UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Entwicklung eines GPGPU-basierten Ray Tracers

Bachelorarbeit

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

vorgelegt von
Daniel Müller

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

Koblenz, im Mai 2015

Erklärung

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

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

.....
(Ort, Datum) (Unterschrift)

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

UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Bachelorarbeit
Daniel Müller
(Mat. Nr. 212 100 331)

Thema: Entwicklung eines GPGPU-basierten Ray-Tracers

Das Ray-Tracing bildet die Grundlage der photorealistischen Computergrafik und ermöglicht das Rendering nahezu real-erscheinender Szenen, wie sie heute beispielsweise in Simulationen oder Werbungen zum Einsatz kommen. Dabei simuliert ein Ray-Tracer das physikalische Verhalten einzelner Lichtstrahlen, wodurch eine realitätsnähere Umsetzung der Szenen-Beleuchtung erzielt werden kann als bei gängigen Scanline-Rendering-Verfahren.

Bisher existieren noch keine speziell für die Ausführung eines Ray-Tracers entwickelten Hardwarebausteine, wie die Graphics Processing Unit (GPU) für das Scanline-Rendering, so dass der Hauptprozessor das Rendering beim Ray-Tracing ausführt. Der Hauptprozessor arbeitet dabei aber größtenteils sequentiell, wodurch es in der Regel beim Bildaufbau zu weniger Bildern pro Sekunde kommt als bei der Abarbeitung durch massiv parallel arbeitende Grafikhardware. Nun sind neue GPU-Modelle heute in der Lage auch allgemeinen Programmcode, der kein Bestandteil der theoretischen Rendering-Pipeline darstellt, mit ihrer Vielzahl an Prozessorkernen parallel auszuführen – auch GPGPU (General Purpose Computation on Graphics Processing Unit) genannt.

Ziel der Arbeit ist die Entwicklung eines Ray Tracers, der möglichst vollständig auf der GPU ausgeführt wird, so dass maximale Parallelisierbarkeit des Programmcodes erreicht wird. Hierzu soll die aktuelle Version 4.5 der Graphik-API OpenGL verwendet werden und mit Hilfe von Compute-Shadern eine GPGPU-Umsetzung des Ray-Tracings erfolgen. Weiterhin ist die Umsetzbarkeit des Aufbaus und der Traversierung eines Uniform Grids auf der GPU zur Beschleunigung des Ray-Tracings zu prüfen und die Integration der Datenstruktur in den GPGPU-Ray-Tracer vorzunehmen.

Durch die Erweiterung des Ray-Tracers mit dieser beschleunigenden Datenstruktur ist eine Steigerung der Performanz zu erwarten. Das Ergebnis dieser Arbeit soll hinsichtlich des Leistungszuwachses evaluiert und interpretiert werden.

Die inhaltlichen Schwerpunkte der Arbeit sind:

1. Recherche und Analysis bestehender sequentieller Ray-Tracer
2. Konzeption eines eigenen parallelisierten GPGPU-Ansatzes
3. Implementierung
4. Erstellung von Benchmarks
5. Evaluation und Interpretation der Ergebnisse
6. Dokumentation der Ergebnisse

Koblenz, 21.10.2014

– Daniel Müller –

– Prof. Dr. Stefan Müller –

Abstract

Ray Tracing enables a close to reality rendering implementation of a modelled scene. Because of its functioning, it is able to display optical phenomena and complex lighting. Though, numerous computations per pixel have to be done. In practice implementations can not achieve computer graphics' aim of real-time rendering close to 60 frames per second.

Current *Graphics Processing Units (GPU)* allows high execution parallelism of general-purpose computations. By using the graphics-API *OpenGL* this parallelism can be achieved and it is possible to design and realize a Ray-Tracer, which operates entirely on the GPU. The developed approach will be extended by an Uniform Grid – a Ray-Tracing acceleration structure. Hence, a speed-up is expected.

This thesis' purpose is the implementation of Ray-Tracer, which operates completely on the GPU, and its expansion by integrating an Uniform Grid. Afterwards, the evaluation of maximum achievable performance takes place. Possible problems regarding GPU-programming will be identified and analysed.

Zusammenfassung

Das Rendering-Verfahren des *Ray-Tracings* ermöglicht die realitätsnahe Umsetzung der Bildgenerierung einer modellierten Szene und ist aufgrund seiner Arbeitsweise in der Lage optische Phänomene und komplexe Beleuchtungsszenarien darzustellen. Allerdings bedarf es bei der Bilderzeugung einer enormen Anzahl an Berechnungen pro Pixel, wodurch Realisierungen eines Ray-Tracers in der Praxis Ergebnisse erzielen, die weit unter der in der Computergraphik angestrebten Echzeitdarstellung von 60 Bildern pro Sekunde entfernt liegen.

Aktuelle Modelle der *Graphics Processing Unit (GPU)* ermöglichen die hochgradige Parallelisierung der Ausführung von allgemeinen Berechnungen. Mit Hilfe der Graphik-API *OpenGL* wird diese Parallelisierung nutzbar gemacht und ein vollständig auf der GPU ausgeführter Ray-Tracer entworfen und realisiert. Der entwickelte Ansatz wird durch die Integration eines *Uniform Grids* – einer beschleunigenden Datenstruktur des Ray-Tracings – erweitert, woraus eine Steigerung der Performanz zu erwarten ist.

Ziel dieser Arbeit ist die Implementierung eines auf der GPU ausgeführten Ray-Tracers und die Erweiterung des Ansatzes durch die Verwendung eines Uniform Grids. Die Ermittlung der erzielbaren Leistung wird im Anschluss durchgeführt. Bei der Entwicklung und Implementierung werden mögliche Probleme bei der Umsetzung bezüglich der GPU-Programmierung aufgezeigt und analysiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

List of Algorithms

1 Einleitung

In der Werbung fährt ein neues Modell eines namhaften Autoherstellers über einen zugefrorenen See, umgeben von einer mächtigen, schneebedeckten Bergkette. Die Sonne steht optimal am Horizont für einen eindrucksvollen Lichteinfall und die Kameraführung fängt die Szenerie genau im richtigen Moment ein. Ein beeindruckendes Gesamtbild – aber ist dies auch real?

Im Alltag begegnen uns die Ergebnisse von Ray-Tracern häufig, ohne dass wir sie unbedingt als solche wahrnehmen. In Wahrheit werden solche Szenen im Computer modelliert und gerendert – dabei existiert solch ein See in den Bergen eventuell nicht. Damit sich aus dem Modell aber tatsächlich ein real-wirkendes Video erzeugen lässt, wird eine mächtige Rendering-Umgebung benötigt. Die mögliche Basis dafür bietet das Verfahren des Ray-Tracings. Die Vorgehensweise beim Rendering durch Ray-Tracing ist der Verfolgung von Sehstrahlen nachempfunden, wodurch physikalische Einflüsse im Vergleich zu anderen Verfahren realitätsnah nachgebildet und umgesetzt werden können.

Aus der Annäherung an die Realität folgt eine vergleichsweise große Menge an auszuführenden Berechnungen im Vergleich zu stärker abstrahierten Rendering-Verfahren, die häufig zur Minimierung der benötigten Rechenleistung genutzt werden, um ein flüssiges Ausgabebild zu erzeugen. Die Aufteilung des Ray-Tracings in einzelne Arbeitsschritte ermöglicht den Entwurf separater Programme zur Hintereinander-Ausführung auf der GPU, so dass die Ausführung möglichst stark parallelisiert werden kann. Durch die Parallelisierung der Rechenschritte wird in der hier behandelten Umsetzung eine möglichst hohe Bildrate bei der Generierung des Ausgabebildes erstrebt.

Durch die Erweiterung des Verfahrens mit Hilfe der Datenstruktur eines Uniform Grids wird der darzustellende Raum gleichmäßig aufgeteilt und es können benötigte Berechnungen eingespart werden. Die tatsächliche Beschleunigung im Vergleich zu der unbeschleunigten Ray-Tracer-Variante soll durch Benchmarks bestimmt und beide Ergebnisse verglichen werden, um eine möglichst optimale Konfiguration zur Darstellung einer Szene zu finden.

2 Grundlagen

Der erste Abschnitt 2.1 dieses Kapitels thematisiert die Grundlagen der Bildgeneration mit Hilfe des Ray-Tracing Rendering-Verfahrens und basiert auf dem Grundlagen-Artikel[1] von Brian J. Ross. Der darauf folgende Abschnitt 2.2 beschäftigt sich mit der Entwicklung der Grafik-Hardware und der Umsetzbarkeit des Ray-Tracings auf heutigen Geräten. Abschließend werden die Grundlagen für eine Nutzung aktueller Hardware durch die Graphik-API OpenGL vorgestellt.

2.1 Ray-Tracing

Das Rendering-Verfahren des Ray-Tracings (*dt. Strahlenverfolgung*) basiert auf der Abtastung einer darzustellenden Szene durch Strahlen. Dabei werden von der Position einer in der Szene befindlichen Kamera aus Sehstrahlen in Blickrichtung erzeugt, die auf die Szene-Objekte treffen. Das Auftreffen eines Strahls auf eine Oberfläche eines Objekts entspricht dem Schnittpunkt des Strahls mit der Ebene eines der Dreiecke aus denen das Objekt aufgebaut ist. An der Position des Schnittpunktes wird der entsprechende Farbwert der Oberfläche abgefragt und in das Ausgabebild eingetragen (siehe Abbildung 1). Im Folgenden werden Strahlen betrachtet, die durch einen Stütz- und einen Richtungsvektor vollständig beschrieben werden. Hierbei bildet der Stützvektor den Strahlursprung. Das realitätsnahe Vor-

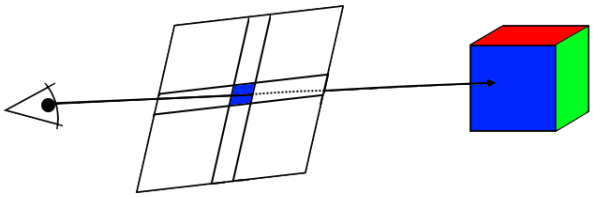


Abbildung 1: schematische Darstellung der Projektion eines Sehstrahltreffers auf ein Pixel des Ausgabebildes

gehen der Strahlenverfolgung ermöglicht die Erzeugung der folgenden optischen Phänome.

- Reflexionen
- Transparenzen
- Transmissionen
- direkte Beleuchtung
- globale Beleuchtung

Im Falle einer transparenten oder reflektierenden Oberfläche werden *Sekundärstrahlen* erzeugt, die wie die primären Sehstrahlen verfolgt werden. Bei einer Spiegelung wird die Normale durch Interpolation der Normalen der Dreieckseckpunkte bestimmt. Die Richtung des reflektierten Sehstrahls entspricht dem an dieser Normalen reflektierten Richtungsvektor des primären Sehstrahls. Handelt es sich um eine transparente Oberfläche entspricht der Strahlursprung des Sekundärstrahls den Schnittpunktkoordinaten und die Richtung des Sehstrahls wird übernommen.

Durch die Angabe von materialspezifischen Brechungsindizes ist es außerdem möglich Sekundärstrahlen für die Brechung von Licht – der Transmission – an dem getroffenen Material zu generieren. Nach der vollständigen Verfolgung der Seh- und Sekundärstrahlen werden die Farbwerte der Oberflächen-Schnittpunkte mit der Szene gewichtet aufaddiert und ein Ausgabebild entsteht.

Für die Bestimmung der direkten Beleuchtung eines Schnittpunktes durch eine Lichtquelle werden *Schattenfühler* erzeugt. Dabei handelt es sich um die Verbindung der Schnittpunktkoordinate mit der Position der Lichtquelle durch einen weiteren Strahl. Eine Oberfläche ist genau dann direkt beleuchtet, wenn der Schattenfühler außer in seinem Ursprung kein Dreieck der Szene schneidet. Die Auswertung der Schattenfühler für Schnittpunkte aller Seh- und Sekundärstrahlen ermöglicht die Anwendung der direkten Beleuchtung, wo sie zutrifft. Abbildung 2 illustriert die aus einem Sehstrahl resultierenden Sekundärstrahlen und den Schattenfühler auf den Schnittpunkt dieses Strahls. Es ist außerdem möglich ein globales Beleuchtungs-

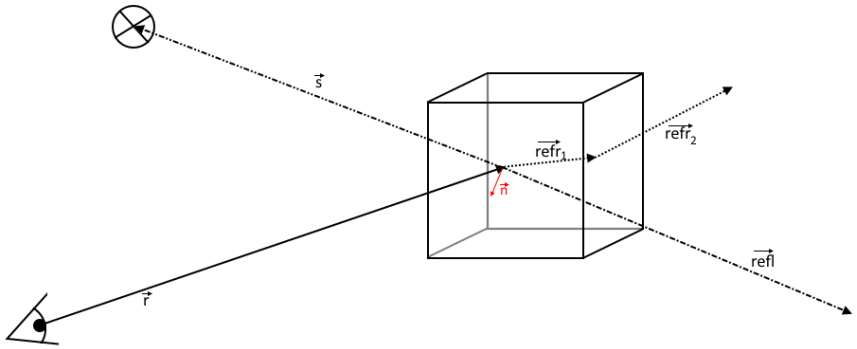


Abbildung 2: Erzeugung eines Schattenfühlers $\vec{s}$ auf den Schnittpunkt eines Sehstrahls $\vec{r}$ und der Sekundärstrahlen beim Treffen eines zum Teil durchsichtigen und reflektierenden Objekts; Schattenfühler für Sekundärstrahl-Treffer wurden ausgelassen

modell zu integrieren, in dem man weitere vom Schnittpunkt ausgehende Strahlen generiert. Mit diesen Strahlen werden die umliegenden Oberflä-

chen abgetastet und die indirekte Beleuchtung des Schnittpunkts durch Auswertung der Helligkeit der benachbarten Dreiecksflächen bestimmt. Allerdings kommt es im Rahmen dieser Ausarbeitung nicht zur Anwendung der Transmissionen und der globalen Beleuchtung beim Rendering der Szenen. Des Weiteren werden die Transparenzen als spezieller Fall einer Reflexion ohne Veränderung der Strahlrichtung behandelt, so dass eine explizite Unterscheidung der beiden Phänomene nicht erfolgen muss.

Zur Bestimmung eines Schnittpunkts eines Strahls mit einer Dreiecksfläche der Szene wird der Algorithmus 1 nach Purcell et al. [2] verwendet. Bei der Anwendung des Algorithmus erhält man neben der Information, dass eine Dreiecksfläche von einem Strahl geschnitten wird, auch die für die Interpolation zwischen den Eckpunkten benötigten Parameter u und v zur Bildung einer baryzentrischen Koordinate. Der aufgezeigte Algo-

Algorithmus 1 Bestimmung des Schnittpunkts eines Strahls, dargestellt durch Stützvektor $\vec{ro}$ und Richtungsvektor $\vec{rd}$, mit der durch die Eckpunkte $\vec{v0}, \vec{v1}, \vec{v2}$ aufgespannte Fläche eines Dreiecks

$\vec{edge1} = \vec{v1} - \vec{v0}$
$\vec{edge2} = \vec{v2} - \vec{v0}$
$\vec{pvec} = \vec{rd} \times \vec{edge2}$
$det = \vec{edge1} \cdot \vec{pvec}$
$\vec{tvec} = \vec{ro} - \vec{v0}$
$u = det^{-1} \cdot (\vec{tvec} \times \vec{pvec})$
$\vec{qvec} = \vec{tvec} \times \vec{pvec}$
$v = det^{-1} \cdot (\vec{rd} \cdot \vec{qvec})$
$t = det^{-1} \cdot (\vec{edge2} \cdot \vec{qvec})$
if $((u \geq 0) \wedge (v \geq 0) \wedge (u + v \leq 1) \wedge (t > 0))$ **then**
  // Schnittpunkt liegt in Dreiecksfläche mit Koordinate
  // $\vec{v0} + u \cdot \vec{edge1} + v \cdot \vec{edge2}$
  // und Strahlparameter t
end if

rithmus verdeutlicht die Anzahl der benötigten Rechenoperationen für den Test eines Strahls an einem Dreieck. Um für einen Sehstrahl eines Pixels des Ausgabebildes zu prüfen, ob er ein Objekt der Szene schneidet, muss dieser auf einen Schnittpunkt mit allen Dreiecksflächen der gesamten Szene getestet werden. Außerdem entstehen beim Auftreffen eines Sehstrahls die bekannten Sekundärstrahlen, die ebenfalls diesen Tests unterzogen werden müssen. Zur Minimierung der Menge an potentiell von einem Strahl geschnittenen Dreiecke existieren beschleunigende Datenstrukturen, die zur Steigerung der Performanz eines Ray-Tracers integriert werden können. Der folgende Unterabschnitt 2.1.1 stellt das *Uniform Grid* als eine solche Ray-Tracing-Beschleunigung vor, um es schließlich in dem entwickelten

Ray-Tracer zu verwenden.

Aber nicht nur die Anzahl an benötigten Berechnungen pro Aufbau des Ausgabebildes erschwert die echtzeitfähige Umsetzung eines Ray-Tracers. Bisher existiert keine massentaugliche, speziell für das Ray-Tracing entworfene Hardware-Komponente für das Ausführen der erforderlichen Schnittpunkt-Berechnungen. Abschnitt 2.2 zeigt die Entwicklung der Grafik-Hardware auf und beschreibt eine daraus resultierende Möglichkeit der Nutzbarmachung dieser Hardware für das Ray-Tracing.

2.1.1 Uniform Grid

Dieser Abschnitt stellt das Uniform Grid als beschleunigende Datenstruktur für das Ray-Tracing vor und basiert dabei auf dem Artikel „Geometric computing and uniform grid technique“ von Akman et al. [3].
Bei dem Uniform Grid handelt sich um ein uniformes Gitter, welches über die darzustellende Szene gelegt wird und so eine räumliche Anordnung der darin befindlichen Objekte ermöglicht. Ein $N \times N \times N$ Raster des Uniform Grids im dreidimensionalen Raum besteht aus N^3 würfelförmigen Gitter-Zellen (im Folgenden als *Voxel* bezeichnet), deren Seiten parallel zu den Koordinatenachsen verlaufen. Bei der Überdeckung der Szene mit dem Voxel-Gitter kommt es zu keiner Überlappung dieser Würfel und der Szenenraum wird vollständig abgedeckt.
Die Einordnung der Szene-Objekte in die Voxel, in denen sie sich befinden, bildet die Grundlage für den Umgang mit dieser Datenstruktur. Nach Aufstellen einer Inhaltsrelation aller Objekt-Dreiecke zu den Voxeln des Uniform-Grids ist es möglich die Schnittpunkt-Tests des Ray-Tracings auf eine Teilmenge aller Voxelinhalte zu reduzieren. Hierfür werden jene Voxel bestimmt, die ein Strahl beim Durchqueren der Szene schneidet. Die Menge der auf Schnittpunkte zu testenden Dreiecke ergibt sich aus der Vereinung der Inhaltsmengen aller von diesem Strahl geschnittenen Voxel (Abbildung 3).

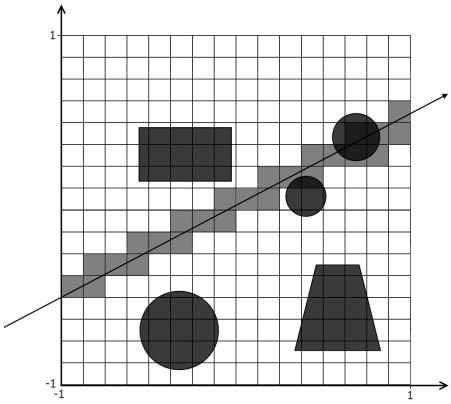


Abbildung 3: Aufteilung einer 2D-Szene in ein 16×16 Uniform Grid mit markierten Voxeln, durch die ein Strahl verläuft

2.2 General-Purpose GPU

Als *General-Purpose Graphics-Processing-Unit (GPGPU)* wird die Nutzung der *Graphics-Processing-Unit (GPU)* für Berechnungen allgemeiner Natur bezeichnet, die nicht notwendigerweise der Erzeugung eines Ausgabebildes dienen. Ansätze, die GPU für allgemeine Berechnungen nutzbar zu machen, existieren seitdem dieser Koprozessor schrittweise programmierbar wurde und schließlich als GPU bezeichnet wurde. Anhand der Generationen des *Software Developer Kits DirectX (DX)* von Microsoft haben Wu und Liu[4] die Entwicklung der GPU hin zur GPGPU kategorisiert. Dieser Abschnitt bietet einen Überblick über diese Entwicklung und erläutert die daraus resultierenden Möglichkeiten zur Implementierung eines GPGPU-Ray-Tracers.

Zur Minderung der Last eines Hauptprozessors bei der Ausgabe generierter Bilder wurde anfangs ein Koprozessor entwickelt, so dass dieser die Übertragung des Bildes an einen Bildschirm steuerte. Erst Mitte der neunziger Jahre wurde dieser um rudimentäre Fähigkeiten zur Verarbeitung von Transformationen und der Beleuchtung der zu erstellenden Grafiken erweitert und folglich als GPU bezeichnet. Erst später erfolgte der Beginn der Programmierbarkeit der Verarbeitungsstufen und eine schrittweise stattfindende Erweiterung dieser Programmierbarkeit.

2001 DX8 GPUs wie die *GeForce 3* von NVIDIA ermöglichen die Ausführung von Vertex-Shader-Programmen auf der GPU

2002 mit DX9 wird der Fragment-Shader programmierbar und die Vertex-Programmierbarkeit erweitert

2006 DX10 unterstützt die Nutzung eines Typs von Shader-Prozessor, der sowohl Vertex- als auch Fragment-spezifische Berechnungen ausfüh-

ren kann. Außerdem wird der *Geometry Shader* als weitere Verarbeitungsstufe innerhalb des Renderings ermöglicht

2006 NVIDIA veröffentlicht die *Compute Unified Device Architecture (CUDA)*

2008 Die Spezifikation von *OpenCL 1.0* [5] wird von der Khronos Group veröffentlicht

2012 OpenGL wird in Version 4.3 um *Compute Shader* erweitert [6]

Aus der etappennweisen Abarbeitung der Berechnungen, die die Shader-Einheiten bis zum finalen Ausgabebild vollziehen, resultiert eine logische Reihenfolge von Verarbeitungsstufen, die *Rendering-Pipeline* genannt wird. Für die Programmierung der GPU ist die Kenntnis der Rendering-Pipeline unerlässlich. Um auch Nicht-Grafik-Programmierern das Ausführen von Code auf der GPU zu erleichtern, wurden Programmier-Schnittstellen wie CUDA und OpenCL entwickelt. Erst später ermöglichte die Graphik-API OpenGL die Ausführung von *general purpose*-Programmen – auch *Kernel* genannt – mit der Erweiterung um Compute Shader.
Das Rendering durch Ray-Tracing wird durch die Hardware der GPU standardmäßig nicht unterstützt. Mit Hilfe der entstandenen Ausführbarkeit von Kerneln durch die GPU-Shader-Einheiten lässt sich die Hardware für die benötigten Schnittpunkt-Berechnungen nutzen. Da es sich bei einem GPGPU-Ray-Tracer um eine Grafikapplikation handelt, wird OpenGL als Grundlage zur Entwicklung gewählt. Dabei wird versucht bestehende Funktionalitäten des Vertex- und Fragment-Shaders zu nutzen und durch Compute-Shader-Kernel einen Ray-Tracer umzusetzen. Einen Einblick in die benötigten OpenGL Grundlagen gewährt der folgende Abschnitt 2.3.

2.3 GPGPU unter OpenGL

Dieser Abschnitt beinhaltet die Grundlagen der GPGPU-Programmierung unter OpenGL 4.5 und basiert auf der offiziellen OpenGL-Internetpräsenz der Khronos Group [7].

Hierzu werden in Abschnitt 2.3.1 Compute-Shader [8] und die Ausführung von Kerneln thematisiert. Der darauf folgende Abschnitt 2.3.2 zeigt die verwendeten GPU-Datenstrukturen auf und erläutert die verwendeten OpenGL-Befehle, die für ihre Erstellung notwendig sind.

2.3.1 Compute-Shader

Anders als Vertex- oder Fragment-Shader sind Compute-Shader kein Bestandteil der von OpenGL verwendeten Rendering-Pipeline. Die Ausführung eines Compute-Shader-Programms – auch *dispatch* genannt – kann an einer beliebigen Stelle im Programmcode erfolgen. Dabei wird die Anzahl

der tatsächlichen Programmaufrufe vom Benutzer festgelegt. Es findet eine Gliederung der Aufrufe in *Arbeitsgruppen* statt, wobei jeder Thread einer *globalen* und einer *lokalen* Arbeitsgruppe zugeordnet ist. Mit Hilfe der Position innerhalb der lokalen Arbeitsgruppe und mit Kenntnis der zugehörigen Arbeitsgruppen ist es möglich einen Thread in der Gesamtheit aller Programmausführungen – dem *compute space* – zu lokalisieren. Diese Positionsabfrage innerhalb des *compute space* ermöglicht eine Zuweisung der einzelnen Aufrufe auf unterschiedliche Aufgabenbereiche wie beispielsweise das Beschreiben eines gewissen Abschnitts eines Datenfeldes oder der Zugriff auf einen Pixel einer Textur. Die Methoden-Signatur des *dispatch*-Befehls in OpenGL zeigt Listing 1. Der *compute space* wird dabei in drei Dimensionen aufgespannt, indem die Anzahl der globalen Arbeitsgruppen festgelegt wird.

```
1 void glDispatchCompute(
2       GLuint num_groups_x, // Anzahl der globalen Arbeitsgruppen
3         GLuint num_groups_y, // innerhalb einer Dimension
4         GLuint num_groups_z
5         );
```

Listing 1: OpenGL dispatch-Befehl zur Ausführung eines aktiven Compute-Shader-Programms

Die Anzahl der lokalen Arbeitsgruppen wird als Eingabewert des Shader-Programms definiert, wodurch sich erst an dieser Stelle die Gesamtzahl der Programm-Aufrufe ergibt (siehe Listing 3, Zeile 2). Des Weiteren handelt es sich bei der Anzahl der lokalen Arbeitsgruppen um die einzige Eingabe, die ein Compute-Shader-Programm verpflichtend erhält. Zusätzlich stehen dem Shader-Programm die in Listing 2 aufgeführten *built-in*-Variablen zur Verfügung, in denen die Position des einzelnen Aufrufs innerhalb des *compute space* gespeichert ist.

```
1 in uvec3 gl_NumWorkGroups;
2 in uvec3 gl_WorkGroupID;
3 in uvec3 gl_LocalInvocationID;
4 in uvec3 gl_GlobalInvocationID;
5 in uint  gl_LocalInvocationIndex;
```

Listing 2: *built-in*-Variablen, die einem Compute-Shader Thread zur Verfügung stehen.

Threads, die zur selben lokalen Arbeitsgruppe gehören, werden gleichzeitig auf der GPU ausgeführt und arbeiten parallel. Innerhalb eines solchen Thread-Verbands ist die Deklaration von Variablen als *shared* im geteilten Cache der Threads möglich. Allerdings werden für die Bewahrung der Speicher-Kohärenz bei paralleler Ausführung geeignete Synchronisationsbefehle wie das Errichten einer *barrier* innerhalb des Shader-Programms benötigt. Eine weitere Möglichkeit zur Wahrung der Speicher-Kohärenz

ist die Verwendung von *atomaren Operationen*. Ruft ein Thread eine atomare Operation auf, so erhält nur dieser Thread bis zur Beendigung der Operation Zugriff auf einen gewählten Speicherabschnitt. Ab Beginn der Ausführung ist sichergestellt, dass das zugrunde liegende Datum nicht von außen verändert wird. Versucht ein anderer Thread in diesem Moment auf das Datum zuzugreifen, wird gewartet bis das Datum wieder freigegeben ist. Im Rahmen dieser Ausarbeitung findet eine Synchronisation der Speicherzugriffe einer lokalen Arbeitsgruppe durch Verwendung von atomaren Operationen auf unterstützten Datenstrukturen statt, die in Abschnitt 2.3.2 vorgestellt werden.
Benötigt ein Compute-Shader-Programm den Zugriff auf eine auf der GPU befindliche Datenstruktur, so muss dieser im Programmcode mit einer passenden Angabe zur Lokalisation definiert sein. Hierzu werden nummerierte *binding points* für unterschiedliche Datenstrukturen verwendet, die Listing 3 ab Zeile 5 aufzeigt.

```
1  // Eingabe der lokalen Arbeitsgruppengröße 32*32*1
2  layout(local_size_x = 32, local_size_y = 32, local_size_z = 1) in;
3  /* Zugriff über Buffer-binding point 0 auf Buffer-Objekt, das
4   * int-Datenfeld beinhaltet */
5  layout(std430, binding = 0) buffer SSBO_Example{
6   int example_array[];
7  };
8  /* Zugriff über Textur-binding point 0 auf Textur mit vier 32 Bit
9   * float-Farbkanälen */
10 layout(binding = 0, rgba32f) uniform image2D exampleTexture;
11
12 void main(){ /* Programmcode */ }
```

Listing 3: Beispiel für den Kopf eines Compute-Shader-Programms mit Eingabe-Größe der lokalen Arbeitsgruppe und Verweisen auf verfügbare Datenstrukturen

2.3.2 GPU-Datenstrukturen

In OpenGL werden verschiedene *OpenGL Objects* unterschieden, die einen *Container* für Datenstrukturen und deren Zustand bilden. Die folgenden Objekte lassen sich von der Klassifikation des *OpenGL Object* ableiten und werden im Rahmen dieser Ausarbeitung verwendet.

- *Buffer Objects*
  - *Vertex Buffer Objects (VBOs)*
  - *Shader Storage Buffer Objects (SSBOs)*
- Texturen
  - *Sampler Objects* (kurz: *Sampler*)

  - *Image formats*

- *Vertex Array Objects (VAOs)*

Buffer-Objekte beinhalten Datenfelder, die im GPU-Speicher allokiert werden und durch eine Referenz im *OpenGL unsigned integer*-Datentyp (*GLuint*) gelesen oder beschrieben werden können. Um ein Buffer-Objekt zu erzeugen wird der in Zeile 1 des Listings 4 aufgeführte Befehl *glGenBuffers* ausgeführt und die Referenz des Buffer-Objekts in den Hauptspeicher geschrieben. Operationen auf dem erzeugten Buffer-Objekt sind erst mit dem *binden*, dem aktiv Setzen, des Buffer-Objekts möglich (Zeile 6).

```
1 void glGenBuffers(
2  GLsizei n,  // Anzahl der zu erzeugenden Buffer-Objekten
3  GLuint* buffers // Speicherabschnitt, in den Referenz des
4                  // erzeugten Buffer-Objekts geschrieben wird
5  );
6 void glBindBuffer(
7  enum target, // Buffer-Objekt-Typ
8  uint bufferName // Referenz auf Buffer-Objekt
9  );
```

Listing 4: Generierung und Aktivierung eines Buffer-Objekts

Die Bereitstellung der Vertex-Daten auf der GPU erfolgt über *VBOs* für jedes der Vertex-Attribute wie beispielsweise Position und Farbe. Hierzu wird zuerst GPU-Speicher allokiert und mit den Vertex-Daten aus dem Hauptspeicher gefüllt (siehe Listing 5, Zeile 1). Anschließend wird, wie in Zeile 8 beschrieben, festgelegt wie die Daten innerhalb des erzeugten Datenfeldes zu interpretieren sind.

```
1 void glBufferData(
2   GLenum target,  // Buffer-Objekt-Typ, der erzeugt werden soll
3   GLsizeiptr size,  // Größe des zu allokierenden Speicherbereichs
4   const GLvoid* data, // Pointer auf zu kopierende Daten im
5                       // Speicher des Hauptrechners
6   GLenum usage  // Benutzungsmuster
7   );
8 void glVertexAttribPointer(
9   GLuint index, // Positions-Index des Vertex-Attributs
10  GLint size, // Anzahl der Komponenten eines Datums in Datenfeld
11  GLenum type,  // Datentyp einer Komponente eines Datums
12  GLboolean normalized, // Normalisierung der Werte
13  GLsizei stride, // Byte-Abstand der Werte innerhalb Datenfeld
14  const GLvoid* pointer   // Referenz auf Speicherabschnitt aus dem
15                          // Daten kopiert werden
```

Listing 5: Allokation des GPU-Speichers für die Erstellung eines VBOs und Anweisung zur Interpretation der enthaltenen Daten

Mit Hilfe eines *Vertex Array Objects* werden die VBOs der einzelnen Vertex-Attribute zusammengefasst und dem Vertex-Shader bereit gestellt.

Zur Bereitstellung der Vertex-Daten für die Compute-Shader und den Aufbau der Datenstruktur für die Realisierung des Uniform Grids werden *Shader Storage Buffer Objects* verwendet. Die Speichergröße eines SSBO beträgt mindestens 16MB und ist nur durch die Größe des verfügbaren GPU-Speichers begrenzt. Die Größe des SSBO kann zur Laufzeit über die *length*-Funktion abgefragt werden. Um einen unveränderbaren Speicherabschnitt für ein SSBO zu allokieren, verwendet man den in Listing 6 aufgeführten OpenGL-Befehl.

```
void glBufferStorage(
 GLenum target,
 GLsizeiptr size,
 const GLvoid* data,
 GLbitfield flags // Art, wie Zugriffe auf Buffer-Objekt erfolgen
 );
```

Listing 6: Allokation von *immutable storage* eines Buffer-Objekts (Übergabewerte gleichen größtenteils der Beschreibung von *glBufferData* aus Listing 5)

Des Weiteren sind atomare Operationen auf Daten eines SSBO möglich, die zur Synchronisation der Zugriffe auf das beinhaltete Datenfeld durch parallel arbeitende Threads verwendet werden können. Listing 7 zeigt Beispiele für verwendete atomare Operationen auf SSBOs. Allerdings ist die Verwendung der Operationen in der aktuellen OpenGL Version 4.5 auf die Datentypen *integer* und *unsigned integer* beschränkt.

```
/* gibt Wert in mem vor Ausführung zurück,
 * schreibt Additionsergebnis von diesem Wert und data in mem */
int atomicAdd(
 inout int mem,
 int data
 );
/* gibt Wert in mem vor Ausführung zurück,
 * wenn Wert in mem dem Wert compare entspricht wird data
 * in mem geschrieben */
int atomicCompSwap(
 inout int mem,
 int compare,
 int data
 );
/* gibt Wert in mem vor Ausführung zurück,
 * wenn data kleiner als Wert in mem, wird data in mem
 * geschrieben */
int atomicMin(
 inout int mem,
 int data
 );
```

Listing 7: Atomare Operationen, die auf *int* oder *uint* (*unsigned integer*) SSBO-Inhalten ausgeführt werden können

Die effiziente Initialisierung aller Daten eines SSBOs innerhalb jedes *rendering frames* kann durch die Ausführung des Befehls *glClearBufferData* realisiert werden (siehe Listing 8).

```
1 glClearBufferData(
2  GL_SHADER_STORAGE_BUFFER, // Typ des Buffer-Objekts
3  GL_R32I, // internes Format für Speicherzugriffe in Buffer-Objekt
4  GL_RED, // Anzahl der Komponenten einer Position des Datenfeldes
5  GL_INT, // Datentyp einer Position des Datenfeldes
6  &data  // Referenz auf in Buffer-Objekt zu kopierende Daten
7  );
```

Listing 8: Initialisierung eines SSBOs mit Hilfe eines referenzierten Speicherabschnitts, dessen Inhalt in das Datenfeld des SSBO kopiert wird

Die einzelnen Shader-Programme, die zur Realisierung des GPGPU-Ray-Tracers verwendet werden, arbeiten zum größten Teil auf Pixeldaten von Bildern innerhalb verschiedener Texturen. Dabei besitzen die verwendeten Texturen die gleiche Auflösung wie die Ausgabetextur, in der das finale Bild des Renderings gespeichert wird. Der Inhalt eines Pixels einer Textur korreliert dabei mit den Inhalten der gleichen Pixelkoordinate der anderen Texturen (siehe Abbildung 4). Texture-Objekte beinhalten ein oder

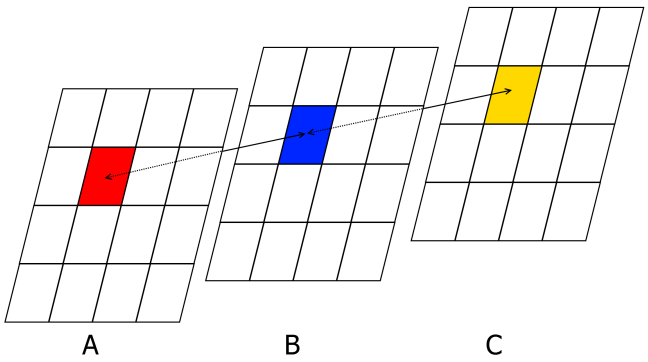


Abbildung 4: Beispiel für die Korrelation der Pixelkoordinaten der verwendeten Texturen. Textur A beinhaltet eine Koordinate $(1, 0, 0)$, Textur B einen Richtungsvektor $(0, 0, 1)$ und Textur C eine dazu gehörige Normale $(0, 1, 0)$

mehrere Bilder des gleichen Bildformates. Ein Bild mit der Auflösung $h \times w$ entspricht einem zweidimensionalen Datenfeld *image[w][h]*, bei dem jede Position des Datenfelds einem Pixel des Bildes entspricht. Die Daten eines Pixels sind durch das Format der Textur gegeben. So beinhaltet eine $h \times w$ Textur im *RGBA32F*-Format Bilder mit $h \cdot w$ Pixeln mit je vier Farbkanälen (rot, gelb, blau, alpha) à 32 Bit *float*-Werten.
Die notwendigen Aufrufe zur Erstellung eines Textur-Objekts auf der GPU zeigt Listing 9. Äquivalent zu der Erzeugung eines Buffer-Objekts generiert der Aufruf *glGenTextures* (Zeile 1) ein Texture-Objekt auf der GPU und

speichert eine GPU-Referenz in einem referenzierten Speicherabschnitt im Hauptspeicher. Anschließend wird eine *texture image unit* der GPU als aktiv gesetzt und an das Texture-Objekt gebunden (Zeile 5 und 8). Die Allokation eines unveränderlichen Speicherabschnitts auf der GPU erfolgt mit der Ausführung des Befehls *glTexStorage2D* und der Eingabe des Texturtyps, des Bildformats und der Auflösung. Zuletzt wird ein Bild an eine *image unit* gebunden, die zu dem Texture-Objekt gehört.

```
1  glGenTextures(
2    1, // Anzahl der zu erzeugenden Texture-Objekte
3    &exampleTexHandle // Speicherziel der GPU-Referenz
4    );
5  glActiveTexture(
6    GL_TEXTURE1 // texture unit, die aktiv gesetzt wird
7    );
8  glBindTexture(
9    GL_TEXTURE_2D, // Spezifizierung des Typs des Texture-Objekts
10   exampleTexHandle // Referenz des Texture-Objekts
11   );
12 glTexStorage2D(
13   GL_TEXTURE_2D, // Textur-Typ für den Speicher allokiert wird
14   1, // Anzahl der anzulegenden Mipmaps
15   GL_RGBA32F, // internes Pixel-Daten-Format
16   512, // Breite der Textur
17   512 // Höhe der Textur
18   );
19 glBindImageTexture(
20   1, // Texture-Unit
21   exampleTexHandle, // Textur, die an image unit gebunden wird
22   0, // Tiefe des verwendeten Mipmaps
23   GL_FALSE, // besteht Textur aus mehreren Schichten?
24   0, // Schicht der Textur, die an image unit gebunden wird
25   GL_READ_WRITE, // Zugriffstyp
26   GL_RGBA32F // Format, wie Daten in Speicher interpretiert werden
27   );
```

Listing 9: Beispiel für die Erstellung eines Texture-Objekts

Lade- und Schreiboperationen eines Compute-Shader-Aufrufs auf den Bildern der Texture-Objekte erfolgen über die *imageLoad* und *imageStore*-Funktionen, die in Listing 10 erläutert werden. Die Befehle greifen jeweils auf eine Pixelkoordinate des Bildes zu. Die Ausgabe des Bildes der Ausgabetextur erfolgt im Fragment-Shader über den lesenden Zugriff auf ein Sampler-Objekt. Der *Sampler* wird implizit beim Erzeugen des Texture-Objekts erzeugt und bei Bedarf an die *texture image unit* gebunden. Innerhalb des Shaders wird mit dem Befehl *texture(sampler, coords)* auf eine Variable vom Typ *sampler2D* zugegriffen. Die Lokalisation des *Samplers* erfolgt wie bei der auszugebenden Textur, so dass man den gleichen *binding point* verwendet.

```
/* Auflösung des Platzhalters gvec4 bzw. gimage2D:
 * Datentyp der Komponente eines Pixels
 * float              --> vec4; image2D
 * int                --> ivec4; iimage2D
 * uint               --> uvec4; uimage2D
 * Die Rückgabe ist unabhängig von der Anzahl der Kanäle der Textur
 * immer ein 4D-Vektor */

/* Rückgabe eines 4D-Vektors im Datenformat der Pixel-Daten */
gvec4 imageLoad(
 gimage2D image, // image unit aus der gelesen wird
 ivec2 position  // Pixel-Koordinate, die gelesen wird
 );
void imageStore(
 gimage2D image, // image unit in die geschrieben wird
 ivec2 position, // Pixel-Koordinate, in die geschrieben wird
 gvec4 data      // Datum im Datenformat eines Pixels des Bildes
 );
```

Listing 10: Lesender und schreibender Zugriff auf *image units*

3 Konzeption

Die Durchführung des Ray-Tracings in OpenGL erfolgt über den Aufruf verschiedener Compute-Shader Kernel, die in ihrer Gesamtheit das Rendering der Szene ausführen.
Der folgende Abschnitt 3.1 definiert den Begriff des Basis GPGPU-Ray-Tracer im Kontext dieser Ausarbeitung und es werden die für diesen Ansatz essentiellen Compute-Shader Kernel und die zugrunde liegende Datenverwaltung auf der GPU aufgezeigt. Die festgelegte Aufrufreihenfolge der Compute-Shader Kernel wird im Anschluss erläutert und der Rendering Ablauf hergeleitet.
Abschnitt 3.2 thematisiert die Erweiterung des Basis Ray-Tracers durch die Integration des Uniform Grids. Die für das Uniform Grid benötigte Datenstruktur wird in Unterabschnitt 3.2.1 erläutert. Für den Aufbau und die Traversierung der Datenstruktur ist eine effiziente Bestimmung der von einem Strahl geschnittenen Voxel notwendig. Die mathematischen Grundlagen und die zugrunde liegenden Algorithmen werden in Unterabschnitt 3.2.2 aufgezeigt. Um die Datenstruktur in den Basis Ray-Tracer zu integrieren bedarf es zwei weiteren Kernel-Klassen und einer Abwandlung der *intersection*-Kernel. Diese werden aus den voran gegangenen Unterabschnitten abgeleitet und in Unterabschnitt 3.2.3 vorgestellt. Als weitere Folge kommt es zu einer Erweiterung des Rendering-Ablaufs, der im letzten Teil 3.2.4 des Abschnitts behandelt wird.
Um den Umgang mit den einzelnen Instanzen zu vereinfachen wurde eine Objektstruktur für die Ausführung des GPGPU-Ray-Tracers in der Programmiersprache C++ entworfen, die in Abschnitt 3.3 vorgestellt wird.

3.1 Basis Ray-Tracing

3.1.1 Aufteilung der Compute-Shader Kernel

Das grundlegende Rendering der Basis-Variante des GPGPU-Ray-Tracers wird in dieser Ausarbeitung durch Umsetzung der folgenden Fähigkeiten definiert.

1. Erzeugen von Sehstrahlen
2. Testen von Strahlen auf Schnittpunkte mit der Szene
3. Erzeugen von Schattenfühlern für die direkte Beleuchtung und Verschattung
4. Erzeugen von reflektierten Sehstrahlen zur Darstellung von Reflexionen
5. Beleuchtung der dargestellten Objektoberfläche mit Hilfe eines Beleuchtungsmodells

Bei der Verfolgung eines Sehstrahls innerhalb der Szene ist es notwendig zwischen den Zuständen des Strahls zu unterscheiden, um eine sinngemäße Behandlung durch die Kernel zu gewährleisten, da es aufgrund der Darstellung von Reflexionen zur wiederholten Ausführung einzelner Shader-Programme kommt. Hierzu wird im Folgenden der Begriff des *rayState* als Zustandsattribut eines Sehstrahls eingeführt. Der *rayState* besitzt einen ganzzahligen Wert aus dem Intervall $[0, 2]$ und bestimmt ob ein Strahl weiter verfolgt wird. Bei der Erstellung eines Strahls wird der *rayState* mit dem Wert 1 initialisiert, wodurch er von folgenden Shader-Aufrufen als aktiv erkannt und verarbeitet wird. Verlässt ein Strahl die Szene oder erreicht er seinen Endpunkt innerhalb der Szene, so wird der Zustand auf 0 gesetzt und es findet keine weitere Verarbeitung durch die Kernel statt. Im Falle einer Reflexion wird der Zustand eines aktiven Strahls auf den Wert 2 erhöht und dieser an der Oberfläche des Schnittpunkts reflektiert. Kommt es zur erneuten Reflexion eines reflektierten Sehstrahls, besitzt dieser Sehstrahl vor der zweiten Reflexion im Zuge der stattgefundenen Bearbeitung erneut den Zustand 1 und der mehrfach-reflektierte Sehstrahl, der daraufhin erzeugt wird verfügt über den $rayState$ = 2.

Um ein Bild mittels Ray-Tracing zu erzeugen, werden die vier wie folgt definierten Klassen von Compute-Shader Kerneln unterschieden. Jeder Aufruf eines Shader-Programms verarbeitet dabei die Verfolgung eines Sehstrahls, der mit genau einem Pixel des gewünschten Ausgabebildes korrespondiert.

- **eye-ray-generation**
  Erzeugen von Sehstrahlen mit Hilfe der Kamera- und Ausgabebildinformationen
- **intersect1**
  Testen der aktiven Sehstrahlen auf Schnittpunkte mit der Szene
- **intersect2**
  Testen der Schattenfühler auf Schnittpunkte mit der Szene
- **shading**
  Anwendung eines ausgewählten Beleuchtungsmodells auf Schnittpunkt-Farbwerte

Beginnend mit dem *eye-ray-generation*-Kernel, der die benötigten Ausgabebild- und Kamerainformationen als Werte von uniform-Variablen von dem Hauptprogramm zur Verfügung gestellt bekommt, finden die weiteren Speicher- und Ladeoperationen der aufgezählten Shader-Programme zum größten Teil auf den auf der GPU befindlichen Texturen statt. Der Abschnitt 3.1.2 beschreibt den exakten Datenfluss der Shader-Ausführungen und den GPU-Datenstrukturen, so dass sich das aktuelle Kapitel auf die getätigten Berechnungen der Kernel ohne die Spezifizierung des Datenflusses

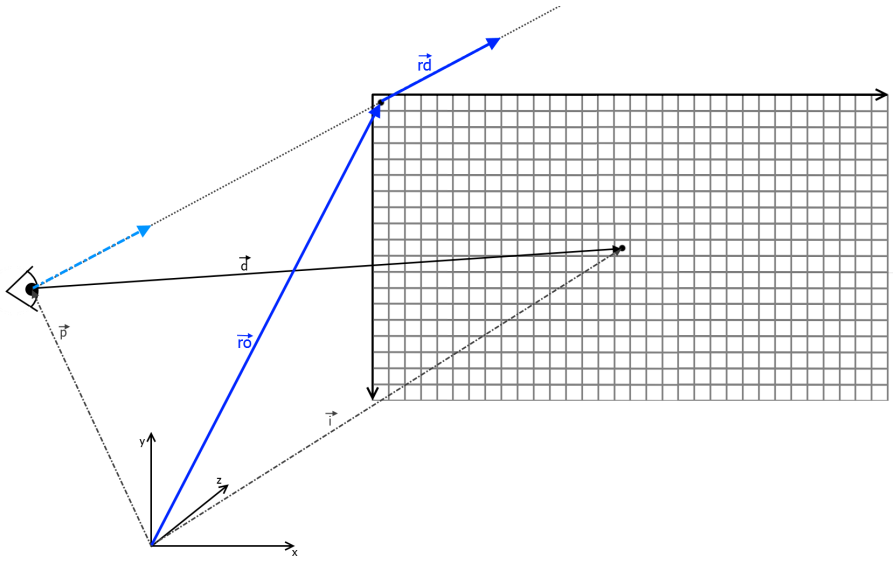


Abbildung 5: Projektion des Ausgabebildes auf imaginäre Ebene vor der Kamera

konzentriert.

Zu Beginn des Ray-Tracings erzeugen die Aufrufe des *eye-ray-generation*-Kernels die primären Sehstrahlen mit denen die Szene abgetastet wird. Mit Kenntnis der Koordinaten der Kameraposition wird das Ausgabebild auf eine gedachte Ebene im Szenenraum projeziert (siehe Abbildung 5). Den Ursprung eines Sehstrahls bildet die Koordinate des korrespondierenden Pixels der Projektion des Ausgabebildes auf diese imaginäre Ebene. Der *eye-ray-generation*-Aufruf speichert die Pixelkoordinate als Stützvektor des Sehstrahls ab. Die Verbindung aus Kameraposition und Pixelkoordinate wird normalisiert als Richtungsvektor des Sehstrahls gespeichert. Nach der Generierung der Sehstrahlen wird der Zustand des Strahls von dem Kernel als aktiv initialisiert.
Alle von dem *eye-ray-generation*-Kernel erzeugten aktiven Sehstrahlen werden anschließend von den Aufrufen des *intersect1*-Programms verarbeitet (siehe auch Abbildung 6). Ein *intersect1*-Aufruf testet dazu alle Dreiecke auf einen darin befindlichen Schnitt mit dem Sehstrahl und bestimmt den zur Kamera am nächsten gelegenen Schnittpunkt. Wird kein Schnittpunkt gefunden, verlässt der Strahl die Szene und als Farbwert wird die Hintergrundfarbe gespeichert. Des Weiteren wird der Zustand des Sehstrahls deaktiviert. Bei gefundenem Schnittpunkt bestimmt der Thread den Farbwert des in dem Dreieck befindlichen Punktes. Dazu wird zwischen den Vertex-Farben der Eckpunkte interpoliert, oder im Falle einer Texturierung des Dreiecks erfolgt der Texturzugriff mit Hilfe der interpolierten UV-Koordinate. Außerdem wird der interpolierte Normalenvektor auf

dem Schnittpunkt bestimmt und gespeichert. Durch das Auslesen des Reflexionsterms des Dreiecks wird dem bestimmten Farbwert ein Gewichtungsfaktor hinzugefügt, der bei der finalen Farbgebung durch den *shading*-Kernel verwendet wird. Kommt es zur Reflexion des Sehstrahls wird der Zustand inkrementiert, der Richtungsvektor an der Normalen des Schnittpunkts reflektiert und bildet mit der Schnittpunktkoordinate als Stützvektor einen reflektierten Sehstrahl. Reflektierte Sehstrahlen werden erst in einem erneuten Aufruf des Shader-Programms verarbeitet. Die detaillierte Reihenfolge der Kernel-Aufrufe wird in Abschnitt 3.1.3 thematisiert.

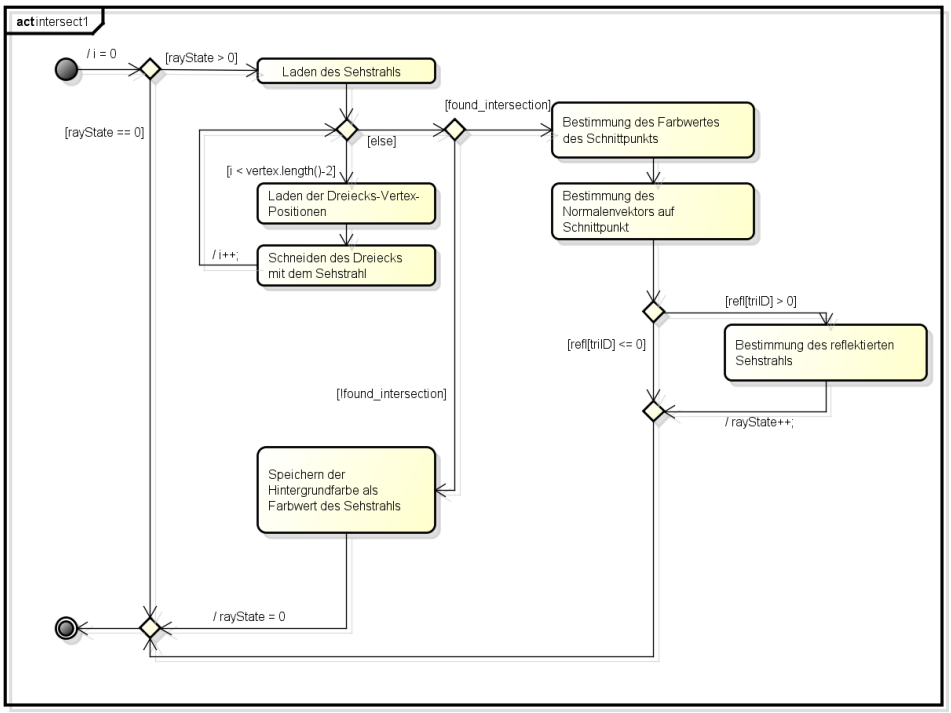


Abbildung 6: Aktivitätsdiagramm eines intersect1-Kernel-Aufrufs

Wurde ein Schnittpunkt gefunden, bestimmen die Aufrufe des *intersect2*-Kernels, ob es zu einer direkten Beleuchtung kommt, indem die Schattenfühler auf Schnittpunkte mit den Objekten der Szene getestet werden. Die Schnittpunktkoordinate bildet dabei den Strahlursprung und die Position der Lichtquelle den Endpunkt des Schattenfühlers. Der Strahl des Schattenfühlers wird ähnlich wie bei der Verarbeitung eines Sehstrahls mit allen Dreiecken der Szene geschnitten. Allerdings werden die Schnittpunkt-Tests beendet sobald ein Dreieck von dem Schattenfühler geschnitten wird. Es erfolgt das Setzen eines *flags*, um zu kennzeichnen, dass der Schnittpunkt im Schatten liegt. Wird bei vollständiger Iteration über alle Dreiecke kein Schnittpunkt gefunden, so wird der vorher gefundene Schnittpunkt

direkt beleuchtet und das *flag* nicht gesetzt. Die Ausführung des *intersect2*-Kernels erfolgt für jeden Sehstrahl pro Lichtquelle.
Die Ausführung des *shading*-Programms befüllt die Ausgabetextur mit den aus den getätigten Berechnungen resultierenden Farbwerten. Die in den vorherigen Kernel-Aufrufen bestimmten Farbwerte der Schnittpunkte werden durch Anwendung eines gewählten Beleuchtungsmodells und in Abhängigkeit von den gesetzten Schatten-Flags verarbeitet und beleuchtet. Am Beispiel des Phong-Shadings kommt es zum Zugriff auf die interpolierten Normalen, um den Lichteinfallswinkel zu bestimmen. Nach Eintrag des finalen Farbwertes in die Ausgabetextur wird der *rayState* dekrementiert. Bei weiterhin aktiven Sehstrahlen enthält die Ausgabetextur neben dem finalen Farbwert außerdem den Gewichtungsfaktor für die weitere Verarbeitung von reflektierten Sehstrahlen, um folgende Farbwerte auf zu addieren.

3.1.2 GPU-Datenverwaltung

Während der Ausführung der Compute-Shader Kernel werden Ergebnisse von getätigten Berechnungen innerhalb von verschiedenen Texturen gespeichert und von darauf folgenden Kernel-Aufrufen ausgelesen. Dabei werden die folgenden RGBA-Texturen und ihre Inhalte unterschieden.

- **rayOriginTexture**
  1) Schnittpunkt-Koordinaten oder Stützvektoren der Sehstrahlen als 3D-*float* Vektor in RGB-Kanälen
  2) rayState als float-Wert im jeweiligen Alpha-Kanal
- **rayDirTexture**
  1) Richtungsvektoren der Sehstrahlen als 3D-*float* Vektoren in RGB-Kanälen
  2) DreiecksID des zuletzt vom Sehstrahl geschnittenen Dreiecks im jeweiligen Alpha-Kanal
- **normalTexture**
  1) Normalenvektoren auf Schnittpunkte in RGB-Kanälen
- **colorBufferTexture**
  1) ermittelte Farbwerte der Schnittpunkte als 3D-*float* Vektor in RGB-Kanälen
  2) Gewichtungsfaktor als float-Wert im jeweiligen Alpha-Kanal
- **shadowTexture**

1) jeder Farbkanal korrespondiert mit einer Lichtquelle und beinhaltet einen float-Wert, der den Zustand des Schatten-Flags beinhaltet

- **finalTexture** (Ausgabetextur)

1) Farbwerte des Ausgabebildes als 3D-*float* Vektor in RGB-Kanälen
2) Gewichtungsfaktor als float-Wert im jeweiligen Alpha-Kanal

Abbildung 7 zeigt das Datenflussdiagramm zu den in Abschnitt 3.1.1 vorgestellten Compute-Shader-Programm Klassen und die aus ihrer Arbeitsweise resultierenden Texturzugriffe.

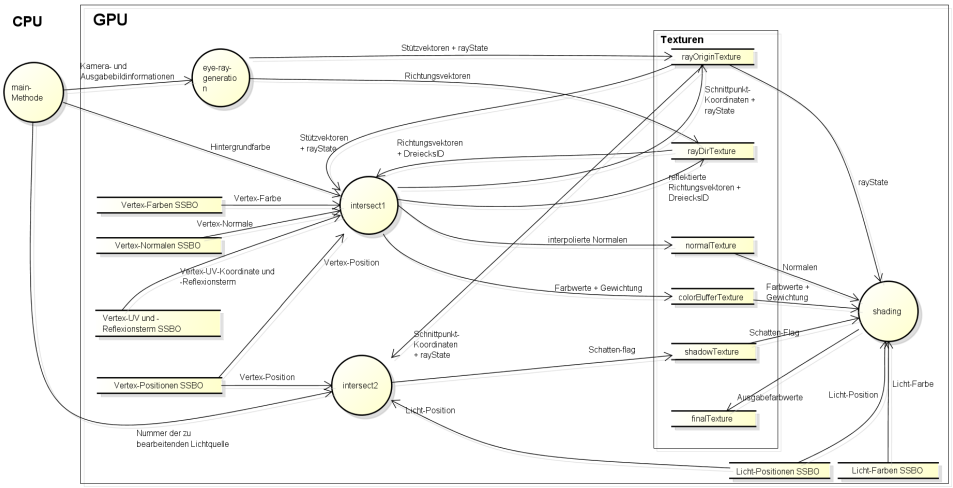


Abbildung 7: Datenflussdiagramm der Compute-Shader-Programme

Neben den genannten Texturen, die Berechnungsergebnisse der Kernel-Aufrufe beinhalten, existieren *Shader Storage Buffer Objects*, welche die grundlegenden Daten der Szeneobjekte beinhalten. Die Bereitstellung der SSBO-Inhalte erfolgt mit der Ausführung eines Vertex-Shader-Programms. Dabei greift der Vertex-Shader auf die Daten der *Vertex Array Objects (VAO)* der Szenenobjekte zu. Jedes VAO besteht dabei aus einer Reihe von *Vertex Buffer Objekts (VBO)*, die alle notwendigen Informationen der zugrunde liegenden Dreiecke beinhalten. Die folgende Aufzählung listet die Vertex-Attribute, auf die das Vertex-Shader-Programm in Form von VBOs zugreift, auf.

- Position
- Farbe
- Normale
- UV-Koordinate
- Reflexionsterm

Uniform-Variablen für die Modelmatrix und die Normalenmatrix eines Objektes ermöglichen es dem Vertex-Shader-Programm die Vertex-Position und -Normale zu transformieren. Bei einer bestehenden Objekt-Hierarchie wird diese vom Vertex-Shader verarbeitet und aufgehoben, so dass alle Vertex-Positionen in das globale Koordinatensystem überführt werden. Das Resultat der Transformation und unveränderte Attribute werden in den dafür vorgesehenen SSBOs auf der GPU gespeichert. Die Aufteilung der SSBOs gleicht dabei jener der VBOs.
Zusätzlich zu den zuvor aufgezählten Vertex-Attributen existieren zwei weitere SSBOs, da bei der Verarbeitung durch die Compute-Shader zwischen Szeneobjekten und Lichtquellen unterschieden wird. Diese SSBOs beinhalten zum einen die Positionen der Lichtquellen und zum anderen die Lichtfarbe. Auch diese SSBOs werden vom Vertex-Shader befüllt.

3.1.3 Rendering Ablauf

Um mehrere Lichtquellen und Reflexionen mit dem Basis-Ray-Tracer darstellen zu können, sind zum Teil mehrfache Aufrufe der Compute-Shader-Programme notwendig. Abbildung 8 zeigt den vollständigen Ablauf der Shader-Aufrufe in Abhängigkeit von der Anzahl der Lichtquellen und der darzustellenden Reflexionstiefe *REFL_DEPTH*.
Zu Beginn werden die Szene-Daten von einem Vertex-Shader-Programm transformiert und in den SSBOs auf der GPU bereitgestellt. Ein Leeren und Initialisieren der Texturen mit Nullen setzt die Texturinhalte zurück und überschreibt die Ergebnisse des vorherigen *frames*.Die Erzeugung der ersten Sehstrahlen erfolgt durch Ausführung des *eye-ray-generation*-Kernels. Im ersten Durchlauf $i = 0$ lädt das *intersect1*-Programm die primären Sehstrahlen aus den Texturen der Stütz- und Richtungsvektoren und verarbeitet diese. Bei $i > 0$ werden die einfach oder mehrfach reflektierten Sehstrahlen aus dem vorherigen Durchlauf auf Schnittpunkte mit der Szene getestet.
Innerhalb jedes Aufrufs des *intersect2*-Shader-Programms werden die Schattenfühler einer Lichtquelle erzeugt und auf Schnittpunkte getestet. Der Aufruf des Programms erfolgt daher pro Lichtquelle und mit der Kenntnis der aktuellen Nummer der zuverarbeitenden Lichtquelle über die Wertzuweisung einer *uniform*-Variablen durch das Hauptprogramm.
Die Verarbeitung der reflektierten Sehstrahlen, die beim ersten Aufruf des *intersect1*-Programms erzeugt wurden, erfolgt erst beim erneuten Ausführen der Kernel *intersect1* bis *shading*. Der erneute Aufruf des *intersect1*-Kernels liest dabei die reflektierten Sehstrahlen aus *rayOrigin-* und *rayDirTexture* aus und testet diese auf Schnittpunkte mit der Szene. Anschließend werden die Schattenfühler der reflektierten Schnittpunkte von den *intersect2*-Aufrufen verarbeitet. Der *shading*-Kernel fügt dem im ersten Durchlauf der Kernel generierten Ausgabebild ohne Reflexionen mit Hilfe der

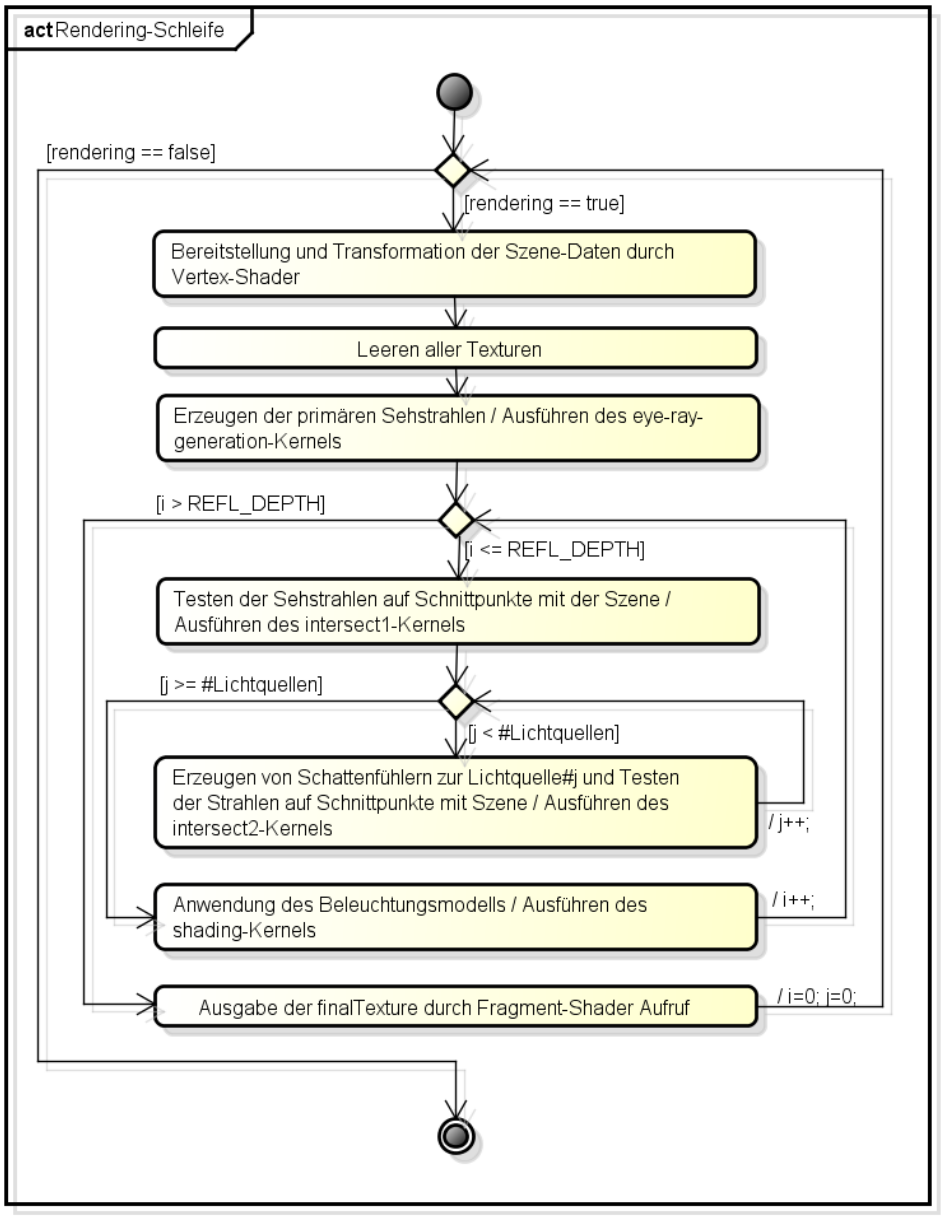


Abbildung 8: Aktivitätsdiagramm der Shader-Aufrufe innerhalb der Rendering-Schleife

Gewichtungen der Farbwerte das im zweiten Durchlauf erzeugte Reflexionsbild hinzu. Bei Mehrfachreflexionen erfolgen Schleifendurchläufe bis der festgelegte Grad der Reflexionsverfolgungen *REFL_DEPTH* erreicht wurde.
Für das Anzeigen der Ausgabetextur auf dem Bildschirm wird ein Fragment-Shader-Programm verwendet, dass die Ausgabetextur auf ein *screenfilling-quad* aufträgt. Die Texturkoordinaten erhält der Fragment-Shader durch ein vorher aufgerufenes Vertex-Shader-Programm, welches ausschließlich die Positionsdaten des *quads* verarbeitet. Der Zugriff auf die Daten der Ausgabetextur erfolgt über den passenden *binding point* der Textur. Über einen *2D-Sampler* ist es dem Fragment-Shader möglich auf die Farbwerte der Ausgabetextur lesend zuzugreifen und das *screen-filling quad* zu texturieren, so dass das gerenderte Bild auf dem Bildschirm angezeigt wird.
Zwischen den einzelnen Shader-Aufrufen wird durch den Aufruf von *glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT)* oder *glMemoryBarrier(GL_IMAGE_ACCES_BARRIER_BIT)* eine Speicher-Barriere erstellt und damit sichergestellt, dass die Zugriffe auf die verwendeten Datenstrukturen des nachfolgenden Shader-Programms erst erfolgen, wenn das vorherige Programm seine Bearbeitung abgeschlossen hat.

3.2 Uniform Grid

Die Aufteilung des Szeneraums in Voxel erfordert die Einteilung der darzustellenden Dreiecke in diese Voxel und eine effiziente Bestimmung der geschnittenen Voxel eines Strahls, der die Szene durchquert. Hierfür werden Listen angefertigt, die auf Dreiecke und Voxel verweisen, so dass die Zuordnungen von beiden durch Verweise erfolgen kann. Dazu ist die Einführung von weiteren Kerneln zum Rendering-Ablauf und die Anpassung vorhandender Shader-Programme erforderlich.
Dieser Abschnitt stellt einen Algorithmus zur Zuordnung von Strahlen zu geschnittenen Voxeln vor, auf dem die zusätzlichen Kernel basieren. Weiterhin wird die Interpretation der auf der GPU erstellten Listen der Datenstruktur erläutert und der erweiterte Rendering-Ablauf aufgezeigt.

3.2.1 Datenstruktur

Die Erweiterung des Basis-Ray-Tracers durch ein Uniform Grid zur Beschleunigung der Berechnungen erfordert zusätzlich den Aufbau des *grids* und die Traversierung der Datenstruktur für zu verfolgende Strahlen. Unter dem Aufbau der Datenstruktur versteht man die Zuordnung aller in der Szene befindlichen Dreiecke in die beinhaltenden Voxel des Uniform Grids. Dafür besitzt jeder Voxel des Uniform Grid eine Liste mit DreiecksIDs der Dreiecke, die sich in ihm befinden. Die Traversierung der Datenstruktur für einen Strahl produziert eine Liste von VoxelIDs, die ein Strahl beim

Durchqueren der Szene schneidet. Mit Hilfe der erstellten Listen ist es den *intersection*-Kerneln möglich nur solche Dreiecke auf einen Schnittpunkt mit dem Strahl zu testen, die innerhalb von den geschnittenen Voxeln des Strahls liegen. Dies benötigt eine Abwandlung der *intersection*-Kernel.
Die jeweiligen Listen werden innerhalb von SSBOs als Abschnitte von 1D-Datenfeldern realisiert. Zur Ermittlung des benötigten Speichers werden folgende Konstanten benötigt.

- Anzahl der Voxel des Uniform Grids $voxel\_amount$
- Anzahl der gespeicherten DreiecksIDs pro Voxel $voxel\_array\_length$
- Anzahl der Pixel/Sehstrahlen $resolution$
- Anzahl der gespeicherten VoxelIDs je Pixel/Sehstrahl ray_array_length

Die folgenden SSBOs werden zur Bereitstellung des benötigenten GPU-Speichers für die zu erstellenden Listen angelegt.

- **voxelslist**
  Menge aller Listen, die DreiecksIDs der Dreiecke beinhalten, die in zugehörigen Voxel liegen
  Speichergröße: $voxel\_amount \cdot voxel\_array\_length$
- **voxelcontains**
  Menge aller Listen, die den Füllstand der Dreieckslisten eines jeden Voxels beinhalten
  Speichergröße: $voxel\_amount$
- **rayslist**
  beinhaltet VoxelIDs der Voxel, die ein zugehöriger Strahl schneidet
  Speichergröße: $resolution \cdot ray\_array\_length$
- **raycontains**
  Menge aller Listen, die den Füllstand der Voxellisten eines jeden Strahls beinhalten
  Speichergröße: $resolution$

Bei dem Füllen der Abschnitte der *voxelslist* und der *rayslist* wird der aktuelle Füllstand innerhalb der mit den Listen korrespondierenden Speicherplätze der *voxelcontains*- und *raycontains*-Arrays aktualisiert. Abbildung 9 zeigt eine beispielhafte Auflösung der Referenzen der Listen bis hin zu den zu testenden Vertex-Positionen der Dreieckseckpunkte. Um diese Dereferenzierung werden die *intersection*-Kernel erweitert. Abbildung 10 zeigt den Ablauf des *intersect2*-Kernels mit dieser Erweiterung. Die Arbeitsweise des *build*-Kernels für die Einteilung der Dreiecke in die Voxel und der *traversal*-Kernel zur Traversierung der Datenstruktur für Strahlen basieren

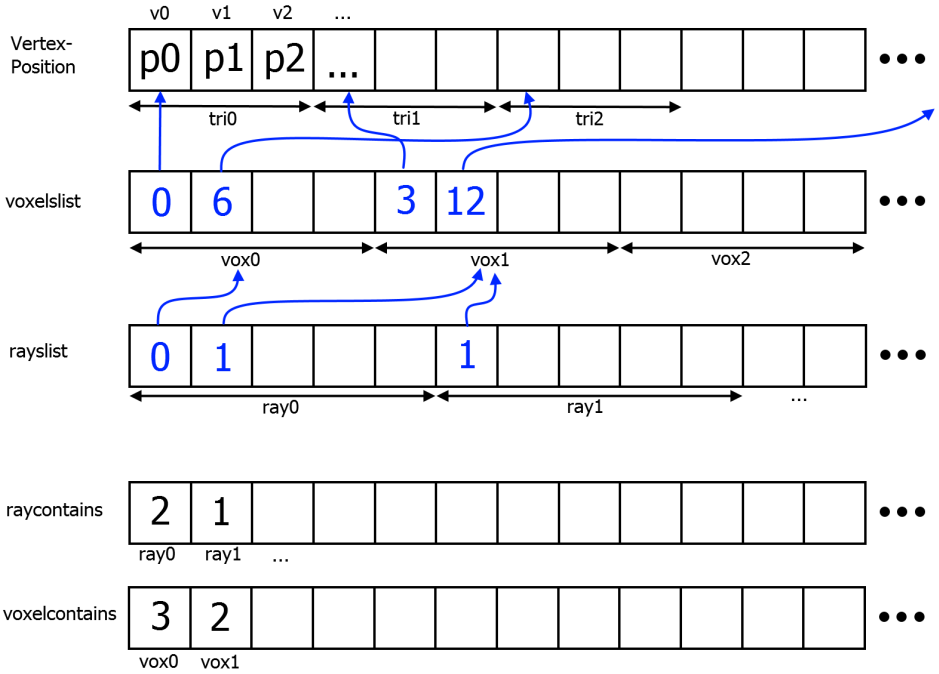


Abbildung 9: Auflösung der Referenzen bei der Interpretation der Voxel- und Dreieckslisten

auf den in Abschnitt 3.2.2 vorgestellten Algorithmus 2 und 3. Die Ausführung des Algorithmus 2 erstellt eine temporäre Liste aller Schnittpunkte eines Strahls entlang der Koordinatenachsen mit den Kanten der Voxel im Cache des ausführenden Threads. Algorithmus 3 arbeitet auf der Ausgabe von Algorithmus 2, bestimmt ausgehend vom Start-Voxel alle Übergänge in die benachbarten Voxel und fügt die besuchten Voxel zu einer Ausgabeliste innerhalb der SSBOs hinzu.
Die Aufteilung eines Dreiecks in seine Seiten und die Behandlung dieser als Strahlen mit Anfangs- und Endpunkt ermöglicht die Verwendung der Algorithmen innerhalb des *build*-Kernels. Es existiert weiterhin die Vorgehensweise eine *axis-aligned bounding box* um das einzuordnende Dreieck zu legen und alle von der Bounding Box geschnittenen Voxel zu markieren, wie beispielsweise in der Zeitschrift „Computer-Aided Design“ von Akman et al. [3] vorgeschlagen. Allerdings führt diese Vorgehensweise zu mehr Einträgen in den Dreieckslisten der Voxel, da die Bounding Box eine ungenauere Annäherung an das Dreieck darstellt als die Verarbeitung der einzelnen Seiten mit den folgenden Algorithmen. Die Anzahl der Ausführung von atomaren Funktionen wäre höher und die Performanz dadurch niedriger.

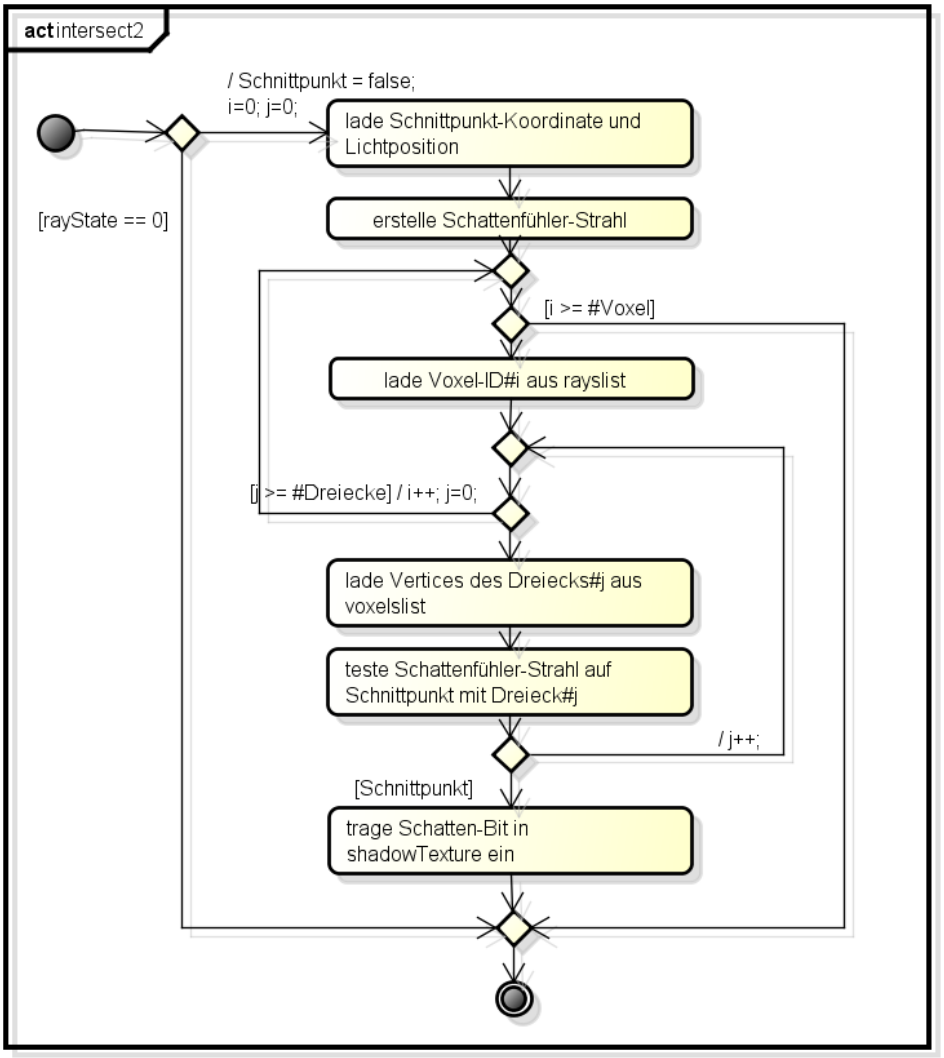


Abbildung 10: Aktivitätsdiagramm des intersect2-Kernel-Aufrufs des mittels Uniform Grid beschleunigten Ray-Tracers

3.2.2 Voxel-Traversierung

Um das Vorgehen der Algorithmen zu erläutern werden zuerst mathematische Grundlagen erläutert und die Transformation von Koordinaten in verschiedene genutzte Koordinatensysteme aufgezeigt.
Die Darstellung eines Strahls $\vec{r}(t) \in \mathbb{R}^3$ erfolgt mit Hilfe des Parameters $t \in [0, t_{max}] \subset \mathbb{R}$, $t_{max} \in \mathbb{R}_+$, dem Stützvektor $\vec{ro} \in \mathbb{R}^3$ und dem Richtungsvektor $\vec{rd} \in \mathbb{R}^3$.

$$\vec{r}(t) = \vec{ro} + t \cdot \vec{rd} \tag{1}$$

Bei der Anwendung auf Sehstrahlen ist es notwendig zu Beginn den Endpunkt des Strahls $r(t_{max})$ zu bestimmen. Dazu bestimmt man für jede Raumdimension des Strahls separat den Schnittpunkt mit der Außenwand der Szene in Strahlrichtung. Gleichung 2 stellt dies beispielhaft für den x-Wert des Strahls dar. Dabei steht etwa $\vec{ro}_x$ für den Wert der x-Komponente des Vektors $\vec{ro}$. Der Endpunkt des Strahls ergibt sich aus dem Minimum aller Schnittpunktparamater der einzelnen Raumdimensionen.

$$\vec{ro}_x + t_x \cdot \vec{rd}_x = \frac{\vec{rd}_x}{|\vec{rd}_x|} \cdot 1 \tag{2}$$

$$t_{max} = min(t_x, t_y, t_z) \tag{3}$$

Zur Anwendung des Algorithmus 2 werden der Anfangs- und der Endpunkt des Strahls im *Voxelkoordinatensystem* benötigt.
Die Unterscheidung der Voxel erfolgt über dreidimensionale *Voxelkoordinaten* $\vec{v}$, siehe Gleichung 5. Dabei besitzt jeder Voxel eine eindeutige Voxelkoordinate.

$$v_x, v_y, v_z \in [0, voxel\_amount - 1] \subset \mathbb{Q}_+ \tag{4}$$

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \in \mathbb{Q}_+^3 \tag{5}$$

Gleichung 6 zeigt die Übertragung einer Raumkoordinate $\vec{p}_r \in \mathbb{R}^3$ der Szene in das Voxelkoordinatensystem $\mathbb{Q}_+^3$. Dabei wird der Koordinatenraum durch die Addition von $(1, 1, 1)$ von $x, y, z \in [-1, 1)$ auf $[0, 2)$ verschoben. Die Division durch die Anzahl der Voxel entlang einer Koordinatenachse ergibt zusammen mit der Gaußklammer die Voxelkoordinate des Voxels, in dem sich der Punkt $\vec{p}_r$ befindet. Man beachte dabei, dass durch das direkte Angrenzen der Voxel – also das Ineinanderliegen der Seiten benachbarter Voxel – ein Punkt, der auf einer Voxelseite liegt, dem Voxel zugeordnet wird, bei dessen Seite es sich um die vordere, linke oder untere Seite handelt.

$$toVoxCoord(\vec{p}_r) = \lfloor \frac{1}{voxel\_amount} \cdot (\vec{p}_r + \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}) \rfloor \tag{6}$$

Die benötigten Start- und End-Voxel vox_min und vox_max erhält man durch Übertragung der Punkte $\vec{r}(0) = \vec{ro}$ und $\vec{r}(t_{max})$ in das Voxelkoordinatensystem.
Algorithmus 2 erstellt eine Liste t_list, in der alle Parameter t der Schnittpunkte entlang der x-Koordinatenachse gespeichert werden, siehe Abbildung 11.

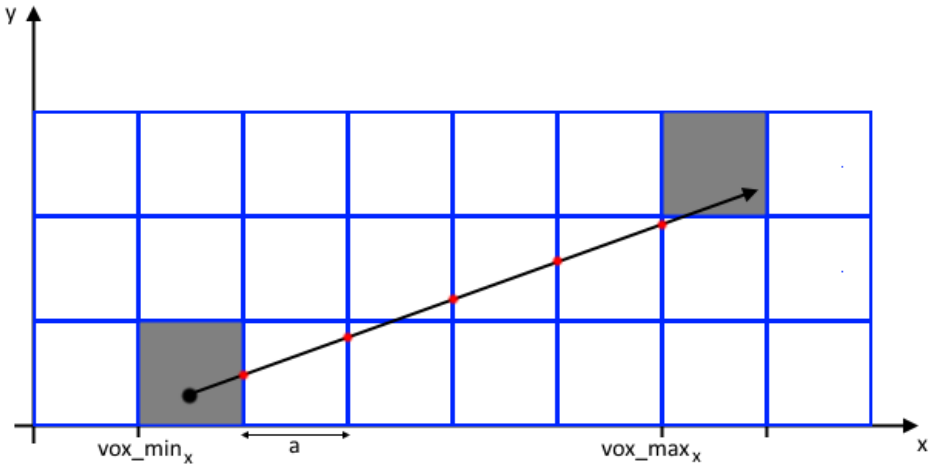


Abbildung 11: Verfolgung eines Strahls entlang der x-Koordinatenachse mit den Schnittpunkten (rot) an den Voxelkantenebenen (blau) mit $x = i \cdot a - 1$

Ein Hilfsparameter i dient der Iteration über alle Voxelkanten-Ebenen und repräsentiert indirekt die in diesem Moment verarbeitete Voxelkanten-Ebene. Beginnend bei einer Kante des Start-Voxels wird die bearbeitete Kante in Laufrichtung *direction* auf alle von Start- und End-Voxel eingeschlossenen Ebenen bis zur letzten zu testenden Kante $j - 1$ verschoben. Wie Abbildung 12 aufzeigt, sind der initiale Wert von *i* und der Wert von *j* von der Strahlrichtung in der verarbeiteten Dimension abhängig. Bei positiver Richtung am Beispiel von $\vec{v_1}$ ist die letzte zu testende Ebene bei vox_max_1, da der Strahlendpunkt in diesem Voxel liegt (siehe Gleichung 6); j entspricht also für die positive Richtung $vox\_max + 1$. Für die negative Richtung gilt $j = vox\_max$, da die linke Kante des End-Voxels nicht mehr geschnitten wird. Des Weiteren folgt für *i*:

$$direction > 0 : i \in [vox\_min + 1, j)$$
$$direction < 0 : i \in [vox\_min, j)$$

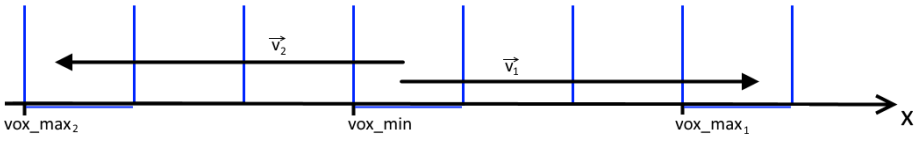


Abbildung 12: Geschnittene Voxelkanten-Ebenen (blau) in Abhängigkeit der Strahlrichtung

Durch Gleichsetzen des Strahlenterms in x-Richtung $\vec{ro}_x + t \cdot \vec{rd}_x$ mit

den Voxelkanten-Ebenen $i \cdot a - 1$, wobei a die Voxelkantenlänge ist, erhält man die Parameter *t* des jeweiligen Schnittpunkts. Durch weitere An-

Algorithmus 2 Bestimmung der Schnittpunkte eines Strahls entlang der x-Koordinatenachse

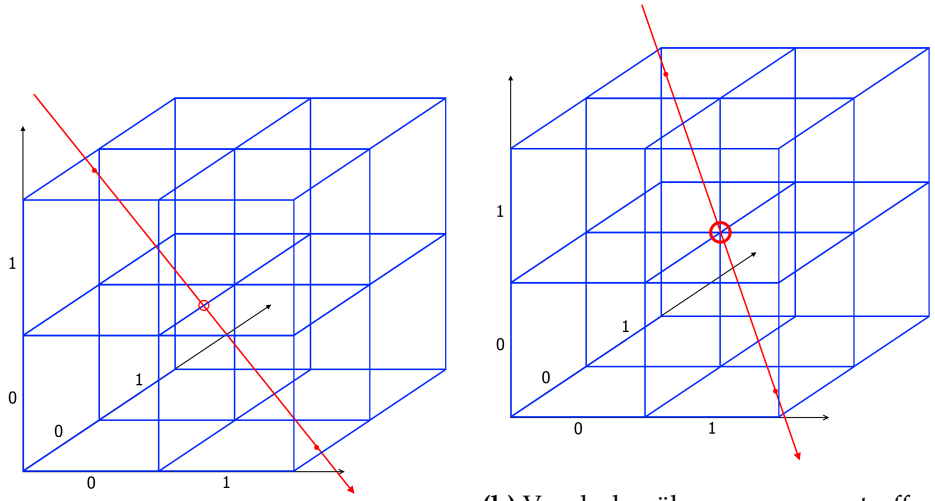
$t\_list \leftarrow$ **new** $list$
$dimension \leftarrow$ **new** $list$
$direction \leftarrow \overrightarrow{rd_x}/|\overrightarrow{rd_x}|$
$a = 2/voxel\_amount$
$i \leftarrow vox\_min_x$
$j \leftarrow vox\_max_x$
if $\overrightarrow{rd_x} > 0$ **then**
  $i++;$
  $j++;$
end if
while $i \neq j$ **do**
  $t = \frac{i \cdot a - 1 - \vec{ro}_x}{\overrightarrow{rd_x}}$
  if ($i \geq 0$ **and** $t \leq t_{max}$**) then**
    $t\_list.add(t);$
    $dimension.add(0);$
  end if
  $i = i + direction \cdot a$
end while

wendung des Algorithmus 2 (ohne die Neuerstellung der Liste) auf die y- und z-Koordinatenachsen, wird *t_list* mit weiteren Schnittpunktparametern gefüllt. Die Liste *dimension* enthält eine Kennung, in welcher Dimension der Parameter t gefunden wurde. Durch aufsteigende Sortierung der Parameter-Liste erhält man alle Schnittpunkte des Strahls mit den Voxelkanten innerhalb der drei Raumdimensionen von Start- bis End-Voxel. Gleichzeitig wird die Liste *dimension* zur Bewahrung der Speicherplatz-Korrelation den gleichen Schritten zur Sortierung unterzogen wie die Parameter-Liste. Um alle durchquerten Voxel des Strahls zu markieren, iteriert man wie in Algorithmus 3 beschrieben über die sortierte Liste *dimension*. Von dem Startvoxel ausgehend wird die Voxelposition innerhalb der Raumdimension verschoben, in der der nächste Schnittpunkt der Schnittpunktsuche über die einzelnen Koordinatenachsen gefunden wurde.
Der Eintrag in die entsprechende Liste wird von der *storeVoxel()*-Methode vorgenommen. Dabei unterscheidet sich je nach Verarbeitung von Sehstrahlen oder Dreiecksseite die Ziel-Liste und die Art des einzutragenden Datums. Verfolgt ein Thread der *traversal*-Kernel einen Sehstrahl, werden alle markierten Voxel der zu erstellenden Liste aus Voxel-IDs des Sehstrahl hinzugefügt. Bei der Einordnung eines Dreiecks in alle beinhaltenden Voxel durch einen Thread des *build*-Kernels, wird die Dreiecks-ID beim Ausfüh-

Algorithmus 3 Bestimmung der Voxel, die ein Strahl durchquert

$current \leftarrow vox\_min$
for int $j = 0$; $j < t\_list.size()$; j++ **do**
  if $dimension[j] = 0$ **then**
    $move \leftarrow \frac{\vec{rd_x}}{|\vec{rd_x}|} \cdot (1, 0, 0)$
  end if
  if $dimension[j] = 1$ **then**
    $move \leftarrow \frac{\vec{rd_y}}{|\vec{rd_y}|} \cdot (0, 1, 0)$
  end if
  if $dimension[j] = 2$ **then**
    $move \leftarrow \frac{\vec{rd_z}}{|\vec{rd_z}|} \cdot (0, 0, 1)$
  end if
  $current = current + move$
  storeVoxel($current$)
end for

ren der *storeVoxel()*-Methode zur Liste des Voxels *current* hinzugefügt. Der Start-Voxel eines Strahls wird von den Algorithmen nicht erfasst, so dass dafür separat anfangs der Aufruf *storeVoxel(vox_min)* erfolgt.
Da es bei der parallelen Einordnung der Dreiecke zu Voxeln zu gleichzeitigen schreibenden Zugriffen auf die Dreiecks-Liste des Voxels kommen kann, bedarf die Implementation der *storeVoxel()*-Methode innerhalb des *build*-Kernels der Synchronisation der Threads.
Des Weiteren muss die Reihenfolge der zu speichernden Voxel-IDs beachtet werden, wenn zwei oder mehrere Schnittpunktparameter verschiedener Dimensionen den gleichen Wert besitzen. In diesem Fall schneidet der Strahl exakt eine Kante oder einen Eckpunkt des Voxels und die Voxelübergänge müssen in einer bestimmten Reihenfolge erfolgen. Siehe zur Verdeutlichung Abbildung 13a. Bevor die Voxel-IDs auf Basis der Schnittpunktparameter *t* in der im Cache befindlichen Liste *t_list* in den SSBOs gespeichert werden, kommt es zur Prüfung auf Gleichheit des aktuell verarbeiteten Parameters und seiner Nachfolger. Sind zwei oder drei aufeinander folgende Parameter gleich, kommt es zur Prüfung der Strahlrichtung innerhalb der Dimensionen, in denen der Schnittpunkt gefunden wurde. Handelt es sich um die gleiche Laufrichtung kommt es nur zu einem Voxelübergang – der simultan Ausführung aller Übergänge – an der Voxelkante (beziehungsweise der Voxelecke) anstatt zu zwei (beziehungsweise drei) an den Voxelseiten. Bei unterschiedlicher Laufrichtung innerhalb der Dimensionen werden zuerst die Voxelübergänge an den Voxelseiten der positiven Laufrichtung vorgenommen, und anschließend jene der negativen Richtungen.



(a) Voxelkantenübergang - getroffene Kante liegt in Voxel (1,1,0); von Voxel (0,1,0) kommend muss zuerst der Übergang in positiver x-Richtung vollzogen werden

(b) Voxeleckenübergang - getroffene Ecke liegt in Voxel (1,1,1); von Voxel (0,1,1) kommend muss zuerst der Übergang in positiver x-Richtung vollzogen werden, danach simultan in negativer y- und z-Richtung

Abbildung 13: Beispiele für Sonderfälle der Voxelübergänge; Eintrittspunkte eines Strahls in einen Voxel als roter Punkt, kritischer Voxelübergang mit rotem Kreis markiert

3.2.3 Uniform Grid Kernel

Mit Hilfe der in Abschnitt 3.2.2 vorgestellten Algorithmen folgt für den *build*-Kernel das in Abbildung 14 illustrierte Aktivitätsdiagramm. Ein Aufruf eines *build*-Kernels ordnet genau ein Dreieck der Szene in die Listen der getroffenen Voxel ein. Dabei werden Strahlen zwischen den Eckpunkt-Vertices erzeugt und jeder Strahl wird gegen die Datenstruktur traversiert. Nach der Anwendung des Algorithmus 2 zur Bestimmung aller Schnittpunkte mit den zwischen den Eckpunkten liegenden Voxelkanten-Ebenen, werden die Voxelübergänge vollzogen (Algorithmus 3) und die Dreiecks-ID in die Listen der besuchten Voxel eingetragen.

3.2.4 Erweiterter Rendering Ablauf

Mit Integration des Uniform Grids in den Basis-RT ergibt sich der in Abbildung 15 Rendering-Ablauf des beschleunigten Ray-Tracers.
Um dynamische Szenen darstellen zu können, kommt es in jedem *frame* zum Aufbau der beschleunigenden Datenstruktur. Dafür kommt es am Anfang des Durchlaufs der Rendering-Schleife zum Leeren der erstellten Listen *voxelslist* und *voxelcontains*. Der Aufruf des *build*-Kernels nach Be-

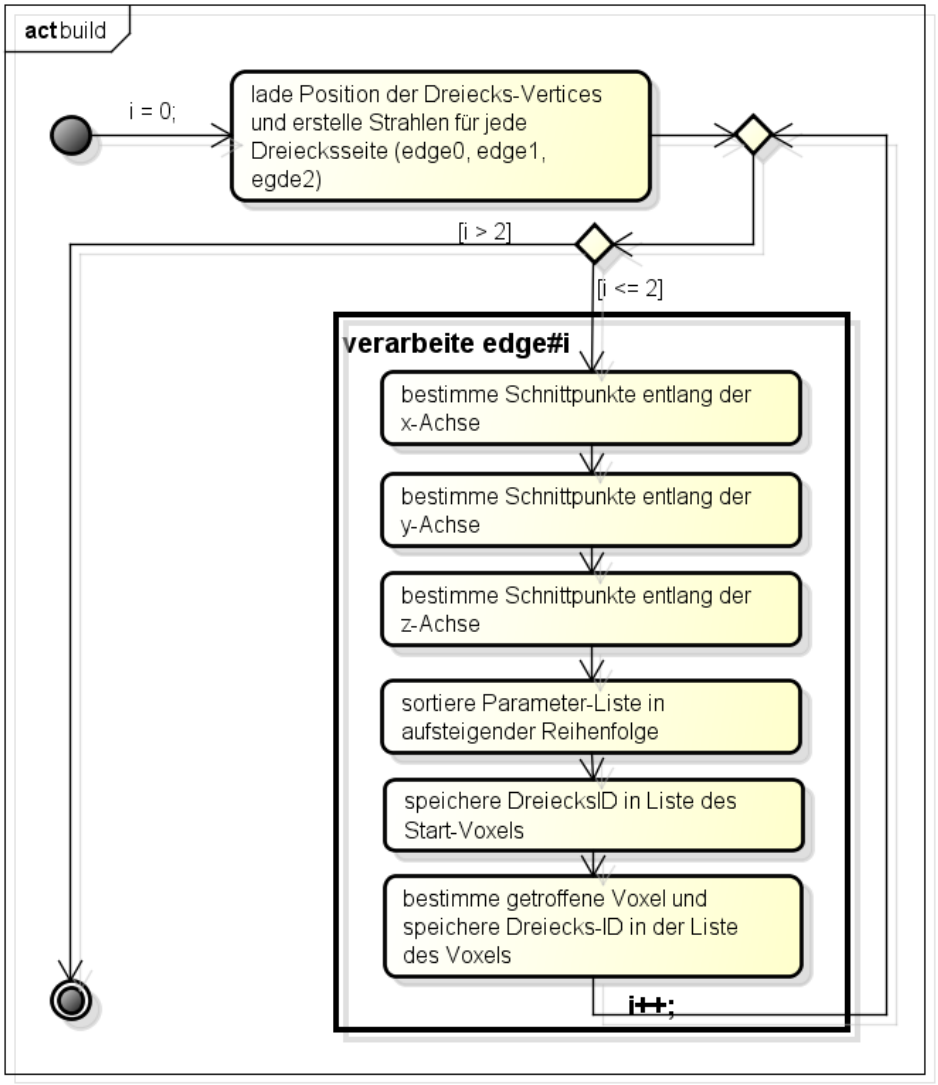


Abbildung 14: Aktivitätsdiagramm des *build*-Kernels zur Einordnung der Szenen-Dreiecke in die beinhaltenden Voxel

reitstellung der Vertex-Daten durch den Vertex-Shader ordnet die Szene-Dreiecke den Voxeln zu, in denen sie liegen.
Bevor ein Strahl von einem der *intersection*-Kernel auf Schnittpunkte mit der Szene geprüft wird, wird der Strahl gegen die Datenstruktur traversiert und eine korrespondierende Liste mit zu testenden Voxeln gefüllt. Dies geschieht sowohl für primäre und (mehrfach) reflektierte Sehstrahlen, als auch für die Schattenfühler der einzelnen Lichtquellen. Bevor die Listen *rayslist* und *raycontains* für die Bearbeitung eines Strahls befüllt werden, werden diese erneut initialisiert und die vorherigen Werte gelöscht.

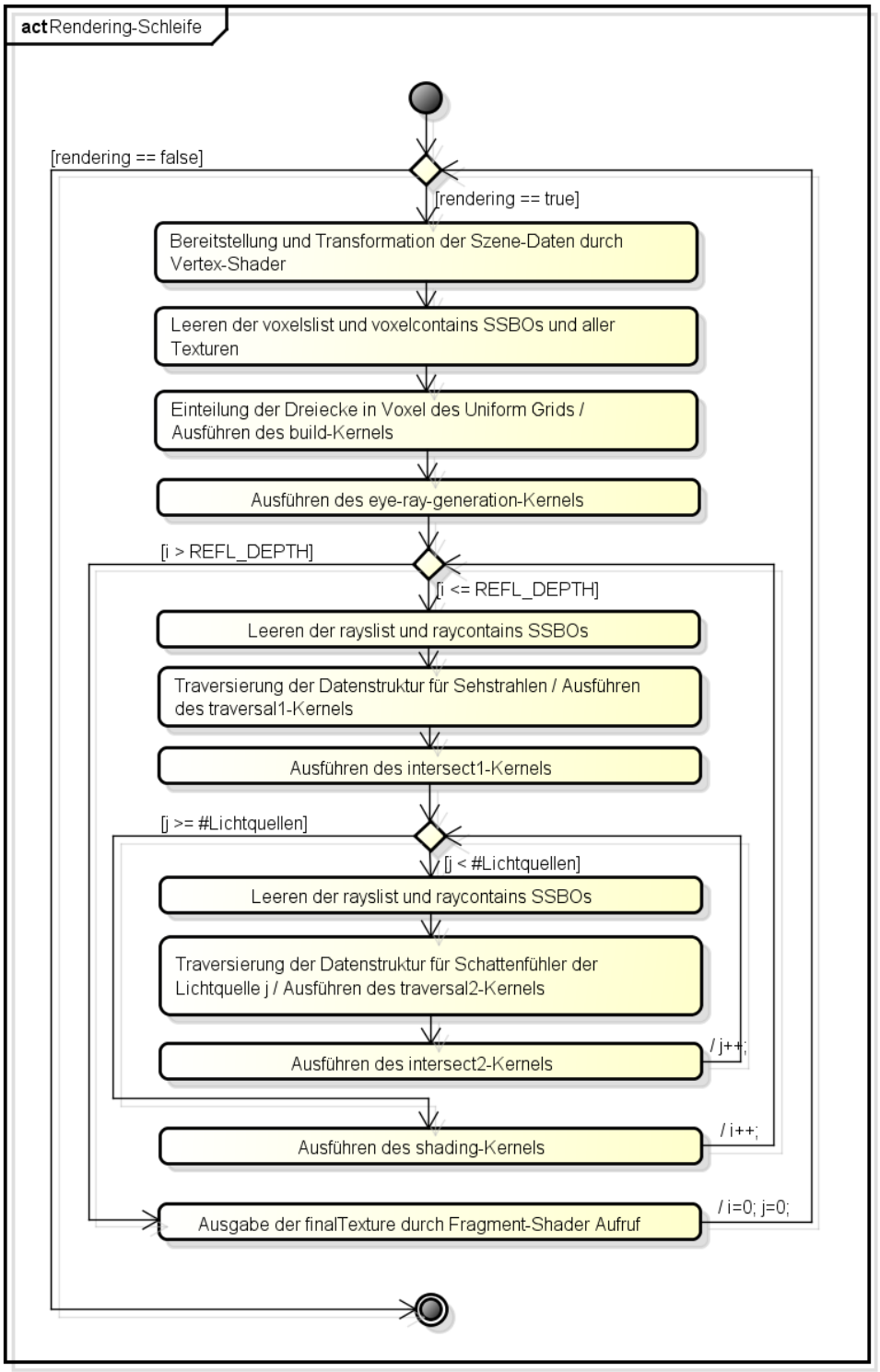


Abbildung 15: Aktivitätsdiagramm zum erweiterten Rendering-Ablauf des Ray-Tracers mit integriertem Uniform Grid

3.3 Objektstruktur

Zur Vereinfachung der zu verwaltenden Aufrufe wurde das *CVK*-Framework der AG Computergrafik der Universität Koblenz-Landau verwendet. Es kommt zur Nutzung der Klasse *CVK::ShaderSet*, die den Shader-Code lädt und eine Kompilierung ermöglicht. Die Klasse *CVK::Geometry* umfasst sämtliche Vertex-Attribute und Instanzen der Klasse können in Szene-Objekte überführt werden.
Eine Übersicht über die erstellten C++-Klassen verschafft die Abbildung 16.

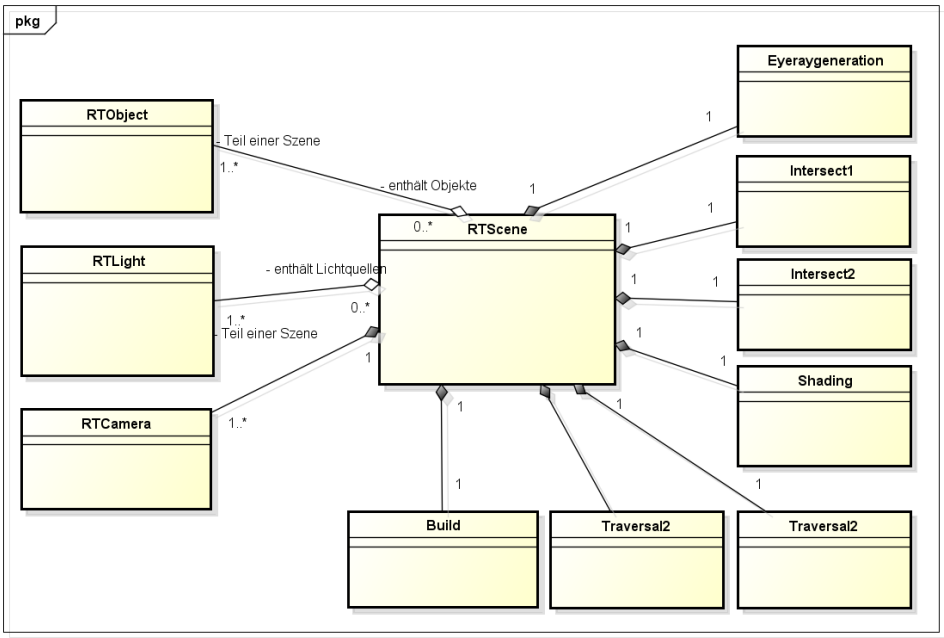


Abbildung 16: Übersicht der Klassen zur Verwaltung der Ray-Tracer Impementierung

Die Objekt-Klasse*RTScene* bildet den Kern des Ray-Tracing Renderings und vereint alle für das Rendering benötigten Szene-Informationen. Dazu zählt die Menge aller Szene-Objekte als Instanzen der *RTObjekt*-Klasse, Lichtquellen der *RTLight*-Klasse und die Kamera als Instanz der *RTCamera*-Klasse. Die folgenden benötigten Daten beinhalten die Attribute des *RTScene*-Objekts, die auch detailliert in Abbildung 17 aufgezeigt werden.

- Auflösung des Ausgabebildes *height* und *width*
- ambiente Lichtfarbe *ambientLightColor*
- GPU-Referenzen
  - Texturen *texture handle*

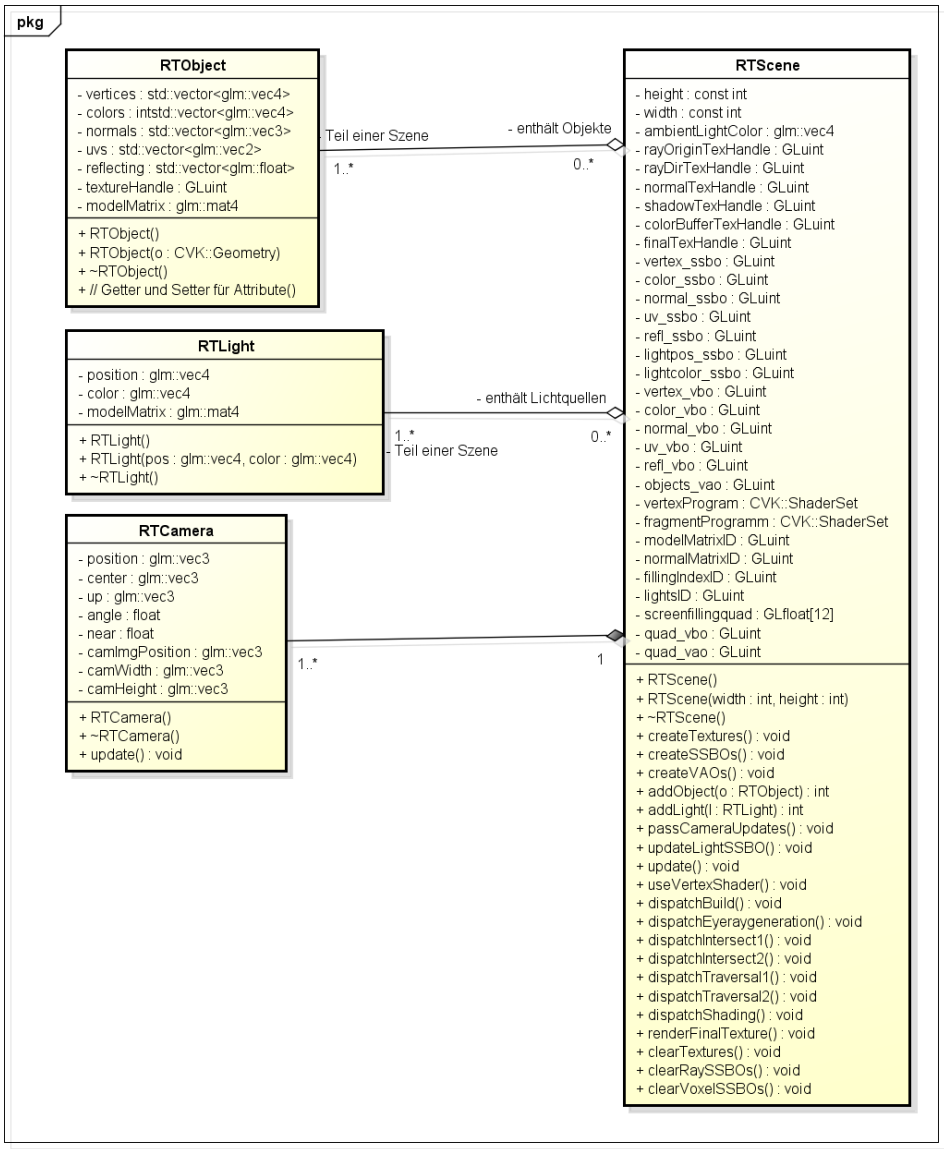


Abbildung 17: Ausschnitt des Klassendiagramms mit den Klassen, die für den Aufbau einer zu rendernden Szene verwendet werden

  - SSBOs
  - VBOs
  - VAOs
- *ShaderSet*-Objekte der Vertex- und Fragment-Shader
- *uniform IDs* zur Werteübergabe an Vertex- und Fragment-Shader
- Screen-Filling Quad Vertex-Positionen

Des Weiteren verfügt das Szene-Objekt über Methoden zur Erstellung der benötigten Datenstrukturen und zur Initialisierung dieser mit vorgegebenen Werten. Darüber hinaus ist eine Verwaltung der Szene-Objekte und Lichtquellen möglich. Die Ausführung der Compute-Shader-Kernel erfolgt über die jeweiligen *dispatch*-Methoden, die eine Wertübergabe in Form von *uniform*-Variablen beinhalten, falls diese für die Ausführung benötigt werden. Bei der Aktualisierung der Kamera werden die Änderungen mittels der *passCameraUpdates()*-Methode an die Kernel übergeben. Bei dem Aufruf des Vertex-Shader-Programms zur Transformation der Vertex-Positionen und der Befüllung der Vertex-Attribut-SSBOs verfügt dieser über den Zugriff auf alle dem Szene-Objekt zugeordneten *RTObject*-Instanzen.
Für die Verwaltung der Kernel-Programme wurde für jeden Kernel eine eigene Klasse erstellt (Abbildung 18). Dabei beinhaltet jede Klasse ein *ShaderSet*-Objekt in dem der Programmcode des Kernels geladen wird. Außerdem existieren, falls benötigt, *uniform IDs* mit denen die Wertübergabe an das Schader-Programm erfolgen kann. Jede Klasse verfügt über eine *dispatch*-Methode über die das Kernel-Programm aufgerufen werden kann.

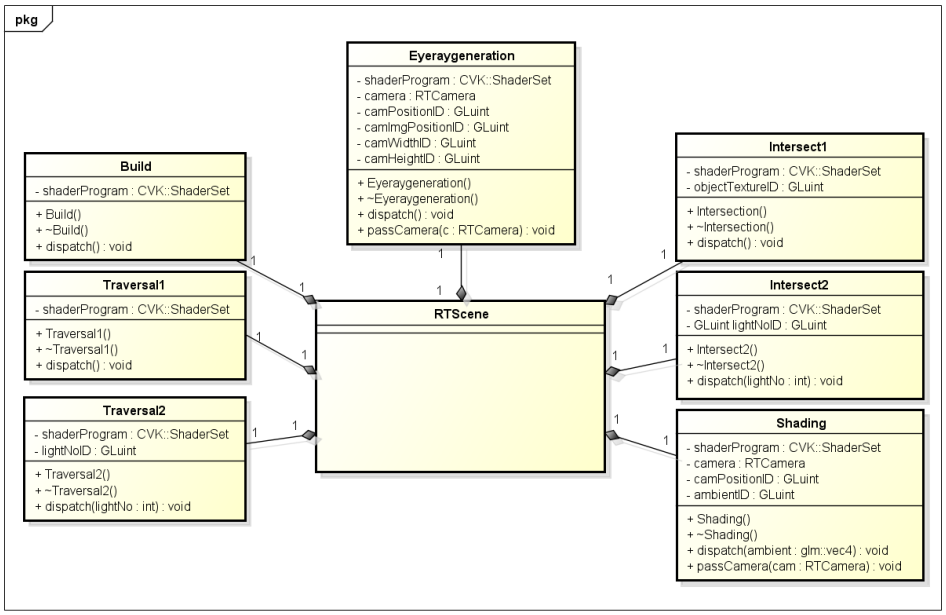


Abbildung 18: Ausschnitt des Klassendiagramms mit den Klassen der Ray-Tracing Kernel

4 Implementierung

Aus der aufgezeigten Konzeption des GPGPU-Ray-Tracers unter Verwendung der OpenGL Compute-Shader resultieren die in diesem Abschnitt behandelten Code-Ausschnitte der Kernel-Programme.
Zuerst werden die grundlegenden Funktionalitäten des Basis-Ray-Tracers in Abschnitt 4.1 realisiert und erläutert. Darauf aufbauend findet die Erweiterung des Ray-Tracers durch Einfügen des Uniform Grids in Abschnitt 4.2 statt. Im Folgenden wird die zu verarbeitende Pixelposition eines Kernel-Aufrufs mit der Konstanten *storePos* bezeichnet. Sie ergibt sich aus den xy-Koordinaten des einzelnen Threads innerhalb der globalen Arbeitsgruppe, so dass jeder Thread genau einen Pixel verarbeitet, wenn der Kernel-Aufruf pro Pixel der Textur erfolgt.

4.1 Basis-Ray-Tracer

In diesem Abschnitt wird die Umsetzung des entwickelten Basis-Ray-Tracers anhand des Programmcodes der einzelnen Kernel aufgezeigt und erläutert. Dabei erfolgt der Dispatch jedes dieser Compute-Shader-Kernel pro Pixel des zu erzeugenden Ausgabebildes.

4.1.1 *eye-ray-generation*-Kernel

Die Grundlage der Generierung der initialen Sehstrahlen durch den *eye-ray-generation*-Kernel bilden die Koordinaten der Ausgabebild-Projektion in den Szene-Raum. Diese werden mit Hilfe der Kamera-Attribute innerhalb des Hauptprogramms vor der Ausführung des Compute-Shader-Kernels aktualisiert. Das Zentrum des projezierten Bildes wird durch die Multiplikation der Verbindung aus Kameraposition und Augpunkt mit einem *near-clipping*-Faktor bestimmt. Die Ausrichtung der Bildhöhe und -breite im Szenen-Raum sind durch die Blickrichtung, dem *up*-Vektor und dem Öffnungswinkel der Kamera bestimmbar. Des Weiteren ist es möglich die Koordinate der oberen linken Ecke der Bild-Projektion zu ermitteln und diese als *uniform*-Vektor *camImgPosition* an den Kernel zu reichen. Die Bild-Projektions-Höhe und -Breite wird mit der jeweiligen Auflösung skaliert und auf den Abstand von einem Pixel zu seinem Nachbarn reduziert. Der Pixelabstand in der Breite wird dabei als Vektor *camWidth* und in der Höhe als *camHeight* an das Shader-Programm übergeben.
Nachdem die Übergabe der Kameraposition und der genannten Koordinaten-Vektoren erfolgt ist, kommt es zur Ausführung des Shader-Programms. Listing 4.1.1 zeigt den Programm-Code der *main()*-Methode des *eye-ray-generation*-Kernels. Von der Koordinate des oberen linken Bild-Eckpunkts ausgehend, bestimmt jeder Aufruf durch wiederholte Addition der Pixel-Abstände *camWidth* und *camHeight* die Position der Pixel-Koordinate, der

er zur Verarbeitung zu geteilt ist. Die ermittelte Raumposition wird als Stützvektor des Sehstrahls in *rayOriginTexture* eingetragen und der *rayState* im Alpha-Kanal mit *1* initialisiert. Durch die Normalisierung der Strecke zwischen der Pixel- und der Kameraposition erhält man den Richtungsvektor des Sehstrahls. Bei dem speichernden Zugriff auf *rayDirTexture* wird zusätzlich zu dem Ablegen des Richtungsvektors der Alpha-Kanal mit *-1* initialisiert – eine ungültige Dreiecks-ID, die symbolisiert, dass noch kein Auftreffen des Strahls stattgefunden hat.

```
// xy-Koordinate des zuverarbeitenden Pixels innerhalb der Textur
float x = float(storePos.x), y = float(storePos.y);

/* Bestimmung der Raumkoordinate der Projektion eines
 * Pixels auf imaginäre Ebene im Szenenraum */
vec4 pixelCoord = camImgPosition
                  + x * camWidth
       // Abfrage der Textur-Höhe zur Invertierung der
       // y-Laufrichtung
       + (imageSize(rayDirTexture).y - y) * camHeight;

vec3 direction = normalize(pixelCoord.xyz - camPosition.xyz);

imageStore(rayOriginTexture, storePos, vec4(pixelCoord.xyz, 1.f));
imageStore(rayDirTexture, storePos, vec4(direction, -1.f));
```

Listing 11: *main()*-Methode des *eye-ray-generation*-Kernels

4.1.2 *intersection*-Kernel

Das *intersect1*-Shader-Programm benötigt den Zugriff auf alle Vertex-Attribut-SSBOs, auf alle Texturen bis auf Schatten- und Ausgabetextur und zusätzlich auf die in Frage gekommenden *sampler2D*-Objekte der Texturen, die für eine eventuelle Texturierung einer Dreiecksfläche benötigt werden. Für den Fall, dass ein Strahl kein Objekt der Szene schneidet, wird dem Kernel der Farbvektor der Hintergrundfarbe übermittelt.
Wird eine aktive Pixelposition bearbeitet, wird der Sehstrahl geladen und der maximale Strahlparameter innerhalb der Szene bestimmt. Die Iteration über alle Dreiecke der Szene erfordert die Deklaration von Puffer-Variablen, die zur Speicherung von gegebenenfalls ermittelten Schnittpunkt-Eigenschaften verwendet werden. Dabei enthalten diese Variablen die folgenden Werte des Schnittpunkts, der bis zum Ausführungszeitpunkt am nächsten zur Kamera gelegen ist.

- *t_min* – Strahlparameter *t*
- *triID* – Dreiecks-ID des Dreiecks, welches geschnitten wurde
- *hit_u* und *hit_t* – uv-Koordinaten innerhalb des Dreiecks *triID*

Die Schnittpunkt-Tests des Sehstrahls erfolgen wie in Algorithmus 1 des Abschnitts 2.1 beschrieben. Die Iteration über die Menge der Dreiecke und die Ausführung der Schnittpunktberechnungen zeigt Listing 12.

```
1  // Iteration über alle Dreiecke der Szene
2  for (int i = 0; i < vertex.length()-2; i+=3)
3  {
4   if(i != lastTriID)
5   {  // Laden der Eckpunkte des Dreiecks #i
6    vec3 v0 = vertex[i].xyz,
7         v1 = vertex[i+1].xyz,
8         v2 = vertex[i+2].xyz;
9             /* Anwendung der Schnittpunkt-Berechnung nach
10             * Purcell et al. */
11    // Schnittpunkt innerhalb des Dreiecks & nächst gelegen?
12    if ((t < t_min)/* && Bedingungen für Schnittpunkt innerhalb
13                    * der Dreiecksfläche */)
14    { // Aktualisierung der zwischengespeicherten Ergebnisse
15     t_min = t;
16     triID = i;
17     hit_u = u; hit_v = v;
18    }
19   }
20  }
```

Listing 12: Schnittpunktsuche innerhalb des *intersect1*-Kernels

Existiert kein Schnittpunkt wird die Hintergrundfarbe der Szene in der Farb-Puffer-Textur eingetragen und der Kernel-Aufruf ist beendet. Wurde ein am nächsten gelegener Schnittpunkt mit einem Dreieck gefunden, beginnt die Auswertung des Schnittpunktes. Dazu wird zu Beginn der Normalenvektor an der Schnittpunkt-Position innerhalb des Dreiecks durch Interpolation der Vertex-Normalen berechnet. Listing 13 zeigt die Interpolation der Normalenvektoren unter Verwendung der genannten Puffer-Variablen.

```
1  if (triID >= 0)
2  {
3   vec3 direction = rd;
4   // Interpolation der Vertex-Normalen des Dreiecks
5   vec4 interpolated_normal =
```

Listing 13: Beispiel für die Vertex-Attribut-Interpolation mit Hilfe der uv-Koordinaten innerhalb einer Dreiecksfläche und der Dreiecks-ID

Zur Bestimmung des Farbwertes des Schnittpunkts wird zuerst geprüft, ob eine Texturierung des Dreiecks besteht. Durch Interpolation der UV-Koordinaten der Eckpunkte wird auf das *sampler2D*-Objekt, das an die benötigte Textur gebunden ist, an der bestimmten Koordinate zugegriffen und der Farbwert ausgelesen. Wird keine Texturierung benötigt, werden die Farben der Dreiecks-Vertices interpoliert. Im Falle einer Reflexion des

Strahls durch die Dreiecksfläche wird die Gewichtung der Schnittpunktfarbe bestimmt und im Alpha-Kanal des Farbvektors gespeichert. Außerdem wird die Strahlrichtung an der Normalen des Schnittpunkts reflektiert (Listing 14).

```
1    // Prüfung, ob Reflexion durch getroffenes Dreieck vorhanden
2    if(refl[triID] > 0.f)
3    {
4     rayState += 1.f;
5     // Bestimmung des Gewichts der Schnittpunkt-Farbe
```

Listing 14: Behandlung bei Reflexion eines Sehstrahls

Die speichernden Zugriffe auf die vorgesehenen Texturen für die ermittelten Werte bilden den Abschluss des *intersect1*-Kernel-Aufrufs. Durch Einsetzen von *t_min* in die Strahlgleichung wird die Schnittpunkt-Koordinate berechnet und als 4D-Vektor mit dem *rayState* als w-Komponente in *rayOriginTexture* an der Position *storePos* gespeichert. Die gegebenenfalls aktualisierte Strahlrichtung und *triID* werden in *rayDirTexture* abgelegt und der bestimmte Farbwert mit seiner Gewichtung im Apha-Kanal in *colorBufferTexture*.

Im Anschluss auf die Schnittpunkt-Berechnungen der Sehstrahlen folgt die Ausführung des *intersect2*-Kernels pro Lichtquelle. Dem Shader-Programm wird dafür der Index der zu verarbeitenden Lichtquelle als *uniform*-Variable übergeben. Im Vergleich zum *intersect1*-Kernel greift ein Shader-Aufruf des *intersect2*-Programms nur auf die Vertex-Positionen und zusätzlich auf die Koordinate der Lichtquelle innerhalb des Licht-Positions-SSBO und die Schattentextur zu. Jeder Thread erzeugt einen Schattenfühler der seinen Ursprung in den Koordinaten des ermittelten Schnittpunkts der *intersect1*-Ausführung besitzt. Den Endpunkt des Strahls bildet die Licht-Position bis zu der er verfolgt und auf Schnittpunkte getestet wird. Eine frühzeitige Abbruchbedingung bildet die Überprüfung des Skalarproduktes des Normalenvektors des Schnittpunkts und des Richtungsvektors des Schattenfühlers. Ist das Ergebnis kleiner oder gleich Null wird automatisch von Schatten ausgegangen, da die Lichtquelle im hinteren Halbraum des Schnittpunkts liegt. Erst wenn diese Bedingung nicht erfüllt ist, beginnt die Schnittpunkt-Suche des Schattenfühlers. Sobald ein Schnittpunkt gefunden ist, bricht die Iteration über die Vertex-Positions-Liste ab und ein Schatten-Flag wird gesetzt (Listing 15).

```
1   for(int i = 0; (i < vertex.length()-2) && (!shadow); i+= 3)
2   { // Verhindern der Selbstverschattung eines Dreiecks
3    if (i != lastTriID)
4    {
5     /* Laden von v0 bis v2 */
6     /* Anwendung der Schnittpunkt-Berechnung nach
7                 * Purcell et al. */
```

```
    // Korrekturfaktor 0.02f um flüssigere Schattenübergänge von
    // direkter Beleuchtung und Beleuchtungsmodell zu erhalten
    if ((t > 0.02f) /* && Bedingungen für Schnittpunkt innerhalb
                     * der Dreiecksfläche */)
     shadow = true;
   }
  }
  // Setzen des Schatten-Flags, falls nötig
  if (shadow)
   switch(lightSource){
    case 0: shadows.x = 1.f; break;
    case 1: shadows.y = 1.f; break;
    case 2: shadows.z = 1.f; break;
    case 3: shadows.w = 1.f; break;
   }
  // Speichern der aktualisierten Schatten-Flag-Belegung
  imageStore(shadowTexture, storePos, shadows);
```

Listing 15: Schnittpunktsuche innerhalb des *intersect2*-Kernels und Setzen des Schatten-Flags

4.1.3 *shading*-Kernel

Dem *shading*-Kernel werden vor seiner Ausführung die Kameraposition und die ambiente Lichtfarbe per *uniform*-Variablen übergeben. Innerhalb der Shader-Ausführung wird zu Beginn die Belegung der Schatten-Flags ausgelesen und evaluiert durch welche der gegebenen Lichtquellen der Farbwert der *colorBufferTexture* beleuchtet wird. Ist kein korelierendes Schatten-Flag einer Lichtquelle gesetzt, kommt es zur Anwendung des gewählten Beleuchtungsmodells und einer Multiplikation des Farbwertes mit der Lichtfarbe der Quelle. Nach Abarbeitung der Lichtquellen und der Bestimmung des temporären Farbwertes kommt es zum Auslesen des Farbwertes der Ausgabetextur und dessen Gewichtung. Beide Farbwerte werden gewichtet aufaddiert und das Gewicht des resultierenden Farbwertes bestimmt. Der Farbwert und sein Gewicht als w-Komponente werden in die Ausgabetextur eingetragen (Listing 16). Auch bei einem *rayState* von 0 wird diese gewichtete Übertragung des Farb-Puffers in die Ausgabetextur ausgeführt, sofern dieser nicht leer ist.
Abschließend kommt es zur Dekrementierung des *rayStates*, falls dieser größer als Null ist, und die Speicheroperationen für die Ausgabefarbe und den *rayState* finden statt.

```
vec4 final_color = imageLoad(finalTexture, storePos);
float weight = final_color.w;  // Extraktion des Gewichts
// Gewichtung des (beleuchteten) Farbwertes und der Ausgabefarbe
final_color = weight * final_color + (1.f-weight) * temp_color;
// Ermittlung des neuen Gewichts des Farbwertes der Ausgabetextur
final_color.w = weight + (1.f-weight) * pColor.w;
```

Listing 16: Gewichtete Addition eines temporären Farbwerts *pColor* und des Ausgabebild-Farbwerts

4.2 Integration des Uniform Grids

Festgelegte Größen des Uniform Grids sind jeweils in den folgenden Kernel-Programmen durch ein *define* enthalten. Listing 17 zeigt eine beispielhafte Definition der Konstanten im Kopf eines Kernel-Programms.

```
#define voxel_amount 40
#define voxel_array_length 512
#define ray_array_length voxel_amount * 3
```

Listing 17: Definition der Uniform Grid-Größen, die bei einem Zugriff des Threads auf die Datenfelder der Datenstruktur benötigt werden

Die Kernel beinhalten weiterhin Hilfsfunktionen zur Identifikation einer Raum-Koordinate mit der beinhaltenden Voxel-Koordinate und eine Abbildungsfunktion von 3D-Voxel-Koordinaten in 1D-Voxel-IDs (siehe Listing 18).

```
/* Abbilung der Raum-Koordinate auf beinhaltende Voxel-Koordinate*/
ivec3 getVoxel(vec3 coords)
{
 // Verschiebung OpenGL-Koordinaten: [-1.0f, 1.0f) -> [0.0f, 2.0f)
 coords += vec3(1.0f);
 // Bestimmung beinhaltenden Voxels durch implizite Gauß-Klammer
 ivec3 voxel = ivec3(coords/vx_edge);
 return voxel;
}
/** Abbildung einer Voxel-Koordinate auf eine Voxel-ID */
int getVoxelID(ivec3 voxel)
{
 if( /* zulässige Voxel-Koordinate */)
  return voxel.z * voxel_amount * voxel_amount
         + voxel.y * voxel_amount + voxel.x;
 else/* unzulässige Voxelkoordinate */
  return -1;
}
```

Listing 18: Hilfsfunktionen zur Identifikation von Voxeln

4.2.1 *build*-Kernel

Die Aufrufe des *build*-Kernels erfolgen pro Dreieck der Szene. Es genügt einen Dispatch des Kernels in einer Dimension der Arbeitsgruppen vorzunehmen, so dass jedes Element der Dreiecksliste durch einen Thread in die räumliche Aufteilung des Rasters eingeordnet wird. Für die Zuordnung von Thread zu Dreieck erfolgt die Abfrage der globalen x-Koordinate

des Threads mit *gl_GlobalInvocationID.x*. Listing 19 zeigt die Kopfzeilen des Kernel-Programms.

```
layout(local_size_x = 128, local_size_y = 1, local_size_z = 1) in;
/* Deklaration des Vertex-Positions-SSBO ausgelassen */
layout(std430, binding = 10) coherent buffer VerticesInVoxel{
 int voxelslist[];
};
layout(std430, binding = 12) coherent buffer ContainedVertices{
 int voxelcontains[];
};
// Länge einer Voxel-Kante
const float vx_edge = 2.0f / float(voxel_amount);
// ID des zu verarbeitenden Dreiecks durch den Thread
const int triID = 3 * int(gl_GlobalInvocationID.x);
```

Listing 19: Kopfzeilen des *build*-Kernels

Für das Befüllen der *voxelslist* und den davon abhängigen Füllstand der Listen innerhalb der Liste *voxelcontains* sind Synchronisations-Maßnahmen erforderlich. Die Buffer-Objekte werden als *coherent* deklariert, um die Speicher-Kohärenz der parallel arbeitenden Arbeitsgruppe bei der Ausführung atomarer Operationen zu gewährleisten und Änderungen des Speicherinhalts durch andere Threads sofort nachzuladen. Listing 20 zeigt die Anwendung der Synchronisation innerhalb einer weiteren Hilfsmethode *storeTriIDInVoxelsList(int voxID)*, die zum Hinzufügen einer Dreiecks-ID in die Liste eines Voxels durch Anwendung atomarer Operationen genutzt wird.

```
void storeTriIDInVoxelsList(int voxID)
{
 // Abfangen unzulässiger VoxelIDs
 if(voxID >= 0)
 {
  int start = voxID * voxel_array_length; // erste Listenposition
  int end = start + voxel_array_length - 1; // letzte Position

  for(int i = start; i <= end; i++)
  {
   int read = atomicCompSwap(voxelslist[i], -1, triID);
   if (read == -1)
   {
    atomicMax(voxelcontains[voxID], i+1-start);
    i = end+1; // Abbruch der Schleife
   }
  }
 }
}
```

Listing 20: Ermittlung einer freien Speicherposition in *voxelslist* und Hinzufügen der ID des verarbeiteten Dreiecks

Der Inhalt der *voxelslist* ist zu Beginn mit dem Wert *-1* initialisiert, so dass bei der Anwendung der Vergleichsoperation *atomicCompSwap* eine Spei-

cherposition, die den initialen Wert beinhaltet, mit der Dreiecks-ID befüllt wird. Kommt es zum Auffinden einer noch nicht belegten Position wird die größte Schreibposition in *voxelcontains* eingetragen.
Zu Beginn der Ausführung des *build*-Kernels werden die Vertex-Positionen der Eckpunkte geladen und die beinhaltenden Voxel der Vertices durch Aufruf der *getVoxel*-Methode bestimmt. Ein Vergleich der einzelnen Komponenten der Voxel-Koordinaten zeigt die Notwendigkeit der Traversierung einer Dreiecksseite gegen das Voxel-Raster mit Hilfe der Algorithmen 2 und 3 auf, wenn ein Eckpunkt in einem anderen Voxel liegt. Für die Traversierung der Seiten wurde eine *void*-Methode *traverseConnection* erstellt, die als Eingabeparameter die beiden Vertex-Koordinaten einer Dreiecksseite und die beinhaltenden Voxel übergeben bekommt. Innerhalb der Methode wird gemäß Algorithmus 2 separat entlang jeder Koordinatenachse traversiert und eine Liste aus Schnittpunkt-Parametern erstellt. Diese Liste wird durch Anwendung des *BubbleSort*-Sortieralgorithmus aufsteigend sortiert. Die Auswertung der besuchten Voxel erfolgt nach Algorithmus 3 inklusive der Behandlung von Grenzfällen (siehe Abbildung 13) und ein Aufruf der *storeTriIDInVoxelsList*-Methode erfolgt pro besuchten Voxel.

4.2.2 *traversal*-Kernel

Die Vorgehensweise bei der Traversierung eines Strahls während der Ausführung eines *traversal*-Kernels entspricht der Traversierung einer Seite des Dreiecks innerhalb des *build*-Kernels. Allerdings sind bei den Speicherzugriffen der *traversal*-Kernel keine Synchronisationsoperationen notwendig. Jeder Thread verarbeitet genau einen Strahl, der eine eigene Liste an durchquerten Voxeln besitzt, wodurch es zu keinen externen Zugriff auf diesen Speicherabschnitt kommt. Bei der Abarbeitung der erstellten und sortierten Parameterliste der Voxel-Ebenen-Schnittpunkte wird für jeden Voxel in der Reihenfolge der Sortierung eine *storeVoxelID*-Funktion aufgerufen. Bevor es allerdings zum Aufruf der Funktion kommt, wird mit Hilfe des *voxelcontains*-Datenfeldes geprüft, ob es sich um einen leeren Voxel handelt. Im Falle eines leeren Voxels wird dieser übersprungen und es kommt nicht zu dem Aufruf der genannten Funktion. Listing 21 zeigt die *storeVoxel*-Funktion. Die globale Zählervariable *voxelListCounter* beinhaltet die Anzahl der bereits eingetragenen Voxel-IDs in die Liste des Strahls. Durch Verschiebung der Startposition um diesen Zählerstand ist ein effizientes und lückenloses Befüllen der Liste möglich.
Am Ende des Kernel-Aufrufs wird der Zählerstand in der zu dem Strahl gehörigen Position innerhalb des *raycontains*-Datenfeldes eingetragen.

```
void storeVoxelID(int voxID)
{
 if(voxID >= 0)
 {
  // Abfangen eines Überlaufs der Liste
  if (voxelListCounter <= ray_array_length)
  {
   // Verschieben eines Zeigers um Füllstand der Liste
   int i = start + voxelListCounter;
   rayslist[i] = voxID;
  }
  voxelListCounter++;
 }
}
```

Listing 21: Eintragen einer Voxel-ID durch Aufruf der *storeVoxelID*-Funktion

4.2.3 Erweiterung der *intersection*-Kernel

Bei Erweiterung der *intersection*-Kernel handelt es sich um die Integration der durch *build*- und *traversal*-Kernel erstellten Listen und die Interpretation der Listeninhalte, die zur Bestimmung der Menge der auf Schnittpunkte zu testenden Dreiecke genutzt wird.
Es findet eine Iteration über die Liste der geschnittenen Voxel des Strahls statt und dabei ein Auslesen der Voxel-IDs. Mit Hilfe einer Voxel-ID wird über die Liste der beinhalteten Dreiecke des Voxels iteriert und jedes dieser Dreiecke auf einen Schnittpunkt mit dem Strahl getestet. Sobald es innerhalb eines Voxels zu einem Schnittpunkt gekommen ist, kann der verbleibende Rest der Voxel-IDs verworfen werden, da der Strahl von seine Ursprung aus traversiert wurde und die Reihenfolge der Voxel beim Speichern beachtet wurde. Der detaillierte Ablauf der Iterationen findet sich in Listing 22.

```
int start = threadID * ray_array_length;
int voxellist_last = start + raycontains[threadID];

for(int j = start;  j <= voxellist_last; j++)
{
 int voxID = rayslist[j];
 if (voxID >= 0)
 {
  int start_vertex = voxID * voxel_array_length;
  int vertexlist_last = start_vertex + voxelcontains[voxID];

  for(int i = start_vertex; i <= vertexlist_last; i++)
  {
   // current vertex handled
   int triID = voxelslist[i];
   // reached end of vertexList?
   if (vID >= 0)
   {
    /* Laden der Eckpunkte des Dreiecks#triID
              /* Anwendung der Schnittpunkt-Berechnung nach
               * Purcell et al. */
    /* Speichern der Schnittpunkt-Eigenschaften */
   }
   else
    i = vertexlist_last;
  }
 }
}
```

Listing 22: Traversierung der Uniform Grid-Datenstruktur innerhalb des *intersect1*-Kernels

5 Evaluation

Im Rahmen der Evaluation des entwickelten GPGPU-Ray-Tracers wurden Benchmarks zur Prüfung der gegebenen Performanz durchgeführt. Dabei wurde die Anzahl der generierten Bilder pro Sekunde (*frames per second*, kurz: FPS) und die Anzahl der getätigten Schnittpunkt-Tests pro Bildaufbau gemessen.
Bei dem zugrunde liegenden Testsystem handelt es sich um einen Desktop-Rechensystem mit *64-bit Windows 8.1*, einer *Intel Core i7-4770*-CPU mit jeweils 3.4GHz und *NVIDIA GeForce GTX 780*-GPU mit 2304 Shader-Einheiten und 3GB Speicher.
In Abschnitt 5.1 werden die Ergebnisse der Benchmark-Durchläufe zur Abhängigkeit der Performanz des Basis-Ray-Tracers bezüglich der Anzahl der darzustellenden Dreiecke vorgestellt und analysiert. Der darauf folgende Abschnitt 5.2 zeigt die Notwendigkeit einer Abstimmung der Uniform Grid-Auflösung an die Szene auf. In Abschnitt 5.3 kommt es schließlich zu dem Vergleich zwischen dem Basis-Ray-Tracer und der beschleunigten Variante des Ray-Tracers mit Hilfe einer komplexeren Szene.
Die gewählte Umsetzung des Ray-Tracers ermöglicht eine Unterscheidung von 71582788 verschiedener positiver Dreiecks-IDs, basierend auf der Nutzung von 32 Bit *integer*-Speicherpositionen. Durch Verwendung des Datentyps *unsigned integer* ist eine Erhöhung der möglichen Dreiecke möglich. Des Weiteren wurden zur Vereinfachung der Umsetzung maximal vier Lichtquellen unterstützt, aufgrund der Verwendung der Schattentextur mit vier Farbkanälen. Ein bitweises Setzen der Schatten-Flags innerhalb der Schattentextur würde die Anzahl der möglichen Lichtquellen ebenfalls erhöhen.

5.1 Einfluss der Anzahl der Dreiecke

Für das Benchmark des Einflusses der Anzahl an Szene-Dreiecken wurden die drei folgenden Objekt-Konstellationen verwendet, wobei eine Bodenfläche mit inbegriffen ist.

14592 Dreiecke: der Utah-Teapot

27392 Dreiecke: drei Utah-Teapots

46592 Dreiecke: sechs Utah-Teapots

Zusätzlich zu den Szene-Objekten wurde eine Lichtquelle und eine Reflexionstiefe von eins verwendet. Jeder der dargestellten Utah-Teapots besitzt eine reflektierende Oberfläche. Es wird eine Drehung der Szene um 360° ausgeführt, wobei ein Bild für jeden Betrachtungswinkel erzeugt wurde. Abildung 19 zeigt die Anzahl der generierten Bilder pro Sekunde in Abhängigkeit der darzustellenden Objekte. Der Zusammenhang zwischen der

Anzahl der Szene-Dreiecke und der FPS-Zahl ist antiproportional. Man erkennt außerdem, dass es bei höherer Zahl an Dreiecken zu erheblichen Schwankungen in der Anzahl der generierten Bilder pro Sekunde kommt. Das in Abbildung 20 enthaltene Diagramm zeigt die Anzahl der Schnitt-

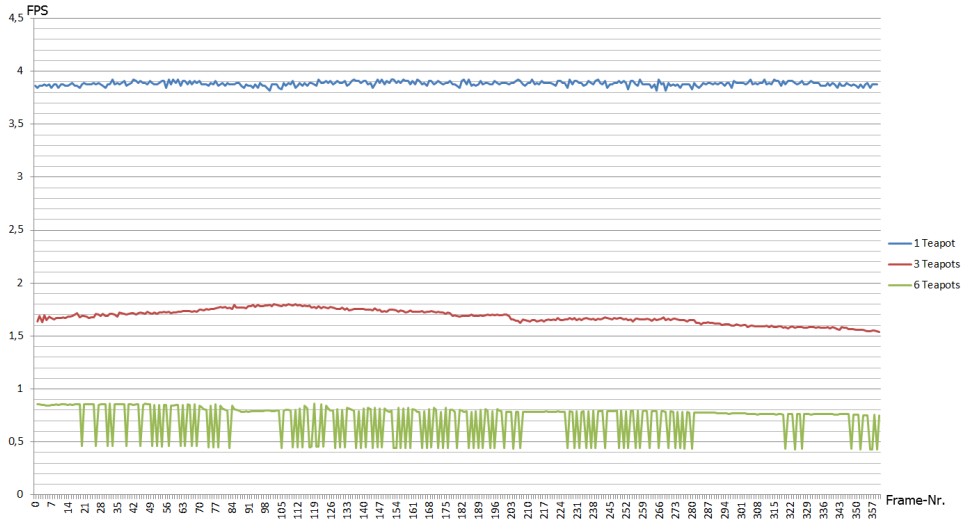


Abbildung 19: Verlauf der Bildrate beim Rendering der aufgezeigten Testszenen mit dem Basis-Ray-Tracer

punkt-Berechnungen, die während des Durchlaufs der Testszene getätigt werden. Es ist ersichtlich, dass es bei einer höheren Zahl an Dreiecken phasenweise zu deutlich mehr getätigten Schnittpunktberechnungen kommt. Die lokalen Minima der Graphen korrelieren mit der Fläche der Schatten, bei der für jedes Pixel die Verfolgung des Schattenfühler-Strahls bei dem ersten gefundenen Schnittpunkt mit einem Szene-Objekt abgebrochen werden kann. Kleinere lokale Maxima sind auf die Verfolgung von Mehrfach-Reflexionen zurück zu führen, die vor allem bei der Testszene mit den sechs Utah-Teapots vermehrt auftreten.

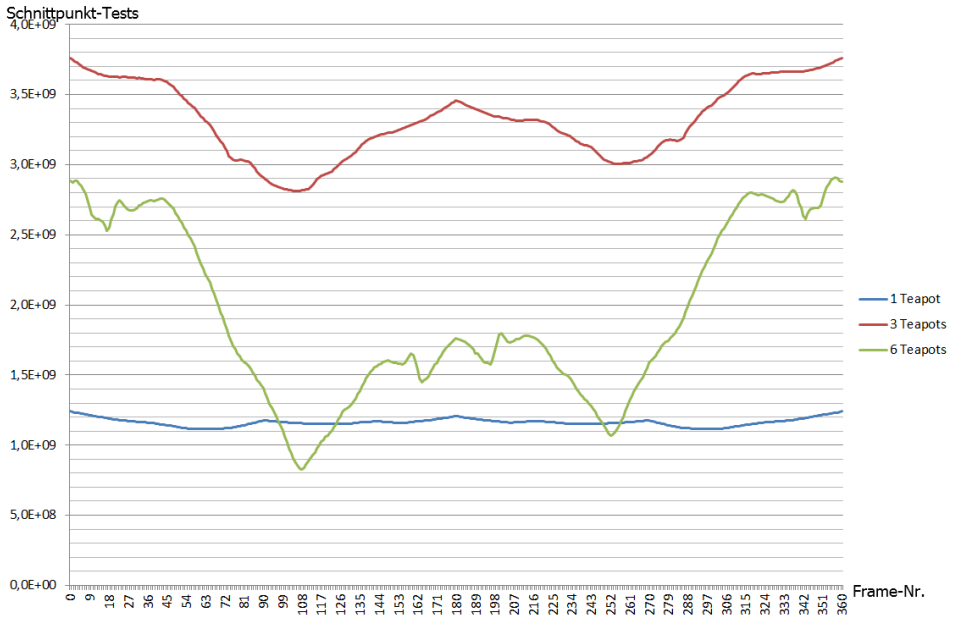


Abbildung 20: Anzahl der Schnittpunkt-Berechnungen, die beim Aufbau eines Bildes durch den Basis-Ray-Tracers erfolgen

5.2 Einfluss der Auflösung des Uniform Grids

Die Performanz des Ray-Tracers mit der Nutzung des Uniform Grids ist je nach Verteilung der Dreiecke innerhalb der Szene abhängig von der gewählten Rasterauflösung. Am Beispiel der Testszene mit sechs beinhalteten Utah-Teapots aus Abschnitt 5.1 wurden Benchmarking-Durchläufe mit verschiedenen Uniform Grid-Auflösungen durchgeführt. Die aus der gewählten Auflösung resultierenden FPS-Zahlen sind in Abbildung 21 dargestellt. Es wird deutlich, dass ein Optimum der Auflösung des Rasters für die verwendete Szene im Bereich zwischen 12^3 und 20^3 verwendeten Voxeln liegt. Die getätigten Schnittpunkt-Berechnungen für eine gewählte Auflösung zeigt Abbildung 22. Obwohl die Anzahl der Schnittpunkt-Tests mit zunehmender Feinheit des Rasters sinkt, sinkt auch die Performanz in Bezug auf die Anzahl der generierten Bilder pro Sekunde. Dies resultiert aus den zusätzlichen Berechnungen, die bei der Einordnung der Dreiecke in das Raster und der Traversierung des Uniform Grids für die Strahlen pro Frame benötigt werden. Einen limitierenden Faktor stellt auch die Größe des zu allokierenden Speichers für die Listen der Dreiecke der Voxel dar. Jeder Listenplatz belegt dabei einen Speicher von 32 Bit. Multipliziert mit der Listenlänge und der Anzahl der Voxel, die aus der Rasterauflösung resultiert, wächst der belegte Speicher im Verhältnis zu der Erhöhung der Auflösung um den Faktor 1 kubisch an.

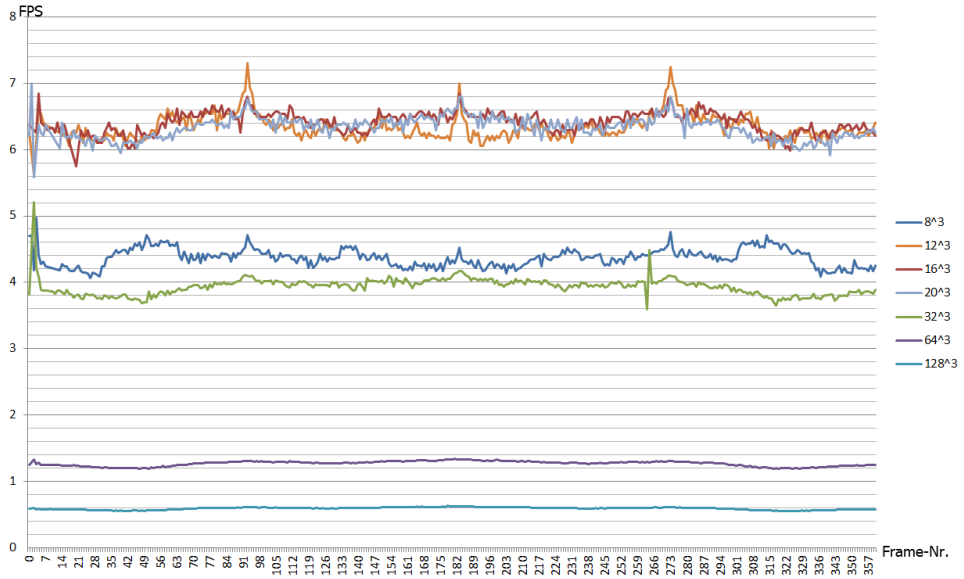


Abbildung 21: Abhängigkeit der Performanz des beschleunigten Ray-Tracers von der Wahl der Uniform Grid-Auflösung

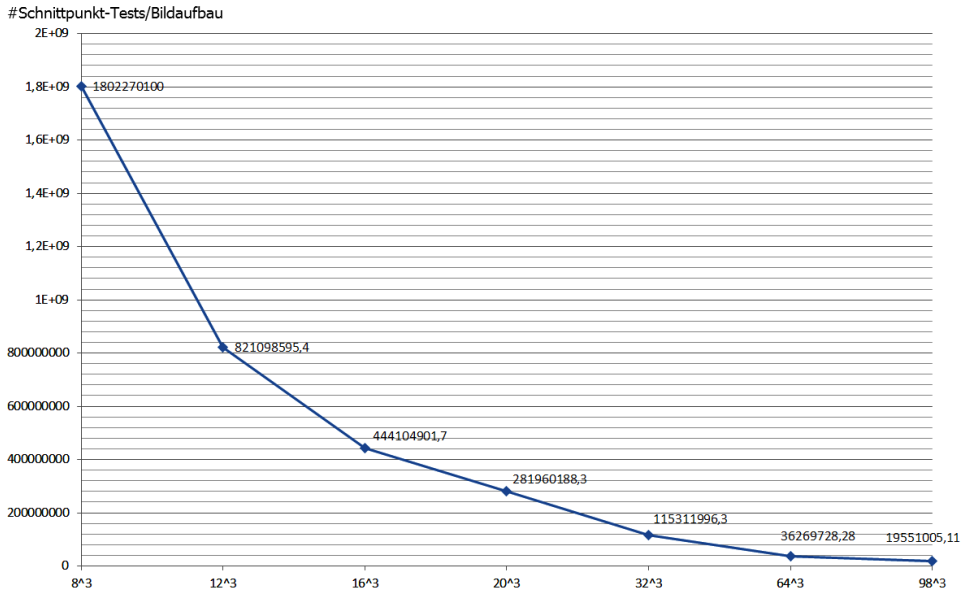


Abbildung 22: gemittelte Anzahl der benötigten Schnittpunkt-Tests zum Aufbau eines Bildes bei unterschiedlich aufgelösten Uniform Grids

5.3 Vergleich der beiden Ray-Tracer

Für den Vergleich des Basis-Ray-Tracers mit der durch das Uniform Grid erweiterten Variante wurde das Modell des Stanford-Hasen verwendet.

Das für das Benchmarking verwendete Modell besteht aus 69451 Dreiecken. Die Szene beinhaltet eine Lichtquelle und Reflexionen kamen nicht zum Einsatz. Abbildung 23 zeigt die gemessenen FPS-Zahlen für verschiedene Rasterauflösungen bei Verwendung des durch das Uniform Grid erweiterten Ray-Tracers und die des Basis-Ray-Tracers. Der Ray-Tracer mit

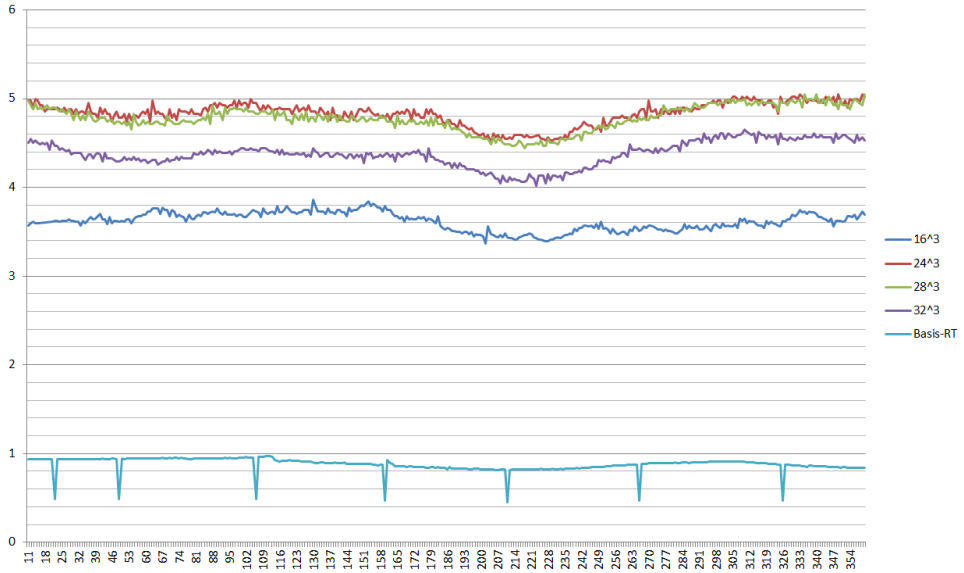


Abbildung 23: Gemessene Anzahl an Bildern pro Sekunde während des Renderings eines skalierten Modells des Stanford-Hasen

dem Uniform Grid der Auflösung 24^3 erreicht dabei eine durchschnittliche FPS-Zahl von 4.8339, was im Vergleich zu dem durchschnittlichen FPS-Wert von 0.8804 des Basis-Ray-Tracers einer Beschleunigung um den Faktor 5.5 entspricht.
Betrachtet man die Anzahl der stattgefundenen Schnittpunkt-Tests – dargestellt in Abbildung 24 – wird deutlich, dass die zu verrichtende Arbeit in Form von Schnittpunkt-Tests bei der Aufteilung des Raums durch ein Uniform Grid enorm gesenkt werden kann. Während der unbeschleunigte Basis-Ray-Tracer im Mittel $1\,137\,720\,816$ Schnittpunkt-Tests für den Aufbau eines Bildes benötigt, sind es bei einem 24^3-aufgelösten Raster mit $287\,728\,727.1$ im Mittel nur ein Viertel der Tests. Es wurde auch versucht ein aus 204088 Dreiecken aufgebautes skaliertes Modell des Stanford-Drachen in einer Szene, die außerdem noch eine Lichtquelle und einen reflektierende Bodenfläche beinhaltet, mit dem Basis-Ray-Tracer zu rendern. Allerdings kam es aufgrund der hohen Rechenzeiten zu einem *time out* des Grafiktreibers durch das Betriebssystem, so dass das unbeschleunigte Rendering nicht möglich war. Die beschleunigte Variante des Ray-Tracers war in der Lage die Szene zu rendern und erzielte dabei im Durchschnitt $1,396$ FPS bei einer Uniform Grid-Auflösung von 28^3 Voxeln.

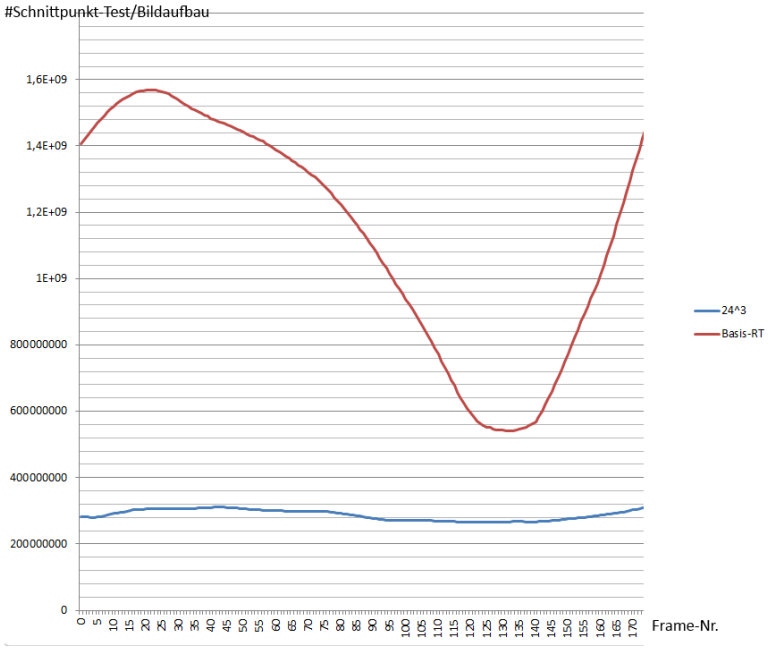


Abbildung 24: Anzahl der Schnittpunkt-Tests, die beim Rendern eines Bildes des Stanford-Hasen, benötigt werden

6 Fazit

Die theoretische Umsetzung eines GPGPU-Ray-Tracers unter Verwendung der Compute-Shader von OpenGL wurde erarbeitet und anhand der aufgezeigten Implementierung funktionsfähig implementiert.

Die Auswertung der Benchmarking-Ergebnisse zeigt deutlich, dass eine erhebliche Beschleunigung des Ray-Tracers durch die Integration des Uniform Grids zu erzielen ist. Allerdings wird für die Annäherung an eine optimale Performanz die erwähnte Anpassung der Rasterauflösung an die gegebene Szene benötigt. Die Verwendung des Uniform Grids ermöglicht die Darstellung komplexer Szenen auch auf schwächeren Systemen, auf denen die Ausführung der unbeschleunigten Variante des Ray-Tracers aufgrund von Treiberbegrenzungen zu keinem Rendering-Ergebnis führt.

Einen möglichen limitierenden Faktor stellt der verfügbare GPU-Speicher dar. Die auf der GPU zu erstellende Datenstruktur zum Aufbau des Uniform Grids wächst mit der Verfeinerung des verwendeten Voxelrasters sehr schnell an. Die aktuellen Entwicklungen moderner GPUs zeigen einen Trend zu noch mehr verfügbarem Grafikspeicher, so dass Datenstrukturen in dieser Größenordnung problemlos im Speicher der GPU abgelegt werden können.

Eine Erweiterung des entwickelten Ansatzes des GPGPU-Ray-Tracers ist durch die Erzeugung von zusätzlichen Sekundärstrahlen möglich. Noch realistischere Ergebnisse lassen sich beispielsweise durch die Verwendung der Refraktion von Licht an bestimmten Materialoberflächen erzielen. Außerdem wäre es möglich eine globale Beleuchtung umzusetzen, in dem weitere Strahlen von den Schnittpunkten aus in den Raum versendet werden, um die Umgebungshelligkeit abzutasten.

Die Anzahl an Shader-Prozessorkernen, die in modernen GPUs enthalten sind, wächst ebenfalls an. Dadurch werden Shader-Programme stärker parallelisierbar und die GPU wird für die Ausführung von noch komplexeren GPGPU-Anwendungen attraktiv.

Literatur

[1] Brian J Ross. Cosc 3p98: Ray tracing basics. 2014.

[2] Timothy J Purcell, Ian Buck, William R Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 703–712. ACM, 2002.

[3] W Akman, Wm Randolph Franklin, Mohan Kankanhalli, and Chandrasekhar Narayanaswami. Geometric computing and uniform grid technique. *Computer-Aided Design*, 21(7):410–420, 1989.

[4] Enhua Wu and Youquan Liu. Emerging technology about gpgpu. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pages 618–622. IEEE, 2008.

[5] Khronos OpenCL Working Group et al. The opencl specification. *version*, 1(29):8, 2008.

[6] Mark Segal and Kurt Akeley. The opengl graphics system: A specification version 4.3. Technical report, Technical report, OpenGL. org, 2012.

[7] Khronos Group. OpenGL Wiki. `http://opengl.org/wiki`. Zugriff: 14.04.2015.

[8] John Kessenich, Dave Baldwin, and Randi Rost. The opengl shading language. *Language version*, 1, 2004.