

Honoursarbeit

„Entwicklung einer Physiks simulation mithilfe des OpenGL Compute-Shader“

Block Name:	Research Project
Module:	SAE 620
Date Submitted:	21.09.2016
Award Name:	Bachelor of Science(Hons.) Games Programming
Course:	GABP 912
Name:	Tristan Firsching
City:	Munich
Country:	Germany
Staffing:	Andreas Friesecke
Word Count:	ca. 8700
Gewichtung:	60% Theorie /40% Praxis

Eigenständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Inhaltsverzeichnis

1. Einleitung

1.1 These	s. 1
1.2 Problemstellung	s. 1
1.3 Zielsetzung	s. 1
1.4 Industrierelevanz	s. 2
1.5 Zielgruppe	s. 3

2. Grundlagen

2.1 GPU	s. 3
2.2 OpenGL	s. 4
2.3 Compute-Shaders	s. 6
2.4 Physiksimulation	s. 7

3. Methodik

3.1 Vergleichbarkeit	s. 7
3.2 Performance-Tests	s. 8
3.3 Codeentwicklung	s. 9

4. Durchführung

4.1 Programm Struktur	s. 14
4.2 Compute-Shader	s. 18
4.3 Performance-Messung	s. 24

5. Ergebnisse

5.1 Ergebnis der Programmierung	s. 25
5.2 Performance-Tests und Analyse	s. 26
5.3 Gesamt-Analyse	s. 37

6. Zusammenfassung

6.1 Zusammenfassung in Bezug zur These	s. 38
6.2 Kritische Betrachtung der Vorgehensweise	s. 41
6.3 Ausblick auf weitere Forschungen	s. 42

Quellen

Literaturquellen	s.43
Internetquellen	s.43
Abbildungsverzeichnis	s.43

1. Einleitung

1.1 These

Mithilfe eines OpenGL Compute-Shaders ist es möglich eine 2D Physiks simulation mit Kollisionserkennung zu entwickeln, die performanter als eine C++ basierte Lösung ist.

1.2 Problemstellung

Physiksimulationen mit vielen beweglichen Objekten sind oft eines der rechenintensivsten Prozesse innerhalb eines Spieles. Da die Grafikkarte dafür optimiert ist, eine große Menge an Float- und Vektorberechnungen möglichst schnell zu lösen, liegt es nahe, auch die Physikberechnung auf den Grafikprozessor auszulagern. Eine solche Lösung könnte den Prozessor entlasten und es ermöglichen, eine größere Anzahl von Physikobjekten gleichzeitig darzustellen.

1.3 Zielsetzung

Die Ausgangslage dieser Arbeit ist die Programmierung einer Zweidimensionalen Physiks simulation unter C++.

Die Theorie hinter der Simulation und Entwicklung des Programms wird dabei beschrieben und erklärt. Ebenso wird auf die Durchführung der wichtigen Schritte in der Entwicklung eingegangen. Es werden andere mögliche Lösungen angesprochen sowie deren Vor- und Nachteile.

In dieser Simulation bewegen sich Festkörper, die miteinander kollidieren, in einem zweidimensionalen Raum. Die Kollisionsmodelle werden als Rechtecke dargestellt. Die Schwerkraft zieht diese Rechtecke an.

Nachdem eine Lösung gefunden wurde, die Physiks simulation allein über C++ zu realisieren, wird eine Möglichkeit gesucht, den Großteil der Berechnungen auf den Grafikprozessor auszulagern.

Mit diesem Outsourcing der Rechenleistung soll eine bessere Performance erlangt werden, die auf eine andere Art nicht möglich wäre.

OpenGL ermöglicht die Auslagerung der Berechnungen auf den Grafikprozessor mit dem GLSL Compute-Shader.

Nach Implementierung des GLSL Compute-Shaders in die Programmierung, werden eine

Reihe von Tests ausgeführt. Es werden Praktikabilität und Performance dieser Technik gegenüber einer C++ basierten Lösung in verschiedenen Situationen untersucht und die verschiedenen Lösungen miteinander verglichen. Dabei wird erhofft, dass der Weg über den Compute-Shader eine deutliche Verbesserung der Performance erreicht.

Als Schlussfolgerung wird untersucht, wie praktikabel eine Physiks simulation mithilfe eines Compute-Shaders für verschiedene Anwendungsfälle ist.

1.4 Industrierelevanz

Physikberechnungen müssen möglichst stark optimiert werden, damit dieser Prozess die Spiele-Engine nicht überfordert. Es ist bisher üblich, Physiks simulationen auf dem CPU zu bearbeiten. Neuere Grafikprozessoren sind mittlerweile in der Lage, generelle Berechnungen zu übernehmen und den CPU dadurch zu entlasten.

Bei der Entscheidung, auf welche Art die Physikberechnung umgesetzt wird, ist es wichtig, die konkreten Vorteile und Performance-Unterschiede der zwei Möglichkeiten zu kennen.

Der Grafikprozessor (Graphics Processing Unit, GPU) ist ein spezialisierter Prozessor für die Verarbeitung von Grafikberechnungen. Durch die stark parallele Struktur ist der GPU effektiver bei der Verarbeitung von großen Mengen an sich wiederholenden Aufgaben, während der CPU sequenzielle Aufgaben schneller verarbeiten kann.

Die Entwicklung von Shaderprogrammen für generelle Berechnungen, beinhaltet aber auch eigene Schwierigkeiten. Es gibt weniger Tools, die das Debuggen von Shadercode ermöglichen. Das kann die Entwicklung verlangsamen. Durch die hohe Parallelisierung sind stark sequenzielle oder verzweigte Programme weniger effizient. Dazu ist der Speicher zwischen GPU und CPU getrennt, wodurch Datenaustausch nur indirekt möglich ist.

Während Grafikprozessoren dafür ausgelegt sind große Mengen an Polygon- und Farbwertberechnungen zu verwerten, ist es jedoch auch möglich die Rechenleistung für andere Zwecke zu nutzen. Die gängigen APIs unterstützen spezielle Shader, die es ermöglichen den GPU für andere Anwendungen zu programmieren. Dies kann genutzt werden, um dafür geeignete Prozesse einer Game-Engine zu beschleunigen. Die Physikberechnung bietet sich dafür besonders an, da diese eine große Menge an Positionsdaten vergleichen muss, sowie Vektorberechnungen verarbeitet. Die Berechnungen ähneln denen der Vertexberechnungen, für die der GPU normalerweise zuständig ist und scheinen damit gut geeignet, für die Berechnung auf dem Grafikprozessor.

Für einen Entwickler, der sich dafür entscheidet die Physikberechnung auf den GPU auszulagern, ist es wichtig, sich vorher über die möglichen Probleme und konkreten Vorteile einer solchen Lösung im Klaren zu sein.

Performance ist besonders bei der Spielentwicklung ein konstantes Problem. Wenn man besonders komplexe Szenen darstellen, oder die Hardwareanforderungen möglichst niedrig halten will, ist es immer wichtig, verschiedene Prozesse auf die effizienteste Art zu verarbeiten.

1.5 Zielgruppe

Physikberechnung in einer Engine ist eine der rechenaufwendigsten Prozesse. Optimierung der Ressourcen und Kapazitäten eines Rechners sind daher unerlässlich. Diese Arbeit untersucht die Möglichkeit einer Optimierung durch GPU-Beschleunigung.

Engine-Entwickler sowie Programmierer, die also eine volle Physik-Engine von Grund auf aufbauen möchten oder eine existierende Engine erweitern wollen, finden hier Lösungsmöglichkeiten. Denn bei der Entscheidung, ob zusätzliche Zeit in die Entwicklung GPU-basierter Berechnung investiert werden soll, ist es wichtig, die genauen Vorteile und Problematiken zu kennen. Es werden eine genaue Herangehensweise und andere Möglichkeiten beschrieben, sodass eine informierte Entscheidung getroffen werden kann.

Es werden jedoch nicht die komplexeren Prozesse einer voll ausgereiften Simulation besprochen. Ziel ist es, lediglich das Grundgerüst zu erstellen und einen Weg zu finden, dieses mithilfe von GPU-Berechnung zu optimieren.

Entwickler, die einen Compute-Shader für andere Zwecke implementieren wollen, finden in dieser Arbeit ebenso nützliche Informationen.

2. Grundlagen

2.1 GPU

Der Grafikprozessor (Graphics Processing Unit oder GPU) ist ein spezialisierter, separater Prozessor für die Verarbeitung von Bildinformation und 3D Vektorberechnungen. Diese Prozessoren, wurden ursprünglich benutzt um das Darstellen von 3D Grafik zu beschleunigen und den Hauptprozessor (CPU) zu entlasten. (siehe Quelle 1)

Mit Prozessen, die direkt in die Hardware eingebaut wurden, waren sie deutlich schneller als der universell angelegte CPU. Mit der Zeit wurden Grafikprozessoren immer fortgeschrittener und die Anzahl an unterstützten Features wuchs.

Quelle 1: <http://www.nvidia.com/object/gpu.html>

Heutige GPUs sind besonders darauf spezialisiert, komplexe 3D-Szenen in Echtzeit zu berechnen. Um große Auflösungen zu unterstützen müssen GPUs Millionen von Bildpunkten im Bruchteil einer Sekunde bearbeiten. Dies kann nur erreicht werden, wenn diese Prozessoren darauf ausgelegt sind, Aufgaben parallel auszuführen.

Ein GPU beinhaltet mehrere programmierbare Recheneinheiten namens Hardware-Shader. Die Programme, die diese Einheiten ausführen sind die Shader. Hardware-Shader sind Koprozessoren, die darauf ausgelegt sind, parallel zu arbeiten. So können sie ein Shader-Programm teilweise mehrmals gleichzeitig mit anderen Werten ausführen und so deutlich schneller arbeiten.

Durch dieses hohe Maß an Parallelisierung sind GPUs in reiner Rechenleistung den CPUs bis heute überlegen. Allerdings sind sie weniger effizient bei Sequenziellen Aufgaben. Trotzdem ist es möglich die bessere Rechenleistung für andere Anwendungen als 3D-Berechnungen zu nutzen. Um das zu ermöglichen, unterstützen moderne Grafikkarten spezielle Shaderprogramme, die unabhängig von den normalen Grafikprozessen aufgerufen werden. Diese sind die Compute-Shader.

Arbeitsspeicher

Der GPU verfügt wie der CPU über seinen eigenen Arbeitsspeicher. Dieser wird VRAM (Video Random-Access Memory) genannt während der generelle Arbeitsspeicher RAM (Random-Access Memory) genannt wird. Beide Prozessoren benutzen diesen Speicher, da Festplatten eine sehr begrenzte Lesegeschwindigkeit haben. Random-Access Memory benutzt Speichermodule mit weniger Kapazität, weniger Dauerhaftigkeit aber deutlich schnellere Lese- und Schreibgeschwindigkeit. Ohne diesen Arbeitsspeicher könnte der jeweilige Prozessor die benötigten Daten nicht schnell genug erhalten und könnte so nicht effektiv arbeiten.

Ein separater VRAM wird benötigt, da die Austauschrate zwischen GPU und RAM begrenzt ist. Da Grafikprozessoren normalerweise Teil einer Erweiterungskarte sind, muss dieser eine Schnittstelle (PCIe) mit begrenzter Bandbreite nutzen. Deswegen werden große Dateien wie Texturen vor der Verwendung direkt auf dem VRAM-Speicher hochgeladen, um dem Grafikprozessor sofortigen Zugriff zu ermöglichen.

2.2 OpenGL

OpenGL ist eine open-source Grafikbibliothek für die Entwicklung von 3D- und 2D-Applikationen. Es bietet ein hardwarenahes Framework für alle Arten von Grafikanwendungen. Mit Unterstützung eines Großteils an Hardware und allen gängigen Plattformen, ist OpenGL eines der meist benutzten Grafik-APIs.

Ein großer Pluspunkt ist auch, dass OpenGL mit verschiedenen Programmiersprachen verwendet werden.

Der größte Konkurrent zu OpenGL ist DirectX, welches von Microsoft entwickelt wird. Im Gegensatz zu DirectX ist OpenGL plattformübergreifend, erweiterbar und bietet eine

Vielzahl an Extensions für spezialisierte Aufgaben an. Allerdings unterstützt es nur Features für Grafik, wobei DirectX Sound, Input, Networking und weiteres unterstützt. (siehe Quelle 2)

OpenGL wird heute von der Khronos Group entwickelt, eine Zusammensetzung verschiedener Hardware und Software Entwickler. Neue Versionen werden regelmäßig veröffentlicht, die das API um neue Funktionen erweitert. Der Quellcode ist öffentlich. Das ermöglicht die Entwicklung und Veröffentlichung zusätzlicher Erweiterungen, durch unabhängige Entwickler und Programmierer.

Um die Entwicklung bestimmter Aspekte zu vereinfachen, können Toolkits verwendet werden. Diese sind öffentlich erhältliche Bibliotheken, welche jeweils auf verschiedene Anwendungen spezialisiert sind. So gibt es SDL oder Allegro, welche sich für 2D Spieleentwicklung eignen. GLFW und Freeglut vereinfachen Fenstererstellung und Input-Behandlung.

Andere Bibliotheken wie GLEW helfen dabei, weitere unterstützte Extensions und Funktionen, zu laden. Viele dieser Erweiterungsbibliotheken sind notwendig, um möglichst effektiv OpenGL zu benutzen.

OpenGL erlaubt es dem Programmierer auf die Grafikkarte zuzugreifen und diese zu programmieren. Dafür hat es eine eigene Shadersprache GLSL. Diese basiert auf C, enthält aber zusätzlich eingebaute Typen für Vektoren, Matrizen und mehr. Allerdings werden andere fortgeschrittenere Features wie Funktionüberladung und Argumenttypen nicht unterstützt (siehe Quelle 3)

Die OpenGL Shading Language (GLSL) wird benutzt, um die programmierbaren Prozessoren im GPU namens Shader zu steuern. OpenGL bietet dabei Funktionen an, die GLSL-Code kompilieren und per Aufruf ausführen kann. (siehe Quelle 3)

Quelle 2 : OpenGL Game Programming, Seite 9-12

Quelle 3 : OpenGL Shading Language, seite 33-35

2.3 Compute-Shader

Der GPU lässt sich mithilfe von Shadern programmieren. Sie sind so genannt, weil sie ursprünglich für Berechnung von Licht- und Schatteneffekten entwickelt wurden. Heute gibt es spezialisierte Shader, die für jeden Schritt der 3D-Rendering-Pipeline zuständig sind. Die Rendering-Pipeline ist die Sequenz von Arbeitsschritten, die ausgeführt werden um eine 3d-Szene zu als Projektion auf einem Bildschirm anzuzeigen. Die Reihenfolge der einzelnen Schritte in der Pipeline sind fest und nicht veränderbar, da die Hardware speziell gebaut ist, um diese Schritte so effizient wie möglich zu bearbeiten. Moderne Grafikchips unterstützen allerdings zusätzliche Pipeline-Schritte und spezialisierte Shader, die nach belieben genutzt oder übersprungen werden können.

Da aber der Grafikprozessor auch nützlich für andere Anwendungen als 3D-Rendering ist, erlauben es APIs wie OpenGL spezielle Shaderprogramme außerhalb der Grafik-Pipeline auszuführen. Diese sind Compute-Shader, die es einem Programmierer erlauben, die Grafikkarte für generelle Berechnungen zu nutzen (siehe Quelle 4)

Kompatibilität

Shader wie Tessellation-Shader und Compute-Shader sind Features, die für Fortgeschrittene Grafikkarten entwickelt wurden und sind deshalb nicht kompatibel mit älterer Hardware.

Compute-Shader sind erst ab der Version OpenGL 4.3 verfügbar (siehe Quelle 5) und nur mit Grafikkarten Kompatibel diese Version von OpenGL unterstützen.

Wenn Compute-Shader also eine nicht-optionale Rolle in der Engine übernehmen sollen, ist es wichtig zu wissen, dass das die Kompatibilität des Programms einschränkt.

