



WHZ Westsächsische
Hochschule Zwickau
Hochschule für Mobilität

GPU-BESCHLEUNIGUNG EINER ASTROPHYSIKALISCHEN N-KÖRPER-SIMULATION

Fakultät für Physikalische Technik und Informatik
Fachgruppe Informatik

Bachelorarbeit

vorgelegt von

Tom Borowsky

im März 2026

Danksagungen

Besonderer Dank gilt dem betreuenden Mentor an der Hochschule, Professor Frank Grimm.

Zusammenfassung

In der vorliegenden Arbeit wird eine GPU-basierte Implementierung einer N-Körper-Simulation entwickelt und hinsichtlich ihrer Leistungsfähigkeit untersucht. Die Umsetzung erfolgt unter Verwendung von HIP für die parallele Berechnung sowie Vulkan zur Visualisierung.

Zur Bewertung der Effizienz werden die Laufzeit und Skalierbarkeit der GPU-Implementierung mit einer direkten CPU-Implementierung sowie einer CPU-basierten Barnes-Hut-Approximation verglichen. Die Ergebnisse zeigen eine deutliche Beschleunigung durch die GPU-basierte Parallelisierung gegenüber der direkten Berechnung auf der CPU. Gleichzeitig weist die Barnes-Hut-Methode aufgrund ihrer geringeren Zeitkomplexität ein überlegenes Skalierungsverhalten für große Körperzahlen auf.

Die Untersuchung verdeutlicht, dass GPU-basierte Umsetzungen insbesondere für Echtzeit-Anwendungen mit moderaten Körperzahlen geeignet sind, während hierarchische Approximationsverfahren ihre Stärken vor allem bei sehr großen Systemen ausspielen.

Inhaltsverzeichnis

Danksagungen	II
Zusammenfassung	III
1. Einleitung	1
2. Physikalische und algorithmische Grundlagen	3
2.1. Physikalisches Modell	3
2.2. Direkte Kräfteberechnung	4
2.3. Softening	5
2.4. Numerische Integration	5
2.5. Barnes-Hut-Verfahren	6
3. Wahl der Technologien	8
3.1. Programmiersprache	8
3.2. GPGPU-Programmierung	9
3.3. Grafik-API	10
3.4. Shader-Sprache	11
4. Implementierung	12
4.1. Gemeinsame Datenstrukturen	12
4.2. Direkte Methode (CPU)	14
4.3. Barnes-Hut-Verfahren (CPU)	18
4.3.1. Baumkonstruktion	19
4.3.2. Baumtraversierung	22
4.4. Direkte Methode (GPU)	24
4.4.1. Programmierungsgrundlagen	24
4.4.2. Simulation	29
4.5. Renderer	32
5. Ergebnisse	36
5.1. Messumgebung und Methodik	36
5.2. Messergebnisse	38
6. Potenzielle Verbesserungen	39
6.1. Ausnutzung des dritten Newtonschen Gesetzes bei der direkten Simulation	39
6.2. Verbesserungen des Barnes-Hut-Verfahrens	40

6.3. GPU-Implementierung des Barnes-Hut-Verfahrens	41
6.4. Behandlung von Kollisionen	42
A. Erklärung zur Verwendung maschineller Hilfsmittel	43

Abbildungsverzeichnis

2.1. Schemazeichnung eines Octrees [2]	6
5.1. Standbild aus einer Simulation	37
5.2. Diagramm der Messergebnisse	38

1. Einleitung

Das N-Körper-Problem der klassischen Mechanik beschreibt die Bewegung mehrerer Körper, die durch eine Kraft, beispielsweise Gravitationskraft oder elektrostatische Kraft, miteinander wechselwirken. Für Systeme mit mehr als zwei Körpern existiert im Allgemeinen keine geschlossene analytische Lösung, sodass ihre zeitliche Entwicklung numerisch berechnet werden muss. Die Ergebnisse solcher Simulationen sind unter anderem in der Astrophysik und Kosmologie gefragt, um die Entstehung von Galaxien und anderen Strukturen des Universums zu untersuchen.

Bei einer direkten N-Körper-Simulation muss in jedem Simulationsschritt die Kraft paarweise zwischen allen Körpern berechnet werden. Daraus folgt für N Körper eine Zeitkomplexität von $\mathcal{O}(N^2)$, sodass die benötigte Rechenzeit mit der Anzahl der Körper schnell ansteigt. Um die Simulation großer Systeme von Körpern dennoch in angemessener Zeit zu ermöglichen, wurden verschiedene Ansätze zur Optimierung entwickelt. Dazu gehören Approximationsverfahren wie die Barnes-Hut-Methode, welche die Zeitkomplexität auf Kosten der Genauigkeit verringern. Ein anderer Optimierungsansatz besteht in der parallelen Ausführung, da die paarweisen Kräfteberechnungen innerhalb eines Simulationsschritts voneinander unabhängig sind.

Grafikprozessoren (Graphics Processing Units, GPUs) verfügen über eine stark parallele Hardwarearchitektur mit einer großen Zahl einfacher Recheneinheiten, was sie nicht nur für Grafikanwendungen geeignet macht, sondern auch zur Beschleunigung allgemeiner Rechenprobleme. Diese Verwendungsweise ist unter dem Namen General Purpose Computing on Graphics Processing Units (GPGPU) bekannt, und hat zur Entwicklung dedizierter Programmierschnittstellen wie Nvidias CUDA geführt, welche heute auch für Zwecke wie Machine Learning eingesetzt werden. Probleme, bei denen die gleichen Rechenoperationen auf viele Daten angewandt werden müssen, eignen sich besonders gut für GPU-Architekturen.

Ziel dieser Arbeit ist die Entwicklung und Untersuchung einer GPU-basierten Implementierung einer N-Körper-Simulation. Die Simulation dient dabei als Beispielproblem zur Untersuchung der effizienten Nutzung moderner GPU-Architekturen für numerische Berechnungen mit hohem Parallelisierungspotenzial. Es soll demonstriert werden, dass die Hardwarearchitektur von Grafikprozessoren geeignet ist, die Simulation performant und mit geringer Latenz zu berechnen, was auch die Visualisierung ihrer Ergebnisse in Echtzeit ermöglichen soll. Die GPU-Implementierung der Simulation wird dazu mit einer direkten CPU-Implementierung sowie mit einer CPU-Umsetzung des Barnes-Hut-Verfahrens hinsichtlich ihrer Laufzeit verglichen.

Für die GPGPU-Programmierung wird dabei AMDs Heterogeneous-Compute Interface for Portability (HIP) verwendet, um die Simulationslogik zu implementieren. Zur Echtzeit-Visualisierung wird die Grafik-API Vulkan eingesetzt.

Die Arbeit wird zunächst die physikalischen und algorithmischen Grundlagen der N-Körper-Simulation erläutern. Anschließend werden verfügbare Technologien zur GPU-Programmierung erkundet und die getroffene Auswahl ergründet. Danach werden die Implementierungen beschrieben und ihre Laufzeit und Skalierbarkeit verglichen. Abschließend werden weitere potenzielle Verbesserungen der Umsetzungen diskutiert.

2. Physikalische und algorithmische Grundlagen

In diesem Kapitel werden die physikalischen Grundlagen sowie die verwendeten algorithmischen Verfahren der N-Körper-Simulation erläutert. Zunächst werden die zugrunde liegenden Bewegungsgleichungen vorgestellt, anschließend deren numerische Umsetzung sowie Optimierungsverfahren zur effizienten Berechnung großer Systeme.

2.1. Physikalisches Modell

In der Simulation werden die betrachteten Objekte als punktförmige Körper modelliert, die durch ihre Masse, Position, Geschwindigkeit und Beschleunigung beschrieben werden. Die Bewegung der Körper ergibt sich aus den auf sie wirkenden Kräften.

Grundlage hierfür ist das zweite Newtonsche Gesetz, das den Zusammenhang zwischen Kraft und Beschleunigung beschreibt:

$$F = m \cdot a$$

$$a = \frac{F}{m}$$

Dabei bezeichnet F die auf einen Körper wirkende Kraft, m seine Masse und a die resultierende Beschleunigung.

Die Wechselwirkung zwischen zwei Körpern wird durch das Newtonsche Gravitationsgesetz beschrieben:

$$F = G \frac{m_1 m_2}{r^2}$$

Hierbei ist F die Gravitationskraft zwischen zwei Körpern, m_1 und m_2 sind deren Massen, r ist der Abstand zwischen ihren Massenzentren und G die Gravitationskonstante mit dem Wert $6,6743 \cdot 10^{-11} \frac{\text{m}^3}{\text{kg} \cdot \text{s}^2}$.

Für die numerische Simulation ist die vektorielle Darstellung der Kraft erforderlich. Für zwei Körper mit Positionsvektoren r_1 und r_2 ergibt sich der Verschiebungsvektor

$$r_{21} = r_2 - r_1.$$

Die Kraft, die von Körper 1 auf Körper 2 ausgeübt wird, kann dann wie folgt geschrieben werden:

$$F_{21} = -G \frac{m_1 m_2}{|r_{21}|^3} r_{21}.$$

Dabei ist $|r_{21}|$ der Betrag des Verschiebungsvektors. Nach dem dritten Newtonschen Gesetz gilt zudem

$$F_{21} = -F_{12}.$$

2.2. Direkte Kräfteberechnung

In einer direkten N-Körper-Simulation wird für jeden Körper die resultierende Kraft durch Aufsummieren der Beiträge aller anderen Körper berechnet. Dies führt auf eine doppelte Schleife über alle Körperpaare hinaus.

Der grundlegende Algorithmus lässt sich in Python-ähnlichem Pseudocode wie folgt darstellen:

```
1 für alle Körper i:
2     für alle Körper j:
3         d_vek = j.pos - i.pos    # Verschiebungsvektor
4         d = betrag(d_vek)
5         kraft = (G * i.masse * j.masse * d_vek) / (d ** 3)
6         i.beschl += kraft / i.masse
```

Da die Masse von Körper i sowohl im Zähler der Kraft als auch im Nenner der Beschleunigungsberechnung auftritt, kann sie gekürzt werden. Dadurch ergibt sich eine vereinfachte und schnellere Berechnung der Beschleunigung:

```
1 für alle Körper i:
2     für alle Körper j:
3         d_vek = j.pos - i.pos
4         d = betrag(d_vek)
5         i.beschl += (G * j.masse * d_vek) / (d ** 3)
```

Die Berechnung aller paarweisen Wechselwirkungen führt zu einer Zeitkomplexität von $\mathcal{O}(N^2)$. Optimierungen auf Grundlage des dritten Newtonschen Gesetzes sind prinzipiell möglich und werden in Abschnitt 6.1 diskutiert.

2.3. Softening

Bei sehr kleinen Abständen zwischen zwei Körpern wird der Nenner der Kraftformel sehr klein, sodass die resultierende Kraft gegen unendlich wächst. Dies führt zu numerischen Instabilitäten.

Um dieses Problem zu vermeiden, wird ein sogenannter Softening-Parameter eingeführt, der den Abstand zwischen zwei Körpern künstlich vergrößert. Die Distanz wird dabei wie folgt berechnet:

```
1 d = sqrt(betrag(d_vek) ** 2 + softening ** 2)
```

Der Softening-Parameter sollte so gewählt werden, dass er numerische Instabilitäten verhindert, ohne das physikalische Verhalten der Simulation wesentlich zu verfälschen.

Ein zusätzlicher Vorteil dieses Ansatzes besteht darin, dass die Berechnung der Selbstinteraktion eines Körpers nicht explizit durch eine Bedingung ausgeschlossen werden muss. Da der Verschiebungsvektor im Zähler in diesem Fall der Nullvektor ist, ergibt sich trotz des endlichen Nenners eine Kraft von null.

2.4. Numerische Integration

Die Beschleunigungen der Körper werden in diskreten Zeitschritten berechnet. Um daraus die zeitliche Entwicklung von Geschwindigkeit und Position zu bestimmen, muss ein numerisches Integrationsverfahren verwendet werden.

Einfache Integratoren wie das explizite Euler-Verfahren sind simpel in der Implementierung und schnell zu berechnen, weisen jedoch eine geringe Genauigkeit und schlechte Energieerhaltung auf. Höherwertige Verfahren wie Runge-Kutta-Methoden bieten eine bessere Genauigkeit, sind jedoch mit einem höheren Rechenaufwand verbunden.

In dieser Arbeit wurde das Velocity-Verlet-Verfahren verwendet, welches in mathematisch äquivalenten Varianten auch als Leapfrog- oder Störmer-Methode bekannt ist. Dieses Verfahren besitzt eine Fehlerordnung von zwei und ist symplektisch, eine Eigenschaft, die in mechanischen Systemen eine gute Energieerhaltung über lange Simulationszeit gewährleistet. [1]

Das Verfahren kann algorithmisch wie folgt beschrieben werden, wobei schritt die zeitliche Länge des Integrationsschritts bezeichnet:

```
1 für alle Körper i:  
2   i.pos += i.geschw * schritt + i.beschl * 0.5 * schritt ** 2  
3  
4 für alle Körper i:  
5   neueBeschl = beschleunigung(i)  
6   i.geschw += (i.beschl + neueBeschl) * 0.5 * schritt  
7   i.beschl = neueBeschl
```

Der Funktionsaufruf `beschleunigung(i)` stellt dabei die innere Schleife der zuvor beschriebenen Kräfteberechnung dar. Die Aufteilung auf zwei Schleifen ist nötig, da `beschleunigung(i)` die Positionen aller Körper liest, welche zuvor vom ersten Schritt der Integration aktualisiert werden müssen.

