Institut für Visualisierung und Interaktive Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 169

GPGPU-basiertes Glyph-Raycasting

Daniel Söll

Studiengang:	Informatik
Prüfer/in:	Prof. Thomas Ertl
Betreuer/in:	Dipl.-Inf. Daniel Kauker, Dr. Guido Reina
Beginn am:	15. August 2014
Beendet am:	15. Februar 2015
CR-Nummer:	I.3.7, D.1.3

Kurzfassung

Um den stetig steigenden grafischen Anspruch aktueller Computerspiele gerecht zu werden besitzen moderne Grafikkarten heutzutage mehrere hundert Rechenkerne welche sich parallel um das Rendern der Spielszene kümmern. Der rasante Anstieg der zur Verfügung stehenden Rechenkerne in einer GPU führte zu einem Aufschwung im Bereich des GPGPU (General-purpose computing on graphics processing units). Dabei werden rechenintensive Berechnungen auf die GPU ausgelagert um dort von der massiven Parallelität zu profitieren. Insbesondere Simulationen in den Naturwissenschaften ziehen aus der Verfügbarkeit von mehreren hunderten Rechenkernen einen starken Vorteil gegenüber der herkömmlichen Berechnung auf einer CPU. Im Gegensatz zu Programmschnittstellen wie OpenGL und DirectX welche das Verhalten der GPU hauptsächlich über Zustandsveränderungen steuern, gibt es für den Bereich der GPGPU einige eigens dafür entwickelte Programmiersprachen mit denen es möglich ist die Rechenkerne für die eigenen Zwecke zu programmieren. Die von Nvidia entwickelte parallele Programmiersprache Cuda ist eine davon und bietet die Möglichkeit die Rechenpower moderner Grafikkarten für fast beliebige Berechnungen zu nutzen. Ein interessanter Anwendungsfall, der von dieser parallelen Stärke einer GPU profitiert, stellt das Raycasting dar da dabei jeder Pixel des Bildes getrennt voneinander gerendert werden kann. Um zu untersuchen ob ein Raycasting Algorithmus in Echtzeit auf einer GPU mithilfe von Cuda lauffähig ist wird ein Raycaster in Cuda umgesetzt und die daraus resultierenden Ergebnisse in dieser Arbeit vorgestellt.

Inhaltsverzeichnis

1 Einleitung **9**
- 1.1 Ziel der Arbeit . . . 9
- 1.2 Bisherige Arbeiten/related work . . . 9

2 Grundlagen **11**
- 2.1 Raycasting . . . 11
  - 2.1.1 Funktionsweise . . . 11
  - 2.1.2 Unterschied zu Rasterisation . . . 12
  - 2.1.3 Unterschied Raytracing - Raycasting . . . 13
  - 2.1.4 Objekt Schnittberechnung . . . 13
- 2.2 Cuda . . . 15
  - 2.2.1 Einleitung . . . 15
  - 2.2.2 CPU vs GPU . . . 16
  - 2.2.3 Nvidia GPU Hardware Aufbau . . . 17
  - 2.2.4 Cuda C . . . 17
  - 2.2.5 Cuda Memory Model . . . 18
  - 2.2.6 Parallelität . . . 19
  - 2.2.7 GPU Eignung für Raycasting . . . 19
  - 2.2.8 Unterschied zu OpenGL mit programmierbaren Shadern . . . 19

3 Umsetzungsmöglichkeiten in Cuda **21**
- 3.1 Tile basierter Ansatz . . . 21
- 3.2 Äußere Schleife über Kugeln oder Pixel . . . 21
  - 3.2.1 Variante 1: Die äußere Schleife läuft über die Pixel des Bildes und es werden verschiedene Pixel gleichzeitig gerendert. . . . 23
  - 3.2.2 Variante 2: Die äußere Schleife läuft über die Objekte der Szene und es werden verschiedene Pixel gleichzeitig gerendert. . . . 23
  - 3.2.3 Variante 3: Die äußere Schleife läuft über die Pixel des Bildes und es werden verschiedene Objekte gleichzeitig gerendert. . . . 23
  - 3.2.4 Variante 4: Die äußere Schleife läuft über die Objekte der Szene und es werden verschiedene Objekte gleichzeitig gerendert. . . . 23
- 3.3 Daten Streaming . . . 24
- 3.4 Speicher Coalsescing Constraints . . . 24
- 3.5 Ausnutzen von shared oder constant memory . . . 25
- 3.6 Z-Buffer und lokaler Framebuffer . . . 26

4 Implementierung **27**
4.1 Einleitung . . . 27
4.2 Variablen und Datentypen . . . 27
4.3 Speicher Initialisierung auf der GPU . . . 28
4.4 Z-Buffer und lokaler Framebuffer . . . 28
4.5 Kugel Sortierung auf CPU Seite . . . 29
4.6 Kugel Boundary Box . . . 30
4.7 Vor- und Nachteile des jeweiligen Ansatzes . . . 32
4.7.1 Variante 1 ohne lokalen Framebuffer und ohne vorherige Sortierung der Objekte 33
4.7.2 Variante 1 mit lokalem Framebuffer und ohne vorherige Sortierung der Objekte 35
4.7.3 Variante 1 mit lokalem Framebuffer und vorsortierten Objekten . . . 36
4.7.4 Variante 2 . . . 37
4.7.5 Variante 3 ohne lokalen Framebuffer und ohne Vorsortierung der Kugeln . . . 38
4.7.6 Variante 3 mit lokalem Framebuffer und mit Vorsortierung der Kugeln . . . 38
4.7.7 Variante 4 . . . 39
4.7.8 Einsatz der Boundary Box . . . 39
4.7.9 Variante 5 . . . 40
4.8 Erreichen hoher GPU Auslastung durch geeignete Grid und Block Größen . . . 41
4.9 Limitierungen durch Register Anzahl, sharedMemory . . . 41
4.10 Vermeidung von thread divergenz . . . 42

5 Performance **43**
5.1 Renderperformance beim rendern von 1000 Kugeln . . . 43
5.1.1 Variante 1: äußere Schleife über Pixel, Parallel über Pixel . . . 43
5.1.2 Variante 2: äußere Schleife über Kugeln, Parallel über Pixel . . . 44
5.1.3 Variante 3: äußere Schleife über Pixel, Parallel über Kugeln . . . 44
5.1.4 Variante 4: äußere Schleife über Kugeln, Parallel über Kugeln . . . 45
5.1.5 Variante 5: äußere Schleife über Kugeln, Parallel über Pixel der Boundary Box 45
5.2 Vergleich der verschiedenen Ansätze . . . 46
5.2.1 Optimierung . . . 47
5.3 Vergleich mit simpler OpenGL Raycasting Implementierung mit Shadern . . . 50

6 Zusammenfassung und Ausblick **53**

Literaturverzeichnis **55**

Abbildungsverzeichnis

2.1 Vereinfachte Darstellung des Rasterization Prozesses. 12
2.2 Veranschaulichung der SIMD Architektur. 16
2.3 Hardware Architektur einer Nvidia GTX 660 auf Basis eines Kepler GK106 Chips. . . 17

3.1 In Tiles aufgeteilte Bildebene des Cuda Raycasters. 22
3.2 Speicher Coalsescing 25

4.1 Eingezeichnete Boundary Box im Cuda Raycaster. 31
4.2 Berechnung der Boundary Box. 32

5.1 Eine vom Cuda Raycaster gerendete Szene mit 10.000 Kugeln. 48
5.2 Eine vom Cuda Raycaster gerendete Szene mit 1.000.000 Kugeln. 51

Tabellenverzeichnis

5.1 Renderzeiten Variante 1 für 1000 Kugeln 43
5.2 Renderzeiten Variante 2 für 1000 Kugeln 44
5.3 Renderzeiten Variante 3 für 1000 Kugeln 44
5.4 Renderzeiten Variante 4 für 1000 Kugeln 45
5.5 Renderzeiten Variante 5 für 1000 Kugeln 45
5.6 Vergleich der Renderzeiten jedes Ansatzes mit verschiedener Gridgröße und einer Blockgröße von 32x4 46
5.7 Vergleich der Renderzeiten jedes Ansatzes mit verschiedener Kugelanzahl und einem Kugelradius von 3 Pixel 47
5.8 Vergleich der Renderzeiten jedes Ansatzes mit verschiedener Kugelanzahl und einem Kugelradius von 12 Pixel 47
5.9 Vergleich der Renderzeiten nach der Optimierung mit verschiedener Kugelanzahl und einem Kugelradius von 3 Pixel 50
5.10 Vergleich der Laufzeiten zwischen OpenGL und Cuda Implementation 50

Verzeichnis der Listings

2.1 Raycasting Pseudocode 11

4.1 Variante 1 ohne lokalen Framebuffer und ohne vorherige Sortierung der Objekte . . . 33
4.2 Variante 1 mit lokalem Framebuffer und ohne vorherige Sortierung der Objekte . . . 35
4.3 Variante 1 mit lokalem Framebuffer und vorsortierten Objekten 36
4.4 Variante 2 37
4.5 Variante 3 ohne lokalen Framebuffer und ohne Vorsortierung der Kugeln 38
4.6 Variante 3 mit lokalem Framebuffer und mit Vorsortierung der Kugeln 38
4.7 Variante 5 40

5.1 Variante 4 vor Optimierung der Sichtstrahlerzeugung 49
5.2 Variante 4 nach Optimierung der Sichtstrahlerzeugung 49

1 Einleitung

1.1 Ziel der Arbeit

In dieser Arbeit wird untersucht ob und wie ein Raycasting Algorithmus mithilfe der NVidia Cuda Architektur umgesetzt werden kann. Dabei werden einige Umsetzungsmöglichkeiten die Cuda bietet vorgestellt sowie untersucht inwiefern sich diese für eine Implementierung eines Raycasters eignen.

Anschließend werden einige Ansätze ausgewählt mit denen ein einfacher Kugel Raycaster realisiert wird. Unter anderem muss dabei überlegt werden wie man die hohe Parallelität der Cuda Architektur am besten nutzt um eine hohe Auslastung der Grafikkarte zu erreichen, oder in wie weit die Speicherbandbreite einer modernen Grafikkarte ein begrenzender Faktor ist.

Um den Raycaster zu evaluieren werden verschiedene Ansätze implementiert und ausgewertet. Für die Auswertung rendert jeder Ansatz eine Szene mit jeweils 1000, 10000, 100000 sowie 1.000.000 Kugeln. Die sich daraus ergebenden Laufzeiten der verschiedenen Ansätze werden dann miteinander vergli chen und analysiert.

Als GPU dient dazu eine Nvidia Geforce GTX 660 mit insgesamt 960 Rechenkerne welche für die parallel Berechnung des Raycasting zur Verfügung stehen.

Zuletzt wird die Performance des Raycasters mit einem simplen OpenGL Raycasters verglichen.

1.2 Bisherige Arbeiten/related work

Die Idee des Raycastings in der Computergrafik wurde schon 1968 von Arthur Appel aufgegriffen. Er beschrieb die Methode damals als „very time consuming, usually requiring for useful results several thousand times as much calculation time as a wire frame drawing“[App68, p. 4].

Der Begriff des Raycastings wurde allerdings erst 1982 erstmals von Scott Roth genutzt um damit die Methode des Renderns von „constructive solid geometry“ beschrieb [Sco82].

Raycasting stellt schon seit langer Zeit eine ziemlich gute Möglichkeit dar um hochqualitative Bilder relativ einfach zu rendern, jedoch wird der Algorithmus nicht durch Hardware unterstützt wie es beispielsweise bei Rasterisation der Fall ist. Durch viele Optimierungen auf Seiten der Software Implementation ist es möglich einen Raycasting Algorithmus auf einfachen Szenen in Echtzeit laufen zu lassen. Im Jahre 2005 stellten Sven Woop, Jörg Schmittler und Phillip Slusallek ein Design und eine Prototyp Implementation für eine „fully programmable Ray Processing Unit (RPU)“ vor. Die Prototyp Implementation mit nur 66 MHz war fähig Szenen mit 20 Bildern pro Sekunde zu rendern und schlug damit Software Lösungen die auf CPUs mit mehreren GHz liefen.

Die OpenGL Pipeline profitiert heutzutage stark von der Hardwarebeschleunigung. Die Abkehr von einer fixen zur teilweise programmierbaren Pipeline ermöglichen auch neue interessante Effekte. Einige Effekte erfordern jedoch auch die Programmierbarkeit einiger Stufen der Pipeline welche in OpenGL noch nicht programmierbar sind. Es gibt Ansätze diese Nachteile zu umgehen indem man die Grafikpipeline versucht in Cuda umzusetzen. FreePipe [LHXHEH10] stellt eine solche programmierbare parallele Pipeline dar welche mithilfe von Cuda z.B. ermöglicht Multi-Fragment Effekte umzusetzen welche mit OpenGL nicht in einem einzigen Durchlauf der Grafikpipeline möglich ist da es keinen programmierbaren Shader für die Blending Stufe gibt.

Das Rendern von Kugeln ist eine in der Naturwissenschaftlichen Forschung oft benötigte Technik um Sachverhalte und Zusammenhänge aus empirischen Daten visuell sichtbar zu machen. Partikelsimulationen generieren dabei mehrere Millionen einzelne Partikel pro Sekunde. Dabei ist es wünschenswert diese Daten in Echtzeit rendern zu können. Arbeiten wie [Rei08] versuchen die Möglichkeiten welche die OpenGL Grafik-Pipeline bietet zu optimieren um genau das zu erreichen.

2 Grundlagen

2.1 Raycasting

2.1.1 Funktionsweise

Die Idee des Raycastings basiert auf der Funktionsweise des menschlichen Auges. Statt jedoch Lichtstrahlen in das Auge hineinfallen zu lassen werden beim Racasting sogenannte Sichtstrahlen vom Augpunkt aus in die zu sehende Szene hineinprojiziert. Dazu befindet sich vor dem Augpunkt eine Bildebene bei der für jeden einzelnen Pixel dieser Ebene ein Sichtstrahl durch die Mitte des jeweiligen Pixels gesendet wird. Dieser Strahl wird mit jedem vorhandenem Objekt in der Szene geschnitten. Der Sichtstrahl liefert dann die Farb- oder Materialinformationen des zur Bildebene sich am nächsten befindlichen Objektes zurück. Mit diesen Informationen wird der betreffende Pixel dann eingefärbt.

```
function renderScene
    for each pixel (x,y):
        create Ray R from Eye to Pixel
        for each object O:
            let P be the intersection or R and O
            if P is closer than previous intersections at this pixel
            pixel(x,y) = color of P
```

Listing 2.1: Raycasting Pseudocode

Listing 2.1 zeigt das Prinzip eines einfachen Raycasters. Die äußeren Schleifen laufen dabei über jeden Pixel des Bildes. Für jeden Pixel werden dann alle Objekte der Szene auf einen Schnittpunkt untersucht. Im Falle eines vorhandenen Schnittpunktes wird überprüft ob der Pixel schon von einem näher befindlichen Objekt eingefärbt wurde. Falls nicht wird die Farbinformation des Schnittpunktes übernommen.

Wie man leicht sieht gibt es in diesem Falle insgesamt $\#Pixel * \#Objekte$ Schnittberechnungen. Dies führt selbst bei einer Szene mit nur einigen wenigen dutzend Objekten zu einer extrem hohen Berechnungsintensität. Dieser triviale Ansatz ist darum kaum geeignet um Bilder auch nur annähernd in Echtzeit zu rendern.

Es gibt jedoch einige Optimierungstechniken die dieses Problem angehen und schon einen erheblichen Geschwindigkeitszuwachs bringen. Da ein Großteil des Algorithmus aus Schnittberechnungen besteht ist die Grundidee dabei die Anzahl der Schnittberechnungen zu reduzieren. Die meisten dieser Ansätze funktionieren durch eine Partition auf den Objekten oder der Bildebene selbst. Es müssen dann nur noch die Objekte auf einen Schnittpunkt getestet werden welche sich in der jeweiligen Partition befinden die der Sichtstrahl auf seinem Weg kreuzt.

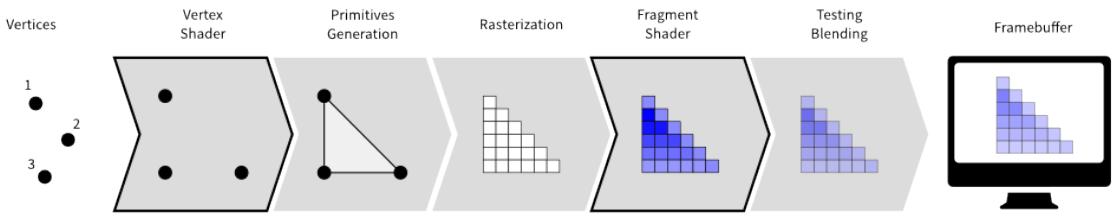

Abbildung 2.1: Vereinfachte Darstellung des Rasterization Prozesses. Vertex Daten werden von der Grafikpipeline zu einzelnen Primitiven zusammengefasst und rasterisiert. Danach eingefärbt und in den Framebuffer geschrieben. [Rou]

Damit der Renderer überhaupt Objekte rendern kann müssen sich diese zuerst einmal in Blickrichtung der Kamera befinden. Der Sichtstrahl wird dann aus dem Ursprungspunkt der Kamera und dem aktuell betrachteten Pixel der Bildebene berechnet.

Dabei gibt es einige verschiedene Kamera Modelle die dazu genutzt werden können. In der Computergrafik unterscheidet man dabei zwischen zwei verschiedenen Arten. Zum einen die parallele Projektion bei der sich alle Sichtstrahlen parallel zueinander befinden. Jeder Sichtstrahl besitzt somit die selbe Richtung und unterscheidet sich nur im Ursprungspunkt. Daraus folgt, dass sich Objekte nicht in ihrer Größe ändern wenn sie sich der Bildebene nähern oder entfernen.

Die andere Möglichkeit ist die der perspektivischen Projektion. Diese ist physikalisch akkurat und entspricht in etwa dem wie unsere Augen unsere Umgebung wahrnehmen. Dabei tritt auch ein Effekt oder eine optische Täuschung auf welche foreshortening gennant wird. Das heißt die Dimensionen eines Objektes erscheinen entlang der Sichtlinie verzerrt. Dies führt dazu dass Kugeln oft als Ellipsen, oder Quadrate als Trapeze dargestellt werden.

2.1.2 Unterschied zu Rasterisation

Moderne Grafikkarten nutzen zum rendern eines Bildes keinen Raycasting Algorithmus sondern nutzen Rasterisation unterstützt durch eine Grafik Hardware-Pipeline. Eine aus Polygonen bestehende 3D Szene wird dann wie folgt gerendert.

