UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

GPGPU-Techniken anhand von Partikelsimulationen

Masterarbeit

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

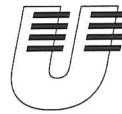
vorgelegt von
Thomas Kipshagen

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)

Zweitgutachter: Gerrit Lochmann, M.Sc.
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im November 2014

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



UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Masterarbeit
Thomas Kipshagen
(Matr.-Nr. 208 210 325)

Thema: GPGPU-Techniken anhand von Partikelsimulationen

In letzter Zeit gewinnt die Auslagerung allgemeiner Berechnungen auf die Grafikkarte immer mehr an Bedeutung. Vor allem bei der Verarbeitung großer Datensätze, beispielsweise Partikelsysteme, kann von der hohen Leistung der GPU gegenüber der CPU profitiert werden. Um die Grafikkarte für allgemeine Berechnungen zu nutzen, haben sich in den letzten Jahren verschiedene Techniken entwickelt. Diese reichen von Datentexturen für den Fragment-Shader bis zu eigenständigen APIs für GPGPU. Einige Techniken sollen im Rahmen dieser Arbeit genauer untersucht werden.

Das Ziel dieser Masterarbeit ist es, Funktionsweise und Unterschiede verschiedener GPGPU-Techniken zu recherchieren. Darauf aufbauend werden mehrere GPGPU-Verfahren implementiert, die die Berechnung für ein Partikelsystem ausführen. Die Simulation und Umsetzung des Partikelsystems ist noch offen und dient als Rahmenbedingung für die Messungen. Anhand der Implementierung sollen die Verfahren genauer analysiert und dabei insbesondere Leistungsunterschiede untersucht werden.

Die inhaltlichen Schwerpunkte der Arbeit sind:

1. Recherche
2. Einarbeitung in GPGPU (z.B. OpenCL, Transform Feedback)
3. Einarbeitung in Partikelsysteme
4. Konzeption des Grundsystems
5. Konzeption der verschiedenen GPGPU-Untersysteme
6. Implementierung der verschiedenen GPGPU-Systeme
7. Dokumentation und Bewertung der Ergebnisse

Koblenz, den 12.05.2014

T. Kipshagen
Thomas Kipshagen

S. Müller
Prof. Dr. Stefan Müller

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☒	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☒	☐

....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Zusätzlich zum Rendern wird die Rechenleistung moderner Grafikkarten immer häufiger auch für allgemeine Berechnungen (GPGPU) genutzt. Für die Umsetzung stehen verschiedene Möglichkeiten zur Verfügung, die von der Verwendung der Renderingpipeline bis zu eigenständigen Schnittstellen reichen.
In dieser Arbeit werden mit Render–To–Texture, Transform Feedback, Compute Shader und OpenCL vier verschiedene GPGPU–Methoden untersucht. Anhand von Partikelsystemen werden sie hinsichtlich der benötigten Berechnungszeit, der GPU–Auslastung, Lines of Code und Portierbarkeit miteinander verglichen. Dazu wurden sowohl das N–Körper Problem, Smoothed Particle Hydrodynamics und ein Partikelschwarm als Partikelsysteme umgesetzt. Es konnte gezeigt werden, dass insbesondere OpenCL und Compute Shader sehr gute Ergebnisse liefern.

Abstract

In addition to rendering the compute power of modern graphic cards is more and more frequently used for general computation (GPGPU). For the implementation, various options are available, ranging from the use of the rendering pipeline to independent APIs.
In this thesis four different GPGPU–methods, namely Render–To–Texture, Transform Feedback, Compute Shader and OpenCL, are evaluated. Based on particle systems, they are compared in terms of the required computation time, the GPU utilization, lines of code, and portability. Therefore, the n–body problem, smoothed particle hydrodynamics and a particleswarm were implemented as particle systems. It could be shown that OpenCL and Compute Shader deliver very good results.

Inhaltsverzeichnis

1 Einleitung **1**

2 Ziel der Arbeit **1**

3 Grundlagen **2**

3.1 GPU 2
3.2 OpenGL 3
3.2.1 GLSL 3
3.2.2 OpenGL Shader 4
3.3 GPGPU 5
3.3.1 Render–To–Texture 5
3.3.2 Transform Feedback 6
3.3.3 OpenCL 6
3.3.4 Compute Shader 8
3.4 Partikelsysteme 10
3.4.1 N–Körper Problem 11
3.4.2 Schwarmverhalten 11
3.4.3 Smoothed Particle Hydrodynamics 13

4 Konzept & Implementierung **14**

4.1 Programmablauf 14
4.1.1 N–Körper Problem 15
4.1.2 Smoothed Particle Hydrodynamics 16
4.1.3 Schwarmverhalten 16
4.1.4 Einfaches Partikelsystem 16
4.2 GPGPU–Programmierung 16
4.2.1 Initialisierung 16
4.2.2 Datenbereitstellung 18
4.2.3 Berechnungen 19

5 Evaluation und Ergebnis **20**

5.1 Testsystem und –ablauf 20
5.2 Berechnungszeit 21
5.2.1 N–Körper Problem 21
5.2.2 Schwarmverhalten 24
5.2.3 Smoothed Particle Hydrodynamics 24
5.3 GPU–Auslastung 26
5.4 Lines of Code 26
5.5 Portierbarkeit 27

6 Diskussion **28**

7 Fazit **30**

Literatur **32**

Anhang **i**
A Tabellen mit Abkürzungen i
B Weitere Messergebnisse i
C Screenshots iii

Abbildungsverzeichnis

1 OpenGL–Pipeline 4
2 OpenCL–Beispiel für Indexraum 7
3 OpenCL–Speichermodell 9
4 OpenGL Compute Shader 9
5 Schwarmverhalten: Zielflug 12
6 Schwarmverhalten: Abgrenzen 12
7 Schwarmverhalten: Anziehen 12
8 Schwarmverhalten: Ausrichten 12
9 Diagramm zur Berechnungszeit: N–Körper Problem 22
10 Diagramm zur Berechnungszeit: N–Körper Problem (3×) . . . 22
11 Diagramm zur Berechnungszeit: Partikelschwarm 23
12 Diagramm zur Berechnungszeit: Smoothed Particle Hydrodynamics 23
13 Diagramm zur Berechnungszeit: SPH + einfache Partikelsysteme 23
14 Diagramm zur GPU–Auslastung: Smoothed Particle Hydrodynamics 25
15 Diagramm zur GPU–Auslastung: N–Körper Problem (OpenCL) 25
16 Diagramm zur GPU–Auslastung: N–Körper Problem (Transform Feedback) 25
17 Screenshot: N–Körper Problem iii
18 Screenshot: N–Körper Problem (3×) iii
19 Screenshot: Smoothed Particle Hydrodynamics iv
20 Screenshot: SPH mit einfachen Partikeln iv
21 Screenshot: Partikelschwarm v

Tabellenverzeichnis

1 Zugriffsrechte unter OpenCL 8
2 Auflistung der implementierten Programme mit den jeweils verwendeten GPGPU–Methoden 15
3 Wichtige Eckdaten zum Testsystem 20
4 Tabelle der verwendeten physikalischen Größen i
5 Durchschnittliche Berechnungszeit der GPGPU–Methoden [ms] i
6 Minimale und maximale Berechnungszeit der GPGPU–Methoden [ms] i
7 Durchschnittliche Auslastung der GPU [%] ii
8 Anteil der erreichten maximalen GPU–Auslastung [%] ii

1 Einleitung

Neben dem Hauptprozessor (CPU) wird der Grafikprozessor (GPU) immer mehr zum zweiten zentralen Element der Computertechnik. Der Einsatz moderner GPUs beschränkt sich nicht mehr ausschließlich auf das Rendering, sondern ermöglicht auch die Ausführung allgemeinerer Berechnungen [BB09]. Die Verwendung der GPU für allgemeine Berechnungen wird dabei als *General Purpose Computation on Graphics Processing Unit*, kurz *GPGPU*, bezeichnet. Für die Nutzung von GPGPU stehen inzwischen wie beim Rendering verschiedene freie und kommerzielle Schnittstellen zur Verfügung. Beispiele sind OpenCL und CUDA.
Die Verwendung von GPGPU bringt einige Vorteile mit sich. Einerseits können diverse Berechnungen durch die GPU beschleunigt werden. Durch die kürzere Rechenzeit ergibt sich dann eine bessere Energieeffizienz, von der vor allem mobile Geräte wie Laptops und Smartphones profitieren. Zum anderen werden durch die Auslagerung Ressourcen auf der CPU frei, die für andere Berechnungen genutzt werden können. Jedoch muss beachtet werden, dass durch die zusätzlichen Berechnungen auf der GPU weniger Rechenleistung für die Grafik bereit steht. [Kip14]
Da alle Verfahren auf derselben GPU–Hardware lauffähig sind, ähnelt sich der grundsätzliche Aufbau vor allem bei den verschiedenen GPGPU–Schnittstellen wie OpenCL und CUDA. Werden zusätzlich die Rendering–basierten Methoden betrachtet, lassen sich zwischen allen Verfahren Unterschiede auffinden, die die Wahl der passenden GPGPU–Methode beeinflussen. Diese Unterschiede werden in [Tho10] mit Hilfe eines Raytracing–Algorithmus anhand verschiedener Szenen untersucht. OpenCL, CUDA, DirectCompute und HLSL Pixel Shader werden dabei hinsichtlich Performanz, Bildqualität, Lernkurve und Entwicklungsumgebung miteinander verglichen. Dabei kommt der Autor zu dem Schluss, dass OpenCL zwar das schnellste Ergebnis liefert, jedoch liegt bei den anderen Kriterien CUDA vorne. Zudem ist festzuhalten, dass die Bewertung für Lernkurve und Entwicklungsumgebung vorwiegend einen subjektiven Eindruck des Autors wiedergeben.

2 Ziel der Arbeit

Diese Arbeit untersucht eine Auswahl verschiedener GPGPU–Berechnungsmethoden. Dabei reicht die Palette der betrachteten GPGPU–Verfahren mit Render–To–Texture, Transform Feedback, Compute Shader und OpenCL von Grafik-Pipeline basierten Methoden bis zu eigenständigen GPGPU–Schnittstellen. Als Grundlage für die Bewertung wurden verschiedene Partikelsysteme implementiert und deren Berechnung durch die unterschiedlichen GPGPU–Methoden durchgeführt.
Die nachfolgenden Kapitel gliedern sich wie folgt:

Kapitel 3 gibt einen Überblick über die zum Verständnis der Arbeit notwendigen Grundlagen. Zunächst wird die Entwicklung und der Aufbau der GPU erklärt. Es folgt eine Einführung in OpenGL, das für diese Arbeit verwendet wurde. In Kapitel 3.3 werden die verwendeten Methoden zur Berechnung auf der Grafikkarte näher erläutert. Abschließend werden die verschiedenen Partikelsysteme, die als Anwendungsbeispiele dienen, vorgestellt.

Kapitel 4 stellt das der Arbeit zugrunde liegende Konzept und die darauf aufbauende Implementierung der Partikelsysteme vor.

Kapitel 5 beinhaltet die gemessenen Ergebnisse. Die unterschiedlichen GPGPU–Implementierungen werden dabei abhängig von Berechnungszeit, GPU–Auslastung, Lines of Code und Portierbarkeit betrachtet.

Kapitel 6 diskutiert die Ergebnisse aus Kapitel 5 und schätzt die Nutzbarkeit der verschiedenen GPGPU–Methoden ein.

Kapitel 7 schließt mit einem kurzen Fazit und gibt einen Ausblick über weiterführende Untersuchungsmöglichkeiten.

3 Grundlagen

3.1 GPU

Als fester Bestandteil in allen modernen computertechnischen Geräten wie PCs und Smartphones lässt sich ein Grafikprozessor wiederfinden. Bereits 1981 legte IBM mit dem CGA (Color Graphics Adapter) den Grundstein der Grafikkarte. Dabei fand die Berechnung der Farbe noch auf der CPU statt und das Ergebnis wurde anschließend aus dem Hauptspeicher an den Grafikspeicher übergeben. 1991 erschien mit der Matrox Impression von Matrox Electronic Systems Ltd. die erste Add–On Karte, die 3D-Berechnungen durchführen konnte. Der Begriff GPU fiel schließlich das erste Mal 1999 mit der Geforce-256 von Nvidia. Sie verwendete den ersten Einzelchip–Prozessor, der verschiedene integrierte Funktionen besaß, beispielsweise Transformation, Beleuchtung und Texturierung. Die ersten programmierbaren Shader (Vertex und Pixel/Fragment Shader) standen ab 2001 zur Verfügung. Dabei musste vorerst mit Assembler gearbeitet werden, da die ersten abstrahierten Shader–Hochsprachen wie HLSL und GLSL erst mit Direct3D 9 bzw. OpenGL 2.0 zur Verfügung standen. [BB09]
Moderne Grafikkarten besitzen eine hohe theoretische Rechenleistung. Die gute Ausnutzung der vorhandenen Leistung ist dabei jedoch „erst seit der Umstellung auf die Unified Shader–Architektur gewährleistet“[Kip14]. Frühere Grafikkarten besaßen eine feste Anzahl der verschiedenen Prozessorein-

heiten, den sogenannten Shadern (wie Vertex und Fragment Shader). Dies führte dazu, dass bei ungleicher starker Beanspruchung der verschiedenen Shader Leistungskapazitäten verloren gingen [Thi10]. Unified Shader können die vorhandene Rechenleistung wesentlich besser ausnutzen, da sie sich an die gegebene Situation anpassen. Statt einer starren Trennung zwischen den verschiedenen Shadertypen entscheidet ein Dispatcher über die Aufteilung. Dies ist nur möglich, da jeder Shader die gleichen Operationen ausführen kann. [Müc07]
Mit ihren vielen, kleinen Recheneinheiten und der hohen Bandbreitenanbindung ist die GPU vor allem auf rechenintensive, hochparallele Berechnungen spezialisiert [Tho10]. Die Funktionsweise der GPU richtet sich dabei nach der SIMT (Single Instruction Multiple Threads) Architektur. Die einzelnen Threads sind in Gruppen organisiert und werden innerhalb dieser „im Gleichschritt ausgeführt“[W-H09]. Dabei müssen alle Threads dieselben Anweisungen durchlaufen. Dies gilt beispielsweise auch für if–Verzweigungen. Muss ein Teil der Gruppe einen if–Teil durchlaufen, wird dieser zwangsläufig auch von den anderen Threads mit ausgeführt. Die so erzeugten nicht relevanten Ergebnisse werden anschließend verworfen. [W-H09]

