UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

RigidBody-Physik-Engine mit Kollisionserkennung auf der GPU

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Daniel Keßelheim

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller Institut für Computervisualistik, AG Computergraphik
Zweitgutachter:	Kevin Keul, M.Sc) Institut für Computervisualistik, AG Computergraphik

Koblenz, im Januar 2015

Erklärung

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

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

.....

(Ort, Datum) (Unterschrift)

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

UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Aufgabenstellung für die Bachelorarbeit
Daniel Keßelheim (Matr.-Nr. 209 210 529)

Thema: **RigidBody-Physik-Engine mit Kollisionserkennung auf der GPU**

Physik-Engines sind schon seit langem Bestandteil von Computerspielen oder anderen Anwendungen. Sie regeln das korrekte physikalische Verhalten von Objekten innerhalb einer virtuellen Umgebung. Durch die steigende Komplexität solcher Welten, tritt die Performanz dieser Engines immer mehr in den Vordergrund. Die wachsende Leistung von Grafikkarten und die daraus resultierende Nutzung als GPGPU (General Purpose Computation Graphics Processing Unit) mit ihrer schnellen Verarbeitung paralleler Prozesse ist dafür nur von Vorteil.

NVIDIA hat dieses Potential vor einigen Jahren erkannt und mit „PhysX“ eine Engine geschaffen, die Kollisionserkennungen von der CPU auf die GPU auslagert. Sie sind momentan aber die einzigen die diese Methode erfolgreich nutzten.

Ziel der Arbeit ist es, eine Physik-Engine mit primitiven RigidBodies zu implementieren und diese durch eine Kollisionserkennung auf der GPU zu erweitern. Dazu werden bestehende Ansätze von Kollisionserkennungs-Algorithmen recherchiert und bewertet. Mit Hilfe einer Testumgebung soll dann der Performanz-Unterschied zwischen Kollisionserkennung auf CPU und GPU verglichen werden.

Schwerpunkte dieser Arbeit sind:

1. Recherche zu den Themen „Physik-Engine“, „RigidBody-Physik“, „Kollisionserkennung“ und „Kollisionserkennung auf GPU“
2. Einarbeitung in GPUs (Aufbau, Programmierung)
3. Einarbeitung in relevante Programmierumgebungen (CVK, etc.)
4. Konzeption und Implementierung der RigidBody-Physik, einer Testumgebung und der Kollisionserkennung
5. Demonstration und Bewertung der Ergebnisse
6. Dokumentation

Koblenz, den 24.06.2014

- Daniel Keßelheim -

- Prof. Dr. Stefan Müller-

Abstract

The present work introduces a rigid-body physics engine, focusing on the collision detection by GPU. The increasing performance and accessibility of modern graphics cards ensures that they can be also used for algorithms that are meant not only for imaging. This advantage is used to implement an efficient collision detection based on particles. The performance differences between CPU and GPU are presented by using a test environment.

Zusammenfassung

Die vorliegende Arbeit stellt eine Rigid-Body Physik-Engine vor, deren Schwerpunkt auf der Kollisionserkennung per GPU liegt. Die steigende Performanz und Zugänglichkeit moderner Grafikkarten sorgt dafür, dass sich diese auch für Algorithmen nutzen lassen, die nicht nur zur Bilderzeugung gedacht sind. Dieser Vorteil wird genutzt, um eine effiziente auf Partikeln basierende Kollisionserkennung zu implementieren. Mit Hilfe einer Testumgebung wird dann der Performance-Unterschied zwischen CPU und GPU dargestellt.

Inhaltsverzeichnis

1 Einleitung

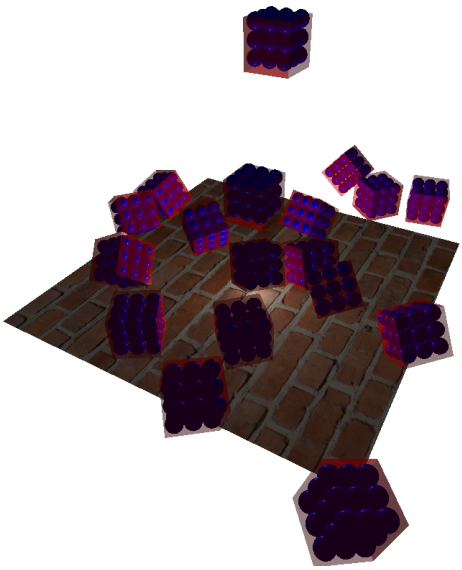


Abbildung 1: Screenshot der Demo
Kollision von durch Partikel angenäherten Boxen

Grafikkarten werden schon länger nicht mehr nur für die Bilderzeugung allein genutzt. Die stetig wachsende Performanz der GPUs (*Graphics Processing Units*) und die hohe Parallelisierbarkeit von Prozessen kann auch für viele allgemeinere Algorithmen genutzt werden. Man spricht von GPGPU (*General Purpose Computation on Graphics Processing Unit*), wenn ein Grafikprozessor über seinen eigentlichen Aufgabenbereich hinaus verwendet wird. Dieses Gebiet umfasst einfache Berechnungen, wie zum Beispiel eine Vektor-Addition, bis hin zu komplexeren Simulationen, von beispielsweise Fluiden. Durch die zunehmende Größe der Datenmengen in modernen Simulationen oder Computerspielen , wird die möglichst schnelle Bearbeitung dieser Daten immer wichtiger. Es kann also sehr ratsam sein, Arbeitsschritte, die parallelisierbar sind, von der GPU durchführen zu lassen.

Ein Bereich, in dem die besondere Leistung der GPU genutzt werden kann, sind Physik-Engines, *Engines* (Motoren) zur Simulation des korrekten physikalischen Verhaltens von Objekten im virtuellen Raum. Dabei kann das Testen vieler Objekte auf gegenseitige Kollisionen auf der CPU (*Central Processing Unit*) einige Zeit in Anspruch nehmen. Jedes Objekt muss nach und nach auf Berührung mit allen anderen Objekten getestet werden. Dieses Vorgehen lässt sich auf der GPU parallelisieren und dadurch beschleunigen, sodass alle Objekte gleichzeitig gegeneinander getestet werden können. Das kann besonders für *real-time* (Echtzeit) Simulation und große Objektanzahlen bedeutende Vorteile bringen.

Nvidia hat das Potential ihrer GPUs vor einigen Jahren erkannt und mit *"PhysX"* eine Engine geschaffen, die viele Berechnungen auf die GPU auslagert. Gute Grafikkarten sind mittlerweile auch erschwinglich und machen die Performanz-Vorteile somit für eine breite Masse nutzbar. Es gibt viele Engine-Hersteller, die diesem Trend folgen und somit Physik-Engines mit GPU-Unterstützung zu einem aktuellen Thema machen.

In dieser Arbeit sollen die Möglichkeiten und Performanz-Vorteile einer Grafikkarte anhand einer Kollisionserkennung aufgezeigt werden. Dazu wird eine Demo implementiert, in der Bewegung und Kollisionen fester Körper sowohl auf der CPU als auch auf der GPU simuliert werden können. Der Fokus liegt dabei in erster Linie auf der Umsetzung auf der GPU, aber auch der Nutzen einer Gitterstruktur zur Beschleunigung der Simulation auf der CPU wird sichtbar.

Das Thema Physik-Engines, ein genauerer Blick auf Kollisionserkennung und eine kurze Einführung in CUDA sind die Grundlagen, die den ersten Teil dieser Arbeit bilden. Im zweiten Teil folgen dann die Erklärungen zur Implementation und die Auswertungen der aufgestellten Performanz-Tests. Die Ausarbeitung endet mit einem kleinen Ausblick in die Zukunft.

2 Physik-Engines

Dieses Kapitel basiert größtenteils auf [Mil07]. Falls doch andere Quelle mit einbezogen wurden, sind diese an entsprechender Stelle angegeben.

Physik-Engines sind schon mehr als 35 Jahre alt. Anfangs waren es meist keine besonders komplexen Systeme, sondern nur Ansammlungen einzeln implementierter Effekte, die beschränkt und selten physikalisch korrekt waren. Beispielsweise konnte eine Pfeil-Simulation nicht einfach für eine Projektil-Simulation verwendet werden, obwohl beide einen Schuss nachahmen. Bis heute haben sich Physik-Engines zu viel umfassenderen Systemen entwickelt, die man als allgemeinen Code beschreiben könnte. Dieser besitzt Wissen über generelle Physik, aber weiß nichts über anwendungsspezifische Szenarios. Grundlegend ist eine Engine ein Rechner, der die Mathematik für physikalische Simulationen bewerkstelligen kann, ohne zu wissen, was simuliert wird. Dazu werden noch spiele- beziehungsweise umgebungsbezogene Daten benötigt, die zum Beispiel durch *Level-Editing Tools* (Bearbeitungswerkzeuge zum Erstellen einer Szenerie) generiert werden können.

Physik wird in Computerspielen, Simulationen oder anderen Anwendungen in vielen Bereichen benötigt. In der Optik für Licht und Reflektion, in der Mechanik für Gravitation, Auftrieb oder Sprungkraft. Effekte wie Funken, Feuerwerk, Projektil-Ballistik und Rauch können mit Partikel-Physik dargestellt werden. Flugsimulatoren, Fahrphysik für Autos oder *Ragdolls* (Stoffpuppe) zur Simulation menschlicher Skelette sind weitere Einsatzgebiete. Aber auch um beispielsweise einfache bewegliche und stapelbare Kisten zu realisieren, braucht man eine Physik-Engine.

Wie bei *Game-Engines* (Spiele-Engine) werden auch Physik-Engines oft in mehreren Spielen verwendet, da eine Neuentwicklung Zeit und Geld in Anspruch nimmt, die Entwickler in Hinsicht auf die anderen Spielinhalte nicht haben. Ein Vorteil ist, dass viele bereitgestellte Strukturen und Funktionen, wie etwa Integration, für verschiedene Simulationen genutzt werden können. Eine Engine kann aber auch Nachteile mit sich bringen, wenn sie falsch genutzt wird. Wenn zum Beispiel eine komplexe Engine für simple Objekte verwendet wird, kann das zu nicht notwendigen Geschwindigkeitseinbußen führen, was heutzutage aber eher ein Problem auf mobilen Geräten anstatt auf dem PC oder Spiele-Konsolen ist. Auch wenn die für eine Simulation benötigten Daten zu komplex oder schwer erreichbar sind, kann sich eine Physik-Engine möglicherweise nicht auszahlen.

Generell besteht eine Physik-Engine aus einer bestimmten Objektrepräsentation, auf die Kräfte wirken können, der für die Kraft-Berechnungen benötigten Mathematik und einer Kollisionserkennung und -behandlung. Ein Kollisions-System ist jedoch nicht zwingend erforderlich.

Für den Ansatz einer Engine muss man die Balance zwischen Komplexität beziehungsweise physikalischer Korrektheit und Eleganz der resultie-

renden Effekte abwägen. Denn oft kann auch eine physikalisch nicht ganz korrekte Simulation den gewünschten Effekt mit einfacherer, schnellerer Berechnung erzielen.

Dazu gibt es einige generelle Design-Faktoren, die bei einer Konzeption zu beachten sind. Die Darstellung der Anwendungsdomäne (Simulationsbereich), genau genommen die geometrische Repräsentation der darin simulierten Objekte, ist einer davon. Abfrage-Typen für die Kollisionserkennung und umgebungsbezogene Parameter sind festzulegen. Hier kommt die oben bereits genannte Abwägung zwischen Genauigkeit und Einfachheit wieder zum Tragen. Punkte wie Performanz und Robustheit sind auch nicht zu vernachlässigen. Durch bestimmte Hierarchien oder Daten-*Caching* (Zwischenspeichern) lässt sich die Leistung einer Engine verbessern. Numerische Robustheit wird durch Genauigkeit der Variablen und Berechnungen bestimmt. Geometrische Degeneration kann wiederum durch gute numerische Beständigkeit vermieden werden. Als Letztes sollte man sich noch über den Implementations-Aufwand und Gebrauch der Engine klar werden. Beinhaltende Features und deren Nutzen sind dabei genauer zu betrachten. [Eri05]

In den folgenden Abschnitten werden nun die verschiedenen Arten an Physik-Engines sowie deren mathematische und physikalische Grundlagen vorgestellt. Die genannten Faktoren werden an entsprechenden Stellen genauer beleuchtet. Ein Basiswissen über Lineare Algebra, Analysis sowie generelle Kenntnisse in Physik und Informatik (Computergrafik) werden vorausgesetzt.

2.1 Grundlagen

Physik-Engines können anhand ihrer verwendeten Objekte in verschiedene Klassen unterteilt werden. So wird zwischen Partikeln, Masse-Feder-Systemen, starren und deformierbaren Körpern unterschieden.

2.1.1 Partikel Physik

Partikel sind kleine Teilchen, die in der Computergrafik durch folgende Attribute repräsentiert werden:

Attribut	**Beschreibung**
Position	Lage im Koordinatensystem
Größe	Radius
Masse	Gewicht
Geschwindigkeit	momentane zeitliche Änderungsrate der Position
Beschleunigung	momentane zeitliche Änderungsrate der Geschwindigkeit

Abbildung 2 veranschaulicht diese Attribute.

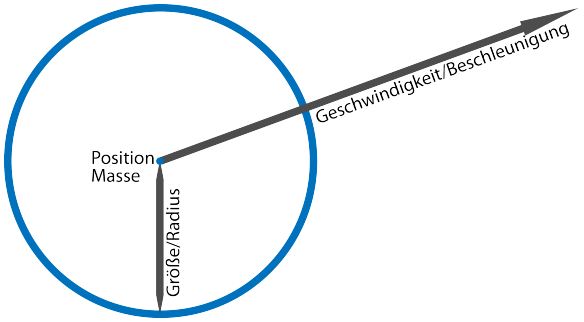


Abbildung 2: Partikel Repräsentation
mit den Attributen: Position, Masse, Größe/Radius,
Geschwindigkeit/Beschleunigung

Partikel haben keine Orientierung im Koordinatensystem, sie sind einfach nur Punktmassen ohne eine Ausrichtung im Raum und bewegen sich ohne Eigenrotation. Das Koordinatensystem kann je nach verwendeter Technik linkshändig (DirectX) beziehungsweise rechtshändig (OpenGL) sein, was jedoch keinerlei Auswirkung auf die physikalischen Berechnungen hat.

Die wichtigste mathematische Grundlage für Partikel Physik sind Vektoren zur Repräsentation von 2D- beziehungsweise 3D-Koordinaten im Raum. Und auch Positions- und Geschwindigkeitsänderungen können durch Vektoren dargestellt werden. Genauer gesagt kann jede Position auch als Positionsänderung betrachtet werden.

Vektoren sind geordnete Listen aus mehreren skalaren Werten, mit ihnen sind die selben Operationen wie mit Skalaren möglich, jedoch laufen sie nicht immer nach dem gleichen Muster ab. Ein Grundwissen über Vektor-Mathematik wird für diese Arbeit jedoch vorausgesetzt und für eine genauere Sicht auf Vektor-Operationen (Addition/Subtraktion, Skalarprodukt (Punktprodukt), Vektorprodukt (Kreuzprodukt) und so weiter) auf [Mil07] und [Ebe04] verwiesen.

$$\boldsymbol{a} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \ \boldsymbol{b} = \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta z \end{pmatrix} \ mit \ \Delta x = x_1 - x_0$$

Eine weitere mathematische Grundlage ist Infinitesimalrechnung (englisch *calculus*), also Differential- und Integral-Rechnung aus dem Bereich der Analysis. Mit ihnen lassen sich Veränderungen einer Position oder Geschwindigkeit über die Zeit beschreiben.

Zentrales Vorgehen in der Differentialrechnung ist das Finden von Extremwerten oder Änderungen und das dadurch verbundene Ableiten einer Funktion. Das gegenteilige Verfahren ist die Integralrechnung. Sie beinhaltet das Finden der Stammfunktion (Aufleiten) und die Flächenberechnung

zwischen Koordinatenachsen und durch Funktionen definierte Kurven. Für einen genaueren Blick wird auch hier auf [Mil07] und [Ebe04] verwiesen.

Die physikalischen Grundlagen basieren auf *Newton's Laws of Motion* (Newtons Gesetze der Bewegung):

1. Gesetz
Ein Objekt bewegt sich mit konstanter Geschwindigkeit solange keine Kraft auf es wirkt.

2. Gesetz:
Eine Kraft auf Objekte erzeugt eine Beschleunigung proportional zu der Masse des Objekts.

3. Gesetz:
Wird Kraft auf ein Objekt ausgeübt, so existiert eine Kraft gleicher Stärke in entgegengesetzter Richtung, die auf dieses oder ein anderes Objekt wirkt. (Aktion + Reaktion)

Für die momentane Partikel-Betrachtung gelten das 1. und 2. Gesetz. Zu 1. sei gesagt, dass in der realen Welt kein Körper eine konstante Geschwindigkeit halten kann (Ausnahme: der Ruhezustand), da Dämpfungen (*damping*) durch Luft und Reibung existieren. Beim 2. Gesetz ist noch einmal hervorzuheben, dass sich die Kräfte auf die Beschleunigung des Objekts und nicht direkt auf Geschwindigkeit und Position auswirken. Aus diesem Grundsatz resultiert Gleichung 1.

$$F = m * a \tag{1}$$

F *Kraft*
m *Masse*
a *Beschleunigung*

Die Masse darf niemals 0 sein, denn ohne Gewicht gibt es auch keine resultierende Kraft. Andererseits kann sie auch nicht unendlich sein, da das Objekt so von keiner realistischen Kraft bewegt werden könnte. In Programmiersprachen gibt es aber auch praktisch keinen Wert "unendlich".

Im Zusammenhang mit Kraft kommt auch der Begriff Impuls (englisch *Momentum*) und der zugehörige Impulserhaltungssatz ins Spiel. Ein Impuls ist das Produkt aus Masse und Geschwindigkeit (siehe Gleichung 2) und beschreibt den Bewegungszustand eines Körpers. Der Impulserhaltungssatz lautet:

Der im abgeschlossenen System beinhaltete Impuls ist vor und nach einer Wechselwirkung gleich.

Bei einem Zusammenstoß geben Objekte ihren Impuls an die Kontaktpartner weiter. Darauf wird an späterer Stelle noch genauer eingegangen. Ein Kriterium, durch das sich Physik-Engines unterscheiden können, ist die Nutzung der Kraft (*force-based*) oder des Impulses (*impulse-based*) bei der Berechnung für Positions- und Geschwindigkeitsänderungen.

$$I_l = m * v \tag{2}$$

I_l *linearer Impuls*
m *Masse*
v *lineare Geschwindigkeit*

Die wichtigste Kraft in einer Physik-Engine ist die Gravitation (Schwerkraft). Sie ist abhängig von der Gravitationskonstante g, den Massen und dem Abstand der beteiligten Körper. Sie definiert das grundsätzliche Verhalten der Objekte innerhalb einer Anwendung.

$$F = g * \frac{m_1 * m_2}{r^2} \tag{3}$$

m_i *Masse von Objekt i*
r *Distanz der beiden Objekte*
g *Gravitationskonstante*

$$F = m * g \tag{4}$$

$$mit\ g = 9,807\ m/s^2$$

Die Massen aus Gleichung 3 können auf der Erde vernachlässigt werden und g kann direkt als Beschleunigung angewandt werden, die in negativer y-Achsen-Richtung wirkt.

$$\mathbf{g} = \begin{pmatrix} 0 \\ -g \\ 0 \end{pmatrix}$$

In Computerspielen werden oft größere Werte ($15 - 20\ m/s^2$) für g benutzt, um die gewünschten Effekte zu erreichen.

Nun werden die mathematischen und physikalischen Grundlagen zusammengeführt und die Bewegung eines Partikels innerhalb einer Simulationsdomäne betrachtet.

In jedem Simulationsschritt (*Frame*) schaut sich die Engine nun jedes Objekt an und findet durch die beteiligten Kräfte deren Beschleunigung heraus. Damit führt sie eine Integration durch, um die resultierenden Geschwindigkeits- und Positionsänderung zu bestimmen. Diese Aktualisierung der genannten

Werte geschieht innerhalb eines definierten Zeitintervalls, das fest gegeben oder dynamisch sein kann.

Unter den Attributen Position, Geschwindigkeit und Beschleunigung herrschen dabei die Beziehungen aus den Gleichungen 5 und 6. Geschwindigkeit ist die Ableitung der Position und Beschleunigung die Ableitung der Geschwindigkeit über die Zeit.

$$\boldsymbol{p}' = \boldsymbol{p} + \dot{\boldsymbol{p}}t \tag{5}$$

$$\dot{\boldsymbol{p}}' = \dot{\boldsymbol{p}} + \ddot{\boldsymbol{p}}t \tag{6}$$

$\boldsymbol{p}$ *aktuelle Position*
$\boldsymbol{p}'$ *neue Position*
$\dot{\boldsymbol{p}}$ *Geschwindigkeit*
$\ddot{\boldsymbol{p}}$ *Beschleunigung*

Wenn die Beschleunigung bekannt ist, lässt sich die neue Geschwindigkeit berechnen. Dadurch ist es ein Leichtes die neue Position zu bestimmen. Kurz gesagt, hängt Geschwindigkeit von Beschleunigung und Position wiederum von Geschwindigkeit ab. Ist die vorausgehende Beschleunigung dabei $a = 0$, so bringt dies keine Veränderungen mit sich. Ist jedoch $a > 0$ oder $a < 0$, erfährt das entsprechende Objekt eine Beschleunigung beziehungsweise ein Abbremsen (negative Beschleunigung).

Damit Partikel in einer Simulationsdomäne nicht unendlich in Bewegung bleiben und ihr Verhalten realistischer wirkt, wird in die Geschwindigkeits-Integration oft ein Dämpfungs-Wert d (englisch *damping*) mit einbezogen. Fortan werden Geschwindigkeit und Beschleunigung mit v beziehungsweise a bezeichnet. Die Gleichungen für Position und Geschwindigkeit sehen somit wie folgt aus:

$$\boldsymbol{p}' = \boldsymbol{p} + \boldsymbol{v}t \tag{7}$$

$$\boldsymbol{v}' = \boldsymbol{v} * d^t + \boldsymbol{a}t \tag{8}$$

mit $\boldsymbol{v} = \dot{\boldsymbol{p}}$ *und* $\boldsymbol{a} = \dot{\boldsymbol{v}} = \ddot{\boldsymbol{p}}$
d^t *Dämpfung pro Sekunde*

Eine Engine, die auf reiner Partikel-Physik beruht, kann zum Beispiel schon für Simulationen verschiedener Ballistik-Effekte oder Partikel-Systeme, wie beispielsweise Feuerwerke, genutzt werden. Eine Kollisionsabfrage ist bis jetzt noch nicht vorhanden, wie man sie jedoch einbinden kann, wird im nächsten Abschnitt beschrieben.

2.1.2 Massen-Gesamtheits Physik

Masse-Gesamtheits-Systeme (englisch *mass-aggregate physics*) verbinden Partikel zu komplexeren Gebilden. Ein Objekt wird durch viele kleine Punkt-Massen repräsentiert, die durch Stangen, Federn oder *Constraints* (Einschränkungen) miteinander verbunden sind, wodurch auch oft von Masse-Feder-Systemen die Rede ist.