Die Eckpunkte jedes Polygons werden als Vertices der GPU übergeben. Diese transformiert die Vertices durch eine Serie von Translationen, Skalierungen und Rotationen auf die Bildfläche des Monitors. Die nun entstandenen 2D Koordinaten können teilweise ausserhalb des sichtbaren Bereiches des Monitors liegen. Diese herausragenden Teile werden nun durch Clipping abgetrennt. Als letztes werden die 2D Polygone mit Farbinformationen gefüllt und in den Framebuffer geschrieben. (siehe Abbildung 5.2

Der Hauptunterschied zum Raycasting besteht darin dass beim rendern der Szene nun die äußerste Schleife über die Objekte der Szene verläuft. Rasterisation erlaubt also jedes Objekt unabhängig voneinander zu betrachten. Da man dazu auch immer nur das Objekt im Speicher des Computers halten muss welches gerade gerendert wird ist es auch problemlos möglich Szenen zu rendern die in ihrer Gesamtheit nicht in den Speicher passen würden.

Da jeder Pixel nun potenziell mehrfach betrachtet wird muss man sich merken welcher Pixel von welchem Objekt überdeckt wird. Dies geschieht mithilfe eines Verfahrens namens Z-Buffering und welches in heutigen Grafikkarten in Hardware implementiert ist. Dazu wird ein weiterer Speicher in der Größe der Anzahl der Pixel bereitgestellt. Dieser wird mit Werten initialisiert welche eine unendliche Entfernung darstellen. Wenn nun ein Punkt eines aktuell gerasterten Objekts näher am Betrachter liegt als der Wert im Z-Buffer wird der Wert im Z-Buffer und im Framebuffer mit den Werten des Objektes überschrieben.

2.1.3 Unterschied Raytracing - Raycasting

Raycasting stellt im Prinzip eine einfachere Form des Raytracings dar bei dem die Szene durch rekursives Raycasting berechnet wird. Raytracing ist um ein vielfaches rechenintensiver, ermöglicht aber leichter sehr realistisch aussehende Szenen zu rendern. Spiegelnde, reflektierende oder durchsichtige Objekte können berechnet werden indem an der Stelle des Schnittpunktes einfach rekursiv ein weiterer Sichtstrahl losgeschickt wird. Zu den Primärstrahlen des Raycasting Algorithmus existieren beim Raytracing deshalb noch sogenannte Schatten-, Spiegelungs- und Brechungsstrahlen die alle zu den sogenannten Sekundärstrahlen gehören. Diese unterschieden sich dadurch dass sie im Gegensatz zum Primärstrahl nicht vom Augpunkt ausgehen.

Findet beim Raytracing der Primärstrahl nun einen Schnittpunkt mit einem Objekt werden an der Stelle des Schnittpunktes Spiegelungs- sowie Brechungsstrahlen ausgesendet. Schattenstrahlen widerrum werden nicht vom Schnittpunkt aus, sondern von einer Lichtquelle zum Schnittpunkt gesendet um zu testen ob sich ein Objekt zwischen Lichtquelle und Schnittpunkt befindet, und dadurch ein Schatten auf den Schnittpunk geworfen wird.

2.1.4 Objekt Schnittberechnung

Der rendering Prozess des Raycastings besteht hauptsächlich aus der Berechnung einer Kollisionsdetektion zwischen Objekten und Strahlen. Die Strahlen werden von der Kamera aus zu jedem Pixel des Bildes gesendet und auf einen vorhandenen Schnittpunkt mit einem Objekt getestet. Dieser Schnittpunkttest unterscheidet sich je nach zu schneidendem Objekt. Aus diesem Grund muss für jedes Objekt welches der Raycaster unterstützt eine eigene Schnittpunkt-Test Funktion implementiert werden. Dreiecke sowie Kugeln gehören zu den am meisten unterstützten Objekten eines Raycasters, weshalb wir kurz die Vorgehensweise für die Schnittpunktberechnung an diesen beiden Objekten nachvollziehen.

Dreieck Schnittberechnung

Für die Schnittberechnung zwischen einem Sichtstrahl und einem Dreieck gibt es verschiedene Ansatzmöglichkeiten. Der Ansatz von Möller und Trumbore [MT97] kommt ohne vorherige Berechnung der Ebenengleichung für die Dreiecke aus und funktioniert wie folgt.

Ein Punkt innerhalb des Dreiecks wird folgendermaßen definiert.

$$p(u, v) = (1 - u - v) * p0 + u * p1 + v * p2$$

wobei p0, p1 sowie p2 die Vertice des Dreiecks darstellen und folgende Bedingungen erfüllt sein müssen.

$$u >= 0$$

$$v >= 0$$

$$u + v <= 1.0$$

Sei nun ein Sichtstrahl mit der Gleichung $x = o + t * d$ mit mit Ursprung o, der Länge t sowie dem Richtungsvektor d gegeben. Ein gemeinsamer Punkt des Strahls und des Dreiecks erfüllt dann folgende Gleichung.

$$o + t * d = (1 - u - v) * p0 + u * p1 + v * p2$$

Es muss also ein Tripel (t, u, v) gefunden werden welches diese Gleichung löst und gleichzeitig die oben genannten Bedingungen erfüllt.

Kugel Schnittberechnung

Um den Schnittpunkt eines Sichtstrahls und einer Kugel zu berechnen müssen die Punkte berechnet werden welche auf dem Strahl sowie der Kugel liegen. Dazu setzt man die Linien Gleichung des Sichtstrahls in die Kugelgleichung ein uns löst nach der Länge auf.

Sei eine Linie mit der Gleichung $x = o + td$ gegeben. Mit Ursprung o, der Länge t sowie dem Richtungsvektor d. Dazu eine Kugel mit der Kugelgleichung $(||x - c||)^2 = r^2$ mit Ursprung c, und Radius r. Nach Einsetzen der Liniengleichung in die Kugelgleichung ergibt sich

$$(||o + td - c||)^2 = r^2 \Leftrightarrow (o + td - c) \cdot (o + td - c) = r^2$$

Ausmultiplizieren ergibt

$$t^2(d \cdot d) + 2t(d \cdot (o - c)) + (o - c) \cdot (o - c) = r^2$$

Dann subtrahiert man auf beiden Seiten r^2

$$(d * d)t^2 + 2(d \cdot (o - c))t + (o - c) \cdot (o - c) - r^2$$

Daraus ergibt sich eine Quadratische Formel der Art

$$ad^2 + bd + c \Leftrightarrow d = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

mit

$$a = d \cdot d = ||d||^2$$

$$b = 2 \cdot (d \cdot (o - c))$$

$$c = (o - c) \cdot (o - c) - r^2 = (||o - c||)^2 - r^2$$

Alle Werte eingesetzt ergibt

$$d = \frac{-(2 \cdot (d \cdot (o - c))) \pm \sqrt{(2 \cdot (d \cdot (o - c))^2 - 4 \cdot ||d||^2 \cdot ((||o - c||)^2 - r^2}}{2 \cdot ||d||^2}$$

Da es sich bei $||d||^2$ um einen Richtungsvektor handelt welcher normalerweise normalisiert ist vereinfacht sich der Term zu 1. Durch weitere Vereinfachungen kommt man zu

$$d = -(d \cdot (o - c)) \pm \sqrt{((d \cdot (o - c))^2 - (||o - c||)^2 - r^2}$$

Im Fall dass sich der Term unter der Wurzel zu einer negativen Zahl evaluiert existiert kein Schnittpunkt. Evaluiert der Term sich zu 0 existiert genau ein Schnittpunkt mit der Distanz

$$d = -(d \cdot (o - c))$$

Sollte der Term sich zu einer Zahl größer als Null evaluieren existieren zwei Schnittpunkte mit der Kugel.

$$d_1 = -(d \cdot (o - c)) + \sqrt{((d \cdot (o - c))^2 - (||o - c||)^2 - r^2}$$

$$d_2 = -(d \cdot (o - c)) - \sqrt{((d \cdot (o - c))^2 - (||o - c||)^2 - r^2}$$

2.2 Cuda

2.2.1 Einleitung

Cuda ist eine von NVidia entwickelte Architektur für parallele Berechnungen auf GPUs. Anstatt wie bisher üblich Berechnungen auf Vertex und Pixel Shader aufzuteilen basiert die Cuda Architektur auf einer sogenannten „unified shader pipeline“ [SK12, p. 7] die es erlaubt jede ALU auf dem Grafikkartenchip für Allgemeine Berechnungen zur Verfügung zu stellen. Die erste Grafikkarte die diese neuartige Architektur unterstützte ist die Geforce 8800 GTX aus dem Jahre 2006.

Damit Programmierer Allgemeine Berechnungen nicht als Grafikprobleme tarnen, und Daten über Texturen austauschen müssen wurde zu diesem Zweck eine eigene Programmiersprache entwickelt die den Umgang mit Cuda erleichtern soll. Dabei handelt es sich um Cuda C die sich im Kern der Sprache C bedient und um einige extra hinzugefügte Schlüsselwörter erweitert wurde. Diese Erweiterungen ermöglichen es auf die speziellen Eigenschaften der Cuda Architektur zuzugreifen.

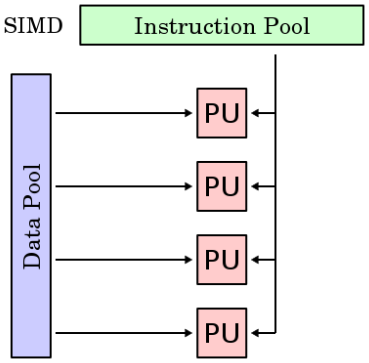

Abbildung 2.2: Veranschaulichung der SIMD Architektur. Auf verschiedenen Datensätzen wird von verschiedenen Kernen die gleiche Instruktion ausgeführt. [wik]

Seit der Veröffentlichung 2006 von Cuda mit compute capabillity 1.0 wurde die Cuda Architektur stetig weiterentwickelt und ist zum Zeitpunkt in compute capabillity 5.2 erhältlich. Hierbei muss zwischen der Cuda compute capabillity und dem Cuda Toolkit unterschieden werden welches aktuell in Version 6.5 erhältlich ist. Die Cuda compute capabillity bezieht sich rein auf die Hardware Spezifikationen einer gegebene GPU Architektur während das Cuda Toolkit eine Softwaresammlung für die Programmierung von Cuda Programmen darstellt.

2.2.2 CPU vs GPU

Aktuelle CPUs bestehen heutzutage meistens aus vier bis acht Kernen die parallel Berechnungen ermöglichen. GPUs hingegen besitzen mittlerweile an die hunderte oder sogar tausende Kerne für die parallele Berechnung. Das bedeutet jedoch nicht automatisch dass eine GPU bei allen Berechnungen um den Faktor hundert oder gar tausend schneller rechnet. CPUs sind spezialisiert auf die Abarbeitung von langem, aus vielen unterschiedlichen Befehlen und Abzweigungen bestehendem Code. Anwendungen für CPUs besitzen somit häufig auch eine hohe Datenlokalität und benötigen deshalb eher eine gute Cache Struktur.

GPUs werden jedoch hauptsächlich mit Blick auf den Computerspiele Bereich entwickelt bei dem die Anforderungen im gleichzeitigen Berechnen von mehreren Millionen Polygontransformationen liegt. Es handelt sich dabei also um Programme bei denen der gleiche Befehl auf einen riesigen Speicherbereich ausgeführt wird. Das parallele Ausführen möglichst vieler Befehle ist für Grafikkarten deshalb essentiell um Computerspiele in Echtzeit darstellen zu können. Die darauf basierende Architektur wird SIMD (Single Instruction, Multiple Data) Architektur genannt. (Abbildung 2.3)

Eine leichte Abwandlung oder auch Erweiterung von SIMD stellt die SIMT (Single Instruction, Multiple Threads) Architektur dar die in Nvidia GPUs zum Einsatz kommt.

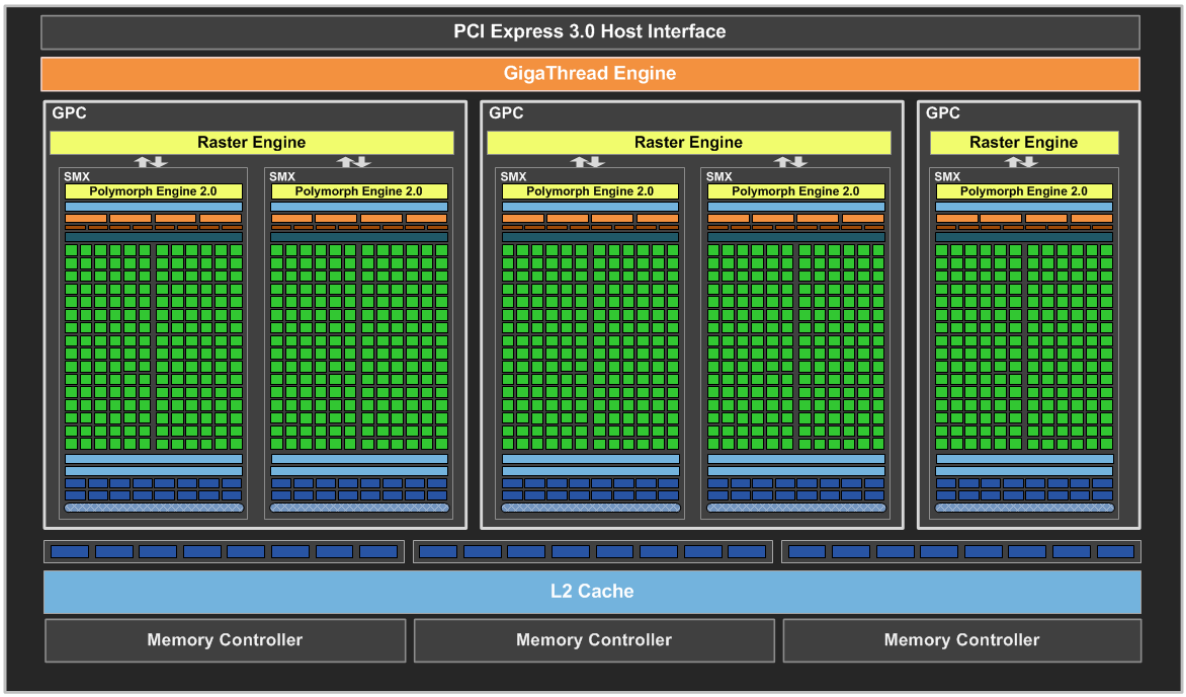

Abbildung 2.3: Hardware Architekur einer Nvidia GTX 660 auf Basis eines Kepler GK106 Chips. Dieser enthält fünf Streaming Multiprozessoren die sich auf drei GPC verteilen. Jeder Multiprozessor enthält seinerseits 192 Recheneinheiten die sich zu einer Gesamtanzahl von 960 Recheneinheiten auf der GPU addieren. [Rya]

2.2.3 Nvidia GPU Hardware Aufbau

Auf einem CPU Kern wird nur ein Teil des Kerns für die Recheneinheiten verwendet. Ein Großteil belegen Cache und Control Strukturen. Aufgrund ihrer massiven parallelen Arbeitsweise benötigen GPUs jedoch keine komplexen Cache und Control Strukturen, weshalb der Kern einer GPU zum größten Teil aus vielen einzelnen Recheneinheiten besteht.

Die Recheneinheiten auf der GPU sind dabei auf sogenannte Streaming Multiprozessoren (SM) aufgeteilt. Jede SM besteht dabei aus einer Vielzahl an Kernen die sich die Ressourcen wie z.B. die verfügbaren Register der SM teilen.

2.2.4 Cuda C

Um die hohe Anzahl an Kernen effektiv nutzen zu können bedarf es ein Prorammiermodel welches sicherstellt dass Systemressourcen synchron arbeiten und keine unnötige Arbeit verrichten. Cuda stellt dabei ein stark skalierbares Programmiermodel dar welches es ermöglicht die hohe Parallelität einer GPU zu nutzen.

Die für Cuda eigens entwickelte Programmiersprache „Cuda C“ stellt eine Erweiterung der Sprache C dar. Der für eine Cuda Anwendung geschriebene Code wird dabei zwischen Host und Device Code unterschieden. Der Host Code stellt den gewöhnlichen auf der CPU ausführbaren Code dar während der Device Code jegliche Funktionen beinhaltet welcher auf der GPU ausgeführt werden soll.

Die auf der GPU ausgeführten Funktionen werden Kernel genannt und auf Seiten des Host Codes durch eine spezielle Syntax aufgerufen. Jedem Kernel werden dabei Parameter für die Anzahl an Blöcken und Threads mitgegeben welche für den Kernel Aufruf gestartet werden sollen.

Ein Cuda Kernel besteht also aus einem dreidimensionalem Grid von Blöcken welche auf der GPU ausgeführt werden. Jeder dieser Blöcke wiederum stellt eine dreidimensionale Anordnung von Threads dar. Selbstverständlich muss nicht zwingend jede Dimension genutzt werden. Insbesondere Anwendungen die im Bildbereich operieren sind oftmals am besten mit einem zweidimensionalem Grid zu realisieren da man den Code dann an das Problem anpassen kann.

2.2.5 Cuda Memory Model

Um die Rechenleistung moderner GPUs effizient auszunutzen bedarf es auch den passenden Speicher um die Daten der Berechnungen zu halten. Für Allgemeine Berechnungen ist der Texturspeicher der Grafikkarte jedoch nicht die beste Wahl. Aus diesem Grund bietet Cuda verschiedene Speichertypen an die genutzt werden können.

Zu unterscheiden ist erstmal zwischen Host und Device Speicher. Der Host Speicher ist der Speicher der mit der CPU verbunden ist. Cuda bietet für diese Art Speicher auch APIs an um den Zugriff darauf zu beschleunigen, oder gar den Speicherbereich für den GPU Zugriff zuzuordnen. Device Speicher ist der physikalisch zur Grafikkarte gehörende Speicher. Um Daten nun für die Berechnung auf einer GPU zu nutzen müssen diese vom Host Speicher zuerst auf den Speicher der Grafikkarte kopiert werden.

Es gibt dabei verschiedene Arten von Device Memory der alloziert werden kann.

- Globaler Speicher der entweder statisch oder dynamisch alloziert wird und auf den mit Pointer zugegriffen werden kann. Die Pointer werden beim Kernelaufruf als Parameter angegeben und können dann im Kernelcode genutzt werden.
- Konstanter Speicher stellt einen read-only Speicher bereit der durch eine Cache-Hierarchie für einen Thread-weiten Broadcast optimiert ist.
- Lokaler Speicher beinhaltet den Stack. Hier landen nicht nur return Adressen von Unterfunktionen sondern auch lokale Variablen für die keine freien Register mehr zur Verfügung stehen. Jeder Thread besitzt dazu einen eigenen lokalen Speicher.
- Textur Speicher auf den durch spezielle Textur und surface load/store Anweisungen zugegriffen werden kann. Wie auch schon beim Konstanten Speicher ist dieser Speicher für read-only optimiert und besitzt einen Cache.