Quelle 4: https://www.opengl.org/wiki/Compute_Shader

Quelle 5: <https://www.opengl.org/registry/>

2.4 Physiksimulation

Die Simulation soll bewegliche Objekte darstellen, die sich möglichst natürlich bewegen und kollidieren. Dazu müssen wir uns zuerst die Theorie und Physik hinter einfacher Bewegung anschauen und Möglichkeiten diese umzusetzen finden.

Geschwindigkeit und Krafteinwirkung

Schwerkraft wirkt auf alle Objekte proportional zu deren Masse. Das bedeutet, dass jedes Objekt mit der selben Beschleunigung nach unten bewegt wird. Dieses lässt sich über eine einfache Vektoraddition des Geschwindigkeitsvektors mit dem Schwerkraftvektors simulieren. Je größer der addierte Vektor, desto größer die addierte Kraft.

Um Schwerkraft zu repräsentieren, kann ein einfacher nach unten zeigender Vektor benutzt werden. Da Schwerkraft konstant ist, muss dieselbe Vektoraddition mit jeder Aktualisierung ausgeführt werden.

Um eine dynamische Aktualisierungsrate zu unterstützen, muss die vergangene Zeit, seit der letzten Aktualisierung gemessen werden. Wenn diese mit den Bewegungsvektoren multipliziert wird, verhalten sich die Geschwindigkeiten invertiert proportional zu der Aktualisierungsrate. Auch bei konstanten Kräften muss dieses beachtet werden, und auf die selbe Art die Krafteinwirkung der Schwerkraft multipliziert werden. So kann versichert werden, dass Objekte sich mit derselben Geschwindigkeit bewegen, unabhängig davon, wie schnell der Computer die Aktualisierung durchführt.

3. Methodik

3.1 Vergleichbarkeit

Das Ziel der Arbeit ist es, die zwei verschiedenen Lösungen zu vergleichen. Um beide Lösungen vergleichbar zu machen, müssen sie im selben Programm eingebaut sein.

Shaderprogramme sind immer abhängig von einem anderen Programm, das diese aufruft. Anders als ein C++ Programm, kann es nicht unabhängig gestartet werden. Es braucht ein Programm, das auf das OpenGL-API zugreift, um dieses zu laden und zu starten. Das heißt ein Compute-Shader kann nicht ein CPU-basiertes Programm ersetzen. Stattdessen kann es einen bestimmten Aspekt der Physiksimulation bearbeiten und die Ergebnisse an das Hauptprogramm weitergeben.

Daher bietet es sich an, ein Programm zu entwickeln, das zwei verschiedene Modi hat. Der ausgewählte Modus kann dann an bestimmten Stellen entscheiden, ob eine einfache Funktion ausgeführt wird oder das Compute-Shaderprogramm zu starten. So kann eine Funktion einmal als C++ Programm oder einmal als Shaderprogramm erstellt werden. Dadurch bearbeiten beide Modi denselben Prozess, nur auf einem anderen Prozessor. Dann können die Performanceunterschiede zwischen beiden Lösungen direkt verglichen werden.

Was auch verglichen wird, ist der Aufwand der Umsetzung. Dazu wird der Prozess der Entwicklung und alle wichtigen Schritte beschrieben.

3.2 Performance-Tests

Die Performance beschreibt, wie schnell der Computer die Prozesse abarbeiten kann. Diese variiert mit verschiedener Hardware und der Anzahl an Berechnungen, die das Programm mit jeder Aktualisierungsschleife ausführen muss.

Um die Performance zu ermitteln kann die Zeit gemessen werden, die das Programm braucht, um eine Schleife fertigzustellen. Mit dieser Zeitmessung (delta time) lässt sich die Bildrate (frame rate) errechnen, die angibt wie viele Bilder pro Sekunde generiert werden.

$$\text{FrameRate} = 1 / \text{DeltaTime}$$

Diese variiert allerdings leicht, entweder weil das Programm mit jeder Aktualisierung unterschiedlich viele Berechnungen verarbeitet, oder Hintergrundprozesse des Betriebssystems die Performance beeinflussen.

Zusätzlich haben moderne Prozessoren variable Taktfrequenzen, die sich den Anforderungen und der Temperatur anpassen.

Es ist also leichte Variation zu erwarten, wodurch es Sinn macht mehrere Messungen zu einem Durchschnitt zusammen zurechnen.

Das Programm benutzt Deltatime auch dazu, um Bewegungsgeschwindigkeiten an die Updaterate anzupassen. So bleiben die Geschwindigkeiten unabhängig von Framerate gleich. Allerdings könnte das leichte Unregelmäßigkeiten verursachen, da teilweise bei niedrigeren Framerates mehr Kollisionen abgearbeitet werden müssen. Um dies zu verhindern, wird eine Funktion eingebaut, die nach Wahl die Deltatime ignoriert und Geschwindigkeiten abhängig von der Bildrate macht. So werden bei jeder Simulation genau die gleichen Berechnungen ausgeführt, solange andere Parameter gleich bleiben.

Bei dem Arbeiten mit dem Grafikprozessor kommen noch mehr Faktoren hinzu. Dadurch dass der GPU über seinen eigenen Arbeitsspeicher verfügt und nicht direkt auf dem RAM-Speicher zugreifen kann, müssen die Daten mit jeder Aktualisierung ausgetauscht werden. Die Schnittstelle für diesen Datentransfer hat allerdings eine begrenzte Geschwindigkeit. Wenn also zu große Datenmengen ausgetauscht werden, kann das andere Prozesse verzögern und so die Leistung verschlechtern.

Da der GPU ein separater Prozessor ist, kann es auch zu Verzögerungen kommen, wenn nur einer der Prozessoren überfordert ist zu einer Verzögerung des ganzen Programm geben.

Um genauere Informationen über diese einzelnen Faktoren zu erhalten, werden zusätzliche Messungen an verschiedenen Punkten des Programms ausgeführt. So können genauere Schlussfolgerungen über die Auslastung der einzelnen Komponenten gezogen werden.

Zusätzlich können verschiedene Anforderungen mithilfe von zusätzlichen, überflüssigen Befehlen simuliert werden. So kann z.B. die Kollisionsabfrage mehrfach ausgeführt werden, um einen komplexeren Algorithmus zu simulieren. Oder der Speicher kann mehrmals mit den selben Werten beschrieben werden, um den Einfluss auf die Gesamtperformance zu untersuchen.

Da sich unterschiedliche Hardware stark in Leistung und Fähigkeiten unterscheiden, ist es auch wichtig, an mehreren Computern zu testen. So können auch Probleme mit Kompatibilität erkannt werden.

3.4 Codeentwicklung

Um das Programm umzusetzen wird OpenGL mit den GLFW, freeglut und glew Erweiterungen verwendet. Diese werden genutzt um OpenGL zu initialisieren und ein Fenster zu erstellen. Um Objekte darzustellen wird ein einfaches Shaderprogramm Polygone rendern.

Sich aktualisierende Programme basieren normalerweise auf einer Endlosschleife. Diese Schleife aktualisiert und rendert kontinuierlich alle Daten, bis das Programm beendet wird.

Für eine einfache Physiksimulation sind drei Schritte notwendig für eine Aktualisierung. Diese müssen für jedes Objekt ausgeführt werden.

Zuerst müssen Kollision gefunden werden. Das heißt alle Objekte müssen miteinander verglichen werden. Bei jedem Vergleich wird überprüft, ob sich zwei Objekte überlappen.

Als nächsten Schritt wird die Kollision simuliert. Zwei Objekte stoßen gegeneinander und wirken mit einer Kraft aufeinander.

Als letzter Schritt werden die Positionen der Objekte aktualisiert.

Wenn diese Schritte fertiggestellt sind, kann das neue Bild erstellt werden und die Schleife neu gestartet werden.

Techniken zur Kollisionserkennung

In der Kollisionsabfrage überprüft man entweder, ob sich zwei Objekte überschneiden oder sich in Zukunft überschneiden werden. Je nach Form der Objekte, die geprüft werden sollen, müssen bestimmte Techniken benutzt werden. Dabei werden jeweils zwei Objekte miteinander verglichen. Um jede Kollision erkennen zu können muss jedes Objekt mit jedem anderen verglichen werden.

Kreiskollision kann am einfachsten untersucht werden, indem man den Abstand der Mittelpunkte mit den Radien beider Objekte vergleicht.

$$\|A-B\| < (r1+r2)$$

Dazu subtrahieren wir die Positionsvektoren beider Objekte, um den Richtungsvektor zwischen den beiden Objekten zu bekommen. Mit dem Betrag kann die Länge dieses Vektors und damit den Abstand ermittelt werden. Wenn dieser kleiner ist als die addierten Radien beider Objekte, überschneiden sich die beiden Kreise.
(Siehe Quelle 6)

AABB Kollision

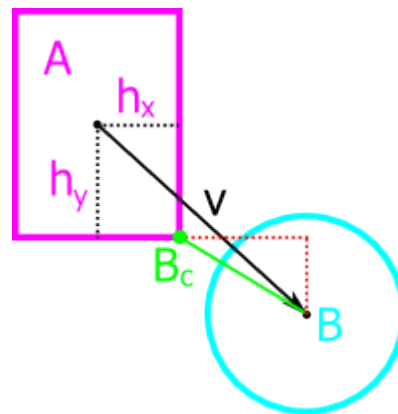


Abb1.

Das AABB Kollisionsalgorithmus erlaubt es, Überlappungen in Rechtecken zu finden. Wenn ermittelt werden soll ob sich zwei Objekte berühren, wird zuerst der Vektor zwischen beiden Mittelpunkten gefunden. In Abbildung 1 ist das der Vektor V . Danach werden die x und y -Werte dieses Vektors auf h_x und h_y begrenzt. Die Längen von h_x und h_y entsprechen der halben Höhe und halben Breite des Rechtecks. Mit dem resultierenden Vektor lässt sich Punkt B_0 auf der Oberfläche des Rechtecks ermitteln. Dieser ist der Punkt auf der Oberfläche des Rechtecks, der dem Mittelpunkt des zweiten Objektes am Nächsten ist. Wenn B_0 sich innerhalb des zweiten Objektes befindet, überlappen sich beide Objekte.

Oriented Bounding Box

Normalerweise wird allerdings eine Oriented Bounding Box anstatt einem einfachen AABB-Algorithmus benutzt. Diese funktioniert nach dem selben Prinzip, beachtet aber auch die Rotation eines Objektes. Um dieses durchzusetzen, müssen die Positionen in das Koordinatensystem des Objektes transformiert werden. So werden alle Positionsvektoren so rotiert, dass der normale AABB Algorithmus angewendet werden kann. Dabei muss beachtet werden, dass die Ergebnisse wie Kollisionspunkte und Normalen wieder in das Hauptkoordinatensystem transformiert werden müssen. (siehe Quelle 6)

Bounding Boxes

Für komplexere Kollisionsmodelle werden oft trotzdem zuerst Boxkollisionsabfragen ausgeführt. Dabei wird die Bounding Box von den größten Ausmaßen des Kollisionsmodell definiert. So können naheliegende Objekte effizienter gefunden werden, bevor komplexere Algorithmen angewendet werden. Mit solchen Detailstufen in Kollisionsmodellen können unnötige Berechnungen verhindert und Performance verbessert werden.

Quelle 6: <http://www.wildbunny.co.uk/blog/2011/04/20/collision-detection-for-dummies/>

Kollisionsreaktion

Nachdem eine Kollision entdeckt wurde, müssen die betroffenen Objekte darauf reagieren.

Ein Objekt, das mit einer Oberfläche kollidiert, wird reflektiert und verliert dabei einen Teil seiner Geschwindigkeit.

Der Geschwindigkeitsverlust kann mit der folgenden Formel berechnet werden.

$$C_R \equiv \frac{v'}{v_0},$$

v_1 ist die Geschwindigkeit vor der Kollision, v' die Geschwindigkeit danach und C_R ist die Elastizität des Objekts. (siehe Quelle 7)

Die Bewegungsrichtung wird dabei reflektiert. Das kann erreicht werden, indem wir den Geschwindigkeitsvektor anhand der Normale der getroffenen Oberfläche reflektieren. Da in GLSL die wichtigsten Vektorfunktionen bereits eingebaut sind, können wir einfach die Funktion `reflect(inVector, normal)` nutzen. (siehe Quelle 8)

Überlappung

Reflexion allein kann allerdings auch Probleme erzeugen.

Nach einer Kollision wird die Bewegungsrichtung reflektiert. So bewegen sich die Objekte wieder auseinander. Wenn aber in der nächsten Aktualisierung sich beide Objekte immer noch überlappen, löst das eine zweite ungewollte Kollision aus.

Das kann passieren, wenn Objekte ineinander platziert sind oder auch zufällig bei normalen Kollisionen.

Um das zu verhindern muss bei einer Kollision nicht nur der Geschwindigkeitsvektor, sondern auch die Position angepasst werden. Die Objekte müssen auf dem kürzesten Weg auseinander geschoben werden, sodass die Überlappung aufgelöst wird.

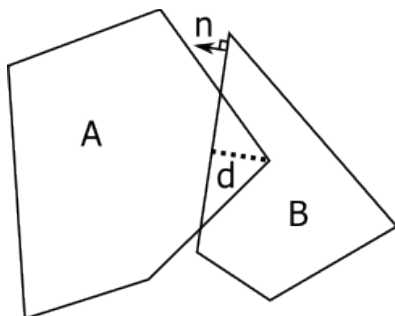


Abb. 2

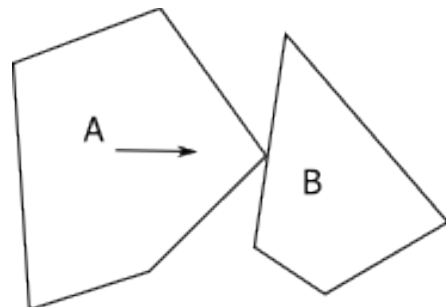


Abb. 3

Um den Status in Abbildung 3 zu erreichen, muss das Objekt A in die Richtung n und um die Strecke d verschoben werden. 'n' ist dabei die Normale der Oberfläche.

$A.Position += n * d$

Es muss also die Position des Objektes A, um den Vektor $n*d$ verschoben werden.

4. Durchführung

4.1 Programm Struktur

Präfixe

Zur besseren Lesbarkeit, werden Präfixe in Form von einfachen Buchstaben vor Variablen gesetzt. Diese repräsentieren die verschiedenen Datentypen.

g	-	vor Opengl Handle, normalerweise ein <code>GLuint</code> , der als Adresse für Opengl-Funktionen verwendet wird.
C	-	vor Klassennamen
f	-	vor Floats
i	-	vor Integers
p	-	vor Pointern

Klassensystem

Das Programm wird objektbasiert aufgebaut sein. Dabei gibt es für jede Aufgabe bestimmte Klassen, die miteinander interagieren sollen.

In der main.cpp Datei wird das Programm gestartet und die Engine-Klasse initialisiert. Die Engine Klasse behandelt die OpenGL Initialisierung, sowie die Initialisierung der Szene und Objekte darin. Außerdem beinhaltet sie die Hauptschleife des Programm, die endlos die Update- und Render-Funktionen wiederholt.

Die Klasse CScene beinhaltet alle Objekte in einem Array und stellt sicher, dass sie alle aktualisiert und gezeichnet werden.

Bewegliche Objekte werden mit in der Klasse CRigidBody behandelt. Diese initialisiert das Objekt und hat Funktionen für Aktualisieren und Zeichnen des einzelnes Objektes. In der Update-Funktion wird die nächste Position des Objektes ermittelt. Dazu wird zuerst überprüft, ob das Objekt mit einem anderen kollidiert oder sich auf dem vorherigen Pfad weiterbewegt.

Präprozessoranweisungen

Präprozessoranweisungen werden dafür benutzt, das Programm zwischen verschiedenen Modi und Optionen zu wechseln. Verschiedene `'#define'` – Anweisungen können in der Uti.h-Datei definiert oder auskommentiert werden, um verschiedene Optionen zu aktivieren oder deaktivieren.

