Raytracing Cubes

Implementierung eines beschleunigten Raytracers mit CUDA auf der GPU

1. Nicolai Stephan
HTWG Konstanz
Konstanz, Deutschland
nicolai.stephan@htwg-konstanz.de

2. Kevin Olomu
HTWG Konstanz
Konstanz, Deutschland
kevin.olomu@htwg-konstanz.de

3. Andreas Roth
HTWG Konstanz
Konstanz, Deutschland
andreas.roth@htwg-konstanz.de

4. Marco Fehrenbach
Institut für Optische Systeme
HTWG Konstanz
Konstanz, Deutschland
mfehrenb@htwg-konstanz.de

***Abstrakt*— Raytracing bezeichnet eine alternative Render-Methode, um virtuelle Szenen auf dem Computer darzustellen. Die Idee basiert auf der Verfolgung von Strahlen vom Auge des Betrachters zu der Lichtquelle. Die normale GPU Pipeline basiert auf dem Verfahren des Rasterizers und kann keine Strahlen in die Szene verfolgen. Deshalb sind Raytracing-Algorithmen größtenteils auf der CPU zu finden und sehr Performance lastig. Da die GPU jedoch äußerst effizient im Verarbeiten von Vektoren (Strahlen) und Matrizen ist, kann mithilfe von CUDA eine (effiziente) eigene Implementierung erfolgen. Das Ziel ist die Implementierung eines Raytracers mit Würfeln in einer Szenerie sowie die Performance Evaluation im Verhältnis zur CPU.**

***Stichworte*—Ray Tracer, CUDA, Würfel, AABB, GPU, OpenGL**

I. Einführung

Raytracing bezeichnet eine alternative Render-Methode, um virtuelle Szenen auf dem Computer darzustellen. Die Idee basiert auf der Verfolgung von Lichtstrahlen von der Lichtquelle in das Auge des Betrachters. Physikalisch korrekt müsste man eine unendliche Anzahl an Strahlen von der Lichtquelle in die Szene verfolgen, nur damit wenigstens ein paar davon die Kamera treffen.

Der Ansatz von Raytracing ist deshalb, Strahlen von der Kamera aus in die Szene zu senden, Schnittpunkte mit den Objekten zu berechnen und aus diesen Schnittpunkten dann die Farbe zu bilden. Von einem definierten ”Blickpunkt” aus werden Strahlen durch jeden Pixel der Bildebene in die Szene geworfen. Dabei ist es möglich auch eine unterschiedliche Anzahl an Strahlen pro Pixel zu senden. Falls ein Schnittpunkt vom Strahl des Betrachters mit dem Objekt existiert, folgt daraufhin ein weiterer Strahl, welcher vom Schnittpunkt des Objekts zur Lichtquelle überprüft, ob ein Schatten vorliegt.

Die normale GPU Pipeline basiert auf dem Verfahren des Rasterizers und kann keine Strahlen in die Szene verfolgen. Deshalb sind Raytracing-Algorithmen größtenteils auf der CPU zu finden und sehr Performance lastig. Da die GPU jedoch sehr effizient im Verarbeiten von Vektoren (Strahlen) und Matrizen ist, kann mithilfe von CUDA eine eigene Implementierung zur parallelen Berechnung der Farbwerte jedes Pixels erfolgen.
Das Ziel dieses Projekts ist die Implementierung eines Raytracers mit einfachen Objekten in der Szene. Als Objekte werden Würfel verwendet, da diese sich leicht durch mathematisch implizite Funktionen darstellen lassen und deshalb sehr günstig in der Schnittpunktberechnung sind. Nachdem der Schnittpunkt vom Auge zum Objekt berechnet worden ist, folgen abhängig vom gewählten Material weitere Strahlen vom berechneten Schnittpunkt des Objekts zur Lichtquelle. Die Menge an Effekten wird dabei der Einfachheit halber auf Spiegelungen sowie undurchsichtiges Material beschränkt.
Als Lichtquelle dient ein diffuses Licht. Diese besteht rein technisch aus einer Ansammlung von punktuellen Lichtquellen mit variabler Helligkeit und sorgt beim Rendern für weiche Schattenübergänge. Damit die Strahlen jedoch nicht immer weitere Strahlen generieren, ist eine geeignete Endbedingung (maximale Baumtiefe) wichtig, welche hier auf eine Tiefe von 3 Ebenen festgelegt wurde.

Für OpenGL wird anschließend die entsprechend berechnete Farbe als Pixel gerendert. Als Vereinfachung wurde hier nur mit den Farben gearbeitet.

II. Grundlagen

A. Ray Tracing

Technisch beschreibt das Raytracing die Verfolgung von Strahlen in eine Szenerie. Diese können sowohl von einer vorhandenen Lichtquelle (forward Raytracing), als auch von einer Kamera ausgehen (backward Raytracing). Da beim forward Raytracing die Wahrscheinlichkeit deutlich geringer ist, dass ein Lichtstrahl überhaupt die Kameraposition erreicht, wird häufig das backward Raytracing für ein schnelleres Ergebnis verwendet.

Allgemein wird bei beiden Verfahren ein Strahl generiert und der erste Schnittpunkt mit einem Objekt in der Szene gesucht. Von diesem ermittelten Punkt aus können beliebige weitere Strahlen für die Pfad- oder Schattenverfolgung ausgehen. Diese unterscheiden sich dahingehend, dass bei Pfadverfolgungsstrahlen nur der erste Schnittpunkt mit einem

Objekt aus der Szene ermittelt wird. Bei den Schattenstrahlen reicht es auch aus einem Schnittpunkt auf der Stecke zwischen Objekt und Lichtquelle zu nehmen. Daraus resultiert ein geringerer Aufwand in der Berechnung.

Die angewandte Path Raytracing Methode ist eine besondere Art des grundlegenden Raytracings. Dabei wird nur genau ein Strahl ausgehend von der Kamera pro Durchlauf und Pixel berechnet und es entsteht dadurch kein sich verzeigender Strahlenbaum. An jedem Schnittpunkt mit einem Objekt wird abhängig vom Material durch einen Zufallsfaktor (Russisch Roulette) der ausgehende Strahl bestimmt. Dieser kann in diesem Fall entweder diffus oder gespiegelt sein. Im gespiegelten Fall ist der nächste Strahl eindeutig (2), während beim diffusen Material die neue Richtung mit dem Normalenvektor plus einem Zufallsfaktor bestimmt wird.

Das Ende der Strahlenverfolgung ist erreicht, wenn kein Schnittpunkt mehr für den Strahl gefunden wurde oder die eingestellte maximale Strahlentiefe erreicht ist. Eine beispielhafte Szene mit Strahlenverfolgung darin ist in nachfolgende Abbildung 1 dargestellt.