Ein weiterer noch nicht erwähnter aber sehr wichtiger Speicher stellt der Shared Speicher dar. Im Gegensatz zu den bisherigen Device Speicherarten befindet sich dieser nicht mit auf den selben Speichermodulen sondern stellt eine Art on-chip Speicher auf jeder SM dar. Dieser dient für einen schnellen Datenaustausch zwischen Threads innerhalb eines Blockes.

2.2.6 Parallelität

Die parallele Funktionsweise von Cuda während der Berechnung funktioniert über sogenannte Warps. Dabei besteht jeder Warp aus 32 Threads. Die Warp Scheduler des SM verteilen die Threads auf die einzelnen Recheneinheiten. Jede Recheneinheit führt dann mit den zur Verfügung stehenden Ressourcen die Instruktionen des Kernels aus.

Ein wichtiges Detail dabei ist, dass alle 32 Threads innerhalb eines Warps zwar die selbe Instruktion ausführen, dabei jedoch meistens auf unterschiedliche Daten aus dem Speicher zugreifen. Sollten die einzelnen Threads durch Verzweigungen im Programmverlauf divergieren, so werden einzelne Threads deaktiviert und nach einer Abzweigung im Programmcode wieder zusammengeführt. An keiner Stelle führt also ein Thread eine Instruktion früher aus als anderen Threads.

Die Herausforderung bei der Programmierung mit Cuda besteht deshalb darin den zu implementierenden Algorithmus so zu gestalten dass alle Kerne der GPU möglichst gleich stark ausgelastet sind. Auch sollte so weit es möglich ist auf Abzweigungen verzichtet werden um eine gute Performance zu gewährleisten.

2.2.7 GPU Eignung für Raycasting

Durch die im Verhältnis zu CPUs überlegene Anzahl an Prozessorkernen sind GPUs geradezu prädestiniert dafür um rechenintensive parallele Berechnungen durchzuführen. Die Hauptlast bei der Raycasting Berechnung liegt wie erwähnt im Schnittpunkt Test mit den Objekten der Szene. Da jeder Pixel des Bildes dabei getrennt voneinander betrachtet werden kann ist eine parallele Umsetzung problemlos möglich. Cuda bietet sich für eine Raycasting Implementierung also bestens an. So könnte z.B. für jeden Pixel des Bildes ein eigener GPU Thread gestartet werden der sich dann um alle Schnittpunkt Tests für diesen Pixel kümmert.

2.2.8 Unterschied zu OpenGL mit programmierbaren Shadern

OpenGL arbeitet wie schon erwähnt mit einem Rasterisation Ansatz bei dem einzelne Vertices zu Primitiven zusammengefasst und nach dem rasterisieren Pixelweise eingefärbt werden. Bei einer fixen Grafikpipeline ist ein Raycasting Algorithmus nicht umsetzbar da OpenGL sich automatisch um die Generierung der Primitiven und die Einfärbung der Fragmente kümmert. Der Programmierer stellt lediglich die benötigten Vertices zur Verfügung und übergibt OpenGL die benötigten Parameter für Transformationen oder Beleuchtungsmodelle.

Einige wenige Schritte dieser automatisierten Pipeline sind mittlerweile unter OpenGL jedoch programmierbar. Die Einfärbung der einzelnen Pixel kann mit einem programmierbaren Fragment

Shader manuell festgelegt werden. Dieser operiert dann Pixelweise auf den Pixeln des übergebenen Fragments.

Für einen Raycasting Algorithmus möchten wir jedoch die Pixel in der Bildebene nacheinander abtasten und untersuchen ob ein Strahl der durch diesen Pixel gesendet wird ein Objekt schneidet, oder nicht. Dies lässt sich mit einem kleinen Trick leicht in einem Fragment Shader umsetzen. Wir definieren uns dafür ein Bildschirmfüllendes Quadrat dessen Vertices wir der OpenGL Pipeline übergeben. Da diese Vertices die Eckpunkte unserer Bildebene darstellen lassen wir den Vertex Shader keinerlei Transformationen auf diesen Vertices ausführen, sondern geben diese so wie sie sind an den nächsten Pipeline Prozess weiter. Dass daraus entstehende Fragment ist also ein Bildschirmumfassendes Quadrat welches wir nun im Fragment Shader Pixelweise einfärben können.

Im Fragment Shader wird dann mithilfe von eingebauten OpenGL Variablen abgefragt an welcher Pixelposition innerhalb des Fragments wir uns befinden. Da das Fragment die gesamte Bildebene umfasst sind dies dann auch gleich die Pixelpositionen unserer Bildebene mit denen wir uns den benötigten Sichtstrahl berechnen können.

Es werden nun nur noch die Objektdaten der Szene benötigt. Unter OpenGL ist es jedoch nicht ohne weiteres möglich beliebige Daten auf die Grafikkarte zu laden um darauf im Fragment Shader zugreifen zu können. Aus diesem Grund müssen diese Daten als Textur getarnt auf die Grafikkarte geladen werden. Im Fragment Shader ist es dann möglich beliebige Positionen auf der Textur auszulesen. Diese müüssen dann nur noch entsprechend intepretiert werden.

3 Umsetzungsmöglichkeiten in Cuda

3.1 Tile basierter Ansatz

Eine naive Umsetzung des Raycasting Algorithmus führt für jeden Sichtstrahl so viele Schnittpunkt Tests aus wie es Objekte in der Szene gibt. Da pro Pixel ein Sichtstrahl gesendet wird ergibt das $pixel_{width} * pixel_{height} * n_{objects}$ Schnittpunkt Tests pro Bild.

Anzahl Objekte	640 * 480	1024 * 1024
10	3.072.000	10.485.760
1.000	307.200.000	1.048.576.000
100.000	30.720.000.000	104.857.600.000
1.000.000	307.200.000.000	1.048.576.000.000

Unsere Cuda Implementierung des Raycasting Algorithmus wird mit einer Szene der Auflösung 1024 * 1024 getestet die bis zu 1.000.000 Objekte enthält. Bei einer naiven Implementierung werden also über 1 Billionen Schnittpunkt Tests ausgeführt. Selbst mit der sehr hohen Leistungsfähigkeit und Parallelität einer modernen GPU ist absehbar dass das Rendern eines einzelnen Bildes mehrere Minuten dauern kann.

Betrachtet man noch einmal den GPU Aufbau sowie die Cuda Architektur scheint ein Tile-basierter Ansatz die beste Möglichkeit darzustellen um den Algorithmus zu beschleunigen. Jedes Tile in unserem Algorithmus kann durch einen einzelnen Cuda Block gerendert werden. Durch eine Vorsortierung der Objekte muss jeder Block dann nur noch über die Objekte laufen die sich innerhalb des dazugehörigen Tiles befinden. Dieser Ansatz skaliert auch sehr gut mit späteren GPUs welche eine höhere Anzahl an SMs sowie Prozessorkernen besitzen. Die Größe eines Tiles kann dann einfach verringert werden um die noch höhere zur Verfügung stehende Parallelität optimal auszunutzen.

3.2 Äußere Schleife über Kugeln oder Pixel

Nun stellt sich auch die Frage wie viele Threads man einem Tile zuweist und ob die einzelnen Threads in der äußeren Schleife über die Objekte der Szene oder über die Pixel des Bildes gehen sollen. Da der Raycasting Algorithmus die einzelnen Pixel als kleinstes unabhängiges Element besitzt liegt die Vermutung nahe die äußere Schleife auch über die Pixel laufen zu lassen.

Wir werden jedoch beide Varianten betrachten und vergleichen in wieweit Cuda sich für die jeweilige Variante eignet und welche sich für einen Echtzeit Raycaster am besten eignet. Dabei können wir

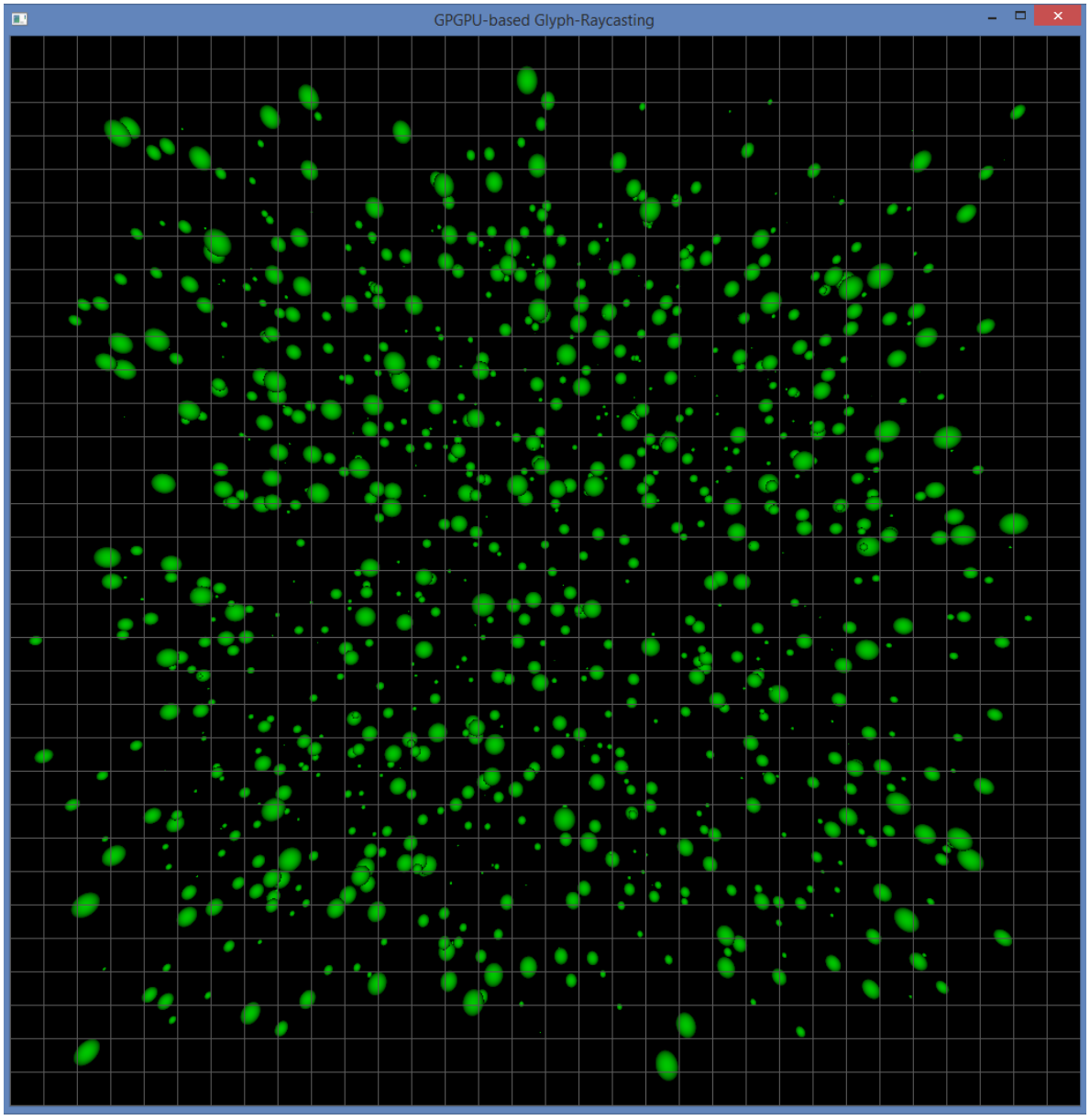

Abbildung 3.1: Eine in 32x32 Tiles aufgeteilte Bildebene des Cuda Raycasters. Jedes Tile rendert nur die in dem Tile befindlichen Kugeln.

grob 4 unterschiedliche Fälle betrachten auf die wir bei der Implementierung aber nochmals näher eingehen werden.

3.2.1 Variante 1: Die äußere Schleife läuft über die Pixel des Bildes und es werden verschiedene Pixel gleichzeitig gerendert.

Dies ist die Variante welche man intuitiv als erstes wählen würde. Jeder Cuda Warp rendert dabei 32 hintereinander folgende Pixel gleichzeitig. Der Vorteil dabei ist, dass dieser Ansatz sehr gut mit der Anzahl der Warps skalierbar ist. Dazu wählt die Grid und Block Parameter am besten so dass die Breite eines Tiles einem vielfachen der Warp Größe entspricht. Als Beispiel nehmen wir an die Breite und Höhe eines Tiles beträgt jeweils 32 Pixel. Wird diesem Tile nun ein einzelner Warp zugewiesen wird die äußere Schleife genau 32 Mal ausgeführt. Bei Zuweisung von 2 Warps entsprechend nur noch 16 Mal usw.

3.2.2 Variante 2: Die äußere Schleife läuft über die Objekte der Szene und es werden verschiedene Pixel gleichzeitig gerendert.

Im Prinzip entspricht dies der Variante 1. Die vertauschten Schleifen bewirken dass Instruktionen wie z.B. die Überprüfung ob ein Objekt innerhalb des Tiles liegt nur noch einmal pro Objekt ausgeführt werden. In der vorherigen Variante muss man durch zusätzlichen Code sicherstellen dass dies der Fall ist da die Überprüfung ansonsten in der inneren Schleife für jeden Pixel neu ausgeführt wird. Und das auch wenn es sich dabei um das selbe Objekt wie zuvor handelt. Diese Variante müsste also minimal schneller sein als Variante 1.

3.2.3 Variante 3: Die äußere Schleife läuft über die Pixel des Bildes und es werden verschiedene Objekte gleichzeitig gerendert.

Im Gegensatz zu Variante 1 werden nun aufeinanderfolgende Objekte anstatt Pixel gleichzeitig betrachtet. Ein Problem welches dabei sofort auftritt ist dass nun von verschiedenen Threads ein und derselbe Pixel gerendert wird. Es muss also sichergestellt werden dass nur derjenige Thread den Pixel rendert welcher das zur Bildebene am nächsten befindliche Objekt hat. Dazu ist ein Informationsaustausch zwischen den Threads zwingend notwendig.

3.2.4 Variante 4: Die äußere Schleife läuft über die Objekte der Szene und es werden verschiedene Objekte gleichzeitig gerendert.

Nach vertauschen der beiden Schleifen aus Variante 3 erhalten wir den gleichen Vorteil wie auch schon bei Variante 2. Ansonsten bleibt zu sagen dass erwartet wird dass Variante 3 und 4 aufgrund der notwendigen Synchronisation zwischen den einzelnen Threads langsamer laufen werden als die beiden vorherigen Varianten.

Variante 2 verspricht nach dieser ersten Analyse das größte Potential. Ob und wie groß der Unterschied zu den anderen Varianten tatsächlich ausfällt werden wir später genauer untersuchen.

3.3 Daten Streaming

Cuda bietet die Möglichkeit des „mapped pinned host Memory“. Dabei handelt es sich um allozierten Speicher auf Host Seite der auf den Cuda Adressbereich abgebildet wird und vom Betriebssystem als „page-locked“ initialisiert wird damit das Betriebssystem diesen Speicherbereich nicht auslagert. Dadurch ist es möglich von einem Cuda Kernel aus direkt in den Host Speicher zu schreiben oder davon zu lesen. Die Daten im Speicher des Hosts können also genutzt werden ohne diese vorher auf die GPU kopieren zu müssen.

Vorteile:

- Der GPU Speicher wird durch das Zur-Verfügung-Stellen von Host Speicher quasi erweitert. Anwendungen bei denen die GPU mehrere Gigabyte an Daten verarbeiten muss können dadurch profitieren da die Daten nicht im GPU Speicher gehalten werden müssen.
- Der Speicherzugriff findet asynchron statt und blockiert dadurch nicht den Codeverlauf auf Host Seite.

Nachteile:

- Der Zugriff auf Host Seite auf diesen Speicher erfordert eventuell eine vorherige Synchronisation mit der GPU.
- Der Speicher ist auf der GPU Seite nicht gecached.

Der Ansatz des „mapped pinned host Memory“ erscheint sinnvoll bei einer großen Datenmenge von der man im voraus zwar weiß dass die GPU diese nur teilweise benötigt, der genaue Datenzugriff aber erst zur Laufzeit bekannt ist. Oder falls man weiß, dass alle Daten genau einmal benötigt werden und deshalb nicht gecached werden müssen.

Unsere Daten des Raycasters bestehen hauptsächlich aus Daten welche die Objekte innerhalb der Szene beschreiben. Wir wissen auch dass jedes Objekt zumindest darauf untersucht werden muss ob es denn gerendert werden soll. Die Objekt Daten deshalb nur bei Bedarf auf die GPU zu streamen bringt keinen Vorteil sofern die GPU genügend Speicher besitzt um die gesamte Szene zwischenzuspeichern.

3.4 Speicher Coalsescing Constraints

Das Lesen und Schreiben von globalem Speicher auf der GPU bestraft unvorteilhafte Speicherzugriffe mit teilweise starken Leistungseinbußen. Dies gilt jedoch hauptsächlich für Hardware mit SM 2.x oder niedriger. Ab SM 3.x fallen die Leistungseinbußen eher moderat aus. Bei Anwendungen mit sehr hohen Performance Ansprüchen kann dies jedoch einen spürbaren Einfluss haben. Um das zu vermeiden sollten beim Zugriff auf diesen Speicher einige „coalescing constraints“ eingehalten werden. Diese Bedingungen gelten auf Warp Basis und lauten wie folgt.

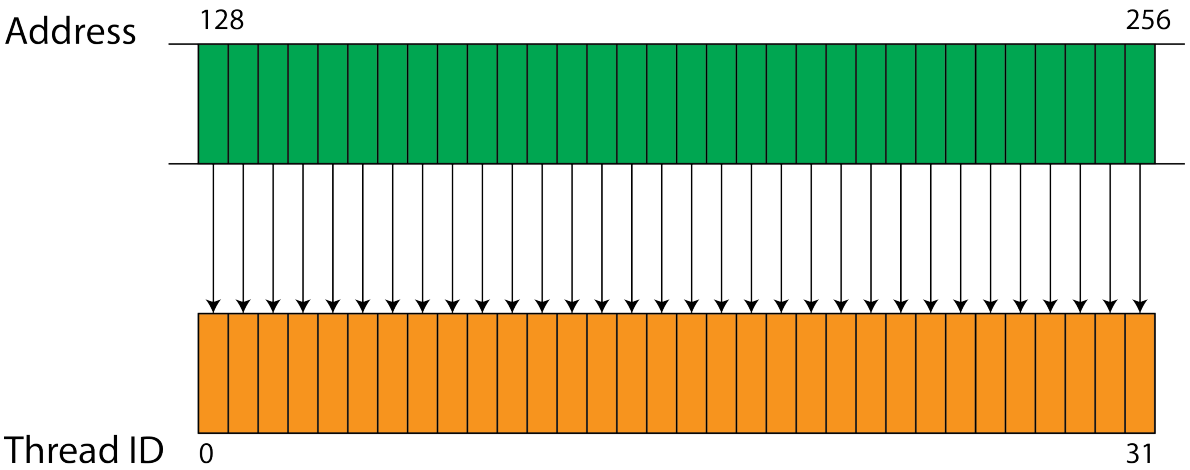

Abbildung 3.2: Beispiel für einen coalescing Speicherzugriff. [cud]

- Gelesene oder geschriebene Wörter müssen eine Mindestgröße von 32 Bit besitzen.
- Die Adressen auf die zugegriffen werden müssen nachfolgend und aufsteigend sein
- Die Basisadresse der gesamten Speicher Transaktion eines Warps muss bündig ausgerichtet sein.