Da diese Anweisungen vor der Kompilierung in Kraft gesetzt werden, haben sie keinen Einfluss auf Performance und werden bei der späteren Messungen kein Störfaktor sein. Zusätzlich werden konstante Werte für Objektzahlen angelegt.

Alle diese Werte werden in der Util.h-Datei abgespeichert, sodass Optionen möglichst schnell geändert werden können.

Update und Rendern

Um einem Programm es zu ermöglichen auf Inputs zu reagieren und sich selbst zu aktualisieren, braucht es eine Aktualisierungsschleife. Diese Schleife wiederholt dieselben Funktionen endlos, bis das Programm beendet wird. Generell werden Berechnungen und Hintergrundprozesse innerhalb einer Updatefunktion und die Berechnung des Bildes innerhalb einer Renderfunktion behandelt.

Die Geschwindigkeit mit der sich die Schleife wiederholt ist dabei abhängig von der Geschwindigkeit des Computers und der Aufwendigkeit der Berechnungen. Das kann allerdings auch zu Problemen führen und Variation an Geschwindigkeiten im Programm erzeugen. Um das zu verhindern müssen Zeitmessung mit jeder Schleife ausgeführt werden. Mit dem Zeitunterschied (DeltaTime) zwischen den letzten Aktualisierungen, können dann Geschwindigkeiten proportional angepasst werden.

Bewegung

Um Bewegung möglich zu machen muss die CRigidBody Klasse Positionsdaten mit jedem Update aktualisieren und diese an den Shader weitergeben bevor das Objekt gerendert wird. Die Bewegungsrichtung muss ebenfalls abgespeichert werden, da nach Newton ein Objekt seine Geschwindigkeit solange behält bis eine Krafteinwirkung dieser entgegensetzt (siehe Quelle 9). Die Position und Geschwindigkeit lassen sich durch Vektoren darstellen. Der Positionsvektor (Position) repräsentiert die x- und y-Werte im globalen Koordinatensystem. Der Geschwindigkeitsvektor (Velocity) ist die Strecke, die der Körper in einer Aktualisierung hinterlegt. So kann die Position durch einfaches Addieren aktualisiert werden.

$\text{Position} += \text{Velocity};$

Um für unterschiedliche Performance auszugleichen kann die Zeit ab dem letzten Frame (DeltaTime) gemessen und mit Velocity multipliziert werden.

$\text{Position} += \text{Velocity} * \text{DeltaTime};$

So bewegen sich die Objekte in der gleichen Geschwindigkeit, unabhängig davon wie schnell aktualisiert wird. (Siehe Quelle 10)

Quelle 9: <http://www.frustfrei-lernen.de/mechanik/newtonsche-gesetze.html>

Quelle 10: <http://www.frustfrei-lernen.de/mechanik/gleichmaessig-beschleunigte-bewegung-physik.html>

Krafteinwirkungen können ebenso mit einer Vektoraddition berechnet werden.

Velocity += Force * Mass;

Diese beeinflusst aber nur den Geschwindigkeitsvektor. Der Effekt ist aber abhängig von der Masse des Körpers, was mit einer einfachen Multiplikation berechnet werden kann. Bei einer konstanten Krafteinwirkung wie Schwerkraft muss auch die Deltatime beachtet werden. (siehe Quelle 8)

Kollision

Wenn sich zwei Objekte berühren, sollen diese sich gegenseitig abstoßen. Um Kollisionen zwischen allen Objekten zu finden, müssen alle Objekte nacheinander verglichen werden.

```
for (int i = 0; i < bodyCount; i++)
{
    for (int k = 0; k < bodyCount; k++)
    {
        // Compares object i to object k
        allBodies[i]->CheckCollision(allBodies[k]);
    }
}
```

Dazu erstellen wir zwei ineinander verschachtelte Schleifen. Diese gehen der Reihe nach durch alle Objekte in dem Array `allBodies[i]` und führt Kollisionsabfragen mit jedem anderen Objekt in der Szene aus.

Allerdings werden auf diese Art unnötige Kollisionsabfragen ausgerufen. Manche Paare werden doppelt überprüft, da sie einmal anstelle von `allBodies[i]` und einmal anstelle von `allBodies[k]`.

```
for (int i = 0; i < bodyCount; i++)
{
    for (int k = i; k < bodyCount; k++)
    {
        // Compares object i to object k
        allBodies[i]->CheckCollision(allBodies[k]);
    }
}
```

Mit dieser Abänderung werden alle bereits überprüften Körper in der zweiten Schleife übersprungen. Dadurch dass `k` mit `i` initialisiert wird überspringt die zweite Schleife alle Objekte die bereits überprüft wurden.

In der Funktion `CheckCollision(allBodies[k])` werden die Daten zweier Objekte verglichen, um zu erkennen ob diese sich berühren oder überlappen.

Wie ermittelt wird, ob sich zwei Objekte überlappen hängt davon ab, welche Formen beide Objekte haben. Es gibt mehrere Algorithmen für verschiedene Formen.

Quelle 8: Game Programming für Dummies, Seite 334-338

Die einfachste Form ist ein Kreis. Wenn beide Objekte eine Kreisform haben, überlappen sie sich wenn der Abstand beider Mittelpunkte kleiner ist als die addierten Radien.

Damit die Kollisionsabfrage nützlich für die Simulation ist muss allerdings auch die Normale der getroffenen Oberfläche ermittelt werden.

Im Fall eines Kreises entspricht die Normale der Oberfläche immer der Richtung des Vektors der von dem Mittelpunkt auf dem Oberflächenpunkt zeigt.

```
normal = normalize(otherColl->position - mCollider.position);
```

Da der Kontaktpunkt auf der Linie zwischen den Mittelpunkten der beiden Kreise liegt, kann stattdessen der Vektor zwischen beiden Mittelpunkten genutzt werden.

Für Kollisionsabfrage von Rechtecken muss ein anderer Algorithmus benutzt werden.

Die AABB Kollision ist eine sehr performancesparende Technik für nicht rotierte Rechtecke. Dabei wird ein Vektor von dem Mittelpunkt des Objektes zu dem zu überprüfenden Punkt ermittelt. Wenn dieser Vektor auf die Ausmaße des Objektes getrimmt wird, erhält man den nächsten Punkt auf der Oberfläche des Objektes.

```
ClosestPoint = getClampedVec2(FromToVec, mCollider.width/2, mCollider.height/2);
```

Bei der Trimmung muss beachtet werden, dass die einzelnen Ausmaße des Vektors und nicht der Vektor als ganzes getrimmt werden soll. Das Vorzeichen muss dabei ignoriert werden. Die maximalen Werte davon entsprechen der halben Höhe und Breite des Rechtecks.

```
fvec2 getClampedVec2(fvec2 _val, float _maxX, float _maxY)
{
    clamp(_val.x, -_maxX, _maxX);
    clamp(_val.y, -_maxY, _maxY);

    return _val;
}
```

Dieser getrimmte Vektor entspricht dem nächsten Punkt auf der Oberfläche des Rechtecks. Wenn sich dieser Punkt innerhalb des anderen Objektes befindet, dann überlappen sich beide Objekte.

Die Normale kann von dem auf dem Punkt zeigenden Vektor abgeleitet werden. Die Achse mit dem größeren absoluten Wert ist die Richtung der Normale.

```
if (abs(normal.x) > abs(normal.y))
{
    normal.y = 0;
}
else normal.x = 0;
```

Das kann erreicht werden, indem man den kleineren Wert auf 0 setzt.

Reflexion

Nachdem eine Kollision entdeckt wurde, wird die Geschwindigkeitsrichtung reflektiert, um einen Stoß zu simulieren.

Anhand der Normale der getroffenen Oberfläche kann der Geschwindigkeitsvektor reflektiert werden.

Die Reflexion wird innerhalb der Rigidbody-Klasse behandelt und erhält Informationen direkt von den Ergebnissen der Kollisionsabfrage.

```
Velocity = glm::reflect(Velocity, normal) *fBounciness;  
Position.x += normal.x * _offset;  
Position.y += normal.y * _offset;
```

Mit der Reflexionsfunktion die 'glm' anbietet, kann der Geschwindigkeitsvektor mithilfe der Normale reflektiert werden. Der neue Geschwindigkeitsvektor wird mit der `fBounciness` Variable skaliert, um Elastizität zu simulieren. `fBounciness` ist dabei eine Eigenschaft jedes Körpers, die bei einem Wert von 1.0 perfekte Elastizität darstellt. Je kleiner dieser Wert, desto mehr Geschwindigkeit wird bei jeder Kollision verloren.

4.2 Compute-Shader

Der Compute-Shader ist ein spezialisierter Shader für generelle Berechnungen. Anders als andere Shadertypen, wird er unabhängig von der Render-Pipeline ausgeführt. Das Shader-Programm wird mithilfe des Befehls `glDispatchCompute(x, y, z)` manuell gestartet. Die `x`, `y` und `z` werte bestimmen dabei wie häufig dasselbe Programm ausgeführt wird.

Invokationen

Compute-Shader sind darauf optimiert ein Shaderprogramm mehrfach und teilweise gleichzeitig auszuführen. Jede Ausführung ist dabei eine Invokation.

Die Invokationen werden in Work Groups unterteilt, welche die kleinste Gruppe an Prozessen ist, die ausgeführt werden kann. Die Reihenfolge nach der die Work Groups abgearbeitet werden ist nicht geordnet. Es ist also wichtig, dass Prozesse unabhängig von einander sind.

Eine Workgroup ruft eine Anzahl an Shader-Invokationen auf, die innerhalb des Shaders als 'local size' der Workgroup definiert werden kann.

Die local size ist dreidimensional, kann aber auch wie (128, 1, 1) definiert werden um eine ein- oder zweidimensionale Liste zu repräsentieren.

Definition der local size in dem Shader:

```
layout(local_size_x = 128, local_size_y = 1, local_size_z = 1) in;
```

Die Workgroups haben ebenso eine mehrdimensionale Größe, die sich innerhalb des Shader-Codes bestimmen lässt.

(siehe Quelle 4)

Inputs und Outputs

Im Shader-Code selbst gibt es allerdings keine Möglichkeit direkt mit anderen Programmen zu kommunizieren. Andere Shadertypen wie Vertexshader haben vordefinierte Inputs und Puffer, auf die zugegriffen werden kann. Im Computer-Shader müssen diese manuell erstellt werden. Andernfalls kann das Programm zwar ausgeführt werden, aber nicht die Ergebnisse weitergegeben werden.

Die effizienteste Art, die Ergebnisse des Compute-Shaders weiterzugeben, ist es sie direkt auf den Grafikspeicher zu schreiben. Dadurch können andere Shader, oder der CPU die Daten auslesen nachdem das Shader-Programm ausgeführt wurde.

Berechnungen mithilfe des Compute-Shaders

Um Shaderprogramme zu laden, erstellen wir eine Klasse 'CShaderProgram'. Von dieser Klasse werden alle verschiedenen Arten von Shaderprogrammen abgeleitet. Hier wird das Laden, Kompilieren und Bindig von Shadern behandelt.

Damit auf die Ergebnisse die im Compute-Shader ermittelt und zugegriffen werden kann, müssen die Daten entweder vom CPU ausgelesen werden oder direkt vom Vertex-Shader benutzt werden. Die effizienteste Art wäre es, die ermittelten Positionen im VRAM-Speicher der Grafikkarte abzuspeichern und damit die Positionen der Objekte im Vertex-Shader zu verschieben. Allerdings hat der CPU dadurch keinen Zugriff auf Positionsdaten. Das kann zu Problemen führen, wenn andere Systeme in der Engine, diese Daten brauchen.

Um die Daten für den CPU sichtbar zu machen, werde ich eine Kopie des GPU-Puffers im Programm erstellen. Beide Versionen werden mit jeder Aktualisierung synchronisiert. So haben sowohl Shaderprogramme und das C++ Programm Zugriff auf dieselben Daten.

Dadurch, dass der Computeshader Invokationen nicht sequenziell abarbeitet, kann die komplette Physiks simulation nicht in einem einzelnen Shader effizient behandelt werden. Die Simulation bearbeitet sequenziell zuerst die Kollisionsabfrage, dann Krafteinwirkung und Positionsaktualisierung. Jeder dieser Schritte ist von den Ergebnissen des vorherigen abhängig und kann deswegen nicht effektiv parallel behandelt werden.

Quelle 4: https://www.opengl.org/wiki/Compute_Shader

Der GPU ist am effizientesten, wenn er eine große Anzahl an kleineren Prozessen gleichzeitig bearbeitet, im Gegensatz zu langen verzweigten Programmen.

Bei der Physiksimulation, auf dem CPU ist der rechenaufwendigste Schritt die Kollisionserkennung mit anderen Objekten. Dabei muss jedes Objekt mit allen anderen in der Szene verglichen werden. Dadurch erhöht sich die Menge an Kollisionstests drastisch mit einer höheren Anzahl an Objekten. Um das auszunutzen, benutze ich einen Compute-Shader, der allein die Kollisionsabfrage behandelt und die Ergebnisse zurückgibt.

Mit dem Befehl `glDispatchCompute(x , y, z)` werden eine Reihe an Invokationen des ausgewählten Shaderprogrammes gestartet. Die Reihe ist dabei ein mehrdimensionaler Array mit den Ausmaßen von x, y und z. Jede mögliche Position in diesem Array repräsentiert eine einzelne Invokation.

Es bietet sich an, einen zweidimensionalen Array an Invokationen zu benutzen, wobei jede Invokation eine Kollisionsabfrage zwischen zwei Objekten behandelt. Dabei bestimmt der x-Wert den Index des ersten Objekts und der y-Wert den des zweiten Objektes.

```
glDispatchCompute(bodyCount , bodyCount, 1);
```

Mit dieser Funktion startet OpenGL `bodyCount*bodyCount` Invokationen des Compute-Shaders. (siehe Quelle 13)

Allerdings werden so mehr Kollisionsabfragen ausgeführt als nötig sind. Jedes Objekt wird der Reihe nach mit jedem anderen verglichen, wodurch einige Paare doppelt überprüft werden. Einmal als erstes Objekt und einmal als zweites. Um das zu verhindern können wir allerdings nicht wie in C++ die bereits behandelten Kombinationen überspringen. Die Anzahl an Invokationen kann nicht geändert werden, während der Shader ausgeführt wird.

AA	BA	CA	DA
AB	BB	CB	DB
AC	BC	CC	DC
AD	BD	CD	DD

Diese Tabelle stellt den zweidimensionalen Array dar und die jeweiligen Objekte, die in jeder Invokation verglichen werden. Rot markiert sind die überflüssigen Vergleiche, die entweder ein Objekt mit sich selbst vergleichen oder bereits von einer anderen Zelle behandelt wird. Welches der Objekte im ersten oder zweiten Platz ist, ist irrelevant, womit entweder nur AB oder nur BA nötig ist.

Quelle 13: <http://wili.cc/blog/opengl-cs.html>

AB	CB	DB
AC	BC	DC
AD	BD	CD

Wenn diese unnötigen Invokationen ausgelassen werden, kann der Array in der X und Y-Dimension um eine Reihe verkleinert werden, wodurch die Performance leicht verbessert werden kann.

```
glDispatchCompute(bodyCount - 1, bodyCount - 1, 1);
```

Mit dem modifizierten Dispatchbefehl werden zwar die korrekte Anzahl an Invokationen verarbeitet, der Shader behandelt aber noch nicht die richtigen Objekte.

Im Shader:

```
YObjectIdx = int(gl_GlobalInvocationID.y) + 1;
```

Um die Oberste Reihe zu überspringen, können wir für jedes Vergleichsobjekt den Index um Eins erhöhen.

Ein Objekt wird jedes mal mit sich selbst verglichen, wenn der X-Wert, dem Y-Wert gleicht. Um diese Fälle auszuschneiden muss der Rest der Vergleiche aufgerückt werden.

```
XObjectIdx = int(gl_GlobalInvocationID.x);  
if(XObjectIdx >= YObjectIdx) XObjectIdx++;
```

Der x-Index wird um einen Wert verschoben, wenn er dem Y-Index gleicht oder größer ist.

