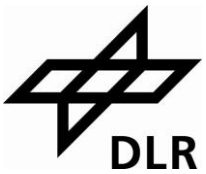

Entwicklung eines Systems für verteiltes, paralleles Echtzeit-Raytracing

BACHELORARBEIT

für die Prüfung zum

Bachelor of Engineering

des Studiengangs Informationstechnik

an der Dualen Hochschule Baden-Württemberg Mannheim

von

Frieder Berthold

23.09.2013

Bearbeitungszeitraum	12 Wochen
Matrikelnummer, Kurs	8582141, TIT10AIN
Ausbildungsfirma	DLR, Braunschweig
Betreuer der Ausbildungsfirma	Dr. Robin Wolff
Gutachter der Dualen Hochschule	Prof. Dr. Rainer Colgen

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass ich
die vorliegende Arbeit selbstständig und nur unter
Verwendung der angegebenen Quellen und
Hilfsmittel angefertigt habe.

Braunschweig, der 20. September 2013

Kurzfassung

Das Projekt „Virtual Reality for On-Orbit Servicing“ hat die Simulation von Wartungsarbeiten an Satelliten im Erdorbit als Ziel. Dafür werden Methoden der Virtuellen Realität eingesetzt. Für die Visualisierung existiert ein Raytracer, der auf einem einzelnen Computer läuft. Dieser erreichte aber nicht die nötigen interaktiven Bildwiederholraten von 30 Bildern pro Sekunde bei HD-Auflösung.
Da jedoch solche Bildraten nötig sind, um als Benutzer sinnvoll mit der Simulation interagieren zu können, wird in der vorliegenden Arbeit untersucht, ob der benutzte Raytracer auch in einem Clustersystem zum Einsatz kommen kann, um Ausschnitte des Bildes parallel zu berechnen und die Rechenzeit zu reduzieren. Die Software muss dafür jedoch angepasst und erweitert werden: Für die Benutzung eines Clusters wird die Software in ein Backend und ein Frontend aufgeteilt. Im Backend wird eine Master-Slave-Architektur eingesetzt. Der bisherige Raytracer wird für ein verteiltes, paralleles Berechnen der einzelnen Bilder in die Slaves integriert. Außerdem müssen geeignete Algorithmen für die Aufteilung des Bildes entwickelt und implementiert werden. Abschließend muss die Möglichkeit geschaffen werden, die großen Datenmengen, welche durch die HD-Bilder entstehen, zu einem anzeigenden Computer effektiv zu transportieren.
Der entwickelte verteilte, parallele Raytracer benutzt einen dynamischen Bildaufteiler, um die Bilder so an die einzelnen Raytracerprozesse zu verteilen, dass deren Berechnungszyklen annähernd ausgeglichen sind. Die Synchronisation auf dem Cluster wird mit MPI realisiert. Für die Komprimierung der Bilder und dem effizienten Senden an das Frontend wird ein H.264 Encoder eingesetzt.
Die Bildwiederholraten konnten im Verhältnis zum vorherigen einzelnen Raytracer in HD-Auflösung fast vervierfacht werden. Durch die aufgezeigten Optimierungsmöglichkeiten sind auch interaktive Bildraten in absehbarer Zeit zu erreichen.

Abstract

The project „Virtual Reality for On-Orbit Servicing“ aims to develope a simulation for maintenance of satellites in space. To reach this goal virtual reality methods are applied. The visualization of this simulation uses an existing raytracer which runs on a single computer. Albeit this raytracer can not reach the framerates of 30 frames per second in HD resolution which are needed for an interactive application. Since those framerates are essential to interact with the simulation as a user, this paper researches the idea whether the used raytracer can be evolved to a distributed, parallel application running on a computer cluster. Thus rendering parts of each frame in parallel and reducing the time needed for each frame.

For using a cluster the software needs to be devided into a backend and frontend. The backend is designed using the master/slave architecture. The existing raytracer has been integrated into the slave to render each frame in parallel. To partition the frame, suitable algorithms have to be developed and implemented. Additionally the computed frames have to be send effectively to the frontend.

The developed distributed, parallel Raytracer applies a dynamic partition of each frame to level the compute times of each slave. Synchronization on the cluster is achieved by using MPI. To compress the frames and send them efficiently to the frontend a H.264 encoder is used.

The framerates could be quadrupled for HD resolution compared the former stand alone raytracer. The proposed optimization measures allow interactive framerates using ratracing in the foreseeable future.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

CPU	**C**entral **P**rocessing **U**nit
DLR	**D**eutsches Zentrum für **L**uft- und **R**aumfahrt
FPS	**F**rames **P**er **S**econd
GPU	**G**raphics **P**rocessing **U**nit
LAN	**L**ocal **A**rea **N**etwork
MPI	**M**essage **P**assing **I**nterface
SRV	**S**oftware für **R**aumfahrtsysteme und interaktive **V**isualisierung
TCP	**T**ranmission **C**ontrol **P**rotocol
UDP	**U**ser **D**atagram **P**rotocol
VR	**V**irtuelle **R**ealität
VR-OOS	**V**irtual **R**eality for **O**n-**O**rbit **S**ervicing

1 Einleitung

In der Abteilung „Software für Raumfahrtsysteme und interaktive Visualisierung“(SRV) der Einrichtung „Simulations- und Softwaretechnik“ des Deutschen Zentrum für Luft- und Raumfahrt(DLR) wird an dem Projekt Virtual Reality for On-Orbit Servicing(VR-OOS) gearbeitet. Dieses Projekt hat das Ziel, Simulationen von Wartungsarbeiten an Satelliten, welche bereits im Erdorbit sind, durchzuführen. Um dieses Ziel zu erreichen werden Methoden der Virtuellen Realität(VR) benutzt. Die Visualisierung dieser VR soll dafür so realistisch wie möglich sein. Dies kann spätere Irritationen beim Wechsel von der Visualisierung in der VR auf echte Kamerabilder aus dem Weltall vermeiden. Außerdem ermöglicht eine Visualisierung die möglichst nah an der Realität ist den Einsatz von virtuellen Kameras, mit deren Hilfe z.B. optische Bilderkennungsalgorithmen vor der realen Anwendung getestet werden können.

1.1 Motivation

Zur Zeit wird in VR-OOS der Ansatz des Shadings als Visualisierungsmethode genutzt. Bei dieser Methode gehen die Algorithmen jeweils von den einzelnen Punkten einer Geometrie in der Szene aus und berechnen dessen Farbe. Danach muss das gesamte Bild noch rasterisiert werden, um die Farben aller auf einem Bildschirm dargestellten Pixel zu berechnen. Der lokale Ansatz macht es jedoch schwer, alle natürlich sichtbaren Phänomene, wie z.B. Schatten oder Reflektion, zu berechnen. Und beim Rasterisieren entstehen Interpolationsfehler. Deswegen wird im Projekt VR-OOS auch ein anderer Ansatz der Visualisierung untersucht: das Raytracing. Dabei werden Strahlen in der Szene verfolgt und auf einen Schnittpunkt mit einer Geometrie untersucht.. Phänomene der Optik, wie z.B. Brechung und Absorption, werden dann durch Algorithmen abgebildet. Dadurch entsteht ein globaler Ansatz der Bildberechnung und die Farbe jedes Pixels wird direkt berechnet. In vorherigen Arbeiten des Autors mit dem Framework OptiX von Nvidia wurde gezeigt, dass dieser Ansatz auch grundsätzlich funktioniert.[3]

1.2 Herausforderungen

Der Nachteil des Raytracens ist der hohe Rechenaufwand. Da nicht einmal durch die Eckpunkte der Geometrien durchiteriert werden muss, sondern jeder zu untersuchende Strahl auf einen Schnittpunkt mit allen Geometrien getestet werden muss, sind viel mehr Rechenoperationen nötig. Außerdem wird dies noch durch neue Strahlen, welche bei Reflektion, Brechung etc. entstehen, verstärkt. Dadurch ist es nicht möglich, so hohe Bildwiederholraten zu ermöglichen wie beim Shading. Geringe Raten stehen jedoch im Gegensatz zur zentralen Anforderung von VR, welche hohe Bildwiederholraten von wenigstens 30 Bildern pro Sekunde braucht, um interaktiv zu sein. Aus diesem Grund soll eine Möglichkeit geschaffen werden, mit dem vorhandenen Raytracer genügend hohe Bildwiederholraten zu erreichen. Als untersuchenswerter Ansatz soll dieser parallel auf einem Rechencluster laufen und so die erhöhte Rechenkapazität ausnutzen. Jedes Bild wird dann durch mehrere Grafikkarten berechnet. Dafür müssen die benötigten Daten verteilt und das Bild aufgeteilt werden. Anschließend müssen die einzelnen Bildteile eingesammelt und zu einem Gesamtbild zusammengesetzt werden. Für diese Aufgaben wird ein passendes Framework benötigt.

1.3 Ziele

Die Entwicklung eines Frameworks zum verteilten, parallelen Echtzeit-Raytracens teilt sich in mehrere Aspekte auf. Der bisherige Raytracer auf Basis von OptiX hat sich bewährt. Deshalb wird er weiterhin benutzt, muss jedoch so aufgeteilt werden, dass die Berechnungen auf der verteilten Hardware eines Clusters laufen. Dafür ist es wichtig, dass es eine Möglichkeit zur Synchronisierung der einzelnen Raytracing-Prozesse gibt. Durch diese Aufteilung darf jedoch nicht die Integration in das VR-OOS Projekt verloren gehen. Außerdem müssen die berechneten Bilddaten der anzeigenden Hardware zur Verfügung gestellt werden. Dafür eignet sich dann jede Workstation mit einem Monitor.
Der Cluster, welcher der Abteilung SRV zur Verfügung steht, benutzt als Netzwerk

zur Kommunikation zwischen den einzelnen Computern InfiniBand. Dieses ist im Bereich der Hochleistungsrechner weit verbreitet, da es besonders hohe Datenraten ermöglicht. Um dieses zu nutzen eignet sich das Message Passing Interface(MPI). Der letzte Aspekt, welcher untersucht wird, ist die effiziente Übertragung der berechneten Bilder zu einem Frontend, um es auf diesem anzuzeigen. Aufgrund der hohen Datenmengen, welche gerade im hochauflösenden Bereich auftreten, werden geeignete Video-Encoder untersucht.