Normalerweise hat man damit keinerlei Probleme da es im Prinzip nur darum geht dass die Threads innerhalb eines Warps bei einem Speicherzugriff nebeneinanderliegende Speicheradressen abfragen. Der Speicherzugriff erfolgt dann in einer einzigen Transaktion. Abbildung 4.2 zeigt wie ein Speicherzugriff aussieht der alle Bedingungen erfüllt. Würde die Basisadresse Beispielsweise bei 129 statt 128 liegen wäre die dritte Bedingung verletzt und der Speicherzugriff wird dann nicht mehr in einer einzelnen Transaktion ausgeführt.

3.5 Ausnutzen von shared oder constant memory

Cuda bietet uns für Berechnungen einen sehr schnellen on-chip shared Memory Bereich der von allen Threads innerhalb eines Warps geteilt wird. Ein Informationsaustausch zwischen einzelnen Threads wäre also möglich und für die obigen besprochenen Varianten 3 und 4 auch notwendig um die Pixel korrekt zu rendern.

Da wir einen tile-basierten Ansatz implementieren kann der shared Memory auch dazu genutzt werden um die Farbinformationen jedes Tiles zwischenzuspeichern um diese dann am Ende gesammelt in den Framebuffer zu schreiben.

Der Konstante Speicherbereich der Cuda uns zur Verfügung stellt unterliegt einigen Beschränkungen um wirklich hilfreich zu sein. Zum einen müssten alle Threads innerhalb eines halb-warps die selbe Adresse anfragen da die Anfragen ansonsten serialisiert werden. Und zum anderen muss es sich dabei um Daten handeln die sich während eines Kernelaufrufes nicht ändern. Da die Größe des Konstanten

Speicherbereichs mit 64kb eher gering ausfällt ist es auch nicht möglich große Teile der Szene darin zu speichern.

3.6 Z-Buffer und lokaler Framebuffer

Wie gerade schon erwähnt ist es für unseren tile-basierten Ansatz möglich zuerst alle Farbinformationen zwischenzuspeichern bevor diese in den Framebuffer geschrieben werden. Da sich der Framebuffer im globalen Speicherbereich der Grafikkarte befindet und dieser beim schreiben eine höhere Latenz besitzt als der shared Memory kann es durchaus sinnvoll sein den shared Memory Bereich als Zwischenspeicher zu nutzen. Nicht jeder Thread innerhalb eines Warps würde nämlich in jedem Schleifendurchlauf in den Framebuffer schreiben. Falls jedoch innerhalb eines Warps nur ein einzelner Thread einen positiven Schnittpunkttest zurückliefert werden alle anderen Threads innerhalb des Warps deaktiviert bis dieser Thread seine Farbinformationen in den Framebuffer geschrieben hat. Was dann einerseits zu einer schlechten Auslastung und damit zu Leistungseinbußen führt.

4 Implementierung

4.1 Einleitung

Der Prozess des Raycasting ist ein relativ simpel zu implementierender Algorithmus. Für einen einfachen Raycaster werden nur wenige Komponenten benötigt. Eine virtuelle Kamera, die Sichtstrahlen mit einem Ursprungspunkt und einer Richtung, Objekte die in der Szene platziert werden und ein zweidimensionales Pixel Array welches das leere Bild darstellt.

Dieser Abschnitt beschreibt den Aufbau des in Cuda Implementierten Raycasters und es werden einige wichtige Aspekte der GPU Programmierung erläutert.

Das Programmieren in Cuda erfordert eine andere Herangehensweise als es z.B. bei der CPU Programmierung mit C++ der Fall ist. Man muss sich vor Beginn der Programmierung schon im Klaren sein dass man für viele parallele Prozessorkerne programmiert und den Code von vornherein darauf auslegen, da die Parallelität im Code über den Zugriff spezieller Thread IDs geregelt wird. Bei der CPU Programmierung muss man sich in dieser Hinsicht eher wenige bis gar keine Gedanken darüber machen.

In unserem Beispiel des Raycasters müssen wir uns also überlegen wie wir das rendern der einzelnen Pixel des Bildes am besten parallelisieren um die verschiedenen Aufteilungsmöglickeiten von Blöcken und Threads am besten ausnutzen zu können.

Wenn Sichtstrahlen in die Szene hinein projiziert werden ist es das Ziel dieser Sichtstrahlen einen einzelnen Farbwert für jeden Pixel zurückzuliefern. Dabei wird der Farbwert eines Pixels nicht durch nebenan liegende Pixel beeinflusst. Dabei müssen also Strahl-Objekt Schnittpunkt Berechnungen über alle Objekte der Szene gegen alle Sichtstrahlen ausgeführt werden. Das kleinste unabhängige Element stellt der Pixel dar und es bietet sich an die Parallelität des Renderns auf die Pixel aufzuteilen. Jeder GPU Thread berechnet also die Ausgangsfarbe eines einzelnen Pixels. Dabei kümmert sich jeder Thread um das Shading und darum den Sichtstrahl des Pixels mit allen Objekten der Szene zu schneiden. Das bedeutet auch das jeder Thread Zugang zur gesamten Szenebeschreibung benötigt um den Pixel rendern zu können.

4.2 Variablen und Datentypen

Bevor die GPU Berechnungen für unseren Raycaster ausführen kann benötigt diese zuerst einmal die gesamten Szenedaten in einem Format das Cuda unterstützt. Cuda besitzt dabei nur limitierte Unterstützung für Klassen und keinen Support für virtuelle Funktionen. Es ist daher unter anderem nicht erlaubt einem Cuda Kernel eine Klasse als Parameter mitzugeben die von einer virtuellen

Basisklasse erbt. Die Klasse darf selbst auch keine virtuellen Funktionen besitzen. Falls der Raycaster unterschiedliche Objekte unterstützt müssen diese im Kernel Code separat gehandhabt werden da keine virtuelle hit Funktion für die Objekte implementiert werden kann.

4.3 Speicher Initialisierung auf der GPU

Um überhaupt Daten von der CPU auf die GPU kopieren zu können ist es nötig vorher Pointer für den GPU Speicher zu deklarieren. Über Cuda spezifische malloc sowie memcpy Funktionen kann dann GPU Speicher alloziert und Daten auf die GPU kopiert werden. Das muss alles vor dem Aufruf des GPU Kernels geschehen. Der Zugriff auf die Daten erfolgt dann über die dafür deklarierten Pointer welche dem Kernelaufruf als Parameter mitgeliefert werden müssen.

4.4 Z-Buffer und lokaler Framebuffer

Beide, der Z-Buffer sowie der lokale Framebuffer werden durch einen shared memory realisiert. Da sich die Größe der beiden Buffer nach der Größe eines einzelnen Tiles richtet und sich diese Größe während der Laufzeit ändern kann muss der shared memory Bereich dynamisch alloziert werden. Cuda erlaubt dabei jedoch pro Block nur einen dynamisch allozierten shared Memory. Aus diesem Grund wird vor dem Ausführen des Kernels die Gesamtgröße beider Buffer berechnet. Dieser setzt sich pro Block zusammen aus der Pixelbreite * Pixelhöhe eines Tiles mal genommen mit der Anzahl bytes eines float Datentyps für den Z-Buffer. Dazu addiert sich nun noch die benötigte Anzahl bytes für den lokalen Framebuffer.

Da der Framebuffer aus RGB Werten besteht werden dafür pro Pixel drei bytes benötigt. Also Pixelbreite * Pixelhöhe eines Tiles mal genommen mit drei mal der Größe eines unsigned chars.

Der shared Memory wird innerhalb des Kernels dann wie folgt definiert.

```
extern __shared__ float buffer [];
```

Das extern signalisiert dass es sich dabei um dynamisch allozierten Speicher handelt dessen Größe zur Laufzeit dem Kernel als Parameter mitgegeben wird. Die Unterscheidung der beiden benötigten Buffer erfolgt dann durch zwei zusätzliche Pointer die auf den jeweiligen Startbereich der einzelnen Buffer zeigt.

```
float *zBuffer = (float*)&buffer;
unsigned char *localFrameBuffer = (unsigned char*)&buffer[ tileSize ];
```

Da die Werte im Buffer noch undefiniert sind ist es notwendig diese vor der Nutzung im Kernel mit sinnvollen Werten zu füllen. Unser Z-Buffer speichert für jeden Pixel die Entfernung des zur Bildebene nächsten Objektes, weshalb der Z-Buffer mit einem Wert initialisiert wird an der sich die hintere Clippingebene befinden soll. Der lokale Framebuffer wird einfach mit Nullen überschrieben.

Bei der Initialisierung der beiden Buffer muss die inhärente Parallelität der GPU berücksichtigt werden, damit nicht versehentlich jedes Feld mehrmals beschrieben wird was zu starken Leistungseinbußen

führt. Jeder Thread beginnt mit dem beschreiben des Buffers an der Stelle seiner eigenen ID und nach jedem Schleifendurchgang wird um die Gesamtzahl der Threads innerhalb eines Blockes erhöht.

```
// clear zBuffer
for (int i = threadID; i < tileSize ; i += blockDim.x*blockDim.y) {
        buffer[i] = 10000.0;
}
// clear localFramebuffer
for (int i = threadID; i < ( tileSize )*3; i += blockDim.x*blockDim.y) {
        localFrameBuffer[i] = 0.0;
}
```

4.5 Kugel Sortierung auf CPU Seite

Die Sortierung der Kugeln erfolgt auf CPU Seite. Dies hat mehrere Gründe. Zum einen steht die Zeit die für das rendern eines Bildes benötigt wird bei dieser Thesis im Vordergrund. Zum anderen ist eine Sortierung auf CPU Seite um einiges einfacher zu implementieren und bietet dennoch eine optimale Sortierung. Über einen optimalen Sortieralgorithmus auf GPU Seite kann man eine eigene Ausarbeitung schreiben weshalb unsere Implementierung sich vorerst mit einer CPU Lösung dieses Problems zufrieden gibt.

Für die Sortierung wird die vector Datenstruktur aus der Standard Template Library (STL) genutzt da diese gegenüber einer Array Implementierung einige Vorteile bietet. Der wichtigste dabei ist dass die Größe des vectors sich dynamisch anpasst und wir problemlos weitere Objekte hinzufügen können. Für jedes Tile wird dabei ein eigener vector erzeugt und in einen umgebenen vector eingefügt.

```
std :: vector< std :: vector<Sphere> > SphereVector;
for (int i = 0; i < device.numberOfTiles; i++) {
        std :: vector<Sphere> temp;
        SphereVector.push_back(temp);
}
```

Wir haben nun einen vector zur Verfügung der aus so vielen vectoren besteht wie wir Tiles in unsere Szene haben. Um die Kugeln nun zu sortieren werden diese nacheinander in die vectoren eingefügt. Und zwar jeweils so, dass jede Kugel in allen vectoren landet in dessen Tile sich die Kugel befindet.

```
for (int i = 0; i < host_spheres_size ; i++) {
        int y_min = (spheres[i].t + 512) / (1024 / device.blocks.y);
        int y_max = (spheres[i].b + 512) / (1024 / device.blocks.y);
        int x_min = (spheres[i].l + 512) / (1024 / device.blocks.x);
        int x_max = (spheres[i].r + 512) / (1024 / device.blocks.x);
        // push sphere to tiles
        for (int m = y_min; m <= y_max; m++) {
                for (int n = x_min; n <= x_max; n++) {
                        if (n + m*device.blocks.x < device.numberOfTiles) {
                                SphereVector[n + m*device.blocks.x].push_back(spheres[i]);
                        }
                }
        }
}
```

Die verschiedenen min und max Variablen stehen dabei für die x bzw. y Koordinaten der Tiles die von einer Kugel berührt werden.

Anschließend wird das Offset Array mit Werten gefüllt damit wir innerhalb des Kernels auch wissen ab welcher Position des Kugel Arrays wir mit dem rendern beginnen müssen.

```
int sum = 0;
offsetArray [0] = 0;
for (int i = 0; i < device.numberOfTiles; i++) {
        sum += SphereVector[i]. size () ;
        offsetArray [i+1] = tempSphereVector[i]. size () + offsetArray [i];
}
```

Da sich eine Kugel in mehreren Tiles befinden kann und wir die Kugeln hinterher wieder in einem Array speichern benötigen wir die sum Variable um die Gesamtgröße des Arrays festlegen zu können.

```
Sphere *atemp = spheres_sorted;
for (unsigned int i = 0; i < tempSphereVector.size () ; i++) {
        std :: copy(tempSphereVector[i].begin () , tempSphereVector[i].end() , atemp);
        atemp += tempSphereVector[i]. size () ;
}
atemp = nullptr ;
```

Um die sortierten Kugeln nun im spheres sorted Array zu speichern wird ein Pointer auf das Array erzeugt. An der Stelle des Pointers werden nun die Kugeln eines vectors kopiert und anschließend der Pointer um die Anzahl Kugeln inkrementiert. Nachdem alle Kugeln kopiert wurden darf nicht vergessen werden den Pointer wieder auf null zu setzen.

4.6 Kugel Boundary Box

In unserem Tile basierten Ansatz läuft die Schleife für die Pixel über alle Pixel eines Tiles. Kugeln die jedoch um einiges kleiner als ein Tile sind oder die sich sehr weit hinten in der Szene befinden überdecken nur einen Bruchteil der Pixel des Tiles. Die Idee ist deshalb die Schleife nur über die Pixel einer Boundary Box laufen zu lassen.

Die Kugel Klasse enthält zu diesem Zweck 4 weitere float Membervariablen die jeweils für den linken, rechten, oberen sowie unteren Pixel der Boundary Box stehen. Für die Berechnung der Boundary Box wird ein eigener Kernel gestartet der bei Bedarf vor dem render Kernel ausgeführt wird. Abbildung 4.2 zeigt die Vorgehensweise bei der Berechnung der Boundary Box. Die einzelnen Längen berechnen sich dabei wie folgt.

$$P = \frac{r^2}{\vec{SC}}, q = |\vec{SC}| - p, h = \sqrt{pg}$$

Die Richtung von h ergibt sich dadurch dass der Vektor $\vec{SC}$ in das von V und R aufgespannte Koordinatensystem projiziert wird. Durch Koordinaten Inversion erhält man nun die zu $\vec{SC}$ orthogonale Richtung. Nun können leicht die Punkte B_1 und B_2 berechnet werden. Die vertikalen Grenzpunkte

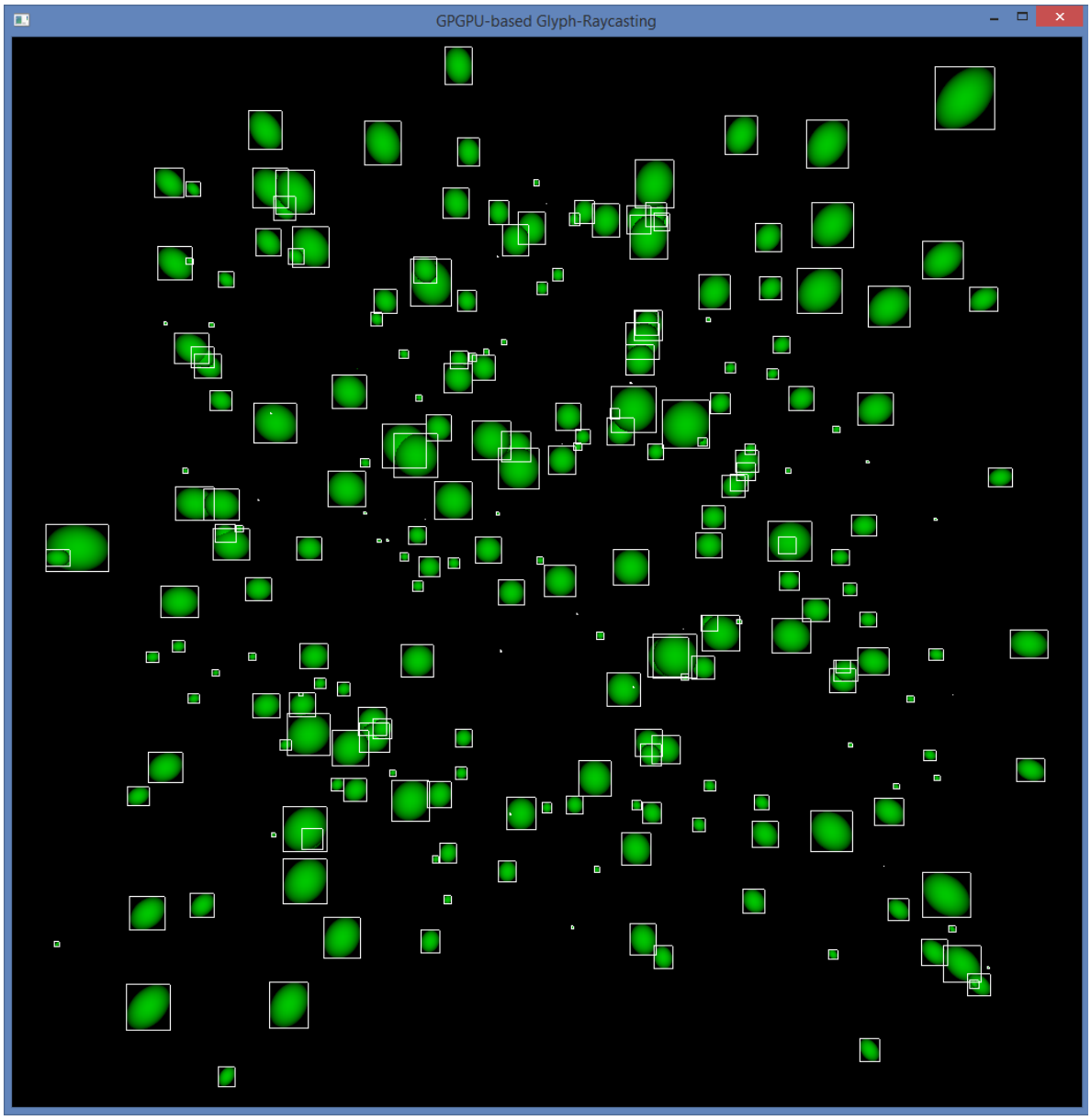

Abbildung 4.1: Eingezeichnete Boundary Boxes im Cuda Raycaster. Die berechnete Boundary Box jeder Kugel ist hier zur Überprüfung visuell sichtbar gemacht. Selbst die von der Perspektive verzerrten Kugeln besitzen eine eng anliegende korrekte Boundary Box