Speicher Management

Der GPU hat seinen eigenen Speicher (VRAM), der von dem normalen Arbeitsspeicher (RAM) getrennt ist. Um Daten zwischen dem RAM und dem VRAM zu teilen, müssen diese manuell aktuell gehalten werden.

Für unsere Berechnungen muss der GPU Zugriff auf Positionsdaten und möglicherweise Geschwindigkeitsdaten haben. Und der CPU muss nach dem Dispatch die Ergebnisse der Berechnung von dem VRAM auslesen.

Da der Compute-Shader keine direkten Inputs und Outputs unterstützt, müssen wir Puffer-Objekte auf dem VRAM erstellen, die innerhalb des Shaders gelesen und beschrieben werden können.

Das Shader Storage Buffer Object (ssbo) erlaubt es, einen selbsterstellten Struct auf dem VRAM zu generieren, der beliebig beschrieben und gelesen werden kann.

```
struct S_Shader_data
{
    float Positions[bodyCount * 2];
    float Velocities[bodyCount * 2];
};
```

Dieser Struct repräsentiert den Puffer, der auf den VRAM geladen werden soll. Arrays von Floats können im Shader auch als Vektoren gelesen werden. Vektoren können als Arrays von Floats gespeichert werden, solange die Größe mit den benötigten Zahlen übereinstimmt.

```
layout (std430, binding=1) buffer shader_data
{
    vec2 Positions[BodyCount];
    vec2 Velocities[BodyCount];
};
```

Im Shader muss ein äquivalenter Puffer erstellt werden.

```
glGenBuffers(1, &gShaderBuffer);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, gShaderBuffer);
glBufferData(GL_SHADER_STORAGE_BUFFER, sizeof(shader_data), &shader_data,
GL_DYNAMIC_COPY);
```

Dieser Code generiert den Puffer auf dem VRAM und speichert die Adresse in gShaderBuffer.

```
block_index = glGetProgramResourceIndex(gProgramID, GL_SHADER_STORAGE_BLOCK, "shader_data");
//connect the shader storage block to the gShaderBuffer
GLuint gShaderBuffer_binding_point_index = 1;
glShaderStorageBlockBinding(gProgramID, block_index, gShaderBuffer_binding_point_index);
```

Danach müssen wir die Adresse der Variablen shader_data finden und diese mit dem Shader verbinden.

Output Puffer

Um auf die Ergebnisse der Shaderberechnungen zuzugreifen, müssen diese direkt von dem Puffer ausgelesen werden. Wir erstellen dazu einen separaten Puffer, der alle nötigen Daten für alle Objekte abspeichern kann.

```
struct S_Shader_Output
{
    float vNormals[bodyCount * 2];
    float vHitPoint[bodyCount * 2];
    float fOffset[bodyCount];
    int iHitCollider[bodyCount];
};
```

Wenn eine Überlappung erkannt wurde, wird die Normale der getroffenen Oberfläche

abgespeichert. Es gibt für jedes Objekt eine Normale, auch wenn keine Kollision stattfindet. In diesem Fall bleibt die Normale ein Nullvektor.

Der Hitpoint ist die Position an der das Objekt einen Kontakt mit einem anderen macht. Offset ist der Abstand und zeigt, wie weit das Objekt mit dem anderen überlappt. Zusammen mit dem Hitpoint werden Überlappungen gelöst, indem die Objekte auseinander geschoben werden.

Schließlich wird in iHitCollider der Index des getroffenen Objektes abgespeichert.

Wenn der Shader ausgeführt wird, schreibt er auf diesen Puffer und nachdem alle Berechnungen fertiggestellt wurden, kann dieser ausgelesen werden.

Allerdings muss darauf geachtet werden, dass der CPU und GPU teilweise gleichzeitig arbeiten. Dadurch ist nicht garantiert, dass der Shader nicht noch auf dem Puffer schreibt, wenn dieser ausgelesen werden soll. Dafür gibt es die `glMemoryBarrier()` Funktion, die wartet bis alle Schreibprozesse fertiggestellt sind.

```
glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
```

Diese Funktion wartet spezifisch auf Shader Storage Buffers und sollte jedes mal aufgerufen werden bevor einen Puffer beschrieben oder gelesen wird.

```
glBindBuffer(GL_ARRAY_BUFFER, gShaderOutputBuffer);
glGetBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(shader_output), &shader_output);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Um einen Puffer zu lesen muss dieser erst mit `glBindBuffer()` ausgewählt und dann mit `glGetBufferSubData()` auf einer Adresse im RAM-Speicher kopiert werden.

Zusätzlich muss der Puffer vor jedem Update auf den Originalzustand zurückgesetzt werden, damit vorherige Ergebnisse gelöscht werden. Dazu setzen wir alle Werte zurück auf Null und überschreiben den Puffer auf dem VRAM mit der Kopie auf dem RAM-Speicher.

```
glBindBuffer(GL_SHADER_STORAGE_BUFFER, gShaderOutputBuffer);
GLvoid* p = glMapBuffer(GL_SHADER_STORAGE_BUFFER, GL_WRITE_ONLY);
memcpy(p, &shader_output, sizeof(shader_output));
glUnmapBuffer(GL_SHADER_STORAGE_BUFFER);
```

Dazu muss zuerst die Adresse des Puffers auf dem VRAM mithilfe von `glMapBuffer()` gefunden werden und die Daten direkt an diesen Ort kopiert werden. (siehe Quelle 14)

Quelle 14: <http://www.geeks3d.com/20140704/tutorial-introduction-to-opengl-4-3-shader-storage-buffers-objects-ssbo-demo/>

4.3 Performance-Messung

Um Performance zu vergleichen, muss gemessen werden wie viel Zeit eine Update und Rendschleife in Anspruch nehmen. Dazu kann der gemessene Zeitpunkt am Ende der letzten Schleife mit dem am Anfang der nächsten verglichen werden.

```
f_DeltaTime = glfwGetTime() - f_lastTime;
```

Hier ist `f_lastTime` die Zeit am Anfang der Hauptschleife. Nachdem Update- und Renderfunktionen ausgeführt wurden, wird die Zeit erneut abgefragt. Wenn von dieser Zeit `f_lastTime` abgezogen wird, erhält man die Deltatime. Das ist die Zeit, die die letzte Schleife in Anspruch genommen hat.

Mit einem einfachen 'cout'-Befehl kann diese Zahl in der Konsole ausgegeben werden. Allerdings sind große Mengen an Konsolenausgaben relativ ineffizient und verlangsamen das Programm. Da das Programm mehrere Hundert mal pro Sekunde aktualisiert werden kann, wären die Ergebnisse verfälscht.

Um das zu verhindern, habe ich die Anzahl an Ausgaben reduziert, indem ich nur durchschnittliche Werte ausgabe. Dafür werden die gemessenen Zeiten addiert und nach einer bestimmten Anzahl an Zyklen zusammengerechnet.

```
if (i_elapsedFrames >= i_SampleSize)
{
    double f_AverageLoopTime = f_AddedLoopTime / i_elapsedFrames;