1.4 Übersicht über die Arbeit

Diese Arbeit teilt sich in vier Hauptkapitel und der Zusammenfassung und dem Ausblick auf. Zuerst werden im Kapitel 2 das Projekt VR-OOS und technische Hintergründe erklärt. Im Kapitel 3 wird dann das umzusetzende Systemdesign des verteilten, parallelen Raytracers gezeigt und erläutert. Auf die Umsetzung des Designs sowie spezielle Aspekte der Implemententierung wird in Kapitel 4 eingegangen. Abschließend werden die Tests des entwickelten Raytracers in Kapitel 5 gezeigt. Bei diesen geht es um die Gesamtleistung des Systems sowie die Effizienz der entwickelten Bildaufteiler.

2 Technische Grundlagen

In den folgenden Kapiteln werden die einzelnen, technisch für diese Arbeit relevanten Aspekte erläutert. Dazu soll im Abschnitt 2.1 zuerst das Raytracing erklärt werden um dann im Abschnitt 2.2 auf das Projekt VR-OOS und den Stand des bisherigen Raytracers einzugehen. Anschließend werden die Rahmenbedingungen durch den vorgegebenen Cluster aufgezeigt. Daraus resultierend wird im Abschnitt 2.4 die Kommunikationsmöglichkeit mit MPI im Cluster und im Abschnitt 2.5 das Streaming zwischen Cluster und Frontend. Um die letztgenannte Kommunikation zu verbessern müssen die Bilder komprimiert werden, was im Abschnitt 2.6 kurz vorgestellt wird.

2.1 Raytracing

Das Hauptziel ist die Entwicklung eines verteilten und parallelen Raytracers. Um dies zu erreichen, ist es wichtig zu wissen, was ein Raytracer ist und wieso er sich für die Verteilung und Parallelisierung eignet. Der Ansatz des Raytracens soll deshalb in diesem Kapitel in seinen wesentlichen Konzepten vorgestellt werden.
Beim Raytracen wird versucht, die physikalischen Gesetzmäßigkeiten nachzumodellieren, welche für die Ausbreitung von Licht gelten. Dadurch ist es möglich, am Computer Bilder zu generieren, die realistischer erscheinen, als andere Berechnungsmethoden. Das menschliche Auge fängt Lichtstrahlen auf, welche von Lichtquellen ausgehen und dann von Objekten reflektiert, gebrochen, oder teilweise durchgelassen werden. Die Strahlen können natürlich auch direkt ins Auge fallen. Wenn ein Objekt von keinem Lichtstrahl getroffen wird, welcher danach in unser Auge fällt, dann können wir dieses Objekt nicht sehen. Dieses Verhalten von Lichtstrahlen und deren Interaktion mit Objekten kann algorithmisch dargestellt und somit auch berechnet werden. Dies für alle möglichen Lichtstrahlen der Lichtquellen einer Szene zu berechnen ist jedoch sehr aufwendig, sodass der umgekehrte Weg beschritten wird: Wie in Abbildung 1 gezeigt wird, werden die Strahlen vom Punkt des Betrachters aus ausgesandt. Sobald ein Objekt getroffen wird, muss ein

weiterer Strahl in Richtung der Lichtquellen gestartet werden, um zu ermitteln, ob der Punkt beleuchtet ist. Die dargestellte Szene ist dabei besonders einfach. Für die Berechnung von Brechung, Transparenz oder Reflektion sowie weiterer Lichtquellen müsste jeweils ein weiterer Strahl berechnet werden. Beim Verfolgen der Strahlen, welche aufgrund der ersten drei Phänomene ausgesendet werden, können Objekte getroffen werden, deren Aussehen ebenfalls berechnet werden muss. Der Algorithmus ist also rekursiv.
Daraus ergibt sich die Notwendigkeit, für jeden anzuzeigenden Pixel eines Bildes ein Strahl zu starten, um diesen Pixel zu berechnen. Diese sind jedoch unabhängig voneinander. Der Vorteil liegt somit auf der Hand: Die einzelnen Pixel können auf unabhängigen Grafikkarten berechnet werden und anschließend müssen sie nur zu einem Bild zusammengesetzt werden. Ein Raytracer eignet sich also für eine verteilte und parallele Verarbeitung.

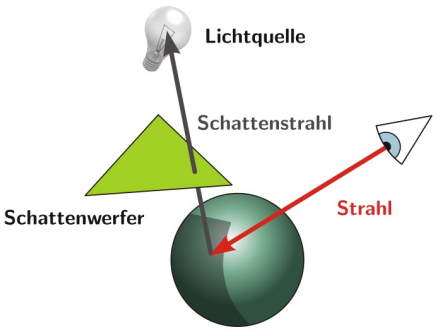

Abbildung 1 – Strahlverfolgung im Raum[1]

2.2 Virtual Reality for On-Orbit Servicing

Das zu entwickelnde System soll in das Softwareprojekt Virtual Reality for On-Orbit Servicing (VR-OOS) eingebunden werden. Aus diesem Grund werden in den beiden folgenden Kapiteln zuerst die Struktur und Aufgabe von VR-OOS vorgestellt und dann der Stand des bisherigen Raytracer-Visualisierungs-Moduls erklärt.

2.2.1 VR-OOS Software Architektur

Das Ziel von VR-OOS ist es, Kosten und Risiken in der staatlichen und zivilen Raumfahrt zu minimieren. Die bisherige Vorgehensweise beinhaltete die eingeplante Abschaltung von Satelliten, nachdem entweder Betriebsmittel, wie z.B. Brenn- oder Kühlstoffe, ausgehen oder einzelne Teile nicht mehr funktionsfähig sind. Der Grund dafür liegt an den großen Risiken und Kosten, die entstehen, wenn Menschen zu den Satelliten fliegen, um sie zu reparieren. Die Kosten und Risiken können minimiert werden, wenn Roboter eingesetzt werden. Diese müssen jedoch aufgrund der Komplexität der Reparaturen von Menschen gesteuert werden. Genau an der Stelle setzt das Projekt VR-OOS an: Es will eine Umgebung bereitstellen, in der der Benutzer über verschiedene Eingabegeräte mit einer Virtuellen Realität (VR) interagiert. In dieser soll z.B. der gewünschte Satellit, aber auch komplexere Szenen, dargestellt werden, an welchen die Reparatur mittels Roboter trainiert werden kann. Dabei wird auch der Roboter simuliert. Wenn es dann zum tatsächlichen Einsatz kommt, ist es wünschenswert, dass der Mensch sich nicht an eine andere Steuerumgebung gewöhnen muss. Aus diesem Grund soll das VR-OOS Projekt in der Lage sein, zwischen VR und Realität hin- und herzuschalten. Die Steuerbefehle müssen dann zum Roboter gesendet werden und Aufnahmen von Kameras der Roboter müssen dem Menschen angezeigt werden. Aus diesem Grund ist auch eine möglichst realistische Visualisierung der Simulation wichtig, um Menschen so wenig wie möglich zu iritieren, wenn er das reale Bild sieht, da dies zu Fehlern führen könnte.
Wie bereits aus dieser Beschreibung klar wird, muss VR-OOS drei verschiedene Aufgaben bewältigen: Es muss in der Lage sein, die Signale von verschiedenen Eingabegeräten zu verarbeiten. Dann muss die physikalische Simulation der Objekte in der darzustellenden Szene erfolgen. Und zum Schluss muss die Szene visualisiert werden. Diese Aufgaben können zum einen klar voneinander abegrenzt werden und zum anderen erfordern sie unterschiedliche Hardware. Die Signalverarbeitung bei Eingabegeräten ohne haptische Rückkopplung benötigt wenig Rechenleistung. Eingabegeräte mit haptischer Rückkopplung stellen jedoch wegen der benötigten

Kollisionserkennung mit Wiederholraten von mindestens 1kHz hohe Anforderungen an die Rechenleistung. Auch die Simulation und Visualsisierung sind rechenintensiv. Des weiteren bestehen große VR-Anlagen zumeist aus einem Verbund aus Grafikrechnern. Diese Überlegungen führten dazu, dass VR-OOS als modulares und verteiltes System implementiert wird. Ein daraus resultierender Vorteil ist es, dass relativ einfach Zusatzfunktionen sowie neue Ansätze für die Funktion bereits existierender Module durch die Implementierung neuer Module umgesetzt werden können.