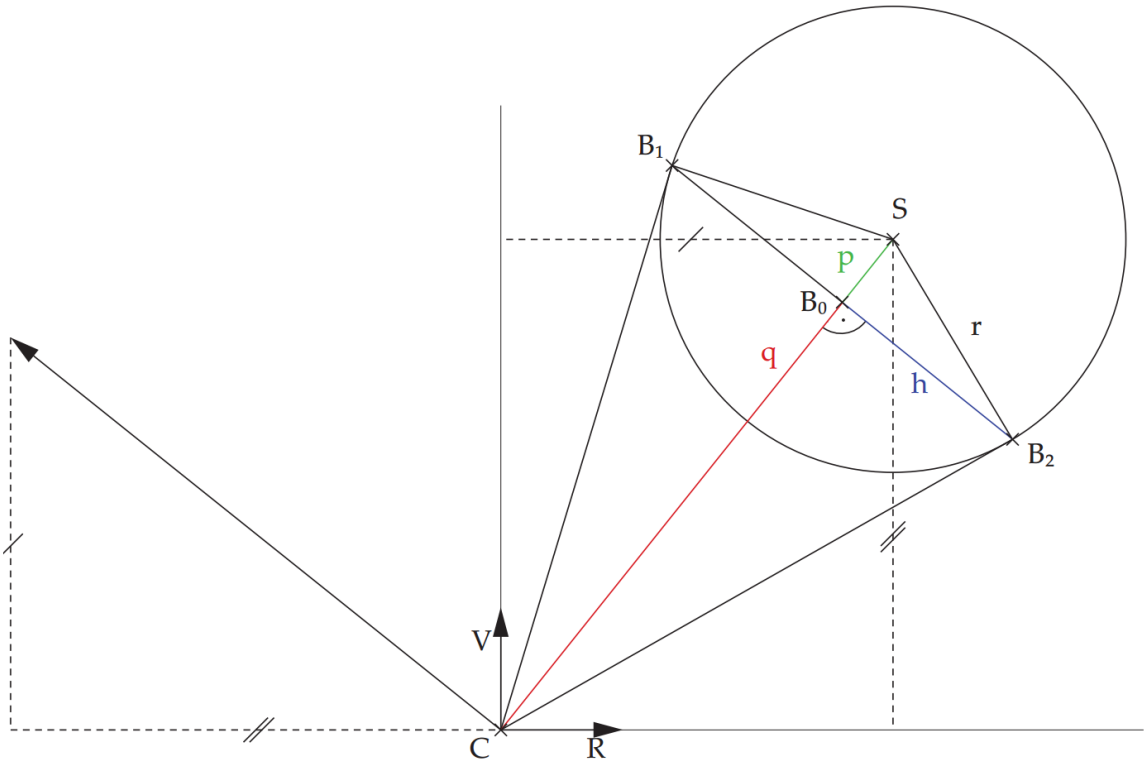

Abbildung 4.2: Berechnung der Boundary Box. Gesucht sind die Punkte B1 sowie B2 welche aus der Sicht der Kamera (C) die äußersten Punkte der Kugel darstellen. Durch die im Text erwähnten Formeln für die Längen und der zuvor ausgerechneten Richtung für $\vec{h}$ ist es problemlos möglich die Punkte B1 und B2 zu berechnen. Für die in der Skizze nicht eingezeichneten vertikalen Punkte B3 und B4 wird dann ähnlich verfahren. [Rei08]

B_3 und B_4 ergeben sich durch das gleiche Vorgehen. Die vier Eckpunkte der Boundary Box liegen nun in Weltkoordinaten vor. Für unsere Sichtstrahlen benötigen wir jedoch die Pixelkoordinaten der Bildebene in denen sich diese vier Eckpunkte befinden. Dazu schneiden wir einfach die von der Kamera zu den Eckpunkten verlaufenden Vektoren mit der Bildebene um die jeweiligen Pixelpositionen zu erhalten. Diese werden dann in den Membervariablen der Kugel gespeichert.

4.7 Vor- und Nachteile des jeweiligen Ansatzes

Hier werden wir näher auf die schon zuvor kurz besprochenen Möglichkeiten der Schleifenanordnungen des Raycasting Algorithmus eingehen.

Noch einmal einen kurzen Überblick über die 4 Varianten welche wir zuvor aufgebaut haben.

- Variante 1: Die Schleife läuft über die Pixel des Bildes und es werden verschiedene Pixel gleichzeitig gerendert.
- Variante 2: Die äußere Schleife läuft über die Objekte der Szene und es werden verschiedene Pixel gleichzeitig gerendert.
- Variante 3: Die äußere Schleife läuft über die Pixel des Bildes und es werden verschiedene Objekte gleichzeitig gerendert.
- Variante 4: Die äußere Schleife läuft über die Objekte der Szene und es werden verschiedene Objekte gleichzeitig gerendert.

Diese Varianten werden noch weiter danach unterschieden ob die Schleifen über alle Pixel laufen oder gar eine Boundary Box für die Objekte genutzt wird und ob ein lokaler Framebuffer zur Zwischenspeicherung genutzt wird. Einigen Varianten steht ein kurzer Beispielcode voran der das Prinzip des jeweiligen Ansatzes veranschaulicht. Um den Code an einigen Stellen kürzer zu halten werden z.B. folgender Codestücke substituiert:

```
xCoord(tile) = info.tileCoordX* tileSizeX
yCoord(tile) = info.tileCoordY* tileSizeY
xCoord(tile+1) = (info.tileCoordX+1)* tileSizeX
yCoord(tile+1) = (info.tileCoordX+1)* tileSizeY
```

Die info Datenstruktur beinhaltet Informationen zu dem jeweiligen Cuda Block. Nähere Informationen zu den dazugehörigen Membervariablen und wie diese berechnet werden sind im Anhang A bei den Datenstrukturen zu finden.

4.7.1 Variante 1 ohne lokalen Framebuffer und ohne vorherige Sortierung der Objekte

```
1   // loop over every pixel of tile
2   for (int world_y = yCoord(tile) + threadIdx.y; world_y < yCoord(tile+1); world_y+=blockDim.y) {
3       for (int world_x = xCoord(tile) + threadIdx.x; world_x < xCoord(tile+1); world_x+=blockDim.x) {
4           Ray ray(eye, Vector3D((world_x)-width/2 +0.5, (world_y)-height/2 + 0.5, -d));
5           ray.d.normalize();
6
7           for (int sphereIndex =0; sphereIndex < numberOfSpheres; sphereIndex ++) {
8
9               if (sh[sphereIndex].l + width*0.5 > xCoord(tile+1))
10                  continue;
11              if (sh[sphereIndex].r + width*0.5 < xCoord(tile))
12                  continue;
13              if (sh[sphereIndex].t + height*0.5 > yCoord(tile+1))
14                  continue;
15              if (sh[sphereIndex].b + height*0.5 < yCoord(tile))
16                  continue;
17
18              // convert world coordinate to local tile coordinate for shared zBuffer
19              int localx = world_x - xCoord(tile);
20              int localy = world_y - yCoord(tile);
21              int localOffset = localx + (localy * (tileSizeX));
22
23              float tmin;
24              float shading;
25              if(hit(ray, sh[sphereIndex], tmin, shading)) {
26                  if (tmin < zBuffer[localOffset]) {
27                  zBuffer[localOffset] = tmin;
28                  // write to framebuffer
29                  int offset = world_x + (world_y * width);
30                  ptr[offset].x = 255 * shading;
31                  ptr[offset].y = 0;
32                  ptr[offset].z = 0;
33                  }
34              }
```

Wir werden an dieser Stelle den Codeausschnitt einmalig ganz durchgehen und bei den Erläuterungen der späteren Ansätze nur noch auf Unterschiede eingehen.

Jedes Tile in unserem Algorithmus entspricht wie schon an vorheriger Stelle erwähnt einem Cuda Block. In den beiden Schleifen in Zeile 2 und 3 laufen wir somit durch alle Pixel eines einzelnen Tiles. Die initialen Pixelpositionen ergeben sich dabei aus der x bzw. y Koordinate des jeweiligen Tiles (dem linken unteren Eck davon) plus der jeweiligen ID des Threads innerhalb des Cuda Blocks. Die Schleifen werden dabei um die Anzahl der jeweiligen zugewiesenen Threads erhöht. Bei einer zugewiesenen Blockgröße von beispielsweise (32,16) Threads erhöht sich die Pixelposition in x-Richtung um 32 und die y-Richtung um 16 Positionen.

In Zeile 4 und 5 erfolgt die Berechnung des Sichtstrahles. Als Ursprung besitzt dieser die Position der virtuellen Kamera. Bei der Berechnung der Richtung ist zu beachten dass die Kamera und die Pixelpositionen getrennte Koordinatensysteme haben. Die Kamera befindet sich in der Mitte ihres Koordinatensystems während bei der Bildebene der Ursprung sich an der linken unteren Ecke befindet. Aus diesem Grund müssen wir die Hälfte der Breite bzw. Höhe des Bildes von der aktuellen Pixelposition abziehen. Da der Strahl auch durch die Mitte eines Pixels projiziert werden soll addieren wir 0.5 Pixel zu den jeweiligen Koordinaten hinzu. Der z-Wert des Richtungsvektors ist der Abstand von der Kamera zur Bildebene und ist durch die Variable d gekennzeichnet. Da die Kamera außerdem in Richtung der negativen z-Achse blickt wird ein Minus davorgestellt. Da der Sichtstrahl später zur Schnittstellenberechnung benötigt wird darf auch nicht vergessen werden diesen zu normieren was in Zeile 5 erledigt wird.

Zeile 7 ist eine weitere Schleife die diesmal über die Objekte der Szene läuft. Da wir die Objekte innerhalb eines Blockes nicht parallel verarbeiten, berechnet hier jeder Thread innerhalb eines Warps den selben sphereIndex.

Zeile 9 - 16 Damit für Objekte die sich überhaupt nicht im Tile befinden Schnitttests ausgeführt werden, überprüfen wir vorher ob das Objekt sich überhaupt innerhalb des Tiles befindet. Dazu wird getestet ob sich ein Teil der Boundary Box der Kugel innerhalb des Tiles befindet. Die Idee dabei ist zu überprüfen ob die zum Tile näher befindliche Kante der Boundary Box außerhalb des Tiles liegt. Damit würde sich dann auch die restliche Boundary Box nicht mehr innerhalb des Tiles befinden können. Falls sich z.B. die linke Kante der Boundary Box rechts neben der rechten Kante des Tiles befindet, dann kann die Kugel gleich übersprungen werden da klar ist dass sich die gesamte Boundary Box außerhalb des Tiles befindet. Mit den restlichen Kanten wird dann ähnlich vorgegangen.

Zeile 19-21 Hier wird die Pixelposition innerhalb der Bildebene auf die Position im shared Z-Buffer umgerechnet. Dies ist notwendig da jeder Cuda Block einen eigenen shared Memory Bereich besitzt in dem der Z-Buffer gespeichert wird. Es kann also nicht ohne weiteres die globale Pixelposition übernommen werden da sich diese nicht auf die einzelnen Tiles beziehen. Der benötigte Offset für den Z-Buffer lässt sich dabei relativ leicht berechnen indem man zu der Pixelposition einfach die x bzw. y Position des Tiles subtrahiert und dann wie gewohnt den Offset bildet.

Zeile 25 Hier wird nun die eigentliche hit Funktion ausgeführt die das Objekt auf einen Schnitttest mit dem Sichtstrahl testet. Der zusätzliche Parameter tmin liefert die Distanz zum nächsten Schnittpunkt zurück, sofern einer vorhanden ist.

Zeile 26 und 27 Hier wird überprüft ob sich der gefundene Schnittpunkt näher zur Bildebene befindet als der gespeicherte Wert im Z-Buffer. Falls dies zutrifft wird der Z-Buffer mit dem neuen Wert aktualisiert.

Zeile 29 bis 32 Zum Schluss wird nun der Pixel im Framebuffer eingefärbt. Der Framebuffer wird dabei durch ein eindimensionales Array dargestellt weshalb es wieder nötig ist unsere Pixelposition in einen entsprechenden Offset umzuwandeln. Gespeichert wird das ganze dann im RGB Format und im unserem Beispiel mit einfachem flat shading welches die Kugel rot einfärbt.

Man kann an dieser Variante schon erkennen dass es eventuell sinnvoll ist die äußeren Schleifen mit der inneren Schleife zu vertauschen. Die vier if Abfragen zum testen ob ein Objekt innerhalb eines Tiles liegt ist nur einmal pro Objekt nötig und sollte deshalb vor den beiden äußeren Schleifen stattfinden. Bevor wir jedoch zu Variante 2 kommen wo dies der Fall ist schauen wir uns an wie der lokale Framebuffer im Code gelöst wurde.

4.7.2 Variante 1 mit lokalem Framebuffer und ohne vorherige Sortierung der Objekte

```
1   // loop over every pixel of tile
2   for (int world_y = yCoord(tile) + threadIdx.y; world_y < yCoord(tile+1); world_y+=blockDim.y) {
3       for (int world_x = xCoord(tile) + threadIdx.x; world_x < xCoord(tile+1); world_x+=blockDim.x) {
4           Ray ray(eye, Vector3D((world_x)-width/2 +0.5, (world_y)-height/2 + 0.5, -d));
5           ray.d.normalize();
6
7           for (int sphereIndex =0; sphereIndex < numberOfSpheres; sphereIndex ++) {
8
9               if (sh[sphereIndex].l + width*0.5 > xCoord(tile+1))
10                  continue;
11              if (sh[sphereIndex].r + width*0.5 < xCoord(tile))
12                  continue;
13              if (sh[sphereIndex].t + height*0.5 > yCoord(tile+1))
14                  continue;
15              if (sh[sphereIndex].b + height*0.5 < yCoord(tile))
16                  continue;
17
18              // convert world coordinate to local tile coordinate for shared zBuffer
19              int localx = world_x - xCoord(tile);
20              int localy = world_y - yCoord(tile);
21              int localOffset = localx + (localy * (tileSizeX));
22
23              float tmin;
24              float shading;
25              if(hit(ray, sh[sphereIndex], tmin, shading)) {
26                  if (tmin < zBuffer[localOffset]) {
27                  zBuffer[localOffset] = tmin;
28                  // write to local framebuffer
29                  localFrameBuffer[localOffset*3 + 0] = 255 * shading;
30                  localFrameBuffer[localOffset*3 + 1] = 0;
31                  localFrameBuffer[localOffset*3 + 2] = 0;
32                  }
33              }
34          }
35      }
36  }
37
38  // copy every pixel information of local frame buffer to global frame buffer
39  for (int row = threadIdx.y; row < tileSizeY; row+=blockDim.y) {
40      for (int column = threadIdx.x; column < tileSizeX; column+=blockDim.x) {
41          int offset = (((info.block % gridDim.x)*(width/gridDim.x)) + column)
42           + ((((info.block/gridDim.x))*(height/gridDim.y)+row)*width);
43          int t = (column + row*(width/gridDim.x)) * 3;
44          ptr[offset].x = localFrameBuffer[t + 0];
45          ptr[offset].y = localFrameBuffer[t + 1];
46          ptr[offset].z = localFrameBuffer[t + 2];
47           offset ++;
48      }
49  }
```

Anstatt nun direkt in den Framebuffer zu schreiben werden in Zeile 29 bis 31 die RGB Farbwerte in den shared Memory Bereich geschrieben.

Das schreiben in den globalen Framebuffer geschieht dann ab Zeile 39 bis 48. Dazu wird der gesamte Inhalt des lokalen Framebuffers Pixel für Pixel in den globalen Framebuffer kopiert. Man kann sich dabei vorstellen dass die beiden äußeren Schleifen über die Pixel des Tiles laufen und dabei der Inhalt dessen kopiert wird. Da das ganze auch möglichst parallel von den zur Verfügung stehenden Threads erledigt werden soll ergeben sich die Initialwerte der Schleifen aus der jeweiligen Thread ID und werden dann dementsprechend hochgezählt. Da der lokale Framebuffer aus einem eindimensionalem Array besteht wir aber zwei äußere Schleifen nutzen benötigen wir wieder eine Umrechnung auf den Offset des lokalen Framebuffers. Dies geschieht in Zeile 42.

Als letztes benötigen wir noch der globale Offset für den globalen Framebuffer damit wir wissen an welche Stelle wir den Inhalt kopieren müssen. In Zeile 41 sieht man die dazugehörige Umrechnung. Was auf den ersten Blick etwas wüst anmutet ist nichts anderes als die gewohnte x + y*width Offset Berechnung.

Die Frage ist nun was uns ein lokaler Framebuffer bringt außer einigen Zeilen mehr Code. Wie im Kapitel der Cuda Möglichkeiten erwähnt ist das schreiben in den shared Memory Bereich um einiges schneller als ein schreiben in den globalen Speicher der GPU. Bei sehr dichten Szenen mit vielen überlagernden Objekten bei denen der Framebuffer relativ oft aktualisiert werden muss sollte deshalb eine Beschleunigung messbar sein.