    // Konsolenausgaben Hier

    i_elapsedFrames = 0;
    f_AddedLoopTime = 0.0f;
    f_AddedComputeTime = 0.0f;
}
f_LoopTime = glfwGetTime() - f_lastTime;
f_AddedLoopTime += f_LoopTime;
i_elapsedFrames++;
```

Hier entspricht `i_SampleSize` der Stichprobenanzahl und `f_LoopTime` der Deltatime. Nachdem die Werte ausgegeben werden, setzen sich die Werte zurück und messen den nächsten Durchschnitt.

Mit einer großen Stichprobenanzahl, können auch Unregelmäßigkeiten in den Messungen verringert werden.

5. Ergebnisse

5.1 Ergebnis der Programmierung

Das fertige Programm zeigt mehrere Rechtecke, an die simuliert werden. Dabei können in der Util.h-Datei verschiedene Modi und Eigenschaften bestimmt werden. So lässt sich einstellen wie viele Objekte dargestellt werden sollen und wie schnell sich diese anfangs bewegen.

Andere Einstellungen bestimmen die Kollisionsalgorithmen. Es kann bestimmt werden, ob die Objekte als Rechtecke oder Kreise behandelt werden. Es gibt auch eine Option, die es den Objekten erlaubt zu rotieren. Dies ist aber nicht komplett fertiggestellt und repräsentiert keine realistische Simulation.

Das Programm unterstützt einen GPU-beschleunigten Modus und einen allein auf dem CPU-basierten Modus. Dieser kann vor der Kompilierung eingestellt werden. Der GPU-beschleunigte Modus nutzt einen Compute-Shader, um die Kollisionsabfragen durchzuführen. Die Ergebnisse werden auf einem VRAM-Puffer geschrieben und von dem C++-Programm ausgelesen. Wenn dieser Modus deaktiviert ist, findet die Kollisionsabfrage innerhalb der CRigidbody Klasse statt.

Nach der Kollisionsabfrage werden die Ergebnisse genutzt, um Geschwindigkeits- und Positionensvektoren zu aktualisieren. Dieser Schritt ist gleich, unabhängig von dem ausgewählten Modus.

Um Performance zu vergleichen, werden Aktualisierungszeiten gemessen und die Daten in der Konsole ausgegeben. Um genauere Daten zu ermitteln, sind die Konsolenausgaben durchschnittliche Werte, von einem einer beliebigen Stichprobenanzahl.

Das Bild wird mithilfe von einem einfachen OpenGL Shaderprogramm erstellt. Ein Vertex-Shader zeichnet vorher definierte Vertices an der gewollten Position und ein Fragment-Shader bestimmt die Farbe der Pixel. Das Programm erstellt einen Vertexpuffer und Array, die an das Shaderprogramm gebunden werden. Die Position des Objektes wird als uniform Variable direkt an den Vertex-Shader gegeben. Diese Variable wird genutzt, um die Vertex-Positionen an die gewollte Stelle zu verschieben.

5.2 Performance-Tests und Analyse

Gemessen werden die ersten 500 Frames der Simulation, von denen eine durchschnittliche Framezeit und Framerate ausgerechnet wird.

Die Ergebnisse variieren leicht aufgrund von Hintergrundprozessen.

Es wird auf zwei verschiedenen Computern getestet.

Computer 1 Konfiguration:

CPU: Intel i7 4790K @4,0Ghz

GPU: Nvidia GTX 970 @1,114Ghz

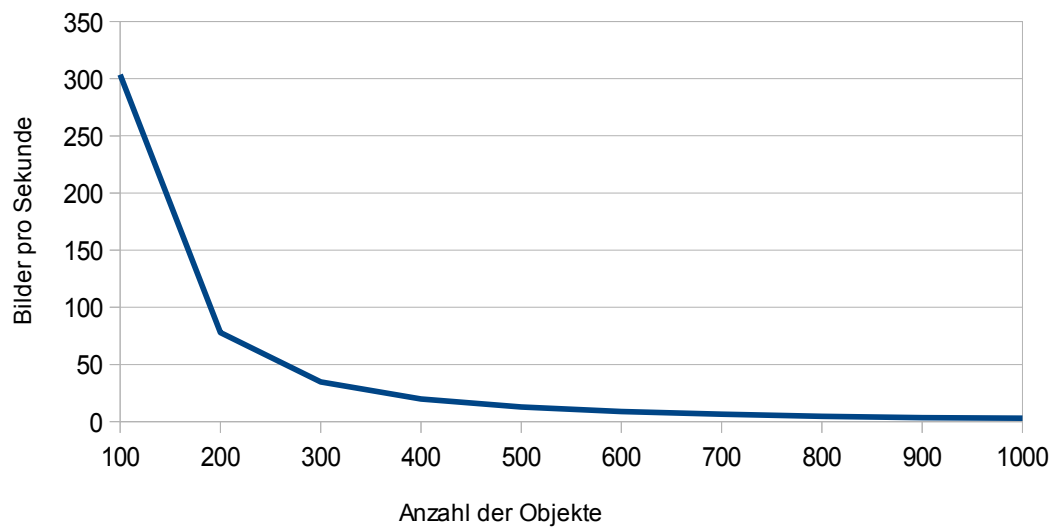
Computer 2 Konfiguration:

CPU: Intel i5-2450M @2.5Ghz

GPU: Nvidia GT 555M

Die meisten der Tests werden auf dem Computer 1 durchgeführt. Die Ergebnisse von Tests auf dem zweiten Computer werden nur angesprochen, wenn relevante Schlussfolgerungen davon geschlossen werden können.

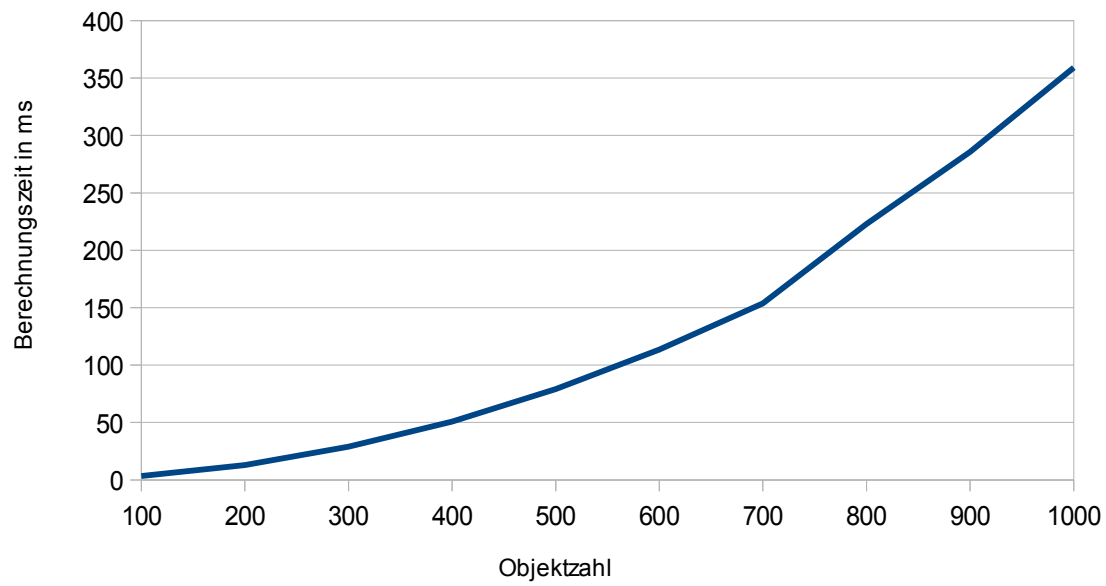
CPU-Berechnung mit Rechteck-Kollision



In diesem Diagramm wird die Bildfrequenz in Relation zu der Anzahl an Objekten angezeigt. Die Berechnung läuft dabei nur über den CPU.

Die Performance verschlechtert sich drastisch mit mehreren Hunderten beweglichen Objekten. Schon bei 400 Körpern wäre mit unter 30 Bildern pro Sekunde die Spielbarkeit eines Spieles eingeschränkt.

Berechnungszeiten



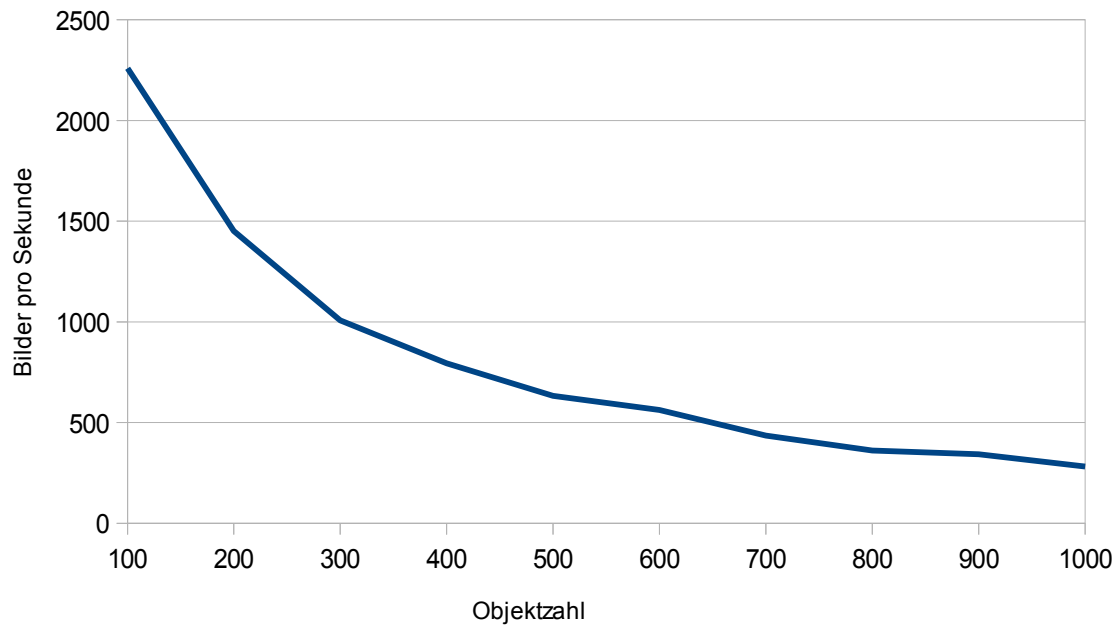
Das Diagramm zeigt die direkte Zeit, die das Programm für eine Aktualisierung braucht. So lässt sich die Relation zwischen Performance und Objektzahl deutlicher vergleichen.

Die Berechnungszeit scheint sich exponentiell zu vergrößern. Das ist erwartet, da jedes Objekt einmal mit jedem anderen verglichen werden muss. So steigt mit jedem neuen Objekt die Anzahl der Kollisionsüberprüfungen um die Gesamtanzahl der Objekte.

GPU-Berechnung

Bildfrequenz

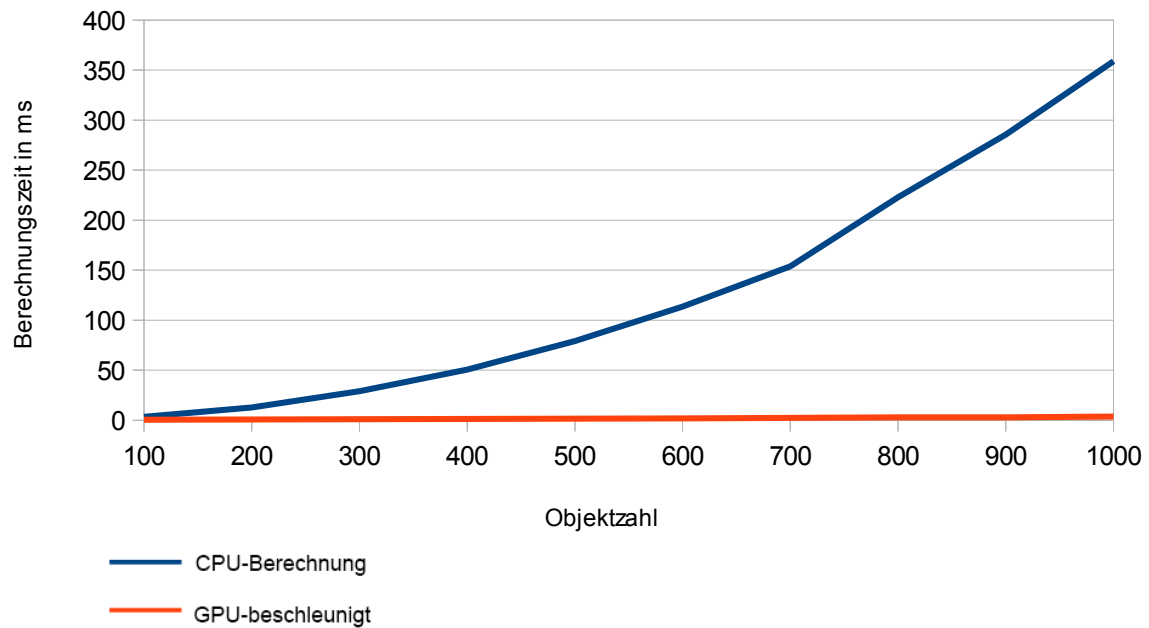
Zum Vergleich habe ich dieselben Tests in dem GPU-beschleunigten Modus ausgeführt. Das heißt hier wird die Kollisionsabfrage mit einem Compute-Shader bearbeitet, um den CPU zu entlasten.



Dieser Test zeigt eine deutliche Verbesserung gegenüber der CPU-Berechnung. Die Kollisionserkennung über den GPU ist wie erwartet erheblich schneller bei hohen Objektzahlen. Aber auch bei der Anzahl bei 100 Objekten zeigt sich eine Verbesserung von über 700%.

Vergleich von CPU gegenüber GPU

Direkter Vergleich

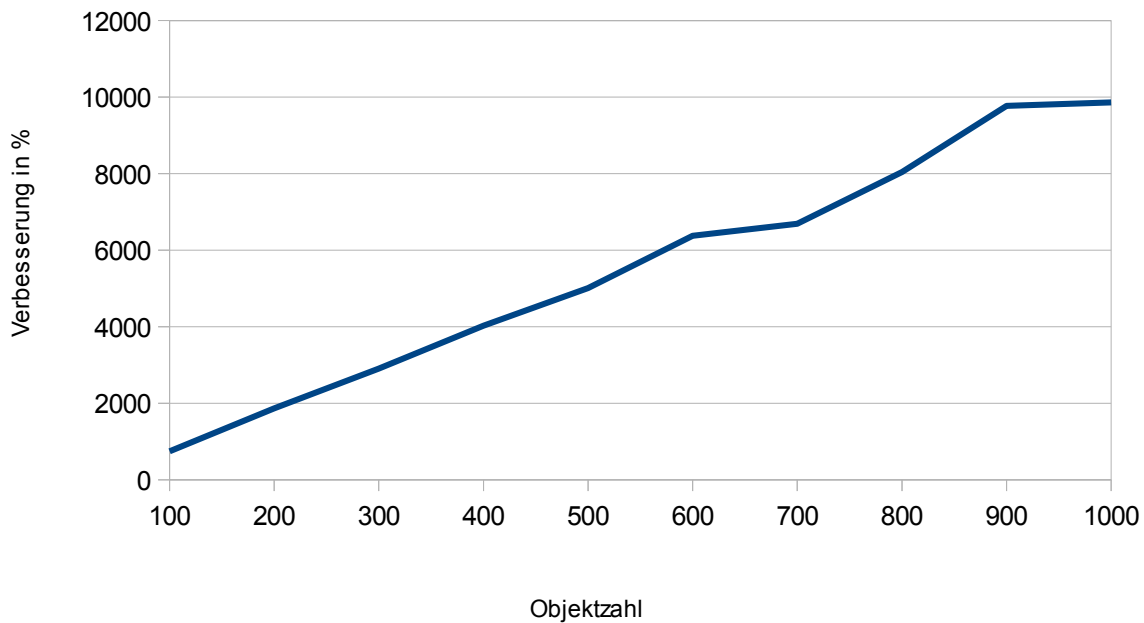


Im direkten Vergleich ist deutlich sichtbar, dass durch die GPU-Beschleunigung deutlich stabilere Performance erreicht werden kann. Selbst wenn wenige Objekte dargestellt werden, liegt die CPU-basierte Lösung hinten. Obwohl der GPU effizienter ist bei großen Anzahlen an Berechnungen, kann der Compute-Shader in allen Tests einen Vorteil erreichen.

Verbesserung in Prozent

Dieser Vergleich zeigt die Performanceverbesserung mit dem Compute-Shader in Prozent, im Gegensatz zu der reinen CPU-Lösung. Diese scheint sich mit höheren Zahlen an Objekten linear zu erhöhen. Obwohl sich beide Lösungen in ihrer Performance verringern, führt die GPU-Lösung immer deutlicher je höher die Anzahl an Objekten.

Mit einer Verbesserung von bis zu 4800% und vermutlich größerer mit größeren bei höheren Objektzahlen als den gemessenen, ist die GPU-Lösung in allen Fällen der deutliche Gewinner.



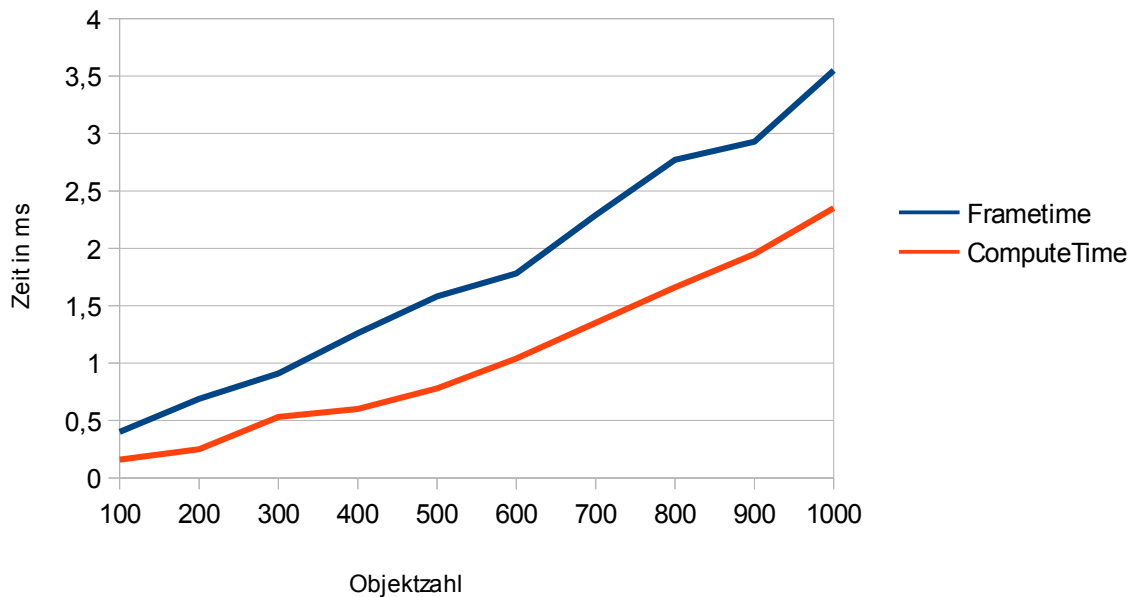
Wenn man die Steigung der Frametime mit der von den Ergebnissen der reinen CPU-Berechnung vergleicht, scheint die Shaderbasierte Lösung linearer zu steigen. Obwohl die Anzahl an Berechnungen exponentiell steigt, ist kaum eine exponentielle Vergrößerung in den Berechnungszeiten zu sehen. Der GPU scheint effektiver zu arbeiten, je größer die Anzahl an Berechnung.

Wie zu sehen ist nimmt die Shader-Berechnung in allen Fällen den Großteil der Zeit in Anspruch.

Anteil der Compute-Shader Berechnung

Um zu erkennen, welcher Anteil der Zeit, die Shader-Berechnung im Gegensatz zu der gesamten Framezeit in Anspruch nimmt, habe ich weitere Messungen ausgeführt. Diese messen die Zeit, die benötigt wird, um den Compute-Shader auszuführen und die VRAM-Puffer zu synchronisieren.

Dieses Diagramm vergleicht die Zeit die gebraucht wird, den Compute-Shader auszuführen und die Puffer auszulesen mit der Gesamtzeit eines Frames.

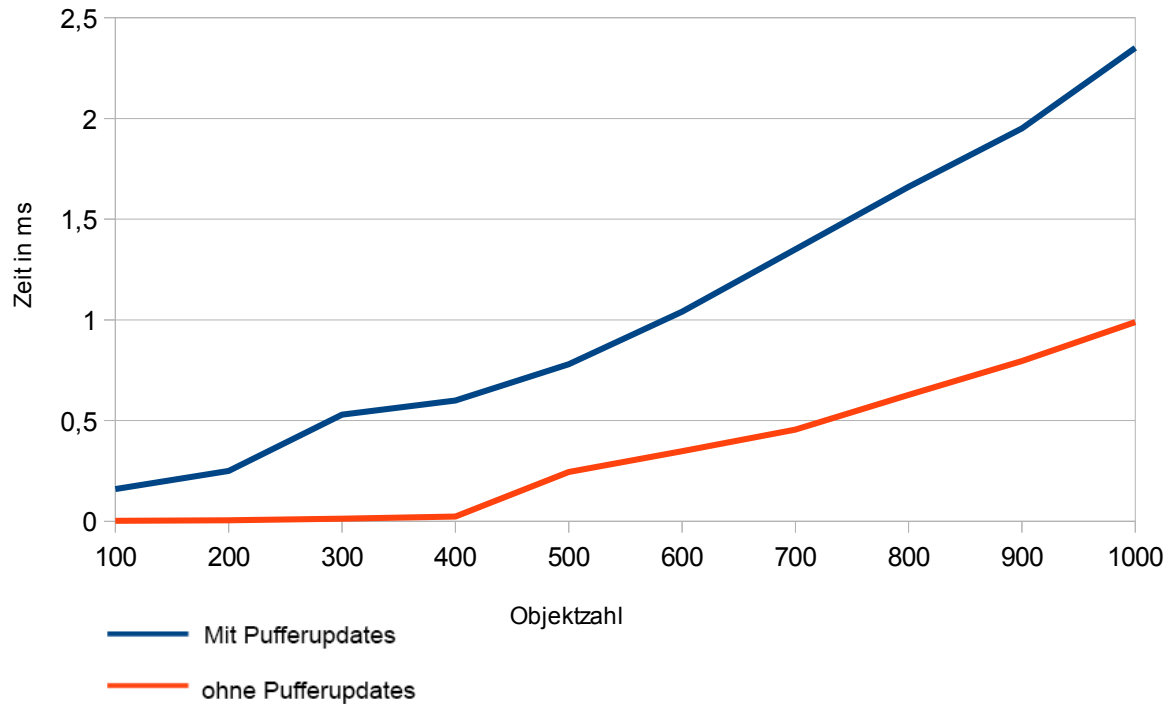


Dieser Vergleich wird die Zeit einer gesamten Aktualisierung, mit dem Anteil die der Compute-Shader in Anspruch nimmt verglichen.

Obwohl die Physiksimulation noch teilweise auf dem CPU verarbeitet wird, ist dessen Anteil besonders bei höheren Objektzahlen relativ klein. Das bedeutet, dass die Auslagerung des restlichen Teils der Berechnungen auf den GPU nur minimale Verbesserungen einbringen kann. Dies würde erst Sinn machen, wenn das Programm die Puffer nicht mehr auslesen und aktualisieren würde.

Puffer Updates

Das Programm wie es jetzt steht erfordert, dass die Puffer auf dem GPU mit jeder Aktualisierung beschrieben und gelesen werden. Um zu erkennen, wie groß der Einfluss davon auf die Performance ist, ignorieren wir die Pufferaktualisierungen und vergleichen die Zeitmessung.



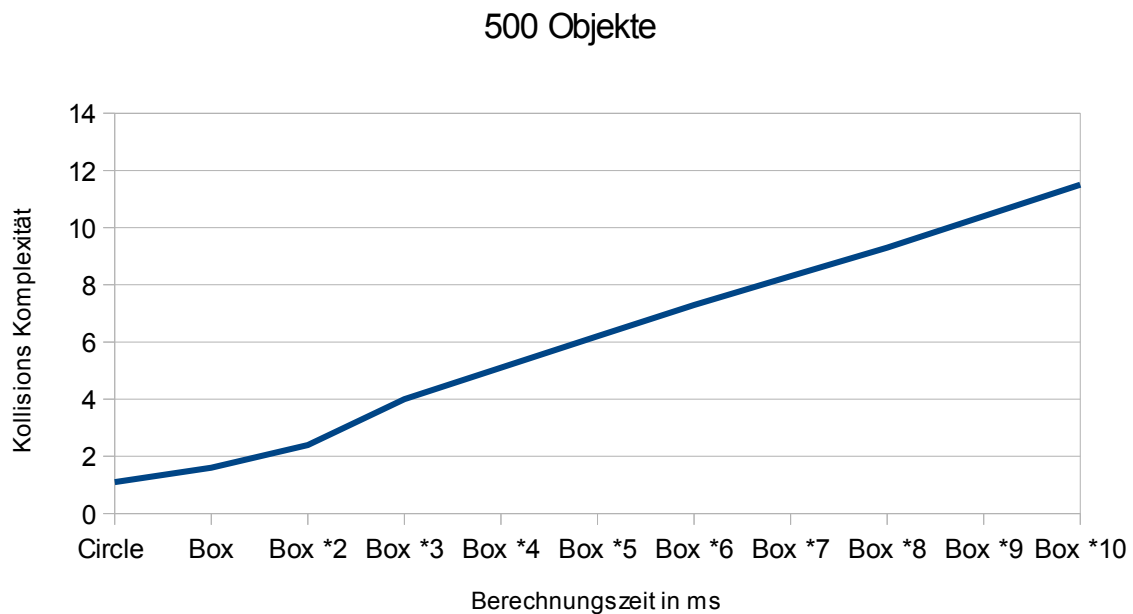
Die blaue Linie repräsentiert die Berechnungszeiten mit Pufferaktualisierungen, während bei der orangen Linie diese übersprungen werden. Die Verbesserung der Performance ist sehr deutlich bei kleineren Objektzahlen. Ab 500 Körpern ist der Unterschied kleiner aber immer noch eindeutig.

Die Schreibprozesse scheinen minimalen Overhead zu haben, der sich selbst bei sehr kleinen Puffern nicht zu reduzieren ist.

Es können also noch weitere Optimierungen erreicht werden indem die Pufferaktualisierungen reduziert werden. Eine Möglichkeit diese Speicherprozesse

Komplexität der Kollision

Um die Effekte einer komplexeren Kollisionserkennung zu simulieren, werden die Kollisionsabfragen mehrmals innerhalb derselben Invokation aufgerufen. Diese überflüssigen Berechnungen dienen keinen Zweck. Sie sollen nur helfen, einen besseren Einblick in das Verhalten des Shaders in anderen Umständen gewähren.

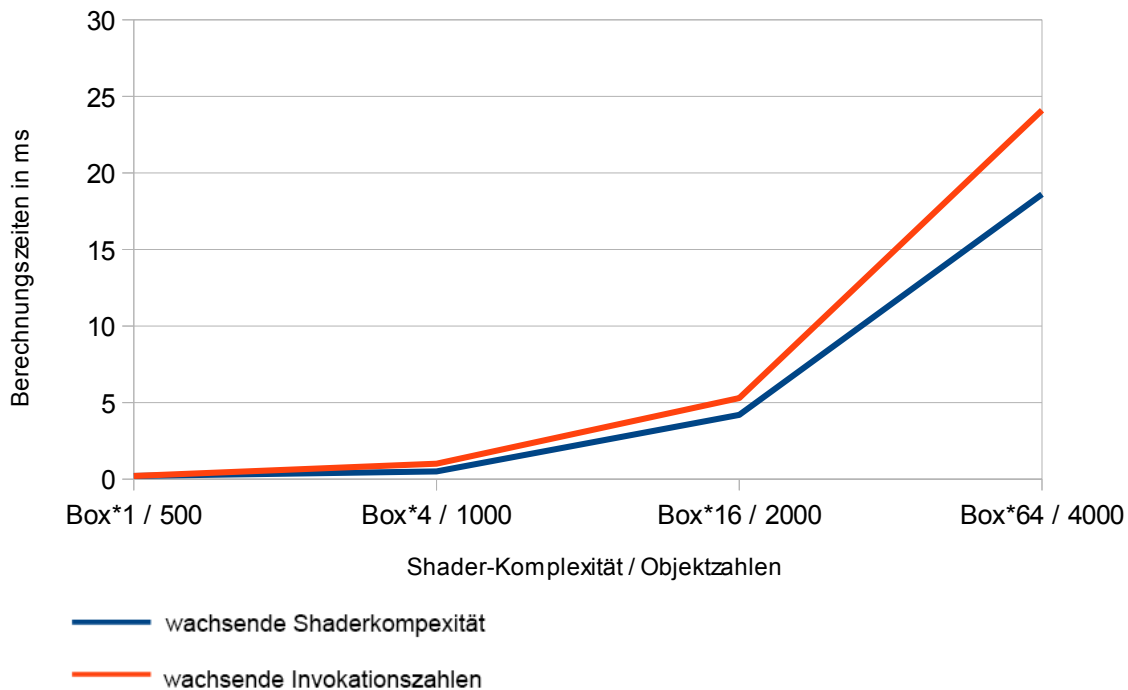


Dieses Diagramm zeigt die Zeit, die der Compute-Shader in Anspruch nimmt. Dabei bleibt die Anzahl an Objekten gleich bei 500, nur die Komplexität des Shaders wird erhöht. Das bedeutet dass die Anzahl der Shader-Invokationen gleich bleibt, aber jede eine größere Anzahl an Berechnungen verarbeitet. Erkennbar ist eine sehr lineare Verschlechterung der Performance.

Interessant ist, dass die Performance bei 'Box *4' mit 500 Objekten leicht besser ist im Vergleich zu vorherigen Tests mit 1000 Objekten. Beide Prozesse sollten die selbe Anzahl an Berechnungen verarbeiten, da in diesem Test die Kollisionsabfrage für 500 Objekte doppelt ausgeführt wird. Der einzige Unterschied ist, dass diese innerhalb 500 * 500 Invokationen berechnet werden anstatt von 1000 * 1000 Invokationen. Dies deutet darauf hin, dass es effizienter wäre, weniger Invokationen zu benutzen, während jeder Shader-Aufruf mehr Arbeit verrichtet.

Vergleich mit Äquivalenten Objektzahlen

Dieser Vergleich soll ermitteln, ob es sich lohnt die Berechnungen auf mehr Invokationen zu verteilen oder in jeder einzelnen Invokation mehr Aufgaben zuzuweisen. So wird die Shaderkomplexität erhöht, während die Invokationsanzahl gleich bleibt.



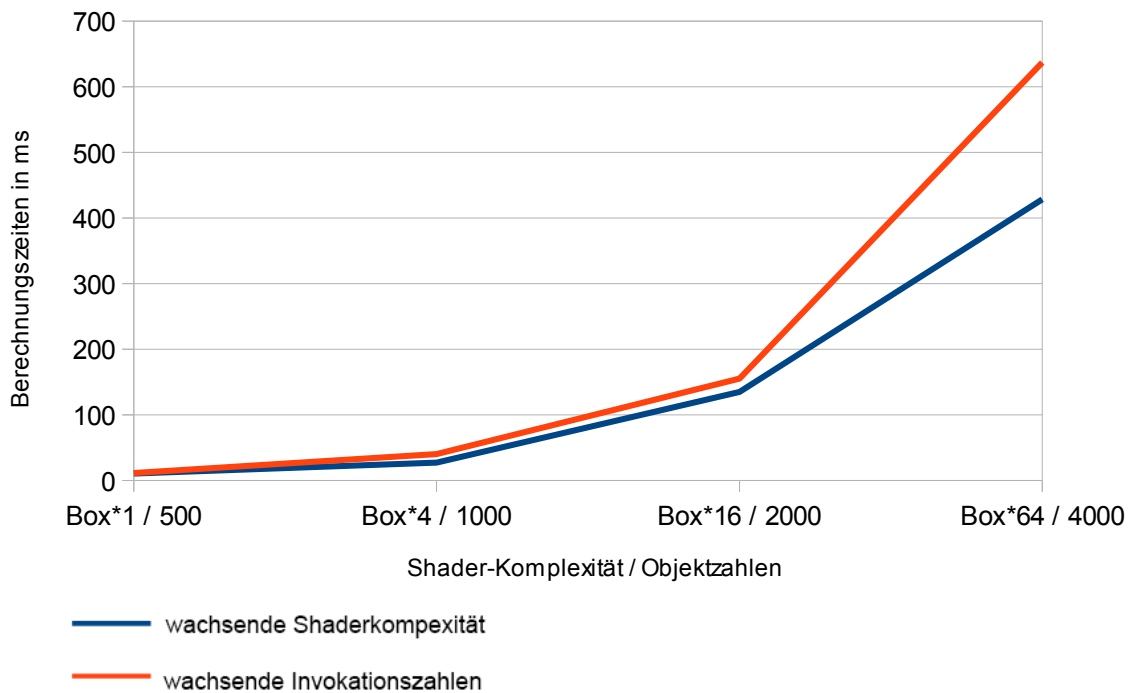
Die blaue Linie zeigt die Berechnungszeiten des Shaders, der nur eine Kollisionsabfrage innerhalb der Invokation bearbeitet. Hier wird die Zahl der Invokationen erhöht um mehr Kollisionsabfragen zu verarbeiten.

Für die orange Linie hingegen bleibt die Anzahl der Invokationen konstant bei 500* 500. Stattdessen werden mit jeder Invokation bis zu 64 Kollisionsabfragen ausgeführt. Die zu vergleichenden Werte verarbeiten dadurch jeweils dieselbe Anzahl an Kollisionsabfragen.

Bei bis zu 2000 Objekten liegen beide Ergebnisse noch relativ nahe aneinander. Sobald aber die Invokationsanzahl größer als 4000*4000 wird, verliert der Shader an Effizienz. Dieselbe Menge an Berechnungen kann schneller bearbeitet werden, indem jede Invokation einen größeren Anteil behandelt.

Möglicherweise ist das eine Begrenzung der Hardware. Zum Vergleich habe ich dieselben Tests auf einem zweiten, weniger leistungsstarken PC ausgeführt.

Vergleich mit Äquivalenten Objektzahlen PC2

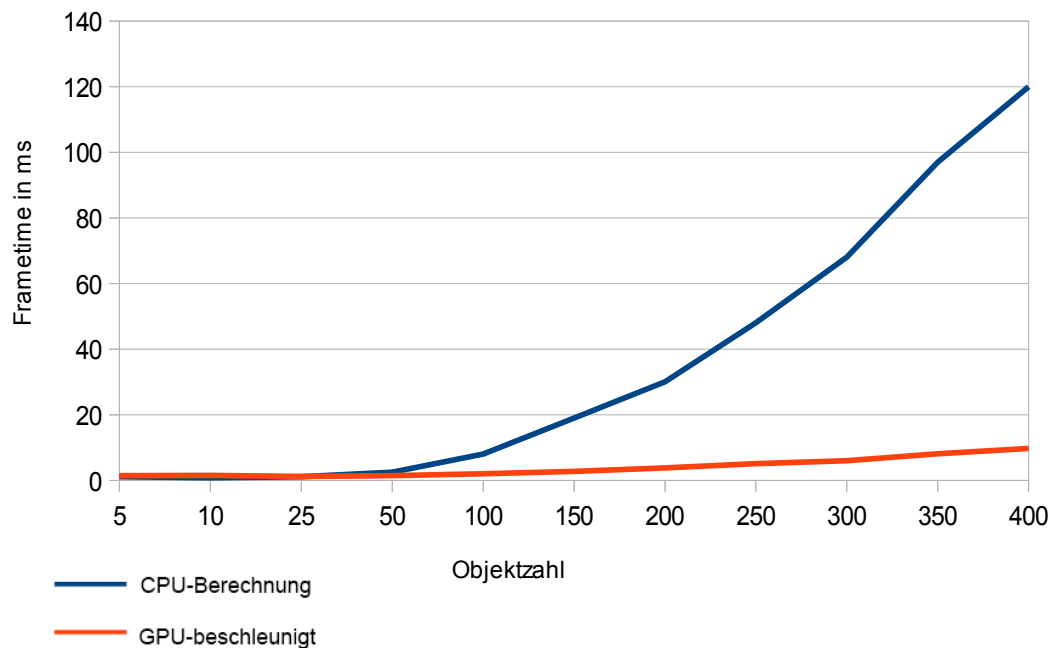


An dem zweiten Computer zeigt sich ein ähnliches Muster. Allerdings ist der Unterschied erst bedeutsam, bei einer sehr starken Auslastung des Prozessors. Beide Computer wären in diesen Situationen so stark ausgelastet, dass kein flüssiges Spiel dargestellt werden kann.

Die Reduzierung der Invokationszahlen kann von Vorteil sein, wie groß aber die Auswirkung ist, hängt stark von der Situation ab. Für eine Spieleengine wäre so eine Optimierung nicht von großen Nutzen. Wenn aber reine Rechenleistung gewollt ist und die Hardware voll ausgelastet werden kann, könnte diese Information von Vorteil sein.

Performance auf dem zweiten Computer

Der zweite Computer ist deutlich leistungsschwächer. Deshalb wird mit kleineren Objektzahlen getestet.



Die blaue Linie repräsentiert die CPU-Berechnung und zeigt wie in vorherigen Tests, einen exponentiellen Anstieg der Berechnungszeit. Allerdings ist die Performance schon bei 200 Objekten deutlich beeinträchtigt mit einer Framerate von 33fps. Die GPU-Berechnungszeiten bleiben wie erwartet deutlich kürzer.

Interessant ist, dass bei sehr kleinen Objektzahlen die CPU-Berechnung einen kleinen Vorteil hat. Dadurch dass keine Daten zwischen CPU und GPU ausgetauscht werden, hat der CPU hier einen kleinen Vorteil.

Tests mit kleinen Objektzahlen an PC 1

Der Unterschied zwischen den beiden Lösungen ist auch hier gering, solange nicht mehr als 100 Rigidbodies dargestellt werden.

5.2 Gesamt-Analyse

Die Vorteile des Compute-Shaders sind eindeutig. Es gab bei allen Tests einen klaren Vorteil für die GPU-Lösung. Obwohl das ständige Kopieren der Puffer zwischen VRAM und RAM zusätzliche Zeit in Anspruch nimmt, beeinträchtigt das die allgemeine Performance nur minimal.

Bei den Tests konnten aber auch manche Irregularitäten entdeckt werden, die Interessant für Entwickler sein können.

Puffer- Schreib und Leseprozesse scheinen bei kleinen Datenmengen ineffizient zu sein. Wenn also relativ unaufwendige Berechnungen auf dem GPU ausgeführt werden sollen, macht es Sinn diese Speicherprozesse möglichs zu reduzieren. So kann die Performance deutlich verbessert werden. Wenn aber das Shaderprogramm größere Datenmengen und Berechnungen behandelt, werden die Speicherprozesse im Verhältniss kleiner und beeinträchtigen die gesamte Performance weniger.

Es gibt verschiedene Arten das jetzige Programm zu Optimieren. Viele dieser Arten sind aber nur in Ausnahmefällen nützlich oder haben andere Nachteile.

6. Zusammenfassung

6.1 Zusammenfassung in Bezug zur These

Es bleibt also die Frage nach der Möglichkeit, mithilfe eines OpenGL Compute-Shaders eine 2D Physiks simulation mit Kollisionserkennung zu entwickeln, die performanter als eine C++ basierte Lösung ist. Nachfolgend sind die wichtigsten Punkte im Bezug auf die wesentlichen Aspekte zusammengefasst.

Kompatibilität der Hardware vorausgesetzt

Arbeitet man mit der Programmierung von Grafikkarten, wird man früher oder später an deren Kapazitätsgrenze stoßen.

Speicher- und Puffer-Objekte haben Begrenzungen, die je nach Hardware unterschiedlich sind.

Ältere Hardware ist zudem mit verschiedenen Features in OpenGL sowie in DirectX unvereinbar und wird nicht unterstützt.

Inkompatibilität der Hard- und Software kann zu unterschiedlichen, unerwarteten Problemen der Hardwarekonfigurationen führen.

Um das reibungslose Zusammenarbeiten der Hardware mit OpenGL bzw. DirectX zu gewährleisten, bedarf es einem größeren Aufwand und das ausgiebige Testen verschiedener Hardware.

Debugging ist erschwert

Das Programmieren in GLSL, besonders mit Compute-Shadern, erschwert das Debuggen erheblich in Gegensatz zur C++ Programmierung.

Zu C++ und anderen auf dem CPU ausgeführten Programmiersprachen gibt es zahlreiche und äußerst ausgereifte Entwicklungsumgebungen (IDEs), die den Entwicklungsprozess vereinfachen und beschleunigen.

Für Compute-Shader gibt es bisher keine vergleichbaren Hilfen. Das macht den Entwicklungsprozess umfangreich, kompliziert und unübersichtlich. Es wird mehr Zeit benötigt, Fehler im Programm aufzuspüren und zu beheben.

Eine voll ausgereifte Physik-Engine kann sehr komplex werden. Ohne effiziente Entwicklungs-Tools ist es der zusätzliche Aufwand in vielen Fällen nicht Wert sie komplett als einen Shader umzusetzen.

Bei der Programmierung mit Compute-Shadern gilt es weiterhin zu beachten:

- Es ist schwieriger auf den Speicher der Grafikkarte zuzugreifen.
- Im Gegensatz zu beliebten Development Environments für C++, könne GPU-Puffer nicht während der Laufzeit durch das Development Environment untersucht bzw. gelesen werden.
- Shaderprogramme können während der Ausführung nicht zum Debugging pausiert oder angehalten werden.

GLSL ist weniger flexibel

GLSL ist die Programmiersprache in der OpenGL-Shader geschrieben ist. Sie ist zwar auf C basiert und übernimmt die meisten der grundlegenden Funktionen, ist aber dennoch weniger flexibel und ausgereift. Dies erschwert die Entwicklung und ist daher für komplexere Programme wenig geeignet.

Getrennte Speicher müssen zur flexiblen Nutzung synchron gehalten werden

CPU und GPU haben eigene Speichercaches, die aktiv auf dem selben Stand gehalten werden müssen.

Es ist zwar möglich Ausgangsdaten über Objektpositionen und Geschwindigkeiten am Anfang der Simulation an die Grafikkarte weiterzugeben und den GPU unabhängig von dem CPU arbeiten zu lassen. Dadurch wird jedoch die Flexibilität der Nutzung eingeschränkt.

Um alle Daten zwischen RAM (CPU) und VRAM-Speicher (GPU) synchron zu halten, wird Bandbreite der Schnittstelle gebraucht. Da diese begrenzt ist, kann ein hoher Maß an Datenaustausch das Programm verlangsamen. In den getesteten Anwendungen wurden allerdings nie die Bandbreite so stark ausgenutzt, dass es zu starken Verzögerungen kam. Je nachdem wie stark andere Aspekte der Engine solche Datenaustausche benötigt, kann es jedoch trotzdem zu einem Problem werden.

Performance ist ein eindeutiger Vorteil

Wie die weiter oben beschriebenen Tests zeigen, ist der Gewinn der Performance durch die Nutzung des GPUs so groß, dass er nicht missachtet werden kann.

Unter Berücksichtigung der Kapazitäten und Stärken, können Berechnungszeiten erheblich verkürzt werden. Werden diese jedoch nicht beachtet, kann die Effizienz des Shaders leiden und dadurch die Hardwarekompatibilität eingeschränkt werden.

Aufwand

Die Implementierung eines Compute-Shaders ist mit Mehraufwand verbunden. Ohne Erfahrung ist diese Arbeit sicherlich eine Herausforderung und mit mehr Zeitaufwand verbunden.

Hinzu kommen noch die Einschränkungen bei Debuggen, die es schwieriger machen, Fehler und Defekte zu erkennen und zu beseitigen.

Durch die Schwierigkeiten mit Shaderprogrammierung und anderen Begrenzungen ist es häufig nicht empfehlenswert, komplexe Prozesse auf den GPU auszulagern. Stattdessen ist es sinnvoller nur einen bestimmten des Prozesses als Shader umzusetzen. Wenn der rechenaufwendigste Schritt eines Verfahrens identifiziert ist, kann dieser durch einen einfachen Shader ersetzt werden. So kann häufig die gesamte Performance verbessert werden, während der Entwicklungsaufwand minimal gehalten wird.

Im Fall der entwickelten Physiksimulation, wäre es zwar möglich den restlichen Teil der Simulation in weiteren Shaderschritten umzusetzen, die Performanceverbesserung wäre aber nur minimal.

6.2 Kritische Betrachtung der Vorgehensweise

Die praktische Durchführung und die Performance-Tests konnten einen guten Einblick in die Vorgehensweise und Schwierigkeiten bei der Entwicklung einer Physik-Engine bieten. Allerdings ist das erstellte Programm von einer realistischen Simulation noch weit entfernt.

Das Programm unterstützt nicht die Interaktion zwischen verschiedenartigen Objekttypen. Das heißt, es können keine Kollisionen zwischen einem Kreis und einem Rechteck behandelt werden. Mit mehr verschiedenen Objekttypen müsste der Compute-Shader komplexer und verzweigter werden.

Rotation konnte ich nur teilweise implementieren. Objekte können zwar rotiert angezeigt werden und die Kollision abgefragt werden. Allerdings konnte ich keinen Prozess erstellen, der es erlaubt, Rotation nach einer Kollision korrekt zu addieren. Der jetzige Prozess ist nur ein Platzhalter und nicht korrekt. Deshalb ist die Rotation in einem separaten Modus aktiviert.

Dazu werden bleibende Kontakte nicht korrekt simuliert. Wenn sich Objekte dauerhaft berühren, sollte das Programm nicht konstant Kollisionen entdecken.

Zusammen mit dem ungenauen Algorithmus für Überlappungsauflösung, verhalten sich berührende, ruhende Objekte nicht realistisch.

Die Kollisionserkennung kann auch nicht mehrere gleichzeitige Berührungen erkennen.

Es ist schwer abzuschätzen, wie die Performance sich verhalten wird, wenn verschiedene Elemente größere Komplexität erreichen. Dadurch sind die Tests teilweise keine genaue Repräsentation davon, wie sich eine volle Physik-Engine verhalten würde.

Verschiedene Tests, die Variablen wie Speicheraustausch oder Shaderkomplexität verändern, sollten einen Einblick in solche Fälle gewähren. Diese sind aber auch teilweise basierend auf Spekulation.

Die CPU-Lösung des Programms ist nicht darauf optimiert, alle Kerne des CPUs optimal zu nutzen.

Im Rahmen dieser Arbeit bleibt mir nicht genug Zeit, mich mehr für diese Art der Optimierung zu informieren. Es ist also wichtig zu beachten, dass die CPU-Performance nicht dem höchstmöglichen Potenzial entspricht.

6.3 Ausblick auf weitere Forschungen

Die Performance-Tests haben gezeigt, dass GPU-Beschleunigung erhebliche Verbesserungen ermöglicht. Das lässt Freiraum für Spekulationen, was in Zukunft in Bezug auf weitere Optimierungen möglich sein könnte.

Die Performance-Tests haben gezeigt, dass obwohl bereits große Verbesserungen erreicht wurden, es immer noch potential zur Optimierung gibt.

Die entwickelte Simulation hat nicht dieselbe Komplexität einer kompletten Physik-Engine.

Die Problematik in der Performance bei hoch verzweigten Shaderprogrammen, bleibt bei der Entwicklung einer vollen Physik-Engine. Sollen mehrere Collidertypen für verschiedene Formen miteinander interagieren, müsste ein System entwickelt werden, das dieses Problem löst.

Eine Möglichkeit der Optimierung ist, die Kollisionsabfrage in Physik-Engines in verschiedenen Detailstufen auszuführen. Dabei wird zuerst die Bounding-Box hergenommen, die die äußersten Ausmaße eines Objektes nimmt, um naheliegende Objekte zu finden. Danach erst wird die eigentliche Kollisionsabfrage ausgeführt.

Um Verzweigungen zu verhindern, könnten auch zwei verschiedene Shaderprogramme nacheinander ausgeführt werden. Im ersten Schritt würden naheliegende Objekte mithilfe von Bounding-Box-Kollision gefunden werden. Die Ergebnisse werden dann zurück an den CPU gesendet. Danach wird ein neuer Dispatch des zweiten Shaderprogramms mit der gebrauchten Größe aufgerufen.

Hier werden die eigentlichen Collider verglichen. So wird auch verhindert, dass komplexe Kollisionsalgorithmen zwischen Objekten ausgeführt werden, die sich nicht nahe sind.

Quellen

Literaturquellen

Quelle 2: Autor: Kevin Hawkins, Dave Astle Jahr: 2001 Titel: OpenGL Game Programming Verlag: Prima Tech

Quelle 3 :Autor: Randi J. Rost, Jahr: 2004, Titel: OpenGL Shading Language

Quelle 8: Autor: André LaMothe, Jahr: 2003 , Titel: Game Programmig für Dummies

Internetquellen

Quelle 1 : <http://www.nvidia.com/object/gpu.html>

Quelle 4: https://www.opengl.org/wiki/Compute_Shader

Quelle 5: <https://www.opengl.org/registry/>

Quelle 6: <http://www.wildbunny.co.uk/blog/2011/04/20/collision-detection-for-dummies/>

Quelle 7: <http://scienceworld.wolfram.com/physics/CoefficientofRestitution.html>

Quelle 9: <http://www.frustfrei-lernen.de/mechanik/newtonsche-gesetze.html>

Quelle 10: <http://www.frustfrei-lernen.de/mechanik/gleichmaessig-beschleunigte-bewegung-physik.html>

Quelle 13: <http://wili.cc/blog/opengl-cs.html>

Quelle 14: <http://www.geeks3d.com/20140704/tutorial-introduction-to-opengl-4-3-shader-storage-buffers-objects-ssbo-demo/>

Abbildungsverzeichnis

Abbildung 1: Quelle: <http://www.wildbunny.co.uk/blog/2011/04/20/collision-detection-for-dummies/>

Abbildung 2: Quelle: <http://www.wildbunny.co.uk/blog/2011/04/20/collision-detection-for-dummies/>

Abbildung 3: Quelle: <http://www.wildbunny.co.uk/blog/2011/04/20/collision-detection-for-dummies/>

Logbuch:

März 2016

Erreichte Ziele:

Ich habe mich zuerst darauf konzentriert ein Grundverständnis von OpenGL zu erlangen. Angefangen habe ich mit grober Recherche über OpenGL und Shaderprogrammierung. Dazu habe ich nach Tutorials über OpenGL Initialisierung gesucht und angefangen, diese praktisch umzusetzen.

Ziele:

Als nächstes werde ich mich darauf konzentrieren die OpenGL Initialisierung praktisch umzusetzen und den Rendering-Prozess zu recherchieren.

April 2016

Erreichte Ziele:

Nachdem ich mithilfe von OpenGL ein Fenster erstellen und Vertices anzeigen konnte, habe ich angefangen eine Programmstruktur zu planen und zu erstellen. Diese soll es erlauben mehrere bewegliche Objekte darstellen, die miteinander interagieren können.

Zusätzlich fing ich an den theoretischen Teil zu strukturieren und meine Recherchen aufzuschreiben.

Ziele:

Implementierung eines einfachen Compute-Shaders und weitere Recherche über die Nutzung. Die Programmstruktur muss weiter ausgebaut werden, um Physiksimulation zu erlauben.

Mai 2016

Meine ersten Versuche einen Compute-Shader zu implementieren scheiterten, da ich OpenGL nicht in einer neueren Version starten kann. Das Tutorial nach dem ich gearbeitet habe war nur für die ältere Version OpenGL 2.1. Ich hatte erwartet, dass ich die Version später ohne Probleme upgraden kann, da die Initialisierung es erlaubt, die gewollte Version anzugeben. Allerdings können höhere Version als 3.2 nicht korrekt initialisiert werden. Die Version 4.2, die den gewollten Compute-Shader unterstützt, kann nicht benutzt werden, ohne das Programm anzupassen.

Entweder der Renderprozess muss für neuere Versionen abgeändert werden, oder es müssen bei der Initialisierung Modifikationen gemacht werden.

Stattdessen konzentriere ich mich mehr darauf Physik-Berechnungen und Kollisionsabfragen zu recherchieren.

Ich konnte eine einfache Kollisionsabfrage implementieren, die die Objekte als Kreise behandelt. Das Programm hat ebenfalls bereits ein System, das es erlaubt Objekte darzustellen, die miteinander, sowie mit einer Außenwand kollidieren. Die benutzten Algorithmen können dann später in einem Compute-Shader implementiert werden.

Die Programmstruktur ist auf verschiedenen Klassen basiert:

CEngine:

CEngine ist die Hauptklasse und steuert die Initialisierung, sowie die Hauptschleife. Diese Schleife steuert die Aktualisierung über `Update()` und die Bilderstellung über `Render()`.

Util.h:

Die Util.h Datei inkludiert alle benutzten Bibliotheken, sowie allgemeine Klassen, Funktionen und Präprozessor-Befehle. Sie ist von allen Klassen sichtbar. Hier werden auch die Modi des Programms gesteuert.

Präprozessor-Anweisungen lassen das Verhalten des Programms an verschiedenen Stellen leicht bestimmen. Und mit Konstanten Werten wie `bodyCount` und `f_InitialSpeeds` lassen sich die Menge und Geschwindigkeit der dargestellten Objekte bestimmen.

Andere Header-Dateien wie `VertexModel.h` und `BoxCollider.h` enthalten Strukturen für Kollisions- und Vertexmodelle.

CRigidbody

Die CRigidbody Klasse behandelt die Aktualisierung und das Rendern eines Einzelnen Objektes.

Polygone Rendern:

Für das Rendern von Polygonen werden mindestens zwei Shader benötigt: einen Vertex- und einen Fragment-Shader.

Um Shader zu laden, habe ich die Klasse CShaderProgram erstellt. Diese enthält Funktionen, die eine Textdatei von der Festplatte lesen und sie als Shader kompilieren kann. Dazu erstellt sie ein Shaderprogramm, das an das beide Shader gebunden werden kann. Wenn dieses Programm mit dem Befehl `glUseProgram(gProgramID)` ausgewählt wird, benutzt OpenGL diese Shader, wenn der Render-Befehl ausgelöst wird.

Diese Klasse basiert auf den Tutorials von <http://lazyfoo.net/tutorials/OpenGL>. Sie ist als rein Virtuelle Klasse gedacht. Das heißt, dass alle Shaderprogramme davon ableiten.

Die Klasse CPlainPolyProgram erstellt ein Shaderprogram für das Rendern von einfachen einfarbigen Polygonen. Der Shader-Quellcode wird direkt aus Textdateien gelesen. Der richtige Pfad wird über die Funktion GetExePath() in CEngine gefunden. Diese Funktion findet zuerst, den Ordner, der die .exe-Datei enthält. So können die Textdateien gefunden werden, auch wenn sich der Pfad des Ordners verändert.

Über die Funktion LoadProgram(std::string _path) werden die Shader geladen, Kompiliert und an ein Shaderprogramm gebunden.