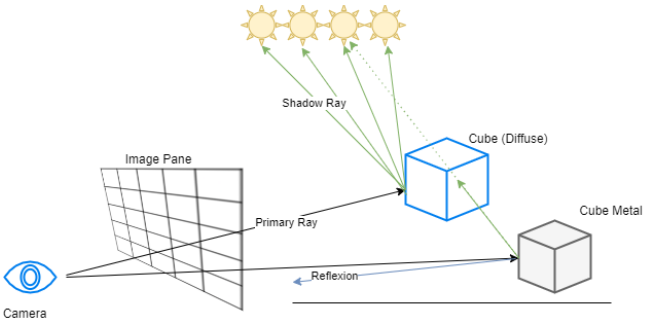

Abbildung 1 Funktionsweise Raytracer

Der wesentliche Vorteil des Path Raytracings liegt in der niedrigen Laufzeit pro Iteration. Der Einfluss der Zufallswerte sowie der Monte-Carlo-Methode (Russisch Roulette) sorgt dabei aber für eine große Varianz und damit ein „rauschendes Bild“ bei niedrigen Durchläufen (kleiner 50), sodass als Standardwert hier mindestens 100 Integrationen benötigt werden. Ein weiteres Plus der zufälligen Verfolgung von Strahlen ist, dass damit eine globale Beleuchtung besser als beim standardmäßigen Verfahren simuliert werden kann, um Effekte wie das „Color Bleeding“ aus Abbildung 2 zu generieren.

B. Anti-Aliasing

Aliasing bezeichnet den Effekt, dass Zacken an den Kanten entstehen. Dadurch, dass die Welt vom Auge aus über die Strahlen abgetastet und dann gerendert wird, können solche "Fehler" bzw. harte Übergänge entstehen. Um das zu vermeiden, gibt es unterschiedliche Anti-Aliasing Techniken. Dafür werden für jeden Pixel mehrere Strahlen mit einem zufällig jeweils leicht veränderten Winkel in die Szene und der berechnete Durchschnitt des Pixels als Farbwert genutzt.

Als Ergebnis sieht man in der rechten Abbildung 2, dass die Würfelkannten im rechten Bild deutlich „weicher“ als im linken Bild erscheinen und Effekte wie „Color Bleeding“ auf der spiegelnden Oberfläche schön zu erkennen sind.

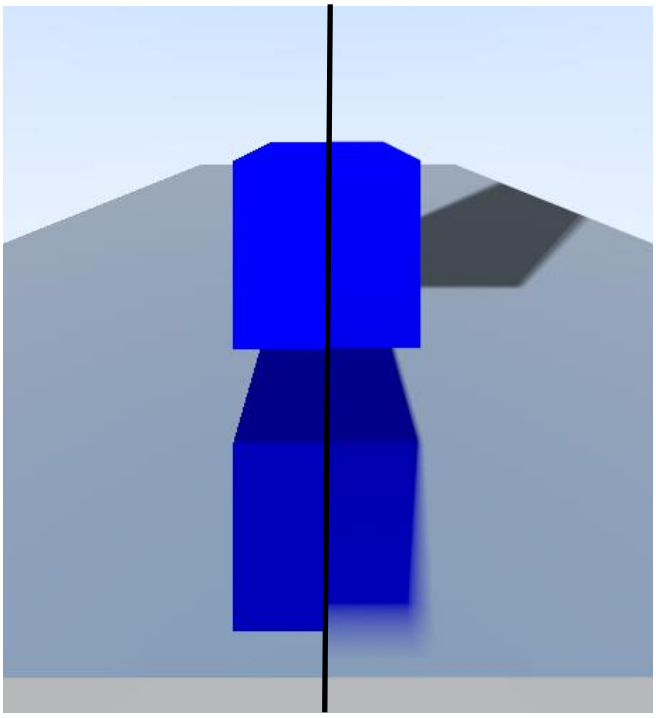

Abbildung 2 Anti-Aliasing, 1600x900, 1000 Strahlen pro Pixel

C. Axis-Aligned-Bounding-Box für Schnittpunktberechnung

Für die Schnittpunktberechnung zwischen einem Strahl und einer Box wird der „Axis-Aligned-Bounding-Box“ (AABB) Algorithmus verwenden. Dabei liegt die Box auf dem gleichen Koordinatensystem wie die Kamera (Axis-Aligned). Für jeden Würfel wird ein Minimum sowie eine Maximum Schranke benötigt. Die Schranken sind dabei Geraden welche parallel zu jeder Koordinatenachse verlaufen. Über die Gleichung ersten Grades und algebraischen Umformungen lässt sich bestimmen, wo der Strahl an der jeweiligen Achse schneidet. Die Gleichungen (1) für jede Achse sind folgende:

$$
\begin{aligned}
t_{Min}x &= (B_0x - Ox)/Dx \\
t_{Max}x &= (B_1x - Ox)/Dx \\
t_{Min}y &= (B_0y - Oy)/Dy \\
t_{Max}y &= (B_1y - Oy)/Dy \\
t_{Min}z &= (B_0z - Oz)/Dz \\
t_{Max}z &= (B_1z - Oz)/Dz
\end{aligned}
\qquad (1)
$$

- t repräsentiert den Wert für den Schnittpunkt der jeweiligen Achse
- B ist die Schranke
- O ist der Ursprungspunkt
- D ist der Richtungsvektor

Für jede Achse wird anschließend überprüft, ob tatsächlich ein Schnittpunkt mit dem Würfel vorliegt. Dafür werden von den zuvor berechneten Parameter *t* der kleinste sowie der größte Wert ermittelt. Angefangen wird mit der X-Achse und überprüft ob *tyMax* kleiner als *tMin* oder *tyMin* größer als *tMax*, falls dies der Fall sein sollte, wurde der Würfel nicht getroffen. Siehe folgende Abbildung:

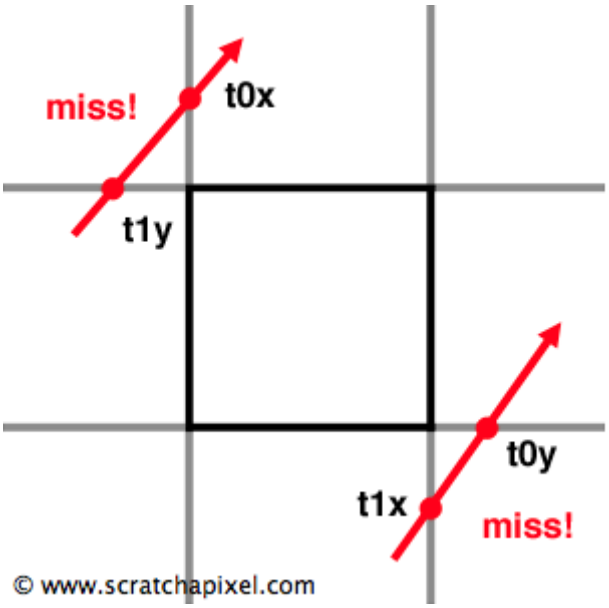

Abbildung 3 AABB [5]

Dieselbe Prozedur wird auch für die z-Achse ausgeführt. Anschließend wird das größte *tMin* und das kleinste *tMax* gespeichert und, sollten alle if-Abfragen, welche „false“ zurückgeben, übersprungen sein, „true“ zurückgegeben. [5]

D. Diffuses Licht

Für eine natürliche Beleuchtung der Szenerie wird eine diffuse Lichtquelle verwendet. Diese sendet ein gleichmäßiges, weichzeichnendes Licht ohne starke Schattenbildung oder Kontraste aus. Dabei sind technisch gesehen mehrere Lichtpunkte auf einer bestimmten Fläche verteilt angebracht, welche mit gleicher oder unterschiedlicher Intensität leuchten.

Für die diffuse Beleuchtung ist der Einfallswinkel der Lichtquelle auf die Oberfläche wichtig. Bei einem senkrechten Strahl des Lichtes auf die Fläche (Licht- und Normalenvektor nahezu parallel) erscheint die Beleuchtung an diesem Punkt am stärksten. Mit Vergrößerung des Winkels zwischen den Vektoren sinkt die Beleuchtungsstärke. Mittels voriger Normierung der beiden Vektoren kann eine Berechnung der Beleuchtungsintensität über das Skalarprodukt erfolgen.

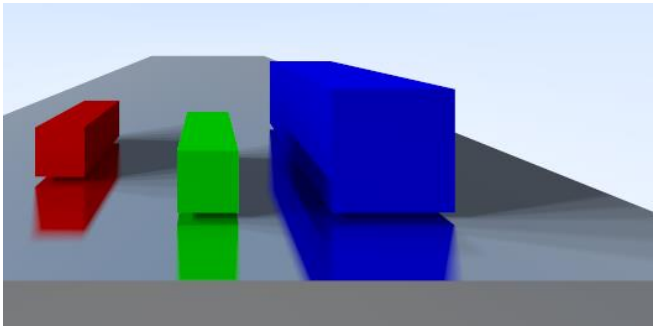

Abbildung 4 Ausschnitt für diffuses Licht,1920x1080, 4000 Strahlen pro Pixel

E. Reflektionseffekte

Für die Reflektion wurde je nach Material eine andere Streumethode (engl. scatter) implementiert. Falls es sich um einen undurchsichtigen Würfel (engl. Lambertian) handelt, wird ein weiterer Strahl vom Schnittpunkt des Objekts zur Normalen (plus eine kleine zufällige Abweichung) erzeugt.

Das Material „Metall“ reflektiert vom Schnittpunkt des Objekts. Dafür bietet „glm“ die “reflect”-Methode an. Der einfallende Strahl (I) und die Normale der Oberfläche (N) werden verwendet um die Reflektion nach der Formel (2):

$$reflect = I - 2.0 * dot(N, I) * N \qquad (2)$$

zu erzeugen.

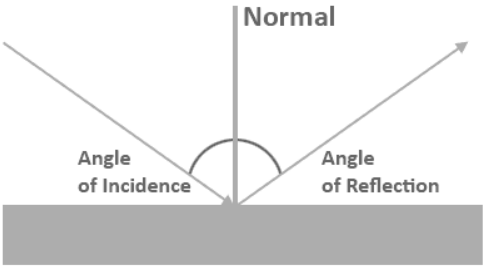

Abbildung 5 Reflektionswinkel [8]

Zusätzlich wurde ein “fuzz”-Parameter erstellt. Dieser gibt an wie stark das Material reflektiert. Der Strahl wird also über die ”reflect”-Methode mit dem “fuzz”-Parameter und einer zufälligen Abweichung erstellt.

F. GPU & CUDA

Eine GPU (Graphics Processing Unit) ist für hoch parallelisierbare Programmsequenzen und das Rendern sehr gut geeignet. So kann eine GPU beispielsweise mehrere Pixel echt parallel berechnen. Durch SIMD Kerne, die Hauptkomponente der GPU, wird eine hohe Parallelität der Datenverarbeitung erreicht [7]. Im Vergleich zu einer CPU weisen GPUs eine große Anzahl an Kerne auf, welche über tausende Threads, parallele Berechnungen ermöglichen. Die CPU hingegen hat weniger Kerne, kann dafür aber komplexere Aufgaben ausführen und ist für die Latenz optimiert.

CUDA steht für (Compute Unified Device Architecture) und ist eine Grafikkarten-Programmierschnittstelle, welche von NVidia entwickelt worden ist. Die Kernel-Funktionen wurden in CUDA-Dateien mit der „.cu“ Endung geschrieben, welche dann von dem CUDA Compiler prozessiert werden.

G. OpenGL

Zur Darstellung des Raytracers wird OpenGL verwendet. OpenGL erlaubt die Darstellung von komplexen Szenen in Echtzeit. Es wird ein Quadrat in OpenGL gezeichnet welches mit einer Textur belegt wird. Die Farbe der Textur ist das Ergebnis des Render-Kernels. Dabei wird das „Pixel-Buffer-Object“ (PBO) genutzt um einen schnellen Datentransfer von und zu der GPU über den (Direct Memory Access) zu erhalten (ohne CPU-Zyklen zu verwenden) [6].

III. Implementierung

Der erste Schritt der Implementierung war die Ausgabe eines Bildes. Die Berechnung der Farbwerte für jeden einzelnen Pixels wird über einen CUDA Kernel berechnet. Die Berechnung erfolgt dabei echt-parallel und das resultierende Ergebnis wird in einer Datei ausgegeben, um zu prüfen, ob die Kommunikation zwischen CPU und GPU funktioniert. Die erstellte Datei war eine PPM Datei, welche für jeden Pixel einen Farbwert für Rot, Grün und Blau enthält (zwischen 0 und 1). Die Methode "*createPPM*" war für die Ausgabe der Farbe zuständig. Hier wurde der zuvor berechnete Wert mit 255.999 multipliziert, um eine Zahl zwischen 0 und 255 zu erhalten.

Für eine hohe Parallelität wurde die Berechnung des Pixelfarbwerts in mehrere Blöcke und Threads aufgeteilt. Dafür wurde zuerst die Blockgröße abhängig von der Bildbreite und Bildhöhe sowie der maximal verfügbaren Threads pro Block für die x und y Dimension berechnet.

Durch Aufteilung der Aufgaben in unabhängige Schritte für den Kernel, konnte aus dem „Loop-Based" Pattern, welcher sequenziell abgearbeitet wird, ein paralleles „Fork-Join" Pattern verwenden werden. Jeder Thread kann pro Pixel den Farbwert berechnen, da es sich hierbei um eine unabhängige Aufgabe handelt.