4.7.3 Variante 1 mit lokalem Framebuffer und vorsortierten Objekten

Bisher sind wir bei jedem Pixel über alle möglichen Objekte gewandert und haben überprüft ob sich das Objekt im Tile befindet welches aktuell gerendert wird. Wenn wir die Objekte jedoch zuvor den entsprechenden Tiles zuordnen die von den Objekten berührt werden, dann würde eine Überprüfung gänzlich wegfallen. Innerhalb eines Tiles müssen dann nur noch ein Bruchteil aller Objekte betrachtet werden.

```
1   for (int world_y = yCoord(tile) + threadIdx.y; world_y < yCoord(tile+1); world_y+=blockDim.y){
2       for (int world_x = xCoord(tile) + threadIdx.x; world_x < xCoord(tile+1); world_x+=blockDim.x){
3           Ray ray(eye, Vector3D((world_x)-width/2 +0.5, (world_y)-height/2 + 0.5, -d));
4           ray.d.normalize();
5
6           for (int sphereIndex = offset[info.block]; sphereIndex < offset[info.block+1]; sphereIndex ++) {
7               // convert world coordinate to local tile coordinate for shared zBuffer
8               int localx = world_x - xCoord(tile);
9               int localy = world_y - yCoord(tile);
10              int localOffset = localx + (localy * (tileSizeX));
11
12              float tmin;
13              float shading;
14              if(hit(ray, sh[sphereIndex], tmin, shading)) {
15                  if (tmin < zBuffer[localOffset]) {
16                  zBuffer[localOffset] = tmin;
17                  // write to local framebuffer
18                  localFrameBuffer[localOffset*3 + 0] = 255 * shading;
19                  localFrameBuffer[localOffset*3 + 1] = 0;
20                  localFrameBuffer[localOffset*3 + 2] = 0;
21                  }
22              }
23          }
24      }
25  }
26  // copy every pixel information of local frame buffer to global frame buffer
27  for (int row = threadIdx.y; row < tileSizeY ; row+=blockDim.y){
28      for (int column = threadIdx.x; column < tileSizeX ; column+=blockDim.x){
29          int offset = (((info.block % gridDim.x)*(width/gridDim.x)) + column)
30           + ((((info.block/gridDim.x))*(height/gridDim.y)+row)*width);
31          int t = (column + row*(width/gridDim.x)) * 3;
32          ptr[offset].x = localFrameBuffer[t + 0];
33          ptr[offset].y = localFrameBuffer[t + 1];
34          ptr[offset].z = localFrameBuffer[t + 2];
35          offset ++;
36      }
37  }
```

In Zeile 7 wird nun anstatt über alle Objekte zu gehen ein offset Array genutzt. Das offset Array ist dabei als Präfix Summe wie folgt definiert.

$$offset[0] = 0$$

$$offset[1] = [\#Objekte\ in\ Tile\ 1]$$

$$offset[n] = offset[n-1] + [\#Objekte\ in\ Tile\ n]$$

Anders ausgedrückt steht in offset[n] also immer der Startoffset für das Objektarray des aktuellen Tiles da alle Objekte vor sh[n] sich in vorherigen Tiles befinden. Die Schleife läuft dann solange bis der Wert erreicht ist der als Startoffset für das nächste Tile gilt.

Durch die Vorsortierung sollte der Kernel nun um einiges schneller laufen. Auch wenn es sich auf den ersten Blick nur um vier if Abfragen handelt die man sich gespart hat. Bei einer Szene mit 32 * 32 Tiles und 100.000 Objekten, von denen wir der einfachheitshalber annehmen dass diese gleichmäßig verteilt sind, muss nur noch ein Bruchteil der Objekte betrachtet werden. Und zwar 1/1024 * 100.000 = 97. Wir sparen uns im Besten Falle also das Ausführen von 400.000 if Abfragen die für 100.000 Kugeln nötig sind.

4.7.4 Variante 2

Bei der zweiten Variante werden nun die Objekt und die Pixel Schleifen miteinander vertauscht. Auch hier kann man wie schon bei Variante 1 wahlweise einen lokalen Framebuffer nutzen oder die Kugeln vorsortieren. Wir gehen nur ganz kurz auf die Variante ohne vorsortiere Kugeln ein, da es nur bei dieser einen Unterschied machen würde.

```
1                       for (int sphereIndex =0; sphereIndex < numberOfSpheres; sphereIndex ++) {
2
3                           if (sh[sphereIndex].l + width*0.5 > xCoord(tile+1))
4                               continue;
5                           if (sh[sphereIndex].r + width*0.5 < xCoord(tile))
6                               continue;
7                           if (sh[sphereIndex].t + height*0.5 > yCoord(tile+1))
8                               continue;
9                           if (sh[sphereIndex].b + height*0.5 < yCoord(tile))
10                              continue;
11
12      // loop over every pixel of tile
13      for (int world_y = yCoord(tile) + threadIdx.y; world_y < yCoord(tile+1); world_y+=blockDim.y) {
14          for (int world_x = xCoord(tile) + threadIdx.x; world_x < xCoord(tile+1); world_x+=blockDim.x) {
15              ...
16          }
17      }
```

Anstatt die vier if Abfragen nun für jeden Pixel auszuführen findet die Überprüfung nun einmalig vor dem durchlaufen der Pixel statt. Diese Variante ist jedoch nur potentiell besser als Variante 1. Nämlich nur wenn ein einzelner Thread mehrere Pixel rendert. Das ist der Fall wenn die Szene aus wenigen großen Tiles besteht. Cuda unterstützt bei compute capability 3.0 nur 1024 gleichzeitige Threads per Block. Sobald also ein Tile mehr Pixel besitzt als zugewiesene Threads per Block ist diese Variante besser geeignet.

4.7.5 Variante 3 ohne lokalen Framebuffer und ohne Vorsortierung der Kugeln

Bei Variante 3 gibt es nun einen grundlegenden Unterschied. Die einzelnen Threads teilen sich nun nicht mehr auf die Pixel auf sondern auf die Objekte der Szene.

```
1   // loop over every pixel of tile
2   for (int world_y = yCoord(tile); world_y < yCoord(tile+1); world_y++) {
3       for (int world_x = xCoord(tile); world_x < xCoord(tile+1); world_x++) {
4
5       for (int sphereIndex = threadID; sphereIndex < numberOfSpheres; sphereIndex += (blockDim.x*blockDim.y)) {
6
7               if (sh[sphereIndex].l + width*0.5 > xCoord(tile+1))
8                   continue;
9               if (sh[sphereIndex].r + width*0.5 < xCoord(tile))
10                  continue;
11              if (sh[sphereIndex].t + height*0.5 > yCoord(tile+1))
12                  continue;
13              if (sh[sphereIndex].b + height*0.5 < yCoord(tile))
14                  continue;
15          ...
16      }
17  }
```

Die Änderungen zu den vorherigen Varianten beschränkt sich nur auf die Nutzung der threadId Variablen. Diese wurden aus den Schleifen welche über die Pixel laufen herausgenommen und als eine Thread ID in die Schleife der Objekte in Zeile 7 eingebaut. Ein einzelner Thread läuft nun also über die gesamten Pixel eines Tiles und testet Pixel für Pixel alle Objekte der Szene.

Ein Problem dass dabei nun entsteht ist dass es vorkommen kann dass verschiedene Threads den selben Pixel zur gleichen Zeit rendern möchten. Dies geschieht sobald mindestens zwei der Threads in einem Schleifendurchgang eine Kugel rendern welche innerhalb des Tiles liegt und die sich von der Kamera aus gesehen überlagern. Dann ist nicht mehr garantiert von welcher Kugel die Farbinformation übernommen wird. Da hier die Kugeln noch unsortiert sind tritt dieser Fall eher selten auf und führt im Bild zu kleineren Artefakten. Um das Bild wieder korrekt zu rendern ist ein Synchronisationsmechanismus vor dem schreiben in den Framebuffer nötig bei dem sich diese Threads untereinander austauschen um zu ermitteln welcher Thread in den Framebuffer schreiben darf.

4.7.6 Variante 3 mit lokalem Framebuffer und mit Vorsortierung der Kugeln

```
1   // loop over every pixel of tile
2   for (int world_y = yCoord(tile); world_y < yCoord(tile+1); world_y++) {
3       for (int world_x = xCoord(tile); world_x < xCoord(tile+1); world_x++) {
4
5           for (int sphereIndex = offset[info.block] + threadID; sphereIndex < offset[info.block+1]; sphereIndex += (blockDim.x*blockDim.y)) {
6
7               // convert world coordinate to local tile coordinate for shared zBuffer
8               int localx = world_x - info.tileCoordX*(tileSizeX);
9               int localy = world_y - info.tileCoordY*(tileSizeY);
10              int localOffset = localx + (localy * (tileSizeX));
11
12              float tmin;
13              float shading;
14
15                  if(hit(ray, sh[sphereIndex], tmin, shading)) {
16                      if (thread has nearest Objekt) {
17                          if (tmin < zBuffer[localOffset]) {
18                          zBuffer[localOffset] = tmin;
19                          // write to framebuffer
20                          localFrameBuffer[localOffset*3 + 0] = ((int)-sh[sphereIndex].z % 255) * shading;
21                          localFrameBuffer[localOffset*3 + 1] = 0;
22                          localFrameBuffer[localOffset*3 + 2] = 255*shading;
23                      }
24                  }
25              }
26          }
```

```
27              }
28      }
29
30      __syncthreads();
31      // copy every pixel information of local frame buffer to global frame buffer
32      for (int row = threadIdx.y; row < tileSizeY ; row+=blockDim.y) {
33              for (int i = threadIdx.x; i < tileSizeX ; i+=blockDim.x) {
34                      int offset = (((info.block % gridDim.x)*(width/gridDim.x))+i) + ((((info.block/gridDim.x))*(height/gridDim.y)+row)*width);
35                      int t = (i + row*(width/gridDim.x)) * 3;
36                      ptr[offset].x = localFrameBuffer[t + 0];
37                      ptr[offset].y = localFrameBuffer[t + 1];
38                      ptr[offset].z = localFrameBuffer[t + 2];
39                      offset++;
40              }
41      }
```

Sind die Kugeln vorsortiert und man prüft nicht welcher Thread das zur Bildebene nächste Objekt besitzt dann äußert sich das nicht mehr nur in kleineren Artefakten. Die ganze Szene wird quasi nicht korrekt gerendert, zumindest kann man nicht mehr erkennen ob überhaupt etwas korrekt dargestellt wird. Kugeln die sich in der Szene hinter anderen Kugeln befinden können dabei plötzlich komplett in den Vordergrund gerendert werden. Das liegt daran dass nun jeder Thread garantiert eine Kugeln rendert welche innerhalb des Tiles liegt. Die Wahrscheinlichkeit dass sich die Kugeln der einzelnen Threads nun überlagern ist so hoch dass man davon ausgehen kann das dies fast immer der Fall sein wird sobald sich mehrere dutzend Kugeln innerhalb eines Tiles befinden.

4.7.7 Variante 4

Wie bei den ersten beiden Varianten auch kann man durch vertauschen der äußeren Schleifen im unsortierten Falle ein klein wenig Leistung gewinnen.

4.7.8 Einsatz der Boundary Box

Bisher haben wir die Boundary Box nur dazu genutzt um festzustellen ob sich eine Kugel innerhalb eines Tiles befindet. Alle bisherigen Varianten laufen dabei über alle Pixel eines Tiles. Um Die Anzahl der zu betrachtenden Pixel zu verringern ist es möglich die Pixelschleifen leicht zu modifizieren damit diese nur Pixel betrachten welche sich innerhalb einer Boundary Box befinden.

```
for (int world_y = b; world_y < t; world_y++) {
        for (int world_x = l; world_x < r; world_x++) {
                ...
        }
}
```

Die dazu benötigten Variablen werden wie folgt berechnet

```
int l = max(info.tileCoordX* tileSizeX , (int)(sh[sphereIndex].l +0.5* width));
int r = min((info.tileCoordX+1)* tileSizeX , (int)(sh[sphereIndex].r +0.5* width));
int b = max(info.tileCoordY* tileSizeY , (int)(sh[sphereIndex].t +0.5* height));
int t = min((info.tileCoordY+1)* tileSizeY , (int)(sh[sphereIndex].b +0.5* height));
```

Dabei muss jedoch bedacht werden dass die einzelnen Threads auf Warp Basis ausgeführt werden. Da die Breite einer Boundary Box meistens nicht genau einem vielfachen von 32 Pixeln entspricht bededeutet dies, dass meistens einige Threads innerhalb des Warps nicht aktiv sind. Die mögliche Gesamtauslastung der GPU sinkt damit also. Jedoch sollte dieser Ansatz in einigen Fällen eine

Beschleunigung bringen da die verringerte Anzahl zu betrachtender Pixel diesen Nachteil aufwiegen kann.

4.7.9 Variante 5

Obwohl wir eingangs nur vier Varianten unterschieden haben betrachten wir hier noch eine weitere Variante. Im Prinzip handelt es sich dabei um eine Abwandlung von Variante 1. Da diese sich vom Code jedoch deutlich von den bisherigen Varianten unterscheidet betrachten wir diese getrennt von den anderen Varianten.

Was sich in diesem Ansatz ändert ist der Wegfall der Pixelschleifen. Stattdessen wird jedem Thread ein Pixel innerhalb der Boundary Box zugewiesen und die Threads wandern dann Pixelweise innerhalb der Boundary Box weiter.

```
1  for (int sphereIndex = offset[info.block]; sphereIndex < offset[info.block+1]; sphereIndex++) {
2
3          int l = max(info.tileCoordX* tileSizeX, (int)(sh[sphereIndex].l+0.5*width));
4          int r = min((info.tileCoordX+1)* tileSizeX, (int)(sh[sphereIndex].r+0.5*width));
5          int b = max(info.tileCoordY* tileSizeY, (int)(sh[sphereIndex].t+0.5*height));
6          int t = min((info.tileCoordY+1)* tileSizeY, (int)(sh[sphereIndex].b+0.5*height));
7
8          int boxSize = ((r-(l))*(t-(b)));
9          int boxOffset = threadID; // offset of boundaryBox
10         int x, y;
11
12         while(boxOffset < boxSize) {
13                 // local tile coordinates which lie in boundary box
14                 int lx = (boxOffset % (r-l));
15                 int ly = boxOffset / (r-l);
16                 // world coord
17                 y = b + ly;
18                 x = (lx + l);
19
20                 Ray ray(camera->eye, Vector3D(((float)x)-width/2+0.5, ((float)y)-height/2+0.5, -camera->d));
21                 ray.d.normalize();
22
23                 float tmin;
24                 float shading;
25
26                 // convert world coordinates to local tile coordinates
27                 int localx = x - info.tileCoordX*(tileSizeX);
28                 int localy = y - info.tileCoordY*(tileSizeY);
29                 int localOffset = localx + (localy * (tileSizeX));
30
31                 if(hit(ray, sh[sphereIndex], tmin, shading)) {
32                         if (tmin < zBuffer[localOffset]) {
33                                 zBuffer[localOffset] = tmin;
34                                 // write to framebuffer
35                                 localFrameBuffer[localOffset*3 + 0] = 255 * shading;
36                                 localFrameBuffer[localOffset*3 + 1] = 0;
37                                 localFrameBuffer[localOffset*3 + 2] = 0;
38                         }
39                 }
40                 boxOffset += blockDim.x;
41         }
42 }
43
44 __syncthreads();
45 // copy every pixel information of local frame buffer to global frame buffer
46 for (int row = threadIdx.y; row < tileSizeY; row+=blockDim.y){
47         for (int i = threadIdx.x; i < tileSizeX; i+=blockDim.x) {
48                 int offset = (((info.block % gridDim.x)*(tileSizeX))+i) + ((((info.block/gridDim.x))*(tileSizeY)+row)*width);
49                 int t = (i + row*(tileSizeX)) * 3;
50                 ptr[offset].x = localFrameBuffer[t + 0];
51                 ptr[offset].y = localFrameBuffer[t + 1];
52                 ptr[offset].z = localFrameBuffer[t + 2];
53                 offset++;
54         }
55 }
```

Da die Boundary Box einer Kugel sich teilweise ausserhalb eines Tiles befinden kann muss die Boundary Box eventuell vorher abgeschnitten werden. Dies geschieht in den Zeilen 3-6 in dem einfach

für jede Seite überprüft wird ob sich die Kante der Boundary Box außerhalb der Tile Kante befindet. In diesem Fall wird dann die Koordinate der Tilekante übernommen.

In Zeile 8 und 9 werden dann zum einen die Größe der Boundary Box in Pixel berechnet und zum anderen ein boxOffset mit der jeweiligen thread id initialisert. Dieser boxOffset wird später jeweils um die Gesamtzahl der Threads inkrementiert.

Zeile 12 Da nur Pixel gerendert werden müssen die sich innerhalb der Box befinden wird in dieser Schleife überprüft ob sich der aktuelle offset schon ausserhalb der Boundary Box befindet.

In den restlichen Zeilen findet die Berechnung von lokalen sowie globalen Koordinaten statt welche für den lokalen Framebuffer sowie für den Sichtstrahl benötigt werden.

In Zeile 40 wird dann nach einem durchlauf der Schleife der boxOffset inkrementiert um den nächsten Pixel innerhalb der Boundary Box zu rendern.

Im Unterschied zu den anderen Varianten mit einer Boundary Box wird hier jedem Thread ein Pixel zugewiesen, selbst wenn die Breite der Boundary Box kein vielfachem der Warpgröße entspricht. Nur im letzten Schleifendurchlauf werden womöglich nicht mehr alle Threads benötigt um die restlichen Pixel der Boundary Box zu rendern. Dieser Ansatz sollte gerade bei Szenen mit größeren Kugeln einen leichten Vorteil besitzen.

4.8 Erreichen hoher GPU Auslastung durch geeignete Grid und Block Größen

Einen sehr großen Einfluss auf die Performance haben die Anzahl der Blöcke und Threads die dem Programm zugewiesen werden. Zum einen sollte die Blockgröße immer ein vielfaches von 32 sein damit jeder Warp mit 32 aktiven Threads voll ausgefüllt werden kann. Und zum anderen darf die Grid Größe nicht zu klein gewählt werden.

Einzelne Blöcke werden jeweils einer SM zugeteilt die sich dann um die Ausführung und die Koordination kümmert. Meistens benötigen verschiedene Blöcke jedoch nicht gleich viel Rechenzeit. Während manche Blöcke innerhalb einer Millisekunde abgearbeitet werden kann es sein dass andere ein vielfaches davon benötigten. Um deshalb nicht Gefahr zu laufen dass ein Großteil der GPU Ressourcen im Leerlauf ist sollte man die Gridgröße im Zweifel lieber zu groß anstatt zu klein wählen damit die einzelnen Blöcke schneller abgearbeitet werden können und es genügend Blöcke gibt um die GPU voll auszulasten.

4.9 Limitierungen durch Register Anzahl, sharedMemory

Man möchte also erreichen dass möglichst alle SMs voll ausgelastet sind. Jeder SM besitzt selbst jedoch nur begrenzte Ressourcen was die Anzahl an Registern oder dem verfügbaren shared Memory angeht. Dieser Umstand muss bei der Auswahl geeigneter Grid und Block Größen beachtet werden.