```
#version 430
layout (location = 0) in vec3 VertexPos;
uniform vec4 ObjectPos;

void main()
{
    gl_Position = ObjectPos + VertexPos;
}
```

Der Vertexshader ist möglichst einfach gehalten. Als Input erhält jeder Vertex eine Position und eine Objektposition. Die Objektposition wird jedes Mal, wenn ein neues Objekt gerendert wird, manuell bestimmt.

```
void CPlainPolyProgram::setVertexOffset(GLfloat r, GLfloat g, GLfloat b, GLfloat a)
{
    //Update Position Offset
    glUniform4f(gVertexOffsetLocation, r, g, b, a);
}
```

Diese Funktion erlaubt es eine uniform Variable jederzeit direkt zu beschreiben. Die Vertexpositionen 'VertexPos' werden direkt von dem ausgewählten Vertexpuffer gelesen.

Der Fragment-Shader ist ebenfalls einfach gehalten.

Der Standardoutput 'outputColor' kann mit einem einfachen Vektor4 beschrieben werden.

```
#version 430
out vec4 outputColor;
void main()
{
    outputColor = vec4(1.0,0.2,0.0, 1.0);
}
```

Der Fragment-Shader produziert dann eine Farbe basierend auf den Werten des Vektors.

Ziele:

Kompatibilität mit OpenGL-4.3 implementieren.

Juni 2016

Mit weiteren Tutorials ist es mir gelungen OpenGL-4.2 zu implementieren, indem ich die Initialisierung und die Render-Aufrufe angepasst habe.

Die Physiksimulation ist mittlerweile komplett, bis auf die Rotation und muss nur noch in einen Compute-Shader implementiert werden. Dazu habe ich mehr recherchiert, um eine Angewandte zu finden, ein Compute-Shader-Programm zu implementieren.

Ich kann bereits einen Compute-Shader kompilieren und ausführen, habe aber noch keinen Weg gefunden Ergebnisse, von dem C++-Programm auszulesen. Da OpenGL sehr verzweigt ist und viele verschiedene Versionen und Extensions hat, erschwert das die Recherche. Viele Artikel nutzen andere Abzweigungen von OpenGL, die nicht kompatibel mit meinen eigenen sind.

Initialisierung von OpenGL:

GLFW ist eine öffentlich erhältliche Bibliothek, die Funktionen anbietet, die die Initialisierung vereinfachen.

```
glfwInit();
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
```

Mithilfe von der Funktion `glfwInit()` lässt sich OpenGL initialisieren, während die Funktion `glfwWindowHint()` zusätzliche Einstellungen bestimmen lässt. Hier wird die OpenGL-Version auf 4.3 gesetzt und die Fenstereinstellungen bestimmt.

Das Fenster wird durch den `glfwCreateWindow()` Befehl mit der gewollten Größe erstellt.

```
// Create Window
m_window = glfwCreateWindow(SCREEN_WIDTH, SCREEN_HEIGHT, "Honours", nullptr, nullptr);
```

Die Initialisierung basiert auf den Tutorials von LearnOpenGL.org

CPhysicsScene:

CPhysicsScene soll es vereinfachen große Mengen an Objekten zu bearbeiten. Die Klasse initialisiert, aktualisiert und rendert alle Rigidbodies in den entsprechenden Funktionen.

Sie hat Zugriff auf eine Liste aller Objekte in der Szene, die der Reihe nach bearbeitet werden. Zusätzlich wird die Kollisionsabfrage innerhalb der Update-

Funktion aus gesteuert.

Der Compute-Shader wird auf dieselbe Art wie die bereits besprochenen Shader behandelt. Allerdings muss dieser manuell ausgelöst werden. Dafür wurde die Funktion `RunShader()` hinzugefügt. Diese startet mit der OpenGL-Funktion `glDispatchCompute`(GLuint num_groups_x,GLuint num_groups_y,GLuint num_groups_z) eine Reihe an Workgroups. Diese werden den Shadercode mehrmals ausführen, je nach der Größe der angegebenen Werte.

Ziel: Eine Art finden um Inputs und Outputs von dem Compute-Shader zu finden.

Viele der Tutorials zu Compute-Shadern, die ich gefunden habe nutzen Texturen. Die könnten eventuell auch für andere Daten als Farbinformationen genutzt werden. Eine Textur könnte beschrieben werden und später ausgelesen werden. Ich werde die Möglichkeit, eine Textur als Output-Puffer zu nutzen recherchieren.

Juli 2016

Da der Compute-Shader keine direkten Outputs unterstützt, müssen die Ergebnisse indirekt ausgelesen werden. Der Shader muss den Speicher auf dem VRAM beschreiben, der dann später von dem C++ Programm gelesen wird. Alternativ könnten der Speicher direkt von dem Vertex-Shader ausgelesen werden und so die Position von Objekten bestimmen, ohne dass Daten vom VRAM auf den RAM-Speicher kopiert werden müssen.

Mein erster Versuch, war es eine Textur auf die Grafikkarte hochzuladen und von dem Compute-Shader beschreiben zu lassen. Die Textur könnte als Daten-Array benutzt werden, in dem Positions- und Geschwindigkeitsdaten abgespeichert werden. Nachdem mir das gelang, merkte ich dass eine Textur für diese Anwendung nicht gut geeignet ist.

Danach recherchierte ich GPU-Puffer weiter und fand ein Artikel, der beschreibt wie Shader-Storage-Buffers benutzt werden können, um von Shadern beschrieben und ausgelesen werden können.

Nachdem es mir gelang dieses zu implementieren, versuchte ich die Physiksimulation mit dem Compute-Shader umzusetzen.

Outputs/Inputs

Um Outputs und Inputs für den Compute-Shader zu erlangen, habe ich 'Shader Storage Buffer Object' (SSBO) eingebaut, die mit dem C++ Programm beschrieben und gelesen werden können. Der SSBO erlaubt es einen Puffer auf dem VRAM zu erstellen. Der VRAM ist Arbeitsspeicher, der speziell für die Grafikkarte zugreifbar ist. Um auf diesen zuzugreifen bietet OpenGL verschiedene Funktionen an.

Dazu habe ich der Klasse `CPhysicsShaderProgram` die Funktionen `GenBuffers()`, `ReadBuffers()` und `Update-ShaderData()` hinzugefügt. In `GenBuffers()` wird ein neuer

Puffer auf dem VRAM erstellt. Dieser basiert auf dem in der CPhysicsShaderProgram definierten Struktur s_shader_data. Dieser Puffer, soll von der Grafikkarte, sowie von dem CPU ansprechbar sein. Dazu muss der Puffer einmal auf dem RAM-Speicher, sowie einmal auf dem VRAM-Speicher existieren. Dabei müssen beide Versionen Synchron gehalten werden. Dazu überschreibt die Funktion UpdateShaderData() mit jeder Aktualisierung den shader_data-Puffer auf dem VRAM, mit der aktuellen Version auf dem RAM-Speicher

Für Outputs wird ein Zweiter Puffer genutzt. Dieser muss vor jeder Berechnung auf null gesetzt und danach gelesen werden.

Um den Puffer innerhalb des Shadercodes aufzurufen muss eine Variable im Shader definiert werden.

```
layout (std430, binding=1) readonly buffer shader_data
{
    vec2 Positions[BodyCount];
    vec2 Velocities[BodyCount];
};
```

Mithilfe des layout()-Befehls lässt sich ein bestimmter Speicherort bestimmen. Der erstellte Puffer kann dann an den korrekten Ort gebunden werden. Dabei muss beachtet werden, dass im C++-Code ein entsprechendes Pufferobjekt vorhanden ist

```
struct S_Shader_data
{
    float Positions[bodyCount * 2];
    float Velocities[bodyCount * 2];
};
```

Dazu habe ich ein Struct erstellt. Um Vektoren abzuspeichern können Arrays von Floats mit doppelter Größe genutzt werden. In der Funktion GenBuffers() wird ein Pufferobjekt auf dem VRAM basierend auf dem 'shader_data' Objekt erstellt.

```
glGenBuffers(1, &gShaderBuffer);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, gShaderBuffer);
glBufferData(GL_SHADER_STORAGE_BUFFER, sizeof(shader_data), &shader_data,
GL_DYNAMIC_COPY);
```

Hier wird zuerst ein Puffer erstellt und die Speicheradresse in 'gShaderBuffer' gespeichert. Danach wird diese Adresse ausgewählt und das Struct-Objekt 'shader_data' an diese Stelle Kopiert.

```
// find the storage block index:
GLuint block_index = 0;
block_index = glGetProgramResourceIndex(gProgramID, GL_SHADER_STORAGE_BLOCK,
"shader_data");
//connect the shader storage block to the gShaderBuffer
GLuint gShaderBuffer_binding_point_index = 1;
glShaderStorageBlockBinding(gProgramID,
block_index, gShaderBuffer_binding_point_index);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER,
gShaderBuffer_binding_point_index, gShaderBuffer);
```

Hier wird der neu erstellte Puffer an den richtigen Index 1 gebunden. Wir haben

vorher im Shader den Binding-Index auf 1 gesetzt (layout (std430, binding=1)). So ist sichergestellt, dass der Shader und das C++-Programm auf den selben Speicherort schreiben.

Insgesamt habe ich mich bei der Recherche zu sehr auf Tutorials konzentriert. Weil alle Tutorials die ich gefunden habe Texturen als Puffer nutzten, war mir nicht klar, dass es eine bessere Art gibt, um Daten von dem Grafikspeicher auszulesen.

Ziel: Den Compute-Shader nutzen, um die Kollisionsabfrage zu verarbeiten.

August 2016

Da die Physiksimulation in einer Reihe von Schritten geschieht, ist es nicht möglich, sie in einem einzigen Shader umzusetzen. Shader werden nicht in einer vorhersehbaren Reihenfolge und teilweise gleichzeitig ausgeführt. So ist es schwer möglich die Kollisionsabfrage und die Bewegung der Objekte im gleichen Shader zu bearbeiten. Es wäre zwar möglich pro Shader-Invokation ein Objekt zu bearbeiten. Allerdings ist das sehr ineffizient.

Der Grafikprozessor ist am effektivsten, wenn er an vielen Instanzen eines kleinen Codestückes gleichzeitig arbeiten kann. Um dies möglichst auszunutzen, ist es am besten nur die Kollisionsabfrage über den Shader zu bearbeiten. Da diese sowieso die meiste Performance beansprucht, wird der Rest der Simulation immer noch über den CPU bearbeitet.

Dazu habe ich daran gearbeitet den theoretischen Teil fertigzustellen.

Compute Shader Implementierung:

In dieser Implementierung des Compute-Shaders wird nur die Kollisionsabfrage mithilfe des GPUs verarbeitet. Dazu werden verschiedene GPU-Puffer genutzt: Der 'shader_data'-Puffer enthält alle Positions und Geschwindigkeitsdaten. Sie werden innerhalb von Vektor-Arrays gespeichert. Die Indexe jedes Vektors entsprechen den selben Indexen der Arrays der Rigidbodies.

```
struct S_Shader_data
{
    float Positions[bodyCount * 2];
    float Velocities[bodyCount * 2];
};
```

Der 'shader_output' enthält alle Ergebnisse der Kollisionen. Dieser muss vor jeder Berechnung auf null gesetzt werden und danach ausgelesen werden.

```
struct S_Shader_Output
{
    float vNormals[bodyCount * 2];
    float vHitPoint[bodyCount * 2];
    float fOffset[bodyCount];
    int iHitCollider[bodyCount];
};
```

Der Puffer Enthält Informationen über Normalen, Kollisionspunkte, Überlappungstiefe und den Index des getroffenen Kolliders.

Zusätzlich wurde noch der 'Shader_colliders' Puffer hinzugefügt, der die Ausmaße der Kollisionsmodelle beinhaltet. Dieser muss nicht aktualisiert werden.

```
struct S_Shader_colliders
{
    float width;
    float height;
};
```

Nachdem alle Compute-Shader Berechnungen fertiggestellt und der Output-Puffer gelesen wurden, wird die Szene Aktualisiert. Dazu werden in CPhysicsScene die Output-Daten genutzt um die Reflexion, der einzelnen Objekte zu bearbeiten. Da der Compute-Shader auf die selbe Art arbeitet wie der C++ Code, können die gleichen Funktionen genutzt werden, um die Ergebnisse der Kollisionsabfrage zu verwerten. Danach werden die Positionen aktualisiert, sowie der Puffer in der CPhysicsShaderProgram Klasse.

Performance Tests

Nachdem das Programm fertiggestellt wurde habe ich angefangen die Performance-Tests auszuführen. Getestet wurden die Zeiten, die das Programm braucht um eine komplette Schleife fertigzustellen. Dazu wird die Zeit bevor das Programm aktualisiert und rendert gemessen, und danach.

Die Differenz ist die Deltatime. Wenn diese kleiner ist, kann das Programm eine hohe Aktualisierungsfrequenz erreichen und ist Effizienter.

Um einen Besseren Einblick zu erreichen werden verschiedene Variablen verändert und die Ergebnisse verglichen. Zuerst werden verschiedene Objektzahlen von 100 bis 1000 getestet.

September 2016

Im September habe ich versucht das Programm aufzuräumen. Manche überflüssigen Klassen und Funktionen wurden gelöscht und übersichtlicher gemacht.

Dazu habe ich versucht Rotation einzubauen. Diese hat aber noch viele Probleme, weshalb sie nur mit einer zusätzlichen Option aktivierbar ist.

Ich habe auch keinen Weg gefunden Vektortransformationen effizient zu bearbeiten, wodurch Performance erheblich schlechter ist, wenn dieser Modus aktiviert ist.

Performance-Messungen:

Folgend sind die gemessenen Daten, die für die Performance-Analyse gesammelt wurden. Zeiten wie Computetime und Frametime sind in Millisekunden angezeigt.

CPU-Berechnung PC1:

Bodies	Frametime	FPS
100	3,294	303,515
200	12,843	77,8635
300	28,848	34,6647
400	50,696	19,7253
500	79,111	12,6405
600	113,412	8,81743
700	153,814	6,50135
800	222,9	4,49
900	285,652	3,5
1000	359	2,86

Compute-Shader Berechnung PC1:

Objekte	Fame Time	Compute Time	FPS
100	0,4	0,16	2259
200	0,69	0,25	1451
300	0,91	0,53	1007
400	1,26	0,6	794
500	1,58	0,78	633
600	1,78	1,04	562
700	2,29	1,35	435
800	2,77	1,66	361
900	2,93	1,95	342
1000	3,55	2,35	282

Puffer Tests PC1:

Um den Effekt der Speicherprozesse zu erkennen, habe ich die Option **IGNOREBUFFERS** eingebaut. Wenn diese aktiviert ist werden Aktualisierung und Leseprozesse de

VRAM-Puffer ignoriert:

Objekte	Ctime NoBuffers
100	0,002
200	0,005
300	0,013
400	0,024
500	0,245
600	0,348
700	0,455
800	0,628
900	0,795
1000	0,988

CPU-Berechnung PC2:

Objekte	Frametime	FPS
5	1,1	917
10	0,8	1180
25	1,1	776
50	2,5	378
100	8	124
150	19	54
200	30	33
250	48	21
300	68	14
350	97	10
400	120	8

Compute-Shader Berechnung PC2:

Objekte	Frametime	ComputeTime	FPS
5	1,4	0,8	691
10	1,5	0,8	676
25	1,17	0,96	564
50	1,44	0,7	693
100	2	1,06	501
150	2,7	1,5	373
200	3,8	2,5	263
250	5,1	3,6	196
300	6	4,4	166
350	8,1	6,2	123
400	9,7	3,4	107

Kollisionskomplexität:

Hier wird die Komplexität des Shaderprogrammes künstlich vergrößert, indem überflüssige Kollisionsabfragen ausgeführt werden.

Komplexität	Frametime	ComputeTime	FPS
Circle	1,7	1,1	556,25
Box	2,3	1,6	443
Box *2	3,1	2,4	317,57
Box *3	4,6	4	215,71
Box *4	5,7	5,1	173,96
Box *5	6,8	6,2	146,9
Box *6	8	7,3	125,76
Box *7	9	8,3	111,47
Box *8	10	9,3	100,4
Box *9	11	10,4	90,38
Box *10	12,1	11,5	82,45

Um das zu erreichen wurde der Shader modifiziert

.


```

for(int i = 0; i < 2; ++i)
{
    CheckOverlapAABB();
}

```

Diese for-Schleife wiederholt die Kollisionsabfrage zwei mal. Die Anzahl der Wiederholungen entsprechen den Angaben in der 'Komplexität' Schleife.

Vergleich mit Äquivalenten Objektzahlen

Um herauszufinden, ob der GPU schneller dabei ist mehrere Invokationen abzuarbeiten oder komplexere Shaderprogramme wurde dieser Test ausgeführt.

In der ersten Spalte wurden 500 Invokationen ausgeführt. Anstatt die Objekzahl zu erhöhen werden mehrere Kollisionsabfragen in einer Invokation behandelt.

Um das äquivalent mit der zweiten Spalte zu machen, werden dort die Invokationen erhöht.

500*500 * 1	= 500*500
500*500 * 4	= 1000*1000
500*500 * 16	= 2000*2000
500*500 * 64	= 4000*4000

Mit diesen Berechnungen habe ich sichergestellt, dass beide Versionen vergleichbar sind.

PC1

	growing compexity	growing Invocation amount
Box*1 / 500	0,2	0,2
Box*4 / 1000	0,5	1
Box*16 / 2000	4,2	5,3
Box*64 / 4000	18,6	24,1

PC2

Box*1 / 500	10,3	11,3
Box*4 / 1000	10,3	11,3
Box*16 / 2000	27,1	40
Box*64 / 4000	135,1	155