3.2 OpenGL

OpenGL (Open Graphics Library) ist eine offene und betriebssystemunabhängige Schnittstelle zur Berechnung von 2D–/3D–Grafik. Die aktuelle Version ist OpenGL 4.50. Eingeführt wurde OpenGL von der SGI (Silicon Graphics Inc.), während es heute jedoch von der Khronos Group verwaltet wird. Mit OpenGL ES und WebGL gibt es zwei freie Ableger, die für eingebettete und mobile Systeme sowie den Webbrowser konzipiert sind. [OGL14, BB09]
Beim Rendern einer Szene werden mittels OpenGL die Primitiven der sichtbaren Objekte gezeichnet, wobei verschiedene programmierbare Shader und Fixed-Function Processing Units durchlaufen werden. Der genaue Ablauf ist in Abbildung 1 dargestellt. Eine Primitive kann als Punkt, Linie, Patch oder Polygon festlegt werden und wird durch mindestens einen Vertex definiert. Ein Vertex wiederum ist als Punkt, Endpunkt einer Linie oder Ecke eines Polygons bzw. Patches definiert und kann mit verschiedenen Daten wie Positionskoordinate, Texturkoordinate, Farbe, Normale usw. assoziiert werden. [OGL14]

3.2.1 GLSL

Für die Programmierung der in Kapitel 3.2.2 genannten Shader wird die OpenGL Shading Language, kurz GLSL, verwendet. Sie basiert auf einer C–ähnlichen Hochsprache und ist ein fester Bestandteil der OpenGL Spezifikation. Aktuell ist GLSL bei der Version 4.50 angelangt. Mit Hilfe von GLSL werden die Operationen in den Shadern spezifiziert, die in den pro-

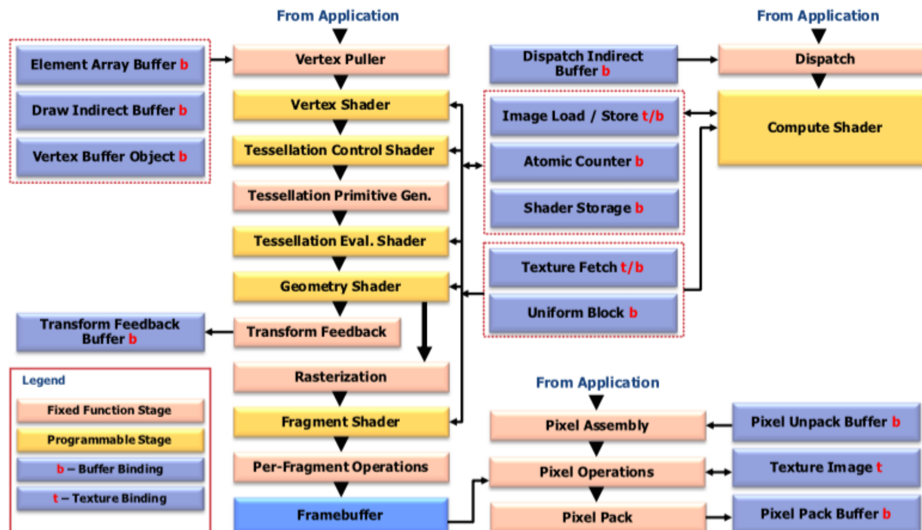


Abbildung 1: Block–Diagramm der OpenGL–Pipeline mit festen und programmierbaren Bestandteilen und den Speicherbereichen [Quelle: [OGL14]]

grammierbaren Teilen der OpenGL–Pipeline ausgeführt werden. [OGL14]

3.2.2 OpenGL Shader

In OpenGL werden fünf verschiedene Shadertypen unterschieden, die im folgenden kurz erläutert werden.

Vertex Shader Der Vertex Shader ist der vorderste programmierbare Shader in der OpenGL–Pipeline und wird für jeden Vertex eines Objekts durchlaufen. Dabei beschreibt der Shader die Operationen, die auf die Vertices und deren assoziierte Daten angewendet werden. [OGL14]

Tessellation Shader Der nächste programmierbaren Shadern ist der Tessellation Shader. Mit Hilfe dieses Shaders können Patch–Primitiven in kleinere Patch–Primitiven unterteilt werden. Der Shader besteht dabei aus drei Einzelteilen, wobei nur der erste und der letzte Teil programmierbare Shader sind. Im Gegensatz zum Vertex Shader ist die Programmierungen und Anbindung des Tessellation Shaders an ein Shaderprogramm optional. [OGL14] Für diese Arbeit ist der Tessellation Shader nicht relevant und soll deshalb nicht näher erläutert werden.

Geometry Shader Nach Vertex und Tessellation Shader folgt der Geometry Shader. Dieser wird für alle Primitiven durchlaufen. Mit Hilfe des Geometry Shaders werden vorhandene Primitive wie *Point*, *Line*, *Line with Adjacency*, *Triangle* und *Triangle with Adjacency* in neue Primitiven wie *Points*, *Line Strips* und *Triangle Strips* überführt. Genau wie der Tessellation Shader ist auch die Anbindung des Geometry Shaders an ein Shaderprogramm optional. [OGL14]

Fragment Shader Der letzte programmierbare Shader der OpenGL–Pipeline ist der Fragment Shader, der für jedes Fragment des Bildes durchlaufen wird. Dabei erhält er als Eingabe das Ergebnis der Rasterisierung und berechnet die Farbe für jedes Fragment. Das Ergebnis wird in einen Framebuffer geschrieben und ggf. auf dem Bildschirm ausgegeben. [OGL14]

Compute Shader Der Compute Shader ist eine GPGPU–Erweiterung von OpenGL und von der Rendering–Pipeline losgelöst. Der genauere Aufbau von Compute Shadern wird in Kapitel 3.3.4 vorgestellt.

3.3 GPGPU

Die Ausführung allgemeiner, nicht–grafischer Berechnungen auf dem Grafikchip wird als *General Purpose Computation on Graphics Processing Unit* (GPGPU) bezeichnet. Wie bereits in Kapitel 3.1 erwähnt, ist aufgrund der Architektur nicht jede Berechnung für den Grafikchip geeignet.
Um die allgemeinen Berechnungen auf der GPU auszuführen, wurde anfangs die programmierbare Renderingpipeline verwendet. Dieser frühe Ansatz hatte jedoch den Nachteil, dass ein Verständnis der verwendeten Pipeline vorhanden sein musste, um die Berechnungen auf die GPU anzupassen. Zwei Möglichkeiten solcher GPGPU–Berechnungen werden in den Kapiteln 3.3.1 (Render–To–Texture) und 3.3.2 (Transform Feedback) erklärt. [Müc07]
Inzwischen stehen verschiedene freie und kommerzielle GPGPU–Schnittstelle bereit, die für Berechnungen verwendet werden können [BB09]. Bei allen Schnittstellen lässt sich dabei ein ähnlicher Aufbau feststellen. Zu den wichtigsten Schnittstellen zählen OpenCL, Compute Shader und CUDA. Auf OpenCL und Compute Shader wird dabei in den Kapiteln 3.3.3 und 3.3.4 eingegangen. CUDA wird für diese Arbeit nicht näher betrachtet, da die Schnittstelle auf Nvidia Hardware beschränkt ist.

3.3.1 Render–To–Texture

Bei Render–To–Texture wird eine aktuelle Szene nicht auf dem Bildschirm ausgegeben, sondern in eine Textur gespeichert. Diese Technik wird für verschiedene Effekte wie beispielsweise Schatten verwendet. [W-Hb]

In [Lat04] wird die programmierbare Renderpipeline genutzt, um ein Partikelsystem zu berechnet. Die Berechnung der Partikelinformationen erfolgen dabei über Render–To–Texture. Dazu sind die Informationen wie Position und Geschwindigkeit in Texturen gespeichert. Sie werden auf ein bildschirmfüllendes Quadrat gelegt, wodurch jedes Partikel durch einen Fragment Shader berechnet wird. Die Ergebnisse werden anschließend wieder in Texturen geschrieben und auf die Partikel angewendet.

3.3.2 Transform Feedback

Mit Transform Feedback ermöglicht OpenGL die Verarbeitung von Geometrie mittels Renderingpipeline, wobei diese nicht komplett durchlaufen wird. Stattdessen wird der Durchlauf vor der Rasteriesierung beendet. Als Ergebnis werden Primitiven ausgegeben. [OGL14]
Transform Feedback kann vor dem eigentlichen Renderdurchlauf genutzt werden, um beispielsweise die Primitiven zu verändern oder neue Punktpositionen zu bestimmen.

3.3.3 OpenCL

OpenCL ist eine offene Programmierschnittstelle, die allgemeine Berechnungen auf der GPU erlaubt. Im Gegensatz zu anderen GPGPU–Schnittstellen ist OpenCL nicht auf den Grafikchip beschränkt, sondern kann auch auf anderen Prozessoren wie beispielsweise modernen CPUs ausgeführt werden. Des Weiteren besteht die Möglichkeit über verschiedene OpenCL–Methoden mit OpenGL und Direct3D zu interagieren. [OCL12]
Der interne Aufbau von OpenCL richtet sich nach vier verschiedenen Modellen: *Platform Model*, *Memory Model*, *Execution Model* und *Programming Model*.

Platform Model OpenCL benötigt nach dem Platform Model einen *Host* und mindestens ein *OpenCL Device*, das mit dem Host verbunden ist. Das Device besteht dabei aus mindestens einer *Compute Unit*. Diese lässt sich wiederum in mindestens ein *Processing Element* unterteilen. Während auf dem Host die OpenCL Anwendungen laufen, sind die OpenCL Devices mit den Processing Elements für die Durchführung der Berechnungen zuständig. [OCL12]

Execution Model Das Execution Model unterscheidet bei OpenCL zwischen *Kernel* und *Host Program*. Seinen Namen entsprechend wird das Host Program auf dem Host ausgeführt. „[Es] definiert den Kontext für die Kernel und verwaltet dessen Ausführung“[OCL12].
Der Kernel beinhaltet die OpenCL Funktionen, die für die Berechnung ausgeführt werden sollen. Beim Ausführen des Kernels wird durch das Host

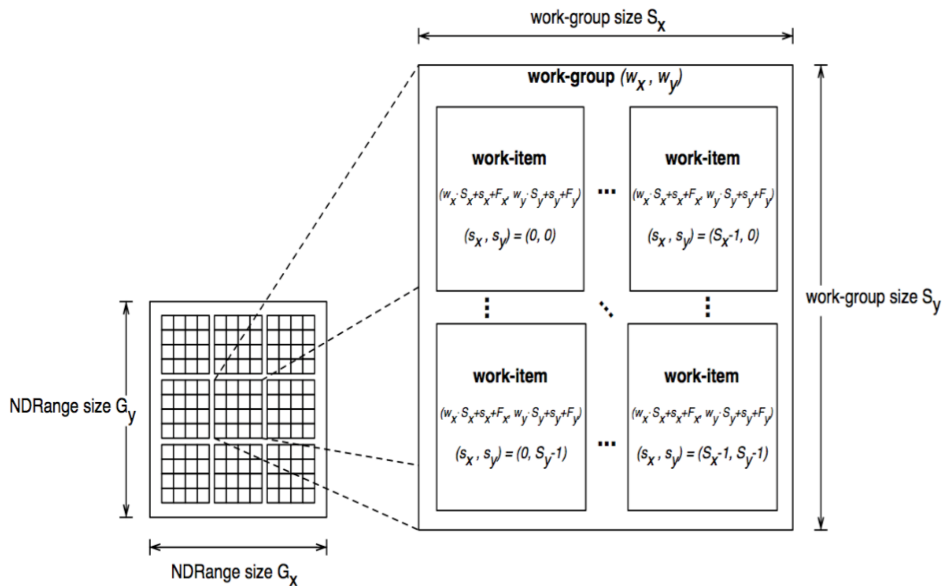


Abbildung 2: Beispiel für den NDRange–Indexraum [Quelle: [OCL12]]

Program ein *NDRange* Indexraum festgelegt, der die Anzahl der Kernel–Instanzen, der sogenannte *Work–Items*, angibt. Der Indexraum kann ein–, zwei– oder dreidimensional angeordnet werden. Zudem werden die Work–Items in *Work–Groups* zusammengefasst, deren Anordnung ebenfalls ein–, zwei– oder dreidimensional sein kann. Dabei wird eine Workgroup jeweils von einer Compute Unit bzw. die Work–Items von den Processing Elementen ausgeführt.[OCL12]
Jedes Work–Item verfügt über verschiedene Kennungen (Abbildung 2), wodurch ein Work–Item identifizierbar wird. Dabei richten sich die IDs nach der eingestellten Dimension des Indexraums und der Work–Group. Zum einen verfügt jedes Work–Item über eine eindeutige globale ID. Des Weiteren wird innerhalb einer Work–Group noch eine lokale ID vergeben. Zusätzlich verfügt auch die Work–Group über eine Kennung, die ein Work–Item zusammen mit der lokalen ID ebenfalls eindeutig identifizierbar macht. [OCL12]

Memory Model Unter OpenCL werden mit *Global Memory*, *Constant Memory*, *Local Memory* und *Private Memory* vier verschiedene Speicherbereiche bereitgestellt, die über unterschiedliche Zugriffsrechte verfügen. In Tabelle 1 werden die einzelnen Zugriffsrechte und die Möglichkeiten der Speicherreservierung durch Host und Device aufgelistet. [OCL12]
Wie aus Abbildung 3 hervorgeht, muss beim Speicherzugriff durch die Work–Items noch auf weitere Punkte geachtet werden. Durch die direkte Anbindung ist der Zugriff auf den privaten Speicherbereich auf das jeweilige Work–

Item begrenzt. Innerhalb einer Work–Group können die Work–Items mit Hilfe des lokalen Speicherbereichs Variablen miteinander teilen. Der gemeinsame Zugriff aller Work–Items auf alle Informationen ist erst durch den Global und Constant Memory möglich. [OCL12]
Für die verschiedenen Speicherzugriff gilt: Je kleiner die Speicher sind und je direkter sie an den Work–Items liegen, desto schneller sind auch die jeweiligen Zugriffe auf diese Bereiche.

	Global	Constant	Local	Private
Host	Dynamic allocation	Dynamic allocation	Dynamic allocation	no allocation
	Read/Write access	Read/Write access	no access	no access
Kernel	no allocation	static allocation	static allocation	static allocation
	Read/Write access	Read/Write access	Read/Write access	Read/Write access

Tabelle 1: Zugriffsrechte unter OpenCL

Programming Model Das Execution Model von OpenCL unterstützt datenparalleles (*data parallel*), aufgabenparalleles (*task parallel*) und hybrides Programmieren. Datenparallelität bedeutet, dass die gleichen Instruktionen zur selben Zeit auf unterschiedlichen Datenelementen ausgeführt werden. Dagegen werden bei der Aufgabenparallelität unterschiedliche Instruktionen zur selben Zeit angewendet. Für GPGPU–Programmierung ist vor allem die Datenparallelität relevant. OpenCL erfordert dabei keine strikte eins–zu–eins Beziehung zwischen Work-Item und Datenelement.

3.3.4 Compute Shader

Mit Compute Shadern besitzt OpenGL eine Erweiterung, um GPGPU–Berechnungen durchzuführen, ohne auf die Verwendung von externen Schnittstellen oder das Umfunktionieren der Renderingpipeline zurückzugreifen. Die Compute Shader sind dabei im Gegensatz zu den anderen Shadern kein direkter Bestandteil der Renderingpipeline. [OGL14]
Wie aus Abbildung 4 zu erkennen ist, ähnelt sich der interne Aufbau von OpenCL und den Compute Shadern. Die Invocations beim Compute Shader entsprechen in der Funktionsweise den Work–Items unter OpenCL und werden ebenso in Work–Groups organisiert. Darüber hinaus können die Invocations und Work–Groups auch ein–, zwei– oder dreidimensional angeordnet sein. Des Weiteren stehen die gleichen IDs wie unter OpenCL zur Verfügung. Sie richten sich ebenfalls nach der festgelegten Dimension. [OGL14, Hun13]

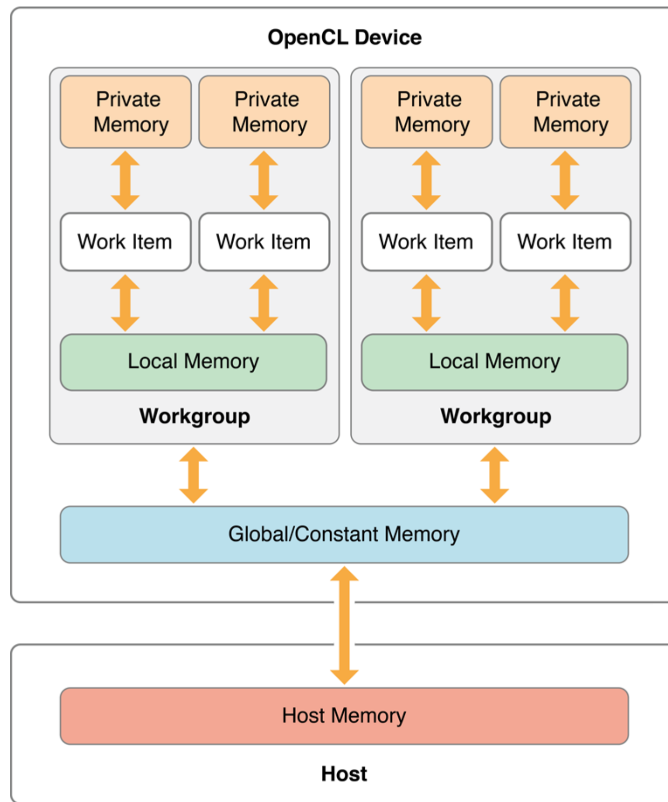


Abbildung 3: Speichermodell unter OpenCL [Quelle: [W-Ha]]

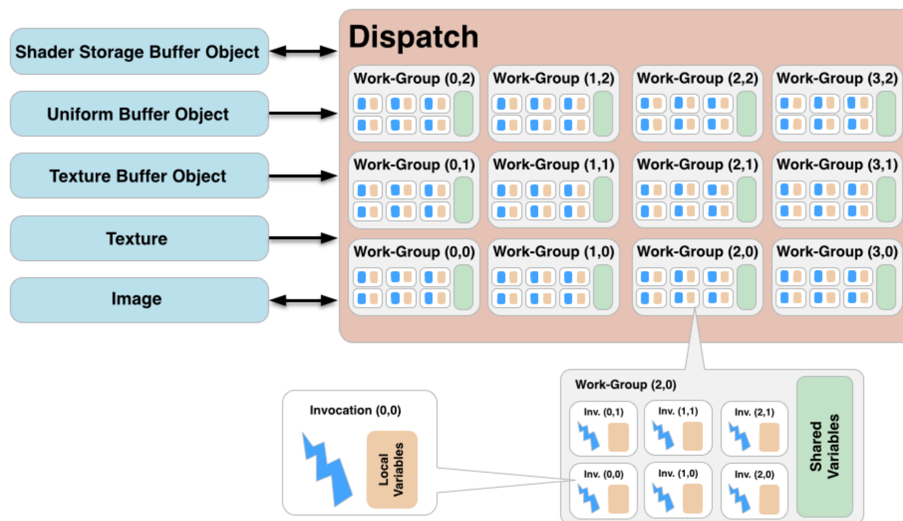


Abbildung 4: Ausführungs– und Speichermodell der OpenGL Compute Shader [Quelle: modifiziert nach [W-Ha]]

Auch der Speicherzugriff der Compute Shader weist Ähnlichkeiten zu OpenCL auf. Der Zugriff auf die Local Variables ist auf die jeweilige Invocation beschränkt. Des Weiteren besitzen alle Invocations einer Work–Group mit den Shared Variables einen gemeinsamen Speicher, über den sie kommunizieren können. Für den gemeinsamen Zugriff durch alle Invocations stehen fünf verschiedene Buffer zur Verfügung, wobei einige nur einen lesenden Zugriff gewähren. Für diese Arbeit sind nur die Shader Storage Buffer, in denen die Informationen der Partikelsysteme gespeichert sind, und die Uniform Buffer, die für die Übergabe weiterer Parameter verwendet werden, von Relevanz. [OGL14, Hun13]

3.4 Partikelsysteme

In der Computergrafik spielen Partikelsysteme eine wichtige Rolle. Sie werden unter anderem in Filmen und Computerspielen für verschiedene Spezialeffekte, darunter Nebel, Rauch, Explosionen und Feuer, genutzt. Doch auch für andere Anwendungsfälle wie beispielsweise Schwarm– und Fluidsimulationen werden Partikelsysteme verwendet. [Wil09] Damit stellt ein Partikel nichts anderes dar als einen Datenpunkt, in dem verschiedene Informationen wie beispielsweise Position, Geschwindigkeit und Masse gespeichert sind.
Die erste wissenschaftliche Beschreibung von einem Partikelsystem stammt 1983 von William T. Reeves. Reeves verwendet die Partikelsysteme zur Darstellung von *fuzzy Objects*. Ein Objekt setzt sich dabei aus einer Menge von Partikeln zusammen, die nicht als Primitiv oder Oberfläche sondern als Volumen definiert sind. Die Verarbeitung des Partikelsystems durchläuft vier Schritte pro Frame: Zunächst wird eine Anzahl von Partikeln geboren. Anschließend werden Partikel, die ihre angegebene Lebenszeit erreicht haben, gelöscht. Im dritten Schritt werden alle relevanten Werte wie beispielsweise die neue Position berechnet und auf das Partikel angewendet. Zum Schluss werden die Partikel zum Beispiel mit Hilfe von Billboards[1] gerendert. [Ree83]
Auch viele moderne Partikelsysteme funktionieren nach dem oben genannten Schema. Dabei werden inzwischen oftmals die sterbenden Partikel nicht gelöscht, sondern mit den Informationen neuer Partikel überschrieben oder, wenn die Partikel nicht mehr benötigt werden, entsprechend markiert und vom System nicht weiter verarbeitet. Durch die feste Partikelanzahl wird die Speicherverwaltung vereinfacht. Des Weiteren existieren Partikelsysteme, bei denen die gleichen Partikel dauerhaft vorhanden sein müssen, so dass der Vorgang des Generierens und Löschens übersprungen werden kann.

[1] Texturierte Fläche zur Vereinfachung der Objektdarstellung; wird beispielsweise in Computerspielen für Gras verwendet [Mül13]

3.4.1 N–Körper Problem

Mit dem N–Körper Problem werden Systeme numerisch gelöst, in denen jeder Körper mit jedem Körper interagiert. Verwendung findet es beispielsweise in der Astrophysik. Hierbei werden die Auswirkungen der Gravitationskraft berechnet, die Sterne und Galaxien aufeinander auswirken. Für diese Arbeit wurde dazu die nachfolgende Formel verwendet:

$$F_i = G * \sum_{j=1}^{N} \left(\frac{m_j * r_{ij}}{(||r_{ij}||^2 + \varepsilon^2)^{(3/2)}} \right) \tag{1}$$

Die Kraft, die auf den Körper i wirkt, ergibt sich aus einer konstanten Gravitation G und der Summe über alle Nachkörper. Die Summe setzt sich dabei aus der Masse m_j des Körpers j und dem Abstand r_{ij} zwischen i und j zusammen. Zudem wird mit ε ein zusätzlicher Faktor mit eingerechnet. Dadurch kann der Abstand zwischen zwei Körpern 0 betragen, so dass die Bedingung $i \neq j$ nicht benötigt wird. [NHP07]

3.4.2 Schwarmverhalten

In diesem Modell, dessen Grundlagen von Craig. W. Reynolds stammen, wird dem Namen entsprechend das Verhalten von Schwärmen simuliert. Typische Einsatzgebiete sind dabei Filme und Spiele, da durch das Schwarmverhalten nicht mehr die Notwendigkeit besteht jedem Individuum einzeln einen Weg zuzuweisen. Stattdessen werden Verhaltensweisen für die Schwarmmitglieder festgelegt, wodurch die Handlungsweise eines Schwarms entsteht. [Tom04]
Die Verhaltensweisen, die für diese Arbeit genutzt werden, sind *Zielflug und Fliehen*, *Abgrenzen*, *Anziehen* und *Ausrichten*. Sie werden in den Abbildungen 5 bis 8 dargestellt. Weitere Verhaltensweisen werden in [Rey87] und [Rey] beschrieben.

Zielflug und Fliehen Für dieses Verhalten wird zunächst der Vektor zwischen dem Individuum und dem anvisierten Ziel benötigt. Bei Zielflug zeigt dabei der Vektor in Richtung des Ziels, während bei Fliehen der Vektor genau in die entgegengesetzte Richtung zeigt. Anschließend wird der Zielvektor von der Geschwindigkeit abgezogen und ergibt die neue Beschleunigung. Diese wird auf die alte Geschwindigkeit gerechnet und ergibt die neue Geschwindigkeit für das Individuum.

Abgrenzen Beim Abgrenzen versuchen die Individuen einen gewissen Abstand zueinander zu bewahren und dadurch nicht zu kollidieren. Dazu werden die Vektoren zwischen einem Individuum und allen Nachbarindividuen in einer festgelegten Umgebung bestimmt. Durch die Addition aller Vektoren kommt dann der Zielvektor zustande.

Anziehen Durch Anziehen versuchen die Individuen zusammen zu bleiben. Dazu wird für jedes Individuum der Mittelpunkt bestimmt, der sich durch alle Nachbarindividuen der Umgebung ergibt. Der so errechnete Mittelpunkt stellt das Ziel für den Zielflug dar.

Ausrichten Beim Ausrichten versucht ein Individuum sich bei Geschwindigkeit und Ausrichtung an seine Nachbarindividuen aus der Umgebung anzupassen. Der Zielvektor ergibt sich dabei aus der Differenz des aktuellen Geschwindigkeitsvektors und dem gemittelten Geschwindigkeitsvektor aller betrachteten Nachbarindividuen.

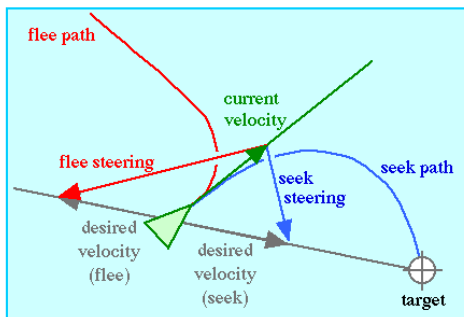


Abbildung 5: Zielflug und Fliehen: Partikel bewegt sich zum Ziel bzw. davon weg [Quelle: [Rey]]

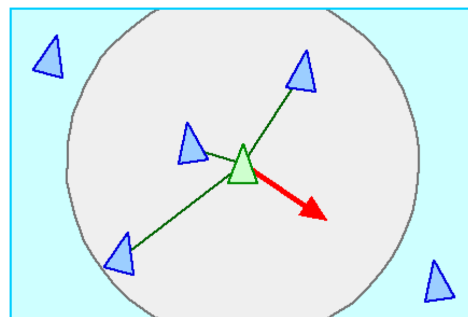


Abbildung 6: Abgrenzen: Partikel entfernt sich von Nachbarpartikeln im Umkreis [Quelle: [Rey]]

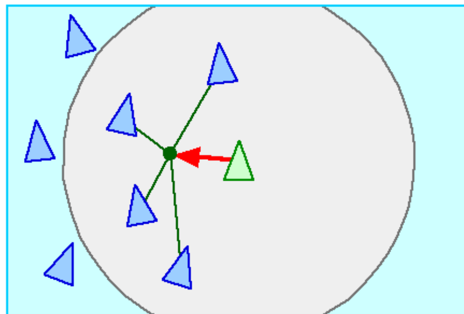


Abbildung 7: Anziehen: Partikel wird zum Mittelpunkt der Nachbarpartikel im Umkreis angezogen [Quelle: [Rey]]

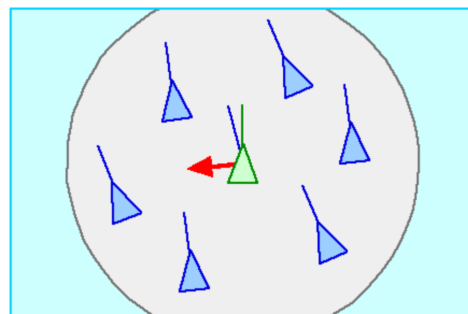


Abbildung 8: Ausrichten: Partikel richtet sich an Nachbarpartikeln im Umkreis aus [Quelle: [Rey]]

3.4.3 Smoothed Particle Hydrodynamics

Die *Smoothed Particle Hydrodynamics* (SPH) sind eine partikelbasierte Methode zur Simulation von Fluiden. Sie wurde ursprünglich von Gingold und Monaghan und ebenfalls von Lucy vorgestellt. Von Müller et al. folgte schließlich ein SPH–Ansatz für interaktive Anwendungen. [MCG03]
Wie das N–Körper Problem sind auch die SPHs ein numerisches Verfahren, wobei an den Partikelpositionen die physikalischen Größen bestimmt werden [Kle06]. Für diese Arbeit werden für jedes Partikel die Dichte, der Druck, die Viskosität und die externen Kräfte benötigt. Für jeden Durchlauf wird zunächst die Dichte bestimmt. Darauf aufbauend werden die internen Kräfte, bestehend aus Druck und Viskosität, berechnet. Zum Schluss werden sowohl die internen Kräfte, als auch die Gravitation als einzige externe Kraft auf die Partikel angewendet.
Für diese Arbeit wurden zur Berechnung der Smoothed Particle Hydrodynamics die folgenden Formeln aus [MCG03] verwendet:

Dichte:

$$\rho_i = \sum_j m_j * W(r_i - r_j, h) \tag{2}$$

Druck (von Desbrun et al. [DG96]):

$$p_i = k * (\rho_i - \rho_0) \tag{3}$$

Druckgradient:

$$f_i^{pressure} = -\sum_j m_j * \frac{p_i + p_j}{2 * \rho_j} * \nabla W(r_i - r_j, h) \tag{4}$$

Viskosität:

$$f_i^{viscosity} = \sum_j m_j * \frac{v_j - v_i}{\rho_j} * \nabla^2 W(r_i - r_j, h) \tag{5}$$

Die zu errechnenden Größen eines Partikels i ergeben sich dabei aus den Werten der lokalen Nachbarpartikel. Der Einflussradius um das Partikel i, ist durch die Smoothing Length h festgelegt. m_j ist die Masse des j-ten Nachbarpartikes. Die Position ist mit r und die Geschwindigkeit mit v angegeben. Über die Funktion W (Smoothing Kernel) wird der Einfluss durch die anderen Partikel geglättet.

Smoothing Kernels Für die verschiedenen Berechnungen werden unterschiedliche Smoothing Kernels verwendet. Die Wahl des Smoothing Kernels

hat dabei direkten Einfluss auf die Stabilität, Genauigkeit und Geschwindigkeit der Berechnungen. Dabei stehen in der Literatur eine Vielzahl an verschiedenen Kerneln zur Verfügung. Für diese Arbeit wurden die gleichen Smoothing Kernels wie in [MCG03] verwendet.

Smoothing Kernel für die Dichte:

$$W_{poly6}(r,h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & 0 \leq r \leq h \\ 0 & \text{sonst} \end{cases} \tag{6}$$

Smoothing Kernel für den Druckgradient (von Desbrun et al. [DG96]):

$$W_{spiky}(r,h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & 0 \leq r \leq h \\ 0 & \text{sonst} \end{cases} \tag{7}$$

Smoothing Kernel für die Viskosität:

$$W_{viscosity}(r,h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & 0 \leq r \leq h \\ 0 & \text{sonst} \end{cases} \tag{8}$$

4 Konzept & Implementierung

Im Rahmen dieser Arbeit wurde eine Reihe von Programmen implementiert, die jeweils unterschiedliche Partikelsysteme realisieren. Zur Umsetzung wurden in jedem Programm mehrere GPGPU–Methoden eingesetzt. Eine Auflistung der Kombinationen ist in Tabelle 2 dargestellt. In Kapitel 4.1.1 – 4.1.4 sind alle Programme näher erläutert. Screenshots der verschiedenen Programme befinden sich in Anhang C.

Um Fehler durch spezifische Änderungen für die einzelnen Umsetzungen zu minimieren, sind die aufgelisteten Partikelsysteme als eigenständige Programme implementiert. Dabei wird ein gemeinsames Grundprogramm verwendet, das für das jeweilige Partikelsystem erweitert wird. Durch die Abgrenzung konnten spezielle Änderungen schnell und flexibel vorgenommen werden, ohne auf die Einstellungen der anderen Partikelsysteme Rücksicht nehmen zu müssen.

4.1 Programmablauf

Der allgemeine Ablauf der Programme ist unabhängig von der Art des Partikelsystems und der GPGPU–Methode. Nach dem Programmstart nimmt der Benutzer verschiedene Einstellungen vor. Neben der zu verwendenden GPGPU–Methode kann der Nutzer außerdem den Bench–Modus aktivieren.

Programm	OCL	OCL+T	CS	TF+U	TF+T	RTT
N–Körper Problem	x	x	x	x	x	x
N–Körper Problem (3×)	x	x	x	x	x	x
Partikelschwarm	x		x		x	x
SPH	x		x		x	x
SPH + einf. Partikel (2×)	x		x		x	x

Tabelle 2: Auflistung der implementierten Programme mit den jeweils verwendeten GPGPU-Methoden. Die Programme umfassen zwei N–Körper Probleme mit jeweils einem bzw. drei Partikelsystemen, einen Partikelschwarm und zwei SPH–Programme, wobei im zweiten Programm zusätzlich zwei einfache Partikelsysteme mitberechnet werden. Als GPGPU–Methoden eingesetzt werden OpenCL, wobei die Daten direkt in Speicherobjekten (OCL) bzw. in Texturen (OCL+T) vorliegen, Compute Shader (CS), Render–To–Texture (RTT) und Transform Feedback, wobei die Daten einmal mittels Uniform (TF+U) und das andere Mal als Textur (TF+T) zur Verfügung stehen.

Ist der Bench–Modus aktiviert, werden die *Frames per Second* und die *Frametime* vom Programm aufgezeichnet. Zudem wird ein fünfminütiger Timer aktiviert, der das Programm automatisch mit Ablauf der Zeit beendet und die gesammelten Informationen in einer Datei speichert. Bei deaktiviertem Bench–Modus werden keine Daten aufgezeichnet und der Benutzer muss das Programm manuell beenden.
Nach der Eingabe durch den Nutzer werden der Renderer und die Shader zum Zeichnen der Partikel initialisiert. Dabei kommt neben Vertex und Fragment Shader auch der Geometry Shader zum Einsatz, der die Billboards für die Partikel erzeugt.
Anschließend werden die benötigten Partikelsysteme angelegt und zusammen in einem Array gespeichert. Beim Anlegen werden für jedes Partikel die benötigten Startwerte abhängig von der Art des Partikelsystems generiert.
Danach wird die ausgewählte GPGPU–Methode initialisiert, die anschließend in jedem Render–Durchlauf zur Berechnung der Partikel angewendet wird. Dabei werden innerhalb des Renderings erst die Partikelberechnungen ausgeführt und danach die Partikelsysteme gezeichnet.

4.1.1 N–Körper Problem

Die Partikel sind die einzelnen Körper, deren Bewegung simuliert werden soll. Die Beschleunigung wird dabei mit Hilfe von Formel 1 berechnet. Die generierten Startpositionen der Partikel werden zufällig auf einer Kugeloberfläche verteilt, während die Startgeschwindigkeit mit null initialisiert wird.
Wie aus Tabelle 2 zu entnehmen, ist das N–Körper Problem sowohl mit einem Partikelsystem als auch mit drei voneinander unabhängigen Partikelsystemen umgesetzt.

4.1.2 Smoothed Particle Hydrodynamics

Mit dem SPH wird in dieser Arbeit eine kleine sprudelnde Quelle dargestellt. Zur Berechnung der Partikel werden die in Kapitel 3.4.3 aufgeführten Formeln verwendet. Da keine Nachbarschaftssuche implementiert ist, müssen für die Berechnungen jeweils alle Partikel durchlaufen werden. Dies geschieht für jedes Partikel in zwei Schleifen, wobei mit der ersten Schleife die Dichte und mit der zweiten Schleife der Druckgradient sowie die Viskosität berechnet wird.
Die Startpositionen der Partikel sind zufällig auf einer Scheibe verteilt, wobei die Startbewegung nach oben gerichtet ist. Zudem wird im Gegensatz zu den anderen verwendeten Partikelsystemen eine Lebensdauer verwendet. Nach Ablauf der variierenden Lebensdauer erhalten die Partikel neue Startwerte, die aus vorher erzeugten Zufallstexturen stammen.

4.1.3 Schwarmverhalten

Die Partikel stellen die einzelnen Individuen des Schwarms dar und sind entweder *Jäger* oder *Gejagter*. Das Ziel der Gejagten ist es, sich mit den Nachbarn der Umgebung in einer Gruppe zusammen zu schließen. Die Jäger wiederum suchen nach der größten Gruppe in ihrer Umgebung und verfolgen diese. Dabei versuchen alle Jäger einen gewissen Abstand zueinander zu halten.

4.1.4 Einfaches Partikelsystem

Das einfache Partikelsystem (im Programm kombiniert mit SPH) erweckt den Eindruck von Regen. Die Position der Partikel wird dabei nur anhand der festgelegten Geschwindigkeit verändert. Die Startpositionen der Partikel liegen zufällig auf einem Quadrat verteilt. Die Startgeschwindigkeit ist immer nach unten gerichtet, wobei Fallwinkel und –geschwindigkeiten für die Partikel zufällig leicht variieren. Wie bei den SPHs werden auch für dieses Partikelsystem zwei Zufallstexturen für neue Startpositionen und –geschwindigkeiten erzeugt. Dabei erhält ein Partikel neue Werte, sobald es eine gewisse Höhe unterschreitet.

4.2 GPGPU–Programmierung

Jede GPGPU–Methode unterscheidet sich sowohl in der Initialisierung als auch in der Durchführung der Berechnungen. Darüber hinaus werden unterschiedliche Möglichkeiten benötigt, um die Partikeldaten bereit zu stellen.

4.2.1 Initialisierung

Im Folgenden wird kurz auf die unterschiedlichen Initialisierungen der verschiedenen GPGPU–Methoden eingegangen.

OpenCL Die Initialisierung von OpenCL erfolgt in mehreren Schritten. Zunächst werden die OpenCL–fähigen Geräte bestimmt. Anschließend wird ein *Context* erzeugt, der für diese Arbeit die Interaktion zwischen OpenCL und OpenGL ermöglicht. Danach werden noch eine *Command Queue* und die *Kernel* erzeugt. Dazu wird für die Kernel ein Programm benötigt, dass aus dem Quellcode generiert wird. Anschließend wird die Größe der Work-Groups und die Gesamtanzahl an benötigten Work-Items, die ein Vielfaches der eingestellten Work-Items pro Work-Group sein muss, auf Grundlage der Partikelanzahl bestimmt. In dieser Arbeit wird dazu entsprechend Listing 1 vorgegangen.

```
void GPUComputeOCL::initWorkGroup()
{
 //Bestimme Maximale Work-Group Größe
 clGetDeviceInfo(device,CL_DEVICE_MAX_WORK_GROUP_SIZE,
    sizeof(size_t),&maxWorkGroup,NULL);
 // Bestimme Anzahl der Work-Items pro Work-Group
 localComputeSize[0]=64;
 localComputeSize[1]=maxWorkGroup/localComputeSize[0];
 // Hilfsvariable zum Bestimmen der Gesamtanzahl an
    Work-Items
 GLuint splitCount = (particleSystemItem_p->
    returnParticleNumber() % localComputeSize[0] != 0)
  ? (particleSystemItem_p->returnParticleNumber() /
     localComputeSize[0] + 1)
  : (particleSystemItem_p->returnParticleNumber() /
     localComputeSize[0]);
 // Bestimmen der Gesamtanzahl an benötigten Work-Items
 globalComputeSize[0] = localComputeSize[0];
 globalComputeSize[1] = (splitCount % localComputeSize
    [1] != 0) ? (splitCount / localComputeSize[1] + 1)
    : (splitCount / localComputeSize[1]);
 globalComputeSize[1] *= localComputeSize[1];
}
```

Listing 1: Code zum Bestimmen der Anzahl an Work-Items pro Work-Group und der Gesamtanzahl an benötigten Work-Items (für eine 2D–Anordnung)

Compute Shader Die Initialisierung der Compute Shader erfolgt wie bei OpenGL-Shadern üblich. Nach dem Erzeugen und Kompilieren wird der Compute Shader mit einem Shaderprogramm verknüpft. Darüber hinaus wird die Menge der benötigten Work–Groups bestimmt, die sich aus der Anzahl der eingestellten Invocations pro Work–Group und der Anzahl der

Partikel ergibt. Dabei wird für die Arbeit die Anzahl der Invocations bereits mit Hinblick auf die verwendete AMD Hardware im Shader fest eingestellt und ergibt geteilt durch die Menge der Partikel die Anzahl der Work-Groups.

Transform Feedback Auch beim Transform Feedback wird zunächst ein Shaderprogramm mit einem Vertex Shader erstellt. Zudem werden die zu speichernden Ausgabewerte der Shader festgelegt. Danach werden die benötigten Transform Feedback Buffer und Zwischenspeicher mit den entsprechenden Einstellungen angelegt.

Render–To–Texture Für Render–To–Texture wird ebenfalls ein Shaderprogramm angelegt, dass einen Vertex Shader und einen Fragment Shader enthält. Anschließend werden die Framebuffer mit den entsprechenden Texturen zum Zwischenspeichern der Ausgabe bzw. der neuen Partikelwerte angelegt.

Anzahl der verwendeten Partikel In fast allen Programmen werden 12288 (64×192) Partikel erzeugt. Nur für das simulierte Schwarmverhalten werden 3072 (64×48) Partikel verwendet. Die Wahl der Partikelanzahl hängt dabei mit der Verarbeitung durch die Hardware zusammen. Bei AMD erfolgt die Verarbeitung durch Wavefronts, bei denen ein Vielfaches von 64 für die Work-Group Größe empfohlen wird. Ein weiterer Effekt ist die gute Teilbarkeit, sodass die Daten optimal in den Texturen gespeichert werden können. Des Weiteren wurde bei der Größe des Partikelsystems darauf geachtet, dass die Belastung der Grafikkarte nicht zu groß ist und dadurch die Unterschiede in der Leistung zwischen den unterschiedlichen GPGPU-Methoden sichtbar werden.

4.2.2 Datenbereitstellung

Damit die GPGPU–Berechnungen durchgeführt werden können, müssen die Daten der Partikelsysteme auf unterschiedliche Weise bereitgestellt werden.

OpenCL OpenCL bietet die Möglichkeit sich Speicherobjekte mit OpenGL zu teilen. Dadurch können beide Schnittstellen auf dem gleichen Partikelsystem arbeiten. Damit sich OpenGL und OpenCL nicht gegenseitig bei der Berechnung stören, wird für jede Berechnung das Partikelsystem durch OpenCL erst reserviert und anschließend für OpenGL freigegeben. Die Zuordnung der Berechnung eines Partikels erfolgt durch die ID der Work–Items.
Neben der Reservierung des Objekts müssen für die Berechnung zusätzlich alle Informationen per Argument für den Kernel bereitgestellt werden.
Für das N–Körper Problem existiert neben der Implementierung mit Speicherobjekten noch eine weitere, in der das Partikelsystem als Textur bereit-

gestellt wird. Als Texturkoordinate wird wie beim Speicherobjekt die ID des Work-Items verwendet.

Compute Shader Bei den Compute Shadern befinden sich die Informationen über die Partikelsysteme in Shader Storage Buffern. Diese ermöglichen sowohl den lesenden als auch schreibenden Zugriff auf alle bereitgestellten Daten. Mit Hilfe der Invocation–ID, die als Zugriffsindex dient, wird jedem Compute Shader ein zu berechnendes Partikel zugewiesen. Weitere allgemeine Informationen werden über die Uniform Buffer bereitgestellt.

Transform Feedback Da beim Transform Feedback nur der Vertex Shader verwendet wird, stehen jeweils nur die Informationen zu einem Partikel zur Verfügung. Um die restlichen Nachbarschaftsinformationen für die Berechnungen bereitzustellen, werden alle nötigen Partikeldaten in Images gespeichert und bereitgestellt. Nach der Berechnung werden die Ergebnisse zudem nicht nur auf die Partikel angewendet, sondern auch in den Images abgespeichert. Um die Informationen im Image dem Partikel zuordnen zu können, verfügt jedes Partikel über Texturkoordinaten.
Neben der Arbeit mit einer Textur gibt es bei den N–Körper Problemen noch eine zweite Variante, in der die Informationen aller Partikel über Uniform Variablen übergeben werden. Da auf Uniform Variablen nicht geschrieben werden kann, muss nach jeder Berechnung das Partikelsystem von der GPU in den Hauptspeicher gelesen und anschließend wieder als Uniform Variable auf der GPU bereitgestellt werden.

Render–To–Texture Bei Render–To–Texture sind alle Informationen des Partikelsystems in Texturen gespeichert. Die Zuweisung der Partikel auf die Fragmente erfolgt durch die interpolierte Texturkoordinate des bildschirmfüllenden Quadrats.

4.2.3 Berechnungen

OpenCL Nach dem Bereitstellen der Daten wird der aktive Kernel mit der Anzahl der benötigten Work-Items ausgeführt. Die errechneten Ergebnisse werden dabei direkt auf das mit OpenGL geteilte Partikelsystem angewendet. Bei der Variante, in der das Partikelsystem als Textur gespeichert wird, erfolgt die Übertragung der Ergebnisse auf die Partikel erst im Vertex Shader beim Rendern. Bevor jedoch die Partikel gerendert werden, müssen alle OpenCL Berechnungen fertiggestellt sein. Dies gilt auch für die Ausführung von mehreren Kerneln oder des gleichen Kernels auf mehreren Partikelsystemen.

Compute Shader Zum Ausführen der Compute Shader wird zunächst das Shaderprogramm aktiviert und anschließend die Berechnung mittels Dis-

patch mit der Anzahl der Work–Groups ausgeführt. Dadurch ähnelt der Aufruf eher der Verwendung von OpenCL als dem normalen Renderingablauf. Zudem wird im Gegensatz zu OpenCL nicht erst vor dem Rendern, sondern nach jedem Berechnungsaufruf auf das Ergebnis gewartet.

Transform Feedback Da in der Implementation von Transform Feedback nur der Vertex Shader Verwendung findet, wird zunächst die Rasterisierung verworfen. Anschließend wird für jede Berechnung der entsprechende Transform Feedback Buffer angebunden und Transform Feedback aktiviert. Der restliche Ablauf entspricht schließlich dem Rendern einer Szene.

Render–To–Texture Die Berechnung mittels Render–To–Texture erfolgt wie das einfache Rendern der Partikel. Anstatt die Partikel direkt zu zeichnen, wird ein bildschirmfüllendes Quadrat mit den Datentexturen gerendert. Zudem muss vor dem Zeichnen nicht nur der Framebuffer angebunden werden, sondern auch die Ausgabe vom Bildschirm auf die entsprechende Ergebnistextur umgeschaltet werden.

5 Evaluation und Ergebnis

5.1 Testsystem und –ablauf

Für die Messung der Berechnungsgeschwindigkeit wurden die in Kapitel 4.1 beschriebenen Programme mit den jeweiligen GPGPU–Implementierungen verwendet. Um Schwankungen bei den Messergebnissen zu verringern, wurde jede GPGPU–Methode für jedes Programm drei Mal durchlaufen und die Ergebnisse anschließend gemittelt. Die FPS wurden vom Programm selbst aufgezeichnet. Für die Messung der GPU–Auslastung kam GPU-Z 0.5.8 zum Einsatz.
Die wichtigsten Eckdaten für die verwendete Hardware sind in Tabelle 3 aufgelistet. In den nachfolgenden Kapiteln werden die durchschnittlichen Messwerte für die Berechnungszeit und die GPU–Auslastung analysiert. Weiterführende Ergebnisse sind im Anhang B beigefügt.

	Hardware	Weitere Informationen
Grafikkarte	AMD Radeon R9 290 (Sapphire Vapor-X OC)	GPU: 1030 MHz, Speicher: 4 GB-GDDR5 (2800 MHz)
Prozessor	Intel Xeon E3-1230v3	Takt: 3,30 GHz (Max: 3,70 GHz), 4 Kerne (8 Threads)
RAM	DDR3-1600	4×4.096 MB
Betriebssystem	Windows 8.1 Pro	Version: 64-Bit
Treiberversion	AMD Catalyst 14.7	

Tabelle 3: Wichtige Eckdaten zum Testsystem

5.2 Berechnungszeit

In diesem Kapitel wird die Berechnungszeit der GPU bei den unterschiedlichen GPGPU–Implementierungen in den verschiedenen Programmen näher betrachtet. Die aufgezeichneten Werten umfassen neben der GPGPU–Berechnungszeit auch die Dauer des anschließenden Renderings der Partikel. Da das Rendering für alle Implementierungen gleich ausfällt und ihr Anteil am Messergebnis im Promillebereich liegt, können die Messdaten ohne Beschränkung der Allgemeinheit als GPGPU–Berechnungszeit aufgefasst werden. Zudem muss noch angemerkt werden, dass bei der Ausführung mit OpenCL und Compute Shader in allen Programmen eine Work-Group Größe von 256 zum Einsatz kam.

5.2.1 N–Körper Problem

Im N–Körper Problem erfolgt die schnellste Berechnung mit durchschnittlich 4,52 Millisekunden (ms) durch die Compute Shader. Danach folgen die beiden OpenCL–Implementierungen, die im Schnitt 4,97 ms mit der Textur Variante und 5,05 ms mit dem direkten Verfahren benötigen. Mit einem Mittelwert von 6,91 ms kann sich Render–To–Texture zwischen Transform Feedback und OpenCL einordnet. Die beiden Transform Feedback–Varianten berechnen das N–Körper Problem mit 9,55 ms (Uniform Variante) und 11,64 ms (Textur Variante) am langsamsten. Zwar zeigt sich, dass die Uniform Variante zügiger rechnet, sie stößt aber an anderer Stellen schnell an ihre Grenzen: Für die Uniform Variablen steht nur ein verhältnismäßig kleiner Speicherbereich zur Verfügung, so dass abhängig von der Grafikhardware das Partikelsystem nicht mehr vollständig in den entsprechenden Speicher geladen werden kann.

Sollen gleichzeitig drei N–Körper Probleme vom Programm berechnet werden, gibt es eine Veränderung bei der vorderen Reihenfolge. Die Compute Shader liegen mit einem Mittelwert von 13,24 ms hinter OpenCL, die eine durchschnittliche Berechnungszeit von 9,65 ms für die Textur Variante und 9,75 ms für die direkte Variante erreichen. Danach folgt mit 20,37 ms Render–To–Texture. Die beiden Transform Feedback Varianten (Uniform Variante: 28,23 ms / Textur Variante: 34,27 ms) sind auch in diesem Programm am langsamsten.

Beim Vergleich zwischen den beiden Programmen zur Berechnung eines bzw. dreier N–Körper Probleme zeigt sich, dass durch die Verdreifachung der Berechnung auch die benötigte Zeit ungefähr verdreifacht wird. Beispielsweise wird bei der Berechnung der drei N–Körper Probleme mit den Compute Shadern eine Gesamtzeit von 13,24 ms benötigt. Damit würden zirka 4,41 ms pro N–Körper Problem benötigt, was der Messung von 4,52 ms im anderen Programm sehr nahe kommt. Eine Ausnahme bilden die beiden OpenCL Varianten, deren Berechnnugszeit nur ungefähr verdoppelt statt verdreifacht wird.

Dies könnte daran liegen, dass die Berechnungen nicht auf den Abschluss des vorherigen Partikelsystems angewiesen sind und damit die Möglichkeit besteht, die nachfolgenden Kernel früher auszuführen.
In den Abbildungen 9 und 10 wird der Verlauf der Berechnungszeit für die beiden N–Körper Probleme abgebildet. Dabei fällt auf, dass die Compute Shader und Render–To–Texture in beiden Programmen einen sehr gleichmäßigen Verlauf haben. Dem gegenüber steht die Textur Variante von Transform Feedback, deren Berechnungszeiten im Verlauf deutlich stärker schwanken. Eine Besonderheit ist unter OpenCL zu beobachten: Der Ablauf wird in beiden Varianten durch die zusätzlichen Berechnungen gleichmäßiger.

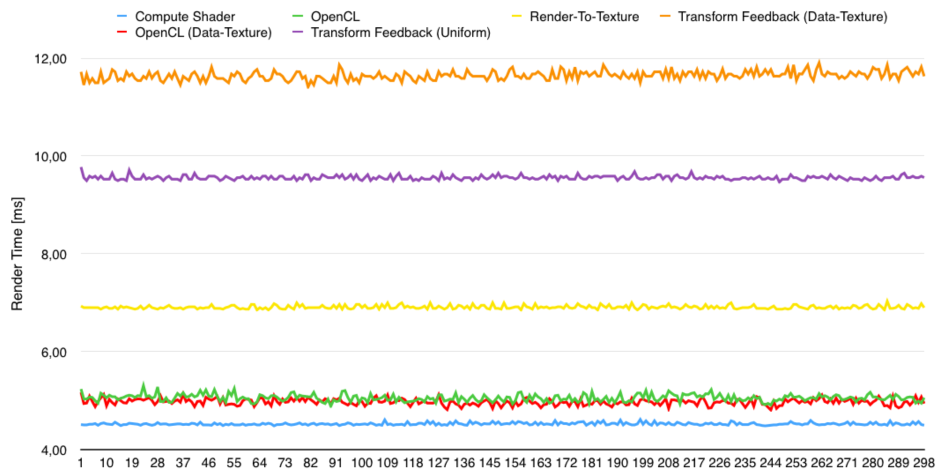


Abbildung 9: Diagramm zur Berechnungszeit beim N–Körper Problem mit einem Partikelsystem

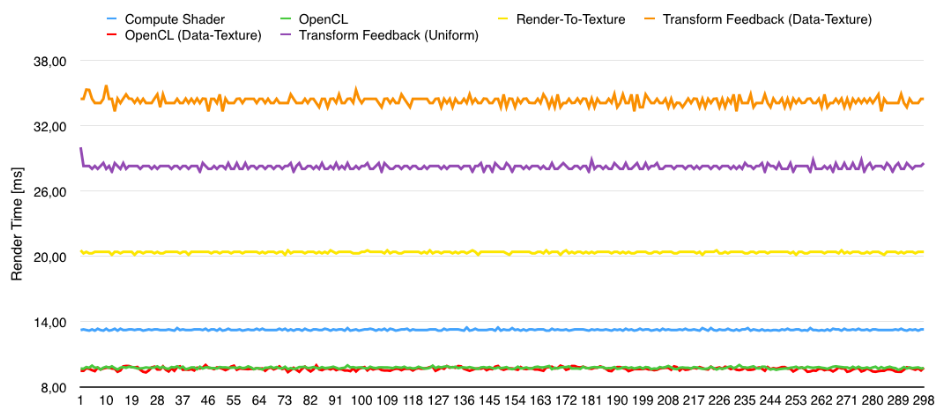


Abbildung 10: Diagramm zur Berechnungszeit beim N–Körper Problem mit drei Partikelsystemen

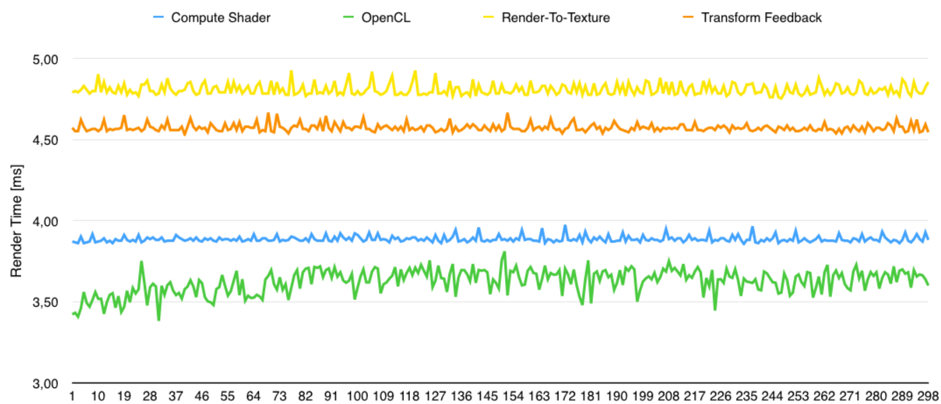


Abbildung 11: Diagramm zur Berechnungszeit beim Partikelschwarm

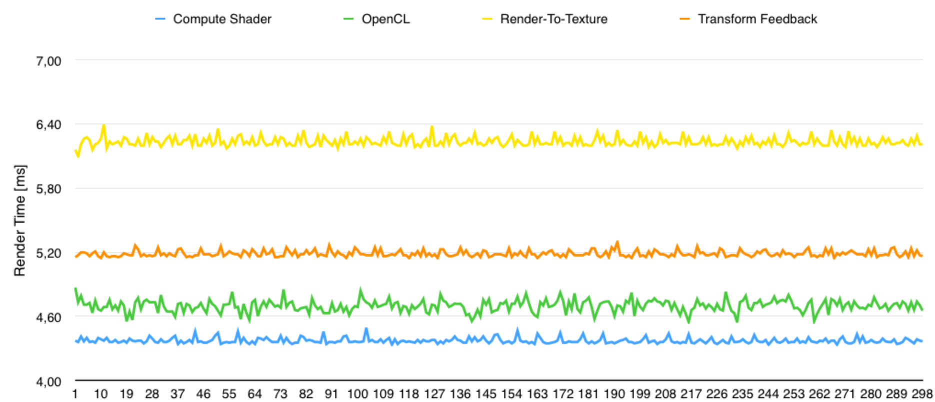


Abbildung 12: Diagramm zur Berechnungszeit bei den Smoothed Particle Hydrodynamics

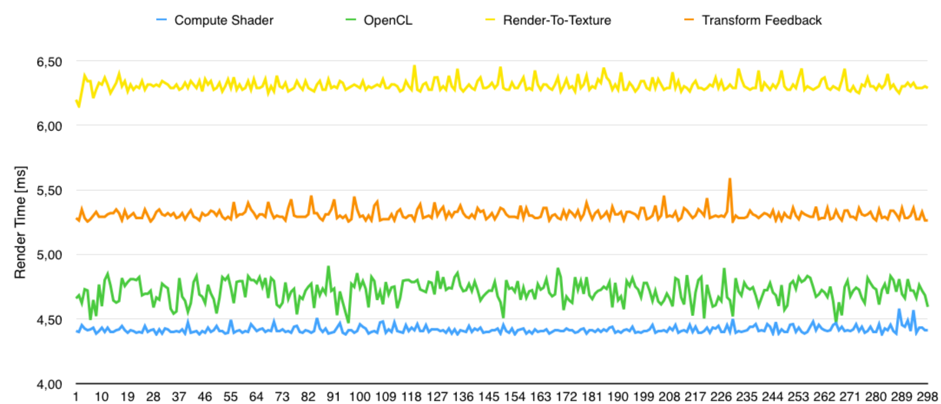


Abbildung 13: Diagramm zur Berechnungszeit beim SPH–Programm mit zwei weiteren einfachen Partikelsystemen

5.2.2 Schwarmverhalten

Wie bereits erwähnt, wird für die Schwarmsimulation eine geringere Anzahl an Partikeln verwendet. Durch den gesunkenen Rechenaufwand rücken die Messergebnisse der einzelnen GPGPU–Methoden zwar näher zusammen, trotzdem lassen sich Leistungsunterschiede feststellen. Obwohl im Programm nur ein einzelnes Partikelsystem zum Einsatz kommt, liegt OpenCL mit 3,62 ms im Schnitt knapp vor den Compute Shadern mit 3,89 ms. Auch Transform Feedback (4,57 ms) schafft es schneller als Render–To–Texture (4,81 ms) das Partikelsystem zu berechnen.
Im Gegensatz zu den vorherigen beiden Programmen gibt es bei allen Methoden wesentlich unruhigere Verläufe (siehe Abbildung 11). Dabei sticht vor allem OpenCL mit seinen starken Schwankungen bei der Berechnungszeit hervor. Wesentlich gleichmäßigere Abläufe kommen bei den Compute Shadern und Transform Feedback zustande.

5.2.3 Smoothed Particle Hydrodynamics

Die schnellste Berechnung des einzigen Partikelsystems im SPH–Programm führen die Compute Shader mit 4,37 ms durch. Nur etwas langsamer mit 4,70 ms folgt die OpenCL Implementation. Anders als bei den vorherigen Programmen verliert Transform Feedback (5,19 ms) im Vergleich zu Render–To–Texture (6,23 ms) nicht ganz den Anschluss an die durchschnittlichen Berechnungszeiten der GPGPU–Schnittstellen.
Bei der zweiten SPH–Implementierung müssen zusätzlich noch zwei weitere einfache Partikelsysteme berechnet werden. Dabei zeigt sich in den Berechnungszeiten, die sich nur geringfügig erhöhen, dass jede Methoden die beiden weiteren Partikelsysteme sehr schnell verarbeitet. Den größten Zuwachs bei der durchschnittlichen Berechnungszeit muss Transform Feedback (5,31 ms) verbuchen. Render–To–Texture benötigt 0,08 ms länger und kommt nun auf eine Gesamtzeit von 6,31 ms. Die schnellste Berechnung der Partikelsysteme erfolgt mit 4,42 ms weiterhin mit den Compute Shadern. Der geringste Zuwachs mit 0,01 ms ist unter OpenCL zu verzeichnen, sodass im Schnitt 4,71 ms zur Berechnung benötigt werden.
Wie bei den Schwarmpartikeln sticht vor allem OpenCL in beiden Programmen mit der ungleichmäßigsten Berechnungszeit hervor (siehe Abbildung 12 und Abbildung 13). Sowohl die Compute Shader als auch Transform Feedback und Render–To–Texture haben in beiden Programmen einen ähnlichen Verlauf, wobei es bei Transform Feedback und Render–To–Texture durch die zusätzlichen Partikelsysteme etwas häufiger zu Ausschlägen bei der Berechnung kommt.

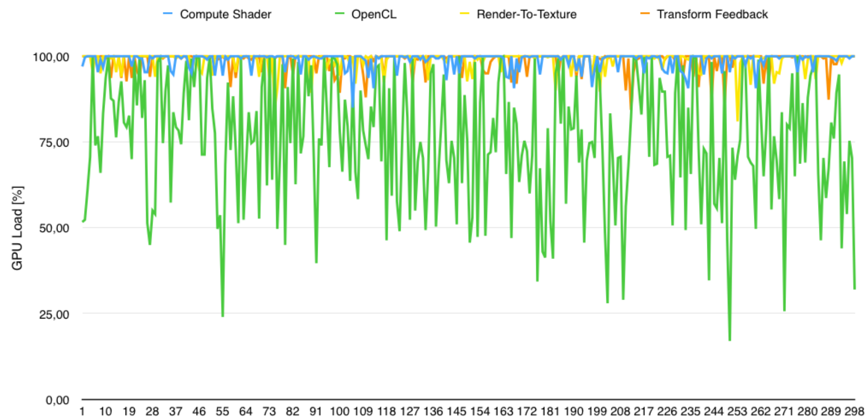


Abbildung 14: Diagramm zur GPU–Auslastung im SPH–Programm

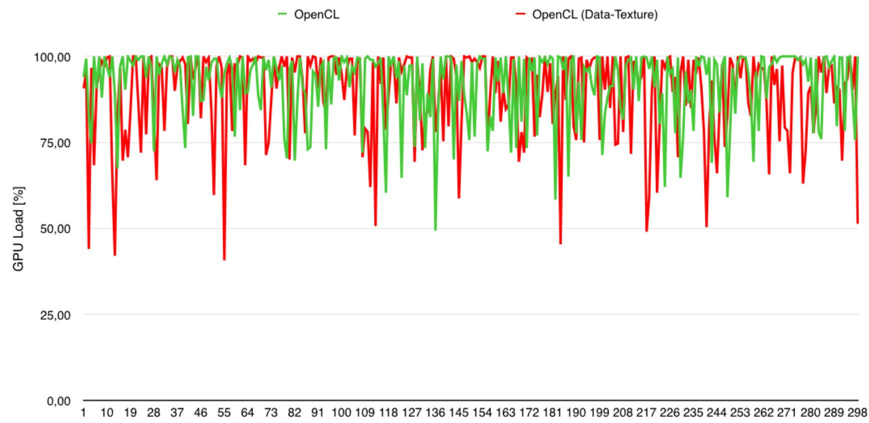


Abbildung 15: Diagramm zur GPU–Auslastung von beiden OpenCL–Varianten im N–Körper Programm mit drei Partikelsystemen

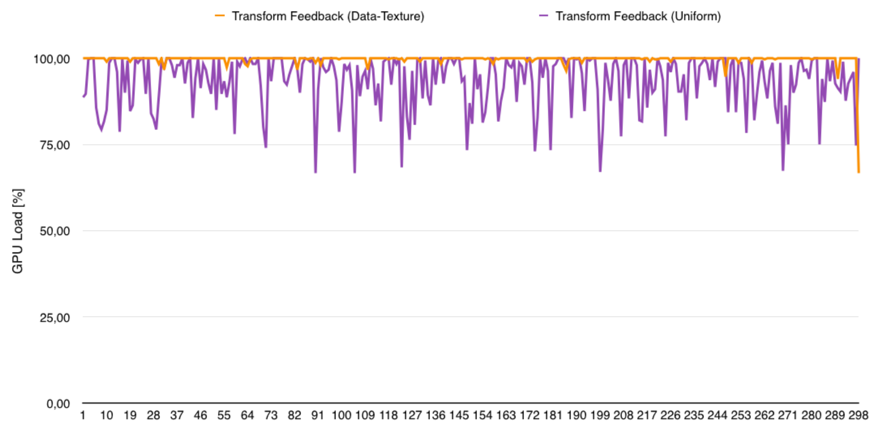


Abbildung 16: Diagramm zur GPU–Auslastung von beiden Transform Feedback–Varianten im N–Körper Programm mit drei Partikelsystemen

5.3 GPU–Auslastung

Neben der Rechenzeit wurden auch Messdaten zur Auslastung der GPU aufgezeichnet. Dabei zeigt sich, dass die Compute Shader, Render–To–Texture und Transform Feedback in allen Programmen eine durchschnittliche Auslastung von 98 - 99 % erreichen. Die Uniform Variante von Transform Feedback, die nur bei den Programmen zum N–Körper Problem verwendet wird, erzielt eine Auslastung von 91,81 % (ein Partikelsystem) und 93,51 % (drei Partikelsysteme) im Schnitt.
Im Gegensatz zu den bereits genannten Methoden fällt die durchschnittliche Auslastung mit OpenCL geringer aus. Mit 67,24 % wird der niedrigste Wert im Partikelschwarm–Programm erzielt. Die beiden SPH–Programme erreichen eine durchschnittliche Auslastung von rund 76 %. Die Betrachtung der beiden Programme zum N–Körper Problem zeigen aber auch, dass durch die zusätzlichen Berechnungen die Auslastung um zirka 10 % steigen kann. Dadurch wird bei der Umsetzung mit drei Partikelsystemen eine Auslastung von 89,63 % für die Textur Variante und 92,11 % für die direkte Methode erreicht. Dabei hat die gesteigerte Auslastung vermutlich ebenfalls einen positiven Effekt auf die in Kapitel 5.2.1 erläuterte, weniger gestiegene Berechnungszeit der drei Partikelsysteme im N–Körper Programm.
Der niedrige Durchschnittswert bei OpenCL kommt auch durch die ungleichmäßige Auslastung zustande. Abbildung 14 zeigt stellvertretend für alle Programme den Auslastungsverlauf des SPH–Programms. Daraus wird ersichtlich, dass es bei OpenCL durchgehend zu starken Schwankungen kommt, während sich die anderen Methoden sehr gut im oberen Auslastungsbereich halten können.
Wie aus den Abbildungen 15 und 16 hervorgeht, wird durch die Nutzung des Texturspeichers eine bessere Auslastung bei OpenCL und noch mehr bei Transform Feedback erreicht. Bei Transform Feedback lässt sich dies vermutlich unter anderem auf die zahlreichen Kopiervorgänge zwischen CPU und GPU zurückführen. Dies müsste in weiteren Tests geklärt werden.

5.4 Lines of Code

Dieses Kapitel beschäftigt sich mit der benötigten Anzahl an Instruktionszeilen für die verschiedenen GPGPU–Methoden. Dazu wird neben der Initialisierung und den Ausführungsbefehlen auch der Code für die verschiedenen Shader und Kernel miteinander verglichen. Exemplarisch wird die Implementierung des N–Körper Problems betrachtet.

Initialisierung Für die Initialisierung benötigen die Compute Shader mit 11 Zeilen die wenigsten Instruktionszeilen. Am anderen Ende befindet sich Render–To–Texture, welches 38 Zeilen benötigt. Mit 20 Zeilen sind die Initialisierungen der beiden OpenCL Varianten gleich lang. Dies trifft auch für

die zwei Transform Feedback–Implementationen zu, die jeweils 24 Instruktionszeilen benötigen.
Dabei muss angemerkt werden, dass sich die Angaben nur auf die Initialisierung der GPGPU–Methoden beziehen. Für die Textur Varianten von Transform Feedback und OpenCL sowie für Render–To–Texture werden die Informationen des Partikelsystems zusätzlich noch als Datentextur benötigt. Dadurch fallen in diesen Methoden zusätzliche 17 Instruktionszeilen gegenüber den anderen Methoden an.

Ausführungsbefehl Genau wie bei der Initialisierung werden auch bei den Ausführungsbefehlen die wenigsten Zeilen durch die Compute Shader (8 Zeilen) und die meisten von Render–To–Texture (23 Zeilen) benötigt. Während die direkte Variante von OpenCL mit 13 Zeilen auskommt, braucht die Textur Variante diesmal zwei Insturktionszeilen mehr. Die Textur Variante von Transform Feedback benötigt nur 1 Zeile mehr als die Uniform Variante und kommt dabei auf 18 Codezeilen.

Berechnen durch Shader & Kernel Innerhalb der Shader bzw. Kernel liegt die Uniform Variante von Transform Feedback mit 17 Zeilen vorne. Die Textur Variante von Transform Feedback benötigt 23 Instruktionszeilen. Ebenso viele benötigt auch die normale Variante von OpenCL, während die Textur Variante auf 26 Zeilen kommt. Render–To–Texture benötigt im Fragment Shader wie die Compute Shader 24 Zeilen. Dazu kommen bei Render–To–Texture weitere 5 Instruktionszeilen für den Vertex Shader.

5.5 Portierbarkeit

Für die Portierbarkeit von Code zwischen den einzelnen GPGPU–Techniken müssen zunächst die Initialisierung, die Bereitstellung der Daten und die Befehle zur Ausführung an die entsprechende Methode angepasst werden. In diesem Kapitel soll jedoch auf die Portierbarkeit der Shader und Kernel näher eingegangen werden.
Vor allem die Portierung zwischen den OpenGL basierten Ansätzen wird durch die gleiche Syntax wesentlich vereinfacht. Dadurch können besonders beim eigentlichen Verarbeitungsalgorithmus schnell große Teile ohne weitere Umschweife übernommen werden. Die größte Aufmerksamkeit muss auf das Lesen und Schreiben der Informationen gerichtet werden, da hierbei unterschiedliche Speichertypen zum Einsatz kommen.
Die Portierung von OpenGL Shadern auf OpenCL ist schon etwas schwerer. Neben dem Zugriff auf die Informationen muss sich auch mit der allgemeinen Syntax bei den Befehlen beschäftigt werden. Dabei gibt es zwischen OpenCL und OpenGL häufig passende Gegenstücke wie vecN(OpenGL) und floatN(OpenCL). Zudem besitzt OpenCL beispielsweise einige Funktionen, die über OpenCL spezifische und herstellerspezifische Alternativen verfügen.

Dabei kann die zweite Möglichkeit, deren Funktionen mit *native_* oder *fast_* gekennzeichnet sind, für eine schnellere aber eventuell auch ungenauere Berechnung genutzt werden.
Gerade die Portierung von OpenCL auf Compute Shadern und umgekehrt hat gegenüber den anderen Shadern einen gewissen Vorteil. Beide Methoden basieren auf einem ähnlichen Speichermodell, dessen geschickte Ausnutzung eine Berechnung erheblich beschleunigen kann. Dieser Geschwindigkeitsvorteil wird durch eine geschickte Portierung direkt übernommen und muss für den jeweiligen Code nicht neu ausgearbeitet werden.

6 Diskussion

Wie bereits in Kapitel 3.1 erwähnt, eignet sich aufgrund der Architektur nicht jede Berechnung für die GPU. Vor allem hochparallele Berechnungen wie beispielsweise das Filtern von Bildern eigenen sich für eine Auslagerung auf die Grafikkarte. Auch die Datenmenge muss ausreichend groß sein, damit die Übertragungszeiten keinen negativen Einfluss auf die gesamte Verarbeitungszeit haben und damit die Verwendung der CPU sinnvoller wäre. Dieses Problem tritt jedoch nicht mehr auf, wenn der Grafikchip im Prozessor integriert und im Idealfall ein gemeinsamer Zugriff auf dieselben Speicherbereiche möglich ist.
Zwar wird in dieser Arbeit eine separate Grafikkarte für die Berechnungen verwendet, trotzdem spielt die Übertragungszeit und die verwendete Datenmenge in den meisten Fällen nur eine untergeordnete Rolle. Die berechneten Ergebnisse werden nicht durch die CPU weiter verarbeitet, sondern verbleiben für das anschließende Rendering direkt auf der GPU. Gerade diese Faktoren würden jedoch bei einem Vergleich mit einer CPU Variante eine entscheidende Rolle spielen. Ein derartiger Ansatz wird in dieser Arbeit deshalb nicht weiter verfolgt. Stattdessen sollte mit Hilfe von verschiedenen Partikelsystemen festgestellt werden, welche GPGPU–Methoden sich für allgemeine Berechnungen am besten eignen.
Dient die Berechnungszeit als Maßstab, liegen Compute Shader und OpenCL vor Transform Feedback und Render–To–Texture. In dieser Arbeit konnten sogar bis zu doppelte Geschwindigkeiten durch die GPGPU–Schnittstellen erreicht werden. Um diese guten Berechnungszeiten zu erreichen, muss jedoch der Code mit Hinblick auf das komplexere Ausführungs– und Speichermodell von Compute Shadern und OpenCL optimiert werden.
Dabei ist vor allem die gut Ausnutzung des schnellen, lokalen Speicherbereiches wichtig, um die Lese– und Schreibzugriffe auf den wesentlich langsameren globalen Speicher zu reduzieren. Eine gute Möglichkeit ist die Aufteilung des Zugriffs auf den globalen Speicher, indem die einzelnen Work–Items parallel verschiedene Daten laden, die dann anschließend im geteilten Work–Group Speicher ablegen werden. Dadurch muss jedes Work–Item nur noch für

einen Bruchteil der benötigten Daten auf den langsamen Speicher zugreifen, während die restlichen Informationen schnell über den geteilten Speicherbereich der Work–Groups abrufbar sind. Um noch weitere Leistung für die Berechnungen bereitzustellen, muss eine gute Work–Group Größe gefunden werden, die jedoch unter anderem von der eingesetzten Hardware abhängt.
Auch abseits der Berechnungszeit haben die beiden GPGPU–Schnittstellen bei der Bereitstellung der Daten und bei den Lines auf Code einen Vorsprung. Vor allem die Compute Shader benötigen wesentlich weniger Instruktionen als die anderen Methoden. Dazu kommt gerade bei den implementierten Partikelsystemen hinzu, dass sowohl die Compute Shader als auch OpenCL ohne Umschweife direkten Zugriff auf alle Partikel haben.
Neben den oben genannten Punkten kann auch das benötigte Grundwissen als weiteres Kriterium für die Wahl einer geeigneten GPGPU–Methode herangezogen werden. Beim benötigten Grundwissen können zwei Ausgangspunkte betrachtet werden. Im ersten Fall werden die Berechnungen in einem Programm durchgeführt, in dem auch ein OpenGL–Rendering stattfinden soll. Hierbei ergibt sich vor allem für die OpenGL–basierten Methoden der Vorteil, dass für das Rendering bereits Grundwissen vorhanden ist und gegebenenfalls wie bei den Compute Shadern nur noch geringfügig erweitert werden muss. Im Fall das nur eine allgemeine Berechnung durchgeführt wird und kein OpenGL–Grundwissen vorhanden ist, könnte die Eigenständigkeit der OpenCL–Schnittstelle als Vorteil gewertet werden.

Die genannten Punkte zeigen, dass Render–To–Texture zwar als erste Methode GPGPU–Berechnungen ermöglichte, aber insgesamt inzwischen nicht mehr mithalten kann. Die Stärke dieser Methode liegt vielmehr in der Nutzung für grafische Effekte.
Auch Transform Feedback scheint für andere Aufgabengebiete besser geeignet zu sein. Gegenüber den anderen Methoden bietet Transform Feedback den Vorteil Primitiven auf der GPU dauerhaft verändern zu können, was verschiedene Möglichkeiten eröffnet. Beispielsweise könnte ein Partikelsystem dynamisch vergrößert oder verkleinert werden. Dabei könnte zum einen Bandbreite eingespart werden, da weniger Daten übertragen werden müssen. Zum anderen könnte auch Speicher eingespart werden, da dauerhaft tote Partikel gelöscht und später wieder neu erzeugt werden, wobei eine erneute Übertragung auf die GPU nicht notwendig wäre.
Insgesamt zeigt sich, dass die speziellen GPGPU–Schnittstellen am besten für die allgemeinen Berechnungen auf der Grafikkarte geeignet sind. Eine klare Präferenz für Compute Shader oder OpenCL lässt sich jedoch nicht erkennen. Die Compute Shader besitzen den Vorteil, dass sie als Bestandteil von OpenGL grafische Anwendungen mit allgemeinen Berechnungen unterstützen können, ohne dass eine zusätzliche Schnittstelle notwenig wird. Zudem ermöglichen gegebenenfalls vorhandene OpenGL–Kenntnisse einen schnelleren und einfacheren Einstieg für den Entwickler. Ein Negativpunkt,

der jedoch im Laufe der Zeit noch geringer zum Tragen kommen wird, ist die notwendige Unterstützung der entsprechenden OpenGL Version durch Software und Hardware zur Verwendung der Compute Shader.
Als eigenständige Schnittstelle schneidet damit OpenCL auf den ersten Blick schlechter ab. Jedoch relativiert sich bei nicht–grafischen Anwendungen der Nachteil, da in diesem Fall OpenGL eine zusätzliche Schnittstelle darstellen würde. Zudem muss auch bei grafischen Anwendungen die eigenständige Schnittstelle nicht unbedingt ein Nachteil sein: OpenCL ermöglicht eine Interaktion sowohl mit OpenGL als auch mit Direct3D, so dass unabhängig von der Rendering–Schnittstelle die gleichen Ergebnisse mittels OpenCL zur Verfügung stehen. Für die Kombination mit OpenCL wird nur eine Grafikschnittstelle ab OpenGL 3.0 bzw. Direct3D 9 benötigt. Jedoch ist zu beachten, dass die verwendete Software und Hardware OpenCL ebenfalls unterstützen muss. Darüber hinaus bietet OpenCL zwar mehr Funktionen als die Compute Shader, nimmt jedoch auch den Entwickler mehr in die Verantwortung. Ein Punkt dabei sind Funktionen, bei denen der Entwickler zwischen Genauigkeit und Geschwindigkeit entscheiden muss.

7 Fazit

In dieser Arbeit wurden verschiedene GPGPU–Berechnungsmethoden mit Hilfe mehrerer Partikelsysteme näher untersucht. Dabei zeigen die Ergebnisse, dass sowohl die Compute Shader als auch OpenCL wesentlich besser für die Berechnung der Partikelsysteme geeignet sind als Transform Feedback oder Render–To–Texture. Es ist anzunehmen, dass sich dieser Sachverhalt auch auf andere allgemeine Berechnungen übertragen lässt. In den aufgenommenen Daten konnten die beiden GPGPU–Schnittstellen bis zu doppelte Geschwindigkeiten gegenüber den anderen Methoden erreichen. Diese müssen jedoch mit einer höheren Optimierungsarbeit, besonders im Hinblick auf ein komplexeres Speicher– und Ausführungsmodell, erkauft werden.
Zudem ist erkennbar, dass sich bei nicht–grafischen Anwendungen die Verwendung von OpenCL empfiehlt; bei grafischen Anwendungen muss abgewogen werden zwischen Compute Shadern, deren Vorteile in der Erweiterung der Grafikschnittstelle OpenGL und der daraus resultierenden einfachen Nutzung liegen, und OpenCL, das aufgrund seiner Unabhängigkeit höhere Flexibilität in der Programmierung bietet.
Es ist abzuwarten, wie sich die einzelnen Schnittstellen in zukünftigen Versionen weiterentwickeln werden. Eine große Rolle spielen dabei auch optimierte Treiber, die zu höheren Leistungen führen sollten.

Ausblick In weiteren Untersuchungen ist zu überprüfen, ob sich das Verhalten der verschiedenen GPGPU–Methoden, wie es sich im Rahmen dieser Arbeit bisher abgezeichnet hat, abhängig von der Partikelanzahl sowie von

der Menge gleichzeitig simulierter Partikelsysteme verändert. Darüber hinaus sollten zusätzliche Szenarien verwendet werden, um den für Partikelsysteme gewonnenen Eindruck auch für andere allgemeine Berechnungen zu bestätigen. Unabhängig davon wären auch weitere Untersuchungen mit exklusiven Schnittstellen wie CUDA (Nvidia) oder Mantle (AMD) interessant, da diese durch ihre besondere Hardwarenähe gegebenenfalls Vorteile gegenüber den bisher getesteten Methoden aufweisen.

Literatur

[BB09] Borgo, Rita ; Brodlie, Ken: State of the Art Report on GPU. In: *Visualization & Virtual Reality Research Group report* (2009), S. 31

[DG96] Desbrun, Mathieu ; Gascuel, Marie-Paule: *Smoothed particles: A new paradigm for animating highly deformable bodies.* Springer, 1996

[Hun13] Hunz, Jochen: *The Possibilities of Compute Shaders - an Analysis*, Universität Koblenz-Landau, Bachelorarbeit, 2013

[Kip14] Kipshagen, Thomas: *SLAM auf der GPU*, Universität Koblenz-Landau, Forschungsseminar: Markerloses Tracking für Augmented Reality, 2014

[Kle06] Kley, Wilhelm: *Kapitel 7: Smoothed Particle Hydrodynamics.* 2006

[Lat04] Latta, Lutz: *Building a million particle system.* Game Developers Conference, 2004

[MCG03] Müller, Matthias ; Charypar, David ; Gross, Markus: Particle-based fluid simulation for interactive applications. In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* Eurographics Association, 2003, S. 154–159

[Müc07] Mücke, Florian E.: Analyse GPU-basierter Feature Tracking Methoden für den Einsatz in der Augmented Reality. (2007)

[Mül13] Müller, Stefan: *Geometry- und Tessellation-Shader.* Vorlesung Echtzeit Rendering, 2013

[NHP07] Nyland, Lars ; Harris, Mark ; Prins, Jan: Fast n-body simulation with cuda. In: *GPU gems* 3 (2007), Nr. 1, 677–696. `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch31.html`, Abruf: 27.10.2014

[OCL12] *The OpenCL Specification.* The Khronos Group Inc., 2012

[OGL14] *The OpenGL Graphics System: A Specification.* The Khronos Group Inc., 2014

[Ree83] Reeves, William T.: Particle systems—a technique for modeling a class of fuzzy objects. In: *ACM SIGGRAPH Computer Graphics* Bd. 17 ACM, 1983, S. 359–375

[Rey] Reynolds, Craig W.: *Steering Behaviors For Autonomous Characters*. http://www.red3d.com/cwr/steer/gdc99/, Abruf: 27.10.2014

[Rey87] Reynolds, Craig W.: Flocks, Herds, and Schools: A Distributed Behavioral Model. In: *ACM SIGGRAPH Computer Graphics* Bd. 21 ACM, 1987, 25–34

[Thi10] Thiel, Philipp: *Einführung in GPU Computing*, Fachhochschule Aachen Standort Jülich, Seminar, 2010

[Tho10] Thomas, Anoop R.: *Parallel Ray Tracing-Analysis of GPU Platforms*, Rochester Institute of Technology, Masterarbeit, 2010

[Tom04] Tomm, Christian: *Ein Modell zur Simulation der Bewegung von Schwärmen*, Universität Ulm, Proseminar, 2004

[W-Ha] *Memory Objects in OS X OpenCL*. https://developer.apple.com/library/mac/documentation/Performance/Conceptual/OpenCL_MacProgGuide/OpenCLLionMemoryObjects/OpenCLLionMemoryObjects.html, Abruf: 27.10.2014

[W-Hb] *OpenGL Framebuffer Objects*. http://www.lighthouse3d.com/tutorials/opengl-short-tutorials/opengl_framebuffer_objects/, Abruf: 27.10.2014

[W-H09] *GPGPU Computing - ein Überblick für Anfänger und Fortgeschrittene*. http://www.planet3dnow.de/vbulletin/showthread.php?t=362621. Version: 5 2009, Abruf: 27.10.2014

[Wil09] Willmitzer, Florian: *Volumeneffekte - Partikelsysteme*, Hochschule München, Hauptseminar, 2009

Anhang

A Tabellen mit Abkürzungen

Zeichen	Beschreibung
m	Masse
G	Gravitation
r	Position von Partikel
r_{ij}	Strecke zwischen Partikel i und Partikel j
ρ	Dichte
p	Druck
f	Kraft

Tabelle 4: Tabelle der verwendeten physikalischen Größen

B Weitere Messergebnisse

	N–Körper	N–Körper (3×)	Partikel-schwarm	SPH	SPH + einfache Partikel
CS	4,52	13,24	3,89	4,37	4,42
OCL	5,05	9,75	3,62	4,70	4,71
OCL+T	4,97	9,65	-	-	-
RTT	6,91	20,37	4,81	6,23	6,31
TF+U	9,55	28,23	-	-	-
TF+T	11,64	34,27	4,57	5,19	5,31

Tabelle 5: Durchschnittliche Berechnungszeit der GPGPU–Methoden [ms]

(min/max)	N–Körper	N–Körper (3×)	Partikel-schwarm	SPH	SPH + einfache Partikel
CS	4,48/4,61	13,16/13,46	3,86/3,97	4,34/4,49	4,38/4,58
OCL	4,88/5,30	9,55/10,00	3,38/3,81	4,54/4,87	4,47/4,91
OCL+T	4,80/5,16	9,32/10,00	-/-	-/-	-/-
RTT	6,85/7,03	20,14/20,55	4,75/4,93	6,10/6,40	6,15/6,47
TF+U	9,46/9,77	27,78/30,02	-/-	-/-	-/-
TF+T	11,41/11,91	33,33/35,74	4,53/4,67	5,15/5,30	5,25/5,59

Tabelle 6: Minimale und maximale Berechnungszeit der GPGPU–Methoden [ms]

	N–Körper	N–Körper (3×)	Partikel-schwarm	SPH	SPH + einfache Partikel
CS	98,98	99,19	98,82	98,91	98,82
OCL	81,18	92,11	67,24	75,55	76,48
OCL+T	79,58	89,63	-	-	-
RTT	98,75	99,06	99,04	98,88	98,41
TF+U	91,83	93,51	-	-	-
TF+T	99,09	99,69	98,75	98,73	98,90

Tabelle 7: Durchschnittliche Auslastung der GPU [%]

	N–Körper	N–Körper (3×)	Partikel-schwarm	SPH	SPH + einfache Partikel
CS	70,13	75,17	65,10	63,76	69,13
OCL	2,01	18,79	0,00	3,69	0,67
OCL+T	1,01	17,11	-	-	-
RTT	68,46	75,84	67,45	70,47	63,76
TF+U	24,83	25,17	-	-	-
TF+T	69,46	85,23	66,11	66,11	66,44

Tabelle 8: Anteil der erreichten maximalen GPU–Auslastung [%]

C Screenshots

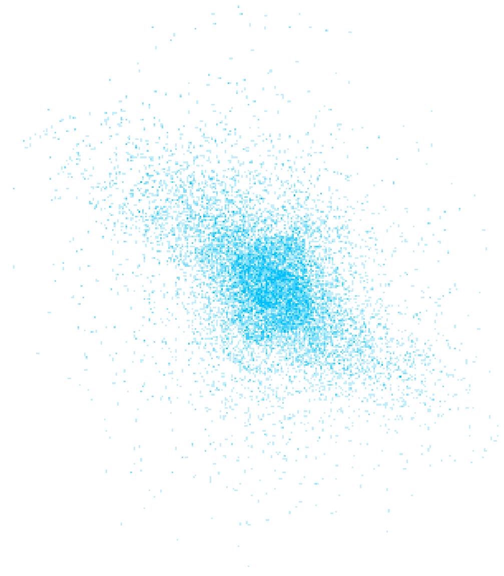


Abbildung 17: Screenshot des N–Körper Problems mit einem Partikelsystem

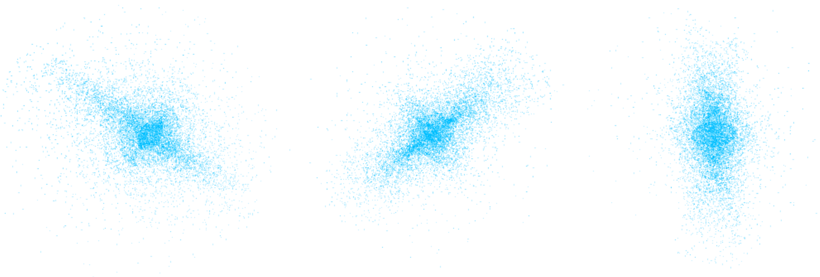


Abbildung 18: Screenshot des N–Körper Problems mit drei Partikelsystemen

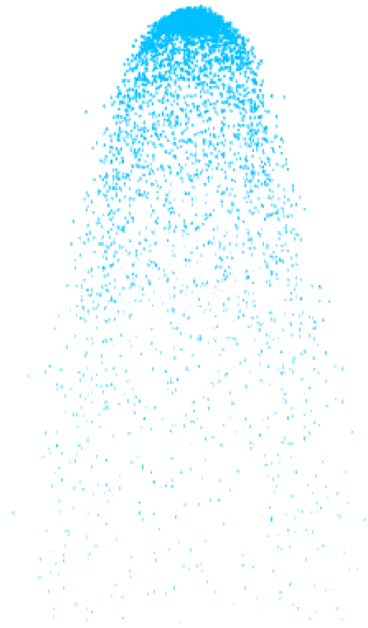


Abbildung 19: Screenshot der Smoothed Particle Hydrodynamics

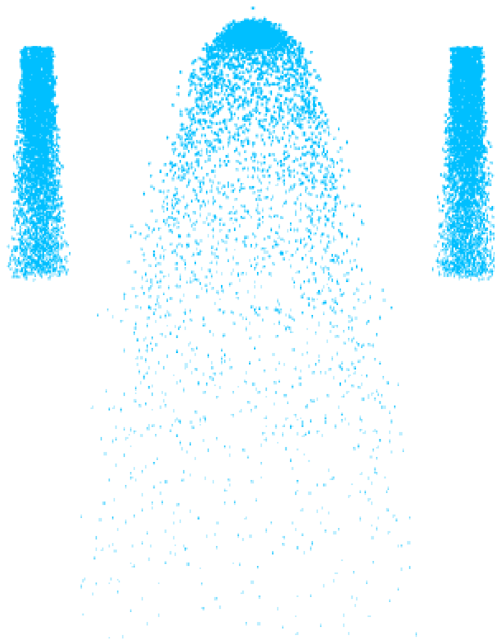


Abbildung 20: Screenshot der Smoothed Particle Hydrodynamics mit zwei einfachen Partikelsystemen

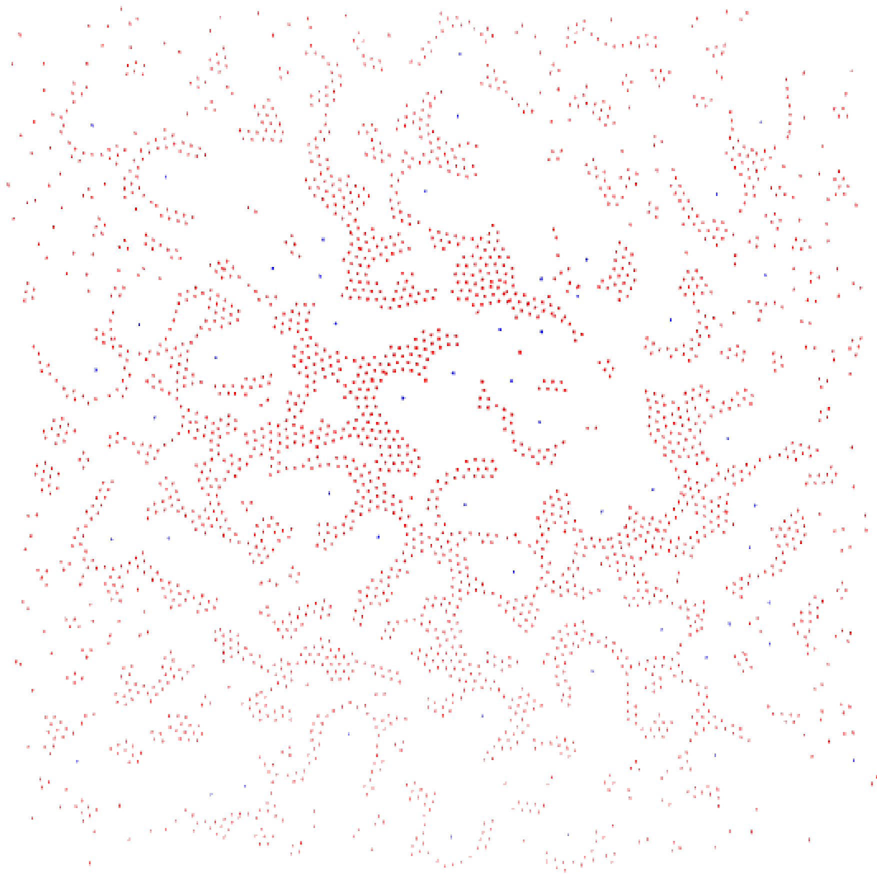


Abbildung 21: Screenshot eines Partikelschwarms mit blau dargestellten Jägern und rot dargestellten Gejagten