Im Tile basiertem Raycaster nutzen wir für einen lokalen Framebuffer den shared Memory. Die Größe des benötigten Speichers hängt dabei von der Größe des jeweiligen Tiles ab, da für jeden Pixel 4 Byte im lokalen Framebuffer benötigt werden. Die Nvidia GTX 660 besitzt 49.152 Bytes an shared Memory pro SM. Ein SM kann dabei bis zu 8 Blöcke aufnehmen. Bei einem voll ausgelastetem SM stehen jedem Block also 6.144 Bytes zur Verfügung. Das langt für eine Tile Größe von 64 * 64 = 4096. Nicht jedoch für ein Tile der Größe 80 * 80 = 6400. Das bedeutet bei einer Bildebene von 1024 * 1024 Pixel, dass die Größe unseres Grids mindestens aus 16 * 16 Blöcken bestehen muss damit die SM voll ausgelastet werden kann. Eine kleinere Gridgröße bedeutet größere Tiles und somit einen höheren Bedarf an shared Memory der dann nicht mehr für alle 8 Blöcke auf der SM ausreicht.

Ähnlich sieht es auch mit der Anzahl der Threads innerhalb eines Blockes aus. Jeder Thread benötigt seinen eigenen Satz an Registern. Je mehr Threads sich in einem Block befinden desto mehr Register werden für diesen Block dann benötigt. Der Nvidia GTX 660 stehen hier pro SM stolze 65536 Register zur Verfügung. Diese sind jedoch schneller aufgebraucht als man denkt. Auch hier teilen sich die Blöcke auf einer SM die Register, was sich dann zu 8192 Registern pro Block verringert. Stellen wir die Gridgröße so ein dass ein Tile eine Größe von 32 * 32 = 1024 Pixel umfasst. Was eine realistische Einstellung ist da dann ein einzelner Warp eine gesamte Pixelreihe gleichzeitig rendern kann. Wenn nun jeder Pixel von einem eigenen Thread gerendert werden soll, also einem Block 1024 Threads zugewiesen werden, dann stehen für einen einzelnen Thread nur noch 8 Register zur Verfügung. Das ist eine unrealistisch kleine Anzahl an Registern für einen Kernel der tatsächlich auch etwas berechnen soll. Da die meisten Kernel mehr Register benötigen als sie hier dann zur Verfügung haben wird das durch die SM so geregelt dass dementsprechend weniger Blöcke auf einer SM Platz finden und die GPU dementsprechend weniger ausgelastet ist was zu Leistungseinbußen führt.

Eine optimale Einstellung der Grid und Block Größen ist nicht ganz einfach zu finden und erfordert ein wenig herumprobieren.

4.10 Vermeidung von thread divergenz

Die einzelnen Blöcke werden nacheinander einem SM zugewiesen und dort ausgeführt. Eine SM kann dabei je nach Ressourcenverfügbarkeit auch mehrere Blöcke gleichzeitig ausführen. Die Instruktionen eines auf der GPU ausgeführten Kernels werden dann auf warp Basis nacheinander ausgeführt. Das bedeutet dass für jeweils 32 Threads die selbe Instruktion ausgeführt wird. Für eine parallele Berechnung ist es jedoch notwendig dass die einzelnen Threads irgendwann auf verschiedenen Daten operieren. Sind im Kernelcode dann Verzweigungen implementiert die von diesen Daten abhängig sind ist es sehr wahrscheinlich dass nicht alle Threads den selben Instruktionspfad folgen. Da im Cuda Modell jedoch nicht vorgesehen ist dass zwei verschiedene Threads innerhalb eines Warps zwei verschiedenen Instruktionen ausführen werden jeweils alle Threads inaktiv gesetzt welche den aktuellen Instruktionspfad nicht gefolgt sind. Die Verzweigungen im Code werden dann quasi seriell ausgeführt. Man kann sich leicht ein Szenario vorstellen indem die einzelnen Threads eines Warps abwechselnd eine bestimmte Bedingung erfüllen und dementsprechend einem Pfad folgen. Ordnet man nun jedoch die Daten so um dass einzelne Threads nicht mehr abwechselnd die Bedingung erfüllen sondern auf Warp Basis, dann halbiert sich die Ausführungsdauer des Codeabschnittes.

5 Performance

Um die verschiedenen Ansätze miteinander zu vergleichen wird erst einmal untersucht wie jeder Ansatz mit dem Rendern von 1000 Kugeln performt. Dazu wird jeder Ansatz mit den Gridgrößen 32x32, 48x48, 64x64, 80x80, 96x96, 128x128, 160x160, 192x192, 224x224, sowie 256x256 getestet. Zu jeder Gridgröße werden dann alle möglichen Blockgrößen zwischen 16x1 bis 32x32 automatisiert durchlaufen. Die am schnellsten laufende Variante jedes Ansatzes wird daraufhin mit verschiedener Anzahl an Kugeln und Kugelgrößen getestet.

Die Varianten bei denen die äußere Schleife über die Kugeln läuft ist es möglich die Pixelschleifen über die Boundary Box der aktuellen Kugel laufen zu lassen. Diese Varianten besitzen deshalb eine weitere Tabellenzeile mit den dazugehörigen Werten.

Zum Schluss vergleichen wir die schnellsten Varianten noch einmal untereinander um bestimmen zu können welcher Ansatz nun das meiste Potential für zukünftige Erweiterungen und Verbesserungen bieten würde.

Um die Renderzeit für einen Kerneldurchlauf zu ermitteln wird der jeweilige Kernel zehn mal ausgeführt und davon das statistische Mittel ausgewählt. Desweiteren werden bei Variante 3 sowie Variante 4 während den Testdurchläufen der normalerweise nötige Datenaustausch zwischen den Threads weggelassen um die verschiedene Ansätze hinterher besser vergleichen zu können. Dieser findet bisher nur rudimentär statt und würde die Ergebnisse deshalb zu stark verfälschen.

5.1 Renderperformance beim rendern von 1000 Kugeln

5.1.1 Variante 1: äußere Schleife über Pixel, Parallel über Pixel

Variante 1	Gridgröße	Blockgröße	Renderzeit
vanilla	32x32	32x32	237,659ms
+ lokaler Framebuffer	64x64	16x16	232,604ms
+ sortierte Kugeln	64x64	16x8	3,896ms

Tabelle 5.1: Renderzeiten Variante 1 für 1000 Kugeln

Was man bei Variante 1 direkt feststellen kann ist das die Nutzung eines lokalen Framebuffers die Renderzeit um einige ms beschleunigt. Nicht überraschend ist die Tatsache dass es bei vorheriger Sortierung der Kugeln einen erheblichen Geschwindigkeitszuwachs gibt. Statt 232ms bei einer unsortierten Variante werden nur noch 3,9ms pro Kernelaufruf benötigt.

Äuffälig ist noch dass bei der Hinzunahme des lokalen Framebuffers sich die optimale Gridgröße von 32x32 auf 64x64 vergrößert und die Blockgröße von 32x32 auf 16x16 erniedrigt hat. Die Nutzung eines lokalen Framebuffers erfordert also mehr Blöcke mit weniger Threads pro Block um optimal laufen zu können.

5.1.2 Variante 2: äußere Schleife über Kugeln, Parallel über Pixel

Variante 2	Gridgröße	Blockgröße	Renderzeit
vanilla	32x32	16x8	50,117ms
+ lokaler Framebuffer	32x32	32x4	52,091ms
+ sortierte Kugeln	64x64	16x8	2,329ms
+ boundary box	64x64	16x8	1,541ms

Tabelle 5.2: Renderzeiten Variante 2 für 1000 Kugeln

Variante 2 besitzt bei allen 3 Versionen relativ ähnliche Grid- und Blockgrößen. Die Vorsortierung der Kugeln ist auch hier für einen enormen Geschwindigkeitszuwachs von 52ms auf nur noch 2,3ms verantwortlich.

Dass Vertauschen der Schleifen hat in den unsortierten Fällen einen erheblichen Einfluss auf die Laufzeit. Waren es bei Variante 1 noch 237,659ms werden bei Variante 2 nur noch 50,117ms benötigt. Dies liegt hauptsächlich daran dass in den unsortierten Fällen vorher überprüft werden muss ob die Kugel innerhalb des Tiles liegt. Diese Überprüfung findet direkt nach der Schleife über die Kugeln statt. Da bei Variante 2 sich diese Schleife außen befindet wird die Überprüfung auch tatsächlich nur noch einmal pro Kugel durchlaufen.

Auffällig ist jedoch dass hier die Nutzung eines lokalen Framebuffers die Renderzeit von 50ms auf 52ms minimal verschlechtert.

Der Einsatz einer Boundary Box verbessert die Laufzeit nochmal merklich. Die zuvor benötigten 2,329ms für das Rendern der Szene wurden auf 1,541ms verringert.

5.1.3 Variante 3: äußere Schleife über Pixel, Parallel über Kugeln

Variante 3	Gridgröße	Blockgröße	Renderzeit
vanilla	96x96	32x4	512,578ms
+ lokaler Framebuffer	96x96	32x4	540,523ms
+ sortierte Kugeln	80x80	32x2	225,536ms

Tabelle 5.3: Renderzeiten Variante 3 für 1000 Kugeln

Bei Ansatz 3 steigen die optimalen Grid- und Blockgrößen erheblich an. Die Zeit um eine Szene zu rendern benötigt im unsortierten Falle knapp 500ms. Werden die Kugeln wieder vorsortiert sinkt diese zwar auf knapp 225ms, ist jedoch noch immer unerwartet hoch. Der Geschwindigkeitszuwachs

durch die Vorsortierung fällt hier also nicht ganz so stark aus wie bei den vorherigen Varianten. Eine Verschlechterung durch die Nutzung des lokalen Framebuffers ist auch hier deutlich zu erkennen. Die Renderzeit steigt dadurch von 512ms auf 540ms an.

5.1.4 Variante 4: äußere Schleife über Kugeln, Parallel über Kugeln

Variante 4	Gridgröße	Blockgröße	Renderzeit
vanilla	160x160	16x16	22,336ms
+ lokaler Framebuffer	192x192	32x4	32.304ms
+ sortierte Kugeln	256x256	32x4	20,581ms
+ boundary box	64x64	16x4	4,555ms

Tabelle 5.4: Renderzeiten Variante 4 für 1000 Kugeln

Die optimalen Grid- und Blockgrößen sind bei Ansatz 4 enorm gestiegen. Im vorsortierten Fall benötigt der Kernel nun eine Gridgröße von 256x256 Blöcken. Das vertauschen der Schleifen macht sich hier mehr bemerkbar als der Unterschied zwischen Variante 1 und 2. Der lokale Framebuffer hat auch hier wieder einen negativen Einfluss auf die Performance und erhöht die Renderzeit im unsortierten Fall von 22ms auf 32ms.

Die Hinzunahme einer Boundary Box verbessert hier das Ergebnis sogar erheblich. Statt 20,581ms werden nur noch 4,555ms benötigt.

5.1.5 Variante 5: äußere Schleife über Kugeln, Parallel über Pixel der Boundary Box

Variante 4	Gridgröße	Blockgröße	Renderzeit
+ sortierte Kugeln	64x64	16x4	1,568ms

Tabelle 5.5: Renderzeiten Variante 5 für 1000 Kugeln

Diese Variante benötigt wieder verhältnismäßig kleine Gird- und Blockgrößeren um optimal zu laufen. Die Gridgröße mit 64x64 Blöcken ist wie bei Variante 1 und 2 die optimale Wahl. Die Blockgröße ist mit 16x4 die bisher kleine Blockgröße. Dies läßt sich damit erklären das jetzt über die Pixel der Boundary Box gerendert wird und deshalb um einiges weniger Pixel pro Schleifendurchlauf betrachtet werden müssen. Bei kleinen Kugeln mit einem Radius von 3 Pixeln umfasst die Boundary Box einer Kugel meist weniger Pixel als es Threads in einem Warp gibt.

5.2 Vergleich der verschiedenen Ansätze

Vergleicht man alle Ansätze untereinander sieht man dass der lokale Framebuffer nur bei Ansatz 1 einen geringen Vorteil bringt. In allen anderen Ansätzen wird die Performance durch den lokalen Framebuffer negativ beeinflusst.

In den Ansätzen in denen Parallel über die Kugeln gerendert wird stellt man ausserdem fest dass diese um einiges höhere Gridgrößen benötigen als die Ansätze in denen parallel über Pixel gerendert wird.

Variante 2 und 5 sind in allen Gridgrößen ähnlich schnell und besitzen in der kleinsten gewählten Gridgröße schon ihre beste Laufzeit. Diese verschlechtert sich nach jeder Erhöhung der Gridgröße um einige wenige ms. Der Grund dafür dürfte sein dass die einzelnen Blöcke schon eine so geringe Laufzeit besitzen dass der Overhead der zusätzlichen Blöcke die Gesamtzeit eher wieder verschlechtert.

Die Varianten 2, 5 sowie 5 laufen alle über eine Boundary Box und bieten die besten Laufzeiten. Bei sehr kleinen Objekten in denen die Boundary Box weniger als 16 Pixel umfasst sind jedoch nicht mehr alle Threads innerhalb eines Halb-Warps aktiv.

	Variante	1	2	3	4	5
Gridgröße						
64x64		7,095ms	2,065ms	288,734ms	5,907ms	2,156ms
96x96		10,833ms	3,316ms	239,404ms	7,044ms	3,334ms
128x128		12,830ms	4,209ms	227,668ms	7,541ms	4,027ms
160x160		18,290ms	5,631ms	229,857ms	9,398ms	5,552ms
192x192		25,062ms	7,427ms	229,092ms	10,813ms	7,152ms
224x224		20,850ms	8,558ms	241,397ms	11,657ms	8,449ms
256x256		26,722ms	10,061ms	260,269ms	13,449ms	10,052ms

Tabelle 5.6: Vergleich der Renderzeiten jedes Ansatzes mit verschiedener Gridgröße und einer Blockgröße von 32x4

In Tabelle 5.6 ist auch der Einfluss der Gridgröße auf die Performance ersichtlich. Die Schwankungen zwischen den Gridgrößen beträgt bei Ansatz 1 zwischen 7,095ms und 26,722ms pro Kernelaufruf.

Bisher wurden Kugeln mit einem Radius von 3 Pixeln gerendert. Tabelle 5.7 zeigt die Renderzeit die für unterschiedlich viele Kugeln mit einem Radius von 3 Pixeln benötigt werden. Variante 5 rendert bei wenigen Kugeln die Szene noch am schnellsten, wir dann jedoch nach erhöhen der Kugelanzahl von Variante 4 abgehängt.

Tabellen 5.7 und 5.8 zeigen den Unterschied zwischen verschiedenen Kugelradien. Der vergrößerte Kugelradius macht sich dabei am stärksten bei Ansatz 4 und 5 bemerkbar. Während sich beim rendern von 1.000.000 Kugeln die Renderzeiten der restlichen Ansätze im Vergleich zum kleineren Radius knapp verdoppelt haben, hat sich die Laufzeit bei Variante 5 verdreifacht und bei Variante 4 sogar mehr als verfünffacht.

	Variante	1	2	3	4	5
Kugeln						
1000		3.896ms	1,541ms	225,536ms	4,555ms	1,508ms
10000		10,430ms	7,818ms	300,817ms	18,716ms	7,457ms
100000		73,360ms	70,513ms	419,924ms	55,425ms	62,996ms
1000000		669,967ms	637,006ms	1310,025ms	274,587ms	613.028ms

Tabelle 5.7: Vergleich der Renderzeiten jedes Ansatzes mit verschiedener Kugelanzahl und einem Kugelradius von 3 Pixel

Man sieht dass der Kugelradius auch eine große Rolle spielt. Bei einer dichten Szene mit 1.000.000 Kugeln und einem Kugelradius von 3 Pixel besitzt Variante 4 mit einer Renderzeit von 274,587ms einen deutlichen Vorsprung zur zweitbesten Zeit mit 637,006ms besitzt. Vergrößert man den Kugelradius jedoch auf 12 Pixel rendert Variante 1 nun mit 1410,737ms am schnellsten. Die zweitbeste Zeit liegt hier dann mit Variante 4 bei 1866,033ms

	Variante	1	2	3	4	5
Kugeln						
1000		4,719ms	3,690ms	259,944ms	51,245ms	2.912ms
10000		19,135ms	28,863ms	420,586ms	198,952ms	18.869ms
100000		156,724ms	252,416ms	687,095ms	419,439ms	177.387ms
1000000		1410,737ms	2497,756ms	2774,516ms	1866,033ms	1752.234ms

Tabelle 5.8: Vergleich der Renderzeiten jedes Ansatzes mit verschiedener Kugelanzahl und einem Kugelradius von 12 Pixel

5.2.1 Optimierung

Nvidia bietet mit Nsight ein Tool zum Profilen von Cuda Applikationen welches ermöglicht zu untersuchen an welchen Codestellen es Leistungseinbrüche gibt. Dabei ist es möglich sich den vom Compiler erzeugten PTX Code, welcher auf der GPU ausgeführt wird, anzeigen zu lassen und wie viele Instruktionen eine Cuda C Codezeile im PTX Code erzeugt. Variante 4 bietet die bisher beste Laufzeit bei dichten Szenen mit kleinen Kugeln. Deshalb untersuchen wir zuerst Variante 4 und schauen auf welche Stelle im Code die meisten Instruktionen fallen um dann zu überlegen wie man diese betreffende Stelle optimieren kann. Die daraus resultierenden Optimierungen werden dann wenn möglich auch in den restlichen Varianten umgesetzt.

Nach dem starten von Nsight und einer Analyse der Daten ist zu erkennen dass die Initialisierung des Sichtstrahles am meisten Instruktionen benötigt.