Für die Berechnung des jeweiligen Pixelindex wurde aus dem Block und Thread die x und y Werte ausgelesen. Der jeweilige Pixel auf dem Image-Plane ergibt sich aus Multiplikation von y und der Höhe des Fensters plus den x Wert, wie in Abbildung 6 ersichtlich. So konnten die Pixelkoordinate in einem 1-dimensionalen Array berechnet werden.

```
const int x = threadIdx.x + blockIdx.x * blockDim.x;
const int y = threadIdx.y + blockIdx.y * blockDim.y;
if ((x >= WIDTH) || (y >= HEIGHT)) return;
const int index =  y * WIDTH + x;
```

Abbildung 6 Pixelindex

Die If-Abfrage ist eine Überprüfung, sodass nur Pixel berechnet werden, die sich innerhalb der Image-Plane befinden. In diesem Schritt wurde dadurch implizit die Kommunikation, Speicherallokation und Synchronisierung zwischen der CPU und GPU eingerichtet. Jeder CUDA spezifische Befehl wird durch das Macro "*CUDA_CALLER*" „gewrappt", sodass falls ein Fehler auftritt, dieser in der Konsole ausgegeben wird.

Der nächste Schritt war das Initialisieren der Strahlen und das Erstellen der Szene. Wie in Abbildung 7 beschrieben, wird dabei wie folgend vorgegangen:

Algorithm 1: Initialisierung der Stahlen und Szene

Result: Generierte Strahlen, Zufallswert je Pixel, Szene
Auswahl und Setzen der CUDA kompatiblen Grafikkarte;
Allokieren GPU Speicher für Strahlen, Farben und Zufallswert jedes Threads;
Bestimmen optimaler Block- und Threadanzahl für Bildauflösung;
function STEUPRAYSKERNEL ▷ Aufruf des Kernel für Initialisierung der Strahlen
  if *Pixelindex größer Bildauflösung* **then**
    Abbruch Kernelberechung;
  else
    Initialisierung Strahlen und Zufallswert pro Thread;
  end
Allokieren GPU Speicher für Objektliste, Szene und diffuses Licht;
function CREATE_WORLD ▷ Aufruf des Kernel für Initialisierung der Szene
  if *Threadindex und Blockindex gleich Null* **then**
    Bodenfläche generieren;
    Zufällige Würfel generieren und alle in Objektliste ablegen;
    Diffuses Licht erzeugen;
  end

Abbildung 7 Algorithmus zum initialisieren der Strahlen und Szene

Hier werden zwei Kernel eingesetzt. Ein Kernel für das Erstellen der Strahlen vom Auge des Betrachters zum Objekt (Primary Rays) sowie einen weiteren Kernel für die Erstellung der Welt. Die beiden Kernels werden beim Starten der Anwendung initial einmal ausgeführt. Die Thread- und Blocksettings pro Kernel sind, wie oben bereits erwähnt, abhängig von der Grafikkarte und Auflösung des Bildes.

Würfel lassen sich erstellen mit dem Durchmesser, dem Mittelpunkt, der Farbe und dem Material. Dabei wurde der Boden als ein großer Würfel mit dem Material „Metall" definiert. Die anderen Würfel werden zufällig auf dem großen Würfel generiert.

Der nächste Schritt war die Schnittpunkt Berechnung für die Primary Rays. Alle Würfel implementieren die „*hitable"* Klasse, um für die Schnittpunkt Berechnung hit-Methode bereit zu stellen. Diese liefert einen Boolean zurück abhängig davon, ob etwas getroffen wurde. Gibt es einen Schnittpunkt so werden die Ergebnisse in dem „struct *hit_record"* hinterlegt. Die hit-Methode wird mit dem „__device__" Keyword annotiert, sodass Threads auf der GPU die Methode aufrufen können.

Die Berechnung, ob etwas getroffen wurde, folgt dem im II.C bereits erwähnten AABB Model. Hier handelt es sich um eine feine Granularität. Da die Aufgabe in sehr kleine Teilaufgaben aufgeteilt wurde, welche wenige Instruktionen pro Thread beinhalten. Die Berechnung jeden einzelnen Pixels ist hoch parallelisierbar und wird daher auf der GPU ausgeführt. Da jeder Thread die Berechnung für seinen jeweiligen Pixel individuell und unabhängig ausführt handelt es sich hierbei um „Map-Operationen."

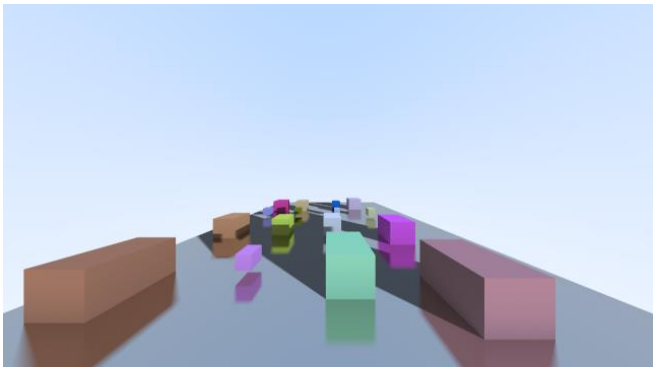

Abbildung 8 Beispielbild 640x360, 1000 Strahlen pro Pixel

Gibt es einen Schnittpunkt für den Primary Ray, wird je nach Material ein weiterer Strahl (Shadow Ray) erstellt, der abhängig vom Schnittpunkt des Objekts zur Lichtquelle überprüft, ob ein Objekt im Weg steht. Üblicherweise wird hier Rekursion eingesetzt, um so beliebig viele Strahlen zu verfolgen. Es wird mit einem „Loop-based" Pattern gearbeitet, welcher eine maximalen Strahlentiefe von 3 besitzt. Wie im Kapitel II.E bereits genannt wird hier abhängig vom Material ein unterschiedlicher Reflektionsstrahl entsendet. Die Berechnung der Pixelfarbe eines individuellen Pixels kann man als Gather-Operation betrachten. Da der Thread mehrere Datenelemente berechnet und anschließend ein Datenelement schreibt.

Bei der Lichtquelle handelt es sich um eine Menge von Lichtpunkten. Die Lichtquelle hat dabei eine Breite von 1 und eine Höhe von 5. Bei der Berechnung des Schattens wird in dem color-Kernel abhängig von der Anzahl der Lichtpunkte

ein Ray erstellt (Loop-based Pattern). Ist ein Objekt im Weg wird die Lichtdämpfung erhöht was wiederum den Schatten erzeugt.

Anschließend wurde Anti-Aliasing implementiert so wie in II.C erwähnt. Falls das Anti-Aliasing Flag gesetzt wurde, wird ein Strahl mit einer kleinen Abweichung zur Lichtquelle gesendet, um die scharfen Kanten zu verbessern.

Der nachfolgende Pseudocode in Abbildung 9 Algorithmus zum Rendern der generierten Szene soll verdeutlichen, wie die zuvor genannten Komponenten in einem Kernel in Verbindung gebracht wurden:

```
Algorithm 2: Rendern der generierten Szene
  Result: Farbwerte jedes Pixels
  Start der Zeitmessung;
  Bestimmen optimaler Block- und Threadanzahl für Bildauflösung;
    function RENDERKERNEL                    ▷ Aufruf des Kernel für Rendern des Bildes
      if Pixelindex größer Bildauflösung then
        Abbruch Kernelberechnung;
      end
      if Anti-Aliasing then
        for i ← 0 bis Anzahl Samples pro Pixel do
          zufällige x- und y-Komponente für Strahl generieren;
          Pixelfarbe berechnen und aufsummieren;          ▷ Aufruf Funktion zur
      Farbberechnung
        end
        Mittelwert der Pixelfarbe bilden;
      else
        Pixelfarbe berechnen;                  ▷ Aufruf Funktion zur Farbberechnung
      end
      Helligkeitsanpassung durch Gammakorrektur;
      Speichern der Farbwerte für Anzeige über OpenGL;
  Stop Zeitmessung;
  Kopieren Farbwerte aus GPU in CPU Speicher;
```

Abbildung 9 Algorithmus zum Rendern der generierten Szene

Die Berechnung für den Farbwert des Pixels ist dann im nächsten Pseudocode in Abbildung 10 dargestellt.

```
Algorithm 3: Farbwert des Pixels
  Result: Farbwert des Pixels
  lokale Variablen initialisieren;
  for i ← 0 to maximale Baumtiefe do
    if Objekt getroffen then
      if Material undurchsichtig oder reflektierend then
        neuen Strahl an Punkt berechnen;
        Farbwert des Material berücksichtigen;
      end
      for j ← 0 bis Anzahl Lichtpunkte do
        for k ← 0 bis Anzahl Lichtpunkte do
          Strahl zur Lichtquelle berechnen
          if kein Objekt im Weg then
            Lichtstärke am Punkt bestimmen;
          end
        end
      end
    else
      if Erster Treffer Himmel then
        return Hintergrundfarbe;
      end
      return aktuellerFarbwert;
    end
  end
  return Gesamtfarbwert;
```

Abbildung 10 Algorithmus für den Farbwert des Pixels

Nachdem die Pixel korrekt berechnet wurden, folgte die OpenGL Anbindung in das Projekt. Durch den Frame-Buffer mit den Eigenschaften der Szene (Breite und Höhe) und dem orthogonalen Viewport wird das Fundament für das PBO und die Textur gelegt. Anschließend wird initial der Speicher für das Bild allokiert (Höhe * Breite * Farb-Tiefe * Größe *GLuByte*). Darauf folgt die Erstellung der Textur in OpenGL. Diese wird als ein 2D Bild initialisiert.

Als nächstes wird, das das PBO erstellt und mit der Textur verbunden. Die Verbindung zwischen einer CUDA Resource und dem PBO (OpenGL) wird anschließend in der Render-Schleife festgelegt. Das PBO wird daraufhin einem Kernel in CUDA übertragen, um die Farbwerte zu berechnen. Nachdem der Kernel die Farbwerte berechnet hat, wird ein Rechteck mit den Dimensionen von 0 bis Breite des Fensters und 0 bis Höhe des Fensters angelegt. Auf diesem Rechteck werden dann die berechneten Farbwerte eingetragen. „Glew“ wird genutzt, um die verfügbaren OpenGL Erweiterungen auf dem aktuellen System zu erfragen und GLFW wird verwendet, um das Fenster zu erstellen und auf User Eingaben zu reagieren.

Durch die Benutzung des PBO müssen Funktionen verwendet werden, die noch aus der OpenGL Version 2 stammen. Da das vermieden werden soll, kann man die berechneten Farbwerte auch mithilfe eines Vertex-Buffer-Objects (abgekürzt VBO) darstellen. Es wird die Zeichenmethode für Punkte verwendet, den Punkten einen Ort im dreidimensionalen Raum zugeteilt (in diesem Fall mit einer Konstanten in z-Richtung) und den Punkten den bereits berechneten Farbwert zugewiesen.

Damit man für eine Bewegung der Lichtquelle nicht jedes Mal den Wert im Code ändern muss, wurde eine Positionsveränderung der Lichtquelle zur Laufzeit durch Tasteneingabe implementiert. Da sich die Lichtquelle auf dem Device, also der GPU, befindet, muss die Positionsveränderung über Kernel Funktionen stattfinden. Beim Bewegen der Lichtquelle müssen auch alle Lichtpunkte innerhalb der Lichtquelle mitbewegt werden.

Um nicht nur Performancemessungen auf der GPU durchführen zu können, sondern um diese auch mit der CPU vergleichen zu können, wurde der Ray Tracer auch auf der CPU implementiert. Hierfür wurden die Parallelisierungsschritte rückgängig gemacht, um die Berechnungen iterativ auf der CPU ausführen zu können. Dieser Schritt ist notwendig, um später den Beschleunigungsfaktor berechnen zu können.

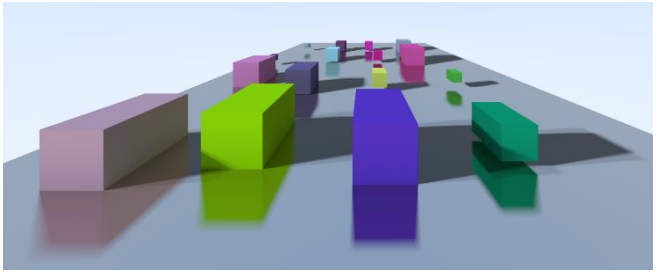

Abbildung 11 Ausschnitt 3840x2160 (4K), 1000 Strahlen pro Pixel

IV. EVALUIERUNG

Die Evaluierung der Implementierung des Raytracers in CUDA werden in diesem Kapitel vorgestellt. Als Bewertungskriterien für die Performance werden die durchschnittlichen Bilder pro Sekunde (abgekürzt FPS), mit und ohne aktives Anti-Aliasing für eine Dauer von 10s des Programmes ermittelt. Dafür wurden in verschiedenen Testszenarien die Anzahl an Würfel, die Anzahl und Position der diffusen Lichtquelle sowie die Auflösung der Bildebene verändert.

Über Messungen der Auslastungsrate der GPU wird das Verhältnis des durchschnittlich aktiven Warps pro aktiven Zyklus zur maximal unterstützten Anzahl von Warps (engl. achieved occupancy). Sowie das Verhältnis der durchschnittlich aktiven Threads pro Warp zur maximal unterstützten Anzahl von Threads pro Warp (engl. warp execution efficiency). Diese Ergebnisse liefern allerdings nur eine einfache Aussage über die Performance des entwickelten Softwarecodes, da Faktoren wie Parallelisierung der Codeanweisungen und Strategien zur Latenzminimierung ebenfalls sehr effektiv sein können. [9]

Die folgenden Auswertungen wurden auf einem System mit diesen Spezifikationen durchgeführt:

GPU: GeForce GTX 1080

- CUDA-Kerne: 2560
- Taktfrequenz: 1695 MHz
- Speicher-Datenrate: 10,01 Gbps
- Speicherbandbreite: 320,32 GB/s
- Dedizierter Videospeicher: 8192 MB GDDR5X
- GridSize: 2147483647x65535x65535
- Maximale Threads pro Block: 1024

CPU: Intel i7-4770K 3,50 GHz

Grundsätzlich gelten für die folgende Legenden diese Informationen:

Legende	**Beschreibung**
GPU AA	Auf der GPU mit Anti-Aliasing durchgeführt
GPU	Auf der GPU ohne Anti Aliasing durchgeführt
CPU	Auf der CPU ohne Anti Aliasing durchgeführt
ThreadSize 8	Wurde mit Threadsettings 8x8 durchgeführt
ThreadSize 16	Wurde mit Threadsettings 16x16 durchgeführt
ThreadSize 32	Wurde mit Threadsettings 32x32 durchgeführt

Falls nicht weiter spezifiziert, wurden folgende Standardwerte verwendet:

Bezeichnung	**Wert**
Anzahl Strahlen pro Pixel	100
Strahlentiefe	3
Anzahl Würfel	17
Anzahl Lichtquellen	9
Auflösung	640x360

Die Auswertungen wurden mit NVIDIA Nsight Systems 2020.4.3 durchgeführt. Die FPS wurde dadurch auf ein Maximum von 144 beschränkt. Bei Werten, die an die 144 grenzen, kann man davon ausgehen, dass der tatsächliche Wert auch weitaus höher liegen kann. Für diese Evaluierung spielt diese Beschränkung allerdings keine Rolle.

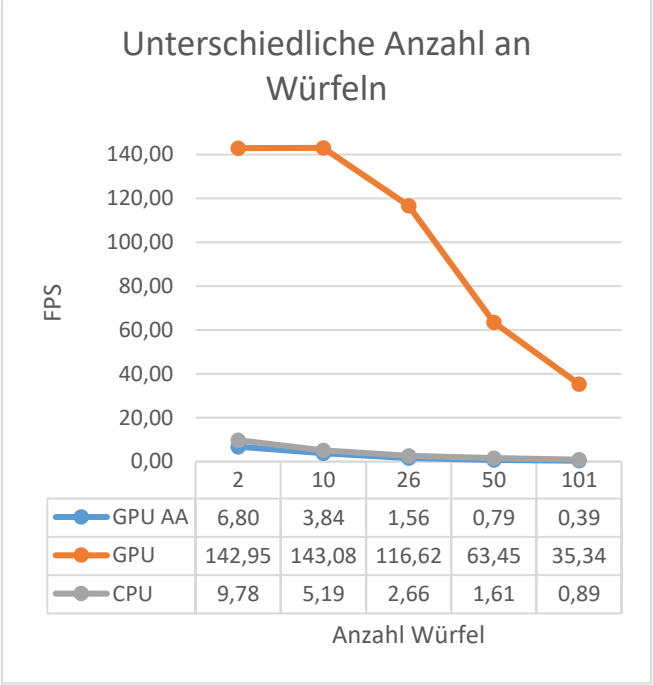

Abbildung 12 Unterschiedliche Anzahl an Würfeln

In der Abbildung 12 wurde das Verhalten der durchschnittlichen FPS in Abhängigkeit unterschiedlicher Anzahl an Würfeln gemessen. Je mehr Würfel in der Szene sind, desto niedriger werden die FPS. Das liegt daran, dass für jeden Strahl, den man berechnet, über die Würfel iteriert werden muss und dabei für jeden Würfel die hit-Methode ausgeführt werden muss, um herauszufinden, ob der Würfel getroffen wird oder nicht.

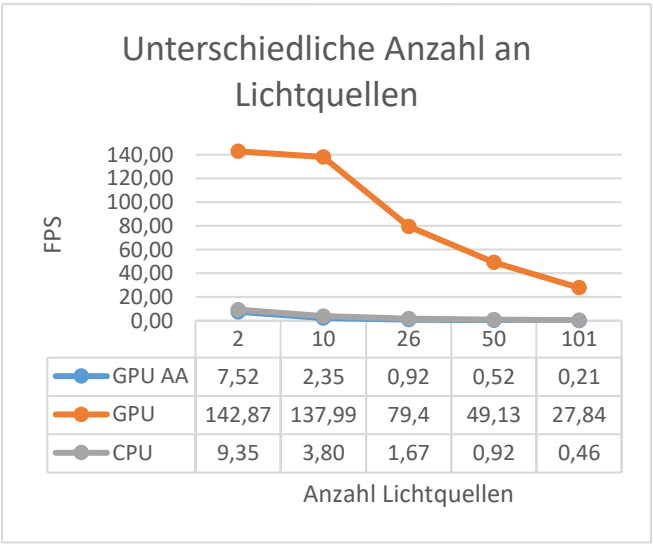

Abbildung 13 Unterschiedliche Anzahl an Lichtquellen

In der Abbildung 13 wurde das Verhalten der durchschnittlichen FPS in Abhängigkeit unterschiedlicher Anzahl an Lichtquellen gemessen. Auch hier kann man erkennen, dass die FPS sinkt, je mehr Lichtquellen berechnet werden. Auch sieht man, dass die FPS stärker sinkt als bei

den Würfeln. Das liegt daran, dass für jeden Strahl pro Auftreffpunkt auf ein Objekt jeweils ein weiterer Strahl pro Lichtquelle berechnet werden muss. Da die Strahltiefe auf 3 eingestellt ist, müssen also bei manchen Strahlen mehrere Male die weiteren Strahlen pro Lichtquelle berechnet werden.

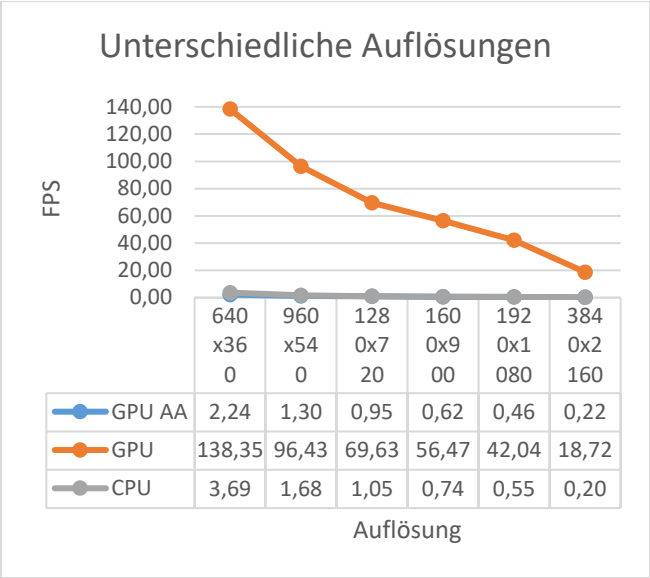

Abbildung 14 Unterschiedliche Auflösungen

In der Abbildung 14 wurde das Verhalten der durchschnittlichen FPS in Abhängigkeit unterschiedlicher Auflösungen gemessen. Hier sinkt die FPS bei steigender Auflösung. Das kann damit erklärt werden, dass man pro Pixel ein Strahl berechnen muss. Werden es also mehr Pixel, so müssen auch mehr Strahlen berechnet werden (Primary Rays = Breite * Höhe).

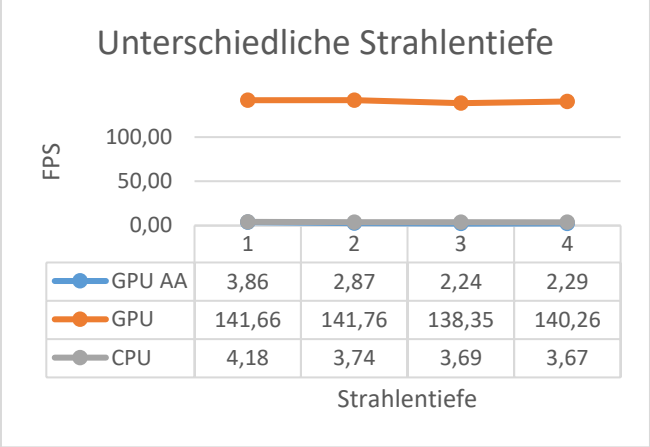

Abbildung 15 Unterschiedliche Strahlentiefe

In der Abbildung 15 wurde das Verhalten der durchschnittlichen FPS in Abhängigkeit unterschiedlicher Strahlentiefe gemessen. Je größer die Strahlentiefe, desto niedriger wird die FPS. Das kann dadurch erklärt werden, dass pro Auftreffpunkt eines Strahls ein weiterer berechnet werden muss, solange die Strahlentiefe noch nicht erreicht wurde. Man kann allerdings sehen, dass sich die Strahlentiefen 3 und 4 kaum unterscheiden. Das liegt daran, dass es in diesem Szenario kaum Möglichkeit gibt, dass Strahlen öfter als drei Mal abprallen.

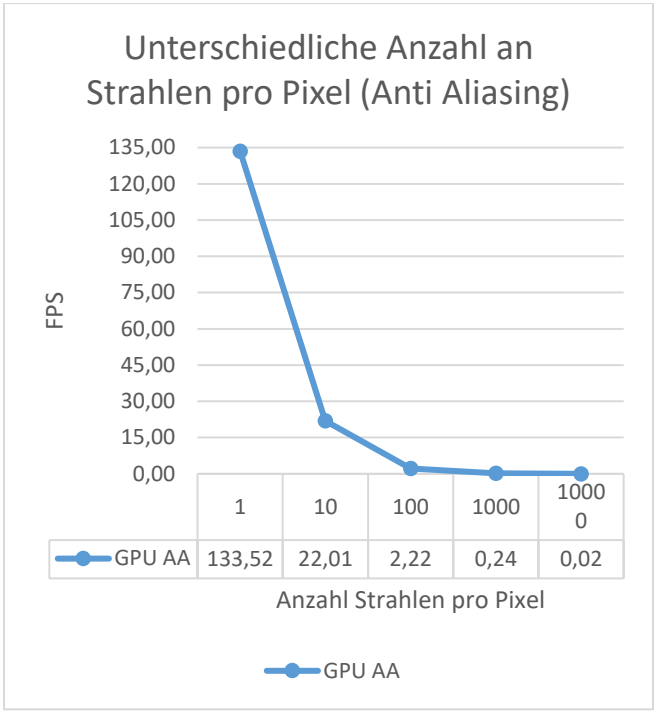

Abbildung 16 Unterschiedliche Anzahl an Strahlen pro Pixel

In der Abbildung 16 wurde das Verhalten der durchschnittlichen FPS in Abhängigkeit unterschiedlicher Anzahl an Strahlen pro Pixel (für Anti-Aliasing) gemessen. Hierbei wurde ausschließlich die GPU genutzt. Auf der CPU konnte man das Experiment nur mit einer Anzahl von 10 berechnen. Ab 100 konnten in NVIDIA Nsight Systems keine Daten mehr ausgewertet werden. Für höhere Anzahl an Strahlen pro Pixel, sinken die FPS. Das kann dadurch erklärt werden, dass jetzt für jeden Pixel nicht nur ein, sondern mehrere Strahlen berechnet werden müssen.

Auf den folgenden beiden Abbildungen wird die y-Achse in Prozent angegeben. Optimale Ergebnisse für beide Auswertungen liegen bei 100%.

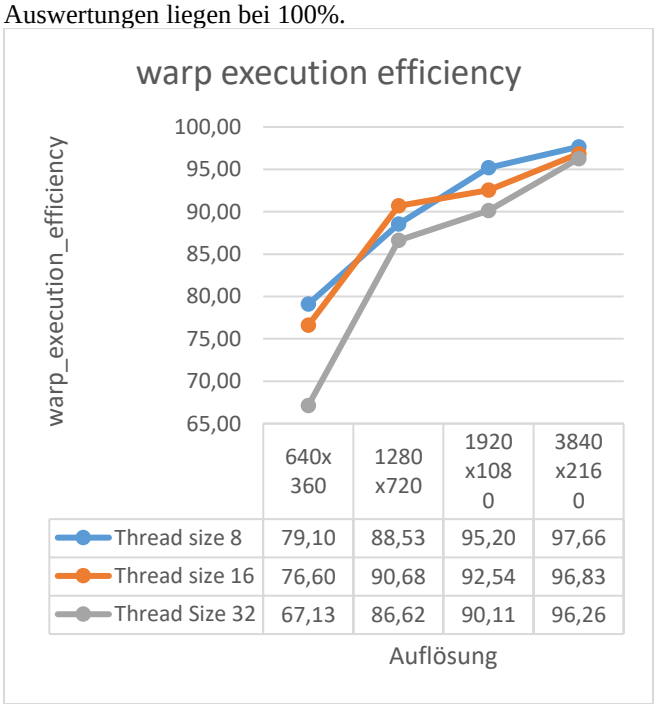

Abbildung 17 warp execution efficiency

Die „warp execution efficiency“ ist das Verhältnis der durchschnittlich aktiven Threads pro Warp zur maximalen Anzahl von Threads pro Warp, die auf einem Multiprozessor unterstützt werden. Man kann erkennen, dass die „warp execution efficiency“ steigt, je höher die Auflösung ist. Hier kann man erkennen, dass die Thread Größe von 32x32 deutlich zurückliegt. Die beiden anderen sind ähnlich, allerdings im Durchschnitt liegt die Auswahl von 8x8 vorne.

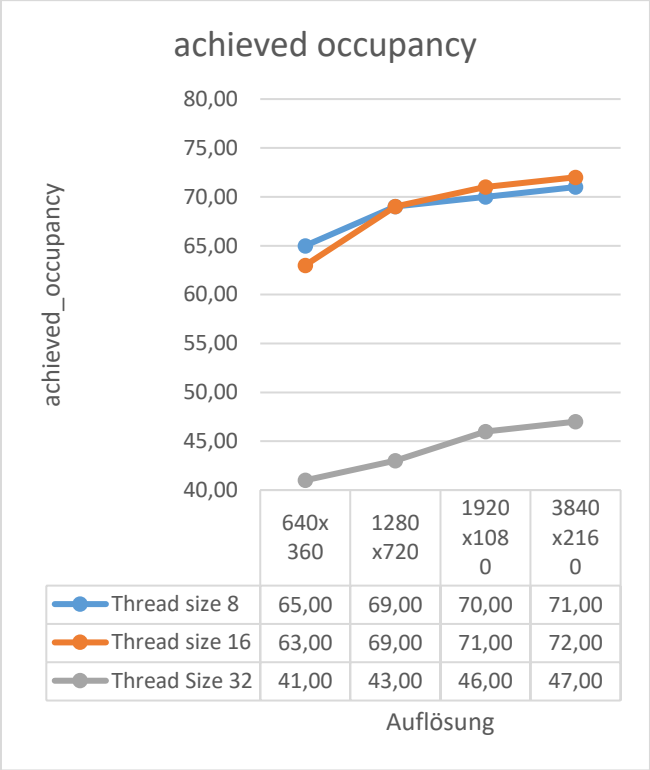

Abbildung 18 achieved occupancy

Die „achieved occupancy“ beschreibt das Verhältnis des durchschnittlich aktiven Warps pro aktiven Zyklus zur maximalen Anzahl von Warps, die auf einem Multiprozessor unterstützt werden. Hier kann man erkennen, dass die Auswahl von 32x32 Threads eine schlechte Wahl ist, wenn man die „achieved occupancy“ maximieren möchte. Die Auswahl zwischen 8x8 und 16x16 unterscheiden sich wieder nur leicht. Durchschnittlich liegt die Thread Auswahl von 8x8 leicht vorne, weshalb sich dafür entschieden wurde.

Für die „achieved occupancy“ von nur ca. 70% können verschiedene Gründe ausgemacht werden. Die Kernel Funktion, die für das Rendern zuständig ist, also für die meiste Arbeit, ist eher grob-granular. Sinnvoller wäre es an dieser Stelle die Aufgaben weiter aufzuteilen. Beispielsweise, dass die Berechnung des Lichts in eine separate Funktion ausgelagert.
Zusätzlich wird beim Berechnen der Farbe, wenn Anti-Aliasing aktiviert ist, eine Schleife aufgerufen, die daten-parallel arbeiten könnte.
Die Objekte in der Szene könnten zudem im konstanten Speicher statt im globalen abgelegt werden. da diese zur Laufzeit in ihrer Größe bereits bekannt sind. Bei einer so hohen Differenz zum optimalen Wert der 100% wäre es sinnvoll diese Verbesserungen umzusetzen, um die volle Leistung der GPU ausnutzen zu können.

Es wurde außerdem darauf geachtet, dass auf der GPU keine double floating point (fp) Instruktionen ausgeführt werden. Diese beanspruchen bei weitem mehr Leistung als die single fp Instruktionen, da diese nur 32 anstatt 64 Bit nutzen.

Kernel Function	**Single fp**	**Double fp**
create_world	689	0
renderKernel	4,5566e+10	0
setupRaysKernel	2880000	0

Der Beschleunigungsfaktor sagt aus, wie hoch die Beschleunigung der Parallelisierung auf der GPU gegenüber einer Implementierung auf der CPU ist.

Beschleunigungsfaktor**:**
1/ (270,70/7,23/2560) = 68,37

V. Fazit

Mit einem Beschleunigungsfaktor von 68,37 hat sich das Auslagern der hoch parallelisierbaren Aufgaben auf die GPU (für die Berechnung der Pixel) definitiv gelohnt.

Die „achieved occupancy“ von 70% lässt sich verbessern, indem man die Lichtquelle auslagert. Dadurch wird es fein granularer und somit kann die GPU mehr parallelisieren.

Zusätzlich lässt sich durch Auslagerung der Berechnung der Inversen für den Richtungsvektor des Strahls eine bessere Performance zusichern, indem weniger Multiplikationsoperationen benötigt werden. Statt die Inverse in der hit-Methode des Würfels zu berechnen, sollte es schon im Ray selber vorberechnet werden.

Die Darstellung mit OpenGL 2 über das „Pixel Buffer Objekt“ lässt sich verbessern, indem man auf „VBO’s“ wechselt.

Zudem lässt sich die Performance verbessern, indem die Anzahl der zu berechneten Schnittpunkte gesenkt werden. Es gibt Datenstrukturen, welche die Schnittpunktberechnung von Objekten reduziert. Beispielsweise gibt es baumartige Datenstrukturen wie „Bounding Volume Hierarchien“ und „kd-trees“. Diese reduzieren die Anzahl der zu berechneten Schnittpunkte [10].

Anbei eine Abbildung 19 mit 17 Würfeln.

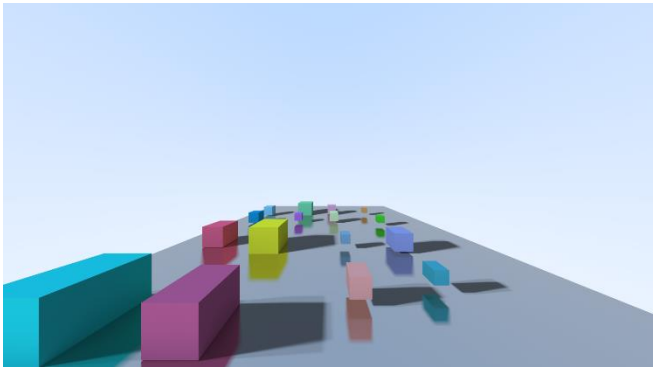

Abbildung 19 Beispielbild 1600x900, 1000 Strahlen pro Pixel

Referenzen

[1] Jag Mohan Singh and PJ Narayanan. *Real-time ray tracing of implicit surfaces on the gpu.* IEEE transactions on visualization and computer graphics, 16(2):261–272, 2010.

[2] Martin Zlatuˇska and Vlastimil Havran. *Ray tracing on a gpu with cuda–comparative study of three algorithms.* 2010.

[3] Roger Allen, *Accelerated Ray Tracing in One Weekend in CUDA*. Nov 05, 2018

[4] Amy Williams et al. *An Efficient and Robust Ray–Box Intersection Algorithm.* 2004.

[5] https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection (03.07.2021)

[6] http://www.songho.ca/opengl/gl_pbo.html (03.07.2021)

[7] https://moodle.htwg-konstanz.de/moodle/pluginfile.php/355753/mod_resource/content/2/03-Parallelisierung (03.07.2021)

[8] http://www.pxleyes.com/blog/2010/03/5-things-you-need-to-know-about-raytracing/ (03.07.2021)

[9] https://stackoverflow.com/questions/23469180/different-occupancy-between-calculator-and-nvprof (03.07.2021)

[10] *Wodniok, D. Higher Performance Traversal and Construction of Tree-Based Raytracing Acceleration Structures* (09.06.2020 02:39)