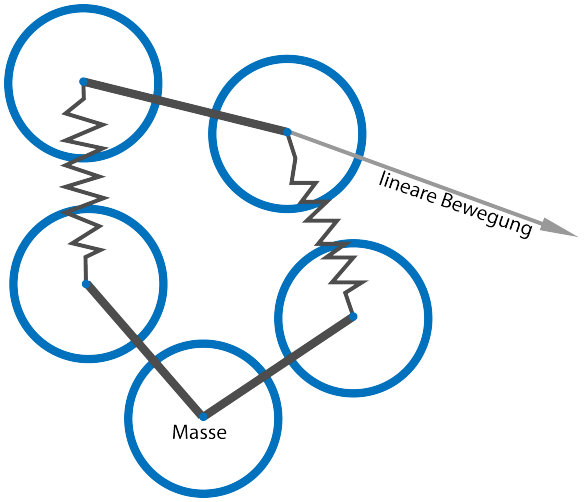


Abbildung 3: Objekt als Kollektion kleiner Massen
Partikel verbunden durch Federn und *Constraints*

Für das Verhalten der Partikel werden erst einmal keine weiteren Grundlagen benötigt, nur für die Elemente, die sie verbinden. Dazu zählt die Mathematik hinter linearen elastischen Verformungen. Sie wird durch *Hook's Law* (Hookesches Gesetz) beschrieben. Ruhezustand, Ausdehnung und Kompression einer Feder werden durch die generelle Gleichung 9 dargestellt. Wie aus dieser Formel hervorgeht, ist die an den Enden wirkende Kraft abhängig von der Federkonstante und Intensität der Verformung.

$$F = -k * \Delta l \tag{9}$$

mit $\Delta l = l - l_o$

F	*Kraft*
k	*Federkonstante*
Δl	*Distanz der Ausdehnung bzw. Kompression*
l	*aktuelle Federlänge*
l_0	*Ruhelänge*

Diese Kraft wirkt an beiden Enden einer Feder (F und $-F$).

Mit diesem Grundgerüst lassen sich verschiedene Variationen von Federkräften modellieren. Verankerte Federn haben statt einem zweiten Objekt einfach eine fixe Position als Endpunkt. Stangen werden durch steife Federn

erzeugt, bei denen eine große Federkonstante die Verformung verhindert. Bei einem Bungee-Seil zum Beispiel werden nur die Zugkräfte bei der Ausdehnung berücksichtigt. Auch Kräfte, die man erst nicht vermutet, lassen sich durch Federn umsetzen. Der Auftrieb, das Gewicht des durch ein Objekt verdrängten Wassers, ist eine davon. Er kann durch 2 entgegenwirkende Federn konstruiert werden, die einen Körper je nach Gewicht mehr über oder unter die Wasseroberfläche ziehen. Für einen genauen Blick auf das Modellieren solcher Federn-Systeme wird auf [Mil07] verwiesen.

Die verschiedenen, durch Federn umgesetzten Kräfte wirken, neben der Gravitation, auch auf die Partikel. Das gleichzeitige Wirken mehrerer Kräfte auf ein Objekt wird durch das D'Alembert's Prinzip beschrieben. Die resultierende Gesamtkraft für ein Teilchen ist einfach die (Vektor-)Aufsummierung aller beteiligten Kräfte. Jedem Integrationsschritt wird dafür eine Akkumulationsstufe vorgeschoben, in der alle Kräfte aufsummiert werden.

$$F = \sum_i f_i \tag{10}$$

F *gesamte Kraft*
f_i *einzelne Kräfte*

Dabei ist Gravitation immer für alle Objekte präsent und konstant, andere Kräfte sind dies jedoch nicht. Innerhalb einer Szene muss also nicht jede Kraft für jeden Körper gefunden werden. Außerdem brauchen die einzelnen Objekt-Klassen (hier nur Partikel) im Code nicht zu wissen, wie die Kraft-Berechnungen aussehen, wodurch diese ausgekapselt werden können. Dieses Auskapseln wird innerhalb des Engine-Codes durch *Interfaces* (Schnittstellen) und Polymorphismus (Vererbung) umgesetzt. Sogenannte *Force-Generators* (Kraft-Generatoren) werden genutzt. Dabei existiert ein Interface für Force-Generators und je eine abgeleitete Klasse pro vorkommender Kraft. So gibt es beispielsweise einen Kraft-Generator für Schleppkraft, der die Berechnung nur für Objekte durchführt, auf die diese Kraft auch wirkt. Auch verschiedene Dämpfungen könnten per Force-Generator ausgelagert werden.

Eine andere Art von Verbindungen zwischen Punktmassen sind die bereits genannten Contraints. Sie werden nicht durch Federn beziehungsweise Kraft-Generatoren umgesetzt und bringen eine zusätzliche Erweiterung der Engine mit sich.

Unter Constraints versteht man Einschränkungen, die zwischen Objekten herrschen können. Allgemein wird mit ihnen ein Abstand zwischen 2 Körpern definiert. Dieser ist mit dem folgenden Verfahren besser modellierbar als mit Federn und deren dazu nötigen Federkonstanten. Verbindungen wie Stangen (fester Abstand ist immer gegeben) oder Kabel (reagiert erst ab bestimmten Abstand) lassen sich so besser realisieren. Um zwei Partikel

in einer bestimmten Distanz zueinander zu halten, muss ihr Radius mit einbezogen werden. Alle Partikel innerhalb einer Simulation haben dabei den selben Radius r. Ist der Abstand zwischen 2 Teilchen kleiner als der Durchmesser d ($d = 2r$) eines Partikels, so existiert ein Kontakt zwischen ihnen. Durch diesen einfachen Test wird eine erste Kollisionserkennung eingeführt. Ab jetzt gilt auch Newtons 3. Gesetz der Bewegung.

Wichtige Parameter für die Auflösung dieser Kontakte sind die *closing velocity* (Schließgeschwindigkeit) der beteiligten Objekte, der im vorangehenden Abschnitt bereits genannte Impuls und der *coefficient of restitution* (Rückgabe-Koeffizient). Die Schließgeschwindigkeit ist die Gesamtgeschwindigkeit der kollidierenden Körper (siehe Gleichung 11). Sie wird für den Impulserhaltungssatz aus Kapitel 2.1.1 benötigt, ist also vor und nach der durch den Zusammenstoß hervorgerufenen Wechselwirkung gleich.

$$\boldsymbol{v}_c = \boldsymbol{v}_a(\widehat{\boldsymbol{p}_b - \boldsymbol{p}_a}) + \boldsymbol{v}_b(\widehat{\boldsymbol{p}_a - \boldsymbol{p}_b}) = (\boldsymbol{v}_a - \boldsymbol{v}_b)(\widehat{\boldsymbol{p}_a - \boldsymbol{p}_b}) \tag{11}$$

$\boldsymbol{v}_c$ *Schließgeschwindigkeit*
$\boldsymbol{v}_a/\boldsymbol{v}_b$ *Geschwindigkeit von Objekt a bzw. b*
$\boldsymbol{p}_a/\boldsymbol{p}_b$ *Position von Objekt a bzw. b*
$\widehat{\boldsymbol{p}}$ *markiert einen Einheitsvektor*
Für sich voneinander entfernende Objekte ist $\boldsymbol{v}_c < 0$

$\widehat{\boldsymbol{n}} = (\widehat{\boldsymbol{p}_a - \boldsymbol{p}_b})$ wird auch Kontakt-Normale (englisch *contact normal*) genannt.

Der Restitutionskoeffizient kontrolliert die Geschwindigkeit der Objekte nach der Kollision. Bei $c_r = 1$ geschieht der Abstoß mit dem selben Tempo wie der Aufprall, während bei $c_r = 0$ die beiden Körper verschmelzen und sich gemeinsam weiterbewegen. Der Koeffizient ist materialabhängig, sodass sich zum Beispiel ein Tennisball anders verhält als ein Schneeball.

$$\boldsymbol{v}'_c = -c_r * \boldsymbol{v}_c \tag{12}$$

$\boldsymbol{v}'_c$ *Schließgeschwindigkeit nach Kollision*
$\boldsymbol{v}_c$ *Schließgeschwindigkeit vor Kollision*
c_r *Restitutionskoeffizient*

$\boldsymbol{v}'_c$ wird oft auch als Abstoßgeschwindigkeit bezeichnet.

Das Thema Kollisionserkennung und -behandlung, also das Finden eines Kontakt-Punktes und das Auflösen dieses Zusammenstoßes, wird in Kapitel 3 tiefgehender und anschaulicher vorgestellt.

Ein Vorteil von Masse-Gesamtheits Physik ist, dass man weiterhin nur Rücksicht auf die lineare Bewegung der einzelnen Teilchen nehmen muss. Durch die Verbindungen bekommt man aber nur sehr schlecht wirklich starre Objekte zustande.

Eine *mass-aggregate* Physik-Engine erweitert also Partikel-Physik um Federn und Contraints, Rotation wird noch immer nicht betrachtet. Die generellen drei Komponenten, aus denen diese Engine besteht, sind Partikel mit ihren Attributen als Objektrepräsentation, Kraft-Generatoren und ein Kollisionssystem. In jedem Frame werden nun zuerst die internen Partikel-Daten berechnet und dann alle dem Partikel entsprechende *Force-Generators* aufgerufen, um danach die neuen Geschwindigkeiten beziehungsweise Positionen durch Integration aktualisieren zu können. Dadurch können Kontakte zwischen Objekten entstehen, die durch das Kollisionssystem gesammelt und aufgelöst werden. Gegebenenfalls vorliegende Einschränkungen werden überwacht.

Diese Art von Engine kann in verschiedene Anwendungen für beispielsweise Seilbrücken, Kabel oder *Blob-Games*, genutzt werden. Größere Objekte sollte man jedoch nicht aus Partikeln zusammensetzen. Für diese gibt es bessere Strukturen, nämlich *Rigid-Bodies*, die im nächsten Abschnitt behandelt werden.

2.1.3 Rigid-Body Physik

Rigid-Bodies (starre Körper) werden nun nicht mehr als reine Partikel oder Konstruktionen aus diesen betrachtet. Sie können jegliche feste Form annehmen.

Die größte Neuerung, die Rigid-Body Physik mit sich bringt, konnte bei den bisherigen partikelbasierten Systemen noch ignoriert werden. Die Rede ist von Eigenrotationen der Objekte. Eine Rotation ist eine Orientierungsänderung über die Zeit, dabei ist die angulare Geschwindigkeit die 1. Ableitung von Orientierung, analog zu linearer Geschwindigkeit und Position.

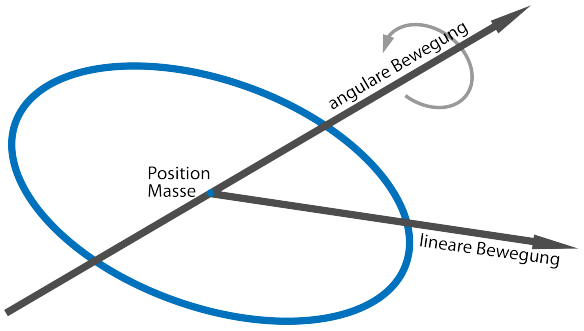


Abbildung 4: Rigid-Body Repräsentation
Körper mit linearer und angularer Bewegung

Objekte werden nun nicht mehr nur durch eine Position im Raum, sondern auch durch einen Winkel relativ zu einer vorgegebenen Richtung angegeben. Diese Orientierung kann als Matrix dargestellt werden, die im 2D-Raum wie folgt aussieht:

$$\boldsymbol{\Theta} = \begin{pmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{pmatrix}$$

$\boldsymbol{\Theta}$	*Orientierung*
θ	*Winkel*

Mit ihr lässt sich der Aufenthaltsort eines Punktes innerhalb eines Körpers nach einer Eigenrotation durch Formel 13 bestimmen.

$$\boldsymbol{p}' = \boldsymbol{\Theta} * \boldsymbol{p}_r + \boldsymbol{p}_o \tag{13}$$

$\boldsymbol{p}'$	*neue Position eines Punktes*
$\boldsymbol{p}_r$	*relative Position des Punktes zum Objekt-Ursprung*
$\boldsymbol{p}_o$	*Objekt-Ursprung*

Diese Formel gilt sowohl im 2-dimensionalen als auch im 3-dimensionalen Raum, lediglich die Definition der Rotationsmatrix unterscheidet sich. Auf die Bedeutung des Ursprungs wird in einem späteren Textabschnitt eingegangen.

Im 3D-Raum sind Rotationen um jede Koordinatenachse möglich, ein Objekt hat so 3 *degrees of freedom* (Freiheitsgrade). Die sogenannten Euler Winkel (englisch *euler angles*) stehen für je eine Drehung um eine der drei Koordinatenachsen. Sie werden in Abbildung 5 anhand der Bewegungen eines Flugzeuges vorgestellt.

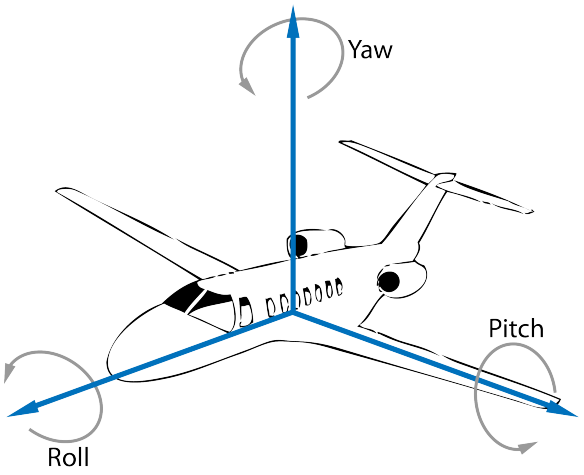


Abbildung 5: Euler Winkel

pitch	Neigen	Rotation um x-Achse
yaw	Gieren	Rotation um y-Achse
roll	Rollen	Rotation um z-Achse

Jede beliebige Rotation kann durch Kombination dieser drei Winkel erzeugt werden und wird durch eine 3x3-Matrix, wie sie auch beim *Rendering* (Darstellen) genutzt wird, definiert.

$$\mathbf{\Theta} = \begin{pmatrix} tx^2 + c & txy - sz & txz + sy \\ txy + sz & ty^2 + c & tyz - sx \\ txz - sy & tyz + sx & tz^2 + x \end{pmatrix} \tag{14}$$

x, y, z	*Koordinatenachsen*
c	$= cos(\theta)$
s	$= sin(\theta)$
t	$= 1 - cos(\theta)$
θ	*Winkel*

Diese Darstellung bringt jedoch einige Nachteile mit sich. Da eine Matrix auch andere Transformationen darstellen kann, können numerische Fehler Probleme verursachen. So muss die Matrix periodisch immer wieder normalisiert und überprüft werden, ob sie überhaupt noch eine Rotation repräsentiert. Andererseits ist es auch nicht ganz optimal, nur 3 Freiheitsgrade in 9 Werten zu speichern.

Eine besseres und oft benutztes Element sind Quaternionen. Sie sind kompakter und benötigen nur 4 Werte zur Definition einer Orientierung und basieren auf imaginären Zahlen $(a + b_i)$. Gleichung 15 zeigt die allgemeine Form eines Quaternions, wobei w, x, y, z Elemente aus den realen Zahlen $(\mathbb{R}^4)$ und i, j, k Imaginäranteile sind. Analog zur Rotations-Matrix ist auch ein Quaternion durch einen Winkel θ und eine Achse (x, y, z) definierbar.

$$q = w + xi + yj + zk \tag{15}$$

$$q = \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} cos(\frac{\theta}{2}) \\ xsin(\frac{\theta}{2}) \\ ysin(\frac{\theta}{2}) \\ zsin(\frac{\theta}{2}) \end{bmatrix}$$

Besitzt ein Quaternion Einheitslänge $(\sqrt{w^2 + x^2 + y^2 + z^2} = 1)$, so repräsentiert er eine Rotation. Diese Operation und auch die Darstellungsform ähnelt einem 4-elementigen Vektor, was sie jedoch nicht sind. Die Mathematik hinter Quaternionen ist nicht sehr geläufig und wird in [Ebe04] und [Mil07] genauer vorgestellt.

Das Speichern einer Orientierung als Quaternion ist zwar effizienter, aber bei einigen Berechnungen kann eine Matrix doch vorteilhafter sein, beispielsweise bei der Aktualisierung des Trägheit-Tensors. Wie man ein Quaternion in eine Rotationsmatrix umwandelt, zeigt Gleichung 16.

$$\boldsymbol{\Theta} = \begin{pmatrix} 1-(2y^2+2z^2) & 2xy+2tw & 2xz-2yw \\ 2xy-2zw & 1-(2x^2+2z^2) & 2yz+2xw \\ 2xz+2yw & 2yz-2xw & 1-(2x^2+2y^2) \end{pmatrix} \tag{16}$$

Durch Rotationen kommen auch weitere Objekt-Attribute hinzu. Die oben bereits genannte angulare Geschwindigkeit, der Objekt-Ursprung (*origin*) und das Massezentrum (*center of mass*) sind einige davon. Der Ursprung ist eine vordefinierte Position, um die sich ein Objekt relativ befindet, dies kann der Mittelpunkt sein, muss es aber nicht. Würfel haben zum Beispiel oft ihren Origin an einer Ecke. Anders als Partikel sind Rigid-Bodies mehr als nur einfache Punkte, müssen aber im Koordinatensystem auch eindeutig positioniert werden können. Der Ursprung hilft dabei, indem er einen Punkt definiert, um den der Körper definiert ist. Das Massezentrum ist der Gleichgewichtspunkt eines Objekts, er wird durch die Masseverteilung innerhalb des Körpers bestimmt. Dabei muss dieser Punkt nicht innerhalb des Objektes liegen, ein Donut hat sein Massezentrum beispielsweise in seinem Loch. Indem man das Massezentrum und den Ursprung auf den selben Punkt legt, lassen sich einige Berechnungen vereinfachen. Angulare Geschwindigkeit ist ein 3-elementiger Vektor, zusammengesetzt aus Rotationsachse und Spinnrate, angegeben in Radiant pro Sekunde. (siehe Gleichung 17).

$$\boldsymbol{w} = \boldsymbol{r} * s \tag{17}$$

$\boldsymbol{w}$ *angulare Geschwindigkeit*
$\boldsymbol{r}$ *Rotationsachse*
s *Spinnrate*

Mit ihr kann die Orientierung eines Objekts aktualisiert werden (siehe Gleichung 18. Analog zu linearer Geschwindigkeit zeigt Gleichung 19, dass auch angulare Geschwindigkeit abhängig von Beschleunigung ist. Ebenso gilt, dass angulare Beschleunigung die Ableitung von angularer Geschwindigkeit über die Zeit ist.

$$q' = q + \frac{\delta t}{2}\boldsymbol{w}q \tag{18}$$

q *Rotations-Quaternion*

$$\boldsymbol{w}' = \boldsymbol{w} * (d_a)^t + \boldsymbol{a}_a t \tag{19}$$

$\boldsymbol{a}_a$ *angulare Beschleunigung*
d_a *angularer Dämpfungskoeffizient*

Auch hier kann, vergleichbar zu linearer Geschwindigkeit, ein Dämpfungskoeffizient verwendet werden. In Gleichung 18 wird der Rotations-Quaternion q per angularer Geschwindigkeit $\boldsymbol{w}$ über eine bestimmte Zeit t aktualisiert. Dabei muss die angulare Geschwindigkeit ebenfalls als Quaternion vorliegen, um die Berechnung durchführen zu können. Dieser muss nicht normalisiert sein, da mit ihm keine Rotation repräsentiert wird. Der Geschwindigkeits-Vektor lässt sich wie folgt als Quaternion darstellen:

$$\boldsymbol{w} = \begin{bmatrix} 0 \\ w_x \\ w_y \\ w_z \end{bmatrix}$$

Auf Seiten der physikalischen Grundlagen gelten Newton's Laws of Motion auch weiterhin, wie bei den vorangegangenen partikel-basierten Engines. Kräfte, die lineare Bewegungen erzeugen, werden auf den Ursprung eines Rigid-Body ausgeführt. Für angulare Bewegung werden die Gesetze jedoch erweitert. Newton 2 für Rotationen besagt somit:

Die Änderungsrate der angularen Geschwindigkeit ist abhängig vom Dreh- und Trägheitsmoment.

$$\boldsymbol{a}_a = \boldsymbol{T}^{-1} * \boldsymbol{\tau} \tag{20}$$

$\boldsymbol{a}_a$ *angulare Beschleunigung*
$\boldsymbol{T}$ *Trägheits-Tensor*
$\boldsymbol{\tau}$ *Drehmoment*

Drehmoment (englisch *torque*) ist bei angularer Bewegung der analoge Begriff zum Impuls bei linearer Bewegung. Es beschreibt also die Änderungsrate von angularem Impuls. Wenn eine Kraft auf einen anderen Punkt innerhalb eines Körpers wirkt als auf das Massezentrum, so erzeugt dies ein Drehmoment. Die Stärke hängt von der relativen Position dieses Punktes zum Massezentrum ab: [Har07]

$$\boldsymbol{\tau} = p_r \times F = \boldsymbol{T} * \boldsymbol{a}_a \tag{21}$$

$\boldsymbol{p}_r$ *relative Position*
$\boldsymbol{T}$ *Trägheits-Tensor*

Trägheit (englisch *inertia*) ist das rationale Äquivalent zur Masse. Sie ist abhängig von Form, Größe und Masse eines Körpers und definiert, wie schwer es ist, die Rotationsgeschwindigkeit dieses Objekts zu ändern, beziehungsweise es überhaupt in Rotation zu versetzen. Dabei sind verschiedene Trägheitswerte pro Koordinatenachse möglich. Trägheit ist abhängig von der

Objektmasse und der Distanz zwischen Massezentrum und Rotationsachse. Repräsentiert wird das Trägheitsmoment als Tensor (englisch *inertia tensor*), einer generalisierten Form einer 3x3-Matrix. Die darin gespeicherten Werte stehen für die Tendenz, in die ein Objekt rotieren würde.

$$\boldsymbol{T} = \begin{pmatrix} T_x & -T_{xy} & -T_{xz} \\ -T_{xy} & T_x & -T_{yz} \\ -T_{xz} & -Tyz & T_z \end{pmatrix}$$

$$mit\ T_{ab} = \sum_{i=1}^{n} m_i a_{p_i} b_{p_i}$$

a, b	*Achsen*
a_p, b_p	*Abstand zu Massezentrum auf entsprechender Ache*

Die Diagonale der Matrix bestimmt das Trägheitsmoment der entsprechenden Achse. Die restlichen Stellen sind mit dem sogenannten *Product of Inertia* (Trägheitsprodukt) gefüllt. Dieses Produkt beschreibt den Transfer einer Rotation von Achse zu Achse. Dadurch wird keine Richtung des Drehmoment-Vektors benötigt. Wenn alle Trägheitsprodukte $= 0$ sind und der Drehmoment-Vektor der x-, y- oder z-Achse entspricht, so wirkt die angulare Beschleunigung in Richtung des Drehmoments.

Nach einer Rotation muss auch der Inertia-Tensor aktualisiert werden. Dies geschieht nach Gleichung 22. [Har07]

$$\boldsymbol{T}(t)^{-1} = \boldsymbol{R}(t)\boldsymbol{T}(0)^{-1}\boldsymbol{R}(t)^T \tag{22}$$

$\boldsymbol{R}(t)$	*Rotationsmatrix*
$\boldsymbol{T}^{-1}$	*markiert eine inverse Matrix*
$\boldsymbol{R}^T$	*markiert eine transponierte Matrix*

Ebenso wie Newtons Gesetze gilt auch das D'Alemberts Prinzip für rotationale Bewegungen. Bei linearer Bewegung können alle auf einen Körper wirkende Kräfte aufsummiert werden, um so die resultierende Beschleunigung in einem Schritt berechnen zu können. Dieses Prinzip gilt bei angularer Bewegung für das Drehmoment. Der Effekt einer Serie von Drehmomenten ist der gleiche wie bei der Kombination all dieser.

$$\boldsymbol{\tau} = \sum_i \tau_i \tag{23}$$

Daher kann, analog zur Kraft, auch ein Akkumulationsschritt für Drehmomente vor jede Integration geschoben werden. Dabei muss beachtet werden, dass Kräfte die auf das Massezentrum wirken kein Drehmoment generieren und so nur dem Kraft-Akkumulator aufaddiert werden müssen. Jede andere

Kraft wird zu beiden, beziehungsweise das resultierende Drehmoment nur zum Drehmoment-Akkumulator hinzugefügt.

Analog zu den Force-Generators aus Kapitel 2.1.2 lassen sich nun auch Torque-Generators (Drehmoment-Generatoren) realisieren. Worauf an dieser Stelle aber nicht weiter eingegangen wird.

Der Integrationsschritt für Rigid-Bodies erweitert den von Partikeln nun um die rotationale Komponente. Basierend auf den entsprechenden Kräften und Drehmomenten wird die Position und Orientierung eines Objekts aktualisiert. Die neue Position wird dabei, wie bereits erwähnt, genau wie bei einer Partikel-Engine bestimmt. Für die Orientierung werden mit den, aus den Kräften resultierenden, Drehmomenten die angulare Beschleunigung berechnet. Mit dieser kann man wiederum die angulare Geschwindigkeit und damit schlussendlich die neue Orientierung bestimmen.

Wie man ein Kollisionsmodell für eine Rigid-Body Engine realisiert, wird in Kapitel 3 ausführlich vorgestellt. Aber auch ohne Kollisionserkennung und -behandlung lassen sich mit Hilfe der Feder-Kräfte und der erweiterten Force-Generators Anwendungen wie zum Beispiel einfache Flug- oder Boots-Simulatoren umsetzen.

2.1.4 Soft-Body Physik

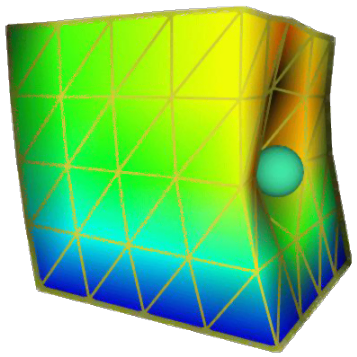


Abbildung 6: Soft-Body Repräsentation [DDCB00]
Umsetzung per *Free Form Deformation*

In der Realität ist kein Objekt wirklich fest, selbst einer Eisenkugel kann man mit genügend Kraft eine Delle verpassen. Um deformierbare Körper simulieren zu können, sind weitere Parameter hinzuzufügen, nämlich Elastizität, Spannung (*stress*) und Belastung. Elastizität beschreibt die Eigenschaft eines Objekts in seine ursprüngliche Form zuzückzukehren. Spannung ist die Größe der zugefügten Kraft durch die Fläche, auf die sie wirkt und Belastung

ist die resultierende Deformation durch Spannung. Um diese Werte herauszufinden, bedient man sich verschiedener Modulus-Berechnungen. Für einen genaueren Blick auf Young's Modulus, *shearing modulus* (Scherungs Modulus) und *bulk modulus* (Massen Modulus) wird auf [Ebe04] verwiesen.

Außer den zusätzlichen Parametern braucht man auch noch eine geeignete Repräsentation der verformbaren Körper. Dazu gibt es verschiedene Ansätze. Eine Möglichkeit ist ein System aus durch Federn verbundene Punktmassen, also ein Massen-Feder- beziehungsweise Massen-Gesamtheits-System, welches in Abschnitt 2.1.2 bereits erklärt wurde. Eine andere Repräsentation kann durch *Control Point Deformation* (Kontrollpunkt Deformation) umgesetzt werden, bei der man die Objekte als parametrische Oberflächen mit Kontrollpunkten definiert. Diese Kontrollpunkte verändern die Fläche entsprechend der Krafteinwirkung und können mit *B-Splines* (Basis-Splines) oder *Nurbs Curves* (nicht-uniforme rationale B-Splines) realisiert werden. Bei *Free Form Deformation* (Freiform-Deformation) wird das *Mesh* (Gittergewebe) direkt entsprechend der auf den Körper wirkenden Kraft verändert. Eine letzte Variante ist die sogenannte *Implicit Surface Deformation* (Implizite Oberflächendeformation), wobei Objekte als Regionen für eine Funktion $F(x, y, z) \leq 0$ angesehen werden. Durch $F(x, y, z) = 0$ ist die Grundform der Oberfläche definiert. Eine auf den Körper wirkende Kraft wird durch Addition der Deformations-Funktion $D(x, y, z)$ dargestellt. Der deformierte Körper ist dann die mit $F(x, y, z) + D(x, y, z) \leq 0$ beschriebene Region, mit der durch $F(x, y, z) + D(x, y, z) = 0$ neu definierten Oberfläche. [Ebe04]

Die Geschwindigkeits- und Positions-Updates sowie die Grundlagen für Rotation bleiben die selben wie bei Rigid-Bodies. Auch eine Kollisionserkennung für starre Körper kann generell beibehalten werden. Lediglich die Kollisionsbehandlung muss auf durch Zusammenstöße resultierende Verformungen Rücksicht nehmen. Da Soft-Bodies in dieser Arbeit jedoch nicht weiter betrachtet werden, wird darauf nicht genauer eingegangen.

3 Kollisionserkennung

Wie aus dem vorangegangenen Kapitel ersichtlich, ist eine Kollisionserkennung beziehungsweise -behandlung kein zwingender Bestandteil einer Physik-Engine. Auch ohne die Möglichkeit von Objekt-Zusammenstößen lassen sich schon einige Effekte umsetzen. Es gibt jedoch nur wenige Engines, die darauf verzichten und jene, die es tun, sind meist auch nur für eine bestimmte Anwendung geschrieben.

Ein Kollisions-System für eine Engine besteht aus zwei Komponenten, einer Kollisionserkennung und einer Kollisionsbehandlung. Die Erkennung findet dabei Kontakte zwischen Objekten innerhalb einer Simulation und die Behandlung löst diese entsprechend wieder auf.

Im Folgenden wird zunächst auf Objektrepräsentationen eingegangen. Darauf folgt eine genaue Beschreibung des Vorgehens einer Kollisionserkennung und deren Möglichkeiten auf der GPU. Im Anschluss wird kurz auf die Aufgaben einer Kollisionsbehandlung eingegangen.

3.1 Objektrepräsentation

Wie bereits gesagt, ist die Hauptaufgabe einer Kollisionserkennung das Finden von Kontakten zwischen Körpern innerhalb einer Simulationsdomäne. Hierbei spielt die Objektrepräsentation eine wichtige Rolle. Bei Partikeln, die im 3D durch Kugeln dargestellt werden, ist das Aufspüren von Kollisionen recht einfach. Wie bereits in Kapitel 2.1.2 kurz beschrieben, existiert ein Kontakt zwischen zwei Partikeln, wenn die Distanz zwischen beiden kleiner gleich dem Partikeldurchmesser (2∗Partikelradius) ist. Rigid-Bodies können jedoch jede mögliche starre Form besitzen und sich innerhalb einer Anwendung voneinander unterscheiden, anders als bei einer Verwendung von Partikeln, bei der alle Teilchen gleich repräsentiert werden.

Grundsätzlich müssen alle Objekte innerhalb einer Simulation paarweise auf Kollision getestet werden. Dabei entstehen zwei Schlüsselprobleme, die behoben werden müssen. Zum einen sind bei einer Simulation mit vielen Objekten ebenso viele Kollisionen möglich und zum anderen können die Tests dazu je nach beteiligten Formen sehr rechenintensiv sein. Wie man die Anzahl der potentiellen Kontakte einschränken kann, wird später beschrieben, zuerst wird das Problem von aufwändigen Tests behoben. [Mil07]

An dieser Stelle kommt ein erster wichtiger Design-Faktor ins Spiel, die geometrische Repräsentation der simulierten Körper. Für physikalische Berechnungen sind die in der Computergraphik gängigen polygonalen, also auf vielen Dreiecken basierenden, Geometrien oft zu komplex. Eine Struktur, die für schnelles Rendern sorgt, ist nicht automatisch gut für eine schnelle Kollisionserkennung. Ein Test von zwei komplexen Geometrien auf gegenseitigen Kontakt kann sehr umfangreich werden. Dieses Problem lässt sich lösen, indem man die Meshes durch primitivere Formen, sogenannte *Boun-*

ding Volumes (Begrenzungsvolumen, kurz BV), annähert. Man unterscheidet also zwischen graphischer (Render-Geometrie) und physikalischer Darstellung (Kollisions-Geometrie) eines Objekts. Diese beiden Darstellungen sollten sich sehr ähneln, können sich in einigen Ausnahmen aber auch komplett unterscheiden. Bei einer Wassersimulation kann zum Beispiel die Kollision-Geometrie aus vielen tausend Partikeln bestehen, wobei die Render-Geometrie ein großes polygonales Mesh ist. Wichtig ist, dass die physikalischen Geometrien mit ihren Attributen während einer Simulation immer beachtet werden müssen, obwohl die graphische Darstellung vielleicht gar nicht gerendert wird. Dadurch wird die korrekte Beständigkeit einer Szene gesichert und merkwürdig wirkendes physikalisches Verhalten vermieden. [Eri05]

Generell ist ein Bounding-Volume ein Raumgebiet, in dem ein Objekt enthalten ist. Dieses Gebiet wird durch einfache Formen, wie Kugeln oder Boxen, definiert und sollte so eng wie möglich an seinem beinhaltenden Objekt anliegen. Eine Kugel wird durch ein Zentrum und einen Radius definiert. Bei Boxen muss man sich entscheiden, ob AABBs (*axis aligned bounding boxes*) oder OBBs (object bounding boxes) verwendet werden sollen. Diese beiden Herangehensweisen unterscheiden sich in der Verwendung des Koordinatensystems. Bei AABBs ist der Würfel nach den Welt-Koordinatenachsen ausgerichtet, wobei sich OBBs am lokalen Koordinatensystem eines Objekts orientieren (siehe Abbildung 7). Grundsätzlich lassen sich auch andere primitive Formen, wie Zylinder oder allgemein konvexe Polygone, verwenden, auf die an dieser Stelle jedoch nicht weiter eingegangen wird. [Mil07]

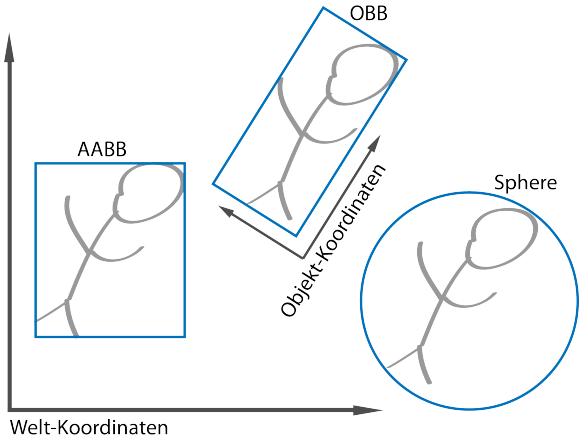


Abbildung 7: Sphere vs. AABB vs. OBB
Sphere (Kugel) in Welt-Koordinaten
AABB orientiert an Welt-Koordinaten
OBB orientiert an Objekt-Koordinaten

Die Vorteile der Aufteilung in Render- und Kollisions-Geometrie überwiegen, aber es sind auch einige Nachteile möglich. Doppelte Daten, höherer

Arbeitsaufwand bei der Modellerstellung oder Fehlpaarungen zwischen den beiden Geometrien sind einige davon. [Mil07]

Rechenintensive Tests lassen sich durch BVs schon einmal minimieren, jedoch muss noch immer jedes Objekt mit jedem getestet werden. Um dieses Problem der vielen potentiellen Kollisionen zu lösen, unterteilt man die Kollisionserkennung in zwei Phasen, die *broad Phase* (weite/breite Phase) und *narrow Phase* (engen Phase). Diese Aufteilung wird im folgenden Abschnitt genauer beschrieben. [Mil07]

3.2 Phasen

Innerhalb der Physik-Engine werden Objekte nun nur hinsichtlich ihrer Kollisions-Geometrien beziehungsweise BVs betrachtet. Diese Geometrien werden dann anhand ihrer räumlichen Aufenthaltsorte sortiert und strukturiert, sodass daraufhin nur relativ nah beieinander liegende Objekte gegenseitig auf Kontakt getestet werden müssen.

3.2.1 Broad-Phase

Das Sortieren und Strukturieren der Körper in einer Szene geschieht in der Broad-Phase. Eine grobe räumliche Einschränkung erfolgt durch Hilfe einer bestimmten Datenstruktur, wie Bäume oder Gitter.

Eine Bounding-Volume-Hierarchy (kurz BVH) ordnet BVs in eine Baumstruktur ein. Dabei sind die einzelnen Objekte als unterste Blatt-Knoten definiert, die bis hoch zur Wurzel, Schritt für Schritt, anhand ihrer räumlichen Position in immer größer werdende Gruppen zusammengefasst werden. Der Wurzel-Knoten ist schließlich ein Bounding-Volumen, das die gesamte Szene enthält. Die Volumen der BVs sollten auch hier pro Stufe im Baum möglichst klein gehalten werden und sich so wenig wie möglich überlappen. Performanz-Vorteile können außerdem geschaffen werden, indem man eine ausbalanzierte Struktur verwendet, wie beispielsweise einen binären Baum, also festlegt, dass pro Elternknoten nur zwei Kinderknoten existieren dürfen. Ist diese Struktur aufgebaut, werden von der Wurzel in Richtung Blätter nach und nach die zwischenliegenden Knoten betrachtet. Wenn sich die Bounding-Volumes von zwei Elternknoten nicht berühren, so tun dies auch deren Kinder nicht. Existiert hingegen ein Kontakt, so müssen die Kinder eine Ebene tiefer im Baum gegeneinander geprüft werden.

Viele Render-Systeme oder Game-Engine verwenden bereits Hierarchien für Bounding-Volume-Tests mit der Kamera, um herauszufinden, welche Objekte gesehen, also auch gerendert werden müssen. Wenn solch eine Baumstruktur bereits besteht, kann diese natürlich auch für die physikalische Darstellung genutzt werden. Beim Konstruieren einer solchen Hierarchie ist jedoch Geschwindigkeit gegen Qualität abzuwägen. Bei einer Welt aus vielen statischen, also nicht beweglichen Objekten kann die Struktur offline im Vor-

aus beziehungsweise beim Laden der Szene erstellt werden. Handelt es sich jedoch um eine dynamische Welt mit vielen beweglichen Objekten muss der Baum in Echtzeit immer wieder aktualisiert werden. Dies kann durch verschiedene Strategien bewerkstelligt werden. Diese sind Bottom-up, Top-down oder Insertion. [Mil07] Für einen genaueren Einblick in die Build-Strategien und Knoten-Abarbeitung-Techniken von BVHs wird auf [Eri05] verwiesen.

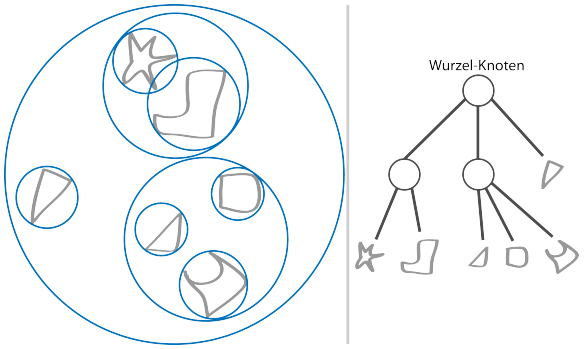


Abbildung 8: *Bounding Volume Hierarchy*
Strukturierung einer Szene durch eine BVH

Bei komplexen Formen, bei denen einfache BVs zu viel leeren Raum erzeugen würden, kann man auch sogenannte Sub-Objekt-Hierarchien verwenden, wobei ein einzelnes Objekt in mehrere BVs unterteilt wird. Es existiert dann eine Hierarchie eines einzelnen Körpers in einer Hierarchie der gesamten Szene. [Mil07]

Eine weitere verwendbare Struktur sind *binary space-partitioning trees* (binäre Raum-Partitionierungs-Bäume, kurz BSPs). Sie gehören zur Klasse der *Spatial Data Structures* (Räumliche Datenstrukturen, kurz SDSs). Mit ihnen wird ein Raum durch eine Trennebene in zwei Teilräume unterteilt. Beginnend beim Wurzel-Knoten, der auch hier die gesamte Szene beinhaltet, wird die Welt rekursiv in immer kleiner werdende Teilbereiche aufgeteilt. Die Trennebenen werden dabei als Vektoren für eine Position irgendwo auf der Ebene und eine Richtung, senkrecht stehend auf einer Seite der Ebene, gespeichert. Um herauszufinden, auf welcher Seite ein Objekt liegt, wird das Skalarprodukt zwischen Richtungsvektor und Abstand des Körpers zur Ebene genutzt (siehe Gleichung 24). [Mil07]

$$c = (\boldsymbol{p}_o - \boldsymbol{p}_p) * \boldsymbol{d}_p \tag{24}$$

$\boldsymbol{p}_o$ *Position der Objekts*
$\boldsymbol{p}_p$ *Position auf der Trennebene*
$\boldsymbol{d}_p$ *Richtungsvektor der Trennebene*

Wenn c positiv ist, dann liegt das Objekt auf der Seite der Trennebene, in die der Richtungsvektor zeigt. Bei negativem c entsprechend auf der anderen Seite und bei $c = 0$ liegt ein Körper direkt auf der Ebene. [Mil07]

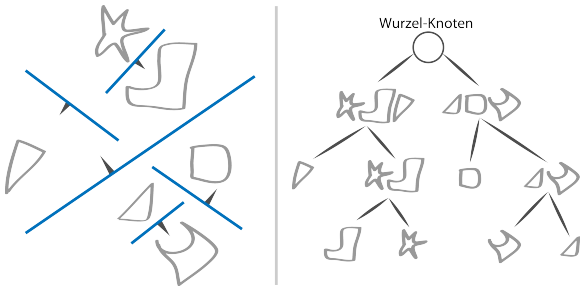


Abbildung 9: *Binary Space Partitioning*
Strukturierung einer Szene durch BSP

Die Objekte werden nicht wie bei BVHs in neu definierten Volumen zusammengefasst, sondern nur anhand ihrer relativen Position und Größe einem Teilraum beziehungsweise mehreren unterschiedlich großen Teilräumen, innerhalb der Szene zugeordnet. Dabei kann jedes Blatt beliebig viele Körper enthalten und nur diese müssen dann genauer auf Kollisionen betrachtet werden.

Die Struktur speichert also Objekte nicht direkt, wie es bei BVHs der Fall ist, sondern nur deren Aufenthaltsorte in den entsprechenden Teilräumen. BVHs klassifizieren Raumregionen um Körper herum, wohingegen SDSs Körper in Raumregionen klassifizieren (hierarchische Objekt-Repräsentation vs. hierarchische Raum-Repräsentation). Spatial Data Structures lassen sich aus diesem Grund einfacher erstellen als BVHs. Bewegt sich ein Körper, muss bei einer BVH die Struktur aktualisiert werden, und zwar alle BVs in denen der Körper lag und jetzt liegt. Bei BSP hingegen bleibt die Struktur, also die einzelnen Teilräume, immer gleich. Sich bewegende Objekte müssen nur entsprechend dem aktuellen Bereich zugeordnet werden. [Mil07]

Auch hier wird für einen genaueren Einblick auf [Eri05] verwiesen.

Eine Alternative zu Baumstrukturen sind Gitter, auch sie gehören zur Gruppe der SDSs. Die einfachste Form sind *uniform grids* (einheitliche Gitter), sie unterteilen eine Szene in eine bestimmte Anzahl gleich große Zellen, anstatt in rekursiv immer kleiner werdende Teilräume. Objekte werden dann mit denjenigen Gitterzellen assoziiert, die sie überlappen. Nur Körper innerhalb einer beziehungsweise benachbarter Gitterzellen müssen dann genauer auf Kollision getestet werden. Bei der Wahl der Zellengröße sind einige Kriterien zu beachten, die hauptsächlich von den Objektgrößen abhängen. Sind die Zellen zu fein, so überlappt ein großer Körper zu viele Zellen. Sind sie zu grob, so beinhalten sie möglicherweise zu viele kleine Objekte und es sind weiterhin viele genaue Tests nötig. Generell ist die optimale Zellengröße ge-

geben, wenn das größte Objekt in einer Szene in allen möglichen Rotationen in eine Zelle passt. Denn ein Körper sollte so wenig Zellen wie nötig überlappen, um möglichst wenige genauere Kollisionsprüfungen zu generieren. In diesem Fall überlappt ein Objekt maximal 4 Zellen. [Eri05]

Innerhalb des Codes werden bei Gittern keine Knoten-Strukturen verwendet, sondern auf Arrays und oder *Linked Lists* (zusammenhängende Listen) zurückgegriffen. So wird das Gitter selbst als ein- oder mehrdimensionales Array repräsentiert und Objekte innerhalb einer Zelle in Linked Lists gespeichert. Dies kann Zugriffe auf Objektdaten vereinfachen, aber auch mehr Platz im Speicher beanspruchen. [Eri05]

Variationen von Uniform Grids sind *Infinite Grids* (unendliche Gitter) und *Implicit Grids* (implizite Gitter). Unendliche Gitter nehmen sich Hash-Tabellen zur Hilfe. Wobei jede Zelle in ein Set von *n buckets* (Eimer) sortiert wird, diese Eimer enthalten dann die Linked Lists mit den Objekten. Der benötigte Speicher ist dabei von der Objektanzahl abhängig und nicht von der Zellengröße. Implizite Gitter besitzen ein Array pro Gitter-Spalte, -Reihe und -Zeile. Sie verwenden das Kartesische Produkt (Kreuzprodukt) dieser 3 Arrays um herauszufinden, ob sich ein Objekt in einer entsprechenden Zelle befindet oder nicht. [Eri05]

Ein Nachteil von einheitlichen Gittern ist, dass sie mit erheblichen Objektgrößenunterschieden nicht sehr gut zurechtkommen. Dabei helfen hierarchische Gitter, die sich aus mehreren uniformen Gittern unterschiedlicher Zellengrößen zusammensetzen, die alle überlappend über die selbe Szene gelegt werden. Körper werden dann gemäß ihrer Größe in das entsprechende Gitter, beziehungsweise in eine Zelle dieses Gitters einsortiert. Bei den genauen Tests auf Kollisionen müssen so aber auch alle möglichen Nachbarn aus den anderen Gittern betrachtet werden. Diese Struktur belegt mit ihren Arrays viel Speicherplatz und sollte am besten mit *hashed storage* (Hash-Speicherung) umgesetzt werden.[Eri05]

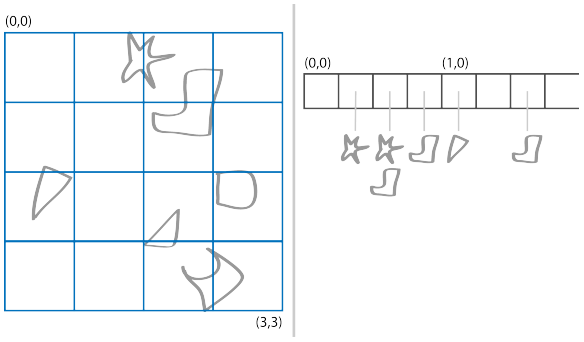


Abbildung 10: *Uniform Grids*
Strukturierung einer Szene durch einheitliche Gitter

Um diesen Speicherplatz zu minimieren, können rekursive Gitter verwendet werden. Sie legen nicht mehr alle Gitter-Level über die gesamte Welt, sondern verwenden ein einheitliches Gitter, für dessen Zellen rekursiv ein feineres Gitter erzeugt wird, sobald sie mehr als k Objekte enthalten. [Eri05]

Auch für Gitter kann [Eri05] für einen genaueren Einblick heran gezogen werden.