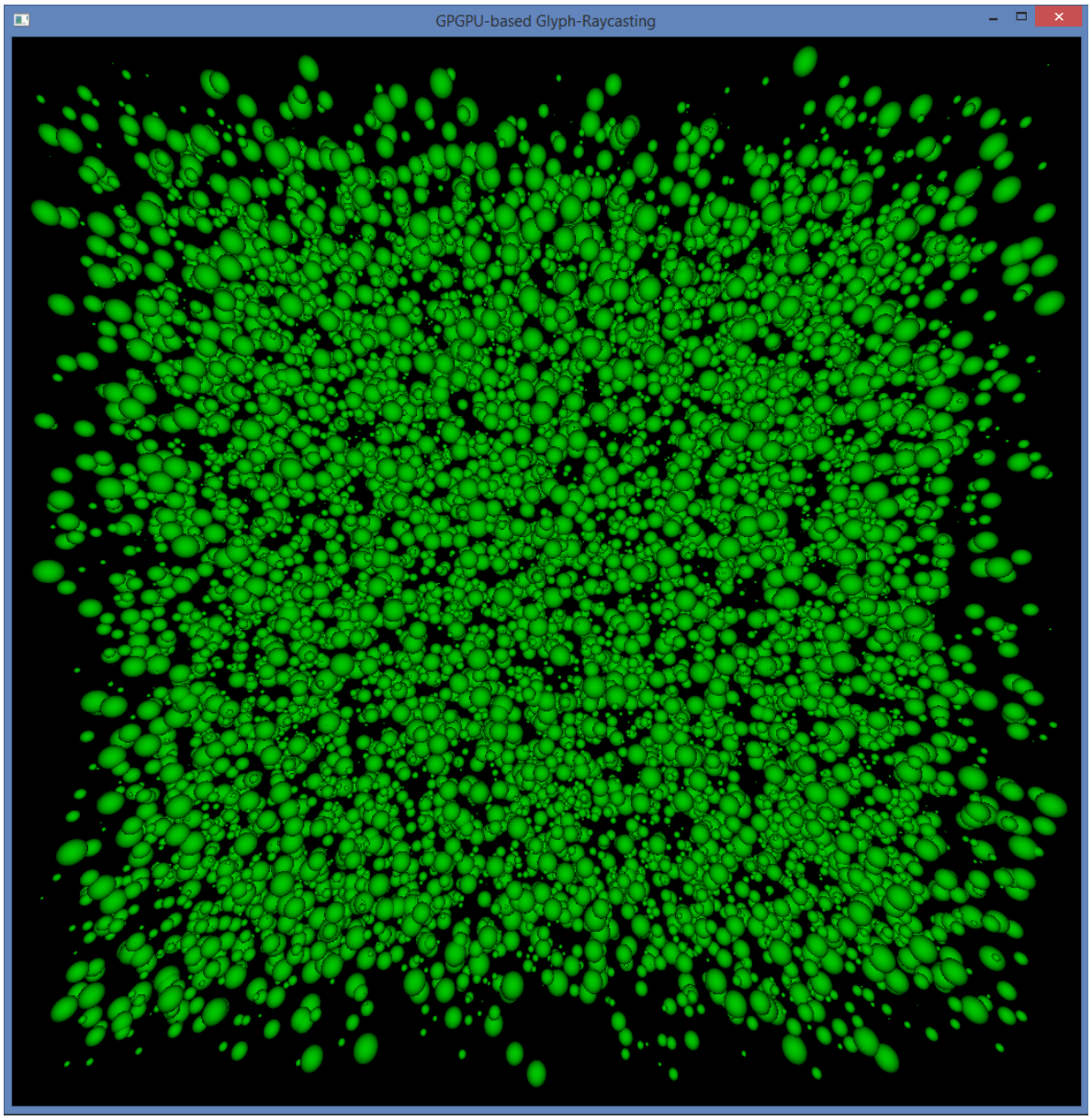

Abbildung 5.1: Eine vom Cuda Raycaster gerendete Szene mit 10.000 Kugeln und einem Kugelradius von 12 Pixel.

```
for (int sphereIndex = threadID + offset[info.block]; sphereIndex < offset[info.block+1];
     sphereIndex+=(blockDim.x*blockDim.y)) {
        ...
        // loop over every pixel of sphere boundary box
        for (int world_y = b; world_y < t; world_y++) {
                for (int world_x = l; world_x < r; world_x++) {
                        Ray ray(camera->eye, Vector3D(((float)world_x)-width/2 +
                                0.5,((float)world_y)-height/2 + 0.5, -camera->d));
                        ray.d.normalize();
                        ...
                }
        }
}
```

Listing 5.1: Variante 4 vor Optimierung der Sichtstrahlerzeugung

Das Problem liegt darin dass für jeden Pixel ein neues Sichtstrahl Objekt erzeugt und initialisiert wird. Um die Anzahl der Instruktionen zu verringern wird die Initialisierung des Sichtstrahles vor die äußere Schleife gelegt. Innerhalb der Pixelschleife werden dann nur noch die x- bzw. y Koordinate der Sichtstahlrichtung aktualisiert. Dazu wird nach jedem Schleifendurchlauf in x-Richtung die x-Koordinate um eins inkrementiert. In y-Richtung dann dementsprechend die y-Koordinate. Jedesmal wenn eine komplette Zeile in x-Richtung gerendert wurde muss die x-Koordinate wieder zurückgesetzt werden. Um auch hier nicht immer wieder diesen Reset-wert neu zu berechnen wird dieser vor den Pixelschleifen einmalig vorher berechnet.

Da die x- und y-Koordinaten nun jeweils inkrementiert werden ist es nicht mehr so einfach möglich den Sichtstrahl innerhalb der innersten Schleife zu normalisieren. Die Normalisierung des Sichtstrahles wurde deshalb in die hit Funktion ausgelagert.

```
Ray ray(camera->eye, Vector3D(0,0, -camera->d));
for (int sphereIndex = threadID + offset[info.block]; sphereIndex < offset[info.block+1];
     sphereIndex+=(blockDim.x*blockDim.y)) {
        ...
        float initialRayCoordX = ((float)l)-width/2 + 0.5;
        ray.d.coord.y = b-height/2 + 0.5;

        // loop over every pixel of sphere boundary box
        for (int world_y = b; world_y < t; world_y++) {
                ray.d.coord.x = initialRayCoordX;
                for (int world_x = l; world_x < r; world_x++) {
                        ...
                        ray.d.coord.x ++;
                }
                ray.d.coord.y ++;
        }
}
```

Listing 5.2: Variante 4 nach Optimierung der Sichtstrahlerzeugung

Des weiteren wurden im Kernelcode und in der hit Funktion einige Zeilen vertauscht um read-after-write Abhängigkeiten von Register Variablen zu entschärfen. Tabelle 5.9 zeigt die nun besten Kernel Laufzeiten nach diesen Optimierungen.

	Variante	1	2	3	4	5
Kugeln						
10000		10,983ms	11,241ms	104,215ms	8,872ms	6,154ms
100000		62,374ms	74,154ms	245,281ms	22,440ms	46,324ms
1000000		574,318ms	643,591ms	469,577ms	131,887ms	440,376ms

Tabelle 5.9: Vergleich der Renderzeiten nach der Optimierung mit verschiedener Kugelanzahl und einem Kugelradius von 3 Pixel

Mit 131,887ms bietet Variante 4 bei einer Kugelanzahl von 1.000.000 die beste Laufzeit. Diese Variante läuft in der äußeren Schleife über die Kugeln und rendert diese auch parallel. Die restlichen Varianten benötigen für das rendern einer Szene weit über 400ms.

5.3 Vergleich mit simpler OpenGL Raycasting Implementierung mit Shadern

Zuletzt schauen wir uns noch einen Vergleich zwischen dem Cuda Raycaster und einer OpenGL Implementierung des Raycasting Algorithmus an. In der OpenGL Version wurde der Algorithmus komplett im Fragment Shader realisiert. Es gibt dabei jedoch keine Partitionierung der Daten oder der Bildebene. Aus diesem Grund sind die Kugeln nicht vorher sortiert und jeder Sichtstrahl wird mit allen Objekten auf einen Schnittpunkt getestet. Zum Vergleich werden noch die Renderzeiten von der optimierten Variante 4 gegenübergestellt, und zwar einmal mit und einmal ohne vorherige Sortierung der Kugeln.

Kugeln	OpenGL Shader Laufzeit	Variante 4 ohne Sortierung	Variante 4 mit Sortierung
1000	61ms	4,806ms	1,904ms
10000	615ms	25,915ms	8,872ms
100000	6250ms	191,437ms	22,440ms
1000000	-	-	131,887ms

Tabelle 5.10: Vergleich der Laufzeiten zwischen OpenGL und Cuda Implementation

Die Laufzeit der OpenGL Implementation steigt linear zur Anzahl der Kugeln an. Während bei einer Szene mit 1000 Kugeln 61ms benötigt werden sind es bei einer Szene mit 100.000 Kugeln 6250ms. Die Cuda Implementierung rendert dagegen um einiges schneller. Selbst im unsortierten Fall benötigt Cuda für eine Szene mit 100.000 Kugeln 191,437ms und rechnet damit 30x schneller als die OpenGL Variante. Sind die Kugeln vorsortiert rendert die Cuda Implementierung sogar fast 300x so schnell als die unsortierte OpenGL Variante.

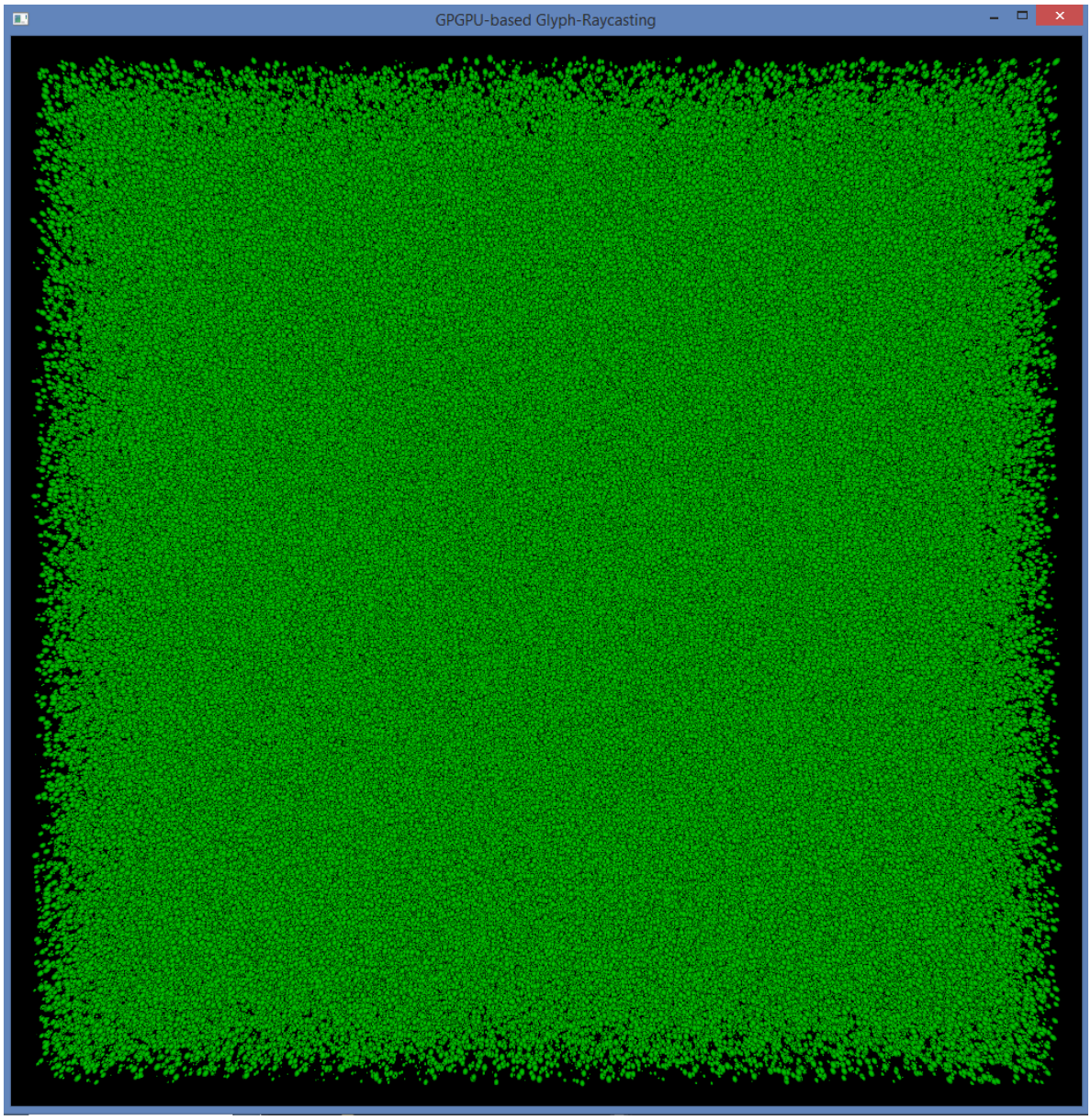

Abbildung 5.2: Eine vom Cuda Raycaster gerendete Szene mit 1.000.000 Kugeln und einem Kugelradius von 3 Pixel.

6 Zusammenfassung und Ausblick

Diese Ausarbeitung hat sich mit der Frage beschäftigt ob und wie es möglich ist einen Raycasting Algorithmus mithilfe der Cuda Architektur zu implementieren. Dazu wurden zuerst die nötigen Grundlagen des Raycastings erläutert und wieso sich dieser optimal für eine parallel Berechnung eignet. Nach der Auswahl geeigneter Ansätze wurde ein Tile-basierter Raycasting Algorithmus in Cuda implementiert. Die Ansätze unterschieden sich bei der Schleifenanordnung und ob Pixel oder Objekte parallel gerendert werden. Die umgesetzten Varianten wurden dann evaluiert indem Szenen mit unterschiedlicher Anzahl und Größen von Kugeln gerendert wurden. Besondere Aufmerksamkeit ist dabei bei der Wahl der Grid- und Blockgrößen notwendig. Selbst kleinste Änderungen haben erheblichen Einfluss auf die Laufzeit des Kernels.

Nach der Evaluierung hat sich herausgestellt dass ein Ansatz welcher in der äußeren Schleife über die Kugeln läuft und diese auch parallel rendert, die besten Leistung liefert. Die Nutzung eines lokalen Framebuffers welcher im shared Memory realisiert wurde brachte dabei auch keinen Vorteil, sondern verschlechterte in einigen wenigen Fällen sogar die Laufzeit.

Mithilfe von Nvidias Nsight wurden die Cuda Kernel genauer analysiert und darauf untersucht welche Stellen im Code die meisten Instruktionen benötigen. Die betroffenen Codeabschnitte wurden optimiert und anschließend die Laufzeiten der Kernen nochmals evaluiert. Um einen Vergleich mit einer OpenGL Implementierung herzustellen wurde ein Raycasting Algorithmus mithilfe des Fragment Shaders realisiert. Das Ergebnis dabei war dass die Cuda Implementation bei der man die Kugeln vorsortiert bei einer Szene mit einer Millionen Kugeln bis zu 300x schneller rendert als eine unsortierte OpenGL Variante.

Ausblick

Die gewonnen Erkenntnisse lassen darauf schließen dass es mit einigen weiteren Optimierungen problemlos möglich ist einen Echtzeit Raycaster in Cuda umzusetzen der eine Szene mit hunderttausend Objekten darstellen kann und auf einer Nvidia GTX 660 lauffähig ist.

Der Raycaster kann dann auch ohne weiteres mit einigen weiteren Objekten erweitert werden. Dazu müssen die jeweils benötigten Schnitttest Funktionen und passenden Sortieralgorithmen implementiert werden.

Potential bietet eventuell auch die Nutzung von Texturspeicher für die Objektdaten. Dadurch umgeht man auch das Problem des „coalesced memory“ Zugriffs welches bei der Nutzung von globalem Speicher auftritt.

Die Kugeln werden in ein zweidimensionales Grid sortiert welches der Aufteilung der Bildebene entspricht. Denkbar ist hier auch eine dreidimensionale Aufteilung in der die Kugeln noch nach ihren z-Werten sortiert werden. Dazu muss die vector Datenstruktur welche die Kugeln beinhaltet um eine Dimension erweitert werden. Kugeln welche sich näher an der Bildebene befinden werden dann innerhalb des Kernels zuerst gerendert. Das aktualisieren des Framebuffers dürfte bei sehr dichten Szenen dann um einiges geringer ausfallen.

Weitere Optimierungen sind eventuell durch zusätzliches Einsetzen von „intrinsic cuda functions“ möglich. Diese bieten für manche Mathematischen Operationen eine kleine Leistungsverbesserung und werden zum Teil schon in der hit Funktion, welche für den Schnittpunkt Test zuständig, ist verwendet. Jedoch verbessert sich durch Nutzen dieser Funktionen nicht automatisch die Laufzeit. An einigen Stellen verschlechterten der Einsatz dieser Funktionen auch die Laufzeit minimal. Da müsste man genauer untersuchen an welchen Stellen der Einsatz davon sinnvoll ist.

Literaturverzeichnis

[Ago05] M. K. Agoston. *Computer Graphics and Geometric Modeling*. Springer-Verlag, 2005.

[App68] A. Appel. Some techniques for shading machine renderings of solids. Technischer Bericht, IBM Research Center, 1968. (Zitiert auf Seite 9)

[Chr05] M. Christen. *Ray Tracing on GPU*. Diplomarbeit, University of Applied Sciences Basel (FHBB), 2005.

[cud] Memory Coalescing. URL https://www.cac.cornell.edu/vw/gpu/coalesced.aspx. (Zitiert auf Seite 25)

[Gla91] A. S. Glassner. *An Introduction to Ray Tracing*. Academic Press, 1991.

[Gor12] S. J. Gortler. *Foundations of 3D Computer Graphics*. MIT Press, 2012.

[HDM+13] J. F. Hughes, A. van Dam, M. McGuire, D. F. Sklar, J. D. Foley, S. K. Feiner, A. Kurt. *Computer Graphics - Princples and Practice*. Ad, 2013.

[Jos12] N. M. Josuttis. *The C++ Standard Library*. Ad, 2012.

[LHXHEH10] F. Liu, M.-C. Huang, L. Xue-Hui, W. En-Hua. FreePipe: a programmable parallel rendering architecture for efficient multi-fragment effects. *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, 2010. (Zitiert auf Seite 10)

[LJM12] S. B. Lippman, L. Josee, B. E. Moo. *C++ Primer*. Add, 2012.

[MT97] T. Möller, B. Trumbore. Fast, Minimum Storage Ray/Triangle Intersection. *Journal of Graphics Tools*, 1997. (Zitiert auf Seite 13)

[Rei08] G. Reina. *Visualization of Uncorrelated Point Data*. Dissertation, Universitaet Stuttgart, 2008. (Zitiert auf den Seiten 10 und 32)

[Rou] Rougier. Modern OpenGL Tutorial. URL http://www.labri.fr/perso/nrougier/teaching/opengl/. (Zitiert auf Seite 12)

[Rya] S. Ryan. The Nvidia GeForce GTX 660 Review. URL http://www.anandtech.com/show/6276/nvidia-geforce-gtx-660-review-gk106-rounds-out-the-kepler-family. (Zitiert auf Seite 17)

[Sco82] R. Scott. Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18(2):109–144, 1982. (Zitiert auf Seite 9)

[SK12] J. Sanders, E. Kandrot. *Cuda by example - An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2012. (Zitiert auf Seite 15)

[Suf07] K. Suffern. *Ray Tracing from the Ground Up*. A K Peters, Ltd., 2007.

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005.

[wik] URL http://en.wikipedia.org/wiki/SIMD. (Zitiert auf Seite 16)

[Wil13] N. Wilt. *The Cuda Handbook*. Add, 2013.

Alle URLs wurden zuletzt am 10. 02. 2015 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift