesem „Context“ werden dann alle weiteren Operationen und Einstellungen, durchgeführt von der GPU-Computing API, vorgenommen und gelten dann auch nur für diese im „Context“ definierten Devices, im Zusammenhang mit diesem Host-Thread. Zum Beispiel wird in OpenCL zu diesem „Context“ noch eine so genannte „Command Queue“ benötigt, welche die Befehle der OpenCL API für die GPU speichert und der Reihe nach abarbeitet. Diese „Command Queue“ (zu Deutsch: Befehlsliste) ist nur innerhalb dieses „Contexts“ gültig und wird somit auch nur für die Devices in diesem „Context“ ausgeführt.

Laden des Device-Codes

Wenn der „Context“, wie im vorherigen Abschnitt erläutert, von der API erzeugt wurde, kann für diesen der Code, der auf dem Device oder den Devices ausgeführt werden soll, geladen werden. Dieser „Device-Code“ kann, je nach API, unterschiedlich gespeichert und auch geladen werden. Bei CUDA z.B. wird während der Entwicklung der CUDA C Code, der sich in einer Textdatei mit der Endung .cu befindet, vom NVCC (Nvidia CUDA Compiler) in eine PTX-Datei (Parallel Thread Execution) konvertiert, welche ebenfalls eine Textdatei ist und Maschineninstruktionen für die GPU beinhaltet. Diese PTX-Datei wird während der Laufzeit im Framework von der CUDA Driver API geladen, und trägt den Namen „CUDA_Algorithms.ptx“, wie auch im Diagramm der Ordnerstruktur im Anhang A in Abbildung 40 gesehen werden kann.

Bei OpenCL wird der Device-Code mit Hilfe der API aus einer .cl-Datei geladen, welche den OpenCL C Code beinhaltet. Dieser Code wird zur Laufzeit im Framework geladen und mit Hilfe des OpenCL-Compilers, der in der OpenCL API integriert ist, für die im „Context“ vorhandenen Devices und für die zuvor bestimmte Plattform kompiliert. Es gibt somit während der Entwicklung keine Umwandlung des Codes, so wie bei CUDA mit dem NVCC, sondern der Quellcode wird direkt geladen und auch zur Laufzeit kompiliert. Diese .cl-Datei kann ebenfalls im Diagramm der Ordnerstruktur im Anhang A in Abbildung 40 gesehen werden und trägt den Namen „OpenCL_Algorithms.cl“.

Nachdem der Code aus der externen Datei geladen und von der API für die GPU vorbereitet bzw. kompiliert wurde, kann der Kernel aus diesem Code ausgewählt werden, der dem zu berechnenden Algorithmus entspricht. Dadurch wird ein Kernel-Objekt erzeugt, welches später mit weiteren API-Funktionen auf der GPU ausgeführt werden kann.

Berechnung der Block-Dim, Grid-Dim und Launches

Bevor ein Kernel auf der GPU gestartet werden kann, muss sowohl in CUDA als auch in OpenCL, die Größe des „Grids“ (Grid-Dim) und des „Blocks“ (Block-Dim) in CUDA bzw. die Größe der „Work-Dimension“ und der „Work-Group“ in OpenCL (siehe Kapitel 3.4.2), bestimmt werden, welche für die Ausführung des Algorithmus auf dem Device regelt. Diese Ausmaße werden, aufgrund von unterschiedlichen Restriktionen der Hardware, welche von den APIs abgefragt werden können, vom Framework berechnet.

Die erste Restriktion besteht in der Größe des globalen Arbeitsspeichers des Device. Es soll eine bestimmte Anzahl an Agenten berechnet werden, wobei jeder Agent seine eigenen Listen für die Pfadplanung benötigt, und zwar zur gleichen Zeit, da sie ja, im Gegensatz zur CPU, zugleich und parallel auf der GPU berechnet werden. Deswegen muss, ausgehend vom Speicherverbrauch der Listen für einen Agenten und der Anzahl der zu berechnenden Agenten, die maximale Anzahl an Threads für einen Durchgang zur Berechnung der Algorithmen bestimmt werden.

In CUDA wird hierfür die API-Funktion `cuMemGetInfo` verwendet, welche die Größe des freien RAM-Speichers auf dem Device zurückgibt. In OpenCL gibt es leider keine vergleichbare Funktion. Der für die Berechnung vorhandene Speicher wird mit Hilfe der Größe des gesamten globalen Arbeitsspeichers minus 20% angenommen. Dies kann aber, falls andere Applikationen auf dem System in Summe mehr als diese 20% des Speichers der Grafikkarte benötigen, in Folge bei den Allokationen für die Algorithmen auf dem Device zu Fehlern führen. Deswegen wurde, vor allem für einen guten Vergleich der Algorithmen während der Laufzeittests, das Programmargument „GPUforceMemUsage“ (siehe Tabelle 12 im Anhang A) eingeführt, welches den zu verwendeten globalen Speicher des Device in Bytes angibt, sodass sowohl die CUDA API als auch die OpenCL API dieselben Größen des vorhandenen globalen Speichers annehmen.

Die zweite Restriktion, die sich direkt aus den Kernels und den Hardwareressourcen der GPU ableitet, ist die Anzahl der Register, die für die Berechnung des Kernels vom Thread benötigt wird. Diese Restriktion besteht innerhalb eines Blocks in CUDA bzw. einer Work-Group in OpenCL, da ein Block bzw. eine Work-Group innerhalb desselben Streaming-Multiprozessors ausgeführt wird und dieser nur eine gewisse Anzahl an Registern für die Threads besitzt. Bei den CUDA-Algorithmen wird die Berechnung der Größe eines Blocks mit Hilfe der Eigenschaften und Hardwareinformationen des Device ermittelt und die maximale Anzahl an Threads pro Block, basierend auf der Anzahl an Registern pro Block für diesen Kernel, unter Zuhilfenahme von Informationen aus dem „CUDA C Programming Guide“ [2], berechnet.

Bei OpenCL ist dies einfacher gelöst, sodass keine plattformspezifische Informationen wie bei CUDA von Nöten sind, um die Größe der Work-Group für einen Kernel zu bestimmen. In der OpenCL API gibt es die Funktion namens `clGetKernelWorkGroupInfo`, welche die Größe der Work-Group bestimmt. Diese entspricht aber nicht der maximalen Anzahl an möglichen Threads so wie sie bei CUDA berechnet werden, sondern einer geringeren Anzahl. So wurde, damit beide APIs mit denselben Werten arbeiten, das Programmargument „GPUforceBlockDim“ (siehe Tabelle 12 im Anhang A) eingeführt, welches die Größe der Blocks bzw. der Work-Groups bestimmt und für die Laufzeitmessungen fixiert.

Die letzte zu beachtende Restriktion basiert auf der Größe der am Device vorhandenen „Shared Memory“ pro Block bei CUDA bzw. „Local Memory“ pro Work-Group bei OpenCL (Definitionen siehe Kapitel 3.4.1). In der Regel benötigen die Kernels zur Berechnung der Pfadplanungsalgorithmen keine „Shared Memory“ bzw. „Local Memory“, nur bei Angabe des Programmarguments „GPUforceSMEM“ (siehe Tabelle 12 im Anhang A) werden die Listen der Algorithmen, bei gleichzeitiger Angabe der beiden Programmargumente „GPUforceBlockDim“ und „scenarioMultiplier“ (siehe ebenfalls Tabelle 12 im Anhang A) innerhalb der „Shared Memory“ der Blocks bzw. innerhalb der „Local Memory“ der Work-Group verwaltet. Dies kann zu einer Laufzeitsteigerung führen, da die „Shared Memory“ bzw. die „Local Memory“ eine viel kleinere Latenz besitzt als der globale Arbeitsspeicher des Device, da sich dieser auf demselben Chip wie der Streaming-Multiprozessor befindet, so wie auch im Kapitel 3.4.1 in Tabelle 2 aufgezeigt und auch im „CUDA C Programming Guide“ [2] nachlesbar.

Starten des Kernels

Wenn alle Restriktionen für die Simulation der Agenten, welche im letzten Abschnitt erläutert wurden, geregelt werden konnten und keine Fehler in den APIs aufgetreten sind, werden die Speicherbereiche der Listen allokiert und die Parameter der Kernels zur Pfadplanung gesetzt. Wenn die Simulation startet, werden die momentan aktuellen Positionen der Agenten und deren Zielpositionen im Raster, welches die Welt repräsentiert, innerhalb von Arrays auf das Device kopiert wird.

Mit diesen beiden kopierten Arrays und der Textur, welche die Welt repräsentiert und sich ebenfalls im globalen Speicher des Device befindet, wird dann der Kernel mit den zuvor berechneten Einstellungen, darunter die Größe der Blocks und die Anzahl der Blocks pro Grid in CUDA bzw. der Größe der Work-Group und der Work-Dimension in OpenCL, gestartet. Dabei ist es möglich, dass mehrere Iterationen bzw. „Launches“ durchgeführt werden müssen, da aufgrund einer Restriktion aus dem vorherigen Abschnitt nicht alle Agenten zugleich auf dem Device berechnet werden können.

Wenn alle „Launches“ des Kernels durchgeführt wurden, wird mit Hilfe einer API-Funktion und mit „Events“ (Details siehe die jeweiligen Spezifikationen der APIs [2] und [3]), sowohl in CUDA als auch in OpenCL, auf den genauen Zeitpunkt des Endes der Berechnung auf der GPU gewartet. Das Array, welches zuvor die momentanen Positionen der Agenten dem Kernel übergab, beinhaltet nun die vom Pfadplanungsalgorithmus neu berechneten Positionen. Dieses wird wieder mittels einer API-Funktion von CUDA bzw. OpenCL zurück auf den Arbeitsspeicher des Hosts kopiert und für die weitere Verarbeitung der Agenten seitens des Hosts auf der CPU bzw. in C++, außerhalb der GPU-Computing API und der Pfadplanungsalgorithmen, verwendet.

6 Laufzeittests der Implementierungen

Im letzten Kapitel wurde das Framework zur Durchführung der Laufzeittests zum Vergleich der Algorithmen und der Technologien namens „Masterthesis_A_Star“ vorgestellt, welches im Laufe dieser Arbeit entwickelt wurde und in der Bedienungsanleitung im Anhang A dieser Arbeit genauer betrachtet wird. Nachdem in dieser Arbeit die Aspekte des GPU-Computing und der Pfadplanung detailliert bearbeitet und präsentiert wurden, sowie auch Vorarbeiten im Bereich der GPU beschleunigten Pfadplanung vorgestellt wurden, wird nun mit der eigens geschriebenen Applikation mittels Benchmarking die Erkenntnisse dieser Vorarbeiten versucht zu bestätigen und deren Ergebnisse nachzuvollziehen.

Die eigens erhobenen Laufzeitergebnisse dieser Tests werden präsentiert und in späterer Folge auch interpretiert und analysiert. Zuvor wird die Testphilosophie sowie die zu testenden Szenarien der Pfadplanung vorgestellt, welche im „Benchmark-Modus“ des Frameworks (Details siehe Anhang A „Der Benchmark-Modus“) auf einem Testsystem ausgeführt werden, welches nun im nächsten Abschnitt genauer vorgestellt wird.

6.1 Das Testsystem

Das Testsystem, auf dem die Laufzeittests der Pfadplanungsalgorithmen, sowohl für die CPU als auch für die GPU, durchgeführt werden, ist ein Desktop-PC mit Windows Betriebssystem, auf dem auch das Framework mittels Microsoft Visual Studio 2008 entwickelt und kompiliert wurde. Genauer handelt es sich beim Betriebssystem um ein deutschsprachiges Windows 7 Professional 64bit mit installiertem Service Pack 1 sowie allen aktuellen Updates. Dieses Betriebssystem sowie auch die Software zur Entwicklung des Frameworks wurden von der Fachhochschule Technikum Wien bereitgestellt.

Hardwaretechnisch besitzt dieses System für die Berechnung der CPU-Algorithmen sowie anderer programmiertechnischer Aufgaben einen Vierkern-Prozessor von Intel, genauer ein „Intel Core2 Quad Q9550“ mit einer Taktung von 2.83 GHz (2830 MHz). Dies entspricht einem heutzutage typischen Vierkern-Prozessor, wie er auch in vielen anderen ähnlichen Systemen gefunden werden kann. Die Größe des Arbeitsspeichers des Systems bzw. in GPU-Computing Termini ausgedrückt die „Host-Memory“ beläuft sich auf 4 GB (4096 MB) und sind passend zur CPU in Form von zwei 2 GB (2048 MB) DDR2-800 RAM-Bausteinen im System vorhanden.

Bei der Grafikkarte des Systems, mit welcher die GPU-Algorithmen sowie andere grafikspezifische Aufgaben berechnet werden, handelt es sich um eine „Nvidia GeForce GTX 460“, welche eine GPU aus der Architektur namens „Fermi“ von Nvidia besitzt. Genauer ist diese mit einem „GF104“ Chip bestückt, welcher sieben Streaming-Multiprozessoren mit je 48 Kernen besitzt, was in Summe eine Anzahl von 336 Rechen-kernen (ALUs) auf der GPU ergibt. Der globale Arbeitsspeicher, bzw. im GPU-Computing

die so genannte „Device Memory“, ist ein GDDR5-RAM mit einem Speicherinterface von 256 Bit und einer Größe von 1 GB (1024 MB). Der Treiber, welcher für diese GPU auf dem System installiert ist und verwendet wird, ist der im Moment des Schreibens aktuellste WHQL-Treiber aus dem Hause Nvidia mit der Version 275.33 (GeForce 275.33 WHQL Driver). Dieser Grafikkartentreiber beinhaltet auch den CUDA-Treiber, welcher für das GPU-Computing mit CUDA und OpenCL notwendig ist. Dieser ist der im Moment des Schreibens aktuellste und besitzt die Versionsnummer 4.0.1 (NVIDIA CUDA 4.0.1 Driver).

Das System ist somit für die Ausführung der Algorithmen auf der GPU in CUDA und in OpenCL, sowohl hardwaretechnisch als auch softwaretechnisch, gut gerüstet. Trotzdem gibt es noch ein Problem, welches während der Entwicklung auftauchte und für die Erhebung der Daten durch Laufzeittests hinderlich ist. Dabei handelt es sich um die Tatsache, dass das Windows-Betriebssystem der GPU zur Berechnung einer Aufgabe in der Regel nur wenige Sekunden Zeit gibt, wenn diese GPU auch gleichzeitig für die Display-Ausgabe verantwortlich ist, was beim Testsystem auch der Fall ist. Ist die GPU also beschäftigt und kann nach einem Timeout von mehreren Sekunden kein neues Bild für den Windows-Desktop o.Ä. rendern, dann nimmt das Betriebssystem einen Fehler an und setzt sicherheitshalber das Device auf dessen Ursprung zurück (*Recovery*). Dabei werden alle momentanen Berechnungen auf der GPU beendet, so auch die Pfadplanung, welche bei einer höheren Anzahl an Agenten länger dauern kann als dieses Timeout.

Zum Glück gibt es für dieses Problem aber eine Lösung, welche in der MSDN-Library von Microsoft [17] gefunden werden kann. Hierfür muss in der Windows-Registry für das „Windows Display Driver Model“ (WDDM) ein neuer Registry-Key eingetragen werden, z.B. mit einem Registry-Editor wie „regedit.exe“ von Windows. Unter dem Registry-Pfad

HKEY_LOCAL_MACHINE → System → CurrentControlSet → Control → GraphicsDrivers

muss ein neuer Wert vom Typ DWORD eingefügt werden, namens „TdrLevel“ (Timeout Detection and Recovery Level), welcher den Wert „0“ bzw. in Hexadezimal „0x00000000“ besitzt. Dieser Registry-Eintrag deaktiviert die Erkennung von Timeouts der GPU sowie die automatische „Recovery“ des Device, wodurch auch Laufzeittests auf der GPU mit größeren Agentenzahlen möglich werden, auch wenn die GPU für das Rendern des Displays in Verwendung ist. Dieses Feature, das „WDDM“, ist erst ab Windows Vista vorhanden und kann somit nicht für Windows XP adaptiert werden.

Schlussendlich sollte noch erwähnt werden, dass für die Durchführung der Laufzeittests beim Windows 7 Betriebssystem das „Aero-Design“ für die Darstellung der Fenster und des Desktops deaktiviert wurde, da dieses einige Grafikressourcen benötigt. Stattdessen wurde das klassische Windows Design ausgewählt, welches nachweisbar weniger Grafikspeicher und Effekte benötigt als das seit Windows Vista vorhandene und standardgemäß aktivierte „Aero-Design“.

6.2 Testphilosophie

Die Laufzeittests werden auf dem im letzten Kapitel präsentierten Testsystem mit den vorgestellten Konfigurationen und Einstellungen durchgeführt. Die Szenarien, welche sich durch eine unterschiedliche Anzahl an zu berechnenden Agenten und der Ausmaße der Welt unterscheiden, werden im Laufe dieses Kapitels vorgestellt und spezifiziert. Die Einstellungen für die Szenarien werden über Programmzeilenargumente des Frameworks erreicht. Da es aber für alle Testläufe allgemein gültige Einstellungen gibt, wurde in der Konfigurationsdatei des Frameworks namens „config.ini“ verwendet, welche in der Bedienungsanleitung des Frameworks im Anhang A der Arbeit vorgestellt wird. Die in dieser Konfigurationsdatei vorgenommenen Einstellungen werden nun im nächsten Abschnitt näher erläutert.

6.2.1 Global gültige Programmargumente

Erstens wurde für alle Testläufe in allen Technologien, da sie im „Benchmark-Modus“ der Applikation durchgeführt werden sollen, welcher genauer im Anhang A der Arbeit erläutert wird, in der Konfigurationsdatei dem Argument „benchmark“ der Wert „true“ zugewiesen. Zudem werden alle Algorithmen mit der so genannten „Moore“-Nachbarschaft, wie sie schon im Kapitel 5.1.1 vorgestellt wurde, berechnet, indem das Programmargument „neighbourhood“ (siehe Anhang A, Tabelle 12) den Wert „Moore“ zugewiesen bekommt.

Für die Einstellung der Berechnung der Algorithmen auf der GPU wird das Programmargument „GPUforceBlockDim“, welches die maximale Größe eines Blocks in CUDA bzw. der Work-Group in OpenCL spezifiziert, in der „config.ini“ für alle Testläufe auf den Wert „768“ gesetzt. Im Grunde kann die „GeForce GTX 460“, die, wie im letzten Kapitel erläutert, für die Testläufe verwendet wird, 1024 Threads pro Block bzw. Work-Items pro Work-Group berechnen. Da aber die Funktion zur Ermittlung der Anzahl an Work-Items innerhalb der Work-Group bei OpenCL namens `clGetKernelWorkGroupInfo`, welche im Kapitel 5.3.2 vorgestellt wurde, immer diesen Wert von 768 zurückgibt, wurde die maximale Anzahl einfach für beide GPU-Technologien auf diese 768 eingestellt, damit auch die Testläufe in CUDA dieselben Ausmaße und Anzahl an Threads und Blocks verwendet wie OpenCL für dessen Work-Items und Work-Groups.

Zusätzlich musste, da die OpenCL API im Gegensatz zu CUDA, wie ebenfalls schon im Kapitel 5.3.2 erläutert, keine Funktion zur Abfrage des freien globalen Speichers der GPU besitzt, das Programmargument „GPUforceMemUsage“ eingestellt werden, welches die maximal zu verwendende Größe des globalen Speichers des Device in Bytes für die Listen der Pfadplanung spezifiziert. Für die „Geforce GTX 460“ mit ihren 1024 MB RAM wurde hier der Wert von „943718400“ angeben, was umgerechnet 900 MB entspricht. Mit dieser Einstellung können alle Szenarien mit ihrer unterschiedlichen Anzahl an Agenten und Ausmaße der Welt, wie nun im nächsten Abschnitt erläutert, berechnet werden.

6.2.2 Die Szenarien

In Summe werden zehn unterschiedliche Szenarien in jeder Technologie mit jedem Algorithmus ausgeführt und mit dem „Benchmark-Modus“ des Frameworks gemessen. Die Erstellung dieser Szenarien passiert unter der Angabe der Programmargumente „agents“ und „worldsize“ des Frameworks, welche, wie im Anhang A der Arbeit in Tabelle 12 vorgestellt und erklärt, die Anzahl der zu berechnenden Agenten und die Seitengröße der rasterförmigen Welt in Feldern spezifizieren.

Die Szenarien, welche vom Framework erzeugt werden sollen, genannt S0 bis S9, werden in Tabelle 4 vorgestellt. Dort werden die Anzahl der zu berechnenden Agenten pro Szenario, anfangend bei nur 12 bei S0 bis hin zu 768.432 bei S9, sowie die Ausmaße der Welt, anfangend mit einer Seitenlänge von 6 bei S0 bis hin zu 30 bei S9, eingestellt mit dem Programmargument „worldsize“, präsentiert. Die gesamte Anzahl an Knoten ergibt sich, wie auch in Tabelle 4 zu erkennen, durch die Multiplikation der Weltgröße, welche die Seitenlängen für die Höhe und die Breite der Welt spezifiziert.

Szenarien	Agenten	Weltgröße	Anzahl Knoten
S0	12	6 x 6	36
S1	48	6 x 6	36
S2	192	6 x 6	36
S3	768	6 x 6	36
S4	3.072	8 x 8	64
S5	12.288	12 x 12	144
S6	49.152	16 x 16	256
S7	196.608	22 x 22	484
S8	393.216	26 x 26	676
S9	768.432	30 x 30	900

Tabelle 4: Spezifikation der zehn im Framework durchzuführenden Szenarien.

Da, wie schon im vorherigen Abschnitt erwähnt, die Größe eines Blocks in CUDA bzw. einer Work-Group in OpenCL auf 768 fixiert wird, ist die Anzahl der Agenten ebenfalls immer auf ein Vielfaches von 768 festgelegt, bis auf die ersten drei, welche weniger als 768 Agenten berechnen. Die Anzahl der Agenten steigt von Szenario zu Szenario immer um das Vierfache an, beginnend bei 12. Nur bei den letzten Szenarien (S7 auf S8 und S8 auf S9) wurde aufgrund der hohen Agentenzahlen stattdessen eine Verdoppelung gewählt.

Da jedes dieser zehn Szenarien mit jedem der drei Algorithmen (BFS, Dijkstra und A*) berechnet werden soll, welche jeweils in drei Technologien (C++, CUDA und OpenCL) entwickelt wurden, ergibt sich eine Summe von 90 Testläufen zu diesen zehn Szenarien (10 Szenarien x 3 Technologien x 3 Algorithmen = 90 Testläufe).

Die Start- und Endpositionen der Agenten werden, wie auch in der Bedienungsanleitung des Frameworks im Anhang A dieser Arbeit erläutert, bei der Angabe von „agents“ und „worldsize“, zufällig für jeden Agenten bestimmt. Die GPU ist aber für eine parallele Durchführung von Threads ausgelegt, welche die beste Performanz erhalten, wenn keine if-Anweisungen oder andere Codeteile die Threads innerhalb des Blocks bzw. der Work-Group dazu bringen, zu divergieren, sodass sie andere Codepfade ausführen. So wie im „CUDA C Programming Guide“ [2] nachgelesen werden kann, werden im Fall von divergierenden Threads innerhalb eines „Warps“, was als ein Teil eines Blocks in einer bestimmten Größe verstanden werden kann, nicht mehr parallel abgearbeitet sondern seriell, wenn sie nicht in ihren Ausführungspfaden übereinstimmen, was zu Einbußen in der Performanz führen kann.

Da dies bei so vielen Agenten mit unterschiedlichen Start- und Endpositionen sehr wahrscheinlich ist, wurde zum Vergleich zu den zehn Szenarien weitere zehn bestimmt, zu sehen in Tabelle 5, welche mit den Szenarien aus Tabelle 4 in der Summe der Agenten und in den Weltgrößen übereinstimmen. Der Unterschied liegt hierbei bei den neuen zehn Szenarien in der Verwendung des Programmarguments „scenarioMultiplier“ (siehe Anhang A, Tabelle 12). Mit dessen Hilfe bekommen alle Agenten innerhalb eines Blocks bzw. einer Work-Group dieselben Start- und Endpositionen zugewiesen, indem jeder Agent mit seiner Start- und Endposition so oft erzeugt wird, wie in „scenarioMultiplier“ angegeben.

Durch die Angaben dieses Arguments und der Anzahl der Agenten wie in Tabelle 5, kann jeder Block in CUDA bzw. jede Work-Group in OpenCL Agenten mit den gleichen Start- und Endpositionen rechnen. Dies führt dazu, dass bei der Durchführung der Algorithmen auf der GPU innerhalb eines Blocks bzw. einer Work-Group immer dieselben Codepfade ausgeführt werden und die GPU somit mehr und besser parallelisieren kann als zuvor bei den zehn Szenarien vorgestellt in Tabelle 4.

Szenarien	Agenten	Scenario Multiplier	Block-Dim	Summe der Agenten
S0	1	12	12	12
S1	1	48	48	48
S2	1	192	192	192
S3	1	768	768	768
S4	4	768	768	3.072
S5	16	768	768	12.288
S6	64	768	768	49.152
S7	256	768	768	19.6608
S8	512	768	768	393.216
S9	1.024	768	768	768.432

Tabelle 5: Spezifikation der zehn Szenarien bei Verwendung von „Scenario Multiplier“.

Es kommen somit zusätzlich 90 weitere Testfälle zu den vorher bestimmten 90 dazu, da wieder jedes der neuen zehn Szenarien mit jeweils jedem der drei Algorithmen in jeder der drei Technologien durchgeführt wird. Das ergibt eine Summe von 180 Testläufen.

Ein weiterer Aspekt der GPU-Algorithmen, der aufgrund von Informationen aus dem „CUDA C Programming Guide“ [2] über die Speicherbereiche des Device, zum Nachdenken verleitet, ist die exzessive Verwendung der „global Memory“ des Device bei den Implementierungen der Algorithmen für die GPU in CUDA bzw. in OpenCL. Dieser Speicherbereich ist im Gegensatz zu den anderen wie z.B. die „Shared Memory“ in CUDA relativ langsam, so wie auch in dieser Arbeit im Kapitel 3.4.1 in der Übersicht der Speicherbereiche in Tabelle 2 gesehen werden kann.

Diese „Shared Memory“ bei CUDA bzw. das OpenCL äquivalent „Local Memory“ ist im Schnitt von den Zugriffslatenzen des Speichers, welcher direkt auf dem Chip der GPU liegt, um ein Vielfaches schneller als die „global Memory“, welche im externen RAM der Grafikkarte liegt. Dabei ist aber zu beachten, dass die Größe der „Shared Memory“ pro Block in CUDA bzw. die Größe der „Local Memory“ pro Work-Group bei OpenCL relativ klein ist gegenüber der „global Memory“ des Device. So hat die „GeForce GTX 460“, welche für die Laufzeittests verwendet wird, in Summe pro Block eine „Shared Memory“ von 48 KB, wobei sich die Größe der „global Memory“ auf Ganze 1024 MB beläuft.

Da bei den Szenarien mit „Szenario Multiplier“ alle Agenten innerhalb eines Blocks dasselbe rechnen und innerhalb dieses Blocks auf dieselbe „Shared Memory“ zugreifen, kann im nächsten Schritt noch eine Verlegung der Listen von der „global Memory“ auf die „Shared Memory“ überlegt werden. Dabei wird aber pro Block bzw. Work-Group nur ein einziges Set an Listen innerhalb der „Shared Memory“ bzw. „Local Memory“ benötigt, da alle Threads pro Block quasi denselben Agenten mehrmals berechnen und somit jeder Agent auch dieselben Inhalte der Listen erzeugt. Dieses Set an Listen pro Block bzw. pro Work-Group würde auch mit dem größten Szenario S9 bestehend aus 900 Knoten innerhalb der „Shared Memory“ bzw. „Local Memory“ der „GeForce GTX 460“ mit einer Größe von 48 KB Platz finden.

Diese zehn neuen Szenarien mit „Shared Memory“ werden mit denselben Programmargumenten wie die Szenarien mit „Scenario Multiplier“ ausgeführt, nur wird zusätzlich das Programmargument „GPUforceSMEM“ (siehe Tabelle 12 im Anhang A) mit dem Wert „true“ in der Programmzeile angegeben. Diese Testfälle werden nur mit den GPU-Algorithmen durchgeführt. Eine Durchführung der Testfälle mit den CPU-Algorithmen ist nicht nötig, da diese mit den CPU-Testläufen mit „Scenario Multiplier“ seitens der Programmeinstellungen des Frameworks ident wären. Sie werden nur in späterer Folge mit den Laufzeitergebnissen der CPU mit „Scenario Multiplier“ verglichen.

So entstehen weitere 60 Testläufe, wo jeder der zehn Szenarien mit jedem der drei Algorithmen mit den beiden GPU-Technologien CUDA und OpenCL ausgeführt werden. In Summe macht das mit den vorher definierten Testläufen eine Summe von insgesamt 240. Diese drei Testläufe mit jeweils 90, 90 und 60 Testfällen werden jeweils in eigene Kategorien ausgeführt und verglichen. Diese Kategorien werden in den folgenden Abschnitten sowie bei den Testläufen wie folgt genannt und unterschieden:

- Zehn Szenarien
- Zehn Szenarien mit „Scenario Multiplier“
- Zehn Szenarien mit „Shared Memory“

Deren Laufzeitergebnisse werden nun übersichtlich im folgenden Kapitel in tabellarischer Form für jeden dieser Kategorien präsentiert und deren Inhalte kurz erläutert.

6.3 Ergebnisse der Laufzeitmessungen

Auf den nun folgenden Seiten werden die Ergebnisse der Laufzeittests, aufgeteilt in die drei zuvor spezifizierten Kategorien, in tabellarischer Form präsentiert. Dabei werden pro Tabelle der Reihe nach die Ergebnisse der Szenarien S0 bis S9, mit den Agentenzahlen und Weltgrößen wie im vorherigen Abschnitt vorgestellt, präsentiert, welche des Weiteren in die jeweils drei Algorithmen aufgeteilt sind.

Zuerst wird der Speicherverbrauch der einzelnen Szenarien mit den Algorithmen auf der CPU, detailliert aufgelistet in Tabelle 6, aufgezeigt. Hierbei wird explizit der Verbrauch der von den Algorithmen verwendeten Listen, wie sie von den im Kapitel 5.2 vorgestellten C++ Klassen verwaltet werden, präsentiert. Dabei wird unter der Spalte „Speicherverbrauch (Bytes)“ der benutzte Arbeitsspeicher des Systems pro Agent aufgezeigt, welcher proportional zur Weltgröße mitsteigt, wie auch in Tabelle 6 erkennbar.

Daraus ergibt sich, aufgrund der Tatsache, dass sich im Testsystem ein Vierkern-Prozessor befindet und vier Threads bzw. Agenten zugleich berechnet werden, ein gesamter Speicherverbrauch vom Vierfachen des Verbrauches pro Agent, zu sehen in der Spalte mit der Überschrift „insgesamt“. In der letzten Spalte der Tabelle 6 mit der Überschrift „KB“ wird dieser Wert noch in Kilobytes anstatt Bytes angezeigt, damit die Zahlen noch verständlicher werden.

Der Speicherverbrauch der GPU-Algorithmen in Tabelle 7 wird auf dieselbe Art und Weise wie für die CPU in Tabelle 6 aufgezeigt. Zusätzlich zum Speicherverbrauch werden hier aber auch die so genannten „GPU Launch-Infos“ angezeigt. Diese beinhalten in der ersten Spalte die Anzahl an Threads pro Block bzw. Work-Items pro Work-Group, in der zweiten die Anzahl die Blocks pro Grid bzw. Work-Groups pro „NDRange“ und in der dritten die Anzahl der „Launches“, wie sie im Kapitel 5.3.2 beschrieben wurden.

Wie in Tabelle 7 zu erkennen steigt der gesamte Speicherverbrauch der GPU-Algorithmen im Vergleich zu den CPU-Algorithmen in Tabelle 6 stärker an, da die GPU mehr Agenten parallel berechnet und somit mehr Listen zugleich im Speicher halten muss als die CPU mit ihren vier Threads. Außerdem kann erkannt werden, dass bei den größeren Szenarien (ab S7) der Speicherverbrauch der Listen nie über 900 MB hinausgeht, wie im Kapitel 6.2.1 durch die Verwendung des Programmarguments „GPUforceMemUsage“ erläutert.

In Tabelle 8 wird der Speicherverbrauch der GPU-Algorithmen unter Verwendung der „Shared Memory“ der GPU aufgezeigt. Dabei wird der Verbrauch pro Block in CUDA bzw. pro Work-Group in OpenCL in der Spalte „SMEM pro Block (Bytes)“ aufgezeigt. Da die Listen nur mehr ein Mal pro Block bzw. pro Work-Group gespeichert werden und sich zusätzlich auch die Restriktionen bezüglich der „global Memory“ durch die Verwendung der „Shared Memory“ des Device bei diesen Testläufen aufheben, ändern sich auch die „GPU Launch-Infos“ im Vergleich zu denen der GPU-Algorithmen aus Tabelle 7, welche die „global Memory“ verwenden. Deren neuen Werte für die Laufzeittests mit „Shared Memory“ werden nun in Tabelle 8 aufgezeigt.

Der Speicherverbrauch der GPU-Algorithmen unter Verwendung der „Shared Memory“ entspricht den Größen der Listen der einzelnen Algorithmen pro Agent, wie sie Tabelle 7 präsentiert werden. Nur werden bei den Implementierungen der GPU-Algorithmen mit „Shared Memory“ zusätzlich zwölf Bytes für drei Integer Werte benötigt, wobei der erste und der zweite den x- und y-Wert des aus der „Priority Queue“ extrahierten Knoten beinhaltet und der dritte die Größe der „Priority Queue“ hält. Somit benötigen die GPU-Algorithmen mit „Shared Memory“ zwölf Bytes mehr pro Block als die GPU-Algorithmen mit „global Memory“ pro Agent, da diese die Größe der „Priority Queue“ und den extrahierten Knoten innerhalb des Kernels verwalten, sozusagen in der „local Memory“ von CUDA bzw. „private Memory“ von OpenCL. Details zu diesen Speicherbereichen können in den jeweiligen Spezifikationen von CUDA [2] und OpenCL [3] sowie in der Übersicht der Speicherbereiche in Tabelle 2 in Kapitel 3.4.1 dieser Arbeit nachgelesen werden.

Szenario	Algorithmus	Speicherverbrauch (Bytes)		
		pro Agent	insgesamt	KB
S0	A*	612	2.448	2,40 KB
	BFS	468	1.872	1,83 KB
	Dijkstra	432	1.728	1,69 KB
S1	A*	612	2.448	2,40 KB
	BFS	468	1.872	1,83 KB
	Dijkstra	432	1.728	1,69 KB
S2	A*	612	2.448	2,40 KB
	BFS	468	1.872	1,83 KB
	Dijkstra	432	1.728	1,69 KB
S3	A*	612	2.448	2,40 KB
	BFS	468	1.872	1,83 KB
	Dijkstra	432	1.728	1,69 KB
S4	A*	1.088	4.352	4,25 KB
	BFS	832	3.328	3,25 KB
	Dijkstra	768	3.072	3,00 KB
S5	A*	2.448	9.792	9,57 KB
	BFS	1.872	7.488	7,31 KB
	Dijkstra	1.728	6.912	6,75 KB
S6	A*	4.352	17.408	17,00 KB
	BFS	3.328	13.312	13,00 KB
	Dijkstra	3.072	12.288	12,00 KB
S7	A*	8.228	32.912	32,15 KB
	BFS	6.292	25.168	24,58 KB
	Dijkstra	5.808	23.232	22,69 KB
S8	A*	11.492	45.968	44,90 KB
	BFS	8.788	35.152	34,33 KB
	Dijkstra	8.112	32.448	31,69 KB
S9	A*	15.300	61.200	59,77 KB
	BFS	11.700	46.800	45,71 KB
	Dijkstra	10.800	43.200	42,19 KB

Tabelle 6: Speicherverbrauch der Listen der CPU-Algorithmen.

Szenario	Algorithmus	GPU Launch-Infos			Speicherverbrauch (Bytes)		
					pro Agent	insgesamt	KB / MB
S0	A*	12	1	1	612	7.344	7,18 KB
	BFS	12	1	1	468	5.616	5,49 KB
	Dijkstra	12	1	1	432	5.184	5,07 KB
S1	A*	48	1	1	612	29.376	28,69 KB
	BFS	48	1	1	468	22.464	21,94 KB
	Dijkstra	48	1	1	432	20.736	20,25 KB
S2	A*	192	1	1	612	117.504	114,75 KB
	BFS	192	1	1	468	89.856	87,75 KB
	Dijkstra	192	1	1	432	82.944	81,00 KB
S3	A*	768	1	1	612	470.016	459,00 KB
	BFS	768	1	1	468	359.424	351,00 KB
	Dijkstra	768	1	1	432	331.776	324,00 KB
S4	A*	768	4	1	1.088	3.342.336	3,19 MB
	BFS	768	4	1	832	2.555.904	2,44 MB
	Dijkstra	768	4	1	768	2.359.296	2,25 MB
S5	A*	768	16	1	2.448	30.081.024	28,69 MB
	BFS	768	16	1	1.872	23.003.136	21,94 MB
	Dijkstra	768	16	1	1.728	21.233.664	20,25 MB
S6	A*	768	64	1	4.352	213.909.504	204,00 MB
	BFS	768	64	1	3.328	163.577.856	156,00 MB
	Dijkstra	768	64	1	3.072	150.994.944	144,00 MB
S7	A*	768	149	2	8.228	941.546.496	897,93 MB
	BFS	768	195	2	6.292	942.289.920	898,64 MB
	Dijkstra	768	211	2	5.808	941.174.784	897,58 MB
S8	A*	768	106	5	11.492	935.540.736	892,21 MB
	BFS	768	139	4	8.788	938.136.576	894,68 MB
	Dijkstra	768	151	4	8.112	940.732.416	897,16 MB
S9	A*	768	80	13	15.300	940.032.000	896,49 MB
	BFS	768	105	10	11.700	943.488.000	899,79 MB
	Dijkstra	768	113	10	10.800	937.267.200	893,85 MB

Tabelle 7: Speicherverbrauch der Listen der GPU-Algorithmen inklusive Launch-Infos.

Szenario	Algorithmus	GPU Launch-Infos			SMEM pro Block (Bytes)
S0	A*	12	1	1	624
	BFS	12	1	1	480
	Dijkstra	12	1	1	444
S1	A*	48	1	1	624
	BFS	48	1	1	480
	Dijkstra	48	1	1	444
S2	A*	192	1	1	624
	BFS	192	1	1	480
	Dijkstra	192	1	1	444
S3	A*	768	1	1	624
	BFS	768	1	1	480
	Dijkstra	768	1	1	444
S4	A*	768	4	1	1.100
	BFS	768	4	1	844
	Dijkstra	768	4	1	780
S5	A*	768	16	1	2.460
	BFS	768	16	1	1.884
	Dijkstra	768	16	1	1.740
S6	A*	768	64	1	4.364
	BFS	768	64	1	3.340
	Dijkstra	768	64	1	3.084
S7	A*	768	256	1	8.240
	BFS	768	256	1	6.304
	Dijkstra	768	256	1	5.820
S8	A*	768	512	1	11.504
	BFS	768	512	1	8.800
	Dijkstra	768	512	1	8.124
S9	A*	768	1.024	1	15.312
	BFS	768	1.024	1	11.712
	Dijkstra	768	1.024	1	10.812

Tabelle 8: Speicherverbrauch der GPU unter Verwendung der „Shared Memory“.

In Tabelle 9, Tabelle 10 und Tabelle 11 werden die Laufzeiten der Algorithmen in den einzelnen Technologien, genannt CPU, CUDA und OpenCL, mit den jeweiligen zehn Szenarien S0 bis S9 präsentiert. Dabei wird pro Technologie ein Wert in Millisekunden angegeben, welcher die durchschnittliche Laufzeit dieses Algorithmus für einen Durchlauf für alle Agenten repräsentiert. Diese Werte entsprechen der Summe der vom Framework berechneten Laufzeiten „Init“, „Run“ und „DeInit“ welche bei jedem Durchlauf der Pfadplanung erhoben werden, wie auch in der Bedienungsanleitung des Frameworks im Anhang A dieser Arbeit erläutert.

In Tabelle 9 werden die Ergebnisse der Laufzeiten der Kategorie „zehn Szenarien“, welche im letzten Kapitel erläutert und spezifiziert wurden, aufgezeigt. In Tabelle 10 hingegen sind die Laufzeitergebnisse der Kategorie „zehn Szenarien mit Scenario Multiplier“ aufgelistet. Bei der in Tabelle 11 dargestellten Testergebnissen der Kategorie „zehn Szenarien mit „Shared Memory“ werden die Laufzeitergebnisse der GPU-Algorithmen CUDA und OpenCL aufgezeigt. Die Werte der CPU, welche in Tabelle 11 dargeboten werden, entsprechen den Werten der CPU-Testläufe mit „Scenario Multiplier“ aus Tabelle 10. Diese Werte der CPU werden mit den GPU-Algorithmen unter Verwendung der „Shared Memory“ verglichen, so wie auch bei der Analyse der Laufzeittests, welche nun im folgenden Kapitel vorgestellt wird.

Szenario	Algorithmus	durchschn. Laufzeit (ms)		
		CPU	CUDA	OpenCL
S0	A*	0,04	0,67	1,04
	BFS	0,05	0,55	0,97
	Dijkstra	0,07	1,06	2,07
S1	A*	0,11	1,14	1,93
	BFS	0,09	0,87	1,54
	Dijkstra	0,21	1,75	4,44
S2	A*	0,21	1,36	2,16
	BFS	0,16	1,01	1,72
	Dijkstra	0,39	2,01	4,92
S3	A*	0,81	2,28	2,87
	BFS	0,57	1,96	2,41
	Dijkstra	1,36	4,46	7,04
S4	A*	4,31	4,68	5,96
	BFS	3,14	3,87	4,57
	Dijkstra	10,33	10,70	19,33
S5	A*	31,19	33,36	36,70
	BFS	20,73	25,99	27,82
	Dijkstra	120,55	113,55	157,03
S6	A*	176,67	236,05	224,88
	BFS	124,64	215,95	188,67
	Dijkstra	789,66	1.224,48	1.240,87
S7	A*	1.102,44	13.54,63	1.490,56
	BFS	785,86	11.69,55	1.217,19
	Dijkstra	6.938,57	9.518,72	11.118,59
S8	A*	2.802,12	3.693,56	4.058,54
	BFS	1.965,97	3.306,29	3.416,49
	Dijkstra	20.253,37	31.434,42	36.133,72
S9	A*	7.095,06	10.030,11	10.807,97
	BFS	4.955,13	8.761,40	9.120,66
	Dijkstra	57.762,82	97.236,04	113.392,06

Tabelle 9: Laufzeitergebnisse der Technologien mit den zehn Szenarien.

Szenario	Algorithmus	durchschn. Laufzeit (ms)		
		CPU	CUDA	OpenCL
S0	A*	0,04	0,38	0,52
	BFS	0,05	0,33	0,52
	Dijkstra	0,05	0,46	0,66
S1	A*	0,05	0,49	0,61
	BFS	0,06	0,41	0,57
	Dijkstra	0,07	0,66	0,84
S2	A*	0,10	0,52	0,67
	BFS	0,11	0,43	0,60
	Dijkstra	0,18	0,75	0,96
S3	A*	0,39	0,98	1,18
	BFS	0,25	0,78	0,98
	Dijkstra	0,54	2,00	2,21
S4	A*	5,21	6,38	6,57
	BFS	3,92	5,45	5,89
	Dijkstra	11,49	12,97	12,98
S5	A*	33,90	43,34	41,57
	BFS	21,55	32,11	32,22
	Dijkstra	114,88	153,77	148,83
S6	A*	178,80	399,37	402,19
	BFS	126,43	290,22	292,87
	Dijkstra	769,97	1.633,12	1.620,03
S7	A*	1.205,02	2.000,70	1.991,51
	BFS	816,81	1.661,09	1.663,52
	Dijkstra	7.104,32	12.041,72	11.912,87
S8	A*	2.771,88	5.342,66	5.350,75
	BFS	1.978,13	4.508,13	4.509,88
	Dijkstra	20.536,30	40.136,57	39.727,48
S9	A*	6.859,68	13.741,14	13.703,60
	BFS	4.844,20	11.807,09	11.771,40
	Dijkstra	57.468,68	121.889,18	120.735,19

Tabelle 10: Laufzeitergebnisse der Technologien mit dem „Scenario Multiplier“.

Szenario	Algorithmus	durchschn. Laufzeit (ms)		
		CPU	CUDA	OpenCL
S0	A*	0,04	0,28	0,44
	BFS	0,05	0,29	0,41
	Dijkstra	0,05	0,35	0,53
S1	A*	0,05	0,27	0,48
	BFS	0,06	0,28	0,46
	Dijkstra	0,07	0,33	0,48
S2	A*	0,10	0,28	0,43
	BFS	0,11	0,27	0,44
	Dijkstra	0,18	0,35	0,54
S3	A*	0,39	0,31	0,51
	BFS	0,25	0,30	0,46
	Dijkstra	0,54	0,36	0,53
S4	A*	5,21	0,67	1,15
	BFS	3,92	0,54	0,68
	Dijkstra	11,49	0,99	1,22
S5	A*	33,90	1,77	2,10
	BFS	21,55	1,22	1,43
	Dijkstra	114,88	4,83	5,51
S6	A*	178,80	8,02	8,68
	BFS	126,43	5,04	5,28
	Dijkstra	769,97	26,54	29,87
S7	A*	1.205,02	51,35	54,82
	BFS	816,81	31,49	32,01
	Dijkstra	7.104,32	218,21	244,89
S8	A*	2.771,88	123,56	130,84
	BFS	1.978,13	75,52	75,90
	Dijkstra	20.536,30	656,25	728,72
S9	A*	6.859,68	299,20	316,46
	BFS	4.844,20	182,64	182,77
	Dijkstra	57.468,68	1.861,32	2.075,84

Tabelle 11: Laufzeitergebnisse der GPU unter Verwendung der „Shared Memory“.

6.4 Analyse der Laufzeittests

Mit Hilfe der Laufzeitergebnisse und Berechnungen des Speicherverbrauchs der einzelnen Testläufe im vorigen Abschnitt werden in den nun folgenden Abschnitten Analysen zu diesen Zahlen gegeben und Schlussfolgerungen gezogen. Hierbei wird im folgenden Abschnitt ein Vergleich der drei Algorithmen BFS, Dijkstra und A* gegeben und versucht, die in Kapitel 4.3 theoretisch erhobenen Eigenschaften auch in der Praxis zu bestätigen. Danach folgt ein Vergleich der Technologien, welcher die Ergebnisse und Erkenntnisse der Vorarbeiten, darunter die Arbeit von Bleiweiss [4] und Karimi et al. [8], bestätigen soll.

6.4.1 Laufzeitvergleich der Algorithmen

Für den Vergleich der Algorithmen werden die Daten der Laufzeiten der „zehn Szenarien“, zu sehen in Tabelle 9, verwendet. Diese sind allgemein repräsentativ und aussagekräftig für die allgemein gültigen Laufzeitverhalten der Algorithmen. Zudem produzierten die Testläufe der anderen Kategorien annähernd dieselben Ergebnisse.

Der Speicherverbrauch der Listen der Algorithmen pro Agent ändert sich proportional zur Größe der Welt, wie auch im Diagramm in Abbildung 29 erkannt werden kann. So besitzen die ersten Szenarien S0 bis S4 denselben Speicherverbrauch, da diese, wie in deren Spezifikation im Kapitel 6.2.2 in Tabelle 4 ersichtlich, die Weltausmaße von 6x6 Knoten besitzen. Bei steigender Weltgröße steigt auch der Speicherverbrauch der Listen, wobei, wie in Abbildung 29 zu erkennen, der Dijkstra-Algorithmus den geringsten Speicherverbrauch besitzt, da dieser nur eine „Priority Queue“ und ein „hashed Array“ benötigt, wie im Pseudocode des Algorithmus im Kapitel 4.2.3 in Abbildung 19 unter den Bezeichnungen `Queue` und `f` gesehen werden kann.

Der Algorithmus mit den nächsthöheren Speicherverbrauch ist der „Best First Search“ (BFS), welcher zusätzlich zu den Listen des Dijkstra-Algorithmus auch eine „Closed-Liste“ hält, welche im Pseudocode des Algorithmus im Kapitel 4.2.2 in Abbildung 16 unter der Bezeichnung `Closed` gesehen werden kann. Dieser besteht aus booleschen Werten, welche nur ein Byte pro Wert benötigen, wodurch der zusätzliche Speicherverbrauch gegenüber dem Dijkstra-Algorithmus mit 6% relativ gering ausfällt, wie in Abbildung 30 beim Überblick über den Speicherverbrauch der Algorithmen zu erkennen.

Der zusätzliche Speicherverbrauch des A* liegt am zusätzlichen „hashed Array“, im Pseudocode des Algorithmus im Kapitel 4.2.4 in Abbildung 21 zu sehen, mit der Bezeichnung `kostenVonStartZu`, deren gespeicherten Gleitkommawerte eine Größe von vier Bytes besitzen. Deswegen ist auch der Speicheranstieg des A* von 24% auch vier Mal größer als der des BFS von nur 6%, wie auch im Vergleich des Speicherverbrauchs in Abbildung 30 erkannt werden kann.

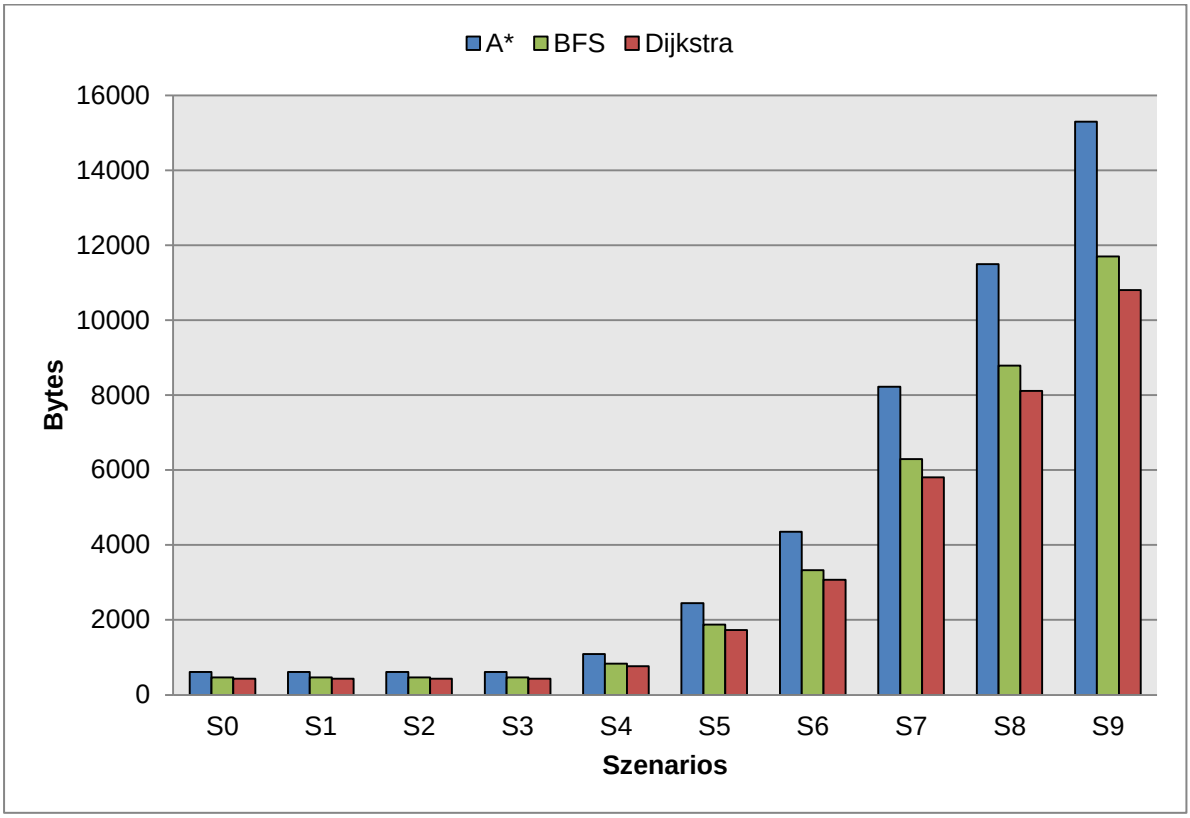


Abbildung 29: Speicherverbrauch der Algorithmen pro Agent bei „zehn Szenarien“.

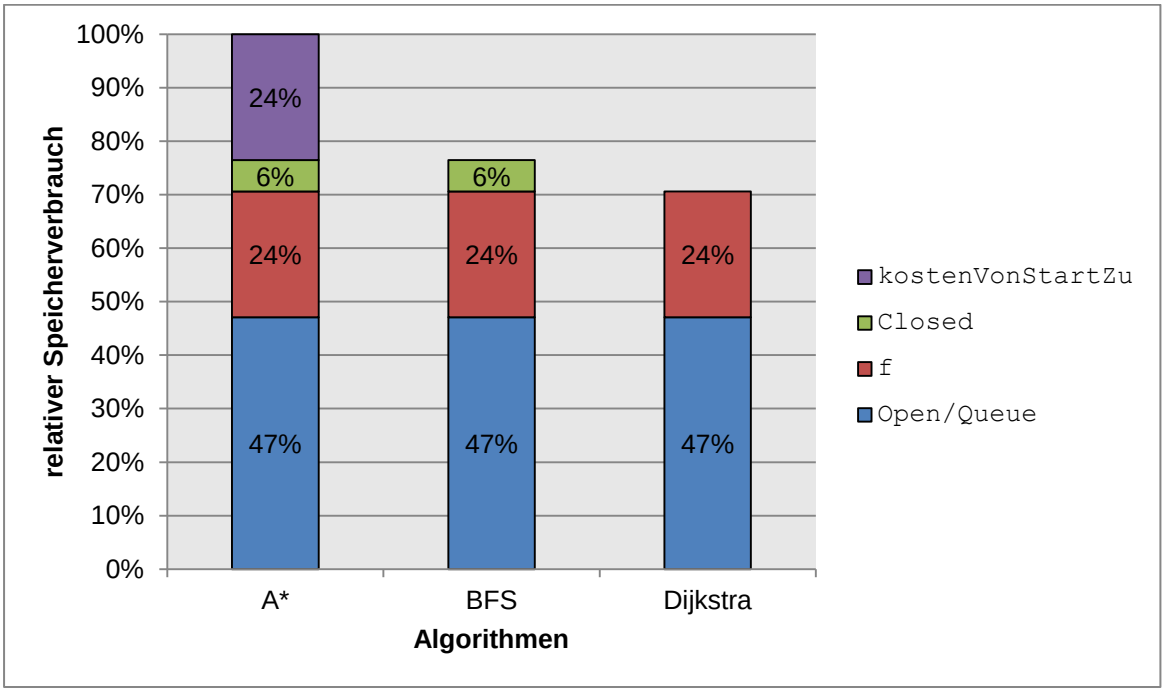


Abbildung 30: Überblick über den Speicherverbrauch der Listen der Algorithmen.

Im Punkto Laufzeit ist der Algorithmus „Best First Search“ (BFS) durchgehend der schnellste, wie auch in Abbildung 31 am grünen Graphen gesehen werden kann. Dieser wird dicht gefolgt vom A*-Algorithmus, dargestellt durch den blauen Graphen. Der Dijkstra-Algorithmus, dargestellt mit dem roten Graphen, ist im Punkto Laufzeit von diesen Algorithmen eindeutig abgeschlagen, wie auch im Diagramm zur Visualisierung der Laufzeittests in Abbildung 31 zu erkennen.

Die theoretischen Laufzeiten der Algorithmen wurden in dieser Arbeit im Kapitel 4.3 zusammen mit anderen Eigenschaften angegeben. Dabei ist zu erkennen, dass sich die im Kapitel 4.3 getätigten Aussagen nicht zu hundert Prozent mit den Laufzeitergebnissen übereinstimmen. So ist der BFS bei den Laufzeitergebnissen auf den ersten Rang, wobei der A* laut den theoretischen Laufzeiten aus Kapitel 4.3 schneller sein sollte. Dies ist aber bei den Laufzeitergebnissen genau anders herum der Fall, da die zu rechnenden Szenarien S0 bis S9 keine Hindernisse beinhalten und somit das „Best Case“-Szenario des BFS, so wie es im Kapitel 4.2.2 in Abbildung 17 (b) dargestellt ist, während der Laufzeittests immer vorliegt. Zudem ist der BFS bei diesen Szenarien auch immer in der Lage den kürzesten Pfad zwischen zwei Knoten zu finden, was nicht immer der Fall sein muss, so wie auch im Kapitel 4.3 erläutert.

Der A*-Algorithmus ist hingegen immer in der Lage sehr schnell den kürzesten Pfad zwischen zwei Knoten zu finden, wobei dies natürlich, wie auch beim BFS, eine für die Suche akzeptable Heuristik voraussetzt. Dies trifft bei den in Kapitel 5.1.2 vorgestellten und während der Laufzeittests verwendeten Heuristiken „Manhattan Distanz“ und „diagonale Distanz“ zu. Deswegen ist der BFS-Algorithmus immer, so wie auch im Diagramm in Abbildung 31 zu sehen, zwischen 10% und 40% schneller ist als der A*, im Durschnitt aber ca. 25%.

Die Laufzeit des Dijkstra-Algorithmus hingegen verhält sich so wie im Kapitel 4.3 erläutert und ist langsamer als der A* bzw. der BFS. Dies liegt eindeutig am Suchverhalten dieses Algorithmus, welcher im Kapitel 4.2.3 vorgestellt sowie in Abbildung 18 bildlich dargestellt wird. Da das Diagramm in Abbildung 31 aber logarithmisch skaliert ist, ist der Unterschied der Laufzeiten des Dijkstra am Ende des Graphen noch größer als auf den ersten Blick wahrnehmbar. Der Unterschied der Laufzeiten kann besser in Abbildung 32 gesehen werden, welche den Beschleunigungszuwachs bzw. den so genannten „Speed-Up“ der Algorithmen relativ zum langsamsten Algorithmus, dem Dijkstra, aufzeigen.

Wie in Abbildung 32 zu erkennen sind der BFS und der A* zu Beginn bei Szenario S0 bis um das Doppelte schneller als der Dijkstra. Dieser Unterschied steigt aber ab S4 an, da bei diesem Szenario, so wie auch in Tabelle 4 spezifiziert, die Weltgröße zum ersten Mal steigt, was die in alle Richtungen gleichmäßig ausbreitende Suche des Dijkstra, so wie im Suchverhalten des Algorithmus in Abbildung 18 erkennbar, um ein Vielfaches verlängert,

sodass diese beim letzten Szenario S9 bis um das Neun- bzw. Zwölffache länger dauert als die Suche des BFS bzw. des A*, wie auch in den Ergebnissen der Laufzeittests des Szenarios S9 in Tabelle 9 gesehen werden kann.

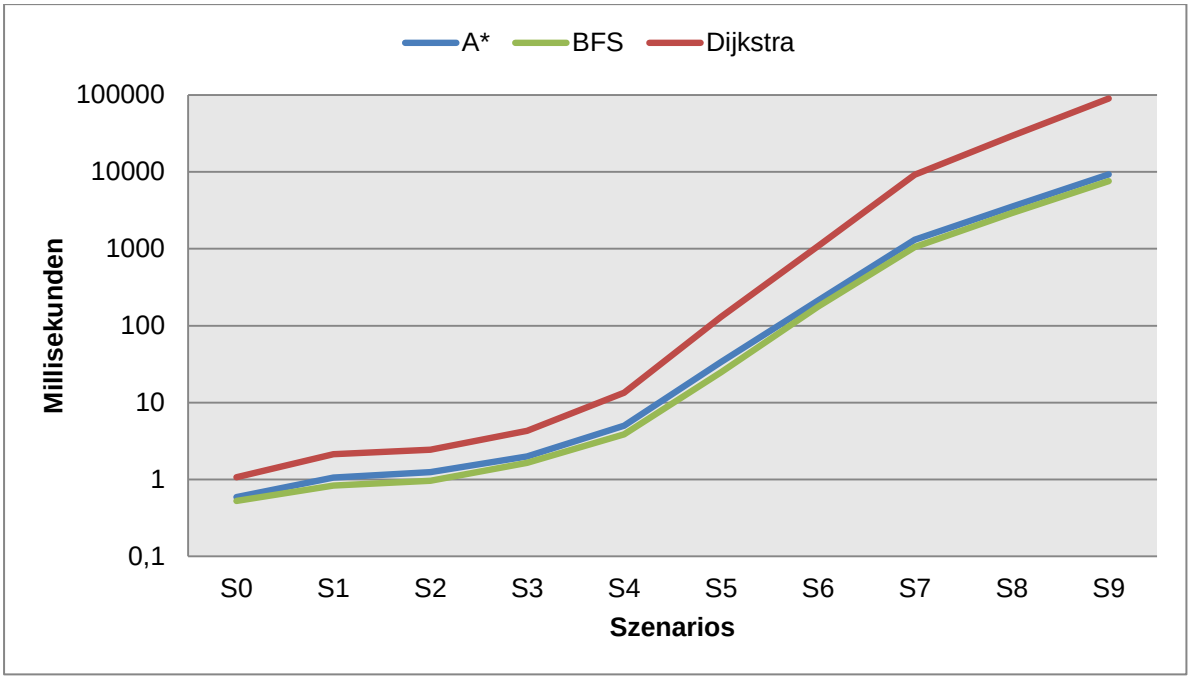


Abbildung 31: Laufzeitvergleich der Algorithmen (logarithmisch skaliert).

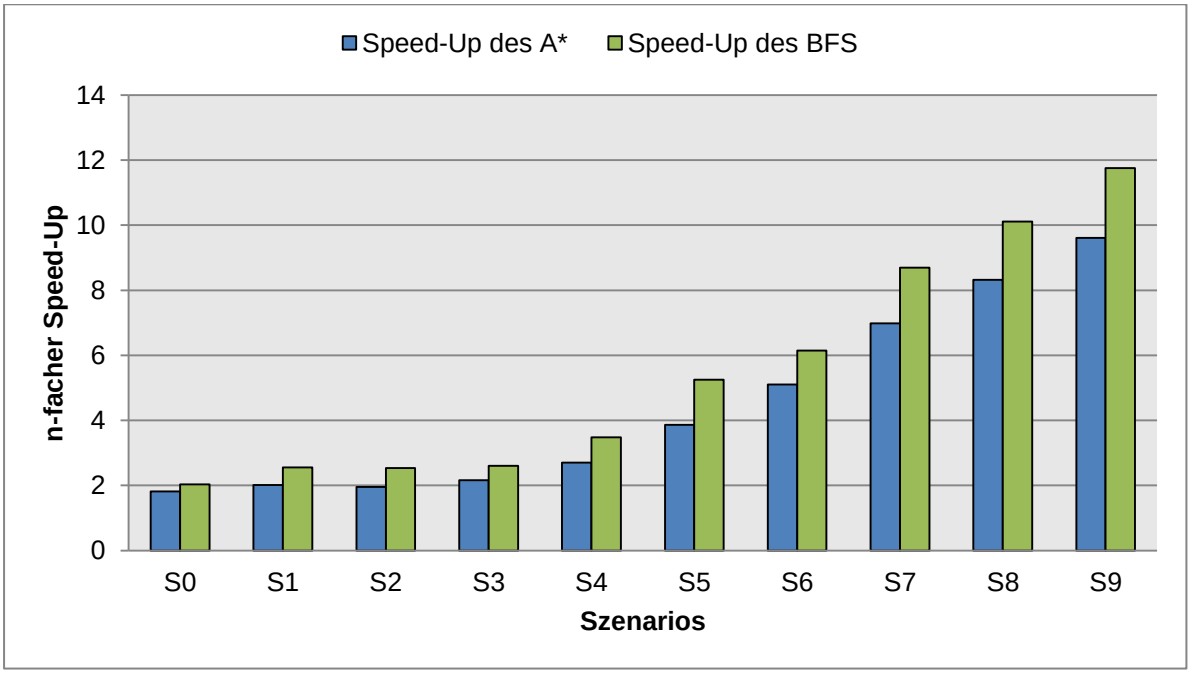


Abbildung 32: Laufzeitbeschleunigung des A* und des BFS relativ zum Dijkstra.

6.4.2 Laufzeitvergleich der Technologien

Nachdem im letzten Kapitel die Algorithmen näher betrachtet und analysiert wurden, werden nun die drei Technologien CPU, CUDA und OpenCL untereinander verglichen und deren Laufzeitunterscheide analysiert. Zuerst wird aber der Speicherverbrauch der Technologien betrachtet, welcher, wie schon beim Vergleich der CPU-Algorithmen im Kapitel 6.3 in Tabelle 6 mit den Werten der GPU-Algorithmen aus Tabelle 7 zu erkennen, sehr unterschiedlich ist. Die GPU benötigt, da sie viele Threads gleichzeitig bzw. parallel durchführt, mehr Speicher pro Szenario als die CPU mit ihren wenigen vier Threads. Dies ist auch sehr gut im Diagramm in Abbildung 33 zu erkennen, in welcher der Speicherverbrauch der CPU aus Tabelle 6 und der GPU aus Tabelle 7 bildlich dargestellt wird, logarithmisch skaliert.

Außerdem kann sehr gut erkannt werden, dass die GPU-Algorithmen ab Szenario S7 nicht mehr Speicher verbrauchen als 900 MB, wie durch die in Kapitel 6.2.1 erwähnte Verwendung des Programmarguments „GPUforceMemUsage“ gewünscht. Dafür steigt, wie auch in Abbildung 33 auf der Sekundärachse zu erkennen, die Anzahl der „Launches“, also eines Berechnungsdurchgangs mit einer Teilmenge der Agenten auf der GPU, an, um die Restriktion des globalen Arbeitsspeichers der GPU zu umgehen. Bis zum Szenario S6 wird immer nur ein „Launch“ benötigt, wie auch in Tabelle 7 ersichtlich. Das Szenario S7 benötigt zum ersten Mal für alle Algorithmen zwei „Launches“, Szenario S8 schon vier für BFS und Dijkstra bzw. fünf für A* und Szenario S9 schon ganze zehn für BFS und Dijkstra bzw. dreizehn für A*, wie auch in Tabelle 7 nachlesbar.

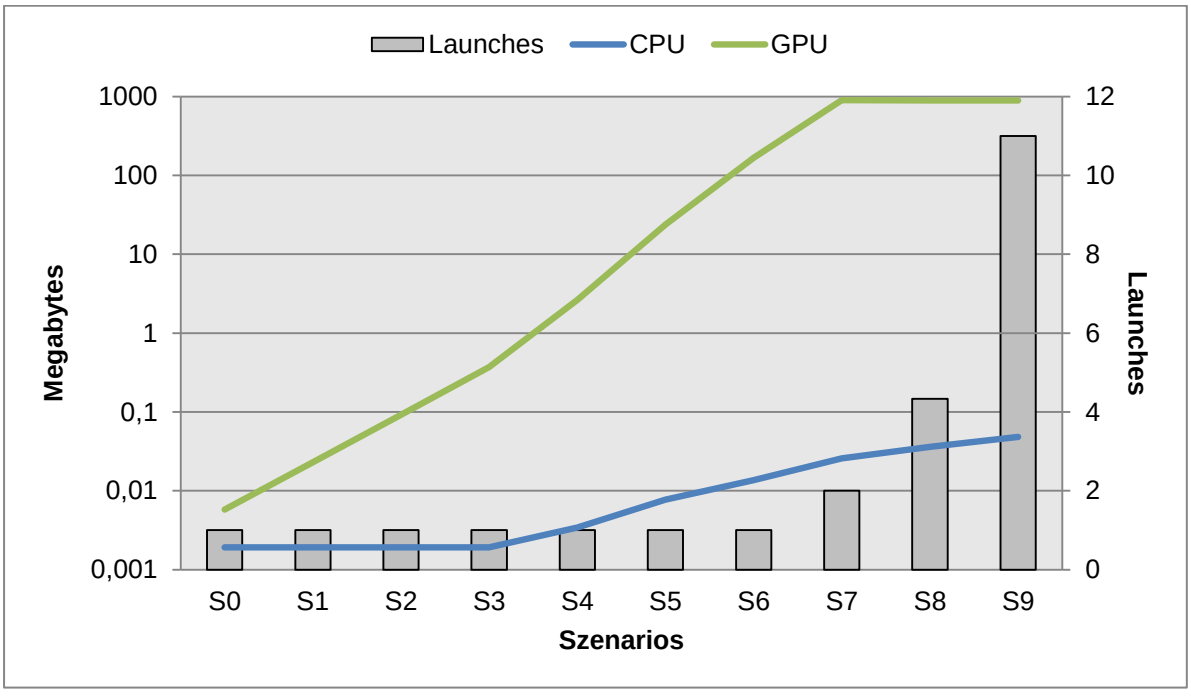


Abbildung 33: Speicherverbrauch der CPU und GPU (logarithmisch skaliert).

Die durchschnittlichen Laufzeiten für einen Berechnungsdurchlauf der Algorithmen fallen bei den Technologien, wie auch im Kapitel 6.3 in Tabelle 9 nachlesbar, bei der Kategorie „zehn Szenarien“ zugunsten der CPU aus. Die CPU besitzt bei allen Szenarien einen Zeitvorteil gegenüber der GPU mit CUDA oder OpenCL. Die Ergebnisse aus dieser Tabelle werden im Diagramm in Abbildung 34 grafisch dargestellt, wodurch der Vorteil der CPU gegenüber den GPU-Technologien ersichtlich wird, da der blaue Graph der CPU immer unterhalb des grünen und roten Graphen der GPU-Technologien CUDA und. OpenCL liegt.

Das Diagramm in Abbildung 34 ist logarithmisch skaliert, wodurch der Laufzeitunterscheid zwischen den Technologien optisch nicht immer richtig proportioniert ist. Eine bessere Darstellung der Laufzeitunterschiede kann in Abbildung 35 gesehen werden, welches den Beschleunigungszuwachs bzw. den „Speed-Up“ der CPU gegenüber der GPU beschreibt. Dabei ist zu erkennen, dass der Unterschied zu Beginn bei wenigen Agenten, so wie erwartet, am größten ist, da der Overhead der API zum Starten des Kernels mit Kopieren der Daten vom Host zum Device und wieder zurück größer ist als die Berechnung des Algorithmus durch die CPU selbst. Da diese Kopierfunktionen aber sehr konstant sind und bei den größeren Szenarien weniger als 1% der Laufzeit ausmachen, so wie auch bei Karimi et al. [8] beobachtet, verringert sich der „Speed-Up“ der CPU solange bis sie bei S9 nur mehr etwas mehr als das Eineinhalbfache schneller ist als die GPU, im Gegensatz zum knapp achtzehnfachen „Speed-Up“ zu Beginn bei S0.

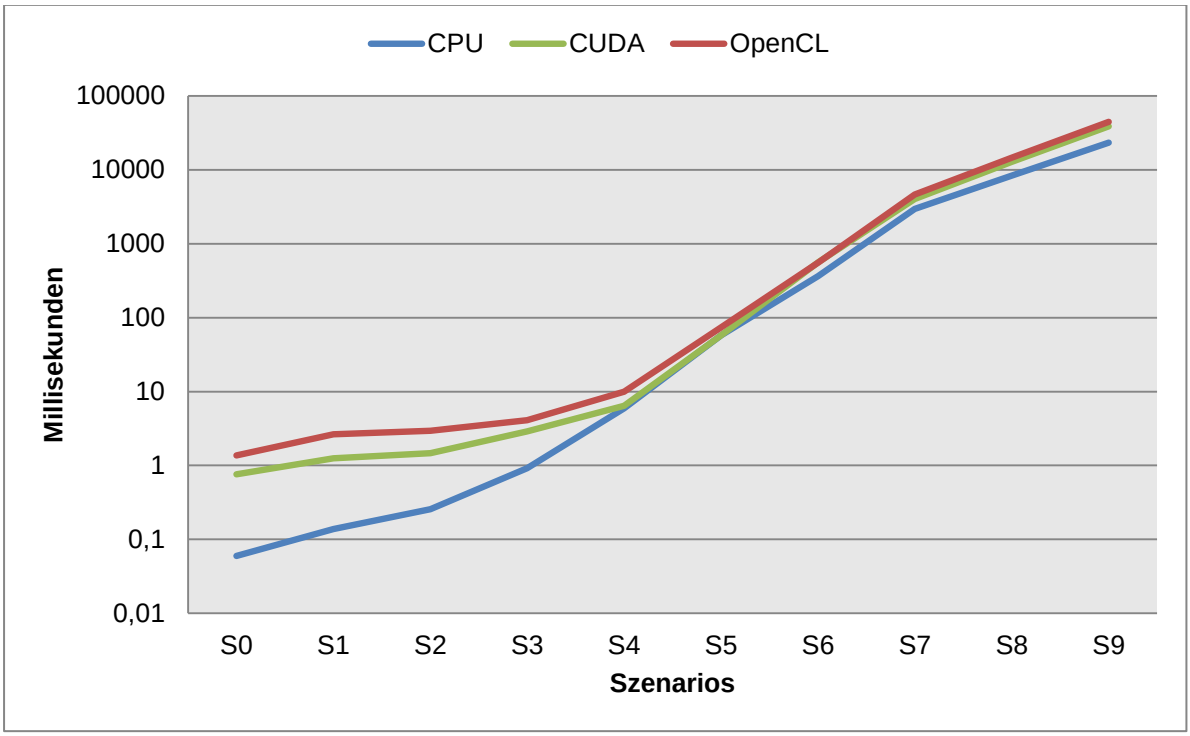


Abbildung 34: Laufzeitvergleich der Technologien (logarithmisch skaliert).

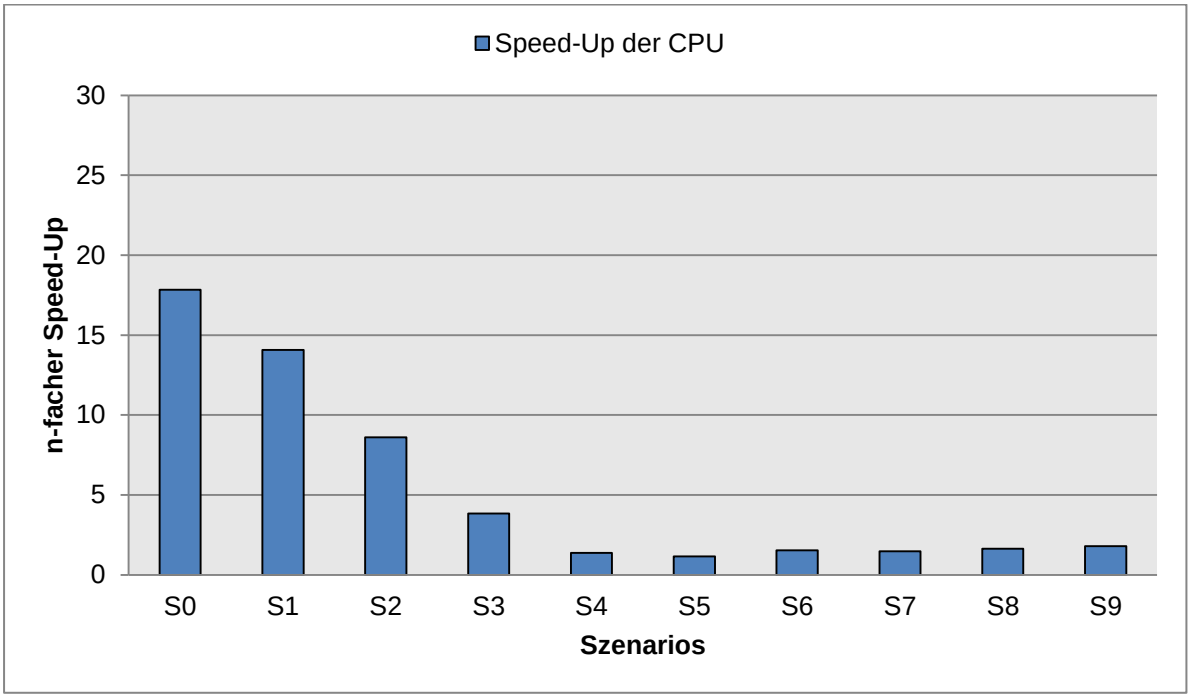


Abbildung 35: Beschleunigungszuwachs der CPU gegenüber der GPU.

Bei der Darstellung der Laufzeitergebnisse der Kategorie „zehn Szenaren mit Szenario Multiplier“ in Abbildung 36, deren Daten in Tabelle 10 präsentiert werden, ist zu erkennen, dass so wie in der vorhergehenden Kategorie die CPU die Oberhand behält und schneller ist als die GPU-Algorithmen. Die Laufzeitergebnisse der CPU-Algorithmen ähneln, wie zu erwarten war, der aus Abbildung 34. Nur bei den Laufzeiten der GPU-Algorithmen in Abbildung 36 kann zu Beginn bei den ersten drei Szenarien S0 bis S3 erkannt werden, dass diese schneller durchgeführt werden als in Abbildung 34, wo sie länger als eine Millisekunde benötigen. In Abbildung 36 benötigen diese immer unter eine Millisekunde.

Der „Speed-Up“ der CPU über die GPU wird für diese Kategorie mit „Scenario Multiplier“ in Abbildung 37 dargestellt. Dabei ist zu erkennen, wie zuvor bemerkt, dass der Beschleunigungszuwachs bzw. „Speed-Up“ der CPU zu Beginn bei den Szenarien S0 bis S3 geringer ausfällt als zuvor in Abbildung 35. Nun beträgt der „Speed-Up“ der CPU zu Beginn nur mehr das Neun- bis Zehnfache der GPU, im Unterschied zum achtzehnfachen „Speed-Up“ der CPU zuvor in Abbildung 35.

Der Grund für dieses schlechte Ergebnis, obwohl alle Blocks in CUDA bzw. Work-Groups in OpenCL bei dieser Kategorie dasselbe rechnen und somit auch dieselben Codepfade ausführen, wird wahrscheinlich im ungeordneten Speicherzugriff der Threads zu finden sein. Da alle Threads das gleiche rechnen greifen sie auch zur gleichen Zeit auf dieselben Adressen der „global Memory“ zu. Das Ergebnis ist besser als zuvor, aber noch immer weit entfernt von einer Beschleunigung der Suche seitens der GPU.

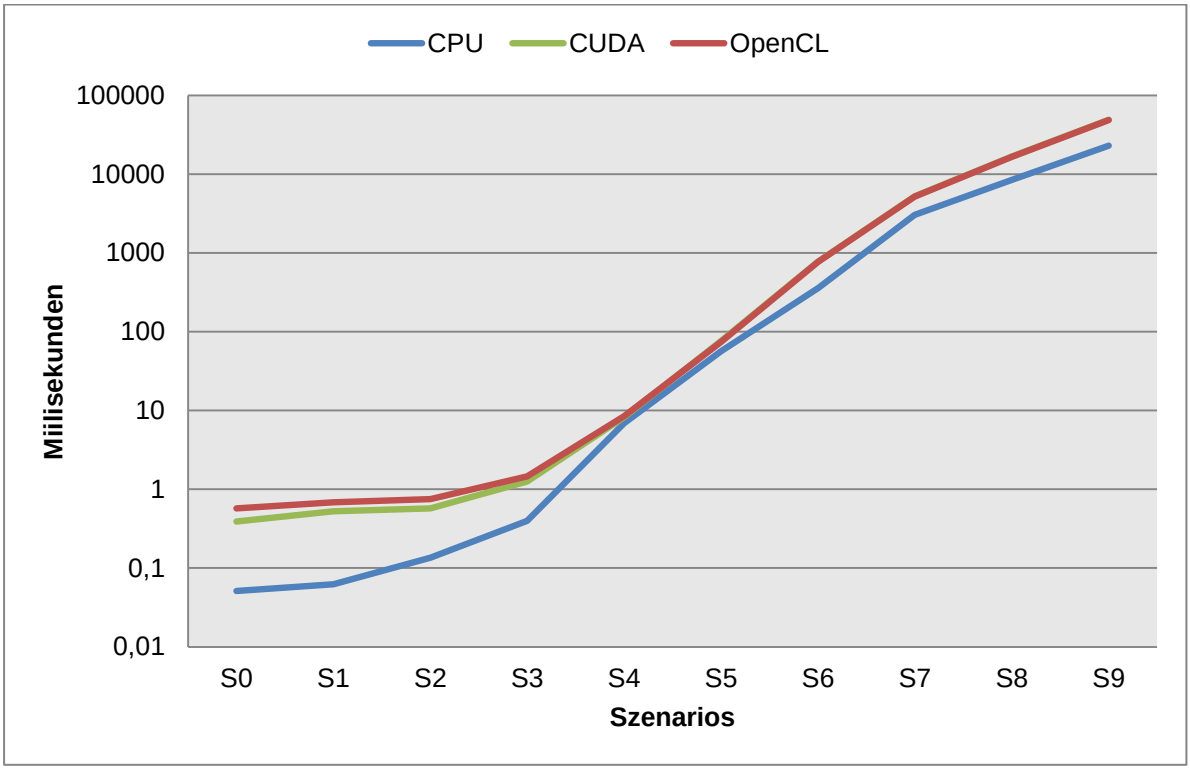


Abbildung 36: Laufzeitvergleich mit „Scenario Multiplier“ (logarithmisch skaliert).

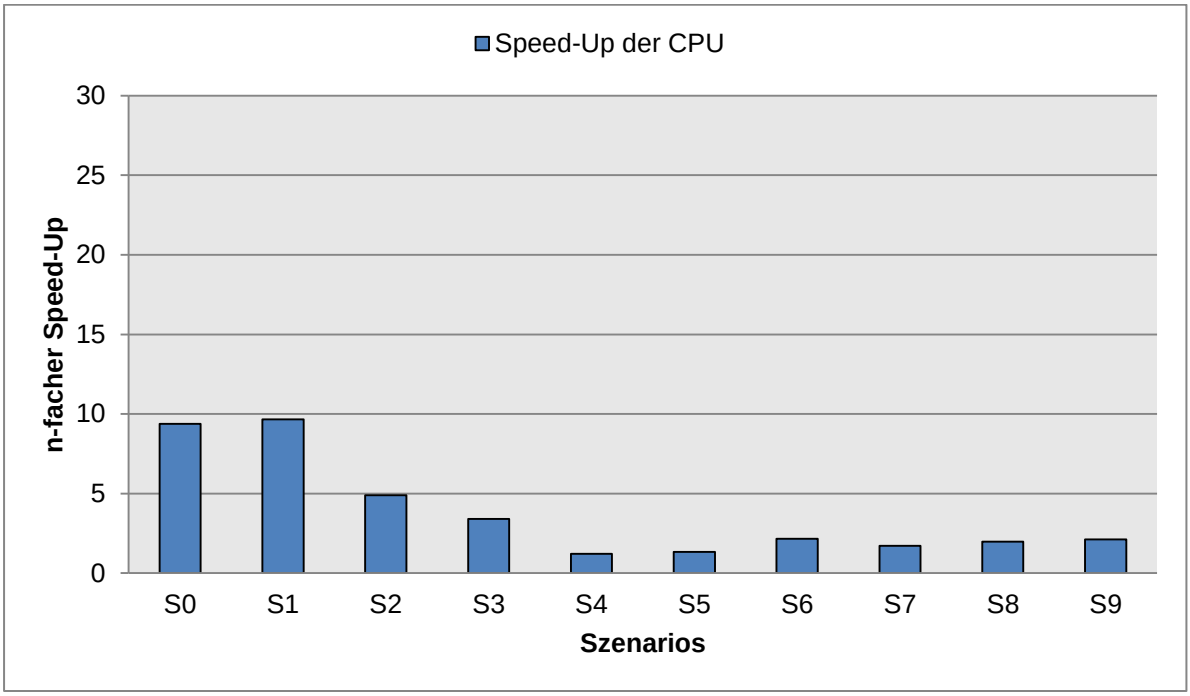


Abbildung 37: Beschleunigungszuwachs der CPU mit „Scenario Multiplier“.

Die „global Memory“ der GPU, welche bei der „Nvidia GeForce GTX 460“ ein GDDR5-RAM ist, ist allgemein schneller als der Arbeitsspeicher der CPU, welcher im Testsystem ein DDR2-800-RAM ist. Trotzdem sind die Latenzen der GPU beim Speicherzugriff höher, vor allem da die CPU einen größeren Cache sowie auch eine bessere Logik beim Caching besitzt, so wie auch bei Glaskowsky [20] erläutert und im Kapitel 3.3 beim Vergleich der CPU und der GPU beschrieben.

Deswegen besitzt die GPU einen eigenen, sehr schnellen Speicherbereich, bei CUDA „Shared Memory“ und bei OpenCL „Local Memory“ genannt. Dieser befindet sich, wie schon des Öfteren erwähnt, auf dem Chip der GPU und ist um ein Vielfaches schneller als die „global Memory“, wie auch im „CUDA C Programming Guide“ [2] erläutert. Auch die Testergebnisse der Kategorie „zehn Szenarien mit Shared Memory“ in Tabelle 11 lassen einen eindeutigen Zuwachs an Performanz seitens der GPU-Algorithmen erkennen. Im Diagramm der Laufzeiten dieser Kategorie in Abbildung 38 wird diese Tatsache bildlich dargestellt. Es ist zu erkennen, dass die CPU, wie angenommen, bei den kleineren Szenarien immer noch im Vorteil gegenüber der GPU-Technologien ist. Ab dem Szenario S3 ziehen diese aber mit der CPU gleich und werden bei den nachfolgenden Szenarien mit steigender Weltgröße und Agentenanzahl um ein Vielfaches schneller als die CPU.

In Abbildung 39 wird der Laufzeitunterschied bzw. der „Speed-Up“ der CPU und GPU-Algorithmen nochmals besser dargestellt. So besitzt die CPU zu Beginn einen Vorteil und ist siebeneinhalb Mal schneller als die GPU. Dieser „Speed-Up“ verringert sich aber bis hin zum Szenario S3, wo sie beinahe gleich schnell sind. Daraufhin überholt die GPU die CPU ab Szenario S4 und besitzt eine über siebenfachen „Speed-Up“, welcher sich noch weiter erhöht bis hin zum Szenario S7, wo sich der „Speed-Up“ einpendelt und bis zum Szenario S9 ca. dem Achtundzwanzigfachen der Laufzeit der CPU entspricht.

Dieses Ergebnis der Kategorie „zehn Szenarien mit „Shared Memory“ entspricht den Ergebnissen der Vorarbeiten, welche im Kapitel 2 dieser Arbeit vorgestellt wurden. Bleiweiss protokollierte hierbei auch in seiner ersten Arbeit [4] eine Laufzeitsteigerung der Pfadplanungsalgorithmen auf der GPU gegenüber der Algorithmen auf der CPU und erhielt ähnliche Ergebnisse mit seinen Messungen. Zudem ist beim Vergleich der Laufzeitergebnisse von CUDA und OpenCL zu erkennen, dass CUDA im Schnitt etwas schneller in der Durchführung der Berechnungen ist als OpenCL, so wie auch bei Karimi et al. [8] beim Vergleich dieser GPU-Computing APIs beobachtet und protokolliert wurde. Hierbei liegen die Vorteile bei CUDA sowohl in der Dauer der Kopierfunktionen zwischen dem Host und dem Device sowie auch in den Laufzeiten der Kernels selbst.

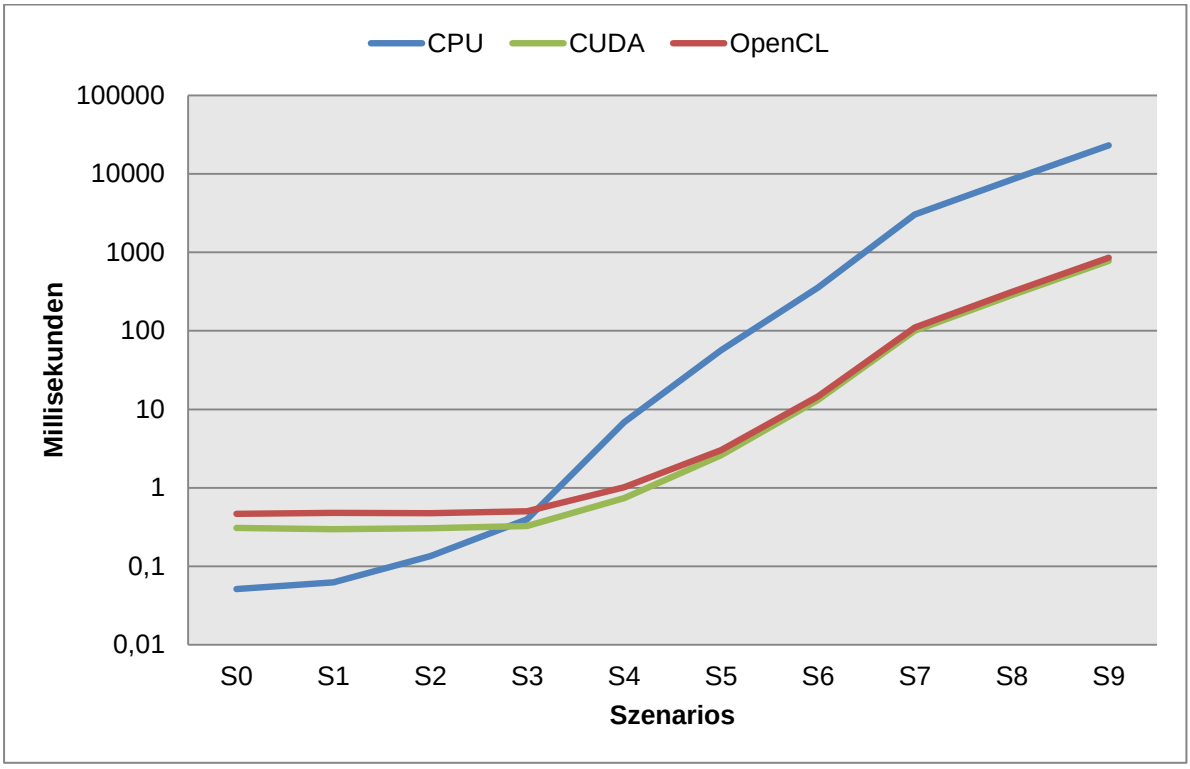


Abbildung 38: Laufzeitvergleich mit „Shared Memory“ (logarithmisch skaliert).

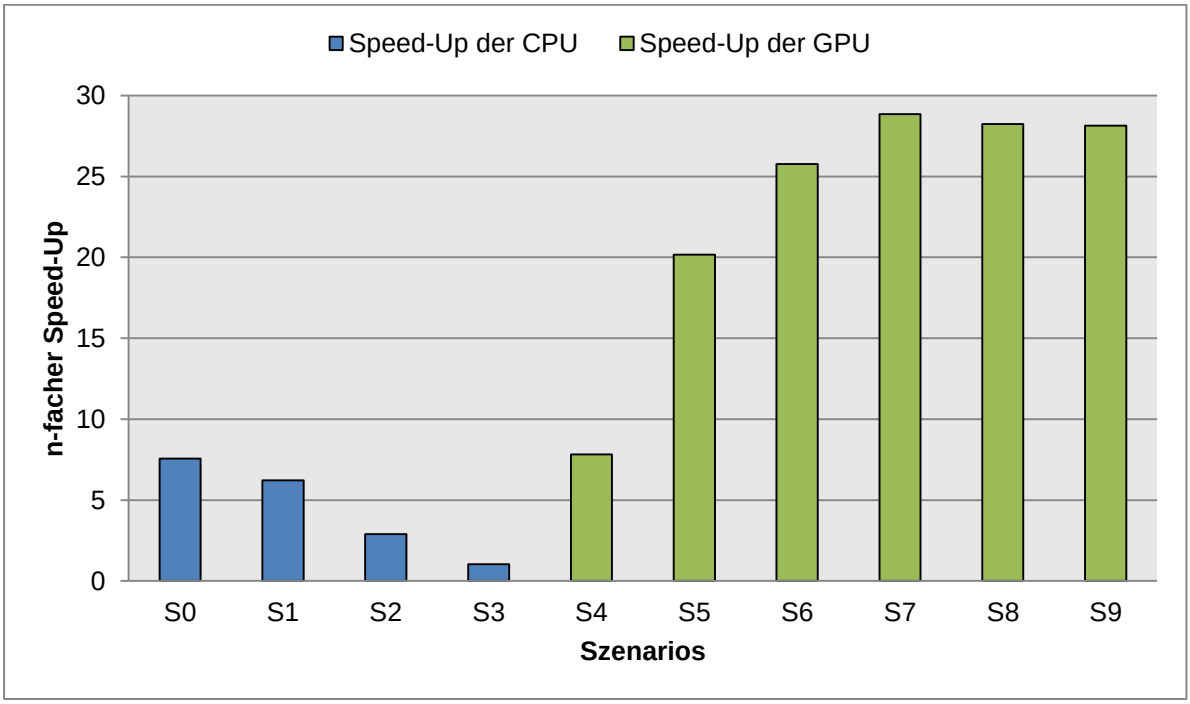


Abbildung 39: Beschleunigungszuwachs der CPU und GPU bei „Shared Memory“.

7 Diskussion

Ziel dieser Arbeit war die Implementierung und Vergleich von Pfadplanungsalgorithmen in unterschiedlichen Technologien, welche entweder die „Central Processing Unit“ (CPU) des Computersystems oder die „Graphics Processing Unit“ (GPU), also die Grafikkarte des Systems, für die Durchführung der Algorithmen verwendet. Die GPU ist eigentlich für grafikspezifische Aufgaben wie der Berechnung und Darstellung dreidimensionaler Daten entwickelt worden. Aufgrund der Fortschritte dieser Technologie in den letzten Jahren wurde auch die Implementierung von nicht grafikrelevanten Aufgaben auf der GPU ermöglicht.

Die Implementierung von nicht grafikrelevanten Aufgaben auf der GPU wird „General Purpose Computing on Graphics Processing Unit“ (kurz GPGPU) oder auch einfach GPU-Computing genannt. Dieses wurde von mehreren Autoren wie Bleiweiss [4] [5], Fischer et al. [6] und Shopf et al. [7] genutzt, um in deren Arbeiten, welche in Kapitel 2 „Analyse der Vorarbeiten“ präsentiert wurden, KI-spezifische Aufgaben wie die Pfadplanung von Agenten auf der GPU im Vergleich zu einer äquivalenten Implementierung für die CPU zu beschleunigen. Hierfür wurden für die Implementierung auf der GPU unterschiedliche Technologien verwendet. So wurde CUDA (Compute Unified Device Architecture) von Nvidia sowohl in den beiden Arbeiten von Bleiweiss [4] und [5] als auch von Fischer et al. [6] verwendet. Shopf et al. [7] verwendeten hingegen die von Microsoft entwickelte HLSL (High Level Shading Language) für ihre Agentensimulation. Eine weitere Technologie, welche für das GPU-Computing interessant ist, aber von keinen der Autoren verwendet wurde, ist der Open-Source Computing Standard namens OpenCL [3].

Diese Beschleunigung der GPU gegenüber der CPU konnte aufgrund der hoch parallel arbeitenden Hardwarearchitektur der GPUs erreicht werden, welche der so genannten SIMD-Architektur (Single Instruction – Multiple Data) entspricht. Diese wurde in dieser Arbeit im Kapitel 3 „Die Graphics Processing Unit“ genauer erläutert. Zudem wurde der Ursprung der GPUs und die „Rendering Pipeline“ mit ihren programmierbaren „Shader Stages“ vorgestellt, aus denen mittels der „Unified Shader“-Architektur das GPU-Computing entstand, so wie bei Akenine-Möller et al. [16] beschrieben. Zudem wurden die Hardwarearchitekturen der GPU und der CPU, wie bei Glaskowsky [20], verglichen.

Bei den Pfadplanungsalgorithmen, welche in einer eigenen Implementierung für die GPU in CUDA und OpenCL beschleunigt wurden, handelte es sich um den „Greedy Best First Search“ (BFS) [21], den Algorithmus von Dijkstra [22] und den A*-Algorithmus [23] (gesprochen A-Stern). Diese drei Algorithmen wurden im Kapitel 4 „Pfadplanung und Pfadfindung“ behandelt sowie die Definition von Graphen und Knoten gegeben, welche als Basis für die Pfadplanung dienen. Die Algorithmen wurden detailliert mit Pseudocodes und bildlicher Darstellung ihres Suchverhaltens vorgestellt und in späterer Folge verglichen.

Diese Algorithmen wurden sowohl für die CPU in C++ als auch für die GPU in CUDA und OpenCL implementiert. Für die Parallelisierung der Algorithmen in C++ wurde das OpenMP Framework [1] verwendet, welches die Berechnung der Pfadplanung auf die im System vorhandenen CPU-Kerne aufteilt. In CUDA und OpenCL wurden jeweils „Kernels“ entwickelt, welche die Pfadplanungsalgorithmen für einen Agenten implementieren. Die genaue Implementierung der Algorithmen innerhalb dieses selbst entwickelten Frameworks namens „Masterthesis_A_Star“, welches detaillierter im Anhang A dieser Arbeit vorgestellt wird, wurde im Kapitel 5 „Das Framework“ präsentiert und erläutert. Dabei wurden die Komponenten der Pfadplanung wie die Repräsentation des Graphen sowie die Erkennung der Nachbarschaften innerhalb dieser erläutert. Auch die Implementierung der Listen, welche von den Pfadplanungsalgorithmen benötigt werden, wie z.B. die „Priority Queue“ oder die Closed-Liste, wurde in diesem Kapitel vorgestellt.

Mit Hilfe des Wissens über die GPU und über die Pfadplanungsalgorithmen wurde das Framework entwickelt und Laufzeittests damit durchgeführt. Wie das von statten gegangen ist wurde im Kapitel 6 „Laufzeittests der Implementierungen“ präsentiert und kann auch im Anhang A dieser Arbeit genauer nachvollzogen werden. Die Szenarien, die von den Algorithmen berechnet wurden, wurden detailliert vorgestellt und deren Laufzeitergebnisse später in tabellarischer Form präsentiert. Diese Ergebnisse wurden analysiert und auch in Diagrammen und Graphen bildlich dargestellt. Zuerst wurden die Algorithmen verglichen und die Übereinstimmung der Ergebnisse mit den Aussagen aus Kapitel 4 „Pfadplanung und Pfadfindung“ geprüft.

Auch die Technologien wurden untereinander mit diesen Testläufen verglichen. Dabei wurde zu Beginn keine Beschleunigung der GPU gegenüber der CPU gemessen. Die CPU war in allen Szenarien schneller als die GPU, wobei der größte „Speed-Up“ der CPU bei den ersten Szenarien mit wenigen Agenten zu messen war und sich mit steigender Agentenanzahl und Weltgröße auch verringerte. Bei weiteren Durchläufen, bei denen die parallele Architektur noch besser ausgenutzt werden soll, konnten etwas bessere Ergebnisse seitens der GPU erzeugt werden, aber der Vorteil lag immer noch auf der Seite der CPU. Nur durch eine eigene Implementierung der Algorithmen, welche die „Shared Memory“ anstatt der „global Memory“ der GPU verwendet, konnte schlussendlich eine Beschleunigung der Algorithmen seitens der GPU erreicht werden.

Diese Implementierung birgt aber streng genommen kein gutes Ergebnis für den Vergleich der beiden Technologien, da bei der GPU-Implementierung quasi „geschummelt“ wurde und sich so sehr von der CPU-Implementierung unterscheidet, dass ein Vergleich nicht mehr fair wäre. Es wurde so lange zu den Vorteilen der GPU hingearbeitet, bis die Beschleunigung erreicht wurde. Der Arbeitsaufwand für die Implementierung der Pfadplanung für die GPU war auch im Unterschied zur Implementierung der CPU-Algorithmen um ein Vielfaches größer und komplexer.

Die CPU besitzt, dank des großen Cache und den komplexen Caching-Mechanismen im Punkto Laufzeit einen Vorteil gegenüber der GPU, welche im Vergleich weniger Cache besitzt und diesen auch viel weniger verwendet. Zudem kommt noch hinzu, wie auch bei Glaskowsky [20] erwähnt, dass die CPU eine „Brach-Prediction“ besitzt, welche die Laufzeit von Code beschleunigen kann, genauso wie die „Out-Of-Order-Execution“, welche die Speicherlatenzen der CPU zum Arbeitsspeicher übergehen bzw. retuschieren kann. All dies besitzt die GPU nicht, sodass Speicherzugriffe, die im Vergleich zur Codeausführung sehr lange dauern, nicht übergangen werden können und die Laufzeit der Algorithmen, so wie sie im Framework implementiert wurden, sehr negativ beeinträchtigen.

Bei diesem Speicherzugriff auf die „global Memory“ liegt auch, wie bei den Testergebnissen in Kapitel 6 zu erkennen, der größte Engpass für die Laufzeiten. Der Zugriff passiert nämlich, genauso wie bei den Algorithmen für die CPU, in unregelmäßigen Abständen und in unregelmäßiger Reihenfolge. Dies führt, wie auch im „CUDA C Programming Guide“ [2] nachlesbar, zu einem nicht performanten Verhalten, da Speicherzugriffe nur dann die gesamte Bandbreite der GPU zur „global Memory“ richtig ausnutzen kann, wenn viele Zugriffe zur gleichen Zeit und in einer bestimmten Reihgenfolge ausgeführt werden. Dies auf diese Weise für die Pfadplanungsalgorithmen auf der GPU zu implementieren, sodass es performant von statten geht und die Bandbreite zur „global Memory“ so gut wie möglich ausnutzt, ist sehr schwierig und würde noch einiges an Entwicklungsaufwand benötigen.

Allgemein wird die beste Verwendung der GPU-Speicherbereiche erreicht, so wie auch im „CUDA C Programming Guide“ [2] nachlesbar, wenn Daten in großen, zusammenhängenden Teilen aus der „global Memory“ in die „Shared Memory“ geladen werden und diese dann in diesem Speicherbereich bearbeitet werden. Das Zurückschreiben in die „global Memory“ funktioniert auch am besten, wenn die Daten als Ganzes in großen und zusammenhängenden Teilen von der „Shared Memory“ kopiert werden. Dies ist aber bei den Pfadplanungsalgorithmen mit ihren Listen nicht möglich, da bei der Anzahl an Threads pro Blocks bzw. Work-Items pro Work-Group die Listen nicht in der „Shared Memory“ Platz finden würden, weil dieser Speicherbereich im Gegensatz zur „global Memory“ zu klein ist. Deswegen wurden bei den Berechnungen der Pfadplanungsalgorithmen auf der GPU mit „Shared Memory“ die Listen nur ein Mal pro Block bzw. Work-Group gespeichert und pro Block bzw. Work-Group immer derselbe Agent gerechnet.

Somit kann als Fazit gesagt werden, dass Pfadplanungsalgorithmen auf der GPU bei ähnlicher bis beinahe gleicher Implementierung wie auf der CPU zu keiner Beschleunigung führen. Erst wenn die GPU-Algorithmen sowie auch die zu berechnenden Szenarien sehr stark auf die Grafikhardware zugeschnitten und die Dynamik der Berechnung verringert werden kann, können Laufzeitvorteile seitens der GPU erreicht werden. Dann ist aber der Vergleich der Implementierungen nicht mehr fair, da die im Code umgesetzten Vorteile der GPU-Plattform zu unterschiedlich zur CPU-Implementierung wären.

Es ist ein unverhältnismäßig großer Aufwand bei der GPU-Implementierung von Nöten, damit die optimale Auslastung der GPU erreicht werden kann. Da die Hardware in einem Computersystem nie gleich ist und eine Optimierung des Device-Codes für eine bestimmte Grafikhardware bei den Laufzeittests gegenüber der CPU-Implementierung unfair wäre, wurden nicht die gewünschten Ergebnisse wie in den Vorarbeiten erreicht. Zudem ist der Aufwand der Entwicklung auf der GPU sehr viel größer als bei einer Implementierung für die CPU, sodass überlegt werden kann, ob der Aufwand nicht besser für die Optimierung der CPU-Algorithmen aufgebracht werden soll.

Wenn die Entwicklung neuer Grafikhardware auf diese Weise voranschreitet und auch das GPU-Computing selbst vereinfacht und vielleicht auch verallgemeinert wird, dann kann es auch sinnvoll und interessant für die Entwicklung von Pfadplanungsalgorithmen für Spiele mit hoch dynamischen Inhalten werden. Spätestens wenn GPUs an die Tausend Rechenkerne bzw. ALUs besitzen und zudem auch schnellere Taktraten und auch größere Caches haben, dann kann das GPU-Computing sowohl technologisch als auch für die Spieleindustrie interessant werden. Wenn dann auch solche für CPUs typischen Entwicklungen wie „Branch-Prediction“ oder „Out-Of-Order-Execution“ in Transistoren auf dem Chip der GPU umgesetzt werden, dann würde das GPU-Computing bzw. die Softwareentwicklung für GPUs erleichtert werden, sodass sie sich mehr der traditionellen Softwareentwicklung für die CPU annähert.

Bis dahin werden sich die CPUs in ihrer Entwicklung den heutigen GPUs annähern, sodass in Zukunft Prozessoren mit mehreren hunderten Kernen erschwinglich werden. Die SISD-Architektur der CPUs wird sicherlich beibehalten werden, aber im Vergleich werden GPUs immer mehr Kerne besitzen. Außerdem werden GPUs mit ihrer SIMD-Architektur für grafikspezifische Aufgaben, aber mit Sicherheit auch für nicht grafikrelevante Probleme mit einer zur GPU passenden hoch parallelen Lösung, optimal geschaffen sein. Bis dahin muss beim GPU-Computing mit viel Wissen über die Hardware und deren Funktionsweise sowie mit „Best Practices“, vorgestellt z.B. im „CUDA C Programming Guide“ [2] oder der OpenCL Spezifikation [3], gearbeitet werden.

Es kann mit hoher Sicherheit gesagt werden, dass die Zukunft des GPU-Computing, speziell im Spielebereich, nicht in CUDA oder OpenCL liegt, sondern in DirectCompute von Microsoft [17], welches ein Teil der in der Spielebranche sehr häufig verwendeten und fast schon als Standard geltenden DirectX API von Microsoft ist. Die erste Umsetzung einer GPU-beschleunigten Pfadplanung o.Ä. in einem Spiel wird mit hoher Wahrscheinlichkeit mit DirectCompute geschehen, da DirectX von allen GPU-Herstellern unterstützt wird. Es verwendet für die Entwicklung des Codes für die GPU die HLSL, welche im Gegensatz zu CUDA und OpenCL, ohne zusätzliche Treiberinstallationen lauffähig und mit jeder Grafikhardware kompatibel ist. OpenCL ist sowohl auf Nvidia als auch auf ATI Hardware lauffähig, benötigt aber bei ATI einen eigenen Treiber (Stichwort: „ATI FireStream“).

Ein Laufzeitvergleich bzw. eine Implementierung einer Pfadplanung mittels DirectCompute wäre noch für das Framework interessant gewesen. Ein Vergleich zu CUDA und OpenCL würde zudem auch zeigen, ob DirectCompute im Punkto Performanz mit CUDA und OpenCL mithalten kann, auch wenn diese beiden Technologien sowohl hardwarenäher als auch einfacher zu handhaben sind als DirectCompute. Zudem werden Programmierkenntnisse in HLSL benötigt, was im Grunde sehr nahe an der Programmiersprache C liegt, aber doch etwas weiter entfernt davon ist als CUDA C [2] oder OpenCL C [3]. Die Lösung mit HLSL und DirectCompute wäre komplett DirectX spezifisch, so wie auch das Rendering des Frameworks, und würde ohne Open Source oder einer Plattformspezifik wie bei CUDA auskommen. Die Kontrolle über die GPU Implementierung wäre mit DirectCompute aber nicht so groß wie bei CUDA oder OpenCL, da die GPU-Computing APIs von CUDA bzw. OpenCL Informationen bereitstellen können, welche DirectCompute mittels HLSL bzw. DirectX nur bedingt können.

Eine gute Kontrolle über die GPU sowie über die Kernels und die Hardware wurde aber in dieser Arbeit benötigt. Die Informationen der GPU-Computing APIs waren sehr wichtig und hilfreich für die erfolgreiche Durchführung der Pfadplanung von großen Szenarien mit mehreren Tausend bis Hunderttausend Agenten, bei denen Hardwareressourcen knapp werden und Berechnungen mehrere Sekunden bis Minuten dauern können.

Literaturverzeichnis

[1] OpenMP. (2002, März) OpenMP C and C++ Application Program Interface, Version 2.0. [Online]. http://www.openmp.org/mp-documents/cspec20.pdf

[2] Nvidia Corporation. (2010, Sep.) NVIDIA CUDA C Programming Guide, Version 3.2. [Online]. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf

[3] Khronos Group - Open Standards for Media Authoring and Acceleration. (2010, September) The OpenCL Specification, Version 1.1, Revision 36. [Online]. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[4] Avi Bleiweiss, "GPU Accelerated Pathfinding," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2008, pp. 66-73.

[5] Avi Bleiweiss. (2009) Multi Agent Navigation on the GPU. [Online]. http://developer.download.nvidia.com/presentations/2009/GDC/MultiAgentGPU.pdf

[6] L. Fischer, R. Silveira, and L. Nedel, "GPU Accelerated Path-planning for Multi-agents in Virtual Environments," in *VIII Brazilian Symposium on Games and Digital Entertainment*, Oktober 2009, pp. 112-118.

[7] J. Shopf, J. Barczak, C. Oat, and N. Tatarchuk, "March of the Froblins: Simulation and Rendering Massive Crowds of Intelligent and Detailed Creatures on GPU," in *Advances in Real-Time Rendering in 3D Graphics and Games Course - SIGGRAPH*, 2008, pp. 52-101.

[8] Kamran Karimi, Neil G. Dickson, and Firas Hamze, "A Performance Comparison of CUDA and OpenCL," in *D-Wave Systems Inc.*, Burnaby, British Columbia, 2010.

[9] J. Van Den Berg, S. Patil, J. Sewall, D. Manocha, and M. Lin, "Interactive Navigation of Multiple Agents in Crowded Environments," in *Symposium on Interactive 3D Graphics and Games*, 2008, pp. 139-147.

[10] F. Dapper, E. Prestes, and L. Nedel, "Generating steering behaviors for virtual humanoids using bvp control," in *Proc. of CGI*, 2007.

[11] A. Treuille, S. Cooper, and Z. Popovic, "Continuum Crowds," in *ACM Transactions on Graphics*, Juli 2006, pp. 1160-1168.

[12] W. K. Joeng and T. Whitaker, "A Fast Eikonal Equation Solver for Parallel Systems," in *SIAM conference on Computational Sience and Engineering*, 2007.

[13] P. Fiorini and Z. Shiller, "Motion Planning in Dynamic Environments using Velocity Obstacles," in *International Journal of Robotics Research*, 1998, pp. 760-772.

[14] S. LaValle. (2006) Planning Algorithms. [Online]. http://msl.cs.uiuc.edu/planning/

[15] J. Tsitsiklis, "Efficient Algorithms for Globally Optimal Trajectories," in *IEEE Transactions on Automatic Control*, September 1995, pp. 1528-1538.

[16] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman, *Real-time Rendering*, 3rd ed.: Peters, Wellesley, 2008.

[17] Microsoft Corporation. (2011, Apr.) Microsoft Developer Network: MSDN-Library. [Online]. http://msdn.microsoft.com/library/aa139818.aspx

[18] OpenGL. (2011, Apr.) The Industry's Foundation for High Performance Graphics. [Online]. http://www.opengl.org/

[19] Gordon E. Moore, "Cramming more components onto integrated circuits," in *Electronics. 19, Nr. 3*, 1965, pp. 114–117.

[20] Peter N. Glaskowsky. (2009, September) NVIDIA’s Fermi: The First Complete GPU Computing Architecture. [Online]. http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA%27s_Fermi-The_First_Complete_GPU_Architecture.pdf

[21] J. Pearl, "Intelligent Search Strategies for Computer Problem Solving," in *Heuristics*.: Addison-Wesley, 1984.

[22] Edsger W. Dijkstra, "A note on two problems in connexion with graphs," in *Numerische Mathematik 1*., 1959, pp. 269-271.

[23] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in *IEEE Transactions on Systems Science and Cybernetics SSC4 (2)*, 1968, pp. 100–107.

[24] Beman Dawes, David Abrahams, and Rene Rivera. (2011, Mai) Boost C++ Libraries. [Online]. http://www.boost.org/

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

3D	Dreidimensional, dritte Dimension
ALU	Arithmetic Logic Unit (Arithmetisch-logische Einheit)
API	Application Programming Interface
BFS	Best First Search
BVP	Boundary Value Problem
Cg	C for Graphics (Shading Language von Nvidia)
CPU	Central Processing Unit
CSV	Comma-Separated Values
CUDA	Compute Unified Device Architecture
DDR	Double Data Rate
DLL	Dynamic Link Library (dynamische Programmbibliothek)
DXUT	DirectX Utility Toolkit
FLOPS	Floating Point Operations Per Second (Gleitkommaoperationen pro Sek.)
GB	Gigabyte
GDDR	Graphics Double Data Rate
GHz	Gigahertz
GLSL	OpenGL Shading Language
GPGPU	General Purpose Computing on Graphics Processing Unit
GPU	Graphics Processing Unit
HLSL	High Level Shading Language
IL	Intermediate Language

KB	Kilobyte
KI	Künstliche Intelligenz
MB	Megabyte
MHz	Megahertz
MSDN	Microsoft Developer Network (http://msdn.microsoft.com)
NPC	Non-Player Character (computergesteuerter Spieler)
NVCC	Nvidia CUDA Compiler
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
OpenMP	Open Multi-Processing
PC	Personal Computer
PS	Pixel Shader
PTX	Parallel Thread Execution
RAM	Random-Access Memory
RGB / RGBA	Red, Green, Blue / Red, Green, Blue, Alpha (Farbwerte)
RVO	Reciprocal Velocity Obstacle
SDK	Software Development Kit
SIMD	Single Instruction – Multiple Data
SISD	Single Instruction – Single Data
STL	Standard Template Library (C++ Standard Bibliothek)
UML	Unified Modeling Language
VS	Vertex Shader
WHQL	Windows Hardware Quality Labs

Anhang A: Bedienungsanleitung des Frameworks

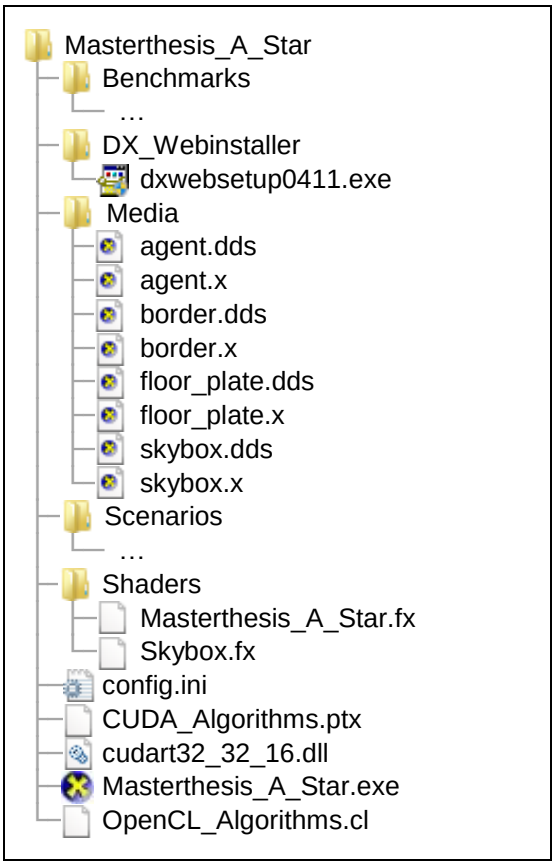


Abbildung 40: Diagramm der Ordnerstruktur des Frameworks „Masterthesis_A_Star“.

Das Framework namens „Masterthesis_A_Star“, welches im Laufe dieser Arbeit entwickelt wurde und in den folgenden Abschnitten meist nur „das Framework“ oder „die Applikation“ genannt wird, wird mit der in der Ordnerstruktur in Abbildung 40 zu sehenden ausführbaren Datei „Masterthesis_A_Star.exe“ gestartet. Es werden zur Benutzung des Frameworks keine zusätzlichen Installationen benötigt. Die einzige externe DLL (Dynamic Link Library) die benötigt wird ist „cudart32_32_16.dll“ aus dem CUDA Toolkit und ist, wie auch in der Ordnerstruktur in Abbildung 40 zu erkennen, dem Framework zugefügt.

Ein Fehler, welcher beim Start des Frameworks auftreten kann, ist das Fehlen von bestimmten DirectX-Dateien aufgrund von Versionsunterschieden bzw. älterer Versionen der DirectX API installiert auf dem Zielrechner. So kann eine Fehlermeldung lauten, dass eine bestimmte DLL fehlt wie z.B. die „d3dx10_43.dll“. Um solche Fehler zu beheben, wurde im Unterordner „DX_Webinstaller“ des Frameworks die „dxwebsetup0411.exe“ mitgegeben, zu sehen in Abbildung 40. Diese ausführbare Datei ist ein Setup zur Aktualisierung der lokalen DirectX Dateien auf die aktuellste Version über das Internet, der so genannte „DirectX Web Installer“. Nach dessen Ausführung sind alle Fehler dieser Art behoben und das Framework kann gestartet werden.

Programmzeilenargumente und Konfiguration

Beim Starten der Applikation über die Kommandozeile bzw. über ein Konsolenfenster können der Applikation eine Konfiguration in Form von Programmzeilenargumenten übergeben werden. Diese Argumente beginnen immer mit zwei Minuszeichen, folgend von einem Argumentnamen und, nach einem Leerzeichen, einem Wert für das Programmzeilenargument, wie auch in Abbildung 41 in einem Beispiel ersichtlich.

```
Masterthesis_A_Star.exe --{Argument1} {Wert1} --{Argument2} {Wert2}
```

Abbildung 41: Beispiel zur Verwendung von Programmzeilenargumenten.

Eine Übersicht über die im Framework vorhandenen Argumente und deren Typen mit detaillierter Beschreibung ihrer Funktion sind in Tabelle 12 zu finden. Der Ausdruck `{Argument1}` bzw. `{Argument2}` entspricht einem Argument aus dieser Tabelle und `{Wert1}` bzw. `{Wert2}` dem zum Argument angegebenen Typ (String, Ganzzahl, etc.). Dabei werden Werte, die dem Typ „String“ entsprechen und per Definition Zeichenketten sind, innerhalb von Hochkommata geschrieben. Dies ist aber nicht verpflichtend und vor allem für die Angabe von Dateipfaden mit Leerzeichen zu empfehlen, um Fehler beim Einlesen der Daten zu vermeiden.

Zusätzlich zur Kommandozeile können auch Konfigurationsdateien, wie z.B. „config.ini“, zu sehen im Diagramm der Ordnerstruktur in Abbildung 40, zur Konfiguration des Frameworks verwendet werden. Innerhalb der Konfigurationsdatei werden die Argumente mit anschließendem Gleichheitszeichen „=“ und deren Werten geschrieben, und zwar Zeile für Zeile. Werte vom Typ „String“ können wie zuvor in der Kommandozeile erwähnt in Hochkommata geschrieben werden, müssen aber nicht. Kommentare sind auch möglich und beginnen mit einer Raute „#“ und enden mit dem Beginn der nächsten Zeile. Die Struktur des Aufbaus solch einer Konfigurationsdatei inklusive eines Kommentars ist in Abbildung 42 dargestellt.

```
# Meine Programmoptionen für Masterthesis_A_Star
{Argument1}={Wert1}
{Argument2}={Wert2}
```

Abbildung 42: Beispiel zur Verwendung von Konfigurationsdateien.

Es ist noch zu erwähnen, dass die Programmzeile und eine Konfigurationsdatei zugleich verwendet werden können, wobei die Werte der Argumente in der Kommandozeile die Werte aus der Konfigurationsdatei überschreiben. Invalide bzw. ungültige Werte werden von der Applikation abgefangen und mittels einer Fehlermeldung darauf hingewiesen.

Argument	Typ	Beschreibung
configfile	String	**Nur innerhalb der Programmzeile verfügbar!** Dateipfad der zu verwendenden Konfigurationsdatei Standardwert: „config.ini“
agents	Ganzzahl	Setzt die Anzahl der zu simulierenden Agenten (Nur zusammen mit Argument „worldsize“ verwendbar)
worldsize	Ganzzahl	Setzt die Seitenlängen der Simulation Anzahl Felder = worldsize x worldsize (Nur zusammen mit Argument „agents“ verwendbar)
scenario	String	Dateipfad der zu ladenden Szenario-Datei (Ersetzt Argumente „agents“ und „worldsize“)
scenarioMultiplier	Ganzzahl	Multipliziert die Anzahl der Agenten für das angegebene Szenario (definiert durch Angabe von „scenario“ oder „agents“) So entstehen Agenten mit derselben Start- und Zielposition
windowWidth	Ganzzahl	Spezifiziert die Breite des Ausgabefensters in Pixel
windowHeight	Ganzzahl	Spezifiziert die Höhe des Ausgabefensters in Pixel
target	String	Spezifiziert die zu verwendende Technologie zur Berechnung der Pfadplanungsalgorithmen Erlaubte Werte: „CPU“ (Standard), „CUDA“, „OpenCL“
randSeed	Ganzzahl	Wert zur Initialisierung des Zufallszahlengenerators (Wird benötigt wenn Argument „scenario“ nicht gesetzt) Standardwert: 1
algorithm	String	Spezifiziert den zu benutzenden Pfadplanungsalgorithmus Erlaubte Werte: „A*“ (Standard) , „BFS“, „Dijkstra“
neighbourhood	String	Spezifiziert die Nachbarschaft der Knoten Erlaubte Werte: „Neumann“ (Standard), „Moore“
benchmark	Bool	Aktiviert den Benchmark-Modus der Applikation wenn „true“ bzw. 1 Standardwert: „false“ bzw. 0
benchmarkIterations	Ganzzahl	Spezifiziert die zu rechnenden Iterationen im Benchmark-Modus Standardwert: 20
benchmarkCSVFormat	String	Spezifiziert das Format der zu exportierenden CSV-Datei Erlaubte Werte: „english“ (Standard), „german“
benchmarkCSVFile	String	Setzt den Namen der zu exportierenden CSV-Datei Standardwert: siehe Kapitel „Der Benchmark-Modus“
GPUforceBlockDim	Ganzzahl	Setzt die maximale Größe der zu verwendenden Blocks bzw. Work-Groups für die Berechnung mit GPU-Algorithmen (Argument „target“ entspricht „CUDA“ oder „OpenCL“)
GPUforceMemUsage	Ganzzahl	Definiert die Größe des zu verwendenden globalen Speichers auf dem Device in Bytes bei der Berechnung mit GPU-Algorithmen (Argument „target“ entspricht „CUDA“ oder „OpenCL“)
GPUforceSMEM	Bool	Aktiviert die Verwendung der „Shared Memory“ der GPU Die Argumente „GPUforceBlockDim“ und „scenarioMultiplier“ müssen hierbei angegeben werden und denselben Wert besitzen

Tabelle 12: Liste mit Beschreibung der Programmargumente des Frameworks.

Vorstellung der Applikation

Nach dem Start der Applikation wird innerhalb eines Fensters die zu simulierende Szene in 3D sowie Informationen in Form von Text dargestellt. Dies kann, je nach Konfiguration des Frameworks (siehe vorheriges Kapitel), so ähnlich wie in Abbildung 43 aussehen, wo ein Test-Szenario geladen wurde. Details zum Laden von Szenarien und der Verwendung von „Szenario-Dateien“ werden später im gleichnamigen Kapitel genauer behandelt.

Die Bildschirmausgabe des Frameworks in Bild und Text kann in unterschiedliche Bereiche unterteilt werden, welche in Abbildung 43 nummeriert hervorgehoben sind und in den nun folgenden Abschnitten genauer erläutert und beschrieben werden.

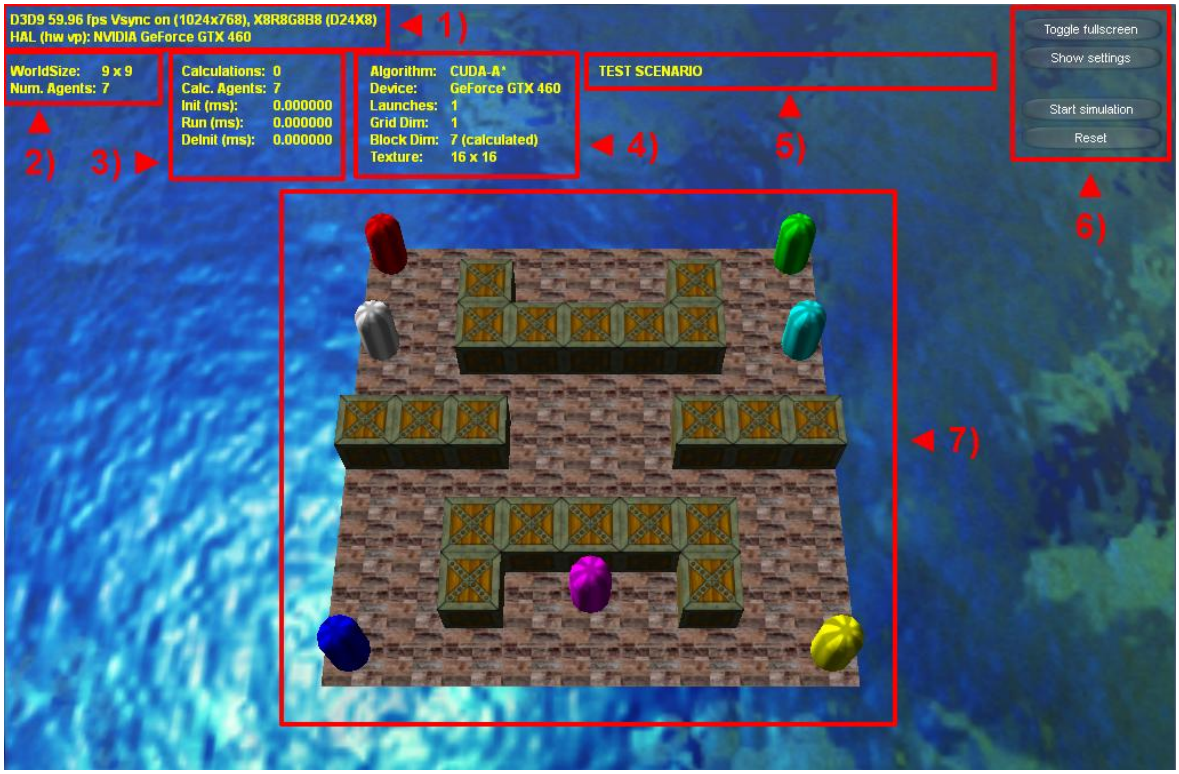


Abbildung 43: Screenshot der Applikation mit Hervorhebungen der Bestandteile.

1) DirectX-Informationen

Der auf der oberen Seite des Applikationsfensters zu findenden Text, zu sehen in Abbildung 43 bei „1)“, in zwei Zeilen sind vom DXUT (DirectX Utility Toolkit) generiert. Diese präsentieren Informationen zur Hardware, der DirectX API, sowie grafikspezifische Einstellungen wie Auflösung, Backbuffer-Format, aber auch die „Framerate“, also die Bilder pro Sekunde generiert von der Applikation. Auch der Name des für das Rendern der Szene verwendeten „Display-Adapters“ namens „NVIDIA GeForce GTX 460“, welche auch die GPU-Algorithmen berechnet, wird, wie in Abbildung 43 erkennbar, angezeigt.

Auf diese Informationen hat der Nutzer größtenteils keinen Einfluss, nur die Bildschirmauflösung kann mittels der Programmargumente, zu sehen in der Übersicht aus Tabelle 12 im vorherigen Kapitel, „windowWidth“ und „windowHeight“, eingestellt werden.

2) Welt-Informationen

Dieser Bereich, in Abbildung 43 mit „2)“ markiert, beinhaltet Informationen über die Welt, in der die Simulation abläuft. Der erste Wert namens „WorldSize“ entspricht der Größe der Welt im Format „{Breite} x {Höhe}“. Jedes Feld ist ein Quadrat, auf dem sich ein Agent befinden kann. Details zum genaueren Aufbau der Welt und deren Repräsentation können in der Arbeit im Kapitel 5.1 „Repräsentation des Graphen“ nachgelesen werden.

Der Wert von „WorldSize“ entspricht dem angegebenen Wert des Programmarguments namens „worldsize“, zu finden in der Übersicht der Programmargumente in Tabelle 12 im vorherigen Kapitel. Der Wert von „Num. Agents“ hingegen entspricht dem Programmargument „agents“. Diese Werte werden entweder über diese Programmargumente angegeben oder durch das Laden eines Szenarios mit einer vordefinierten Welt aus einer „Szenario-Datei“ bestimmt.

3) Laufzeitinformationen des Algorithmus

In diesem in Abbildung 43 mit „3)“ rot markierten Bereich werden Informationen zum Laufzeitverhalten des Algorithmus preisgegeben. Dabei handelt es sich beim Wert von „Calculations“ um die seit dem Simulationsstart durchgeführten Berechnungen zur Pfadplanung. „Calc. Agents“ hingegen gibt an, wie viele Agenten im letzten Durchlauf der Pfadplanung berechnet wurden.

Die Gleitkommawerte bei „Init“, „Run“ und „DeInit“ repräsentieren verschiedene Laufzeiten in Millisekunden (ms) des letzten Durchgangs der Pfadplanung. Dabei ist der Wert von „Init“ interessant für GPU-Algorithmen, da dieser die Dauer des Datentransfers vom „Host“ zum „Device“ (Definition siehe Kapitel 3.4.1 in dieser Arbeit) repräsentieren, welche die CPU-Algorithmen nicht durchführen müssen. „Run“ bezieht sich auf die Laufzeit zur Berechnung aller Agenten, und „DeInit“ entspricht der Dauer des Zurückladens der Daten vom „Device“ zum „Host“. Diese Daten werden vom „Benchmark-Modus“ für dessen Output verwendet. Genauere Details zum „Benchmark-Modus“ und seinem Output können später im Kapitel „Der Benchmark-Modus" nachgelesen werden.

4) Algorithmus- und Technologieinformationen

Der Inhalt des Bereiches, welcher in Abbildung 43 mit „4)“ markiert ist, ist je nach konfigurierter Technologie und Algorithmus variabel. So besteht der Wert von „Algorithm“ aus dem Namen der ausgewählten Technologie im Programmargument „target“ (siehe Tabelle 12) und dem Namen des Algorithmus, so wie er im Programmargument „algorithm“ (siehe Tabelle 12) angegeben wird. In Abbildung 43 ist der Wert „CUDA-A*“ zu erkennen. Somit sind die Argumente „target“ und „algorithm“ auch auf „CUDA“ und „A*“ gesetzt.

Außerdem wird bei der Angabe des Programmarguments „GPUforceSMEM“ (siehe Tabelle 12) bei den GPU-Algorithmen (Argument „target“ entspricht „CUDA“ oder „OpenCL“) zusätzlich zu dem Namen der Technologie und des Algorithmus, ein „(SMEM)“ angezeigt. So lautet der Algorithmus „CUDA-A*“, wie in Abbildung 44 (b) gesehen werden kann, bei Angabe von „GPUforceSMEM“ „CUDA-A* (SMEM)“. Details zu den einzelnen Algorithmen und deren Implementierung in den unterschiedlichsten Technologien und Modi können in der Arbeit im Kapitel 5 „Das Framework“ nachgelesen werden.

Die restlichen Ausgaben dieses Bereiches unterscheiden sich von Technologie zu Technologie und können in Abbildung 44 für „target=CPU“ (a), „target=CUDA“ (b) und „target=OpenCL“ (c) gesehen werden, jeweils mit gewähltem Algorithmus „algorithm=A*“.

(a) (b) (c)

Abbildung 44: Technologieinformationen von (a) CPU, (b) CUDA und (c) OpenCL.

Bei „target=CPU“ werden nur zusätzlich die Anzahl der im System vorhandenen Prozessor-Kerne (CPU-Cores) mit Hilfe der OpenMP API [1] ausgelesen und angegeben. In Abbildung 44 (a) wären dies vier Kerne, da das verwendete System mit einem „Intel Core2 Quad“ Vierkern-Prozessor ausgestattet ist. Bei den GPU-Algorithmen mit der Konfiguration „target=CUDA“ oder „target=OpenCL“ wird zusätzlich noch das „Device“ angegeben, also der für die Pfadplanungsberechnungen ausgewählte Grafikadapter. Im Fall von (b) und (c) in Abbildung 44 wäre das die „GeForce GTX 460“, welche im verwendeten System vorhanden ist und von den jeweiligen APIs gefunden wurde.

Der Wert bei „Launches“ bezieht sich auf die Anzahl der durchzuführenden Iterationen pro Pfadplanungsberechnung, da aufgrund von begrenzten Hardwareressourcen seitens des „Device“ nicht alle Agenten auf einmal auf der GPU gerechnet werden können. Bei diesen Ressourcen handelt es sich z.B. um den globalen Grafikkartenspeicher (RAM) oder auch die Anzahl der Multiprozessoren auf der GPU oder der Register pro Multiprozessor, wie im Kapitel 5.3.2 „GPU Implementierung“ genauer beschrieben.

Auch bei den nächsten Werten „Grid Dim“ und „Block Dim“, zu sehen bei CUDA in Abbildung 44 (b), bzw. „Groups“ und „Group Dim“, zu sehen bei OpenCL in Abbildung 44 (c), handelt es sich wiederum um berechnete Restriktionen aufgrund von gewissen Hardwareressourcen seitens der GPU. Dabei werden die Ausmaße eines Blocks bzw. einer Work-Group sowie des Grids und der Anzahl an Work-Groups („Groups“) bestimmt,

um eine bestmögliche parallele Auslastung der GPU zu erreichen. Wie diese Berechnung genauer vonstattengeht und innerhalb des Frameworks, sowohl für CUDA als auch OpenCL implementiert ist, kann im Kapitel 5.3.2 „GPU Implementierung“ der Arbeit nachgelesen werden. Des Weiteren können ebenfalls die Definitionen eines „Grids“ bzw. eines „Blocks“ bei CUDA bzw. von „Work-Groups“ bei OpenCL sowie weitere Details zu diesen Begriffen in Kapitel 3.4.2 „Prinzip von Threads, Blocks und Grids“ in der Arbeit nachgelesen werden.

Bei „Block Dim“ bzw. „Group Dim“ ist in Abbildung 44 (b) bzw. (c) außerdem zu erkennen, dass der Begriff „*calculated*“, zu Deutsch „berechnet“, dahintersteht. Dies hat mit dem Programmargument „GPUforceBlockDim“ (siehe Programmargumente in Tabelle 12) zu tun, mit dem ein fixer Wert für die Größe eines Blocks bzw. einer Work-Group angegeben werden kann. Ist dieser Wert akzeptabel und kleiner als der vom Framework berechnete Wert, wird dieser angenommen und neben dem Wert steht dann „*forced*“, was zu Deutsch so viel bedeutet wie „erzwungen“ oder „bekräftigt“.

Der letzte Wert, der von den GPU-Algorithmen ausgegeben wird, lautet „Texture“ und gibt die Größe der Textur an, in der die Repräsentation der Welt gespeichert wird. Diese wird in Breite x Höhe angezeigt, wobei die Werte von Breite und Höhe der nächsthöheren Zweierpotenz entsprechen, in der die Weltrepräsentation passt, so wie auch im Kapitel 5.3.2 „GPU Implementierung“ in dieser Arbeit erläutert. So hat die Welt, zu sehen in Abbildung 43, welche die Ausmaße bzw. eine „WorldSize“ von 9x9 besitzt, einen Wert bei „Texture“ von 16x16, da 16 die nächsthöhere Zweierpotenz nach der Zahl 9 ist. Diese Werte sind auch in Abbildung 44 (b) und (c) zu sehen, die mit demselben Szenario wie in Abbildung 43 aufgenommen wurden.

5) Szenario-Name

In diesem in Abbildung 43 mit „5)“ markierten Bereich wird der Name des gerade berechneten Szenarios angezeigt. Der Name des Szenarios, wie er auch in Abbildung 43 mit dem Wert „TEST SZENARO“ zu sehen ist, stammt aus der „Szenario-Datei“, welche in Abbildung 47 im nächsten Kapitel beschrieben und dargestellt wird. Für den Fall, dass eine Welt per Zufallszahlengenerator unter Angabe der Programmargumente „agents“, „worldsize“ und „randSeed“ anstatt der Angabe eines Szenarios mittels des Programmargumentes „scenario“ (siehe Programmargumente in Tabelle 12) erzeugt wird, kann der in Abbildung 45 zu erkennende Aufbau für den Szenario-Namen innerhalb des Applikationsfensters gesehen werden.

```
Random world with seed {randSeed} and size {worldsize}x{worldsize}
```

Abbildung 45: Aufbau des Szenario-Namens bei zufällig generierter Welt.

Die Werte, die statt den Begriffen `{randSeed}` und `{worldsize}` aus Abbildung 45 eingesetzt werden, entsprechen den angegebenen Werten der Programmargumente „randSeed“ und „worldsize“. Es ist zu erkennen, dass beim Erzeugen von zufälligen Szenarien nur quadratische Welten konstruiert werden können, da der Wert des Programmarguments „worldsize“ die Seitenlänge der Welt repräsentiert und sowohl für die Höhe als auch die Breite verwendet wird. Der Wert des Programmargumentes „randSeed“ wird verwendet, um Start- und Zielpunkte für die zufällig platzierten Agenten zu definieren, wobei bei der Angabe der gleichen Werte für „randSeed“, „worldsize“ sowie „agents“ immer dasselbe Szenario generiert wird. Dies führt zu einer Reproduzierbarkeit der Szenarien, auch wenn „zufällig“ erstellt.

6) Buttons zur Applikationssteuerung

Der in Abbildung 43 mit „6)“ markierte Bereich beinhaltet die Buttons zur Steuerung der Applikation sowie der Simulation.

Der erste Button mit der Aufschrift „Toggle fullscreen“ lässt die Applikation in den „Fullscreen-Modus“ wechseln, sowie auch wieder zurück in den „Fenster-Modus“.

Bei Betätigung des Buttons „Show settings“ wird ein Dialog geöffnet, in dem DirectX spezifische Einstellungen eingesehen und auch geändert werden können. Dieser Dialog ist Standard von DXUT und wurde in die Applikation integriert.

Durch Betätigen des Buttons mit der Aufschrift „Start simulation“ kann die Berechnung und Simulation der Agenten gestartet werden. Wird dieser Button betätigt, ändert er seine Aufschrift in „Stop simulation“. Daraufhin kann die Simulation der Pfadplanung und der Agenten gestoppt werden und die Aufschrift ändert sich wieder zurück in „Start simulation“.

Der Button mit der Aufschrift „Reset“ setzt die Simulation zurück auf den Ursprung. Es werden alle Agenten auf deren Startpositionen sowie auch die Laufzeitinformationen des Algorithmus (siehe „3)“ und Abbildung 43) auf deren Ursprung zurückgesetzt.

7) 3D Darstellung der Szene

In der Mitte des Applikationsfensters wird die zu simulierende Welt mit den Agenten angezeigt, in Abbildung 43 mit „7)“ rot markiert. Die Ansicht auf die Szene kann mit Hilfe der Maus gedreht werden, indem bei gehaltener linker Maustaste die Maus innerhalb des Applikationsfensters bewegt wird. Außerdem kann mit Hilfe des Mausrades die Szenerie herangezoomt bzw. weggezoomt werden.

Die Agenten werden vom Pfadplanungsalgorithmus zu ihren Zielen geleitet und bewegen sich durch die Welt auf den zu sehenden Boden, vorbei an den Kisten, die Hindernisse darstellen. In der Simulation ist keine lokale Kollisionserkennung oder Kollisionsvermeidung zwischen den Agenten implementiert, sodass sie sich untereinander nicht in ihrer

Pfadfindung beeinflussen können. Die Pfade der Agenten können sich somit überschneiden und Agenten können auch durch jeweils andere Agenten ungehindert durchgehen.

Im „Benchmark-Modus“ der Applikation, welcher später genauer im Kapitel namens „Der Benchmark-Modus“ erläutert wird, ist diese Darstellung der Simulation nicht zu sehen. Es werden, wie auch in Abbildung 46 ersichtlich, nur die Welt-Informationen „2)“, die Algorithmus- und Technologieinformationen „4)“, die Laufzeitinformationen des Algorithmus „3)“ und der Szenario-Name „5)“ angezeigt. Buttons zur Applikationssteuerung „7)“ werden nicht angezeigt, weil im „Benchmark-Modus“ die Pfadplanung nur gerechnet, aber nicht dargestellt wird. Außerdem werden die DirectX Informationen „1)“ nicht angezeigt, da keine Szenerie gerendert sondern nur Text ausgegeben wird.

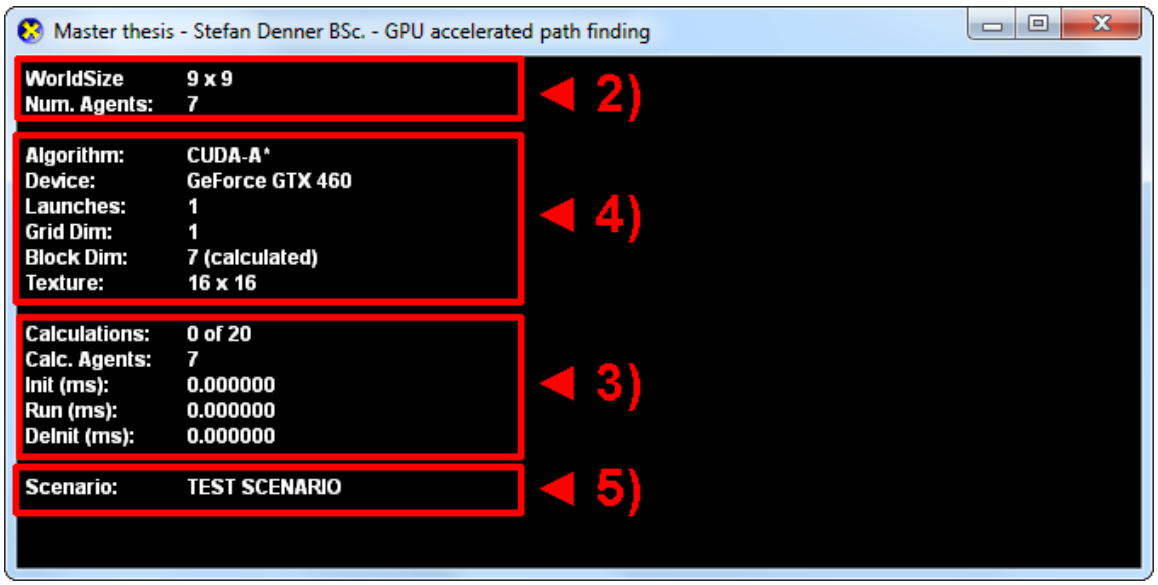


Abbildung 46: Der Benchmark-Modus mit Hervorhebungen der Bestandteile.

Szenario-Dateien

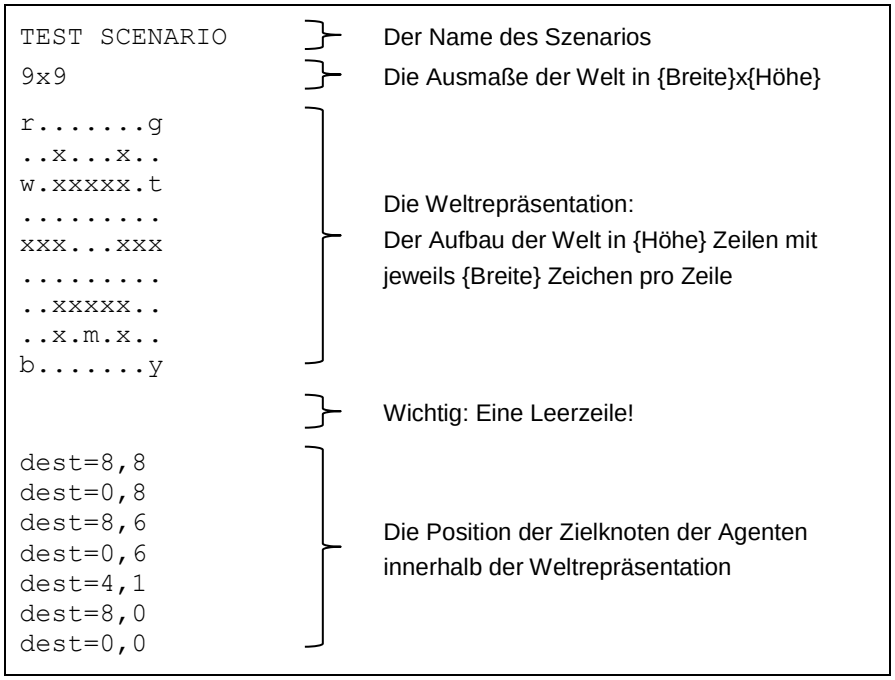


Abbildung 47: Beschreibung des Aufbaus einer Szenario-Datei anhand eines Beispiels.