Eine letzte Datenstruktur vereint das Prinzip von Bäumen und Gittern. Octrees (3D) sind architypische baum-basierte Raumaufteilungen. Jeder Knoten hat dabei ein endliches würfelförmiges Volumen, wobei die Wurzel wieder die gesamte Szene beinhaltet. Jeder Eltern-Knoten besitzt 8 gleich große Kind-Knoten. Deren rekursive Aufteilung endet, wenn eine gegebene maximale Tiefe im Baum erreicht ist oder sobald die Würfel kleiner als ein bestimmtes Volumen werden. Ein Quadtree ist die analoge Struktur im 2-dimensionalen Raum mit 4 quadratischen Flächen als Kinder pro Elternknoten. [Eri05]

Der Verweis auf [Eri05] für einen genaueren Einblick gilt auch an dieser Stelle.

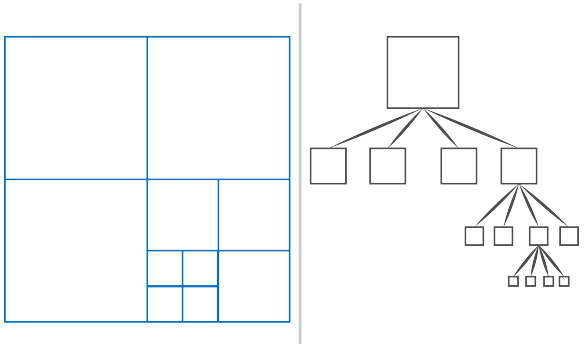


Abbildung 11: *Quadtrees*
Strukturierung einer Szene durch einen Quadtree (2D)

Nach dem Aufbau einer Datenstruktur, in die alle Objekte der Szene einsortiert und gegliedert wurden, ist die erste Phase der Kollisionserkennung abgeschlossen. Eine genaue Prüfung auf Kollision hat bis dahin noch nicht stattgefunden. Die Aufgabe der broad Phase ist lediglich das Einschränken dieser Tests, die dann in der nächsten Phase angegangen werden.

3.2.2 Narrow-Phase

In der Narrow-Phase findet dann der eigentliche Prozess der Kollisionserkennung statt. Durch *basic primitive tests* (grundlegende Primitiven-Tests) werden die Nachbarn der zuvor erstellten Hierarchie auf Kontakte geprüft. Diese Tests richten sich nach den vorliegenden Kollisions-Geometrien. Gene-

rell sind aber nur folgende Kontakt-Typen möglich:

point-point	Kontakt zw. zwei Punkten
point-edge	Kontakt zw. Punkt und Kante
point-face	Kontakt zw. Punkte und Fläche
edge-edge	Kontakt zw. zwei Kanten
edge-face	Kontakt zw. Kante und Fläche
face-face	Kontakt zw. zwei Flächen

Dabei kann man point-point und point-edge Kontakte ignorieren, da sie selten vorkommen, einfach zu handhaben sind und auch durch andere Typen repräsentiert werden können. Face-face Kontakte kommen auch nur sehr selten vor, nämlich nur wenn eine der beiden Flächen gebogen ist. Weiterhin können edge-face Kontakte oft auch durch ein Paar von edge-edge Tests erkannt werden. Aus diesen Einschränkungen folgt, dass grundsätzlich zuerst auf point-face und edge-edge Kontakte getestet wird. Diese Typen kommen während einer Simulation auch am häufigsten vor. [Mil07]

Um nun eine, beziehungsweise alle Kollisionen zwischen zwei Objekten herauszufinden, werden eine Kontakt-Normale (englisch *contact-normal*) und bei sich überlappenden Körpern, eine Überlappungstiefe (englisch *penetration-depth*) benötigt. Die Kontakt-Normale gibt dabei die Richtung des Impulses, der beim Aufprall wirkt, an. Die penetration-depth wird entlang der contact-normal gemessen. Sie wird miteinbezogen, da, je nach Dauer eines Simulationsschrittes und Objektgeschwindigkeiten, Körper bereits ineinander liegen können, bevor die Kollision erkannt werden konnte. Diese Daten werden relativ zu einem der beiden beteiligten Objekte betrachtet. [Mil07]

Beim point-face Typ ist die Normale durch die Oberflächennormale des *faces* (der Fläche) gegeben. Bei einer Überlappung wird der Kontakt-Punkt einfach entlang der Normalen auf die Oberfläche zurück bewegt. Die Kontakt-Normale bei einer edge-edge Kollision liegt im rechten Winkel zu den Tangenten beider Kanten und kann per Vektorprodukt bestimmt werden. Ein contact-point ist dann der in kürzester Reichweite liegende Punkt zwischen den Kanten und die Überlappungstiefe ist durch den Abstand der beiden gegeben. [Mil07]

Im Folgenden werden einige Kontakt-Fälle zwischen bestimmten primitiven Formen vorgestellt.

Kugel-Kugel (*sphere-sphere*):

Eine Kollision existiert, wenn die Distanz zwischen den beiden Mittelpunkten kleiner als die Summe der beiden Radien ist. Es ist ein face-face Kontakt mit zwei gebogenen Flächen, wobei nur ein Kontakt-Punkt möglich ist und die Überlappungstiefe auf der Linie zwischen den beiden Zentren der Kugeln liegt. [Mil07]

Kugel-Ebene (*sphere-plane*):

Eine Kollision liegt vor, wenn die Distanz zwischen Kugelmittelpunkt und Ebene kleiner als der Radius der Kugel ist. [Mil07]

$$\boldsymbol{d} = \boldsymbol{p} * \boldsymbol{n} - \boldsymbol{l} \tag{25}$$

$\boldsymbol{d}$ *Distanz*
$\boldsymbol{p}$ *Position des Kugelursprungs*
$\boldsymbol{n}$ *Normale der Ebene*
$\boldsymbol{l}$ *Offset der Ebene*

Kugel-Würfel (*sphere-box*):

Zwischen Kugel und Würfel sind drei Kontakt-Typen möglich (face-face, edge-face oder point-face), bei denen die Berechnungen für Kontakt-Normale und Überlappungstiefe die selben bleiben. Hier reicht nun keine einfache Distanz mehr aus, um eine Kollision zu bestätigen. Man bedient sich bei sogenannten *Separating Axis* (Trenn-Achsen) die als Early-Out Test (frühes Ausstiegskriterium) genutzt werden. Dazu wird zuerst der Kugelmittelpunkt in Objektkoordinaten der Box konvertiert. Wenn man jede Richtung im Raum finden kann, in der sich die beiden Objekte nicht berühren, dann kollidieren sie überhaupt nicht. Dafür wird jede Achse des Würfels mit dem Kugelmittelpunkt geprüft: [Mil07]

$$h_b + r > p_i \tag{26}$$

h_b *halbe Seitenlänge der Box*
r *Radius der Kugel*
p_i *relative Position des Kugelzentrums*
mit $i = x-, y-$ *oder* $z-$ *Achse*

Wird jedoch eine Richtung gefunden, in der sich die Körper überlappen, so muss ein Kontakt-Punkt gefunden werden. Dies geschieht, indem man den nächsten Punkt innerhalb der Box zum gefundenen Überlappungspunkt bestimmt. [Mil07]

Würfel-Würfel (*box-box*):

Eine Box-Box Kollision ist der komplexeste Fall, dabei können alle 6 Kontakt-Typen auftreten. Auch hier werden Separating Axis als Ausstiegskriterium genutzt. Einfach betrachtet, kann man die Tests jedoch auf point-face und edge-edge beschränken. Um auf Punkt-Fläche-Kontakte zu prüfen, muss jedes *Vertex* (Eckpunkt) von jeder Box mit dem jeweils anderen Würfel getestet werden. Dazu werden sie auf die Hauptachsen der jeweils anderen Box projiziert und die selben Tests wie bei sphere-box, nur mit dem Radius= 0, durchgeführt. [Mil07]

Ein genaueres Vorgehen für Box-Box Kollisionen und auch generell weitere basic primitive tests werden in [Mil07] sowie [Eri05] vorgestellt.

Für eine Kollisionserkennung von allgemeinen Polyedern, also beliebig konvexer Objekte, sind **V-Clip** (Voronoi-Clip Algorithmus) und **GJK** (Gilbert-Johnson-Keerthi Distanz Algorithmus) besonders geeignete Algorithmen. [Mil07]

Ist ein Kontakt-Punkt gefunden, können weitere Daten zur Kollision gesammelt und in einer Struktur untergebracht werden. So kann man einen Kontakt im Code zum Beispiel als Klasse oder *Struct* realisieren, in der die beteiligten Objekte, Kontakt-Normale, Überlappungstiefe und Schließgeschwindigkeit gespeichert werden. [Mil07]

Nachdem alle Kollisionen zwischen Objekten innerhalb der Szene gefunden wurden, können deren Daten, zum Beispiel in Form einer Liste, an die Kollisionsbehandlung weiter gegeben werden.

3.3 Kollisionserkennung auf der GPU

Auf der Grafikkarte bestehen einige Möglichkeiten, um eine Kollisionserkennung zu beschleunigen oder gar ganz darauf umzusetzen.

Eine spezielle Methode, Kontakte zu erkennen, ist zum Beispiel *fast image-space-based intersection* (schnelle Bild-Raum-basierte Überschneidung). Sie wird innerhalb der Render-Pipeline angesetzt, wenn die zu zeichnenden Objekte in *Color-*, *Depth-* und *Stencil-Buffer* (Farb-, Tiefen- und Schablonen-Puffer) rasterisiert werden. Der entsprechende Algorithmus ist zwar einfach zu implementieren und braucht keine komplexe Datenstruktur, da auf rasterisierbaren Primitiven gearbeitet wird. Bei weit entfernten Objekten, die nur mit wenigen Pixeln dargestellt werden, kann es jedoch zu Problemen kommen. Außerdem werden Körper außerhalb des Sichtfeldes ignoriert, was bei einer Sichtänderung merkwürdig wirkendes physikalisches Verhalten verursachen könnte. [Eri05]

Die GPU kann auch als Co-Prozessor verwendet werden, auf dem mathematische Berechnungen ausgelagert und durch mögliche Parallelisierung beschleunigt werden können. Dazu sollten jedoch alle benötigten Werte im Speicher der Grafikkarte vorliegen, um teure Zugriffe zwischen CPU und GPU zu vermeiden. Das Speichern von Objekt-Attributen in Texturen ist eine effiziente Möglichkeit. [Eri05]

Der größte Vorteil gegenüber der CPU ist die schnelle Bearbeitung parallelisierbarer Prozesse. Indem man in allen Komponenten einer Physik-Engine Strukturen verwendet, die gut auf die GPU zuschneidbar sind, kann man diesen Vorteil auch für eine Kollisionserkennung nutzen. Die beste Objektrepräsentation sind dabei Partikel. Ihre Kollisions-Prüfungen gegeneinander sind so einfach wie schnell und lassen sich sehr gut parallel ausführen. Anstatt nach und nach jedes Teilchen gegen jedes andere in der Szene zu

testen, können alle Teilchen gleichzeitig gegen die anderen gepüft werden. Somit sind Partikel in Hinsicht auf GPU-Umsetzung eine gute Wahl für die Narrow-Phase. Für eine Broad-Phase auf der GPU sind uniforme Gitter am besten geeignet. Sie lassen sich gut konstruieren und parallel befüllen oder aktualisieren.

Das Zusammenspiel dieser Komponenten wird in Kapitel 5 genauer beschrieben denn in der arbeitszugehörigen Implementation werden genau diese Strukturen genutzt.

3.4 Kollisionsbehandlung

Die in der Kollisionserkennung gefundenen Kontakte zwischen Objekten müssen nun aufgelöst werden. Dazu werden die aus dem Zusammenstoß resultierenden Impulse benötigt. Genau genommen, der Impuls und das Drehmoment von jedem beteiligten Körper, also insgesamt 4 Werte. Diese Werte lassen sich durch folgende 6 Schritte ermitteln: [Mil07].

1. Änderung in Kontakt-Koordinaten. Kollisionen relativ zu beteiligten Objekten zu betrachten, vereinfacht viele Berechnungen. Dazu wird eine Transform-Matrix bestimmt, um zwischen verschiedenen Koordinatensystemen konvertieren zu können.

2. Schließgeschwindigkeit am Kontakt-Punkt ermitteln. Durch Formel 11 aus Kapitel 2.1.2.

3. Abstoßgeschwindigkeit aus Restitutionskoeffizient und Schließgeschwindigkeit berechnen.

4. Benötigte Geschwindigkeitsänderung aus Differenz von Schließ- und Abstoß-Geschwindigkeit bestimmen.

5. Aus benötigter Geschwindigkeitsänderung die dazu nötigen Impulse berechnen.

6. Impulse in lineare und angulare Komponente aufteilen und auf jedes Objekt anwenden.

Danach können die Integrationen zur Positionsänderung der Körper durchgeführt werden.

Bevor die Impulse bestimmt werden können, müssen jedoch noch vorliegende Überlappungen beseitigt werden. Die einfachste Möglichkeit ist, eine Überlappung per linearer Projektion zu lösen, indem man die Objekte entlang der Kontakt-Normale wieder auseinander bewegt. Die Bewegung jedes Objekts ist dabei proportional zu seiner inversen Masse. Bei Rigid-Bodies

kann diese Methode jedoch seltsam wirkendes Verhalten hervorrufen, da eine mögliche Rotation nicht beachtet wird. Nonlineare Projektion kombiniert lineare und angulare Bewegung. Die Objekte werden weiterhin entlang der Kontakt-Normalen auseinander bewegt. Dabei muss für jeden Körper die Menge an linearer und angularer Bewegung bestimmt werden. Die Balance zwischen linearer und angularer Geschwindigkeit hängt vom inversen Inertia-Tensor der beiden Objekte ab. Körper mit einem hohen Trägheitsmoment werden weniger rotiert und mehr in linearer Richtung bewegt. Dieses Verfahren ist immer noch nicht ganz realistisch, erzeugt aber glaubwürdige und selten ungewöhnliche Bewegungen. Bei Überlappungen ist darauf zu achten, dass durch deren Auflösen neue Kontakte generiert werden können. [Mil07]

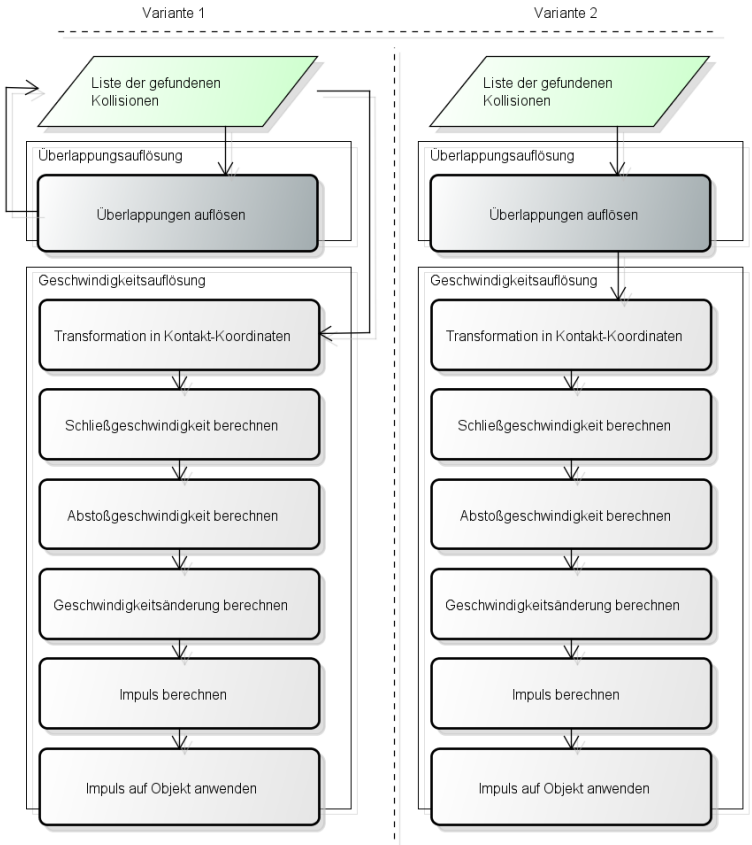


Abbildung 12: Kollisionsbehandlung
Variante 1: erst Überlappungen aller Kontakte lösen, dann Impulse berechnen (2-mal über Liste iterieren)
Variante 2: Überlappung eines einzelnen Kontakts lösen und direkt Impuls berechnen (1-mal über Liste iterieren)

Der gesamte Ablauf einer Kollisionsbehandlung besteht also aus 2 Komponenten, dem Überlappungs- und Geschwindigkeits-Auflösungs-System, die

unabhängig von einander agieren können. Als Eingabe bekommen sie die Liste der durch die Kollisionserkennung gefundenen Kontakte. Nun können zuerst interne Daten der Kollisionen, wie Transform-Matrix und Schließgeschwindigkeit, schon einmal vorbereitet werden. Die einzelnen Kontakte werden dann nach und nach abgearbeitet, wobei es von Vorteil sein kann, sie nach ihren Schließgeschwindigkeiten zu sortieren und mit der schnellsten zu beginnen. Dann werden die vorliegenden Überlappungen beseitigt, um danach die Geschwindigkeits-Auflösung durchzuführen. [Mil07]

Das Abarbeiten der Liste von vorliegenden Kollisionen kann auf der GPU beschleunigt werden, indem alle Einträge parallel aufgelöst werden können. In einigen Fällen bietet es sich auch an, nicht erst alle Zusammenstöße zu suchen und dann zur Behandlung weiterzugeben, sondern einen gefundenen Kontakt direkt aufzulösen.

4 CUDA

Abbildung 13: Nvidia CUDA [Geo13]

CUDA ist eine von Nvidia entwickelte Architektur, um deren Grafikkarten programmieren zu können. Die Syntax basiert auf C/C++ und die erste Version erschien 2007. Momentan ist Version 6.5 die stabil laufende, die im August 2014 veröffentlicht wurde.

Durch die Basis von C/C++ können auch Funktionen auf der CPU definiert werden. Die Dateiendungen für CUDA-Dateien sind ".cu" und ".cuh", analog zu ".cpp" und ".h" bei C++. [Kre14]

In den folgenden Abschnitten wird zuerst genauer auf einige Hardwarekomponenten, die Bestandteil einer Grafikkarte sind, eingegangen. Dann wird die grundlegende Terminologie von CUDA vorgestellt und diese zum Schluss anhand eines kleinen Beispiels veranschaulicht.

4.1 GPU-Hardware

Abbildung 14: Kepler-Architektur (1) [NVI12]

Die Hardware einer Grafikkarte besteht wie die CPU aus Prozessoren und Speichern. Anders als bei der CPU, besitzt eine GPU jedoch nicht nur einen beziehungsweise eine sehr überschaubare Anzahl an Prozessoren, sondern sehr viele mehr. Genau genommen werden viele einzelne Prozessoreinheiten zu mehreren größeren Einheit zusammengefasst, sogenannten Multiprozessoren. Außerdem kommen verschiedene Speicher-Typen hinzu. Die erste Hardware-Struktur, die erweiterte Programmierbarkeit bot, war 1999 die Tesla-Architektur. 2009 erfolgte eine Weiterentwicklung zur Fermi-Architektur und 2012 entsprechend eine Entwicklung zur Kepler-Architektur. [MG14]

Bei der aktuellen Kepler-Struktur wird ein neues Fertigungsverfahren verwendet, 28nm im Gegensatz zu 40nm bei den alten Architekturen, wobei durch die angegebene Zahl auf die Leistungsfähigkeit pro Fläche geschlossen werden kann. Die Prozessoren werden Streaming-Multiprozessoren (kurz SMX) genannt und bestehen aus 192 *single precision cores* (Kerne mit einfacher Genauigkeit). Weiterhin existieren *double precision cores* (Kerne mit doppelter Genauigkeit) und *special function units* (Spezial-Funktionseinheiten) zur Berechnung transzendentaler Funktionen und Attributinterpolation. [MG14]

Die Speicherhierarchie ist durch den in konstanten, globalen und lokalen Speicher unterteilte DRAM (*Dynamic Random Access Memory*) gegeben. Auf den globalen und konstanten Speicher kann auch die CPU zugreifen, sie kopiert alle benötigten Daten in global-Memory, ruft einen Kernel auf und kopiert die Ergebnis-Daten wieder zurück in ihren Arbeitsspeicher. Der konstant-Memory wird für Texturen und andere konstante Daten verwendet. Jeder SMX besitzt einen 48KB *read-only Cache* (Speicher auf den nur Lesezugriff möglich ist) und einen 64KB großen Speicher, der in L1-Cache und *shared-Memory* (gemeinsamer Speicher) aufgeteilt werden kann. Durch den shared-Memory ist der Datenaustausch zwischen Threads innerhalb eines Warps gegeben. [MG14]

Im Anschluss folgt eine Auflistung der verschiedenen Speicher-Typen: [Kre14]

Register Memory:
Ein Register ist ein schneller Speicher für lokale Variable, auf die nur der ausführende Thread Zugriff hat. Die Anzahl der verfügbaren Register pro Thread hängt von der Blockaufteilung ab und die Lebensdauer der Daten endet mit Abbruch eines Threads.

Local Memory:
Der lokale Speicher wird genutzt, um private Variablen eines Threads auf-
zufangen, falls diese zu groß für ein Register sind.

Shared Memory:
Ein gemeinsamer Speicher existiert für jeden Block, auf ihn können alle

Threads eines Blocks zugreifen und so untereinander Daten austauschen.

Global Memory:
Der globale Speicher dient als Verbindung zwischen CPU und GPU und Zugriffe sind entsprechend langsam. Alle Threads eines Kernels haben auf ihn Zugriff und seine Lebensdauer ist gleichgesetzt mit der des gesamten CUDA-Programms.

Constant Memory/Texture Memory:
Der konstante Speicher wird für Variablen beziehungsweise Texturen genutzt, die ein Programm nur auslesen jedoch nicht verändern kann.

Neu in der Kepler-Architektur ist auch, dass auf der GPU laufende Kernel dynamisch neue initiieren können. Dadurch können komplexe Programme unabhängig von der CPU ausgeführt werden. Dieses Vorgehen wird *Dynamic Parallelism* genannt. [MG14]

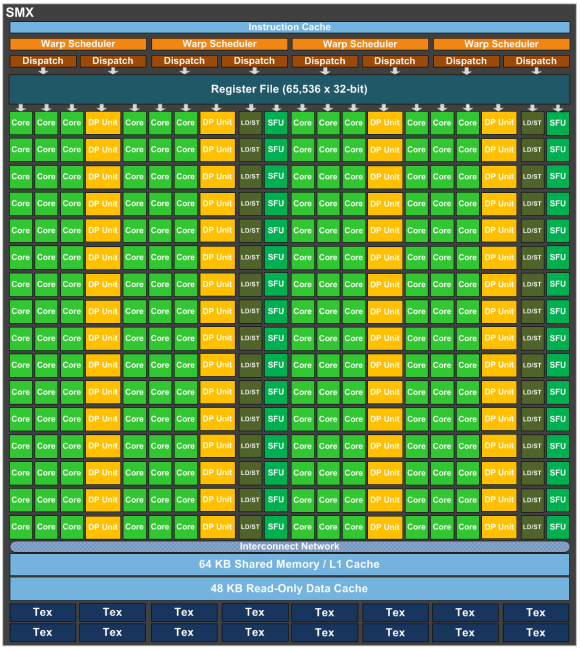


Abbildung 15: Kepler-Architektur (2) [NVI12]

Die neueste Architektur von Nvidia ist Maxwell. Die ersten Grafikkarten, die auf ihr basieren, erschienen 2014. In ihr wurden die Streaming-Multiprozessoren weiterentwickelt, indem die Kernanzahl auf einen auf Zweierpotenzen basierenden Aufbau reduziert wurde. [MG14]

Für einen genaueren Blick auf die unterschiedlichen Nvidia-Architekturen, wird auf [MG14] verwiesen.

4.2 Terminologie

In CUDA wird die CPU als *host* (Gastgeber) und GPUs als *device* (Gerät) bezeichnet. Bei Computern mit mehreren Grafikkarten sind diese durch eine *device id* unterscheidbar. Vergleichbar zu Methoden auf der CPU, sind *kernels* (Kernel) Funktionen, die auf der GPU ausgeführt werden. Der Unterschied liegt darin, dass Methoden nur einmal aufgerufen werden und Kernels von mehreren *Threads* (Themen) gleichzeitig ausgeführt werden. 32 Threads bilden sogenannte *Warps* (Ketten). Mehr als 32 Threads werden in Blöcken zusammengefasst, die wiederum in einem *Grid* (Gitter) strukturiert sind. Diese Einteilung ist der Hardware verschuldet (siehe Abbildung 16) und kann in jeder Ebene bis zu 3 Dimensionen besitzen. Durch die Definition dieser Dimensionen können Kernels gezielt für eine gewünschte Anzahl an Threads gestartet werden. [Kre14]

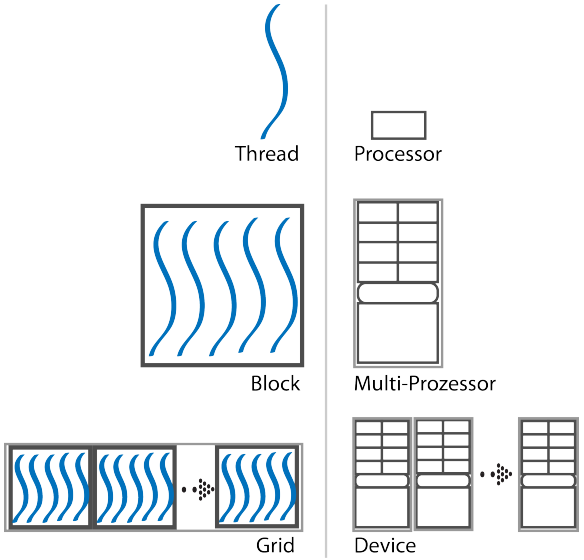


Abbildung 16: Thread/Block-Einteilung
Hardware- vs. Software-Sicht

Bei der Definition einer Funktion muss außerdem ein entsprechender *Function Type Classifier* (Funktion-Typ-Klassifizierung) gewählt werden. Dieser ist abhängig vom Aufrufs- und Ausführungsort der Funktion. Folgende Klassifizierungen sind möglich: [Kre14]

Schlüsselwort	**Erklärung**
__host__	Funktion auf CPU aufgerufen, auf CPU ausgeführt
__global__	Funktion auf CPU aufgerufen, auf GPU ausgeführt
__device__	Funktion auf GPU aufgerufen, auf GPU ausgeführt

Die verschiedenen Bereiche des DRAM, deren Aufteilung schon kurz im vorangehenden Kapitel genannt wurde, können ebenfalls gezielt angesprochen werden. Um Variablen in einem gewünschten Speicher unterzubringen, stehen die nachfolgenden Schlüsselwörter zur Verfügung: [Kre14]

Schlüsselwort	Erklärung
__shared__	Variable im shared-Memory eines Thread-Blocks unterbringen
__constant__	Variable im konstanten Speicher unterbringen

Eine CUDA-Implementation sollte die Multiprozessoren und Speichertypen so auslasten, dass deren optimale Leistungsfähigkeit abgerufen wird. Dazu können einige Aspekte beachtet werden: [Kre14]

- möglichst viele Threads auslasten und auf ihnen die gleiche Logik anwenden
- Abhängigkeiten zwischen Threads vermeiden
- Datenaustausch zwischen CPU und GPU minimal halten
- verschiedene GPU-Speicher sinnvoll nutzen und Speicherzugriffe optimieren

4.3 Programm-Beispiel

Nun wird der generelle Aufbau und Ablauf eines CUDA-Programms anhand des Beispiels einer Vektoraddition vorgestellt. Dabei sollen zwei beliebig große Vektoren elementweise addiert werden und das Ergebnis in einem dritten Vektor gespeichert werden. Der dazu nötige Kernel sieht folgendermaßen aus:

```
1 __global__ void vectorAdd(float *A,float *B,float *C,int n){
2         //get thread id
3         int i = blockDim.x * blockIdx.x + threadIdx.x;
4
5         //addition
6         if (i < n){
7                 C[i] = A[i] + B[i];
8         }
9 }
```

Listing 1: vectorAdd (1)

`__global__` gibt an, dass der Kernel von der CPU aus aufgerufen und dann auf der GPU ausgeführt wird. Als Eingabe werden zwei zu addierende Vektoren (A und B), ein Ergebnisvektor (C) und die Anzahl der beinhalteten Elemente (n) erwartet.

Auf der CPU werden nun die Arrays, die die Vektoren repräsentieren anhand einer definierten Elementanzahl zugewiesen und beliebig gefüllt:

```
1         float *h_A = (float *)malloc(size);
2         float *h_B = (float *)malloc(size);
3         float *h_C = (float *)malloc(size);
```

Listing 2: vectorAdd (2)

Ebenso werden Pointer und Platz für Arrays im GPU-Speicher vergeben:

```
1         //allocate device arrays
2         float *d_A = NULL;
3         float *d_B = NULL;
4         float *d_C = NULL;
5
6         cudaMalloc((void **)&d_A, size);
7         cudaMalloc((void **)&d_B, size);
8         cudaMalloc((void **)&d_C, size);
```

Listing 3: vectorAdd (3)

Dann werden die Daten von der CPU an die Grafikkarte übertragen:

```
1         //copy host data to device arrays
2         cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
3         cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

Listing 4: vectorAdd (4)

Nun befinden sich alle benötigten Daten auf der GPU und die Berechnung kann durchgeführt werden. Dazu werden zuerst die Thread- beziehungsweise Block-Dimensionen bestimmt und mit diesen der Kernel gestartet:

```
1 //define threads/blocks
2 int threadsPerBlock = 256;
3 int blocksPerGrid=(n + threadsPerBlock - 1)/threadsPerBlock;
4
5 //launch kernel
6 vectorAdd<<<blocksPerGrid,threadsPerBlock>>>(d_A,d_B,d_C,n);
```

Listing 5: vectorAdd (5)

Innerhalb des Kernels wird dann mit `blockDim.x * blockIdx.x + threadIdx.x` die Thread-Id bestimmt und die Addition für alle Elemente der Vektoren parallel durchgeführt.

Nach der Durchführung des Kernels kann der Ergebnisvektor mit `cudaMemcpy (h_C, d_C, size, cudaMemcpyDeviceToHost)` zurück auf die CPU kopiert werden, um gegebenenfalls weitere Operationen durchzuführen. Vor der Beendigung des Programms, muss noch der zugewiesene Speicher auf GPU und CPU wieder freigegeben werden:

```
1       //free device memory
2       cudaFree(d_A);
3       cudaFree(d_B);
4       cudaFree(d_C);
5
6       //free host memory
7       free(h_A);
8       free(h_B);
9       free(h_C);
```

Listing 6: vectorAdd (6)

Informationen zum gesamten Code können unter Anhang A.2.1 eingesehen werden. Er wurde den Beispielprogrammen des CUDA-Frameworks entnommen.

5 Implementation

Das Hauptziel bei der Implementation ist es, den in [Har07] vorgestellten Algorithmus für eine Kollisionserkennung auf der GPU umzusetzen. Ein weiteres Ziel ist die Implementation der dazu nötigen Physik-Engine.

In der Umsetzung dieser Arbeit wurden die Vorteile von Rigid-Body-, Partikel-Physik und einheitlichen Gittern, hinsichtlicht einer guten Kollisionserkennung auf der GPU, genutzt.

Die Engine ist in C++ geschrieben und die Kollisionserkennung auf der GPU mit CUDA umgesetzt. Das von der Arbeitsgruppe Müller im Institut für Computervisualistik im Fachbereich 4 bereitgestellte CVK-Framework wurde für die graphische Ausgabe verwendet. Außerdem wurde Doxygen für das Erstellen einer Code-Dokumentation genutzt.

Im Folgenden wird auf die generelle Konzeption und danach genauer auf die einzelnen Bestandteile der Implementation eingegangen.

5.1 Konzeption

Es wird eine Engine geplant, die sowohl eine Kollisionserkennung auf der CPU als auch auf der GPU bereit stellt. Zum besseren Performanz-Vergleich dieser Umsetzungen soll der selbe Algorithmus verwendet werden.

5.1.1 Erster Entwurf

Bei der ersten Konzeption diente das Entwurfsmuster aus [Mil07] als Grundlage, Abbildung 17 zeigt das zugehörige Klassendiagramm. Im Code wurden größtenteils die Formeln aus Kapitel 2.1.3 umgesetzt.

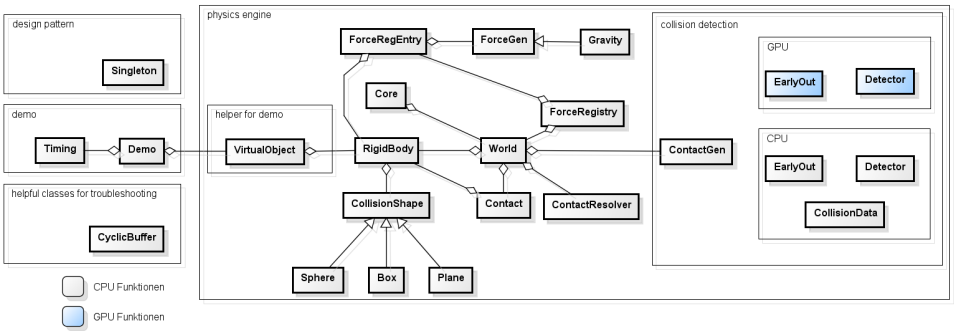


Abbildung 17: UML-Diagramm der ersten Konzeption

Der Kern der Physik-Engine war die Klasse “Core“, sie enthielt systemübergreifende Parameter und baute die Grundstruktur der Engine auf. Eine

weitere zentrale Klasse war "World‘“, sie beinhaltete die Liste aller vorhandenen Rigid-Bodies. Weiterhin wusste sie durch das “ForceRegistry“ über alle Körper und deren Kraft-Generatoren (“ForceGen“) Bescheid. Die Instanzen der Kollisionserkennung (“ContactGen“) und der Behandlung (“ContactResolver“) sowie die Liste aller Kontakte innerhalb einer Simulation waren der Welt-Klasse ebenfalls bekannt. “RigidBody“ enthielt alle wichtigen Attribute, wie Position, Geschwindigkeit und so weiter, für die Objekte innerhalb der virtuellen Umgebung. Durch die abstrakte Klasse “CollisionShape“ sollte ein Body die Formen einer Kugel (“Sphere“), eines Würfels (“Box“) oder einer Ebene (“Plane“) einnehmen können. Der Origin der Form war dabei gleichgesetzt mit der Position des Rigid-Bodies. Von der abstrakten Klasse “ForceGen“ konnten verschiedene Kräfte abgeleitet werden, wobei erst einmal nur eine Klasse für Schwerkraft (“Gravity“) existierte. “VirtualObject“ war die Verbindung zur “Demo“ und vereinte physikalische und graphische Darstellung eines Objekts. “ContactGen“ sollte die Kollisionen innerhalb einer Simulation finden und als Liste ausgeben, die dann vom “ContactResolver“ abgearbeitet und die Kontakte durch entsprechende Funktionen wieder aufgelöst werden konnten. Von “ContactGen“ aus sollte dann die Kollisionserkennung auf der CPU oder GPU gestartet werden. Für beide Versionen war ein Early-Out-Test (“EarlyOut“), durch das einheitliche Gitter, und eine Klasse für die daraus resultierenden Basis-Primitiven-Tests (“Detector“) angedacht. “Timing“ organisierte die FPS einer Anwendung und “CyclicBuffer“ war als Hilfsklasse für eine späteres Debuggen (Fehlerbehebung) der Engine vorgesehen, die jedoch nicht mit zur Physik-Engine gehören sollte und somit weggelassen werden konnte.

Bei dieser Konzeption gab es jedoch eine Schwierigkeit, der Algorithmus aus [Har07] war nur schwer daran anzupassen. Hauptsächlich aus dem Grund, dass die Kollisionserkennung und -behandlung in zwei separate Schritte unterteilt war und die Behandlung eine Liste der gefundenen Kontakte von der Erkennung als Eingabe erwartete. Der Algorithmus aus [Har07] vereint Erkennung und Auflösung jedoch in einem Schritt, gefundene Kollisionen werden direkt gelöst. Da für den Performanz-Vergleich von CPU und GPU der selbe Algorithmus verwendet werden sollte und der Schwerpunkt der Arbeit Kollisionserkennung auf der GPU ist, wurde der erste Entwurf abgeändert. Auch da dieser Algorithmus auf Partikeln basiert, die bis dato noch nicht beachtet wurden, mussten einige Änderungen an der bis dahin konzipierten und implementieren Engine vorgenommen werden. Unnötige beziehungsweise nicht mehr passende Klassen wurden entfernt und die Engine allgemein mehr auf eine repräsentative Demo zugeschnitten. Der neue Entwurf richtet sich so mehr nach [Har07] und [Fos11], Abbildung 18 zeigt das entsprechende Klassendiagramm.

5.1.2 Zweiter Entwurf

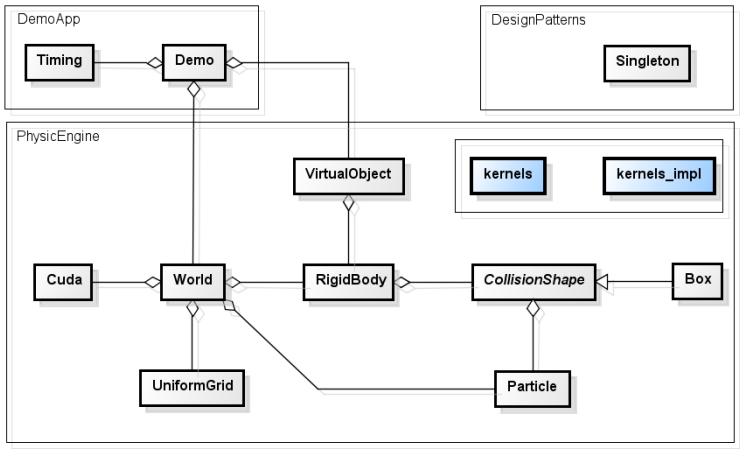


Abbildung 18: UML-Diagramm der Engine
für genaue Klassendefinitionen siehe Anhang A.1

Im neuen Entwurf der Physik-Engine ist die "World"-Klasse der zentrale Kern. Sie beinhaltet Attribute für die Größe der virtuellen Welt, den für alle Partikel gültigen Radius, Feder- und Dämpfungs-Koeffizienten, die Anzahl der Rigid-Bodies und Partikel in einer Simulation und zwei Arrays, die diese enthalten. Außerdem ist ein Parameter für einen festen Wert der Gravitation $(9,81)$ vorhanden, da keine Kraft-Generatoren genutzt werden,. Die World erzeugt eine Instanz des "UniformGrids" sowie der "Cuda"-Klasse und startet mit der Funktion "stepSimulation()" einen Simulations-Schritt.

Die Klasse "Demo" enthält einige anwendungsbezogene Parameter, wie Fenstergröße, eine Instanz von "Timing" für das Berechnen der FPS, eine Kamera und Szenegraphen-Wurzel aus dem bereits angesprochenen CVK-Framework. Eine Liste aller in der Simulation enthaltener "VirtualObject", die aus physikalischer (Rigid-Body) und graphischer Komponente (CVK-Geometrie beziehungsweise Szenegraphen-Knoten) bestehen, liegt ebenfalls vor. Die Demo-Klasse erstellt die "World" und damit die Physik-Engine, initialisiert die Szene mit der Funktion "run()" und startet dann die Simulation.

Ein Virtual Objekt (kurz VO) dient, wie bereits erwähnt, als Bindeglied zwischen physikalischer und graphischer Darstellung. Es besitzt eine Modelmatrix, die nach jedem Simulations-Schritt durch die aktuellen Werte der physikalischen Komponente neu berechnet wird, um damit die graphische Geometrie an aktualisierter Stelle rendern zu können.

Die Klassen "UniformGrid" und "Cuda" beinhalten alle nötigen Attribute und Funktionen für das einheitliche Gitter beziehungsweise um die Kollisionserkennung auf der GPU durchzuführen.

Ein Rigid-Body besitzt alle Attribute, die zur linearen und angularen

Bewegung nötig sind, sowie die Funktionen, um diese Werte in einem Schritt aktualisieren zu können. Wie im ersten Entwurf, kann durch das "CollisionShape" die Form des starren Körpers definiert werden, die jedoch momentan nur auf Würfel ("Box") festgelegt ist.

Die neue "Particle"-Klasse enthält alle Daten und Methoden für ein Teilchen, aus denen ein Rigid-Body beziehungsweise in der Implementation dessen Collision-Shape besteht.

Objektrepräsentation

Objekte werden einerseits als starre Körper betrachtet. Um die Berechnungen für ihr physikalisches Verhalten und besonders die Kollisionserkennung zu vereinfachen, werden diese Körper durch ein Set von Partikeln angenähert. Diese Partikel besitzen eine feste Position relativ zueinander und zum Ursprung des abstrahierten Rigid-Bodies. Alle Teilchen eines Objekts beziehungsweise alle Teilchen in einer Simulation teilen sich dabei den selben Radius. Abbildung 19 zeigt Form-Annäherungen mit verschiedenen Partikelgrößen. Diese Abstraktion lässt sich durch Voxelisierung bestimmen, indem ein 3D-Gitter um den starren Körper gelegt wird und pro Voxel dieses Gitters, das innerhalb des Objekts liegt, ein Partikel gesetzt wird. Mit Strahlen durch das Mesh entlang der Gitterzellen werden In- und Out-Punkte bestimmt, durch die entschieden wird, ob sich ein Voxel noch in einem Körper befindet oder nicht. Dies ist eine Möglichkeit der Voxelisierung, es existieren aber auch noch weitere. Auf der GPU kann die Annäherung durch *depth peeling* (Tiefenpeeling) und dem daraus resultierenden Tiefenbild des 3D-Objekts beschleunigt werden. [Har07]

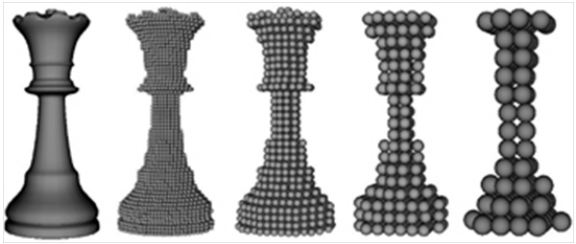


Abbildung 19: Objektrepräsentation [Har07]
Objekt-Annäherungen mit verschiedenen Partikel-Radien

In der zugehörigen Implementation dieser Arbeit werden vorerst nur Würfel als Form eines Rigid-Bodies zugelassen. Sie werden durch 27 Partikel angenähert, 3 in jeder Dimension (siehe Abbildung20). Man kann auch andere Partikelanzahlen wählen, jedoch wird eine 2x2x2-Formation schon zu unstabil. Durch die Teilchen-Abstraktion kommen Fehler auf, zum Beispiel beim Stapeln von Boxen, die durch eine feinere Aufteilung, also eine größere Partikel-Formation, minimiert werden können. Andererseits wird die Performanz der Engine und besonders der Kollisionserkennung mit zuneh-

mender Teilchenanzahl schlechter. Denn auch wenn Tests zwischen Partikeln sehr leicht und schnell sind, müssen bei hoher Anzahl, innerhalb eines Simulations-Schritts, viele dieser Tests durchgeführt werden. Auch hier gilt wieder das, zu Beginn schon genannte, Abwägen zwischen Komplexität und physikalischer Korrektheit. Es hat sich herausgestellt, dass die 3x3x3-Formation, fürs Erste, eine gute Wahl ist. [Fos11]

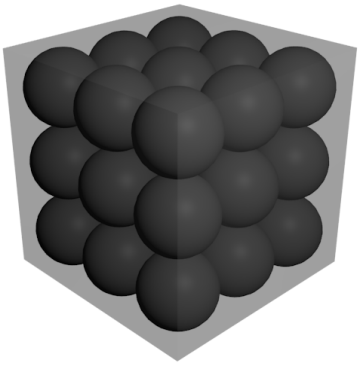


Abbildung 20: Würfel-Abstraktion [Fos11]
Würfel-Annäherung durch 27 Partikel $(3 * 3 * 3)$

Uniform Grid

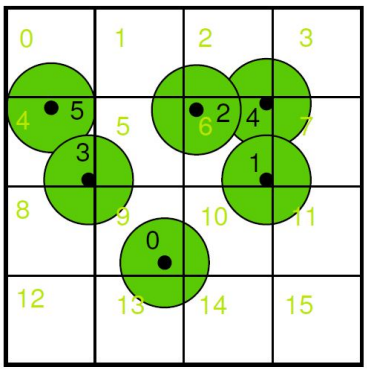
Wie erwähnt wird für die narrow-Phase ein einheitliches Gitter verwendet. Dieses wird über die Szene gelegt und alle beinhalteten Partikel der starren Körper werden in dieses einsortiert. Durch diese Strukturierung müssen in der broad-Phase dann nur noch Partikel innerhalb benachbarter Zellen auf Kollision getestet werden. Die verwendete Zellengröße entspricht dabei den Partikeldurchmessern (2 * Partikelradius), was gewährleistet, dass ein Partikel maximal 8 Zellen überlappen kann. Es wird ein sogenanntes loses Gitter verwendet, indem jeder Partikel nur in einer Gitterzelle zugewiesen wird. Aus diesem Grund müssen bei den genaueren Prüfungen auf Kontakt die Nachbarzellen mit einbezogen werden, da ein Partikel diese möglicherweise ebenfalls überlappen könnte. Eine Zelle besitzt dabei 26 Nachbarn, innerhalb eines 3x3x3-Blocks. Eine Alternative ist es ein Teilchen in jeder Zelle, die es überlappt, abzuspeichern. Das erleichtert den Aufwand bei den Kontakt-Tests, erhöht aber die Arbeit bei der Gittererstellung.[Gre10]

Die Gitterstruktur wird in jedem Simulations-Schritt neu aufgebaut. Sie besteht aus zwei Arrays, dem "gridCounters" und "gridCells". In gridCounters wird die Anzahl der in einer Zelle beinhalteten Partikel gespeichert und in gridCells sind die Indizes dieser Teilchen hinterlegt. In einer Update-Funktion wird für jeden Partikel in der Szene die entsprechende Gitterzelle ermittelt und dessen Index in ihr gespeichert. Dafür wird folgende Gleichung genutzt: [Gre10]

$$i_z = \frac{(p - min)}{d} \tag{27}$$

i_z	Zellenindex
p	Partikelposition
min	Gittereckenposition mit kleinsten Koordinaten
d	Zellengröße

Um dafür zu sorgen, dass eine Gitterzelle nicht mehr als die möglichen 4 Partikel beinhaltet, werden die gridCounters im Auge behalten. Auf der GPU helfen dabei atomare Operationen. In einem ersten Durchlauf werden alle Partikelanzahlen innerhalb der Zellen gezählt, um sie danach in das zusammenhängende Gitter-Array einzufügen. [Gre10]



Cell id	Count	Particle id
0	0	
1	0	
2	0	
3	0	
4	2	3, 5
5	0	
6	3	1, 2, 4
7	0	
8	0	
9	1	0
10	0	
11	0	
12	0	
13	0	
14	0	
15	0	

Abbildung 21: Uniform-Grid Strukturaufbau [Gre10]
Counter untergebracht in Array "gridCounters" der Größe [n]
Particle-Id untergebracht in Array "gridCells" der Größe [n*4]
mit n = Anzahl der Gitterzellen

Datenstrukturen

Die am häufigsten verwendete Struktur, sind Listen beziehungsweise Felder. So besitzt die Demo-Klasse einen Vektor als Liste von Virtual-Objekts. Überwiegend werden jedoch, auf Grund des besseren Zugriffs, 1D-Arrays verwendet. In der World-Klasse werden in zwei Arrays alle Rigid-Bodies und alle Partikel innerhalb einer Szene gespeichert. Das Partikel-Feld ist dabei 27-mal so groß wie das der Bodies, da jeder Würfel durch 27 Partikel abstrahiert wird. Jede Instanz der Rigid-Body-Klasse beziehungsweise dessen CollisonShapes besitzt ein Array, in dem diese 27 Partikel hinterlegt sind. Für das einheitliche Gitter existieren, wie im vorangehenden Abschnitt bereits erwähnt, ebenfalls noch 2 Arrays. Alle weiteren Daten wie Rigid-Body-Position, -Geschwindigkeit, -Masse oder entsprechende Werte für Partikel sind in den entsprechenden Klassen als Attribute gespeichert. Für die CPU-Version der Kollisionserkennung ist diese Aufteilung ausreichend, für

die Ausführung auf der GPU kommen aber weitere Felder hinzu. Da die GPU die Klassenattribute nicht ansprechen kann beziehungsweise diesen teuren Zugriffe vermieden werden sollen, wird für jedes Rigid-Body und Partikel-Attribut ein Array benötigt, dass in den Speicher der Grafikkarte kopiert werden kann. In den GPU-Speicher müssen auch einige feste Werte übertragen werden. Was bei der GPU-Version jedoch im Vergleich zur CPU-Variante anders gemacht werden muss, wird in Kapitel 5.3 genau erklärt.

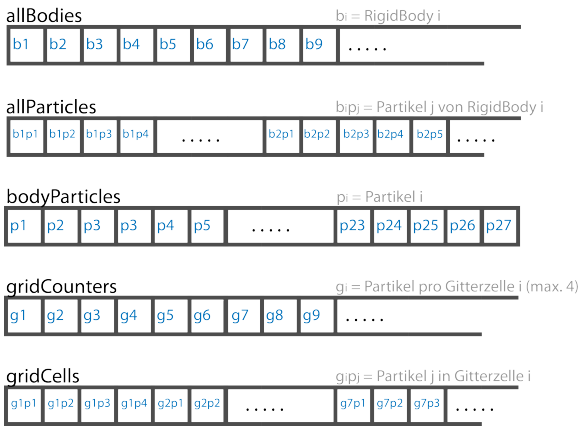


Abbildung 22: verwendete 1D-Arrays

allBodies	= alle Rigid-Bodies in der Simulation
allParticles	= alle Partikel in der Simulation
bodyParticles	= einen Rigid-Body repräsentierende Partikel
gridCounters	= Partikelanzahl in Gitterzelle
gridCells	= Verweise auf in Zelle beinhaltete Partikel

Rendering

Für die graphische Darstellung der Demo wird auf das von der Arbeitsgruppe Müller im Insitut für Computervisualistik im Fachbereich 4 bereit gestellte CVK-Framework zurückgegriffen. Damit werden die Render-Geometrien innerhalb der Szene in einer einfachen Hierarchie gespeichert. So wird die Wurzel eines Szenegraphen in der Demo-Klasse erstellt. Weiterhin existiert pro VirtualObjekt in der Szene ein weiterer Knoten, der an den Szenegraphen direkt unter die Wurzel angehängt wird. Da in der Engine vorerst nur Würfel als Rigid-Body Formen verwendet werden, ist nur eine CVK::Cube Geometrie im Speicher der Grafikkarte nötig. Auf diesen Cube verweisen alle VO-Knoten. Unterschiedliche Positionen und Rotationen sind durch die Modelmatrix der Nodes möglich. Die Bewegung der physikalischen Komponente eines VirtualObjekts, also eines Rigid-Body, wird durch die Engine simuliert und nach jedem Simulations-Schritt wird mit Hilfe der aktualisierten Position und Orientierung die Modelmatrix des VO-Knotens neu berechnet. Eine CVK::Plane Geometrie und ein zugehöriger Knoten wird erstellt, um den Boden der Simulationsdomäne zu modellieren. Der Szene-

graph besteht also aus einer Wurzel, unter die ein Knoten für den Boden und einer pro Objekt in der Szene gehängt wird. Nötige Geometrien und deren Daten im GPU-Speicher sind nur die einer Ebene und eines Würfels. Die Ebene für den Boden bleibt dabei statisch und wird nicht verändert. Die VO-Knoten sind jedoch dynamisch und müssen entsprechend aktualisiert werden. Dies geschieht durch Instanzing. Alle Knoten, die auf die Cube-Geometrie verweisen, können in einem Schritt gemeinsam gerendert werden. Ein und die selbe Geometrie wird so vielfach, entsprechend den zugehörigen Knoten-Modelmatrizen, in der Szene gezeichnet. Abbildung 23 zeigt noch einmal den Aufbau des Szenegraphen.

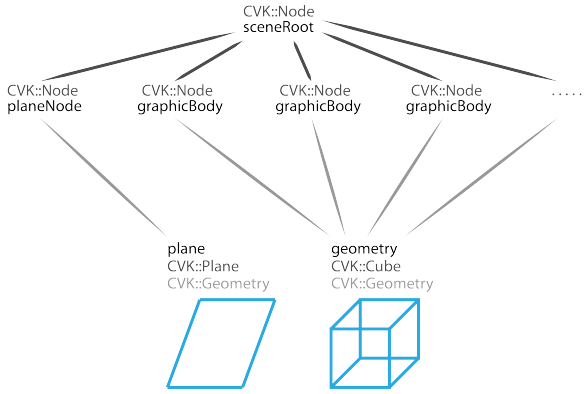


Abbildung 23: Szenegraph der Demo
aufgestellt mit Hilfe des CVK-Frameworks

Die Möglichkeit nur die Partikel darzustellen existiert ebenso. Dafür wird eine weiterer Wurzelknoten, also ein zweiter Szenegraph verwendet und die Partikel-Klasse um ein CVK::Node Attribut erweitert. Alle Teilchen-Knoten einer Simulation verweisen nun auf eine CVK::Sphere Geometrie und ein entsprechendes Material und hängen unter dem *partRoot*. Per bool-Abfrage kann zwischen den beiden Renderarten gewechselt werden.

In den folgenden Abschnitten wird der Ablauf eines Simulations-Schritts erklärt und auf die Unterschiede zwischen CPU- und GPU-Umsetzung eingegangen.

5.2 CPU-Version

Nachdem durch die Demo-Klasse eine Szene initialisiert wurde und alle Rigid-Bodies und deren Partikel erstellt und in den entsprechenden Arrays gespeichert wurden, kann die Simulation beginnen. Daraufhin werden solange Simulations-Schritte duchgeführt, bis das Programm abgebrochen wird. Ein Schritt unterteilt sich dabei in 6 Teilabschnitte (sihe Abbildung 24).

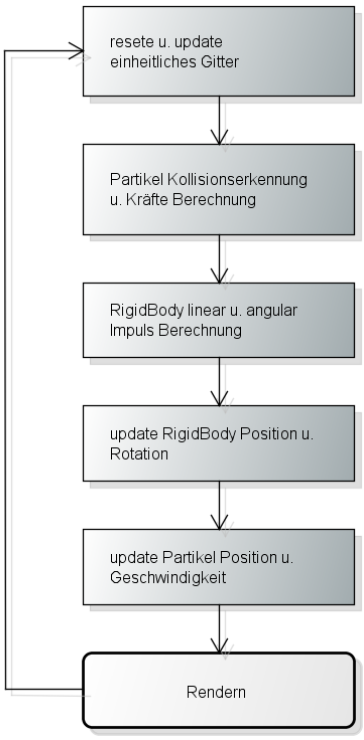


Abbildung 24: Flussdiagramm der CPU-Version
Ablauf der Kollisionserkennung auf der CPU

1. Aktualisiere Uniform-Grid:

Zuerst wird das Gitter aktualisiert, indem alle Einträge im gridCounters-Array auf 0 gesetzt werden, um dann über alle Partikel innerhalb der Szene zu iterieren und deren zugehörige Gitterzelle zu finden. Man berechnet durch Gleichung 27 die Zellen-Id, überprüft dann anhand ihres gridCounters wie viele Teilchen bereits in dieser Zelle liegen und speichert den Partikel-Index an entsprechender Stelle im gridCells-Array. Im Code ist die Funktion "updateGrid()" der UniformGrid-Klasse dafür zuständig. Sind alle Partikel einer Simulation in das Gitter eingeordnet, können die genauen Kollisions-Tests durchgeführt werden. [Fos11]

2. Partikel Kollisionen:

Nun werden alle Partikel innerhalb ihrer eigenen und deren 26 Nachbarzellen auf Kontakte überprüft. Dazu werden die Nachbarn eines Partikels aus der Gitterstruktur herausgelesen und nacheinander mit ihm getestet. Eine Kollision liegt vor, wenn die Distanz zwischen den beiden Teilchen kleiner ist als deren Durchmesser. Ist eine Kollision gefunden, wird diese direkt aufgelöst, indem durch Gleichung 28 die Reaktionskräfte berechnet werden. Die Kollisionserkennung und -behandlung werden also in einem Schritt

durchgeführt, per sogenannter *discrete element methode* (Diskrete-Element-Methode, kurz DEM). [Har07]

$$F_{i,s} = -k(d - |\boldsymbol{p}_{ij}|)\frac{\boldsymbol{p}_{ij}}{|\boldsymbol{p}_{ij}|} \tag{28}$$

$F_{i,s}$	*Abstoß-Kraft (lineare Feder)*
k	*Federkoeffizient*
d	*Partikeldurchmesser*
$\boldsymbol{p}_{ij}$	*realtive Position von Nachbarpartikel j zu Partikel i*

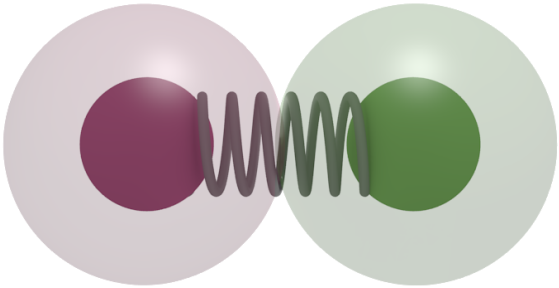


Abbildung 25: Partikel Abstoß-Kräfte [Fos11]
Abstoß-Kraft modelliert durch lineare Feder

$$F_{i,d} = \eta * \boldsymbol{v}_{ij} \tag{29}$$

$F_{i,d}$	*Dämpfungs-Kraft*
η	*Dämpfungskoeffizient*
$\boldsymbol{v}_{ij}$	*relative Geschwindigkeit*

Eine Dämpfungs-Kraft wird durch Formel 29 berechnet, um einen Energieverlust und damit ein leichtes Abbremsen der Objekte zu erreichen. Die dazu nötige Geschwindigkeit ist $\boldsymbol{v}_{ij}$, die relative Geschwindigkeit von Nachbar-Partikel j zu Partikel i.

Um Kollisionen mit dem Rand der Simulationsdomäne abzufangen, wird Gleichung 26 benutzt. Sie berechnet Rand-Federkräfte individuell zu jeder Achse, die dann in Rand-Normalenrichtung wirken. Formel 26 bestimmt die Federkraft für ein Teilchen, das mit einem Rand in x-Achsen-Richtung zusammenstößt. Für eine Kollision mit dem gegenüberliegenden Rand muss das $+r$ entsprechend in ein $-r$ geändert werden. [Fos11]

$$F_{i,b} = k(l_x - p_x - r) \tag{30}$$

$F_{i,b}$	*Rand-Feder-Kraft*
l_x	*Randposition auf der x-Achse*
p_x	*x-Komponente der Partikelposition*
r	*Partikelradius*

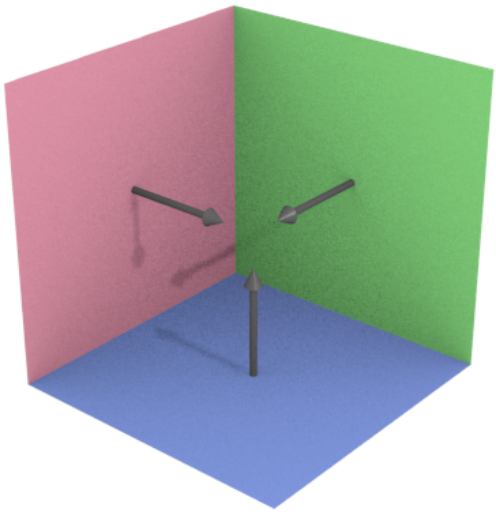


Abbildung 26: Rand Abstoß-Kräfte [Fos11]
an "Wänden" in x-/z-Richtung (+/-) und am Boden in y-Richtung (-)

Nachdem alle Kräfte für jeden Partikel berechnet wurden, können diese durch Formel 10 aus Kapitel 2.1.2 zu einem Wert aufsummiert werden. Mit den aktualisierten Partikel-Kräften können Kraft und Drehmoment ihrer Rigid-Bodies bestimmt werden. [Har07]

3. Rigid-Body Impulsberechnung:
Um die Kraft auf einen Rigid-Body zu bestimmen, werden die Kräfte der 27 formannähernden Partikel aufsummiert. Für die Kraft wird dazu auch Formel 10 genutzt. An dieser Stelle wird auch die auf den Körper wirkende Gravitation mit einbezogen. Da Partikel nur lineare Bewegung abdecken, muss nun noch das aus den Kräften resultierende Drehmoment eines Rigid-Bodies bestimmt werden. Dies wird durch die folgende Gleichung bewerkstelligt: [Har07]

$$\boldsymbol{\tau}_B = \sum_{i \in RB} (\boldsymbol{p}_{r,i} \times F_i) \tag{31}$$

τ_B	*Rigid-Body Drehmoment*
$\boldsymbol{p}_{r,i}$	*relative Position von Partikel i zum Rigid-Body Ursprung*
F_i	*auf Partikel i wirkende Kraft*

Durch die neu berechneten Werte können die Änderungen in Impuls und Drehmoment bestimmt werden und diese dann mit der *duration* (Dauer eines Iteration) eines Simulations-Schritts aktualisiert werden. Dies geschieht durch die Formeln 32 und 33 [Fos11]. Wird beim linearen Impuls ein vordefi-

nierter terminaler Wert überschritten, muss der zuvor berechnete auf diesen Wert gesetzt werden. Der terminale Impuls wird genutzt, um seltsam wirkendes Verhalten einzuschränken. [Har07]

$$I_l' = I_l + F * \Delta t \tag{32}$$

$$I_a' = I_a + \tau * \Delta t \tag{33}$$

I_l *linearer Impuls*
I_a *angularer Impuls*
Δt *Dauer*

4. Aktualisiere Rigid-Body Positionen:

Nachdem die Werte für Impuls und Drehmoment aktualisiert wurden, kann nun auch die neue lineare und angulare Geschwindigkeit eines Körpers und daraus wiederum die neue Position und Orientierung durch Iteration berechnet werden. Für lineare Geschwindigkeit wird Gleichung 2 und für angulare Geschwindigkeit Gleichung 20 und 19 verwendet. Danach wird anhand der linearen Geschwindigkeit und den Formeln 7 und 8 die Positions- und Geschwindigkeitsänderung berechnet. Mit Hilfe der angularen Geschwindigkeit und Gleichung 18 wird entsprechend der Rotations-Quaternion, also die Orientierung des Körpers, aktualisiert. [Har07]

5. Aktualisiere Partikel Positionen:

Zum Schluss müssen dann noch die Positions- und Geschwindigkeitsänderung der Rigid-Bodies auf deren Partikel angewandt werden. Dies geschieht durch Gleichung 34 und refeq:partveloc, für die die relative Partikelposition zum Massezentrum des starren Körpers r_i benötigt wird. [Har07]

$$\boldsymbol{p}_i = \boldsymbol{p}_j + \boldsymbol{p}_{r,i} \tag{34}$$

$$\boldsymbol{v}_i = \boldsymbol{v}_j + \boldsymbol{w}_j \times \boldsymbol{p}_{r,i} \tag{35}$$

mit

$\boldsymbol{p}_i$ *Partikelposition*
$\boldsymbol{p}_j$ *Rigid-Body Massezentrum*
$\boldsymbol{p}_{r,i}$ *relative Position von Partikel zu Massezentrum*
$\boldsymbol{v}_i$ *Partikelgeschwindigkeit*
$\boldsymbol{v}_j$ *lineare Rigid-Body Geschwindigkeit*
$\boldsymbol{w}_j$ *angulare Rigid-Body Geschwindigkeit*

$$\boldsymbol{p}_{r,i} = Q_j \boldsymbol{p_{r,i}}^0 {Q_j}^*$$

Q_j = *Rotations-Quaternion von Body j*
$\boldsymbol{p_{r,i}}^0$ = *initiale relative Partikelposition*

Dieser Schritt könnte auch als erstes ausgeführt werden, also bevor das einheitliche Gitter neu strukturiert wird.

6. Rendern der Objekte:
Wie oben bereits erklärt, kann, nachdem die Position und Orientierung eines Rigid-Bodies aktualisiert wurde, die Modelmatrix des zugehörigen VirtualObjekts neu berechnet werden. Dies geschiet durch Gleichung 36. Nach dieser Berechnung ist ein Simulations-Schritt der Engine beendet. Es wird nun über die Liste der VOs in einer Szene iteriert, um sie an der durch die ModelMatrix definierten Position darzustellen. Nach dem Rendern beginnt der nächste Simulations-Schritt wieder bei 1., der Aktualisierung des einheitlichen Gitters.

$$\boldsymbol{M} = \boldsymbol{p} * Q \tag{36}$$

$\boldsymbol{M}$ = *Modelmatrix*
$\boldsymbol{p}$ = *Position*
Q = *Rotations-Quaternion*

5.3 GPU-Version

Auf der GPU existiert im Prinzip der selbe Ablauf wie auf der CPU, lediglich die Aufteilung in Prozesse und die verwendeten Datenstrukturen unterscheiden sich. Hier kommen die Klassen "Cuda", "kernels" und "kernels_impl" aus Diagramm 18 zum Tragen.

Die Cuda-Klasse beinhaltet alle benötigten Array-Pointer auf den Speicher der CPU und GPU sowie einige Konstanten als Attribute. Welche Daten dies genau sind, wird im nächsten Abschnitt genauer erklärt. Mit der Funktion "initCUDA()" wird der benötigte Speicher allokiert. Auf der CPU werden neue Arrays per `new Typ[Array-Groesse]` angelegt und auf der GPU per `cudaMalloc()` Speicher in der selben Größe reserviert. "initCUDAGrid()" initialisiert alle für das einheitliche Gitter erforderlichen Daten und "updateHostArrays()" füllt die zuvor angelegten Arrays auf der CPU. In der Methode "hostToDevice()" werden die Daten von der CPU in den Speicher der Grafikkarte übertragen. Eine letzte wichtige Funktion ist "stepCUDA()", in ihr wird der Ablauf der Simulation auf der GPU durch die entsprechenden Kernel-Aufrufe definiert (siehe Abbildung 27). Diese Aufrufe starten jedoch die eigentlichen Kernel noch nicht direkt, für jeden ist noch eine Methode zur Berechnung der richtigen Block- und Thread-Anzahlen zwischengeschoben, diese sind in der Cuda-Klasse "kernles.cu" untergebracht. Die eigentlichen Kernel sind in "kernles_impl" implementiert. Sie werden dann aus der entsprechenden Funktion in "kernels.cu" mit den berechneten Block- und Thread-Werten gestartet.

Datenstrukturen

Alle objektspezifischen Daten müssen auf der GPU anders gespeichert werden, da die entsprechenden Klassen nur auf der CPU vorliegen. Dazu werden die Daten aller Rigid-Bodies und Partikel zusammengefasst und je in einem 1D-Array untergebracht. Zum Beispiel enthält das Array “d_rbPos"die Positionen aller Rigid-Bodies innerhalb der Simulation. Außerdem müssen auch einige objektunabhängige konstante Werte, wie Gittergröße, Partikelradius oder Terminalgeschwindigkeit für die Berechnungen vorliegen. Folgende Arrays und Konstanten werden auf der Grafikkarte benötigt: [Har07]

Rigid-Body Arrays		
Name	*Typ*	*Inhalt*
d_rbMass	float	Masse
d_rbForce	glm::vec3	Kraft
d_rbPos	glm::vec3	Position
d_rbVeloc	glm::vec3	lineare Geschwindigkeit
d_rbLinMom	glm::vec3	linearer Impuls
d_rbRotQuat	glm::quat	Rotations-Quaternion
d_rbRotMat	glm::mat3	Rotations-Matrix
d_rbAngVeloc	glm::vec3	angulare Geschwindigkeit
d_rbAngMom	glm::vec3	angularer Impuls
d_rbInitInverseInertTensDiago	glm::vec3	Diagonale des inversen initial Inertia Tensor
d_rbInverseInertTens	glm::mat3	inverser Inertia Tensor
Größe	je ein Eintrag pro Rigid-Body	
Partikel Arrays		
Name	*Typ*	*Inhalt*
d_pMass	float	Masse
d_pPos	glm::vec3	Position
d_pVeloc	glm::vec3	Geschwindigkeit
d_pForce	glm::vec3	Kraft
d_pGridIndex	glm::ivec3	Gitterindex
Größe	je ein Eintrag pro Partikel	
Gitter Arrays		
Name	*Typ*	*Inhalt*
d_gridCounters	int	Anzahl in Zelle beinhalteter Partikel
d_gridCells	glm::ivec3	Verweis auf beinhaltende Partikel
Größe	je ein Eintrag pro Gitterzelle	

Konstanten		
Name	*Typ*	*Inhalt*
d_voxelS	float	Voxelgröße
d_gridSL	int	Gitterseitenlänge
d_worldS	float	Simulationsdomänengröße
d_springC	float	Federkoeffizient
d_dampC	float	Dämpfungskoeffizient
d_pRadius	float	Partikelradius
d_duration	float	Dauer eines Simulationsschritts
d_termVeloc	float	terminal Geschwindigkeit
d_gridMinPosVecX	float	minimale Gitterposition (x-Komponente)
d_gridMinPosVecY	float	minimale Gitterposition (y-Komponente)
d_gridMinPosVecZ	float	minimale Gitterposition (z-Komponente)

Zu jedem dieser Arrays oder Konstanten existiert auch eine entsprechende Variable auf der CPU. Es gibt beispielsweise zu "d_rbMass" auf der GPU ein "h_rbMass" auf der CPU. Durch diese werden die initialen Werte festgelegt und dann mit Hilfe der entsprechenden CUDA-Befehle in den Grafikspeicher geladen. Für die Arrays wird dazu `cudaMemcpy()` genutzt, wohingegen die Konstanten durch `cudaMemcpyToSymbol()` in den constant-memory geladen werden. Sind die Werte auf der GPU, verweilen sie dort bis zum Ende der Simulation. Alle Änderungen werden in den Arrays aktualisiert und ein Austausch mit den Klassen auf der CPU ist nicht weiter nötig.

Kernel

Die Berechnungen, die auf der CPU in den entsprechenden Methoden der einzelnen Klassen untergebracht sind, werden nun zur Ausführung auf der GPU in einzelne Kernel gepackt. So existieren Kernel für die Funktionen der Einheitsgitter-, RigidBody- und Partikel-Klasse.

Die zuvor bereits genannten vorgeschobenen Funktionen aus "kernels.cu", die Block- und Thread-Dimensionen berechnen, agieren wie folgt.

Die Anzahl an Threads pro Block hängt von der Architektur der Grafikkarte und der unterstützten CUDA-Version ab, kann aber mit Rücksicht auf diese Beschränkungen frei gewählt werden. Ist die Blockgröße definiert, muss berechnet werden wie viele Blocks benötigt werden, um die gewünschte Threadzahl zu erreichen. Dabei hilft die Funktion "nearHighVal" (siehe Listing 8), die das Ergebnis von a/b auf den nächst größeren ganzzahligen Wert rundet, wobei a der gewünschten Threadanzahl und b der Blockgröße entspricht. So kommt es natürlich vor, dass die berechnete Threadanzahl größer als die benötigte ist. Eine if-Abfrage in jedem Kernel sorgt dafür, dass nur die gewünschte Anzahl an Threads Code ausführt und alle Threads mit höheren IDs ignoriert werden. Mit den bestimmten Werten für Blockanzahl

und Threads pro Block wird dann der eigentlichen Kernel per `kernelname <<< numBlocks, numThreads >>>()` gestartet.

Wenn zum Beispiel 100 Boxen, also 2700 Partikel in der Simulation vorliegen und in einem Kernel, in dem die Blockgröße mit 64 definiert ist, ein Thread pro Partikel benötigt wird, ist die entsprechende Blockanzahl $2700/64 = 42,1875$, gerundet also 43. Der Kernel wird nun mit `<<< 43, 64 >>>` gestartet und mit der Abfrage `if(thread-id < 2700)` alle überflüssigen Threads von der Codeausführung ausgeschlossen.

```
1 int blockSize = 64;      //64, 128, 256, 512, 1024
2 int numThreads = blockSize;
3 int numBlocks = nearHighVal(g, numThreads);
4
5 resetGridC <<< numBlocks, numThreads >>>(gridCounters,
      gridCells, g);
```

Listing 7: Thread-/Block-Berechnung

```
1 int nearHighVal(int a, int b){
2         return (a % b != 0) ? (a / b + 1) : (a / b);
3 }
```

Listing 8: Hilfsfunktion zur Thread-/Block-Berechnung

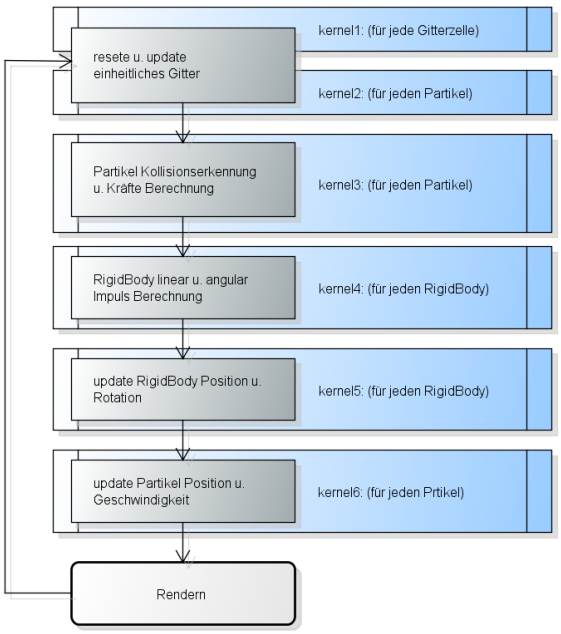


Abbildung 27: Flussdiagramm der GPU-Version
Ablauf der Kollisionserkennung auf der GPU

In den weiteren Abschnitten werden die Aufgaben der Kernels erklärt und mit Pseudocode veranschaulicht. Informationen zum genauen Code können in Anhang A.2.2 eingesehen werden.

Der erste Schritt der Simulation, also das Aktualisieren des einheitlichen Gitters, wird auf der GPU in 2 Kernel aufgeteilt. Da das Gitter zuerst zurückgesetzt wird, ist ein Zugriff pro Gitterzelle vorausgesetzt. Danach wird die Zellen anhand der Partikelindices neu befüllt, wobei Zugriffe pro Partikel erforderlich sind. Daher ist es effizienter diese Aufgaben aufzuteilen. [Fos11]

Der "resetGridC"-Kernel wird also mit einem Thread pro Gitterzelle gestartet und setzt alle Werte im "gridCounters"-Array auf 0 und im "gridCells"-Array auf -1. Im "updateGridC"-Kernel wird dann pro Partikel deren Gitterindex berechnet und an der entsprechenden Stelle der Gitter-Arrays eingefügt. Eine atomare Operation (`atomicInc()`) hilft dabei, nicht mehr als 4 Teilchen pro Gitterzelle zuzulassen. Denn da der Kernel für alle Partikel gleichzeitig aufgeführt wird, wäre es möglich, dass mehrere Teilchen gleichzeitig auf die selben Array-Positionen zugreifen wollen. Die atomare Operation verhindert dies und limitiert die Zugriffe auf eine Position auf 4. [Gre10]

```
__global__ void resetGridC(gridCounters,gridCells,gridSize){
 get thread-id;
 if(id<gridSize){
  gridCounters[id] = 0;
  gridCells[id] = -1;
 }
}
```

Listing 9: resetGrid-Kernel

```
__global__ void updateGridC(gridCounters,gridCells,pPos,
    pGridIndex,numPart){
 get thread-id;
 if (id < numPart){
  pGridIndex berechnen;
  bool validIndex = testen ob gültiger Index;
  if (validIndex){
   x/y-Offset berechnen;
   int fGI = mit Offsets Array-Index berechnen;

   //atomare Operation
   unsigned int* atom = (unsigned int*)(&gridCounters[fGI]);
   int particlesInCell = atomicInc(atom, 4);

   id an entsprechender Stelle in gridCells einfügen;
  }
 }
}
```

Listing 10: updateGrid-Kernel

Ist das einheitliche Gitter aktualisiert, wird der “calcCollForcesC“-Kernel mit einem Thread pro Partikel gestartet. In ihm werden mit entsprechenden for-Schleifen über das Gitterzellen-Array die Nachbarn eines Teilchens herausgesucht und dann auf ihren Abstand zu diesem getestet. Nach der Berechnung der aus dem Abstand resultierenden Kraft folgt noch eine Randbehandlung, in der der Partikel auf Kollision mit Boden und Wänden der Simulationsdomäne geprüft wird. Zum Schluss wird noch ein Dämpfungswert auf die Kraft angewandt, damit im System Energie verloren geht und die Boxen zur Ruhe kommen können. [Fos11]

```
__global__ void calcCollForcesC(pMass,pPos,pVeloc,pForce,
    pGridIndex,gridCounters,gridCells,numPart){
 get thread-id;
 if (id < numPart){
  pForce[id] = 0;
  x/y-Offset berechnen;
  int fGI = mit Offsets Array-Index berechnen;
  //Partikel-Kollisionen
  glm::vec4 neighborCells[27] = Nachbar-Partikel beschaffen;
  für alle Nachbar-Partikel{
   auf Abstand testen;
   entsprechende resultierende Kraft berechnen;
  }
  //Randbehandlung
  bool collisionOccured = false;
  wenn Kollision mit Boden{
  collisionOccured = true;
  etsprechende resultierende Kraft berechnen;
  }
  wenn Kollision mit Wand in x- oder z-Richtung{
   wenn in positiver Richtung{
    collisionOccured = true;
    etsprechende resultierende Kraft berechnen;
   }
   wenn in negativer Richtung{
    collisionOccured = true;
    etsprechende resultierende Kraft berechnen;
   }
  }
  Dämpfung mit einbeziehen;
 }
}
```

Listing 11: calculateForces-Kernel

Der “updateMomC“-Kernel wird nun mit einem Thread pro Rigid-Body ausgeführt. Mit ihm wird die Kraft der Gravitation in y-Richtung auf einen Body ausgeübt, alle deren Partikel-Kräfte aufsummiert und das resultierende Drehmoment berechnet. Aus gesamter Kraft und Drehmoment eines Körpers kann dann der lineare und angulare Impuls aktualisiert werden. [Har07]

```
__global__ void updateMomC(rbMass,rbForce,rbPos,rbLinMom,
    rbAngMom,pPos,pForce,numBodies){
 get thread-id;
 particleIndex = id * 27;
  if (id < numBodies){
   //Gravitation mit einbeziehen
   rbForce[bi].y = rbMass[bi] * -9.81f;
   Drehmoment = 0;
   für alle Nachbar-Partikel{
    Kraft aufsummieren;
        entsprechendes Drehmoment berechnen;
   }
   Terminalimpuls auf Terminalgeschwindigkeit berechnen;
   linearen Impuls berechen;
   Impuls durch Terminalimpuls limitieren;
   angularen Impuls berechnen;
  }
}
```

Listing 12: updateMomentum-Kernel

Mit Hilfe der aktualisierten Impulse kann die Geschwindigkeit und Position eines Bodies durch Iteration bestimmt werden. Dazu wird der "iterateC"-Kernel einmal pro Rigid-Body gestartet, in dem lineare Geschwindigkeit und Position durch Euler Integration berechnent werden und auch die angularen Komponenten entsprechend aktualisiert werden. [Har07]

```
__global__ void iterateC(rbMass,rbPos,rbVeloc,rbLinMom,
    rbRotQuat,rbRotMat,rbAngVeloc,rbAngMom,initIITDiago,
    inverInertTens,numBodies){
 get thread-id;
 if (bi < numBodies){
  inversen Inertia-Tensor aktualisieren;
  //linearen Schritt durchführen
  //Geschwindigkeit und Position berechenen
  rbVeloc[bi]= rbLinMom[bi] / rbMass[bi];
  rbPos[bi] = rbPos[bi] + rbVeloc[bi] * d_duration;

  //angularen Schritt durchführen
  Geschwindigkeit aktualisieren;
  Rotations-Quaternion aktualisieren;
  Rotations-Quaternion normalisieren;
  Rotations-Matrix aktualisieren;
 }
}
```

Listing 13: iterate-Kernel

Zum Schluss müssen die aktualisierten Werte der Rigid-Bodies auf ihre repräsentierenden Partikel übertragen werden. Dabei hilft der "updatePartC"-Kernel, der mit einem Thread pro Partikel deren relative Position bestimmt

und damit die Partikel-Position und -Geschwindigkeit aktualisiert. [Fos11]

```
__global__ void updatePartC(rbPos,rbVeloc,rbRotMat,
    rbAngVeloc,pPos,pVeloc,numPart){
 get thread-id;
 int bi = pi / 27;
  if (pi < numPart){
   relative Position bestimmen;
   Partikel-Position aktualisieren;
   Partikel-Geschwindigkeit aktualisieren;
  }
}
```

Listing 14: updateParticle-Kernel

Rendering

Alle Daten über Rigid-Bodies, Partikel und das Gitter bleiben über die Laufzeit der Simulation im Grafikspeicher und werden auch nur dort verwendet. Lediglich für das Rendern der Würfel wird momentan in jedem Simulationsdurchlauf noch ein Datentransfer von GPU zu CPU benötigt. Da für das Darstellen das CVK-Framework genutzt wird und dieses auf OpenGL basiert, werden die Daten von Rigid-Body Positionen und Orientierungen (oder Positionen für das Rendern der Partikel) per `cudaMemcpy()` zurück auf die CPU kopiert. Mit ihrer Hilfe wird die Modelmatrix der entsprechenden Szenegraphen-Knoten aktualisiert.

Auch wenn nur wenige Daten zwischen GPU und CPU ausgetauscht werden, beeinflusst dies die Performanz der Anwendung. Eine effizientere Methode wäre ein CUDA-OpenGL-Austausch, durch den OpenGL die Daten aus den CUDA-Arrays übergeben bekommt. Auch für das Rendern blieben so alle Daten über die Dauer der Simulation auf der Grafikkarte. Im Ausblick (Kapitel 7.1) wird noch einmal etwas genauer auf dieses Verfahren eingegangen.

6 Vergleich und Ergebnisse

In diesem Kapitel wird die Performanz der Simulation auf CPU und GPU aufgezeigt und verglichen. Tabellen und entsprechende Graphen zeigen die durchschnittlichen FPS und den Performanz-Vorteil zwischen den einzelnen Versionen.

Alle Tests wurden auf einem Computer mit folgenden technischen Daten ausgeführt:

OS	Windows 7 (Professional)
CPU	AMD Athlon 64 X2 Dual Core Processor 6000+ 3.00GHz
RAM	3,00 GB
GPU	NVIDIA GeForce GTX 650 (with 2GB GDDR5)

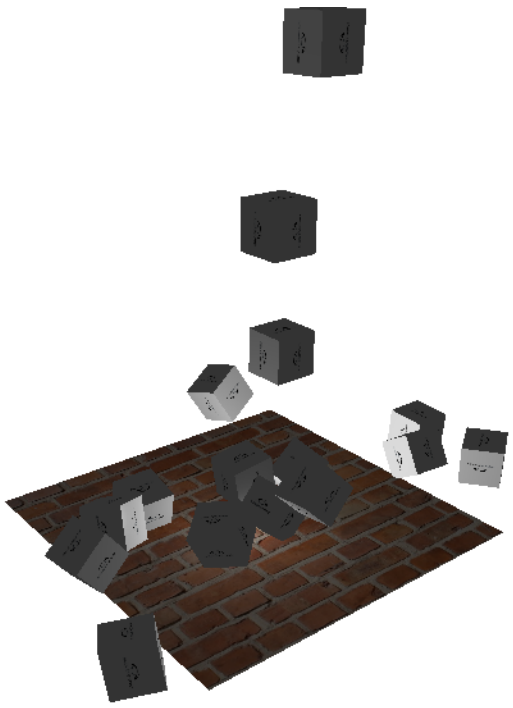


Abbildung 28: Simulationsszenario
in verschiedenen Anzahlen herab fallende Würfel

Das für die Tests simulierte Szenario sind von oben in die Simulationsdomäne fallende Boxen (siehe Abbildung 29). In [Fos11] haben sich die folgenden Paramter als günstig erwiesen und werden auch in dieser Demo verwendet:

Variablenname	Wert	Beschreibung
worldsize	15.0	Größe der Simulationsdomäne
springCoeff	100.0	Federkoeffizient
dampCoeff	0.5	Dämpfungskoeffizient
duration	0.01	Dauer eines Simulationsschritts
partRadius	0.40	Partikelradius
terminalVeloc	20.0	Terminalgeschwindigkeit

Erste Test ergaben, dass bei einer Anzahl von 100 Körpern, also 2700 Partikeln die CPU als auch die GPU mit konstanten 60 FPS (Frames per Second) liefen. Das 60 Frames-Limit wurde jedoch von OpenGL, also der grafischen Ausgabe verursacht, dies wurde für die folgenden Testergebnisse auskommentiert, um die korrekten Simulationszeiten zu erlangen.

6.1 CPU

Auf der CPU wurden Simulationen mit und ohne Gitterstruktur durchgeführt. Für die Tests ohne Gitter wird das Erstellen und Aktualisieren der Struktur weg gelassen und nach und nach jeder Partikel mit jedem anderen innerhalb der Simulation auf Kontakt überprüft. Die folgende Tabelle zeigt die durchschnittlichen FPS der Simulation bei verschiedenen Würfelanzahlen:

#Würfel #Partikel	10 270	50 1350	100 2700	500 13500	1000 27000
ohne Gitter	411,3	26,3	6,6	0,0	0,0
mit Gitter	804,5	384,2	218,0	61,8	34,4
PV	$\times\mathbf{2,0}$	$\times\mathbf{14,6}$	$\times\mathbf{33,1}$	$\times\boldsymbol{kVm}$	$\times\boldsymbol{kVm}$

PV = Performanz-Vorteil gegenüber Version ohne Gitterstruktur
kVm = kein Vergleich möglich, da durch 0 geteilt wird

Hier erkennt man schon den großen Vorteil einer Narrow-Phase. Wohingegen die Simulation ohne Gitterstruktur schon bei circa 100 Boxen (2700 Partikel) ihre Grenzen erreicht, schafft der Durchlauf mit einheitlichem Gitter die 33-fache Leistung.

Zum Verhalten ist zu sagen, dass es vorkommen kann, dass die Würfel leicht ineinander und versetzt liegen, wenn sie gestapelt werden. Dies ist durch die Annäherung durch Partikel verschuldet, sie rutschen quasi ineinander und füllen die Lücken auf.

6.2 GPU

Die Kollisionserkennung auf der GPU läuft immer mit Gitterstruktur. Die durchgeführten Tests unterscheiden sich in der festgelegten Blockgröße der

Kernel. Es wurden 2er-Potenzen als Größen gewählt, da auch die Kernanzahl der Grafikkarte mit 384 einer 2er-Potenz entspricht.

Die folgende Tabelle zeigt den Performanz-Vorteil der verschiedenen GPU-Versionen gegenüber der Simulation auf der CPU mit Gitter:

#Würfel #Partikel	10 270	50 1350	100 2700	500 13500	1000 27000
CPU	804,5	384,2	218,0	61,8	34,4
GPU 64	1250,7	1080,7	909,8	409,0	227,4
PV	**×1,6**	**×2,8**	**×4,2**	**×6,6**	**×6,6**
GPU 128	1222,5	1077,0	938,1	400,3	226,9
PV	**×1,5**	**×2,8**	**×4,3**	**×6,5**	**×6,6**
GPU 256	1206,9	1076,2	934,6	403,0	226,5
PV	**×1,5**	**×2,8**	**×4,3**	**×6,5**	**×6,6**
GPU 512	1206,8	1051,8	912,7	375,1	224,9
PV	**×1,5**	**×2,7**	**×4,2**	**×6,1**	**×6,5**
GPU 1024	1199,0	1041,4	852,9	355,3	198,4
PV	**×1,5**	**×2,7**	**×3,9**	**×5,8**	**×5,8**

PV = Performanz-Vorteil gegenüber CPU

Durch diese Werte wird klar, dass bei sehr wenigen zu simulierenden Objekten mit der GPU keine nennenswerte Leistungsvorteile zu erreichen sind. Doch schon ab 100 Boxen ist die Simulation auf der GPU 4-mal und bei 1000 Würfeln schon 6,5-mal so schnell.

Außerdem zeigen die Performanz-Vorteile, dass eine kleinere festgelegte Blockgröße, also die Anzahl an Threads pro Block, die Simulation am meisten beschleunigt (64, 128, 256).

Um die Grenzen der CPU, aber auch der GPU noch besser zu überprüfen, wurden noch Tests mit höheren Würfelanzahlen durchgeführt. Hier werden aber nur noch die CPU-Version mit Gitter und die GPU-Version mit Blockgröße 128 verglichen:

#Würfel #Partikel	10 270	50 1350	100 2700	500 13500	1000 27000	3000 81000	6000 162000	10000 270000
CPU	804,5	384,2	218,0	61,8	34,4	19,6	13,2	9,6
GPU 128	1222,5	1077,0	938,1	400,3	226,9	161,8	117,4	73,9
PV	**×1,5**	**×2,8**	**×4,3**	**×6,5**	**×6,6**	**×8,3**	**×8,9**	**×7,7**

Wo bei 6000 Boxen die CPU in die Knie geht, schafft die GPU-Umsetzung immer noch über 100 durchschnittliche FPS und ist somit mit der 9-fachen Leistung auf dem Höhepunkt des Performanz-Vorteils gegenüber der CPU. Ab 10000 Würfeln lässt dann auch die Simulation der GPU wieder deutlicher nach.

Für die Simulation von 1000 Boxen (27000 Partikel) und einer Gittergröße von 64000 Zellen ($40 * 40 * 40$) wurden auch die Laufzeiten der einzelnen Kernel überwacht:

Kernel	resetGrid	updateGrid	calcForces	updateMom	iterate	updatePart
ausgeführt pro	Gitterzelle	Partikel	Partikel	Box	Box	Partikel
Zeit (ms)	0,153	0,157	0,695	0,370	0,143	0,143

ms = Millisekunden

Diese Zahlen zeigen, dass der Kernel zur Findung der Nachbarpartikel und dem anschließenden Berechnungen der Abstoß-Kräfte den aufwendigsten Arbeitsschritt beinhaltet. Das Aktualisieren der Rigid-Body Impulse im updateMomentum-Kernel braucht auch seine Zeit. Die restlichen Kernel sind entsprechend ihres Umfangs sehr schnell abgearbeitet. Die Gesamtzeit eines Simulationsdurchlaufes auf der GPU beträgt circa $0,001661$ Sekunden.

Auch in der GPU-Version kommt es vor, dass die Boxen leicht versetzt und ineinander liegen, wenn sie gestapelt werden. Zusätzlich liegen einige Boxen auf ihren Kanten am Boden auf. Dies ist jedoch vermutlich nur ein Fehler in der Berechnung der Modelmatrix, da ihre Partikel sich so verhalten wie sie es sollten.

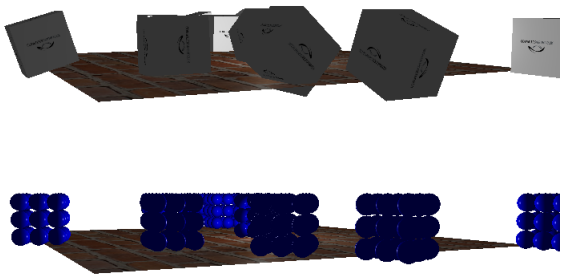


Abbildung 29: Würfel- vs. Partikel-Rendering
Partikel verhalten sich korrekt, einige Boxen werden falsch dargestellt

6.3 Fazit

Die oben aufgelisteten Werte zeigen, dass schon das einheitliche Gitter in der CPU-Version einen großen Performanz-Vorteil liefert. Die Berechnungen auf der GPU bringen bei einer Blockgröße von 64, 128 und 256 ab einer Anzahl von 100 Boxen (2700 Partikel) einen nennenswerten Leistungsschub. Bei 6000 Boxen (162000 Teilchen) besitzt die GPU-Version mit der 9-fachen

Leistung ihren Höhepunkt. Die Zeitmessung der Kernels hebt deutlich die Berechnung der Abstoß-Kräfte als aufwendigsten Arbeitsschritt hervor. Die Blockgrößen 64, 128 und 256 sind effizienter als 512 und 1024, weil die 384 Kerne der GeForce GTX 650 mit ihnen besser ausgenutzt werden.

7 Ausblick

In den folgenden Abschnitten wird zunächst auf mögliche Erweiterungen und Optimierungen der eigenen Umsetzung eingegangen. Zum Schluss folgt eine generelle Aussicht auf die Zukunft von Grafikkarten, Physik Engines und GPU-unterstüzter Kollisionserkennung.

7.1 Ausblick auf eigene Implementation

Es lassen sich in verschiedenen Bereichen noch Optimierungen vornehmen. Geometrische, strukturelle und mathematische Aspekte können geändert werden, um die Engine zu erweitern und zu verbessern.

Objektrepräsentation

Momentan können in der Anwendung nur Würfel, bestehend aus 27 Partikeln, simuliert werden. Weitere Formen würden die Sache natürlich noch interessanter machen. Um neue Primitive hinzuzufügen, müssten entsprechende Klassen von "CollisionShape" abgeleitet werden, in deren "calcParticles()"-Methode definiert wird, wie die Form durch Partikel repräsentiert wird.

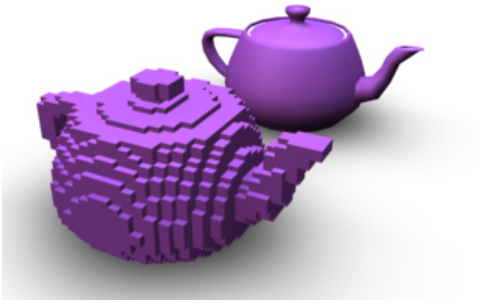


Abbildung 30: Voxelisierung [Vox12]
Annäherung einer beliebigen Form durch Würfel

Ein anderer Ansatz für weitere Formen basiert auf den schon vorhandenen Würfeln. Durch Voxelisierung (englisch *Voxelisation*) könnten beliebige Objekte durch Voxel, also Würfel, angenähert werden, so wie es vergleichsweise momentan schon mit den Boxen und ihren Partikeln geschieht. Die Form muss nicht als Collision-Shape definiert sein. Denn durch das Verfahren der Voxelisierung erhält man eine Datenstruktur, die die Angleichung speichert. So läge quasi eine doppelte Annäherung vor, eine beliebige Form abstrahiert durch Boxen, die wiederum durch Partikel repräsentiert werden. Analog zur Aufsummierung aller Partikel-Kräfte wird dann auch eine Aufsummierung aller Würfel-Kräfte benötigt, um die Änderungen der gesamten Form in einem Simulationsschritt zu erhalten.

Durch Voxelisierung könnte eine weit größere Anzahl an verschiedenen Formen mit der Engine simuliert werden. Auch wäre das Einfügen des Verfahrens einfacher als jedes Primitive einzeln als abgeleitete Klasse von "CollisionShape" zu modellieren.

Kräfte

Im aktuellen Stand der Demo werden außer der Gravitation, die als fester Wert in der "updateMomenta()"-Methode der Rigid-Body-Klasse oder im entsprechenden Kernel definiert ist, und den Abstoßkräften am Rand der Simulationsdomäne und der Partikel untereinander, keine weiteren Kräfte betrachtet. Um beliebige weitere hinzuzufügen müssten diese ebenso an den entsprechenden Stellen im Code angehangen werden. Dann würden sie jedoch, wie die Gravitation, für alle Objekte innerhalb einer Simulation gleich gelten. Besser sind da die in Kapitel 2.1.2 und 5.1.1 vorgestellten Force-Generators und entsprechende Register, die speichern, welche Art von Kraft auf welche Objekte wirken soll.

Datenstrukturen

Bessere Datenstrukturen als Arrays wären, besonders für die GPU-Version, Texturen. Da die Architektur einer Grafikkarte für das Arbeiten mit Texturen ausgelegt ist, wäre mit Sicherheit ein Performanzschub erkennbar. Mit Flat 3D-Texturen beziehungsweise geschichteten 2D-Texturen wäre vor allem die Struktur des einheitlichen Gitters noch effizienter und übersichtlicher zu speichern. Auch könnten möglicherweise bestehende Restriktionen von 1D-Arrays umgangen werden, indem die Rigid-Body und Partikel Daten in 2D Texturen untergebracht werden. [Har07]

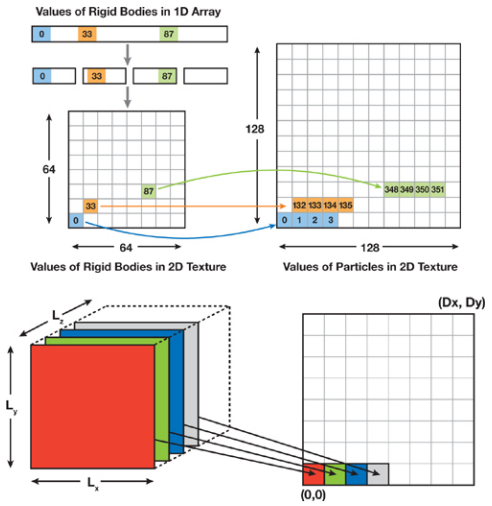


Abbildung 31: 1D-, 2D- und Flat 3D-Texturen [Har07]

Integration

Um die Positions- und Geschwindigkeitsänderung über die Dauer eines Simulationsschritts zu bestimmen, wird in der "iterate()"-Funktion beziehungsweise -Kernel die Euler Integration genutzt. Sie ist ein einfaches und nachvollziehbares Verfahren, weist jedoch Schwächen in Genauigkeit, Stabilität und Rechenzeit auf. [Rie03]

Hier könnten andere Integrationsverfahren für bessere numerische und dadurch auch geometrische Robustheit sorgen. Mögliche Verfahren sind: [Kum10]

- Störmer-Verlet-Algorithmus
- Leapfrog-Algorithmus
- Runge-Kutta-Methoden

Rendering

Eine letzte Verbesserung beschäftigt sich mit dem in Kapitel 5.3 schon kurz angesprochenen CUDA-OpenGL-Austausch. Durch das Verwenden dieses Verfahrens ist kein Kopieren von GPU auf CPU mehr nötig. OpenGL hat so direkten Zugriff auf die Daten innerhalb CUDA-Arrays. Die Interaktion verläuft über Buffer-Objekte (PBO, VBO), die von OpenGL angelegt und von CUDA registriert werden. Um einen Buffer in einem Kernel zu verwenden, muss dieser auf einen entsprechenden Pointer gemappt werden. Soll OpenGL wieder Zugriff erlangen, so muss das *Mapping* (Abbildung) des Buffers wieder rückgängig gemacht werden. [DSM]

Die folgenden CUDA-Funktionen sind für die Interaktion bereit gestellt:

```
1 //erstelle bufferObj
2 cudaGLRegisterBufferObject(bufferObj);
3 cudaGLMapBufferObject((void**)&devPtr,bufferObj);
4 //benutzt bufferObj in kernel
5 cudaGLUnmapBufferObject(bufferObj);
6 //benutzt bufferObj mit opengl
7 cudaGLUnregisterBuffer(bufferObj);
```

Listing 15: cuda-opengl-interaction [DSM]

7.2 Genereller Ausblick

Generell kann man sagen, dass Grafikkarten in Zukunft immer wichtiger für parallelisierbare Algorithmen abseits der Bilddarstellung werden und somit der Bereich GPGPU weiter voran getrieben wird.

Dass durch Optimierungen von GPU-Architekturen sowie kleiner beziehungsweise billiger werdenden Bauteilen immer noch Luft nach oben ist,

hat Nvidia erst kürzlich mit der Veröffentlichung der Geforce GTX 970 und Geforce GTX 980 gezeigt.

Auf dem mobilen Markt hat Nvidia 2014 mit dem TEGRA K1 einen Chip geschaffen, der ernst zu nehmende Grafikpower auf Tablets und Handys bringt. Er kombiniert eine ARM-CPU mit einer Kepler-GPU. [MG14]

Diese Kombinationen können in Zukunft auch auf stationären Computern zum Einsatz kommen. GPUs haben in den letzten Jahren eine extrem rasante Entwicklung durchgemacht, wobei CPUs nur sehr langsam voran geschritten sind. In Zukunft könnte sich der Fokus also ändern und auf Weiterentwicklung von CPUs oder Hybrid-Systemen, also Kombinationen aus CPU und GPU (auch APU, *accelerated processing unit* genannt), verschieben. [MG14]

Physik-Engines werden in Zukunft umfangreicher und ihre Simulationen werden schneller und genauer. Da sich an den physikalischen Grundlagen wohl kaum noch etwas ändern wird, stehen die Nähe zur Realität der resultierenden Effekte und deren möglichst schnelle Berechnung im Fokus der Entwicklung. “PhysX“ von Nvidia war eine der ersten Physik-Engine, die GPU-Beschleunigung bei ihren Berechnungen nutzte. Mittlerweile sind auch andere Engines wie “Havok Physics“ und “Bullet Physics Library“ nachgezogen.

A Anhang

A.1 UML-Diagramme

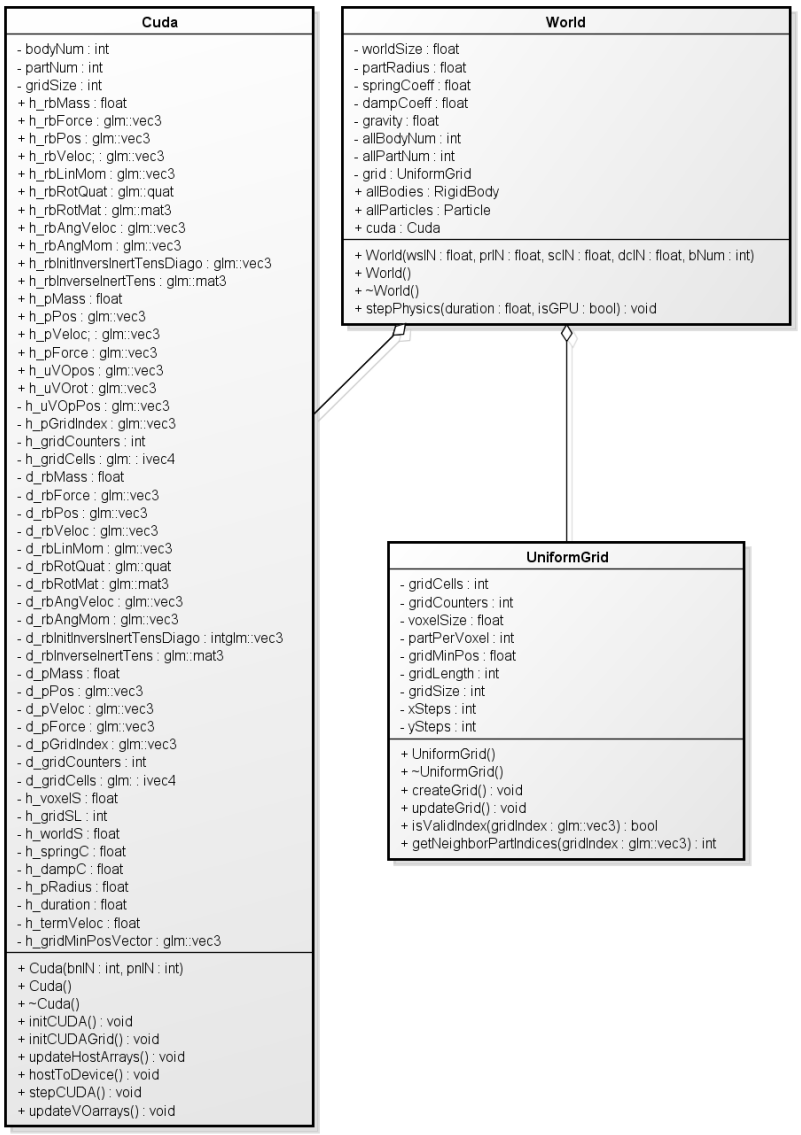


Abbildung 32: Genaue Klassendefinitionen der Engine (1)
die Klassen “World“, “Cuda“ und “UniformGrid“

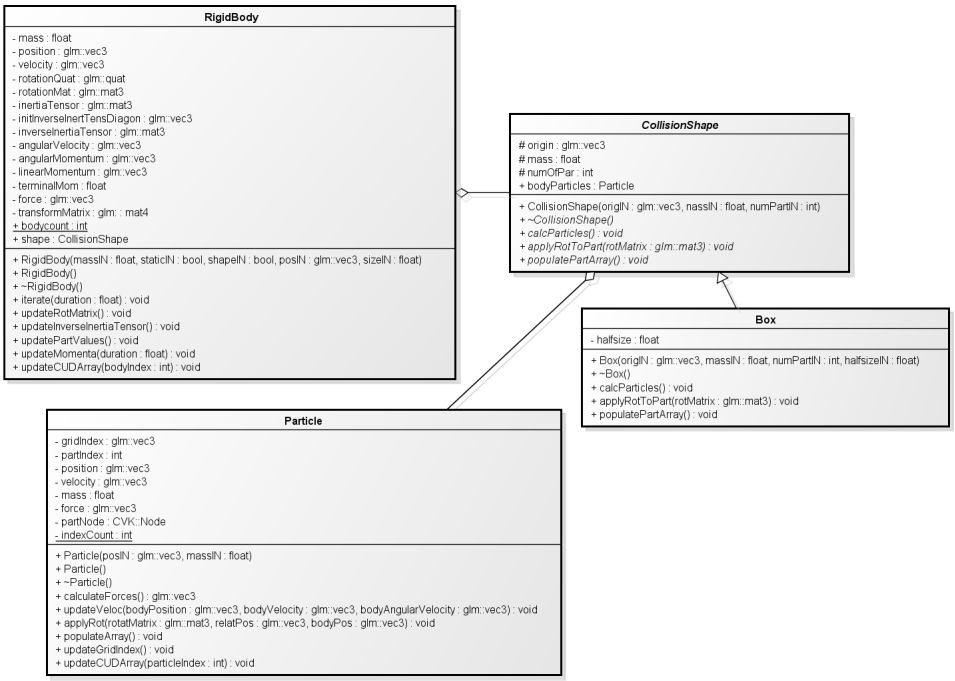


Abbildung 33: Genaue Klassendefinitionen der Engine (2)
die Klassen “RigidBody“, “Particle“, “CollisionShape“ und “Box“

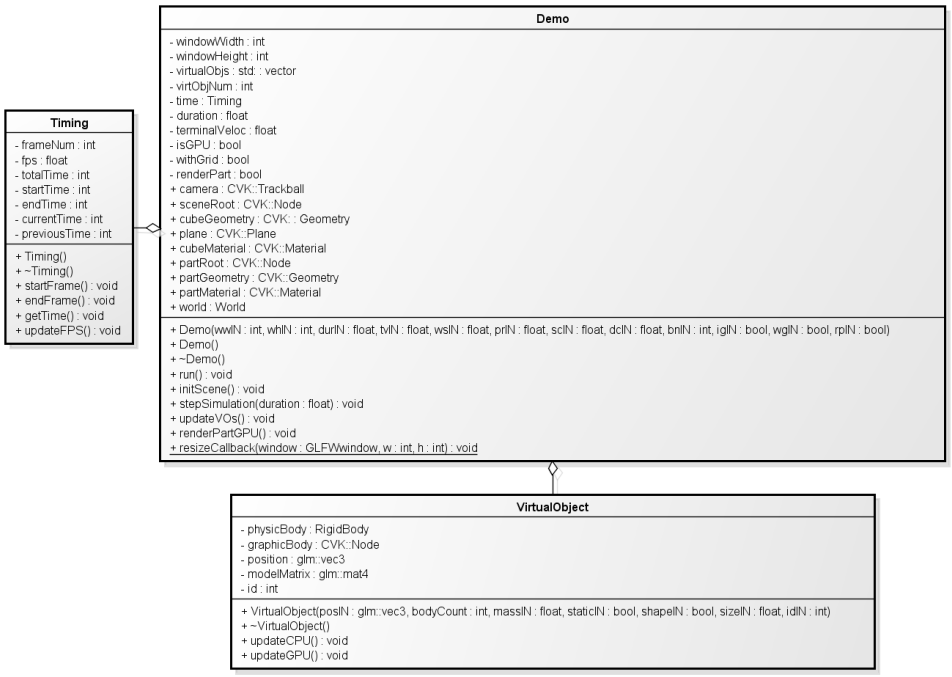


Abbildung 34: Genaue Klassendefinitionen der Engine (3)
die Klassen “Demo“, “Timing“ und “VirtualObjekt“

Abbildung 35: Genaue Klassendefinitionen der Engine (4) die CUDA-Dateien “kernels“ und “kernels_impl“

A.2 Code

A.2.1 CUDA-Programm-Beispiel

Im Download des CUDA-Toolkits sind viele Beispielprogramme enthalten. Im Kapitel 4.3 stehender Code wurde einem dieser Beispiele entnommen (CUDA Samples/v6.5/0_Simple/vectorAdd).

Das Toolkit kann unter `https://developer.nvidia.com/cuda-downloads` heruntergeladen werden.

A.2.2 Code der Implementation

Der Code zur Implementation dieser Arbeit kann unter dem folgenden Link eingesehen werden:

`https://github.com/traubenzucker3001/RBPE-source/`

Eine Weiterleitung zur Dokumentation ist dort auch zu finden.

Literatur

[DDCB00] Debunne, Gilles ; Desbrun, Mathieu ; Cani, Marie-Paule ; Barr, Alan H.: Adaptive Simulation of Soft Bodies in Real-Time. In: *Computer Animation 2000, April 2000*. Philadelphia, Etats-Unis, Juni 2000, S. 133–144. – online avalible at `http://www-ljk.imag.fr/Publications/Basilic/com.lmc.publi.PUBLI_Inproceedings%401172c0fd434_1e6e6d8/`, last accessed: 21.01.2015

[DSM] Durant, Luke ; Szalay, Tamas ; McClellan, Russell: *CUDA Particle System*. GPU Programming, California Institute of Technology, . – online avalible at `http://courses.cms.caltech.edu/cs101gpu/lec9_rec4.pdf`, last accessed: 21.01.2015

[Ebe04] Eberly, David H.: *Game Physics*. Morgan Kaufmann Publishers, 2004 (Series in interactive 3D Technology). – ISBN 1–55860–740–4

[Eri05] Ericson, Christer: *Real Time Collision Detection*. Morgan Kaufmann Publishers, 2005 (Series in interactive 3D Technology). – ISBN 1–55860–732–3

[Fos11] Fossum, Fredrick: *Real-Time Rigid Body Interactions*, NTNU: Norwegian University of Science and Technology, Diplomarbeit, 2011. – online avalible at `http://daim.idi.ntnu.no/masteroppgaver/006/6230/masteroppgave.pdf`, last accessed: 21.01.2015

[Geo13] Georgiadis, Panos: *How to Install CUDA 5.0 Toolkit in Ubuntu*. Website, August 2013. – online avalible at `http://www.unixmen.com/how-to-install-cuda-5-0-toolkit-in-ubuntu/`, last accessed: 21.01.2015

[Gre10] Green, Simon: *Particle Simulation using CUDA*. NVIDIA Corporation, Mai 2010. – online avalible at `http://www.dps.uibk.ac.at/~cosenza/teaching/gpu/nv_particles.pdf`, last accessed: 21.01.2015

[Har07] Harada, Takahiro: Chapter 29. Real-Time Rigid Body Simulation on GPUs. In: *GPU Gems 3* nVIDIA, Addison-Wesley, 2007. – online avalible at `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch29.html`, last accessed: 21.01.2015

[Kre14] Krehenbrink, Marcel: *Nvidia CUDA*. Proseminar GPGPU Programmierung, Universität Koblenz-Landau, Mai 2014. – online avalible at https://owncloud.uni-koblenz.de/owncloud/public.php?service=files&t=a0fd2e6d6e57188c49fd5acfa0397207&path=/Ausarbeitungen/2_CUDA, last accessed: 21.01.2015

[Kum10] Kummerow, Jakob: *Implementierung und Vergleich weiterer Zeit-Integrationsverfahren IDP "Numerische Aspekte bei Molekulardynamik-Simulationen"*. Technische Universität München, September 2010. – online avalible at http://www5.in.tum.de/pub/kummerow2010.pdf, last accessed: 21.01.2015

[MG14] Müller, Daniel ; Geilen, Nils: *Einführung in die GPU*. Proseminar GPGPU Programmierung, Universität Koblenz-Landau, Mai 2014. – online avalible at https://owncloud.uni-koblenz.de/owncloud/public.php?service=files&t=a0fd2e6d6e57188c49fd5acfa0397207&path=/Ausarbeitungen/1_Einf%C3%BChrung%20GPU, last accessed: 21.01.2015

[Mil07] Millington, Ian: *Game Physics Engine Development*. Morgan Kaufmann Publishers, 2007 (Series in interactive 3D Technology). – ISBN 978–0–12–369471–3

[NVI12] *NVIDIA'S Next Generation CUDA Compute Architecture: Kepler GK110*. NVIDIA Corporation, 2012. – online avalible at http://www.nvidia.de/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, last accessed: 21.01.2015

[Rie03] Riegler, Peter: *Kurzeinführung in die numerische Integration*. Fachhochschule Braunschweig/Wolfenbüttel, September 2003. – online avalible at http://public.fh-wolfenbuettel.de/~riegler/Mathematik3/numint.pdf, last accessed: 21.01.2015

[Vox12] *Voxelisation*. Website, October 2012. – online avalible at https://hydrogen2014imac.wordpress.com/2012/10/30/voxelisation/, last accessed: 21.01.2015

Abbildungsverzeichnis

Listings