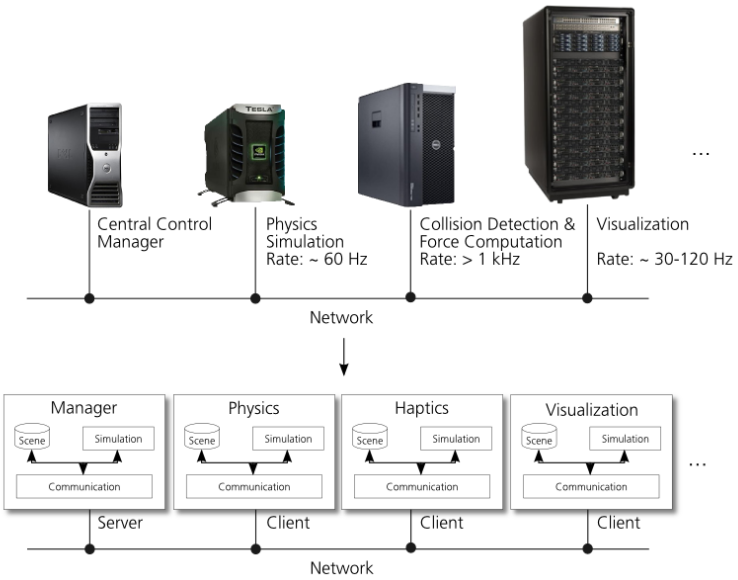

Abbildung 2 – Architektur von VR-OOS[2]

Das Schema in der Abbildung 2 zeigt den logischen Aufbau des Systems. Dabei ist zu sehen, dass außer den bereits angesprochenen Aufgaben für Physik, Visualisierung und Eingabe auch ein sogenanntes Manager-Modul vorhanden ist. Dieser fungiert im System als Server, während alle anderen Module Clients sind. Die Kommunikation wird immer über den Manager abgewickelt. Zum einen brauchen die Module auf diese WEise keine Verbindung zu allen anderen Modulen. Zum anderen kann so einfach eine Filterfunktion eingebaut werden, welche z.B. die Updateraten auf ein

gewünschtes Maß drosselt, oder Nachrichten gegen Randbedingungen prüft und gegebenenfalls eingreift und den Inhalt der Nachricht anpaßt, um die Sicherheit des Menschen und der Geräte zu gewährleisten.
Jedes Modul hat, wie dargestellt, drei Hauptbestandteile: Es muss ein Thread für die Kommunikation nach außen zur Verfügung stehen. Ein anderer Thread muss die Berechnungen ausführen, für die das Modul gedacht ist. Außerdem müssen die Daten der gerade dargestellten Szene vorgehalten werden, um die zugedachte Aufgabe erfüllen zu können.[4]

2.2.2 Stand-alone Raytracing Modul

Neben den oben erwähnten vier Standard-Modulen, gibt es inzwischen einige Erweiterungsmodule, wie z.B zur Unterstüzung von sperzieller Tracking-Hardware, oder zur Sprachsteuerung. In den vorhergehenden Praxisphasen wurde ein alternatives Visualisierungsmodul als Erweiterungdmodul entwickelt, welches das Raytracing-Verfahren zur Berechnung der Bilder nutzt. Dieses Modul nutzt das Framework `OptiX` von Nvidia zum Raytracen. OptiX stellt eine Programmierschnittstelle (engl.: Application Programming Interface (API)) für Grafikkarten (engl.: Graphics Processing Unit (GPU)) bereit. Das Framework kümmert sich um die Parallelisierung auf der Grafikkarte sowie einer verbesserten Kreuzungsabfrage von Strahlen und Objekten in der Szene. Dafür muss vorher ein sogenannter Context aufgebaut werden. In diesem befinden sich alle Objekte der Szene, hierarchisch angeordnet in einem Baum. Die Anordnung wird dabei durch ihre Zuordnung zueinander bestimmt. So kann sich z.B. ein größeres Objekt aus mehreren kleinen zusammensetzen. Der Vorteil daran ist, dass erstmal nur das übergeordnete Objekt auf eine Kreuzung mit dem Lichtstrahl getestet werden muss und nur wenn es getroffen wird, auch die untergeordneten Objekte. Da OptiX in dieser Arbeit nicht stärker benutzt wurde, sei hiermit auf die offizielle Dokumentation sowie die vorrangegange Arbeit des Autors verwiesen.[5][3]
Mit Hilfe dieser Funktionalität konnte ein funktionierender Raytracer entwickelt werden, der sich in das VR-OOS Projekt integriert. Dafür wurden zum einen die

Funktionen der VR-OOS Module implementiert, wie z.B. das Verarbeiten von ankommenden Updates. Zum anderen wurde durch entsprechende Klassen die Übersetzung der Szene von der VR-OOS Datenstruktur zur OptiX Struktur geschaffen. Dies betrifft Geometrien, Lichter und Kameras sowie die ensprechende Hierarchiemöglichkeiten.
Das entstandene Raytracer-Visualisierungsmodul ist in der Lage, die Szenen aus der VR-OOS Datenstruktur anzuzeigen und Updates vom VR-OOS Manager zu empfangen und einzuarbeiten. Aufgrund der Implementierung ist das Modul jedoch nur in der Lage, auf einem einzelnen Computer geschehen. Außerdem erreichte es bei komplexeren Szenen und einer HD-Auflösung nur Bildwiederholraten von fünf Bildern pro Sekunde. Für eine interaktive VR-Anwendung wird jedoch ungefähr das sechsfache benötigt.

2.3 High-Performance Cluster

Der zu entwickelnde Raytracer wird auf einem Computer Cluster laufen. Dies ist ein Hardwaresystem, welches für Hochleistungsberechnungen zusammen gestellt wurde. Bei der Planung eines solchen Systems gibt es zwei grundsätzliche Ansätze: Zum einen kann ein spezieller Supercomputer gebaut werden. Bei diesem wird die Hardware nur für dieses eine System entwickelt und zusammen gestellt. Während ein solches System genau auf die Anforderungen zurecht geschnitten ist und somit die Hochleistung bringt, sind die Kosten in der Regel extrem hoch. Aus diesem Grund hat sich ein zweiter Ansatz etabliert: Aus Massenwaren aus dem Regal werden Zusammenschlüsse von Computersystemen zusammen gebaut. Die Auswahl der Komponenten geschieht wieder anhand der Anforderungen an das System. Dabei kann es vorkommen, dass Abstriche in Kauf genommen werden müssen, z.B. ein nicht optimales Verhältnis von Rechenleistung und Arbeitsspeicher. Ein solches Cluster kann jedoch oft die gleiche oder ähnliche Leistung wie ein spezieller Hochleistungsrechner, kostet jedoch meistens Größenordnungen weniger.[6] Der GPU-Cluster der Abteilung SRV is ein solches System, das aus Standardkomponenten zusammengestellt wurde. Er besteht aus vier einzelnen Computern, welche

jeweils zwei CPUs und drei GPUs beinhalten. Die CPUs sind Intel Xeon Westmere X5670 und die GPUs sind Quadro 6000 von Nvidia. Die Verbindung zwischen den Computern erfolgt über das spezialisierte InfiniBand, welches für die Verbindung von Computern in einem Hochleistungscluster entwickelt wurde. Es erreicht Übertragunsraten von 40GBit/s.[7]
Der zweite Aspekt von Clustern betrifft das Ansprechverhalten nach „außen“. Auch dafür gibt es wieder zwei verschiedene Ansätze. Zum einen ist es möglich, dass es nur eine Ansprechstation im Cluster gibt. Jede Kommunikation läuft über diese Stelle. Dieser koordiniert die Arbeitsverteilung im Cluster. Dies hat zum Vorteil, dass es nach außen wie ein einzelner Computer wirkt. Der Nachteil ist jedoch die besondere Anfälligkeit für einen Totalausfall, wenn die Ansprechstation ausfällt. Demgegenüber sind beim zweiten Ansatz alle Computer im Cluster gleichberechtigt und einzeln von „außen“ ansprechbar. Dadurch kann zum einen jeder auch einzeln genutzt werden und wenn einer ausfällt, geht nur dessen Rechenleistung verloren. Die vier Computer des GPU-Clusters bei SRV sind einzeln über eine 1GBit Ethernet Verbindung ansprechbar. Eine verteilte Anwendung auf dem Computer muss sich jedoch selbst darum kümmern, wenn ein Teil von ihr auf einem Computer läuft, welcher gerade dann ausfällt.

2.4 Message Passing Interface

Ein Cluster ist ein verteiltes System aus mehreren CPUs mit verteiltem Speicher. Eine Anwendung, die auf einem Cluster läuft, muss demzufolge aus mehreren Prozessen bestehen, um auf mehreren CPUs zu laufen und die erhöhte Rechenleistung nutzen zu können. Daraus entsteht aber das Problem der Kommunikation zwischen Prozessen, sodass diese sich synchronisieren und Daten austauschen können. Da der zu entwickelnde Raytracer für die Arbeit auf einem Cluster aufgeteilt werden soll und sich die einzelnen Prozesse für die Berechnung des Gesamtbildes koordinieren müssen, muss eine effektive Lösung für die Kommunikation genutzt werden. Im Bereich des Hochleistungsrechnens hat sich dabei in den letzten Jahren der Message-Passing Interface(MPI) Standard durchgesetzt.[8]

Dieser Standard, welcher zur Zeit in der Version 3.0 vorliegt, definiert dabei Funktionen in den Sprachen C und Fortran sowie deren explizites Verhalten. Diese Funktionen haben alle zum Ziel, Nachrichten zwischen Prozessen auszutauschen. Da das Interface systemunabhängig und Programme somit portabel sein sollen, wird die eigentliche Implementierung Hardware-Händlern, Software-Konzernen und unabhängigen Einrichtungen überlassen. Dadurch wurde der MPI Standard in vielen Bibliotheken implementiert, welche meistens für spezielle Hardware gedacht sind. Durch diesen Ansatz ist gewährleistet, dass der Nachrichtenaustausch durch einfache, standardisierte Aufrufe erfolgen kann. Die dann benutzte Bibliothek nutzt jedoch die Hardwarespezifikationen so gut wie möglich aus, da sie darauf angepasst ist.[9]
Wie im Kapitel 2.3 geschrieben wurde, sind die einzelnen Knoten im GPU-Cluster der Abteilung SRV über InfiniBand angebunden. Speziell für diese Hardware wurde die MPI-Bibliothek MVAPICH von der Ohio State University entwickelt.[10]
Für die Nachrichtenübermittlung im zu entwickelnden parallelen Raytracer werden fünf Grundfunktionen von MPI genutzt, wobei zwei direkt zusammen gehören, sowie die zwei wichtigen Übertragunsmodi. Sie sollen im Folgenden vorgestellt werden.