Mit Hilfe von den in den letzten Abschnitten des Öfteren erwähnten Szenario-Dateien können vordefinierte Szenarien für das Framework entwickelt und auch von der Applikation berechnet werden. Zur Veranschaulichung des allgemeinen Aufbaus einer Szenario-Datei wird dieser in Abbildung 47 anhand eines Beispiels dargestellt.

Wie in Abbildung 47 ersichtlich besteht die erste Zeile der Szenario-Datei aus dem Namen des Szenarios, der frei wählbar ist und, wie schon erwähnt, innerhalb der Applikation auch angezeigt wird. Bei der dargestellten Szenario-Datei handelt es sich um das schon in den letzten Abschnitten verwendete Szenario namens „TEST SZENARIO“, dessen Aussehen direkt nach dem Laden innerhalb der Applikation in Abbildung 43 gesehen werden kann.

In der zweiten Zeile der Szenario-Datei werden die Ausmaße des in den Zeilen danach folgenden Aufbaus der Welt angegeben, im Format {Breite}x{Höhe}. Beim Beispiel in Abbildung 47 besitzt die zu konstruierende Welt für die Simulation eine Breite und eine Höhe von 9 Feldern. Somit wird innerhalb der Szenario-Datei, wie auch in Abbildung 47 gesehen werden kann, „9x9“ angegeben.

Danach folgt in den darauffolgenden, der Anzahl von {Höhe} entsprechenden Zeilen der Aufbau der Welt, wobei jede Zeile genau eine Anzahl von {Breite} Zeichen besitzt. Im Beispiel in Abbildung 47 sind dies jeweils 9 Zeilen zu je 9 Zeichen. Folgende Zeichen sind erlaubt und repräsentieren folgende Elemente der Welt:

1. Ein `.` repräsentiert ein von Agenten begehbares Feld
2. Ein `x` repräsentiert ein nicht begehbares Feld
3. Die Zeichen `w`, `r`, `g`, `b`, `y`, `t` und `m` repräsentieren die Startpositionen von Agenten, welche jeweils eine der Farben aus Tabelle 13 besitzen.

Beispiele für Agenten mit jeweils einer der Farben aus Tabelle 13 kann im Screenshot der Applikation aus Abbildung 43 gesehen werden, wo die Szenario-Datei aus Abbildung 47 geladen wurde. Jeder der sieben Agenten besitzt, sowohl im Screenshot als auch in der Szenario-Datei erkennbar, eine dieser sieben möglichen Farben.

Nach der Weltrepräsentation folgt eine Leerzeile, wiederum gefolgt von den Angaben der Zielknoten der jeweiligen Agenten. Dabei wird ein Index in Form von „dest={x},{y}“ angegeben, welches ein Feld innerhalb des Aufbaus der Welt repräsentieren. Der Index bzw. die Werte von {x} und {y} beginnen links oben, also erste Zeile, erstes Zeichen, mit „0,0“ zu zählen und endet rechts unten mit „{Breite-1},{Höhe-1}“. Jede weitere Zeile besitzt einen um 1 höheren Wert von {y} als die vorhergehende. Jedes Zeichen innerhalb einer Zeile besitzt einen um 1 höheren Wert für {x} als das vorhergehende. So werden die einzelnen Felder identifiziert und können als Zielknoten für Agenten angegeben werden.

Zeichen	Farbe	Bezeichnung		RGB Farbwert			Hexadezimal Farbwert
		englisch	deutsch	R	G	B	
`w`		white	Weiß	255	255	255	#FFFFFF
`r`		red	Rot	255	0	0	#FF0000
`g`		green	Grün	0	255	0	#00FF00
`b`		blue	Blau	0	0	255	#0000FF
`y`		yellow	Gelb	255	255	0	#FFFF00
`t`		turquoise	Türkis	0	255	255	#00FFFF
`m`		magenta	Magenta	255	0	255	#FF00FF

Tabelle 13: Übersicht über die zuweisbaren Farben der Agenten.

Welcher Agent welches Ziel zugewiesen bekommt hängt von der Reihenfolge des Erscheinens der Agenten innerhalb der Weltrepräsentation ab. Der erste Wert, im Fall vom Szenario in Abbildung 47 der Wert „8,8“, wird dem ersten Agenten zugewiesen, der von links oben (Zeile 0, Zeichen 0) von Zeile zu Zeile bis nach „{Breite-1},{Höhe-1}“ gefunden werden kann. Dies ist der rote Agent „`r`“. Der zweite Wert wird dann dem nächsten Agenten zugewiesen, der gefunden werden kann, nämlich dem grünen Agenten „`g`“. Danach bekommt der dritte Wert der Agent „`w`“, der vierte der Agent „`t`“, der fünfte der Agent „`m`“, der sechste der Agent „`b`“ und der siebente der Agent „`y`“, entsprechend der Reihenfolge ihrer Erscheinung innerhalb der Weltrepräsentation, zugewiesen.

Der Benchmark-Modus

Der so genannte „Benchmark-Modus“ der Applikation dient zum schnellen Durchführen von Testläufen der Algorithmen und Technologien. Dabei werden die Laufzeitinformationen des Algorithmus, darunter auch die Millisekunden Werte von „Init“, „Run“ und „DeInit“, zu sehen im Screenshot der Applikation in Abbildung 43, protokolliert und am Ende des Benchmarking gespeichert. Zur Messung dieser Laufzeiten wurde die hochauflösende Zeitmessungsfunktion „QueryPerformanceCounter“ der Windows API verwendet [17].

Wie schon bei der Vorstellung der Applikation im Kapitel „Vorstellung der Applikation“ erwähnt und auch im Screenshot des „Benchmark-Modus“ in Abbildung 46 ersichtlich, wird die Szene zur Darstellung der Welt mit den Agenten nicht gerendert, sondern nur der Informationstext ausgegeben. Überdies werden auch Ressourcen, die für diese Darstellung mittels DirectX benötigt werden, wie Shader, 3D-Modelle und Texturen, nicht geladen und benötigen somit auch keinen zusätzlichen Grafikspeicher. Dadurch werden Hardwareressourcen der GPU gespart, welche dann mehr für die Durchführung der GPU-Pfadplanungsalgorithmen verwendet werden können, anstatt für das Rendering.

Die Speicherung der Ergebnisse des Benchmarkings erfolgt im Unterordner „Benchmarks“, auch im Diagramm der Ordnerstruktur in Abbildung 40 zu sehen, in Form einer CSV-Datei. Solch eine CSV-Datei („Comma-Separated Values“, zu Deutsch „Kommagetrennte Werte“) beinhaltet die Laufzeitwerte der Applikation im Tabellenformat, sodass sie von Tabellenkalkulationsprogrammen wie z.B. Microsoft Excel gelesen und interpretiert werden können. Die Namensgebung dieser Datei erfolgt entweder mit Hilfe des Programmarguments „benchmarkCSVFile“ (siehe Tabelle 12) oder erfolgt nach einer Namenskonvention, in der das Datum der Erstellung samt Uhrzeit sowie die Technologie, der Algorithmus sowie auch die Agentenanzahl und der Szenario-Name Bestandteil dieser sind. Auch die Formatierung der Datei, welche mit dem Programmargument „benchmarkCSVFormat“ (siehe Tabelle 12) eingestellt werden kann, ist Bestandteil dieser Namenskonvention, welche wie in der Darstellung des allgemeinen Aufbaus in Abbildung 48 definiert werden kann.

```
Format_JJJJ.MM.TT_HH.MM.SS_Technologie-Algorithmus_Agentenanzahl_Szenario.csv
```

Abbildung 48: Darstellung der Namenskonvention der CSV-Datei.

Das Format der CSV-Datei entspricht entweder dem Wert „english“ oder „german“ und repräsentiert die jeweiligen sprachspezifischen Versionen des CSV-Formats in englischer und deutscher Sprache. So werden in der deutschsprachigen CSV-Datei die einzelnen Werte der Felder mittels einem einfachen Komma (,) getrennt, wobei das Dezimaltrennzeichen für Kommazahlen ein Punkt (.) ist. So werden die Werte innerhalb der CSV-Datei wie folgt abgelegt:

```
"1.123456","1.234567","1.345678",…
```

Im Gegensatz hierzu wird im deutschsprachigen Format ein Strichpunkt (;) zum Trennen der einzelnen Werte der CSV-Datei verwendet. Zudem ist auch das Dezimaltrennzeichen der Kommawerte nicht wie zuvor der Punkt (.), sondern das Komma (,), welches zuvor als Trennzeichen für die Werte diente. So werden dieselben Werte wie oben innerhalb der CSV-Datei nun wie folgt dargestellt:

```
"1,123456";"1,234567";"1,345678";…
```

Diese Unterscheidung ist wichtig, um die Kompatibilität für das englischsprachige und das deutschsprachige Microsoft Excel gewährleisten zu können, mit dem die Ergebnisse weiterverarbeitet werden sollen. Zudem ändert sich der Vollständigkeit halber auch die Sprache der Überschriften.

Beispiele für die Ausgabe einer CSV-Datei innerhalb eines Tabellenkalkulationsprogramms in deutscher und englischer Sprache sowie mit den dazu passenden sprachspezifischen Einstellungen können in Tabelle 14 für „benchmarkCSVFormat=german“ und in Tabelle 15 für „benchmarkCSVFormat=english“ eingesehen werden. Dabei werden die Mittelwerte bzw. „Averages“, zu sehen am Ende der Tabelle in Zeile 10, vom Framework berechnet. Auch die durchschnittliche Zeit pro Agent, zu sehen in der Spalte F unter „durchschn. Zeit pro Agent“ bzw. „avg. time per agent“, wird vom Framework berechnet, indem die Werte unter „Run“ in Spalte C durch die Anzahl der berechneten Agenten in Zeile 1, Spalte F, neben „Agenten“ bzw. „Agents“, in den beiden Tabellen jeweils 7, dividiert wird.

Die Werte neben „Launch-Infos“ in Zeile 5 entsprechen im Fall von GPU-Algorithmen, wie dem „CUDA-A*“, zu sehen in den beiden Tabellen, der Anzahl der Threads pro Block bzw. Work-Items pro Work-Group, der Anzahl der Blocks bzw. Work-Groups pro Durchlauf des Algorithmus und der Anzahl der Durchläufe selbst. Diese Werte entsprechen den Angaben der „Algorithmus- und Technologieinformationen“, vorgestellt im Kapitel „Vorstellung der Applikation“, Abbildung 43, markiert mit „4)“, und wird, da dieser zur Laufzeit mit Hilfe von Informationen über die GPU-Hardware berechnet wird, ebenfalls in die CSV-Datei geschrieben.

Für Details zu Blocks und Threads bei CUDA bzw. Work-Groups und Work-Items bei OpenCL siehe Kapitel 3.4.2 „Prinzip von Threads, Blocks und Grids“ in der Arbeit. Diese drei Werte innerhalb der CSV-Datei gelten auch für OpenCL-Algorithmen. Nur bei den CPU-Algorithmen wird stattdessen die Anzahl der für die Berechnung verwendeten Threads angegeben, welche der Anzahl der Hardware-Threads der im System vorhandenen CPU entsprechen. Dies entspricht bei einem „Intel Core2 Quad“ Prozessor dem Wert „4“, für vier Prozessorkerne auf der CPU.

	A	B	C	D	F
1	Technologie:	CUDA		Agenten:	7
2	Algorithmus:	A*		Weltbreite:	9
3	Iterationen:	3		Welthöhe:	9
4	Szenario:	TEST_SCENARIO			
5	Launch-Infos:	7	1	1	
6	Laufnummer	Init (ms)	Run (ms)	DeInit (ms)	durchschn. Zeit pro Agent (ms)
7	1	0,092710	1,286399	0,037156	0,183771
8	2	0,127341	1,321751	0,054832	0,188822
9	3	0,100286	1,262951	0,078641	0,180422
10	Mittelwerte:	0,106779	1,290367	0,056877	0,184338

Tabelle 14: Inhalte der CSV-Datei bei Argument benchmarkCSVFormat="german".

	A	B	C	D	F
1	Technology:	CUDA		Agents:	7
2	Algorithm:	A*		WorldWidth:	9
3	Iterations:	3		WorldHeight:	9
4	Scenario:	TEST_SCENARIO			
5	Launch-Infos:	7	1	1	
6	Run number	Init (ms)	Run (ms)	DeInit (ms)	avg. time per agent (ms)
7	1	0.092710	1.286399	0.037156	0.183771
8	2	0.127341	1.321751	0.054832	0.188822
9	3	0.100286	1.262951	0.078641	0.180422
10	Averages:	0.106779	1.290367	0.056877	0.184338

Tabelle 15: Inhalte der CSV-Datei bei Argument benchmarkCSVFormat="english".

Anhang B: Die Heap-Sortierungsfunktion „Heapify“

Die Heap-Sortierungsfunktion „Heapify“, welche die Prioritätenliste des Frameworks für die Operation „Extract“ verwendet, vorgestellt im Kapitel 5.2.1, ist ein Teil des in der Informatik bekannten „Heap-Sort“-Algorithmus. „Heapify“ ist in der Lage, ein beliebiges ungeordnetes Array so umzusortieren, dass die Inhalte der gewünschten Heap-Eigenschaft bzw. der so genannten „Heap-Order“ entsprechen. Im Fall der „Priority Queue“ ist dies die so genannte „Min-Heap-Order“, bei der das Elternelement eines Knotens immer einen kleineren Wert besitzt als dessen Kinderelemente.

Die „Priority Queue“ wird als Heap innerhalb eines Arrays verwaltet, so wie schon im Kapitel 5.2.1 vorgestellt, und wird im Pseudocode der Funktion „Heapify“ in Abbildung 49 `pqArray` genannt. Zudem wird auch, so wie im Kapitel 5.2.1 beschrieben, die assoziative Liste `f` zur Speicherung der Werte der Bewertungsfunktion für Knoten des Algorithmus verwendet. Informationen zu `pqArray` und `f`, sowie zu deren Implementierungen im Framework als „Priority Queue“ bzw. „hashed Array“, können im Kapitel 5.2.1 und 5.2.2 nachgelesen werden.

```
1:    // pqArray[]: Priority Queue mit Knoten in Heap-Order nach f[]
2:    // f[]: Bewertung für Knoten in Queue
3:    function Heapify()
4:      int start = pqArray.size/2 - 1
5:      int end ← pqArray.size-1
6:      while start >= 0 do
7:        int root ← start
8:        while root*2 + 1 <= end do
9:          int left ← root*2 + 1       // linkes Kinderelement
10:         int right ← root*2 + 2      // rechtes Kinderelement
11:         int smallest ← root
12:         if f[pqArray[left]] < f[pqArray[smallest]] then
13:           smallest ← left
14:         if right <= end and
15:         f[pqArray[right]] < f[pqArray[smallest]] then
16:           smallest ← right
17:         if smallest != root then
18:           pqArray[root] ↔ pqArray[smallest]   // tausche Elemente!
19:           root ← smallest
20:         else
21:           break
22:       --start
23:   end
```

Abbildung 49: Die Heap-Sortierungsfunktion „Heapify“ der Priority Queue.

Auch wenn die Heap-Sortierungsfunktion, zu sehen im Pseudocode in Abbildung 49, sehr viele Zeilen besitzt und sehr komplex wirkt, besteht deren Aufgabe im Grunde nur darin, die Positionen der Knoten im Array mit deren Kinderknoten solange zu tauschen, bis die Elternknoten immer einen kleineren Wert in `f` besitzen als deren Kinderknoten, also die „Min-Heap-Order“ für alle Knoten in der Liste zutrifft. Begonnen wird hierbei beim Index des letzten Knotens des Arrays, welcher noch zumindest einen Kinderknoten besitzt. Die Berechnung des Index dieses Knotens ist im Pseudocode in Zeile 4 bei der Initialisierung der Variable `start` zu sehen.

Diese Variable `start` wird in der ersten, äußeren `while`-Schleife, zu sehen in Zeile 6 des Pseudocodes, bis hin zum ersten Element des Arrays mit dem Index 0, um 1 verringert, zu sehen im Pseudocode am Ende dieser `while`-Schleife in Zeile 22 (`--start`). Der Index von `start` wird dann für die nächste, innere `while`-Schleife, zu sehen in Zeile 8 des Pseudocodes, der Variable `root` zugewiesen, deren Kinderknoten mit Hilfe der Werte gespeichert in `f` auf die Einhaltung der „Min-Heap-Oder“ überprüft werden.

Die Berechnung der Indices der beiden Kinderknoten können im Pseudocode in den Zeilen 9 und 10 für den jeweils linken Kinderknoten (`left`) und den rechten Kinderknoten (`right`) gesehen werden. Diese Berechnungen entsprechen der Kalkulation der Indices von Kinderknoten innerhalb eines Arrays, vorgestellt bei der „Priority Queue“ bzw. beim Heap in den Formeln (8) und (9) im Kapitel 5.2.1.

In der inneren `while`-Schleife wird solange die „Min-Heap“-Eigenschaft der Kinderknoten überprüft und getauscht, bis diese bis hin zum letzten Element im Array bei Index `end` zutrifft. Hierfür wird der gerade geprüfte Knoten in `root` als der kleinste angenommen und in der Variable `smallest` gespeichert. Dann wird überprüft, wie in der `if`-Abfrage in Zeile 12 zu sehen, ob der linke Kinderknoten bei Index `left` einen kleineren Wert besitzt als der bei Index `smallest`, der dem Index `root` entspricht. Wenn dies stimmt und somit der Kinderknoten einen kleineren Wert besitzt als dessen Elternknoten, was die „Min-Heap-Order“ bricht, dann wird der linke Knoten `left` als der kleinste in der Variable `smallest` gemerkt, so wie auch in Zeile 13 ersichtlich.

Nach dem linken Kinderkonten wird auch der Wert des rechten Kinderknotens bei Index `right` mit dem des Knotens bei Index `smallest` verglichen, welcher bis dahin entweder dem Knoten `root` oder `left` entspricht. Dies kann in der `if`-Abfrage im Pseudocode in Abbildung 49 in den Zeilen 14 und 15 gesehen werden, in der zusätzlich auch überprüft wird, ob der rechte Kinderknoten bei Index `right` überhaupt innerhalb der Grenzen des Arrays `pqArray` liegt (`right <= end`). Hat der Wert des Knoten beim Index `right` einen kleineren Wert als der bei `smallest`, dann wird, so wie auch in der Zeile 16 zu sehen, der Wert von `right` dem Indexwert `smallest` zugewiesen.

Der Indexwert von `smallest`, der zu Beginn dem Wert `root` entspricht, kann sich nach den beiden Vergleichen mit den Kinderknoten, falls diese kleiner sind und die „Min-Heap-Order“ brechen, geändert haben und nun dem Index eines der beiden Kinderknoten entsprechen. Ist dies der Fall und der Wert von `smallest` entspricht nicht mehr dem von `root`, so wie in der `if`-Abfrage in Zeile 17 des Pseudocodes überprüft wird, dann werden die Elemente von `pqArray` bei den jeweiligen Indices `root` und `smallest` getauscht und auch der Wert des Index `smallest` der Variable `root` zugewiesen, so wie in den Zeilen 18 und 19 des Pseudocodes gesehen werden kann.

Durch diesen Tausch wird im nächsten Durchlauf der inneren `while`-Schleife der neue kleinste Knoten bei `smallest`, der ein Kinderknoten von `root` war und nach dem Tausch nun nicht mehr ist, da die „Min-Heap-Order“ nicht stimmte, dessen Kinderknoten auf dieselbe Wiese auch überprüft. Dies geht weiter bis zum Index des letzten Knotens im Array, dessen Kinderknoten nicht mehr die „Min-Heap-Order“ brechen. Dann ändert sich auch nicht der Wert von `smallest` und behält bis zur `if`-Abfrage in Zeile 17 den Wert von `root`, wodurch die `if`-Abfrage fehlschlägt und die innere Schleife, so wie auch im `else`-Zweig in Zeile 21 gesehen werden kann, mittels `break` beendet wird.

Bis dahin ist die „Min-Heap“-Eigenschaft für die Kinderknoten des Knotens, der im ersten Durchgang der inneren `while`-Schleife geprüft wurde und dem Indexwert von `start` in der äußeren `while`-Schleife entspricht, gegeben. Es werden dann die Kinderknoten des Knotens beim nächst niedrigen Index des Arrays bei `start` überprüft, was dann der Prüfung des nächsten Teilbaums des Heaps entspricht, wenn der Heap als ein Binärbaum interpretiert wird, so wie auch in Abbildung 23 (a) im Kapitel 5.2.1 ersichtlich.

Wenn die Operationen innerhalb der äußeren `while`-Schleife bis hoch zum ersten Element durchgeführt wurden, also `start` nach deren letzten Verringerung in Zeile 22 im Pseudocode nicht mehr der Bedingung der `while`-Schleife in Zeile 6 entspricht und größer oder gleich 0 ist, dann ist die Sortierung fertig und alle Knoten sowie Kinderknoten im Array besitzen die „Min-Heap-Order“ und repräsentieren einen validen „Min-Heap“.