2.5. Barnes-Hut-Verfahren

Zur Reduktion der Zeitkomplexität der direkten N-Körper-Simulation kann das Barnes-Hut-Verfahren eingesetzt werden. Dieses basiert auf einer hierarchischen Raumaufteilung in Form eines Baums, der den Simulationsraum rekursiv in Teilregionen unterteilt. Im dreidimensionalen Raum wird hierfür ein Octree verwendet, bei dem jeder innere Knoten über acht Kindknoten für die Oktanten seiner Region verfügt. Abbildung 2.1 veranschaulicht diese Datenstruktur.

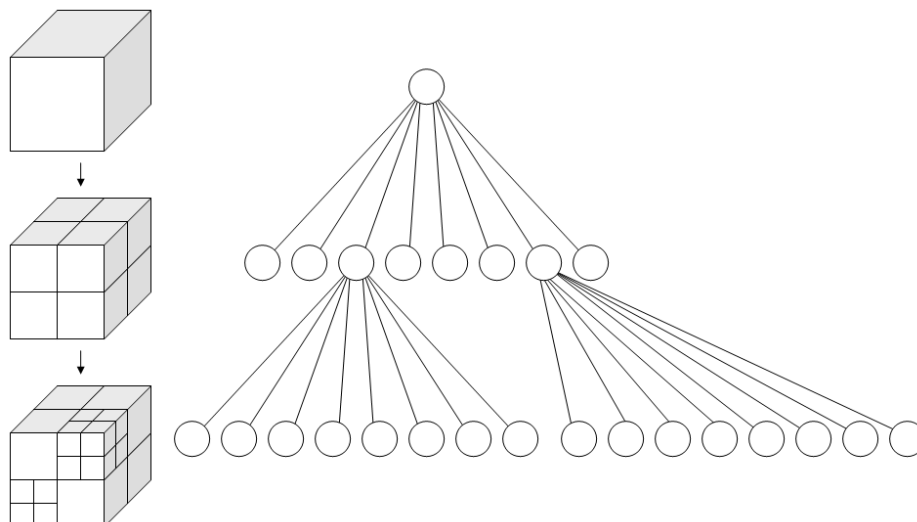


Abbildung 2.1.: Schemazeichnung eines Octrees [2]

Die Blattknoten des Baums enthalten entweder keinen oder genau einen Körper. Innere Knoten repräsentieren eine Gruppe von Körpern und speichern deren Gesamtmasse sowie den Schwerpunkt.

Bei der Berechnung der Kräfte auf einen Körper wird der Baum von der Wurzel aus traversiert. Für jeden betrachteten Knoten wird entschieden, ob dessen Inhalt als Ganzes approximiert werden kann oder ob weiter in die untergeordneten Knoten abgestiegen werden muss. Hierfür wird das Verhältnis zwischen der Größe der Region und ihrer Entfernung zum betrachteten Körper mit einem Schwellenwert θ verglichen.

Ist dieses Verhältnis kleiner als θ , wird die gesamte Region als ein einzelner Körper mit der entsprechenden Gesamtmasse und dem Schwerpunkt behandelt. Andernfalls wird die Traversierung in den Kindknoten fortgesetzt.

Durch dieses Verfahren kann die Zeitkomplexität im Mittel auf $\mathcal{O}(N \log N)$ reduziert werden. Der Parameter θ steuert dabei den Kompromiss zwischen Genauigkeit und Rechenaufwand.

Alternative Approximationsmethoden existieren ebenfalls. Dazu gehört das Particle-Mesh-Verfahren, das auf einer Diskretisierung der Massendichte auf einem Gitter basiert und die zugrunde liegende Poisson-Gleichung effizient mittels schneller Fourier-Transformation löst [3]. Da die räumliche Auflösung hier aber durch die Gitterweite limitiert ist, eignet sich das Verfahren nicht für Kräfte über kurze Distanzen und wird in der Praxis mit dem direkten Verfahren oder anderen Approximationsmethoden kombiniert [3]. Die Fast Multipole Method (FMM) ist ein weiteres Baum-basiertes Verfahren, welches auf Multipolentwicklungen basiert und die Zeitkomplexität im Gegensatz zum Barnes-Hut-Verfahren auf $\mathcal{O}(N)$ reduzieren kann [4]. Da die FMM jedoch eine komplexe Abfolge von Transformationsoperationen zwischen Baumknoten erfordert, ist sie mit einem erheblich höheren Implementierungsaufwand verbunden und wurde daher für den Umfang dieser Arbeit als ungeeignet bewertet [4].

3. Wahl der Technologien

Für die Implementierung der Simulation mussten einige technologische Entscheidungen getroffen werden. Diese betreffen insbesondere die Programmiersprache der Anwendung, die Programmierschnittstelle für das GPGPU-Computing sowie die Grafik-API und die Shader-Sprache für die Visualisierung der Simulation.

Viele der betrachteten Technologien werden von der Khronos Group spezifiziert, einem offenen Konsortium, welches Standards im Bereich von Grafik und Parallel Computing entwickelt [5]. Der Khronos Group gehören zahlreiche Unternehmen an, darunter AMD, Apple, Intel und Nvidia [6].

3.1. Programmiersprache

Die Implementierung der Simulation erfolgte in der Programmiersprache C++. Viele Grafik- und Compute-APIs sind in Form von C-kompatiblen Schnittstellen definiert, wodurch sie sich besonders einfach aus C oder C++ heraus verwenden lassen.

Gegenüber der Programmiersprache C bietet C++ zusätzliche Sprachmechanismen und „zero-cost abstractions“, die eine strukturiertere und sicherere Implementierung ermöglichen, ohne dabei zusätzlichen Laufzeitaufwand zu verursachen. Hierzu gehören unter anderem Klassen für objektorientierte Programmierung, RAII (Resource Acquisition Is Initialization) zum Vorbeugen von Fehlern bei der Ressourcenverwaltung, Namespaces zur Programmstrukturierung sowie Templates und constexpr-Ausdrücke zur Realisierung generischer und zur Kompilierzeit ausgewerteter Programmelemente, wodurch auch oft die Nutzung von Präprozessor-Makros vermieden werden kann.

3.2. GPGPU-Programmierung

Für die Implementierung der Simulationslogik auf der GPU existieren mehrere etablierte Programmierschnittstellen.

Einer der ältesten offenen Standards ist OpenCL, der ursprünglich von Apple entworfen wurde und seit 2008 von der Khronos Group spezifiziert wird. OpenCL ist eine plattformübergreifende Lösung für "Heterogeneous Computing", was die verteilte Programmausführung über verschiedene Prozessortypen hinweg beschreibt, beispielsweise CPUs und GPUs. [7]

CUDA ist eine proprietäre GPGPU-Plattform von Nvidia und wird ausschließlich auf Nvidia-Grafikprozessoren unterstützt [8]. Sie stellt eine umfassende Programmierschnittstelle sowie Werkzeuge zur Entwicklung und Ausführung von GPU-Programmen bereit und wird in verschiedenen Anwendungsbereichen wie dem wissenschaftlichen Rechnen und Machine Learning eingesetzt. Der Einsatz von CUDA führt jedoch zu einer starken Bindung an Nvidia-Hardware.

Als Alternative wurde HIP (Heterogeneous-Compute Interface for Portability) von AMD entwickelt. HIP stellt eine Programmierschnittstelle bereit, die stark an CUDA angelehnt ist und sich daher mit geringem Aufwand aus CUDA-Code ableiten lässt, was teilweise durch das HIPIFY-Werkzeug automatisiert werden kann [9]. Gleichzeitig ermöglicht HIP eine portablere Nutzung der Programme, da HIP-Code sowohl auf AMD- als auch auf Nvidia-Grafikprozessoren ausgeführt werden kann. Hierfür kann der Compiler-Driver `hipcc` wahlweise für CUDA- oder ROCm-Plattformen kompilieren. ROCm ist AMDs Software-Stack, der Laufzeitumgebungen und weitere Werkzeuge für Programmierschnittstellen wie HIP und OpenCL bereitstellt [10].

Darüber hinaus versucht das Projekt chipStar, HIP- und CUDA-Code auf weitere Hardwareplattformen zu portieren. Dabei wird der Code auf Intel Level Zero, einer hardwarenahen Programmierschnittstelle von Intel, oder OpenCL übersetzt, um unter anderem die Ausführung auf Intel-Grafikprozessoren zu ermöglichen. Dieses Projekt wird vom Argonne National Laboratory für den Exascale-Supercomputer Aurora unterstützt, der auf Intel-Hardware basiert. [11]

Eine weitere offene Programmierschnittstelle ist SYCL, welche ebenfalls von der Khronos Group standardisiert wird. SYCL verfolgt das Ziel, Heterogeneous Computing in standardkonformes C++ zu integrieren [12]. Zwei der ausgereiftesten Implementierungen sind Intels oneAPI Data Parallel C++ (DPC++) und das Community-getriebene AdaptiveCpp [13]. Beide implementieren die Unterstützung von AMD-GPUs über HIP bzw. ROCm [14, 15].

Alternativ könnten auch Compute Shader innerhalb der Vulkan-API für GPGPU-Berechnungen verwendet werden. Dieser Ansatz ermöglicht es, Rechenoperationen direkt über die Grafik-API auszuführen. Die Implementierung ist jedoch vergleichsweise aufwendig, da die Compute-Logik in Form separater Shaderprogramme geschrieben werden muss und zusätzlicher Boilerplate-Code für Descriptor Sets, Pipeline-Objekte und Speicherverwaltung erforderlich ist.

Für die vorliegende Arbeit wurde schließlich HIP gewählt. Der primäre Grund hierfür ist die verwendete AMD-Grafikhardware, für die HIP eine besonders direkte und effiziente Programmierschnittstelle darstellt. Darüber hinaus orientiert sich HIP stark an CUDA, sodass viele Konzepte und Lernressourcen aus dem CUDA-Ökosystem auch für HIP relevant sind.

3.3. Grafik-API

Für die Visualisierung der Simulation wurde eine geeignete Grafik-API benötigt. Hierfür kommen insbesondere OpenGL, Vulkan sowie Microsofts Direct3D in Betracht.

OpenGL ist ein etablierter offener Grafikstandard der Khronos Group und wird als plattform-unabhängige Grafik-API beschrieben. Die Implementierung erfolgt über herstellerspezifische Treiber, die OpenGL-Befehle in GPU-Befehle übersetzen. Dadurch kann die API zwar vieles abstrahieren, ihre Laufzeiteigenschaften hängen jedoch stärker von der jeweiligen Implementierung ab. [16, 17]

Vulkan ist ein moderner Grafikstandard, der ebenfalls von der Khronos Group entwickelt wird. Im Vergleich zu OpenGL verwaltet bei Vulkan die Anwendung mehr Aspekte der Ressourcen- und Speicherverwaltung selbst. Dieser explizitere Ansatz erhöht die Komplexität, kann dafür allerdings für geringeren Overhead und bessere vorhersagbare Performance sorgen. [18, 19]

Direct3D ist eine proprietäre Grafik-API von Microsoft und wird hauptsächlich unter Windows eingesetzt. Mit Direct3D 12 verfolgt auch diese API einen expliziteren Ansatz ähnlich zu Vulkan [20]. Die Plattformabhängigkeit von Direct3D stellt jedoch einen Nachteil für portable Anwendungen dar.

Für diese Arbeit wurde Vulkan gewählt, da es eine portable und leistungsfähige Grafik-API darstellt und gleichzeitig eine gute Kontrolle über die Ressourcennutzung ermöglicht. Dies ist insbesondere relevant, da die verfügbare Rechenleistung der GPU primär für die Simulationsberechnung genutzt werden soll.

3.4. Shader-Sprache

Neben der Grafik-API musste auch eine geeignete Shader-Sprache für die Implementierung der Grafikkipeline gewählt werden.

Eine verbreitete Shader-Sprache ist GLSL (OpenGL Shading Language), die von der Khronos Group entwickelt wurde und ursprünglich für OpenGL konzipiert ist. GLSL wird auch im Vulkan-Ökosystem verwendet, indem die Shaderprogramme zu SPIR-V kompiliert werden, einer Intermediate Representation (IR), die ebenfalls von der Khronos Group spezifiziert und sowohl für Grafik-APIs wie Vulkan als auch für Compute-APIs wie OpenCL verwendet wird. [21]

Eine alternative Option ist HLSL (High Level Shading Language), die von Microsoft für Direct3D entwickelt wurde [22]. Auch HLSL kann inzwischen in SPIR-V übersetzt werden und wird daher ebenfalls im Vulkan-Kontext eingesetzt [23].

Eine neuere Shader-Sprache ist Slang, die ursprünglich von Nvidia entwickelt und später als offenes Projekt an die Khronos Group übergeben wurde [24]. Slang basiert syntaktisch weitgehend auf HLSL, ist aber keine genaue Obermenge, und bietet zudem zusätzliche Funktionen wie Typinferenz für Variablen und Module [25].

Für die Implementierung der Shaderprogramme wurde Slang gewählt, da sie eine portable Option mit modernen Sprachfeatures darstellt.

4. Implementierung

Dieses Kapitel beschreibt die praktische Umsetzung der zuvor beschriebenen Konzepte. Nach den gemeinsamen Datenstrukturen wird zunächst die Implementierung der direkten N-Körper-Simulation auf der CPU behandelt, welche als Referenzpunkt für die folgenden Implementierungen dient. Als nächstes erfolgt die Beschreibung der Barnes-Hut-Approximation auf der CPU, gefolgt von der Übertragung der direkten Methode auf die GPU. Abschließend wird auf die Implementierung des Renderers eingegangen, der den Zustand der Simulation visualisiert und damit die Analyse und Interpretation ihrer Ergebnisse unterstützt.

4.1. Gemeinsame Datenstrukturen

Um die initialen Bedingungen der Simulation zu spezifizieren, wird zunächst eine Datenstruktur eingeführt, die einen einzelnen Körper repräsentiert.

```
1 struct Body {  
2     float3 position;  
3     float3 velocity;  
4     float mass;  
5     float radius{std::cbrt(mass)};  
6 };
```

Für vektorielle Größen wie Position, Geschwindigkeit und später auch Beschleunigung wird hier stets das Struct `float3` genutzt, welches von HIP zur Verfügung gestellt wird und einige Operatoren für elementweise arithmetische Operationen implementiert. Der Radius der Körper ist hier nur für den Renderer relevant, welcher Körper wie Kugeln darstellt, und wird standardmäßig als dritte Wurzel der Masse berechnet, da das Volumen einer Kugel und dazu proportional ihre Masse mit der dritten Potenz des Radius wächst. An dieser Stelle könnte noch ein Parameter für die Dichte des Körpers ergänzt werden; da der Radius aber nur zur Visualisierung genutzt und später vom Renderer ohnehin zur besseren Sichtbarkeit skaliert wird, wurde hier darauf verzichtet.

Es könnte nun mit einer Liste von Objekten dieses Structs gearbeitet werden, die fortlaufend im Speicher liegen. Diese Anordnung wird auch als Array of Structures (AoS) bezeichnet, und lässt sich intuitiv verwenden, um die Körper zu spezifizieren, mit denen die Simulation ausgeführt werden soll.

Eine Alternative ist die Structure of Arrays (SoA), bei der für eine Liste von Objekten die verschiedenen Attribute der Objekte in separaten Arrays gehalten werden.

```
1 struct Bodies {
2     std::vector<float3> positions;
3     std::vector<float3> velocities;
4     std::vector<float> masses;
5     std::vector<float> radii;
6
7     Bodies(const std::vector<Body> &bodies) {
8         const auto numBodies{bodies.size()};
9         positions.reserve(numBodies);
10        velocities.reserve(numBodies);
11        masses.reserve(numBodies);
12        radii.reserve(numBodies);
13        for (const auto &body : bodies) {
14            positions.emplace_back(body.position);
15            velocities.emplace_back(body.velocity);
16            masses.emplace_back(body.mass);
17            radii.emplace_back(body.radius);
18        }
19    }
20 };
```

Bei dieser Anordnung liegen jeweils die Positionen, Geschwindigkeiten, und weiteren Attribute aller Körper an aufeinanderfolgenden Speicheradressen in separaten Arrays. Dies kann insbesondere bei der Hardwarearchitektur von GPUs, auf die in Abschnitt 4.4 genauer eingegangen wird, zu effizienteren Speicherzugriffsmustern führen. Auch bei den CPU-basierten Implementierungen wurde mit SoA eine bessere oder vergleichbare Performance im Vergleich zu AoS festgestellt. Weiterhin müssen für den Renderer zur Laufzeit der Simulation nur die Positionen aktualisiert werden. Er arbeitet daher mit einem Speicherpuffer, der ausschließlich die Positionen enthält, womit die SoA-Anordnung der Simulation besser kompatibel ist.

4.2. Direkte Methode (CPU)

Die Simulationslogik wurde in einer Klasse umgesetzt, welche über Datenfelder für die Anzahl, Positionen, Geschwindigkeiten, Beschleunigungen und Massen der Körper verfügt. Der Einstiegspunkt für einen Simulationsschritt ist die Methode `update`.

```

1 void update(const float step) {
2     const auto halfStep{step * 0.5F};
3     const auto halfSquaredStep{step * step * 0.5F};
4
5     for (unsigned idx{0}; idx < numBodies; ++idx) {
6         positions[idx] += (velocities[idx] * step) +
7                           (accelerations[idx] * halfSquaredStep);
8     }
9     for (unsigned idx{0}; idx < numBodies; ++idx) {
10        const float3 newAccel{computeAcceleration(idx)};
11
12        velocities[idx] += (accelerations[idx] + newAccel) * halfStep;
13        accelerations[idx] = newAccel;
14    }
15 }

```

Diese Funktion setzt die in Abschnitt 2.4 beschriebene numerische Integration um, wobei `halfStep` und `halfSquaredStep` vorab berechnet werden, statt in jeder Iteration erneut.

Da jede Iteration der Schleifen auf verschiedene Indizes der Arrays zugreift und nicht von Ergebnissen der vorherigen Iterationen abhängig ist, lassen sich hier mehrere CPU-Kerne ausnutzen, indem die Iterationen nebenläufig ausgeführt werden. Dies lässt sich syntaktisch kompakt mit der OpenMP-API für parallele Programmierung umsetzen.

```

1 #pragma omp parallel for
2     for (unsigned idx = 0; idx < numBodies; ++idx) {
3         positions[idx] += (velocities[idx] * step) +
4                           (accelerations[idx] * halfSquaredStep);
5     }
6 #pragma omp parallel for
7     for (unsigned idx = 0; idx < numBodies; ++idx) {
8         const float3 newAccel{computeAcceleration(idx)};
9
10        velocities[idx] += (accelerations[idx] + newAccel) * halfStep;
11        accelerations[idx] = newAccel;
12    }

```

OpenMP wird in Form von `#pragma`-Direktiven genutzt, die von Compilern implementiert werden. `parallel` startet dabei ein Team von Threads, woraufhin durch `for` die Iterationen der folgenden Schleife auf diese Threads verteilt werden. [26]

Um jedoch nicht für beide parallelen Schleifen separate Thread-Teams zu starten, wird ein äußerer paralleler Block eingeführt.

```

1  #pragma omp parallel
2  {
3  #pragma omp for
4      for (unsigned idx = 0; idx < numBodies; ++idx) {
5          positions[idx] += (velocities[idx] * step) +
6                          (accelerations[idx] * halfSquaredStep);
7      }
8  #pragma omp for
9      for (unsigned idx = 0; idx < numBodies; ++idx) {
10         const float3 newAccel{computeAcceleration(idx)};
11
12         velocities[idx] += (accelerations[idx] + newAccel) * halfStep;
13         accelerations[idx] = newAccel;
14     }
15 }
```

Die Kräfteberechnung gemäß Abschnitten 2.2 und 2.3 kann wie folgt implementiert werden.

```

1  float3 computeAcceleration(const unsigned bodyIdx) const {
2      float3 accel{0.F, 0.F, 0.F};
3      const float3 pos{positions[bodyIdx]};
4
5      for (unsigned otherIdx{0}; otherIdx < numBodies; ++otherIdx) {
6          const float3 dist{positions[otherIdx] - pos};
7          const float distMag{
8              std::sqrt(dot(dist, dist) + SOFTENING_SQUARED)};
9          accel +=
10              G * masses[otherIdx] * dist / (distMag * distMag * distMag);
11      }
12      return accel;
13 }
```

Ähnlich wie `halfStep` zuvor wird die Position `pos` des betrachteten Körpers einmalig vor der Schleife aus dem Speicher geladen. Die Funktion `dot` berechnet hier das Skalarprodukt von `dist` mit sich selbst, also die Summe der Quadrate der Vektorkomponenten, deren Quadratwurzel den Betrag des Vektors ergibt.

G ist die Gravitationskonstante, hier mit dem Wert $4\pi^2$, was dem Einheitensystem von astronomischen Einheiten für Länge, Sonnenmassen für Masse und Jahren für Zeit entspricht. Mit den SI-Standardeinheiten Meter, Kilogramm und Sekunden würden bei den verwendeten initialen Bedingungen, die einzelne Planetensysteme modellieren, große Werte zustande kommen, welche bei Arithmetik mit single-precision Gleitkommazahlen zu Überläufen führen. Beispielsweise liegt die Masse der Sonne bei etwa $2 \cdot 10^{30}$ kg, was in der Kräfteberechnung mit Distanzen in der Größenordnung von 10^{11} m multipliziert werden würde, während der maximale Wert von single-precision Gleitkommazahlen (binary32-Typ unter IEEE-754) nur bei etwa $3,4 \cdot 10^{38}$ liegt [27].

Da der Betrag des Verschiebungsvektors `distMag` ausschließlich im Nenner der Kraftformel vorkommt, konnte an dieser Stelle eine Optimierung erzielt werden, indem direkt ihr Reziproke berechnet und die verhältnismäßig aufwendige Division durch eine Multiplikation ersetzt wird.

```

1  for (unsigned otherIdx = 0; otherIdx < numBodies; ++otherIdx) {
2      const float3 dist{positions[otherIdx] - pos};
3      const float invDistMag{rsqrt(dot(dist, dist) + SOFTENING_SQUARED)};
4      accel += G * masses[otherIdx] * dist * invDistMag * invDistMag *
5              invDistMag;
6  }

```

```

1  inline float rsqrt(const float number) {
2      const float approx{_mm_cvtss_f32(_mm_rsqrt_ss(_mm_set_ss(number)))};
3      return approx * (1.5F - 0.5F * number * approx * approx);
4  }

```

Für das Reziproke der Quadratwurzel existiert in C++ keine Standardbibliotheksfunktion, daher wurden Compiler-Intrinsics verwendet, welche direkteren Zugriff auf Hardware-Befehle erlauben. Diese Intrinsics werden auf Anweisungen der Befehlssatzerweiterungen SSE bzw. AVX der x86-64-Architektur abgebildet, welche eine Reihe von Single Instruction, Multiple Data (SIMD)-Befehlen definieren. SIMD beschreibt die gleichzeitige Ausführung desselben Befehls auf mehreren Datenpunkten, und wird in Abschnitt 4.4 zur GPU-Implementierung genauer behandelt. Da in dieser Funktion aber jeweils nur mit einer Zahl gerechnet wird, werden die skalaren Varianten der Befehle (ss-Suffix) genutzt. Diese agieren auf der Gleitkommazahl in den unteren Bits eines Vektorregisters, sodass die Anweisungen `_mm_set_ss` und `_mm_cvtss_f32` genutzt werden, um die Zahl in ein solches Register zu befördern und wieder zu extrahieren. `_mm_rsqrt_ss` ist eine Approximation, die mindestens eine Präzision von etwa 12 Bits liefert, weshalb sie hier durch eine Iteration des Newton-Raphson-Verfahrens ergänzt wird, um die Genauigkeit zu erhöhen. [28]

Auch OpenMP bietet Direktiven für SIMD, durch die die Kräfteberechnung noch effizienter umgesetzt werden konnte.

```

1 float3 computeAcceleration(const unsigned bodyIdx) const {
2     float accelX{0.F}, accelY{0.F}, accelZ{0.F};
3     const float3 pos{positions[bodyIdx]};
4
5     #pragma omp simd reduction(+ : accelX, accelY, accelZ)
6     for (unsigned otherIdx = 0; otherIdx < numBodies; ++otherIdx) {
7         const float3 dist{positions[otherIdx] - pos};
8         const float invDistMag{rsqrt(dot(dist, dist) + SOFTENING_SQUARED)};
9         const float coeff{G * masses[otherIdx] * invDistMag * invDistMag *
10                          invDistMag};
11         accelX += coeff * dist.x;
12         accelY += coeff * dist.y;
13         accelZ += coeff * dist.z;
14     }
15     return {accelX, accelY, accelZ};
16 }

```

Durch diese Implementierung kann die Summierung der Beschleunigungswerte, eine Reduktion, über die Schleifeniterationen hinweg mit SIMD-Befehlen vektorisiert werden. Das bedeutet, dass die `accelX`-Werte mehrerer Iterationen in einem Vektorregister gleichzeitig berechnet werden können, ebenso die `y`- und `z`-Komponenten. Da hierfür der Beschleunigungsvektor explizit in seine Komponenten zerlegt werden musste, wird der skalare Teil der Berechnung als Zwischenergebnis in `coeff` gehalten.

4.3. Barnes-Hut-Verfahren (CPU)

Für die Umsetzung des Barnes-Hut-Verfahrens, welches in Abschnitt 2.5 beschrieben wurde, musste der Octree ergänzt werden. Die Knoten des Baums werden durch ihre Position und Größe, sowie ihre gesamte Masse und ihr Massenzentrum definiert. Weiterhin enthalten sie Verweise auf ihre untergeordneten Oktanten und enthaltenen Körper, falls vorhanden.

```

1 struct OctreeNode {
2     float3 center;
3     float width{};
4     float3 centerOfMass{0.F, 0.F, 0.F};
5     float mass{0.F};
6
7     std::array<int, 8> children{-1, -1, -1, -1, -1, -1, -1, -1};
8     int bodyIdx{-1};
9
10    [[nodiscard]] bool isInternal() const { return children[0] >= 0; }
11    [[nodiscard]] bool isLeaf() const { return children[0] < 0; }
12    [[nodiscard]] bool isEmpty() const { return bodyIdx < 0; }
13 };
14
15 class Octree {
16 public:
17     // ...
18
19 private:
20     std::span<const float3> positions;
21     std::span<const float> masses;
22     float theta;
23     std::vector<OctreeNode> pool;
24     int root{};
25
26     int allocNode() {
27         pool.emplace_back();
28         return static_cast<int>(pool.size()) - 1;
29     }
30     // ...
31 };

```

Um zu vermeiden, dass die Knoten über den Heap-Speicher verstreut liegen, wurde statt `new OctreeNode`-Allokationen ein `std::vector` als Pool-Allocator verwendet, damit die Knoten im Speicher näher beieinander liegen und der CPU-Cache effizienter genutzt werden kann. Aus diesem Grund werden die Verweise auf Wurzel- und Kindknoten als Indizes in den Pool statt als Zeiger repräsentiert.

Dass ein Knoten nicht weiter in Oktanten zerlegt wurde, kann somit durch negative Indizes signalisiert werden. Weiterhin wird ein Knoten als leerer Knoten mit einem negativen `bodyIdx` initialisiert, woraufhin ein Körper in ihm abgelegt werden kann. Erst, wenn einem Knoten, der bereits einen Körper enthält, ein weiterer Körper zugewiesen werden soll, wird er in Oktanten zerlegt und so zu einem inneren Knoten. In diesem Fall bleibt der `bodyIdx` als Index des ersten abgelegten Körpers erhalten. Diese Zuweisung kann von der Programmlogik folglich ignoriert werden und ermöglicht zugleich, in `isEmpty()` nicht auch auf `isLeaf()` prüfen zu müssen.

4.3.1. Baumkonstruktion

Zur Laufzeit der Simulation wird der Octree in jedem Simulationsschritt neu aus den aktuellen Positionen der Körper gebildet. Während der grundlegende Ansatz des Barnes-Hut-Verfahrens darin besteht, den Raum rekursiv in Teilregionen zu unterteilen, wurde die Baumkonstruktion nicht mit einer solchen Rekursion begonnen, da in diesem Fall bei jeder Unterteilung die Liste der N Körper erneut nach denjenigen Körpern durchsucht werden müsste, die in jedem neuen Oktanten liegen. Der Suchraum könnte dabei fortschreitend eingegrenzt werden, indem jeder innere Knoten die Liste seiner enthaltenen Körper speichert, sodass jeder neue Oktant nur die Liste seines Elternknotens durchsuchen müsste, was jedoch wiederum den Speicherbedarf deutlich erhöhen würde. Stattdessen wurde die Baumkonstruktion umgesetzt, indem iterativ alle Körper in den Baum einsortiert werden, was dann zu rekursiven Raumunterteilungen führen kann.

```
1 void rebuild() {
2     pool.clear();
3     pool.reserve(positions.size() * 8);
4     root = allocNode();
5     float3 minCoord{positions[0]};
6     float3 maxCoord{positions[0]};
7     for (const auto pos : positions) {
8         minCoord = min(minCoord, pos);
9         maxCoord = max(maxCoord, pos);
10    }
11    pool[root].center = (minCoord + maxCoord) * 0.5F;
12    pool[root].width = std::max({
13        maxCoord.x - minCoord.x,
14        maxCoord.y - minCoord.y,
15        maxCoord.z - minCoord.z,
16    });
17    for (int bodyIdx{0}; bodyIdx < positions.size(); ++bodyIdx) {
18        insert(root, bodyIdx);
19    }
20 }
```

Beim Pool-Allocator wird zunächst ausreichend Speicher reserviert, um aufwendige Re-Allokationen und Copy-/Move-Operationen an den Knoten zu vermeiden. Diese Re-Allokationen, wenn auch unwahrscheinlich, sind dennoch möglich, weshalb in der folgenden Baumkonstruktion nur mit Pool-Indizes gearbeitet werden sollte, nicht mit Zeigern oder Referenzen, da diese nach Re-Allokation des Pools auf ungültigen Speicher zeigen würden. Anschließend werden die minimalen und maximalen Koordinaten entlang jeder Achse unter den Körpern ermittelt, um Zentrum und Größe der Region des Wurzelknotens so festzulegen, dass alle Körper enthalten sind. Danach beginnt das Einfügen aller Körper, ausgehend vom Wurzelknoten.

```

1 void insert(const int nodeId, const int bodyIdx) {
2     const auto pos{positions[bodyIdx]};
3     const auto mass{masses[bodyIdx]};
4     if (pool[nodeId].isEmpty()) {
5         pool[nodeId].bodyIdx = bodyIdx;
6         pool[nodeId].centerOfMass = pos;
7         pool[nodeId].mass = mass;
8     } else {
9         pool[nodeId].centerOfMass =
10             (pool[nodeId].centerOfMass * pool[nodeId].mass +
11              pos * mass) /
12             (pool[nodeId].mass + mass);
13         pool[nodeId].mass += mass;
14
15         if (pool[nodeId].isLeaf()) {
16             for (int idx{0}; idx < 8; ++idx) {
17                 const auto childIdx{allocNode()};
18                 pool[nodeId].children[idx] = childIdx;
19                 pool[childIdx].center = getOctantCenter(
20                     pool[nodeId].center, pool[nodeId].width, idx);
21                 pool[childIdx].width = pool[nodeId].width * 0.5F;
22             }
23             const auto octant{getOctant(positions[pool[nodeId].bodyIdx],
24                                         pool[nodeId].center)};
25             insert(pool[nodeId].children[octant], pool[nodeId].bodyIdx);
26         }
27         const auto octant{getOctant(pos, pool[nodeId].center)};
28         insert(pool[nodeId].children[octant], bodyIdx);
29     }
30 }

```

Wenn der Knoten, in den der aktuelle Körper eingefügt werden soll, leer (und damit ein Blattknoten) ist, kann der Körper einfach abgelegt werden.

Ansonsten muss in jedem Fall die gesamte Masse des Knotens erhöht und das neue Massenzentrum als gewichtetes arithmetisches Mittel des bisherigen Zentrums und der

Position des neuen Körpers berechnet werden. Daraufhin muss der Körper in einen Oktanten des betrachteten Knoten eingefügt werden, aber zuvor muss dieser Knoten in Oktanten unterteilt werden, sofern er noch kein innerer Knoten ist. Falls diese Unterteilung nötig ist, muss auch der Körper, der bisher als einziges im Knoten lag, in einen der neuen Oktanten eingefügt werden.

Um die Oktanten des Raums den Indizes im `children`-Array der Knoten zuzuordnen, wurden die unteren drei Bits der Indizes aus den Vorzeichen des Richtungsvektors entlang jeder Achse abgeleitet.

```
1 inline unsigned char getOctant(const float3 pos, const float3 center) {
2     const auto dir{pos - center};
3
4     const unsigned xSign{dir.x < 0 ? 1U : 0U};
5     const unsigned ySign{dir.y < 0 ? 1U : 0U};
6     const unsigned zSign{dir.z < 0 ? 1U : 0U};
7     return (zSign << 2) | (ySign << 1) | xSign;
8 }
9
10 inline float3 getOctantCenter(const float3 center, const float width,
11                               const unsigned char octant) {
12     return center + width * float3{
13         (((octant & 1) != 0) ? -0.25F : 0.25F),
14         (((octant & 2) != 0) ? -0.25F : 0.25F),
15         (((octant & 4) != 0) ? -0.25F : 0.25F),
16     };
17 }
```

4.3.2. Baumtraversierung

Nach der Konstruktion wird der Baum für jeden Körper einmal traversiert, um die auf ihn wirkenden Kräfte zu approximieren.

```

1  [[nodiscard]] float3 computeAcceleration(const int bodyIdx) const {
2      return traverse(root, bodyIdx);
3  }
4
5  [[nodiscard]] float3 traverse(const int nodeId,
6                               const int bodyIdx) const {
7      const auto &node{pool[nodeId]};
8      const auto pos{positions[bodyIdx]};
9
10     const auto dist{node.centerOfMass - pos};
11     const auto invDistMag{rsqrt(dot(dist, dist) + SOFTENING_SQUARED)};
12     if (node.isLeaf() or node.width * invDistMag < theta) {
13         return (G * node.mass * dist) *
14             (invDistMag * invDistMag * invDistMag);
15     }
16
17     float3 accel{0.F, 0.F, 0.F};
18     for (const auto childIdx : node.children) {
19         accel += traverse(childIdx, bodyIdx);
20     }
21     return accel;
22 }

```

Da beim Traversieren der Baum nicht modifiziert wird, können hier Referenzen genutzt werden. Beim Erreichen eines Kindknotens kann die Rekursion beendet werden, ebenso wenn die Größe eines inneren Knotens im Verhältnis zur Entfernung vom betrachteten Körper kleiner ist als der Parameter θ . In beiden Fällen wird die Gravitationskraft, die vom erreichten Knoten auf den betrachteten Körper wirkt, berechnet, was für innere Knoten einer Approximation der untergeordneten Körper entspricht. Andernfalls wird die Traversierung rekursiv in allen Kindknoten fortgesetzt und die resultierenden Beschleunigungen aufsummiert.

Die update-Methode der Simulation kann dann wie folgt geschrieben werden.

```

1 void update(const float step) {
2     const auto halfStep{step * 0.5F};
3     const auto halfSquaredStep{step * step * 0.5F};
4     #pragma omp parallel
5     {
6         #pragma omp for schedule(dynamic)
7         for (unsigned idx = 0; idx < numBodies; ++idx) {
8             positions[idx] += (velocities[idx] * step) +
9                             (accelerations[idx] * halfSquaredStep);
10        }
11
12        #pragma omp single
13        octree.rebuild();
14
15        #pragma omp for schedule(dynamic)
16        for (unsigned idx = 0; idx < numBodies; ++idx) {
17            const float3 newAccel{octree.computeAcceleration(idx)};
18
19            velocities[idx] += (accelerations[idx] + newAccel) * halfStep;
20            accelerations[idx] = newAccel;
21        }
22    }
23 }

```

Da die Baumkonstruktion sequentiell abläuft, wird die parallele Ausführung hierfür mit `single` unterbrochen. Zudem ermöglicht das dynamische Scheduling, dass Threads, die ihre Iteration der Schleife erledigt haben, sofort eine weitere zugewiesen bekommen, bis alle Iterationen abgearbeitet sind. Dies ist insbesondere beim Barnes-Hut-Verfahren von Vorteil, da die Traversierung für verschiedene Körper unterschiedlich tief rekursiv absteigt und die Rechenarbeit pro Iteration damit weniger gleichmäßig ist als bei der direkten Simulation.

4.4. Direkte Methode (GPU)

4.4.1. Programmierungsgrundlagen

Damit die GPU mit den Daten der Simulation arbeiten kann, müssen diese zunächst vom Hauptspeicher in einen GPU-zugänglichen Speicherpuffer befördert werden. Für Allokation solcher Puffer und Transferoperationen bietet HIP eine API ähnlich zu Funktionen der C-Standardbibliothek.

```

1  std::vector<float> masses{...};
2
3  float *pMasses;
4  std::size_t sizeBytes{masses.size() * sizeof(float)};
5  hipCheck(hipMalloc(&pMasses, sizeBytes));
6  hipCheck(hipMemcpy(pMasses, masses.data(), sizeBytes,
7                    hipMemcpyHostToDevice));

```

In diesem Beispiel werden die Daten aus einem `std::vector` in einen Puffer im Grafikspeicher kopiert. Im Gegensatz zu `malloc` in C liefert `hipMalloc` den Zeiger auf den allozierten Speicher nicht als Rückgabewert, sondern legt ihn an der Adresse ab, die durch den ersten Parameter spezifiziert wird. Der Rückgabewert wird hingegen, wie bei den meisten Funktionen der HIP-API, für das Signalisieren eventueller Fehlerzustände genutzt. `hipCheck` ist eine eigene Hilfsfunktion, welche diese Rückgabewerte auswertet und gegebenenfalls Fehler durch Exceptions oder Konsolenausgaben anzeigt. `hipMemcpy` ist wiederum ähnlich zur C-Funktion `memcpy`, nimmt aber einen weiteren Parameter entgegen, der die Art des Speichertransfers angibt. Als Host werden in diesem Programmiermodell die CPU und der dazugehörige Hauptspeicher bezeichnet, als Device hingegen die GPU und ihr Grafikspeicher [29]. In diesem Fall wird also aus dem Hauptspeicher, mit dem Zeiger `masses.data()`, in den Grafikspeicher, mit dem GPU-lesbaren Zeiger `pMasses`, kopiert. [30]

Ähnlich wie mit `free` in C muss der Speicher auch wieder mit `hipFree` freigegeben werden. Um diese Speicherverwaltung zu abstrahieren und Programmierfehler vorzubeugen, wurden für Ressourcen wie Speicherpuffer C++-Klassen angelegt, die das RAII-Muster umsetzen, indem die Ressourcen im Konstruktor alloziert und im Destruktor freigegeben werden. Weiterhin werden Move-Semantics implementiert und Operationen mit den Ressourcen, wie hier der Speichertransfer, in Methoden gekapselt.

```

1  template <typename T>
2  class DeviceBuffer {
3  public:
4      DeviceBuffer(std::size_t size) : sizeBytes_{size * sizeof(T)} {
5          hipCheck(hipMalloc(&data_, sizeBytes_));
6      }
7      ~DeviceBuffer() {
8          if (data_ != nullptr) {
9              hipCheck(hipFree(data_));
10         }
11     }
12     DeviceBuffer(const DeviceBuffer<T> &) = delete;
13     DeviceBuffer<T> &operator=(const DeviceBuffer<T> &) = delete;
14     DeviceBuffer(DeviceBuffer<T> &&other) noexcept
15         : data_{other.data_}, sizeBytes_{other.sizeBytes_} {
16         other.data_ = nullptr;
17         other.sizeBytes_ = 0;
18     }
19     DeviceBuffer<T> &operator=(DeviceBuffer<T> &&other) noexcept {
20         if (this == &other) {
21             return *this;
22         }
23         if (data_ != nullptr) {
24             hipCheck(hipFree(data_));
25         }
26         data_ = std::exchange(other.data_, nullptr);
27         sizeBytes_ = std::exchange(other.sizeBytes_, 0);
28         return *this;
29     }
30
31     [[nodiscard]] T *data() noexcept { return data_; }
32
33     void copyFrom(std::span<const T> view) {
34         hipCheck(hipMemcpy(data_, view.data(), sizeBytes_,
35                             hipMemcpyHostToDevice));
36     }
37     void copyTo(std::span<T> view) const {
38         hipCheck(hipMemcpy(view.data(), data_, sizeBytes_,
39                             hipMemcpyDeviceToHost));
40     }
41
42 private:
43     T *data_;
44     std::size_t sizeBytes_;
45 };

```

Grundlegende Konzepte zur Ausführung von Code auf der GPU sollen anhand von folgendem Codebeispiel erklärt werden.

```

1  __global__ void kernel() {
2      const auto globalIdx{blockIdx.x * blockDim.x + threadIdx.x};
3      printf("Hello world from thread %d in block %d!\n", globalIdx,
4             blockIdx.x);
5  }
6
7  void runKernel() {
8      Stream stream;
9      Event event;
10
11     kernel<<<dim3{2, 1, 1}, dim3{3, 1, 1}, 0, *stream>>>();
12     hipCheck(hipGetLastError());
13     event.record(*stream);
14     event.synchronize();
15 }

```

Device-Code wird im Allgemeinen asynchron zum Host ausgeführt, wobei Streams asynchrone Warteschlangen für Device-Arbeit darstellen [31]. Stream ist hier eine weitere RAII-Klasse, welche das darunterliegende `hipStream_t`-Objekt der HIP-API über den Dereferenzierungsoperator `*` liefert.

Die Einstiegspunkte des Device-Codes, welche vom Host aufgerufen werden können, werden Kernel genannt und durch den Function-type Qualifier `__global__` markiert. Die anderen Function-type Qualifier sind `__host__`, welches auf dem Host ausführbare Funktionen markiert und implizit für Funktionen ohne Qualifier definiert wird, und `__device__` für Funktionen, die auf dem Device ausgeführt und nur von anderen `__device__`- oder `__global__`-Funktionen aufgerufen werden können. [32]

Kernels können mit der „triple-chevron syntax“ (`<<< >>>`) aufgerufen werden, welche eine der Spracherweiterungen von HIP und CUDA darstellt, die nicht konform mit Standard-C++ sind. Beim Kernel-Aufruf werden Konfigurationsparameter angegeben, welche die Anzahl der Blöcke und Threads im Grid angeben, auf das im Folgenden eingegangen wird, sowie die Menge an dynamischen geteilten Speicher, welcher in diesem Programm nicht genutzt wird, und den Stream, auf dem die Kernel-Ausführung eingereicht wird. Zum Schluss können die gewöhnlichen Argumente der Kernel-Funktion in Klammern übergeben werden, was später beispielsweise für die Zeiger der Speicherpuffer genutzt wird. Mit `hipCheck(hipGetLastError())` können Fehler beim Starten des Kernels gefangen werden.

Um die asynchrone Ausführung des Device-Codes explizit mit dem Host zu synchronisieren, können unter anderem Events eingesetzt werden, für die ebenfalls ein RAII-Wrapper ergänzt wurde. Durch den `record`-Aufruf wird das Event auf dem angegebenen Stream eingereicht.

Durch den `synchronize`-Aufruf kann der Kontrollfluss blockiert werden, bis das Event vorne in der Warteschlange des Streams angekommen ist. Indem zuerst der Kernel-Aufruf eingereicht, das Event dann aufgenommen und synchronisiert wird, kann also der Host auf den Abschluss der Kernel-Ausführung warten. [31]

Für die parallele Ausführung der Kernel kommt ein hierarchisches Thread-Modell zum Einsatz. Threads sind dabei die einzelnen Instanzen der Kernel-Operationen auf dem Device, während das Grid die Gesamtheit aller Threads eines Kernel-Aufrufs beschreibt. Zwischen diesen beiden existiert die Stufe der Blöcke, auch Work-Groups genannt, welche Threads innerhalb des Grids gruppieren. Die Dimensionen des Grids und der Blöcke, also die Anzahl der Blöcke pro Grid und Threads pro Block, können mit drei Komponenten `x/y/z` spezifiziert werden, was bei bestimmten Problemen eine bessere semantische Zuordnung der Threads ermöglicht. Da hier aber nur entlang der `x`-Achse mehrere Blöcke und Threads angeordnet werden, kann die Konfiguration der Dimensionen auch mit skalaren Werten statt `dim`-Structs erfolgen. Im Kernel kann über global verfügbare Konstanten wie `blockIdx`, `threadIdx` und `blockDim` auf die jeweiligen Dimensionen und Indizes zugegriffen werden, um die Position des aktuellen Threads im Grid zu ermitteln, und damit die Zugehörigkeit zu Daten oder Teilaufgaben. Alle Threads eines Blocks werden auf derselben Compute Unit ausgeführt, was zusätzliche Synchronisierung und geteilten Speicher ermöglicht. [29]

Compute Units (CUs) sind in AMDs Hardwarearchitektur Recheneinheiten, die unabhängig voneinander arbeiten können und wiederum aus einer Vielzahl von arithmetisch-logischen Einheiten (ALUs) und Registern sowie weiteren Komponenten bestehen. Die einzelnen ALUs werden auch SIMD-Lanes genannt, da sie in Lockstep arbeiten, das heißt, alle Lanes führen zur gleichen Zeit den gleichen Befehl mit verschiedenen Daten aus, ähnlich den SIMD-Anweisungen der CPU, nur mit breiteren Vektoren. Die Gruppen der Anweisungen, die gemeinsam in Lockstep ausgeführt werden, werden als Warps bezeichnet. Für die Ergebnisse dieser Arbeit wurde eine GPU der RDNA3-Architektur genutzt, auf der Warps aus 32 Threads bestehen, weshalb die Blockgröße als ein Vielfaches von 32 gewählt werden sollte. [33, 34]

Das Programmiermodell in HIP und CUDA wird als Single Instruction, Multiple Threads (SIMT), bezeichnet, was sich von SIMD unterscheidet, da jeder Thread in SIMT seinen eigenen Kontrollfluss inklusive Program Counter und Call Stack besitzt, der unabhängig verzweigen kann. Auf Hardwareebene werden die SIMT-Threads dann jedoch SIMD-Lanes zugeordnet, um hochgradig parallele Ausführung zu ermöglichen. Wenn Threads innerhalb eines Warps unterschiedlich verzweigen, müssen die Befehle daher dennoch für alle Lanes ausgeführt werden, wobei die ungenutzten Ergebnisse für verzweigte Threads ausgeblendet werden. Dieses Konzept wird als Thread Divergence bezeichnet und sollte möglichst vermieden werden, da es zu weniger effizienter Ausnutzung der Recheneinheiten führt. [29, 35]

Während auf einer CU nur von 32 Threads (unter RDNA3) gleichzeitig Befehle ausgeführt werden, kann sie noch wesentlich mehr Threads nebenläufig ausführen. Das ist möglich, weil die Daten von vielen Threads gleichzeitig in den Registern der CU gehalten werden, das heißt viele Threads bleiben zugleich „resident“ auf einer CU, bis zu 1024 auf RDNA3. Der Anteil der tatsächlich residenten Warps auf einer CU zu der maximalen Anzahl, die sie halten kann, wird als Occupancy bezeichnet [36]. Dadurch kann Context Switching zwischen Threads sehr schnell durchgeführt werden, sodass beispielsweise auf einen anderen Thread gewechselt werden kann, während der zuvor ausgeführte Thread auf einen Speicherzugriff wartet. Mehr Threads auf einer CU resident zu halten ist allerdings nur möglich, solange der Kontext weiterer Threads in ihre Register passt. Eine weitere Metrik, welche die Performance beeinflussen kann, ist daher die Register Pressure, also wie viele Register ein Thread für seine Ausführung benötigt. Höhere Register Pressure kann die Occupancy verringern. [33, 34]

Ein weiteres Konzept zur GPU-Optimierung ist Memory Coalescing. Darunter wird verstanden, dass aufeinanderfolgende Threads gleichzeitig auf fortlaufende Speicheradressen zugreifen, was die Zusammenfassung von Speichertransaktionen erlaubt und so die gesamte Latenz von Speicherzugriffen deutlich reduzieren kann [29]. Die SoA-Anordnung, die in Abschnitt 4.1 beschrieben wurde, kann dabei helfen, zu solchen „coalesced memory accesses“ zu führen. Der SoA-Ansatz kann noch weiter vertieft werden, indem die x-, y- und z-Komponenten der Positions-, Geschwindigkeits- und Beschleunigungsvektoren auch in separaten Arrays gehalten werden. Damit konnte jedoch in diesem Programm keine signifikante Verbesserung der Performance beobachtet werden, während die Lesbarkeit des Codes deutlich beeinträchtigt wird.

4.4.2. Simulation

Ähnlich der zwei parallelen Schleifen für die Integration in den bisherigen Implementierungen wurden in der GPU-Simulation zwei Kernel geschrieben.

```

1  __global__ void updateKernel1(const float step,
2                                const float halfSquaredStep,
3                                float3 *__restrict__ positions,
4                                const float3 *__restrict__ velocities,
5                                const float3 *__restrict__ accelerations,
6                                const uint32_t numBodies) {
7      const auto idx{(blockIdx.x * blockDim.x) + threadIdx.x};
8      if (idx >= numBodies) {
9          return;
10     }
11     positions[idx] += (velocities[idx] * step) +
12                      (accelerations[idx] * halfSquaredStep);
13 }
14 __global__ void updateKernel2(const float halfStep,
15                                const float3 *__restrict__ positions,
16                                float3 *__restrict__ velocities,
17                                float3 *__restrict__ accelerations,
18                                const float *__restrict__ masses,
19                                const uint32_t numBodies) {
20     const auto idx{blockIdx.x * blockDim.x + threadIdx.x};
21     const auto newAccel{
22         computeAcceleration(idx, positions, masses, numBodies)};
23     if (idx >= numBodies) {
24         return;
25     }
26     velocities[idx] += (accelerations[idx] + newAccel) * halfStep;
27     accelerations[idx] = newAccel;
28 }

```

Hier wird der Memory-Space-Qualifier `__restrict__` von HIP genutzt, welcher auch als `restrict` in C existiert, jedoch nicht in C++. Durch `__restrict__` garantiert der Programmierer dem Compiler, dass kein anderer Zeiger in der Funktion auf die gleiche Speicheradresse zeigt wie der markierte Zeiger, das heißt, es besteht kein „Aliasing“ zwischen den Zeigern. Dies ermöglicht dem Compiler zusätzliche Optimierungen. [32]

Da, wie zuvor beschrieben, Threads auf der GPU stets in Gruppen (Warps) ausgeführt werden und die Blockgröße in der Regel als Vielfaches der Warp-Größe gewählt wird, kann das Grid mehr Threads enthalten als Körper zur Simulation gehören. Aus diesem Grund wird der Index des jeweiligen Threads mit der Anzahl der Körper verglichen und die Ausführung gegebenenfalls durch ein `return` frühzeitig beendet. Im zweiten Kernel findet

das aber erst nach der Kräfteberechnung statt, was sich später aus der Implementierung der `computeAcceleration`-Funktion ergeben wird.

Die `update`-Funktion ist nun für das Starten der Kernel verantwortlich.

```

1 void update(const float step) {
2     const auto blocks{1 + ((numBodies - 1) / BLOCK_SIZE)};
3     const auto halfStep{step * 0.5F};
4     const auto halfSquaredStep{step * step * 0.5F};
5     updateKernel1<<<blocks, BLOCK_SIZE, 0, *stream>>>(
6         step, halfSquaredStep, positions.data(), velocities.data(),
7         accelerations.data(), numBodies);
8     hipCheck(hipGetLastError());
9     updateKernel2<<<blocks, BLOCK_SIZE, 0, *stream>>>(
10        halfStep, positions.data(), velocities.data(),
11        accelerations.data(), masses.data(), numBodies);
12    hipCheck(hipGetLastError());
13    stepDone.record(*stream);
14    stepDone.synchronize();
15 }
```

Hier ist `stepDone` das Event, welches zur Synchronisierung genutzt wird, während `positions`, `velocities` etc. Instanzen der `DeviceBuffer`-Klasse sind. Auf die Wahl der Blockgröße wird im Folgenden eingegangen, und die Anzahl der Blöcke ergibt sich dann aus einer aufrundenden Division von `numBodies` durch `BLOCK_SIZE`.

Die Kräfteberechnung könnte zunächst relativ simpel implementiert werden.

```

1 __device__ float3 computeAcceleration(
2     const uint32_t idx, const float3 *__restrict__ positions,
3     const float *__restrict__ masses, const uint32_t numBodies) {
4     float3 accel{0.F, 0.F, 0.F};
5     if (idx >= numBodies) {
6         return accel;
7     }
8     const auto idxPos{positions[idx]};
9     for (uint32_t idx2{0}; idx2 < numBodies; ++idx2) {
10        const float3 dist{positions[idx2] - idxPos};
11        const float invDistMag{
12            rsqrtf(dot(dist, dist) + SOFTENING_SQUARED)};
13        accel += (G * masses[idx2] * dist) *
14            (invDistMag * invDistMag * invDistMag);
15    }
16    return accel;
17 }
```

Die reziproke Quadratwurzel wird von HIP durch die Funktion `rsqrtf` vorgegeben.

In der Schleife müssen die Positionen und Massen der Körper aus dem globalen Grafikspeicher geladen werden, was verhältnismäßig lange dauert. In dieser Variante des Codes laden verschiedene Threads unabhängig voneinander die Daten der gleichen Körper aus dem Speicher. Es ist daher eine Optimierung möglich, indem geteilter Speicher (Shared Memory) genutzt wird. Shared Memory wird unter AMD-Architekturen auf den Local Data Share (LDS) abgebildet, einen schnell erreichbaren Cache in der GPU, der vom Programmierer explizit kontrolliert werden kann und unter allen Threads eines Blocks geteilt wird [36, 37].

```

1  __device__ float3 computeAcceleration(
2      const uint32_t idx, const float3 *__restrict__ positions,
3      const float *__restrict__ masses, const uint32_t numBodies) {
4      __shared__ float4 shBodies[BLOCK_SIZE];
5      float3 accel{0.F, 0.F, 0.F};
6      const auto idxPos{idx < numBodies ? positions[idx]
7                          : float3{0.F, 0.F, 0.F}};
8      const uint32_t numTiles{(numBodies + BLOCK_SIZE - 1) / BLOCK_SIZE};
9      for (uint32_t tile{0}; tile < numTiles; ++tile) {
10         const uint32_t loadIdx{tile * BLOCK_SIZE + threadIdx.x};
11         if (loadIdx < numBodies) {
12             const auto pos{positions[loadIdx]};
13             shBodies[threadIdx.x] = {pos.x, pos.y, pos.z, masses[loadIdx]};
14         } else {
15             shBodies[threadIdx.x] = {0.F, 0.F, 0.F, 0.F};
16         }
17         __syncthreads();
18         for (uint32_t idx2{0}; idx2 < BLOCK_SIZE; ++idx2) {
19             const uint32_t globalIdx2{(tile * BLOCK_SIZE) + idx2};
20             if (globalIdx2 >= numBodies) {
21                 continue;
22             }
23             const float3 dist{
24                 shBodies[idx2].x - idxPos.x,
25                 shBodies[idx2].y - idxPos.y,
26                 shBodies[idx2].z - idxPos.z,
27             };
28             const float invDistMag{
29                 rsqrtf(dot(dist, dist) + SOFTENING_SQUARED)};
30             accel += (G * shBodies[idx2].w * dist) *
31                     (invDistMag * invDistMag * invDistMag);
32         }
33         __syncthreads();
34     }
35     return accel;
36 }

```

Die zuvor erwähnte Menge an geteiltem Speicher, die bei der Kernel-Startkonfiguration angegeben werden kann, betrifft dynamisch allozierten Shared Memory. Hier wird stattdessen statische Allokation genutzt, da die benötigte Größe durch die `BLOCK_SIZE` zur Kompilierzeit feststeht. [38]

Die Idee hinter dieser Implementierung ist es, die Körper, über die iteriert wird, in Abschnitten (Tiles) aus dem globalen Grafikspeicher in den LDS zu laden. Daraufhin können alle Threads des Blocks mit allen Körpern im Tile die Kräfte berechnen, bevor der nächste Tile von Körpern geladen wird. Dies erfordert allerdings auch zusätzliche Synchronisierung zwischen den Threads des Blocks, da jeder Thread erst anfangen darf, Kräfte zu berechnen, wenn alle Körper des Tiles in den LDS geladen wurde. Ebenso darf der nächste Tile erst in den LDS geladen werden, wenn alle Threads mit der Kräfteberechnung auf dem aktuellen Tile fertig sind. Dafür werden die `__syncthreads()`-Aufrufe genutzt, welche den aktuellen Thread blockieren, bis alle Threads des Blocks diese Stelle erreicht haben und ihre Speicheroperationen für alle Threads sichtbar sind [32]. Diese Aufrufe sind auch der Grund, weshalb ein frühzeitiges `return` für Threads, deren Index außerhalb der Reichweite der Körper liegt, nicht verwendet werden kann, da die Threads die Synchronisierungspunkte erreichen sollen.

Zuletzt wurden im Shared Memory `float4`-Werte genutzt, die sowohl Positionen als auch Massen in gepackter Form enthalten. Diese Anordnung vermeidet Bank Conflicts, bei denen LDS-Zugriffe so versetzt sind, dass mehrere Threads gleichzeitig auf dieselben Speicherbänke des LDS zugreifen, was zu einer ineffizienten Serialisierung der Zugriffe führt. [36]

4.5. Renderer

Der Renderer zur Visualisierung der Simulation wurde durch eine simple, dafür performante Grafikpipeline in Vulkan umgesetzt. Vulkan benötigt dafür viel explizite Konfiguration des Geräts, der Grafikpipeline, Speicherpuffer und Synchronisierungsobjekte, sowie Descriptor Sets, welche genutzt werden um Ressourcen wie Speicherpuffer an die Shader-Stages zu binden [39]. Weiterhin ist zur Präsentation der gerenderten Bilder auf dem Bildschirm eine Window-Management-Bibliothek nötig, welche ein Fenster initialisiert und eine Surface liefert, auf die gezeichnet werden kann. Hierfür wurde die Bibliothek SDL3 verwendet [40].

Die dargestellten Körper sollten die Erscheinung von Kugeln haben. Dafür mussten aber keine dreidimensionalen Modelle von Kugeln erzeugt werden, sondern lediglich flache Kreisflächen, die immer der Kamera zugewandt sind. Diese Kreisflächen werden gebildet, indem von einem Quadrat, dessen Seitenlänge dem Durchmesser des Kreises entspricht, diejenigen Teile verworfen werden, die sich um mehr als den Radius vom Mittelpunkt entfernt befinden.

Ein Vertex Buffer und Index Buffer wurden eingesetzt, um die Eckpunkte eines zweidimensionalen Quadrats zu repräsentieren. Ein Paar aus Vertex Shader und Fragment Shader setzt dann das Laden und Transformieren der Koordinaten sowie das Ausschneiden des Kreises um.

```

1  struct VSInput {
2      float2 vertPos;
3      uint instanceId : SV_InstanceID;
4  };
5
6  struct UniformBuffer {
7      float4x4 view;
8      float4x4 proj;
9  };
10 ConstantBuffer<UniformBuffer> ubo;
11 StructuredBuffer<float3> positions;
12 StructuredBuffer<float> radii;
13
14 struct VSOutput {
15     float4 pos : SV_Position;
16     float2 quadPos;
17 };
18
19 [shader("vertex")]
20 VSOutput vertMain(VSInput input) {
21     VSOutput output;
22     output.pos =
23         mul(ubo.proj,
24             float4(input.vertPos * radii[input.instanceId], 0.0, 0.0) +
25             mul(ubo.view, float4(positions[input.instanceId], 1.0)));
26     output.quadPos = input.vertPos;
27     return output;
28 }
29
30 [shader("fragment")]
31 float4 fragMain(VSOutput vertIn) : SV_TARGET {
32     let distSquared = dot(vertIn.quadPos, vertIn.quadPos);
33     if (distSquared > 1) {
34         discard;
35     }
36     if (distSquared > 0.9) {
37         return { 0.0, 0.0, 0.0, 1.0 };
38     }
39     return { 1.0, 1.0, 1.0, 1.0 };
40 }

```

Der Vertex Shader wird einmal pro Vertex, also pro Knotenpunkt der Geometrie im Vertex Buffer aufgerufen und bekommt dabei die Position seines Vertex über `vertPos` übergeben. Darüber hinaus wird Instanced Rendering eingesetzt, um für jeden Körper eine Instanz des Quadrats im Vertex Buffer zu zeichnen. Dabei erhält der Shader über `instanceId` den Index seiner Instanz, mit dem er die entsprechende Position und den Radius des jeweiligen Körpers aus den Speicherpuffern laden kann. Über den Uniform Buffer `ubo` bekommt der Shader zudem zwei Transformationsmatrizen übergeben. `view` transformiert Koordinaten vom World Space, in dem die Positionen der Körper definiert sind, in den View Space, der relativ zur Kamera ausgerichtet ist. Im View Space wird dann mittels der zweidimensionalen Koordinaten aus dem Vertex Buffer und dem Radius das Quadrat aufgespannt, wodurch es immer der Kamera zugewandt ist. Schließlich wird mit der Projektionsmatrix `proj` mit einer perspektivischen Projektion vom View Space in den Clip Space transformiert, in dem der Rasterisierer der GPU weiterarbeitet. Dabei werden dem Rasterisierer auch die zweidimensionalen Koordinaten der Quadrateckpunkte mitgegeben, welche dieser automatisch zwischen den Vertices, über Fragmente hinweg, interpoliert.

Diese interpolierten Koordinaten innerhalb des Quadrats kommen nach der Rasterisierung im Fragment Shader an, wo sie genutzt werden, um die Entfernung des jeweiligen Fragments vom Mittelpunkt des Quadrats zu bestimmen und Fragmente außerhalb der Kreisfläche zu verwerfen. Dabei wird den Körpern auch ein schwarzer Rand verliehen, um überlappende Körper besser erkennen zu können.

Die Projektionsmatrizen werden dabei mithilfe der GLM-Bibliothek erzeugt, welche dafür intuitive Parameter wie Kameraposition, -Ausrichtung und Blickwinkel entgegennimmt.

Wie in Abschnitt 4.1 erwähnt, werden die Radien der Körper vom Renderer zu Beginn logarithmisch skaliert

```

1  std::vector<float> scaleRadii(const std::vector<float> &radii,
2                               const float minRadius,
3                               const float maxRadius) {
4      std::vector<float> result;
5      result.reserve(radii.size());
6
7      const auto logRadii{radii |
8                          std::views::transform([](const float radius) {
9                              return std::log(radius);
10                         }) |
11                          std::ranges::to<std::vector>()};
12      const auto minLogRadius{*std::ranges::min_element(logRadii)};
13      const auto maxLogRadius{*std::ranges::max_element(logRadii)};
14
15      for (const auto logRadius : logRadii) {
16          if (floatEquals(minLogRadius, maxLogRadius)) {
17              result.emplace_back((minRadius + maxRadius) / 2.);
18          } else {

```

```
19     // normalize to [0,1]
20     const auto normalizedRadius{(logRadius - minLogRadius) /
21                                 (maxLogRadius - minLogRadius)};
22     // scale to [minRadius,maxRadius]
23     result.emplace_back(
24         ((maxRadius - minRadius) * normalizedRadius) + minRadius);
25     }
26 }
27
28 return result;
29 }
```

Die logarithmische Skalierung sorgt für bessere Sichtbarkeit aller Körper, wenn ihre Massen, und damit ihre initialen Radien, sich um viele Größenordnungen unterscheiden, beispielsweise zwischen Sternen und Asteroiden.

Abschließend wurde der gemeinsame Zugriff auf die Positionen der Körper durch Simulation und Renderer möglichst effizient gestaltet, indem Vulkans Erweiterungen für externen Speicher genutzt wurden, um den in Vulkan allozierten Positionspuffer zu exportieren und in HIP zu importieren. Somit können beide Komponenten direkt auf dieselbe Speicherregion zugreifen, statt die Daten kopieren zu müssen. Dies ist nur für die GPU-Implementierung der Simulation möglich; bei den CPU-Varianten müssen die Positionen nach jedem Simulationsschritt aus dem Hauptspeicher in den Grafikspeicher kopiert werden.

5. Ergebnisse

Nachdem die verschiedenen Implementierungen der Simulation im vorhergehenden Kapitel beschrieben wurden, werden nun einige Messungen ihrer Laufzeit durchgeführt, um Performance und Skalierungsverhalten für verschiedene Körperzahlen N zu ermitteln und zu vergleichen.

5.1. Messumgebung und Methodik

Die nachfolgenden Messungen wurden auf einem System mit einer AMD Ryzen 7 7700X-CPU, einer AMD Radeon RX 7900 XT-GPU sowie 32 GiB DDR5-Arbeitsspeicher durchgeführt. Als Betriebssystem kam die Linux-Distribution openSUSE Tumbleweed mit Kernel-Version 6.19.7 zum Einsatz. Der C++-Code wurde mit GCC 15 und der HIP-Code ROCm-Clang 22 unter HIP-Version 7.2.0 kompiliert, beide mit den Optionen `-O3 -march=native`, welche Optimierungen aktivieren und Gebrauch von den unterstützten Befehlssatzerweiterungen der CPU machen können.

Für die Laufzeitmessung wurden mit jeder Implementierung je 100 Simulationsschritte ausgeführt und die mittlere Ausführungszeit pro Schritt bestimmt. Die Messungen wurden für mehrere Körperzahlen wiederholt, um das Skalierungsverhalten der jeweiligen Implementierung zu erfassen.

Bei der Barnes-Hut-Methode wurde der Parameter θ , welcher den Kompromiss zwischen Genauigkeit und Geschwindigkeit regelt, auf einen typischen Wert von 0,8 festgelegt. [41]

Für die initialen Bedingungen der Simulation wurde eine Reihe von Funktionen genutzt, die Bedingungen für elliptische Umlaufbahnen berechnen und die zufällige Erzeugung einer beliebigen Anzahl von Körpern ermöglichen. Die Funktionen sind im beigefügten Code enthalten und der Seed der verwendeten Mersenne-Twister-Engine wurde auf 0 festgelegt, damit die Bedingungen deterministisch sind. Abbildung 5.1 zeigt ein Beispiel dieser initialen Bedingungen mit etwa 26000 Körpern nach einigen Simulationsschritten.

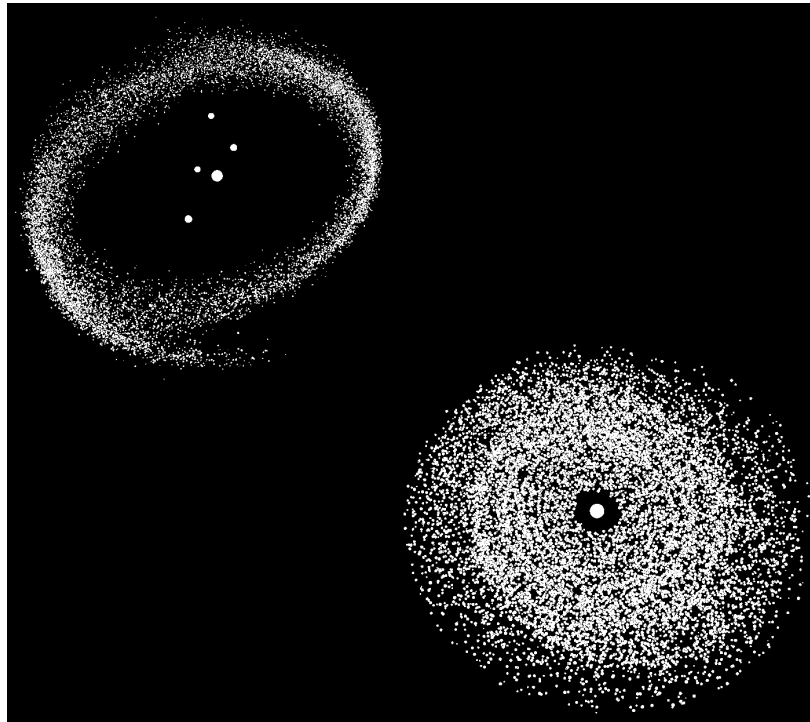


Abbildung 5.1.: Standbild aus einer Simulation

Da die Performance der Barnes-Hut-Methode von den Bedingungen der Körper abhängig ist und, wie in Abschnitt 6.2 erläutert, stark abnehmen kann, wenn sich die Körper zu weit voneinander entfernen, wurden diese initialen Bedingungen so gewählt, dass die Körper über die gemessenen Zeitschritte hinweg nah genug beieinander bleiben. Die Messungen selbst wurden zudem ohne den Renderer ausgeführt, um die Performance der Simulation zu isolieren.

5.2. Messergebnisse

Abbildung 5.2 zeigt die Ergebnisse der Laufzeitmessungen.

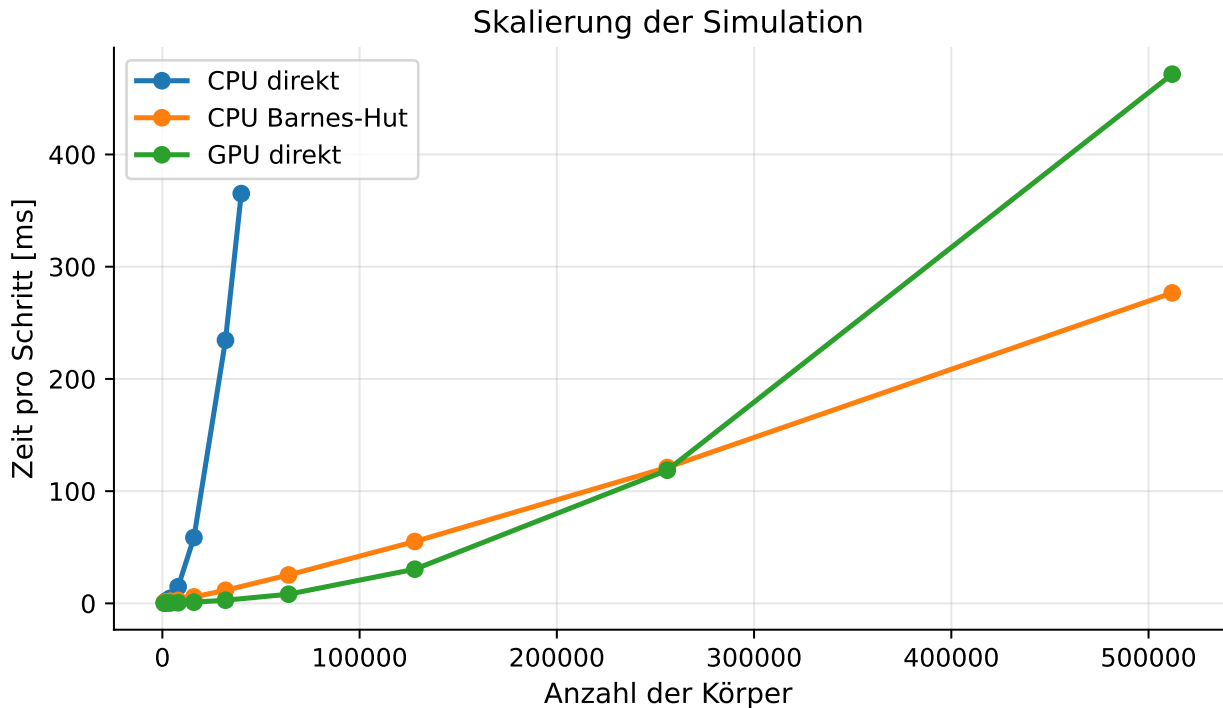


Abbildung 5.2.: Diagramm der Messergebnisse

Die Ergebnisse zeigen deutliche Unterschiede in der Skalierung der betrachteten Verfahren. Gegenüber der Referenzimplementierung der direkten Simulation auf der CPU stellen sowohl die Parallelisierung auf der GPU als auch die Barnes-Hut-Approximation auf der CPU eine erhebliche Beschleunigung dar.

Die GPU-Implementierung profitiert insbesondere bei kleineren bis mittleren Problemgrößen von der hohen Parallelität und erzielt hier die besten Laufzeiten. Aufgrund der quadratischen Laufzeitkomplexität wird der Vorteil gegenüber der Barnes-Hut-Methode jedoch fortlaufend geringer.

Die Barnes-Hut-Implementierung zeigt aufgrund ihrer geringeren Zeitkomplexität von $\mathcal{O}(n \log n)$ ein deutlich besseres Skalierungsverhalten. Ab einer Problemgröße von etwa $2,6 \cdot 10^5$ Körpern ergibt sich ein Schnittpunkt mit der GPU-Implementierung, ab dem die CPU-basierte Approximation geringere Laufzeiten aufweist.

Es ist jedoch zu beachten, dass die absoluten Laufzeiten in diesem Bereich bereits deutlich oberhalb der für Echtzeitanwendungen geeigneten Größenordnung liegen. Der beobachtete Vorteil der Barnes-Hut-Methode ist daher primär im Kontext großer, nicht interaktiver Simulationen relevant.

6. Potenzielle Verbesserungen

Im Rahmen der vorliegenden Arbeit wurden mehrere Implementierungen einer N-Körper-Simulation entwickelt und miteinander verglichen. Während der Entwicklung ergaben sich einige mögliche Erweiterungen und Optimierungen, die in zukünftigen Arbeiten weiter untersucht werden könnten.

6.1. Ausnutzung des dritten Newtonschen Gesetzes bei der direkten Simulation

In der direkten N-Körper-Simulation wird die Kraft zwischen allen Körperpaaren berechnet. In einer naiven Implementierung wird dabei für jedes Körperpaar (i, j) sowohl die Kraft auf Körper i durch Körper j als auch die Kraft auf Körper j durch Körper i separat berechnet. Dadurch werden insgesamt N^2 Wechselwirkungen ausgewertet.

Nach dem dritten Newtonschen Gesetz sind diese beiden Kräfte jedoch betragsgleich und entgegengesetzt gerichtet. Es wäre daher ausreichend, die Kraft nur einmal für jedes Körperpaar zu berechnen und anschließend mit umgekehrtem Vorzeichen auf beide Körper anzuwenden. Die Anzahl der benötigten Berechnungen würde sich dadurch auf $\frac{N(N-1)}{2}$ reduzieren, wenn gleichzeitig die Berechnung der Selbstinteraktion jedes Körpers ausgelassen wird. Dieser Ansatz wird in Tabelle 6.1 veranschaulicht.

Tabelle 6.1.: Kräfte nach dem dritten Newtonschen Gesetz

Körper	1	2	3
1	0	F_{12}	F_{13}
2	$-F_{12}$	0	F_{23}
3	$-F_{13}$	$-F_{23}$	0

Bei einer parallelen Implementierung führt dieser Ansatz allerdings zu zusätzlichen Herausforderungen. Wenn jeder Thread nicht mehr nur für die Kraft, die auf einen der Körper wirkt, zuständig ist, sondern die Kraft- oder Beschleunigungs-Akkumulatoren aller beteiligten Körper aktualisiert, können überlappende Lese- und Schreibzugriffe auftreten. Ohne geeignete Synchronisation entstehen dabei Race Conditions, die zu inkorrekten Ergebnissen führen.

Eine mögliche Lösung dieser Race Conditions besteht in der Verwendung atomarer Operationen auf den Akkumulatorvariablen. Dabei werden die Lese- und Schreibvorgänge beim Inkrementieren der Akkumulatoren atomar ausgeführt, sodass konkurrierende Zugriffe korrekt serialisiert werden. Atomare Operationen verursachen jedoch zusätzlichen Synchronisationsaufwand, was insbesondere bei den vielen Threads auf GPUs zu einem deutlichen Leistungsabfall führen könnte.

Ein alternativer Ansatz besteht darin, für jeden Thread eigene Akkumulatoren zu halten. Während der parallelen Kräfteberechnung kann jeder Thread seine lokalen Akkumulatoren ohne Synchronisation aktualisieren. Nach Abschluss der Berechnung müssen die lokalen Akkumulatoren zu einem globalen Ergebnis aufsummiert werden. Diese Reduktion kann entweder sequentiell oder mittels eines parallelen Reduktionsverfahrens mit logarithmischer Komplexität durchgeführt werden. Der zusätzliche Speicherbedarf für thread-lokale Akkumulatoren sowie der Aufwand für die anschließende Reduktion können hier ebenfalls einen erheblichen Overhead verursachen.

In beiden Fällen entsteht zusätzlicher Synchronisations- oder Speicheraufwand, der den Vorteil der reduzierten Anzahl von Kräfteberechnungen teilweise oder vollständig aufheben kann. Insbesondere auf GPU-Architekturen, bei denen viele Threads gleichzeitig auf gemeinsame Daten zugreifen, muss daher sorgfältig untersucht werden, ob sich dieser Optimierungsansatz tatsächlich positiv auf die Gesamtleistung der Simulation auswirkt.

6.2. Verbesserungen des Barnes-Hut-Verfahrens

Beim Barnes-Hut-Verfahren wird der Simulationsraum rekursiv in Teilbereiche unterteilt, sodass weit entfernte Körpergruppen durch approximierte Massen ersetzt werden können. Die Effizienz dieses Verfahrens hängt jedoch stark von der räumlichen Verteilung der Körper ab.

In der aktuellen Implementierung wird der Wurzelknoten des Baums so gewählt, dass er alle Körper des Systems umfasst. Wenn einzelne Körper das zentrale System verlassen und sich weit entfernen, vergrößert sich dadurch der vom Wurzelknoten abgedeckte Raum immer weiter. Für die im Zentrum verbleibenden Körper führt dies dazu, dass der Baum zunehmend tiefer unterteilt werden muss, um die nötige räumliche Auflösung zu erreichen. Dadurch steigen sowohl die Laufzeit für die Baumkonstruktion und -traversierung als auch der Speicherbedarf pro Simulationsschritt. Zudem wächst mit zunehmender Baumtiefe die Gefahr eines Stapelüberlaufs bei der Rekursion.

Eine mögliche Gegenmaßnahme besteht darin, Körper, die sich weit vom Hauptsystem entfernen, aus der Simulation auszuschließen, wodurch die Größe der simulierten Region begrenzt werden könnte. Eine solche Erweiterung würde jedoch eine Anpassung der Implementierung erfordern, da die Anzahl der simulierten Körper derzeit als konstant angenommen wird.

Die Gefahr eines Stapelüberlaufs besteht auch unabhängig von der Problematik des wachsenden Wurzelknotens, insbesondere bei großen Körpersammlungen auf kleinem Raum. Um Stapelüberläufe auszuschließen, müsste die maximale Baumtiefe begrenzt werden. Wird diese maximale Tiefe während der Baumkonstruktion erreicht, dann könnten „Bucket Nodes“ erzeugt werden, in denen mehrere Körper gemeinsam gespeichert werden. Beim Traversieren des Baums würde in diesem Fall nicht weiter in Unterteilungen abgestiegen, sondern stattdessen auf eine direkte Berechnung der Kräfte zwischen den enthaltenen Körpern zurückgegriffen werden.

Um zusätzlichen Speicherbedarf zu vermeiden, könnten die Körper innerhalb einer solchen Bucket Node beispielsweise über eine einfach verkettete Liste organisiert werden. Dazu könnte jeder Körper um ein zusätzliches Datenfeld erweitert werden, das auf den nächsten Körper in derselben Bucket Node verweist. Auf diese Weise ließe sich die Gruppierung der Körper realisieren, ohne separate Datenstrukturen für jede Bucket Node anlegen zu müssen.

6.3. GPU-Implementierung des Barnes-Hut-Verfahrens

Neben der direkten N-Körper-Simulation stellt auch eine GPU-basierte Umsetzung des Barnes-Hut-Verfahrens eine mögliche Erweiterung dar. Da dieses Verfahren die Anzahl der zu berechnenden Wechselwirkungen reduziert, könnte es insbesondere bei sehr großen Körperzahlen eine weitere Beschleunigung ermöglichen.

Die Übertragung des Barnes-Hut-Algorithmus auf GPU-Architekturen ist jedoch mit zusätzlichen Herausforderungen verbunden. Während sich die direkte Berechnung der Kräfte zwischen Körperpaaren gut für die Parallelisierung eignet, weist das Barnes-Hut-Verfahren eine deutlich komplexere Speicherzugriffsstruktur auf. Die Traversierung des Baums erfordert zahlreiche indirekte Speicherzugriffe, bei denen von einem Knoten zu seinen Kindknoten navigiert wird. Dieses „Pointer Chasing“ führt zu unvorhersehbaren Speicherzugriffsmustern.

Grafikprozessoren erreichen ihre hohe Speicherbandbreite vor allem dann, wenn viele Threads gleichzeitig auf benachbarte Speicherbereiche zugreifen können, sodass coalesced memory accesses entstehen, wie in Abschnitt 4.4.1 beschrieben. Die unregelmäßigen Zugriffsmuster bei der Traversierung eines Baums erschweren eine solche Zusammenfassung von Speicherzugriffen erheblich. Dadurch kann die effektive Speicherbandbreite deutlich reduziert werden, was sich negativ auf die Gesamtleistung auswirkt.

Ein weiteres Problem stellt die Konstruktion des Baums dar. In der vorliegenden Implementierung wird der Barnes-Hut-Baum durch sukzessives Einfügen der Körper aufgebaut. Dieser Einfügealgorithmus ist inhärent sequentiell, da das Einfügen eines Körpers die Struktur des Baums verändern kann und somit Einfluss auf die Position später eingefügter

Körper hat. Eine effiziente Parallelisierung dieses Schritts ist daher nicht ohne Weiteres möglich.

Für eine effiziente GPU-Implementierung des Barnes-Hut-Verfahrens wären daher alternative Ansätze erforderlich, beispielsweise spezielle parallele Algorithmen zur Baumkonstruktion oder Datenstrukturen, die besser an die Speicherzugriffsmuster von GPUs angepasst sind. Die Entwicklung und Untersuchung solcher Verfahren stellt ein mögliches Thema für zukünftige Arbeiten dar.

6.4. Behandlung von Kollisionen

In der aktuellen Simulation werden Körper als punktförmige Massen ohne räumliche Ausdehnung modelliert. Für die Vermeidung von Singularitäten in der Kraftberechnung wird das Softening verwendet, welches den Abstand zwischen Körpern im Kraftgesetz künstlich vergrößert. Diese Methode verhindert numerische Instabilitäten, bildet jedoch physikalische Kollisionen nicht ab.

Wenn sich zwei Körper sehr stark annähern, entstehen daher große Gravitationskräfte, obwohl die Körper in einer realistischen Simulation miteinander kollidieren würden. Eine mögliche Erweiterung der Simulation besteht also darin, solche Kollisionen explizit zu erkennen und zu behandeln.

Hierfür müssten die Körper mit einem Radius modelliert werden, aus dem hervorgeht, bei welcher Distanz zwischen Körpern eine Kollision eintritt. Wird eine Kollision erkannt, könnten verschiedene physikalische Modelle angewandt werden, beispielsweise die Simulation elastischer oder unelastischer Stöße zwischen den Körpern. Alternativ könnten kollidierende Körper zu einem neuen Körper zusammengeführt werden, dessen Masse und Impuls aus den Eigenschaften der ursprünglichen Körper berechnet werden.

Die Einführung einer Kollisionsbehandlung würde zusätzlichen Rechenaufwand bedeuten, könnte jedoch zu einer physikalisch realistischeren Simulation führen, insbesondere bei Szenarien mit hoher Körperdichte oder häufigen Nahbegegnungen.

A. Erklärung zur Verwendung maschineller Hilfsmittel

Zur Überprüfung von Rechtschreibung, Grammatik und Ausdrucksstil wurden für Teile des Textes Machine Learning-basierte Werkzeuge wie LanguageTool und DeepL Write genutzt.

Literatur

- [1] Ernst Hairer, Christian Lubich und Gerhard Wanner. „Geometric numerical integration illustrated by the Störmer–Verlet method“. In: *Acta Numerica* 12 (Mai 2003), S. 399–450. ISSN: 1474-0508, 0962-4929. doi: 10.1017/S0962492902000144. URL: <https://hal.science/hal-04957214> (besucht am 17.03.2026).
- [2] Nü at German Wikipedia. *Bildbeschreibung: Schemazeichnung eines Octrees, einer Datenstruktur der Informatik*. Lizenz: CC BY-SA 3.0. 7. Apr. 2006. URL: <https://commons.wikimedia.org/wiki/File:Octree2.png> (besucht am 18.03.2026).
- [3] Paul Bode, Jeremiah P. Ostriker und Guohong Xu. „The Tree Particle-Mesh N-Body Gravity Solver“. In: *The Astrophysical Journal Supplement Series* 128.2 (Juni 2000), S. 561. ISSN: 0067-0049. doi: 10.1086/313398. (Besucht am 19.03.2026).
- [4] Diptajyoti Mukherjee u. a. „Fast Multipole Methods for N-body Simulations of Collisional Star Systems“. In: *The Astrophysical Journal* 916.1 (Juli 2021), S. 9. ISSN: 0004-637X. doi: 10.3847/1538-4357/ac03b2. (Besucht am 18.03.2026).
- [5] The Khronos® Group Inc. *About The Khronos Group*. URL: <https://www.khronos.org/about> (besucht am 17.03.2026).
- [6] The Khronos® Group Inc. *Khronos Members List*. URL: <https://www.khronos.org/members/list> (besucht am 17.03.2026).
- [7] The Khronos® Group Inc. *The Khronos Group Releases OpenCL 1.0 Specification*. Press Release. 9. Dez. 2008. URL: https://www.khronos.org/news/press/the_khronos_group_releases_openc1_1.0_specification (besucht am 17.03.2026).
- [8] NVIDIA Corporation. *NVIDIA CUDA GPU Compute Capability*. URL: <https://developer.nvidia.com/cuda/gpus> (besucht am 21.03.2026).
- [9] Advanced Micro Devices, Inc. *What is HIP?* URL: https://rocm.docs.amd.com/projects/HIP/en/docs-7.2.0/what_is_hip.html (besucht am 21.03.2026).
- [10] Advanced Micro Devices, Inc. *What is ROCm?* URL: <https://rocm.docs.amd.com/en/docs-7.2.0/what-is-rocm.html> (besucht am 21.03.2026).

- [11] Paulius Velesko u.a. „chipStar: Making HIP/CUDA applications cross-vendor portable by building on open standards“. In: *International Journal of High Performance Computing Applications* (14. Feb. 2026). ISSN: 1094-3420. DOI: 10.1177/10943420261423001. URL: <https://urn.fi/URN:NBN:fi:tuni-202603093048> (besucht am 17. 03. 2026).
- [12] The Khronos® SYCL™ Working Group. *SYCL™ 2020 Specification (revision 11)*. 7. Nov. 2025. URL: <https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf> (besucht am 17. 03. 2026).
- [13] The Khronos® Group Inc. *SYCL Implementations*. URL: <https://www.khronos.org/sycl/#sycl-implementations> (besucht am 21. 03. 2026).
- [14] Ruyman Reyes. *Bringing Nvidia® and AMD support to oneAPI*. URL: <https://oneapi.io/blog/bringing-nvidia-and-amd-support-to-oneapi/> (besucht am 21. 03. 2026).
- [15] AdaptiveCpp. *Compilation model - AdaptiveCpp Documentation*. URL: <https://adaptivecpp.github.io/AdaptiveCpp/compilation/> (besucht am 21. 03. 2026).
- [16] The Khronos® Group Inc. *OpenGL - The Industry's Foundation for High Performance Graphics*. URL: <https://www.khronos.org/opengl/> (besucht am 21. 03. 2026).
- [17] The Khronos® Group Inc. *Portal:OpenGL Concepts - OpenGL Wiki*. URL: https://wikis.khronos.org/opengl/Portal%3AOpenGL_Concepts (besucht am 21. 03. 2026).
- [18] The Khronos® Vulkan Working Group. *Vulkan® 1.4.347 - A Specification (with all registered extensions)*. URL: <https://registry.khronos.org/vulkan/specs/latest/pdf/vkspec.pdf> (besucht am 21. 03. 2026).
- [19] The Khronos® Group Inc. *Vulkan™ Overview*. Feb. 2016. URL: <https://www.khronos.org/assets/uploads/developers/library/overview/vulkan-overview.pdf> (besucht am 21. 03. 2026).
- [20] Microsoft. *Important Changes from Direct3D 11 to Direct3D 12*. URL: <https://learn.microsoft.com/en-us/windows/win32/direct3d12/important-changes-from-directx-11-to-directx-12> (besucht am 21. 03. 2026).
- [21] Graeme Leese u. a. *The OpenGL® Shading Language, Version 4.60.8*. URL: <https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.pdf> (besucht am 21. 03. 2026).
- [22] Microsoft. *High-level shader language (HLSL)*. URL: <https://learn.microsoft.com/en-us/windows/win32/direct3dhls/dx-graphics-hls> (besucht am 21. 03. 2026).
- [23] The Khronos® Group Inc. *Use Glslang to Translate HLSL Shaders to SPIR-V*. 12. Mai 2017. URL: <https://www.khronos.org/news/permalink/use-glslang-to-translate-hls-shaders-to-spir-v> (besucht am 21. 03. 2026).

- [24] The Khronos® Group Inc. *Khronos Hosts Open Source Slang Shading Language and Compiler*. 21. Nov. 2024. URL: <https://shader-slang.org/news/2024/11/21/khronos-hosts-open-source-slang-shading-language-and-compiler/> (besucht am 17.03.2026).
- [25] shader-slang. *Basic Convenience Features*. URL: <http://shader-slang.org/slang/user-guide/convenience-features> (besucht am 21.03.2026).
- [26] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. Version 4.5. URL: <https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf> (besucht am 22.03.2026).
- [27] W Kahan. „Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic“. In: (1996). URL: <https://faculty.cc.gatech.edu/~hyesoon/spr09/ieee754.pdf> (besucht am 22.03.2026).
- [28] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C, & 2D): Instruction Set Reference, A-Z*. URL: <https://cdrdv2.intel.com/v1/dl/getContent/671447> (besucht am 22.03.2026).
- [29] Advanced Micro Devices, Inc. *Introduction to the HIP programming model*. URL: https://rocm.docs.amd.com/projects/HIP/en/docs-7.2.0/understand/programming_model.html (besucht am 23.03.2026).
- [30] Advanced Micro Devices, Inc. *Memory management*. URL: https://rocm.docs.amd.com/projects/HIP/en/docs-7.2.0/reference/hip_runtime_api/modules/memory_management.html (besucht am 23.03.2026).
- [31] Advanced Micro Devices, Inc. *Asynchronous concurrent execution*. URL: https://rocm.docs.amd.com/projects/HIP/en/docs-7.2.0/how-to/hip_runtime_api/asynchronous.html (besucht am 23.03.2026).
- [32] Advanced Micro Devices, Inc. *HIP C++ language extensions*. URL: https://rocm.docs.amd.com/projects/HIP/en/docs-7.2.0/how-to/hip_cpp_language_extensions.html (besucht am 23.03.2026).
- [33] Advanced Micro Devices, Inc. *Hardware implementation*. URL: https://rocm.docs.amd.com/projects/HIP/en/docs-7.2.0/understand/hardware_implementation.html (besucht am 23.03.2026).
- [34] Advanced Micro Devices, Inc. *Hardware features*. URL: https://rocm.docs.amd.com/projects/HIP/en/docs-7.2.0/reference/hardware_features.html (besucht am 23.03.2026).
- [35] NVIDIA Corporation & affiliates. *SIMT Execution Model*. URL: <https://docs.nvidia.com/cuda/cuda-programming-guide/03-advanced/advanced-kernel-programming.html#simt-execution-model> (besucht am 23.03.2026).
- [36] Advanced Micro Devices, Inc. *Understanding GPU performance*. URL: https://rocm.docs.amd.com/projects/HIP/en/docs-7.2.0/understand/performance_optimization.html (besucht am 23.03.2026).

- [37] Advanced Micro Devices, Inc. *Performance guidelines*. URL: https://rocm.docs.amd.com/projects/HIP/en/docs-7.2.0/how-to/performance_guidelines.html (besucht am 23.03.2026).
- [38] Advanced Micro Devices, Inc. *Device memory*. URL: https://rocm.docs.amd.com/projects/HIP/en/docs-7.2.0/how-to/hip_runtime_api/memory_management/device_memory.html (besucht am 23.03.2026).
- [39] The Khronos® Group Inc. *Descriptor Sets*. URL: <https://docs.vulkan.org/spec/latest/chapters/descriptorsets.html> (besucht am 23.03.2026).
- [40] Simple Directmedia Layer. *SDL3/CategoryVulkan*. URL: <https://wiki.libsdl.org/SDL3/CategoryVulkan> (besucht am 23.03.2026).
- [41] U. Becciani, V. Antonuccio-Delogu und M. Gambera. „A modified parallel tree code for N-body simulation of the Large Scale Structure of the Universe“. In: *Journal of Computational Physics* 163.1 (Sep. 2000), S. 118–132. ISSN: 00219991. doi: 10.1006/jcph.2000.6557. URL: <http://arxiv.org/abs/astro-ph/0004249> (besucht am 23.03.2026).



Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit in allen Teilen selbstständig angefertigt und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt habe, und dass die Arbeit in gleicher oder ähnlicher Form in noch keiner anderen Prüfung vorgelegen hat. Mir ist bewusst, dass ich Autor/in der vorliegenden Arbeit bin und volle Verantwortung für den Text trage. ☒

Ich erkläre, dass ich wörtlich oder sinngemäß aus anderen Werken – dazu gehören auch Internetquellen – übernommene Inhalte als solche kenntlich gemacht und die entsprechenden Quellen angegeben habe. ☒

Mir ist bewusst, dass meine Arbeit auf Plagiate überprüft werden kann. Mir ist bekannt, dass es sich bei der Abgabe eines Plagiats um ein schweres akademisches Fehlverhalten handelt und dass Täuschungen nach der für mich gültigen Prüfungsordnung geahndet werden. ☒

Zusätzlich versichere ich, dass ich auf künstlicher Intelligenz (KI) basierende Werkzeuge nur in Absprache mit den Prüfern verwendet habe. Dabei stand meine eigene geistige Leistung im Vordergrund, und ich habe jederzeit den Prozess steuernd bearbeitet. ☒

Diese Werkzeuge habe ich im Quellenverzeichnis in der Rubrik „Übersicht verwendeter Hilfsmittel“ mit ihrem Produktnamen und einer Übersicht des im Rahmen dieser Prüfungs-/Studienarbeit genutzten Funktionsumfangs unter Angabe der Textstelle in der Arbeit vollständig aufgeführt. ☒

Ich versichere, dass ich keine KI-basierten Tools verwendet habe, deren Nutzung die Prüfer explizit schriftlich ausgeschlossen haben. Ich bin mir bewusst, dass die Verwendung von Texten oder anderen Inhalten und Produkten, die durch KI-basierte Tools generiert wurden, keine Garantie für deren Qualität darstellt. ☒

Ich verantworte die Übernahme jeglicher von mir verwendeter maschinell generierter Passagen vollumfänglich selbst und trage die Verantwortung für eventuell durch die KI generierte fehlerhafte oder verzerrte Inhalte, fehlerhafte Referenzen, Verstöße gegen das Datenschutz- und Urheberrecht oder Plagiate. ☒

Grimmitschau, 23.03.2026 T. Zandy

Ort, Datum

Unterschrift