2.4.1 Send und Receive

Die einfachste Art und Weise Nachrichten auszutauschen besteht durch die Nutzung der MPI-Funktionen `Send` und `Receive`. Sie sorgen dafür, dass eine Nachricht von einem Prozess zu einem anderen Prozess transferiert wird. Beim Funktionsaufruf von `Send` muss dabei der Speicherort der Nachricht, die Länge, der Datentyp, die eindeutige Identifikationsnummer des Zielprozesses sowie eine Nummer als Nachrichtentyp übergeben werden. Ein für den Empfang dieser Nachricht bestimmtes `Receive` muss mit fast den gleichen Parametern aufgerufen werden. Nur muss es die Adresse angeben, wo die Nachricht gespeichert werden soll sowie die Identifikationsnummer des Senders. Als Datentyp sind dabei einfache Typen wie `Integer` oder `Character` möglich aber auch selbst zusammengestellte Strukturen.

2.4.2 Scatter

`Send` und `Receive` sind nur für die Punkt-zu-Punkt Kommunikation gedacht, für die Aufteilung des Bildes müssen jedoch Nachrichten an alle Prozesse verschickt werden. Dabei soll jeder Prozess natürlich nur die für ihn wichtigen Daten erhalten. Für einen solchen Fall stellt MPI die Funktion `Scatter` bereit. Bei dieser Funktion werden Daten von einem Sender an mehrere Empfänger verteilt, wobei jeder Empfänger einen anderen Teil dieser Daten erhält. Die Parameter der Funktion teilen sich in drei Bereiche auf. Der erste Bereich ist für den Sender wichtig: Es muss die Startadresse der Daten, die Anzahl der Elemente für jeden Empfänger sowie der Datentyp der Elemente spezifiziert werden. Der zweite Bereich ist für die Empfänger wichtig: Entsprechend dem Sender müssen sie die Adresse des vorgesehenen Speicherortes für die Daten, die Anzahl der Elemente sowie den Datentyp bereitstellen. Der letzte Bereich ist für Sender und Empfänger wichtig und muss gleich sein: Die Identifikationsnummer des Senders.

2.4.3 Gather

Am Ende des Berechnungsvorganges müssen die Bildteile in einem Prozess gesammelt werden, um sie zusammensetzen zu können. Dafür bietet sich die MPI-Funktion `Gather` an, welche umgekehrt zu `Scatter` funktioniert. Bei dieser werden von allen Prozessen, welche im gleichen MPI Kontext gestartet wurden, Daten in einem Prozess gesammelt. Die Parameter des Befehls setzen sich aus drei Bereichen zusammen. Der erste Teil ist für die sendenden Prozesse wichtig: Sie spezifizieren die Startadresse der Nachricht, die Anzahl der Elemente in der Nachricht sowie den Datentyp der Elemente. Der zweite Bereich ist für den Empfängerprozess wichtig: Die Speicheradresse für die empfangenen Daten muss genannt werden sowie wieder die Anzahl der von einem Sender zu empfangenden Elemente und deren Datenyp. Der dritte Bereich muss bei allen identisch sein: Die Identifikationsnummer des Empfängers muss bereitgestellt werden.

2.4.4 Broadcast

Die letzte Grundfunktion, der `Broadcast`, ist für das Verbreiten einer einzigen Nachricht von einem Prozess an alle anderen Prozesse gedacht. Dies ist z.B. für das Übermitteln von Updates der Szene geeignet. Die Funktion ist ähnlich der `Send` und `Receive` Funktionen. Es muss die Adresse für die Nachricht spezifiziert werden, wobei beim Sender die Nachricht bereits vorliegen muss und sie beim Empfänger an diese Stelle kopiert wird. Außerdem muss, wie bei allen Kommunikationsbefehlen, die Länge des Datensatzes und der Datentyp genannt werden. Als letztes fehlt dann noch die Identifikationsnummer des Senders.

2.4.5 Blockierende und Nicht-Blockierende Kommunikation

Während die bisher vorgestellten Funktionen durch eine verschiedene Anzahl an Sendern und Empfängern gekennzeichnet sind, gibt es auch noch zwei wichtige Modi, wie jede dieser Kommunikation ablaufen kann: Blockierende und nicht-blockierende Funktionsaufrufe. Dabei beschreiben die Namen bereits die Besonderheiten. Bei einem blockenden Funktionsaufruf kehrt die Funktion erst zurück, wenn die Kommunikation von der genutzten MPI-Bibliothek ausgeführt wurde. Dabei reicht es für die meisten Sendefunktionen, wenn die Nachricht in einen Systempuffer geschrieben wurde, welcher von MPI verwaltet wird. Die Empfangsaufrufe werden jedoch erst dann beendet, wenn die Nachricht im bereitgestellten Puffer der Anwendung liegen. Die nicht-blockenden Aufrufe kehren sofort zur Anwendung zurück. Die MPI-Bibliothek erledigt im Hintergrund ihre Arbeit. Auf die angegebenen Speicherbereiche darf dabei von der Anwendung nicht zugegriffen werden, bis die Kommunikation vollendet wurde. Der Stand der Kommunikation kann dabei über eine `Text`-Funktion abgefragt werden. Außerdem gibt es eine `Wait`-Funktion. Diese ist dann wieder eine blockende Funktion und kehrt erst zurück, wenn die Kommunikation beendet ist. Die blockenden Funktionen können deshalb für eine Kommunikation genutzt werden, welche auch gleichzeitig zur Synchronisierung gebraucht wird. Nicht-blockende Funktionen sind hingegen nützlich, wenn auf sich regelmäßig wiederkehrende Nach-

richten getestet wird, wie z.B. für Updates der Szene.

2.5 Streaming

Die berechneten Bilder werden am Ende auf einen Frontend angezeigt werden. Dafür müssen kontinuierlich Daten übertragen werden, wobei gerade für den Empfänger(Frontend) nicht klar ist, wann wieviel Daten ankommen. Einen solchen kontinuierlichen Datenstrom zwischen zwei Endpunkten nennt man Streaming. Für die Übertragung von Daten über Ethernet gibt es zwei weit verbreitete Protokolle, dessen Vor- und Nachteile in Bezug auf Streaming nachfolgend kurz erklärt werden sollen.

2.5.1 User Datagram Protocol

Das User Datagram Protocol(UDP) ist ein paketorientiertes, und damit verbindunsloses, Protokoll der Internetfamilie. Es benötigt einen minimalen Protokollaufwand von acht Bytes. Durch dieses Protokoll wird jedoch keine Ankunft der Daten am Empfänger garantiert und es besteht die Möglichkeit, dass durch ungeschicktes Routing Duplikate beim Empfänger eintreffen.[11]
Beim Streamen müssen viele Pakete geschickt werden, dadurch erscheint der geringe Protokollaufwand als großer Vorteil. Dies kann sich als Fehlannahme herausstellen, da für jedes Paket ein neuer Weg im Netz gefunden werden muss. Auch darf die Eigenschaft von UDP beim Streamen nicht vernachlässigt werden, dass sich der Empfänger um das richtige Sortieren kümmern muss, um Anzeigefehler zu vermeiden. Außerdem ist die Nutzlast von UDP-Paketen auf 65527 Byte beschränkt, da im UDP-Kopf nur 16 Bit für die Längenangabe reserviert sind und 8 Byte für den Kopf selbst benötigt werden. Dies kann also zum Problem werden, wenn ein zu übertragendes Bild größere Datenmengen erfordert.

2.5.2 Transmission Control Protocol

Im Gegensatz zu UDP ist das Transmission Control Protocol(TCP) ein verbindungsorientiertes Protokoll. Es wurde dahingehend konzipiert, eine verlässliche Verbindung zwischen zwei Endpunkten zu gewährleisten, welche durch eigentlich weniger verlässliche physikalische Netze verbunden sind. Daraus resultiert ein größerer Protokollaufwand, um zum einen die Verbindung herzustellen, aber auch die Überbringung einzelner Nachrichten sicher zu gestalten und zu überprüfen.[12]
Die sichere Übertragung ist von Vorteil für das Streamen, da sich so das Frontend nicht um die Sortierung kümmern muss. Der größere Protokollaufwand erscheint für die vielen Pakete als großer Nachteil, da so die Bandbreite für die Nutzdaten sinkt. Jedoch haben bereits erste Tests gezeigt, dass dies eher von Vorteil ist, weil dadurch nicht jedesmal ein neuer Weg durch das Netz gesucht werden muss.

2.5.3 Zero-MQ

Bereits nach kurzer Testzeit hatte sich herausgestellt, dass TCP trotz des Protokollaufwandes für das Streaming zum Frontend geeignet ist. Da bei VR-OOS für die einfache Benutzung von TCP-Verbindungen die Bibliothek Zero-MQ benutzt wird, soll diese vorgestellt werden.
Die selbstständige Implementierung vom TCP-Protokoll ist fehleranfällig, weshalb immer auf bereits bestehene, und somit länger getestete, Implementierungen zurückgegriffen werden sollte. Zero-MQ ist eine solche Bibliothek, die neben TCP auch Multicast beherrscht, was die Verbindung von mehreren Frontends an einen einzelnen Raytracer erleichtert. Die Benutzung der Bibliothek geschieht durch wenige Zeilen Code: Ein Kontext muss erstellt werden, eine Sockel mit einer Verbindungsstrategie, z.B. Publish-Subscribe, geöffnet werden und dann dieser Sockel an eine Adresse gebunden werden. Ab diesen Punkt können Nachrichten versendet werden.

2.6 Video Encoding

Während der generelle Transport durch das Streaming gelöst wird, gibt es noch das Problem der Datenmengen, wie bereits frühere Versuche von interaktiven Raytracern, z.B. in [13] zeigen. Bei einem Bild mit einer Auflösung von 1920x1080, vier Werten pro Pixel (Rot, Grün, Blau und Transparenz) und acht Bit pro Wert entstehen 8294400 Byte pro Bild, welche übertragen werden müssen. Wenn das vorhandene 1Gbit-Netzwerk nur diese Nutzdaten übertragen würde, könnten maximal 15,07 Bilder pro Sekunde gesendet werden. Aus diesem Grund muss die Möglichkeit geschaffen werden, die berechneten Bilder für den Transport zum Frontend zu komprimieren. Dieses komprimieren wird encodieren genannt. Das Wiederherstellen des Ursprungsbildes wird decodieren genannt. Die Entwicklung eines solchen Algortihmus würde jedoch diese Arbeit sprengen. Aus diesem Grund wird auf existierende Codecs (`Co`dieren, `dec`odieren) zurückgegriffen. Zur Zeit ist ein führender Codec der von der International Telecommunication Union entwickelte Standard H.264. Für die Funktionsweise des Standards sei hierbei auf [14] verwiesen. Der Standard wurde in verschiedenen, offenen und proprietären, Bibliotheken implementiert. Am weitesten verbreitet ist dabei die x264-Bibliothek[15], welche Open Source ist. Ein encodiertes Bild in der Auflösung von 1920x1080 ist dabei nur noch ungefähr 20000 Byte groß.

3 Systemdesign

Das Ziel dieser Arbeit ist ein verteilter, paralleler Raytracer. Daraus ergibt sich, dass das Gesamtsystem aus mehreren Komponenten besteht, welche jedoch aufeinander abgestimmt sein müssen. Die folgenden Kapitel beschäftigen sich zuerst mit dem Design des Gesamtsystems, um darauf aufbauend das interne Design der einzelnen Komponenten zu erläutern.

3.1 Design des Gesamtsystems

Das Gesamtsystem hat einige Rahmenbedingungen aufgrund der Einbettung in das VR-OOS zu beachten. Wie im Kapitel 2.2.1 gesagt, muss mindestens eine Komponente der Visualisierung als VR-OOS Modul fungieren. Dessen Aufgabe ist es dann, Änderungen in der Szene, z.B. von Geometrien, Materialien oder Transformationen, mit dem restlichen System zu synchronisieren. Dafür stellt das VR-OOS Framework bereits Methoden zur Verfügung. Außerdem muss die in Kapitel 2.1 angesprochene Aufteilung des Bildes an die einzelnen Grafikkarten sowie die Zusammensetzung des Bildes erfolgen, bevor das berechnete Bild an den anzeigenden Computer geschickt wird. Die grundlegende Verteilung muss in eine Aufteilung in Front- und Backend bestehen. Das Frontend läuft auf dem Computer, auf dem das berechnete Bild dargestellt werden soll. Das Backend hingegen läuft auf der spezialisierten Hardware, welche geeignet für die Grafikberechnung ist. Die Frage ist jedoch, wie der Datenfluss und die Anbindung an VR-OOS geschehen soll.
Die Einbindung des VR-OOS Moduls kann durch zwei Ansätze geschehen. Zum einen könnte das Frontend diese Funktion beinhalten. Es müsste sich danach also um die Weiterleitung der Daten an das Backend kümmern. Dies würde eine Abkopplung des Backends von VR-OOS bedeuten. Beim zweiten Ansatz hingegen wird das VR-OOS Modul in das Backend integriert. Die VR-OOS Module sowie das Visualisierungs-Backend werden immer im gleichen LAN liegen, sodass die Nachrichten vom VR-OOS Manager auch immer beim Backend zur Verfügung

stehen. Somit wäre es die doppelte Nachrichtenmenge, wenn die Updates erst zum Frontend und von da aus zum Backend geschickt werden. Außerdem kann das Frontend mit diesem Ansatz sehr schlank gestaltet werden. Aus diesem Grund wurde sich für dafür entschieden, das VR-OOS Modul in das Backend zu integrieren. Somit hat das Frontend nur eine Verbindung zum Backend und nicht zum VR-OOS Manager.
Wie bereits gesagt, muss die Aufteilung des Bildes geschehen. Die einzig sinnvolle Möglichkeit dafür ist eine Aufteilung des Backends in Master und Slaves. Der Master teilt das Bild auf, schickt diese Aufteilung an die Slaves, diese berechnen dann die ihnen zugeteilten Pixel und schicken die Ergebnisse an den Master. Dieser kann dann das Bild zusammen setzen und an das Frontend schicken.

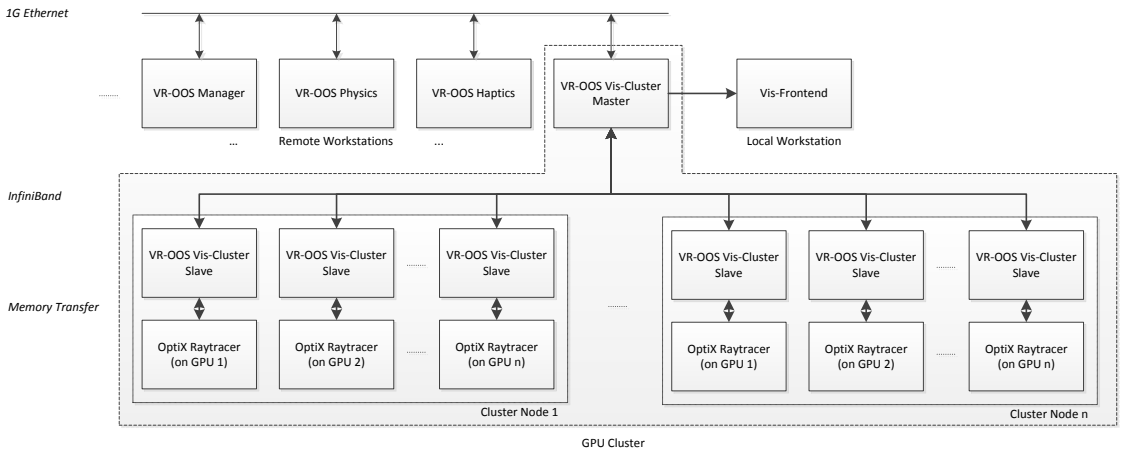

Abbildung 3 – Architektur des Gesamtsystems

Als letztes ist für das Gesamtsystem zu bedenken, dass eine Grafikkarte am besten nur von einem Visualisierungsprozess genutzt werden sollte, um ständige Prozess und somit auch Speicherwechsel auf der Karte zu vermeiden. Pro Grafikkarte in der ausführenden Hardware sollte also ein Slave-Prozess gestartet sein.
Diese Überlegungen führen zu einer Architektur des Gesamtsystems, wie es in Abbildung 3 zu sehen ist. Der Master- und die Slaveprozesse werden auf der spezialisierten Hardware, z.B. einem GPU Cluster, ausgeführt. Die Kommunikation

zwischen diesen Prozessen verläuft über die schnellst mögliche Verbindung, also dem InfiniBand durch MPI. Der Master kommuniziert über Ethernet-LAN mit den anderen VR-OOS Modulen und das Visualisierungsfrontend, welches z.B. auf einem lokalen Arbeitsrechner läuft, ist nur mit dem Master verbunden.

3.2 Teilprozesse des Backends

Die Aufgabe vom Master und jedem Slave lassen sich in einzelne Teilaspekte runterbrechen, welche für die Berechnung jedes neuen Bildes wiederholt werden müssen. Diese werden in Abbildung 4 gezeigt. Der Nachrichtenaustausch zwischen Master und Slave erfolgt dabei, wie schon in Kapitel 3.1 gesagt, durch MPI-Funktionen.

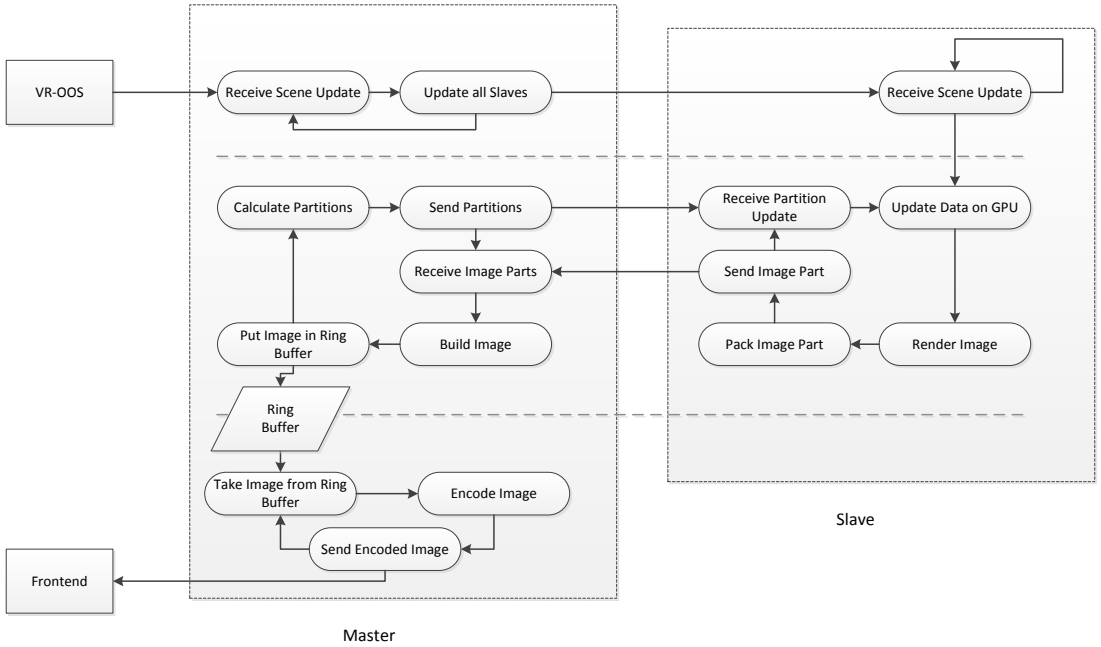

Abbildung 4 – Teilprozesse des Backends

Der Master hat drei Aufgaben: Er soll Updates der Szene vom VR-OOS Manager entgegennehmen und diese an alle Slaves weiterleiten. Daneben berechnet er die

Aufteilung des Bildes, teilt jedem Slave dessen zu berechnende Pixel zu und sammelt die von jedem Slave berechneten Bildteile wieder ein. Abschließend baut er das Gesamtbild zusammen und stellt es zum Encodieren und Verschicken zur Verfügung. Anschließend muss die Bildaufteilung wieder dahingehend überprüft werden, ob sie noch optimal in Hinblick auf die Gesamtperformance. Die dritte Aufgabe besteht demzufolge im Encodieren der Bilder und dem Verschicken an das Frontend.
Die Hauptaufgabe der Slaves besteht darin, endlos wiederholend das gerade aktuelle Bild zu berechnen. Dazu muss jeder zuerst die Aufteilung empfangen. Anschließend werden die Daten auf der GPU über OptiX auf den aktuellen Stand gebracht um anschließend das Bild zu berechnen. Da jedoch aufgrund der Beschaffenheit von OptiX das gesamte Bild in den Speicher geschoben wird, müssen die von diesem Slave berechneten Pixel extrahiert werden, um diese dann an den Master zu senden. Nebenbei muss der Slave die Szenenupdates empfangen, welche jederzeit gesendet werden können. Diese müssen dann bereit gehalten werden, um während des Updates der GPU mit kopiert zu werden.

3.3 Internes Design des Masters

Die interne Struktur des Master sowie die Abhängigkeiten an die anderen Komponenten sind in Abbildung 5 dargestellt. Wie bereits in Kapitel 3.2 deutlich wurde, hat der Master drei mehr oder weniger unabhängige Aufgaben. Dies wäre zum einen die Verbindung zum Manager, um VR-OOS Updates zu empfangen, welche dann an die Slaves weitergeleitet werden müssen. Das Grundgerüst für diese Aufgabe ist durch das VR-OOS Modul vorgegeben, wodurch die VR-OOS Funktionen automatisch in einen eigenen Thread gekapselt ist.
Vollkommen unabhänig davon ist die Aufteilung des Bildes, welche an die Slaves kommuniziert werden muss sowie das Zusammensetzen der einzelnen Bildteile. Diese Aufgaben können auch in einen eigenen Thread gekapselt werden, da sie die grundsätzlich dauerhaft wiederholte Kommunikation mit dem Slave beeinhaltet.
Die letzte Aufgabe ist die Vermittlung des fertigen Bildes an das Frontend. Dafür muss ein fertig berechnetes Bild erst encodiert und dann verschickt werden. Beson-

ders das Encodieren braucht vorraussichtlich einen größeren Zeitanteil, weswegen wieder ein eigener Thread von Vorteil für diese Aufgabe ist, um moderne Multi-Threaded-CPUs auszunutzen und die anderen Kommunikationen und Berechnungen nicht unnötig aufzuschieben.

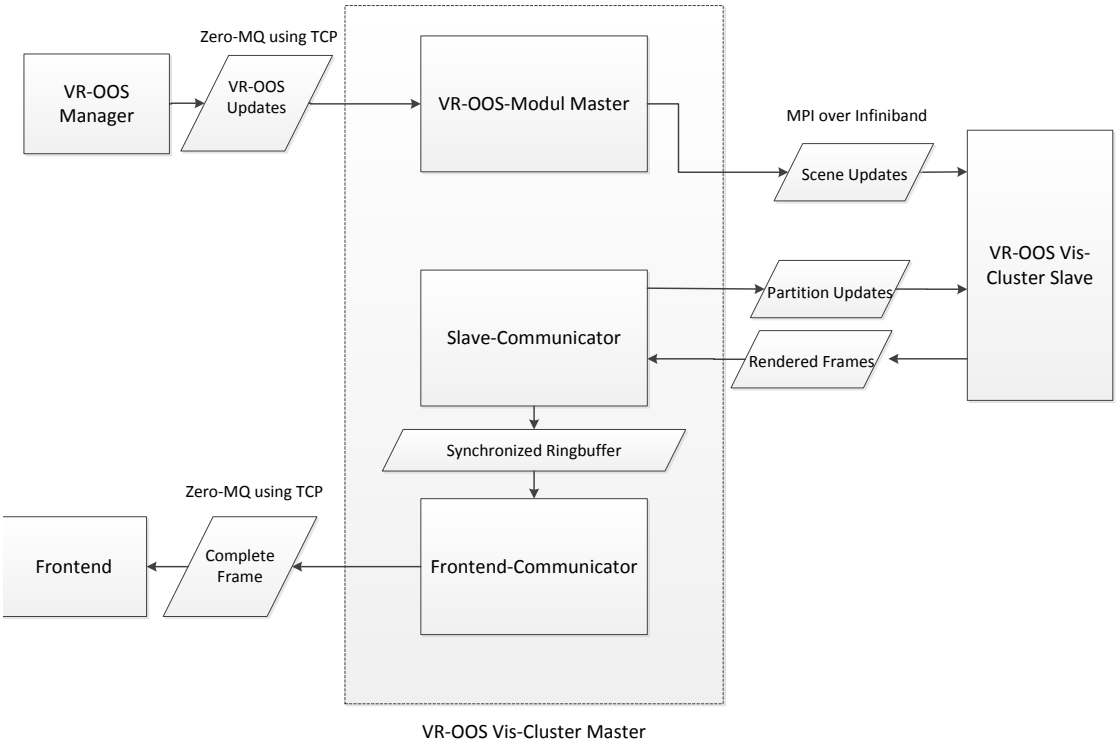

Abbildung 5 – Aufgabenaufteilung im Master und Kommunikation mit den anderen Komponenten

3.4 Internes Design der Slaves

Die Slaves haben, ähnlich dem Master, drei mehr oder weniger unabhängige Aufgaben, welche in Abbildung 6 dargestellt sind.
Zum einen muss er dauerhaft bereit sein, Updates der Szene zu empfangen. Diese werden dann erstmal in der Datenstruktur auf der CPU gespeichert, und erst vor

dem Beginn der nächsten Bildberechnung auf die GPU kopiert. Daraus folgt, dass die Kommunikation für Szenenupdates unabhängig von der eigentlichen Bildberechnung ist und demzufolge in eigenen Thread ausgelagert werden kann. Dies ist besonders wichtig, da in der Zeit in der ein Bild berechnet wird, mehrere Objekte in der Szene sich ändern können. Würde die Abfrage, ob eine MPI-Nachricht empfangen wurde, jedoch innerhalb des Berechnungszyklus geschehen, könnte zwischen jedem Bild nur ein Objekt ein Update bekommen.

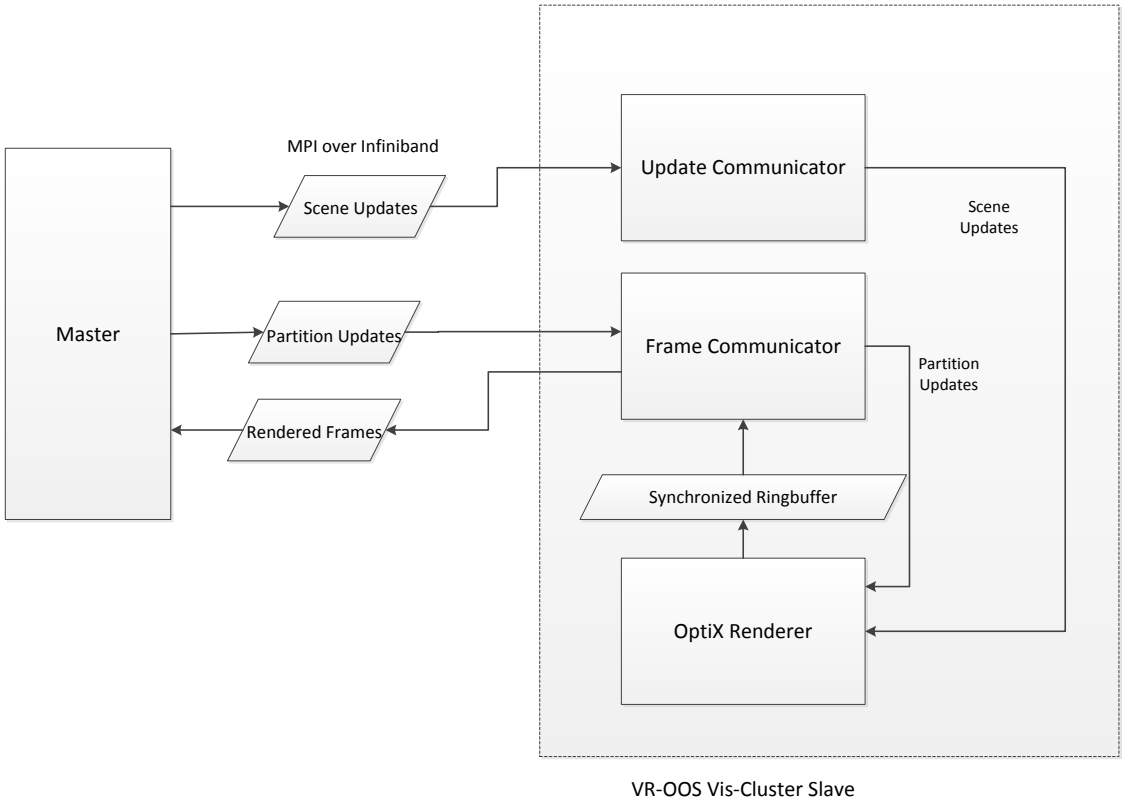

Abbildung 6 – Aufgabenaufteilung im Slave und Kommunikation mit den Master

Die anderen beiden Aufgaben sind etwas enger aneinander gebunden. Einerseits muss die Regelmäßige Kommunikation mit dem Master bewältigt werden, also das Empfangen des zugeteilten Bildausschnittes und das Extrahieren und Senden der berechneten Pixel. Andererseits muss die die Kommunikation mit der Grafik-

karte erfolgen, wozu das Übertragen der Daten sowie die Berechnung des Bildes gehört. Da eine Bildberechnung nur Sinn macht, wenn Master und Slave wissen, wie das jeweils betrachete Bild aufgeteilt wurde, erscheint es sinnvoll, diese inhaltlich getrennten Aufgaben zusammen in einem Thread zu bearbeiten. Dann wartet dieser Thread jedoch darauf, dass eine neue Aufteilung empfangen wird, bevor mit der Berechnung des nächsten Bildes begonnen wird. Es ist jedoch möglich, dass zukünftige Verbesserungen dazu führen, dass vorsorglich bereits das nächste Bild berechnet werden kann, während noch gar keine Aufteilung dafür da ist. Aus diesem Grund wird die Bildkommunikation mit dem Master sowie die Berechnung des Bildes in getrennten Threads ausgeführt und die Übertragung der Bilddaten zwischen diesen Threads geschieht über einen synchronisierten Ringpuffer.

3.5 Internes Design des Frontends

Im Gegensatz zu Master und Slaves des Backends hat das Frontend eine einfache Struktur. Es muss Bilder empfangen, diese decodieren und dann anzeigen. Die Implementierung von Zero-MQ sorgt dafür, dass das Empfangen von Nachrichten ohne Blockierung des nutzenden Threads geschehen kann. Außerdem macht das Anzeigen des nächsten Bildes nur Sinn, wenn ein neues empfangen und decodiert wurde. Im Gegensatz zum Master und Slave gibt es im Frontend also Prozesse die nebenläufig programmiert werden müssen.

3.6 Interface der Bildaufteiler

Die Parallelisierung des Raytracens beruht darauf, dass das Bild in mehrere Teile aufgeteilt werden kann. Um verschiedene Strategien ausprobieren zu können, ohne viel im Quellcode ändern zu müssen, aber auch um später neue Ansätze problemlos integrieren zu können, ist es sinnvoll, ein einheitliches Interface für alle Bildaufteiler zu schaffen. Da die Arbeit in C++ geschrieben wurde, geschieht dies durch eine abstrakte Klasse. Alle Klassen, welche einen Bildaufteiler implementieren, müssen dann von dieser abstrakten Klasse erben. Die zu implementierenden Funktionen

sind in den Quelltexten 1 und 2 zu sehen. Die Funktionen im ersten Quelltext werden vom Mster benötigt. Die Slaves benötigen die Funktionen, welche im zweiten Quelltext gezeigt werden.

```
1  virtual std::vector<std::vector<int> > partitionNewFrameparts(
       int slaveCount, std::vector<int> lastTimes, int width, int
       height) = 0;
2  virtual int* buildFrameReceiverLengthsBuffer() = 0;
3  virtual int* buildFrameReceiverStartsBuffer() = 0;
4  virtual void assembleFrame(std::vector<unsigned char> *
       outputBuffer, unsigned char *inputBuffer) = 0;
```

Quellcode 1 – Interface der Bildaufteiler des Masters

Die Funktion `partitionNewFrameparts()` soll vor jedem zu berechnenden Bild aufgerufen werden. Es berechnet aus der Anzahl der Slaves sowie der aktuellen Bildauflösung die neue Bildaufteilung, welche dann abhängig von der jeweiligen Implementierung ist. Um eine dynamische Auftielung zu ermöglichen und die Berechnungszeiten der Slaves anzugleichen, wird außerdem ein Vektor übergeben, der für jeden Slave die benötigte Zeit des letzten Bildes beinhaltet. Wie dies genutzt wird, liegt jedoch an der einzelnen Strategie. Die Funktion liefert als Rückgabe einen Vektor von Integer-Werten für jeden Slave. Um die ohne große Umstände per MPI zu übertragen, ist es von Vorteil, eine Größe für diesen Vektor vorzugeben. Diese wird an einer Stelle in der abstrakten Klasse festgelegt und nennt sich `FRAMEPARTSIZE`. Dafür sollte die maximal benötigte Größe der implementierten Aufteiler herangezogen werden, welche in Kapitel 4.2 beschrieben werden.
Um die MPI-Methoden relativ einfach zu halten, müssen bei deren Aufrufen die Größe der zu empfangenden Daten bekannt sein, sowie die Stelle in einem Puffer, in dem die Daten gespeichert werden sollen. Da jedoch aufgrund der unterschiedlichen Aufteilungen jeder Slave eine andere Anzahl an Pixel berechnen könnte, muss der Bildaufteiler die einzelnen Größen zu Verfügung stellen. Die Methode `buildFrameReceiverLengthsBuffer()` gibt diese Information zurück. Die Methode `buildFrameReceiverStartsBuffer()` hingegen berechnet die Startposition im

Puffer für die Daten der einzelnen Slaves und gibt sie zurück.
Die letzte Methode für den Master ist `assembleFrame()`. Diese werden Pointer auf zwei Puffer übergeben: Einen, in dem die empfangenen Bildteile gespeichert sind und einem, in dem das richtig zusammengesetzte Bild von dieser Funktion gespeichert werden soll.

```
1  virtual void buildSendFrame(float *inputBuffer, unsigned char *
       outputBuffer) = 0;
2  virtual int getSendFrameLength() = 0;
3  virtual bool isCorrectPartitioner(int in_frameParts[
       FRAMEPARTSIZE]) = 0;
```

Quellcode 2 – Interface der Bildaufteiler der Slaves

Die Slaves müssen auch die einzelnen Implementierungen von Bildaufteilern verwenden können. Dazu dienen, wie bereits gesagt, die drei Methoden in Quelltext 2. Da nur die berechneten Pixel übertragen werden sollen, aber OptiX immer einen Puffer für das gesamte Bild beschreibt, müssen die ensprechenden Werte extrahiert werden und in einen zusammenhängenden Puffer gespeichert werden. Diese Aufgabe übernimmt die Funktion `buildSendFrame`. Sie hat außerdem noch eine zweite Aufgabe: Die Farbwerte in VR-OOS sind in `Float` angegeben. Im Raytracer werden deshalb die Farbwerte der Pixel auch in `Float` berechnet. Um die zu übertragenden Daten jedoch so gering wie möglich zu halten, müssen sie auf `char` verringert werden. Da jeder Pixel aus vier `Float` für rot, grün, blau und den Alphawert besteht und jeder `Float`, zumindest auf den benutzten Systemen, vier Byte benötigt, werden so pro Pixel 12 Byte eingespart. Da diese Skalierung nur für die benötigten Pixel durchgeführt werden sollte um Rechenzeit zu sparen, muss jeder Bildaufteiler diese Aufgabe übernehmen.
Da für eine Übertragung durch MPI wieder die Anzahl der berechneten Werte bekannt sein muss, gibt es eine Funktion, die genau diese Information liefert: `getSendFrameLength`.

3.7 Interface der Bildkomprimierer

Die Algorithmen zum Codieren von Bildern zum Zweck der Bandbreitenverringerung sind selbst so komplex, dass sie weit über den Umfang dieser Arbeit hinaus gehen. Es gibt jedoch Implementierungen, welche durch Softwarebibliotheken genutzt werden können. Da es möglich sein soll, später verschiedene Codierbibliotheken zu nutzen und auf ihre Tauglichkeit für das Echtzeitraytracen zu testen, soll, wie bereits bei den Bildaufteilern, eine abstrakte Klasse ein Interface zur Verfügung stellen. Für jede zu nutzende Bibliothek soll dann eine Klasse erzeugt werden, welche die Benutzung abstrahiert.
Im Quelltext 3 sind die Funktionen zu sehen, welche im Backend benutzt werden sollen. Jeder Encoder muss mit Werten initialisiert werden, die für das Encodieren einzelner Bilder mit so wenig wie möglich Latenz am besten sind. Diese Parameter werden vorraussichtlich durch Ausprobieren bzw. die Dokumentation der jeweiligen Bibliotheken angepasst werden und brauchen dann nicht während der Laufzeit dynamisch verbessert werden. Die Funktion `init()` nimmt deshalb nur die Bildschirmauflösung als Parameter an.
Die zweite Funktion ist `encode()`. Diese nimmt zwei Pointer entgegen, eine auf einen Puffer mit den zu codierenden Bild und einen auf den Platz, an den das codierte Bild gespeichert werden soll.

```
1  virtual bool init(int windowWidth, int windowHeight) = 0;
2  virtual bool encode(std::vector<unsigned char> *inFrame, std::
       vector<unsigned char> *outFrame) = 0;
```

Quellcode 3 – Interface der Encoder des Masters

Im Frontend müssen die empfangenen, codierten Daten wieder decodiert werden, um sie anzeigen zu können. Dafür muss für jeden Encodierer des Masters ein passender Decodierer implementiert werden. Diese müssen die Funktionen, welche in Quelltext 4 gezeigt werden, implementieren. Wie die Encodierer müssen auch die Decodierer mit passenden Parametern initialisiert werden. Dies geschieht durch

das Aufrufen der Funktion `init()`.
Die eigentliche Aufgabe übernimmt aber die Funktion `decode()`. Diese nimmt zum einen ein Pointer auf den encodierten Puffer entgegen. Außerdem noch einen Pointer auf einen Puffer vom Typ `GLubyte`. Da das Anzeigefenster mit OpenGL gezeichnet wird, welches genau so einen Typ verlangt, ist es am einfachsten, wenn der Decoder direkt in den richtigen Puffer schreibt.

```
1  virtual bool init(int windowWidth, int windowHeight) = 0;
2  virtual bool decode(std::vector<unsigned char> *inFrame, GLubyte
       *outFrame) = 0;
```

Quellcode 4 – Interface der Decoder der Slaves

4 Aspekte der Implementierung

Im Kapitel 3 wurden die Überlegungen zum Systemdesign und die daraus resultierenden Schlüsse vorgestellt. Diese wurden bei der Implementierung auch bis auf eine Ausnahme umgesetzt. Deswegen wird in Kapitel 4.1 nur auf den Aufbau des Gesamtsystems eingegangen sowie die Ausnahme erläutert. Im Kapitel 4.2 werden dann die implementierten Bildaufteiler näher erläutert. Abschließend werden im Kapitel 4.3 alle Nachrichten, welche im verteilten System ausgetauscht werden, vorgestellt.

4.1 Übersicht des implementierten Systems

Der verteilte, parallele Renderer teilt sich in zwei eigenständige Programme auf. Das Klassendiagramm des Backends, welches auf dem Cluster läuft, ist in Abbildung 7 zu sehen. Alle Prozesse, egal ob Master oder Slave, werden durch die Klasse `VisualizationBackend` gestartet. In dieser entscheidet sich dann aufgrund der durch MPI vergebenen Identifikationsnummer, ob der jeweilge Prozess ein Master ist oder ein Slave.
Der Master ist von der VR-OOS Klasse `Modul` abgeleitet. In der Initialisierung erstellt er außerdem einen `SlaveCommunicator` und einen `FrontCommunicator`. Diese drei Objekte starten dann jeweils einen Thread, um die in Kapitel 3.3 erläuterten Aufgaben zu erfüllen: Die Funktionalität vom VR-OOS Modul ist für die Kommunikation mit dem VR-OOS Manager verantwortlich und verarbeitet empfangene Updates der Szene. Der SlaveCommunicator erstellt je nach Startparametern einen der implementierten Bildaufteiler und übernimmt die Kommunikation mit den Slaves, welche für die Berechnung jedes einzelnen Bildes benötigt wird. Dazu gehört die Aufteilung des Bildes, das Verschicken der Aufteilung, das Einsammeln der Teilbilder und das Zusammensetzen dieser zu einem Gesamtbild. Die Übergabe von Werten zu den Bildaufteilern geschieht jeweils über Pointer. Der FrontCommunicator ist für das Versenden der berechneten Bilder an das Frontend verantwortlich. Dafür benötigt er einen der implementierten Encoder. Dieser kommprimiert das

per Pointer übegebene Bild, welches dann über das Netzwerk an das Frontend geschickt werden kann.
Wenn der Prozess durch die MPI-Identifikationsnummer ein Slave ist, wird ein Objekt der Klasse `VisualizationSlave` erstellt. Dieser benötigt einen `Renderer`, in dem die Logik für den Raytracer steckt, einen Bildaufteiler sowie einen `MasterUpdateCommunicator`, der für das Empfangen und Einpflegen der Szenenupdates verantworlich ist. An dieser Stelle wird deutlich, dass vom Design aus Kapitel 3.4 leicht abgewichen wurde. Die Berechnung des Bildes und das Verschicken des Teilbildes wurden nicht getrennt sondern in einem Thread sequenziell bearbeitet. Dies wurde bisher so implementiert, um Abhängigkeiten im Programm zu vereinfachen. Eine Veränderung hin zum entwickelten Design ist jedoch geplant, um bessere Leistungen zu erzielen.

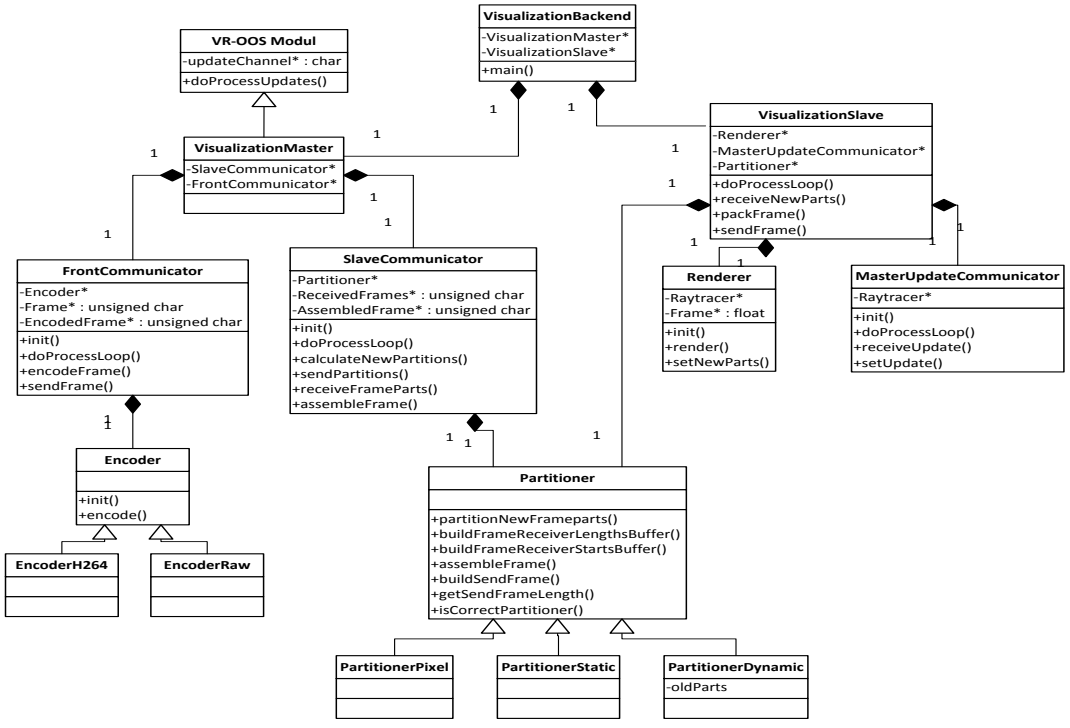

Abbildung 7 – Klassendiagramm des Backends

Das Frontend hat eine vergleichsweise einfache Klassenstruktur, die in Abbildung 8 dargestellt ist. Das Frontend muss, wie im Kapitel 3.5 beschrieben, eine Verbindung zum Backend aufbauen, die einzelnen encodierten Bilder empfangen und diese dann anzeigen. Dafür benötigt es nur einen Decoder. Dieser hat eine `init`-Methode, welchen den benutzten Decoder initialisiert sowie eine `decode`-Methode, die für jedes Bild aufgerufen werden muss und das originale Bild wiederherstellt. Es stehen hierbei zwei Decoder zur Verfügung: Ein Decoder, der zu Testzwecken unkomprimitierte Bilder unangetastet lässt sowie ein H.264-Decoder.

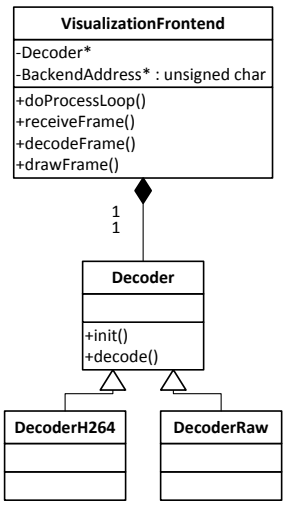

Abbildung 8 – Klassendiagramm des Frontends

4.2 Implementierte Bildaufteiler

Die erreichten Bildraten eines verteilten, parallelen Raytracers hängen maßgeblich von der Leistungsfähigkeit der eingesetzten Bildaufteiler ab. Mehrere wissenschaftliche Arbeiten beschäftigen sich mit der Verteilung der Berechnungsarbeit mit dem Ziel, möglichst ausgeglichene Zeiten zu haben. Die üblichen Verfahren werden bei Mantler et al.[16] sowie Plachetka[17] vorgestellt. Die Verfahren unterscheiden sich dabei grundsätzlich in statische und dynamische Vorgehen. Bei statischen

Verfahren wird das gesamte Bild auf einmal aufgeteilt, was durch die fehlende Anpassungsfähigkeit ein schlechteres Ergebnis bei sich verändernden Szenen verspricht. Die dynamischen Verfahren werden oft Process Farming genannt. Dabei wird erstmal nur ein Teil des Bildes an die Raytracer-Prozesse verteilt. Wenn ein Prozess sein Bildteil berechnet hat, fragt er nach einen neuen Abschnitt. Dies geschieht dann so lange, bis das gesamte Bild berechnet wurde.
Bei diesen Verfahren wird nicht von einem Echtzeit-Raytracer ausgegangen, sodass mehr Zeit für das Anfragen von neuen Bildausschnitten verfügbar ist. Desweiteren startet OptiX immer die Berechnungen für alle Pixel des gesamten Bildes, wodurch es Sinn macht, in einem Durchlauf alle Pixel zu verteilen. Daraus folgt, dass Ideen der bisher verbreiteten Verfahren aufgegriffen werden können, diese jedoch für den Echtzeit-Raytracer mit OptiX angepasst werden müssen. Es wurden zwei statische Verfahren implementiert, welche in den Kapitel 4.2.1 und 4.2.2 vorgestellt werden. De entwickelte dynamische Algorithmus wird danach in Kapitel 4.2.3 erläutert.

4.2.1 Aufteilung in einzelne Pixelspalten

Die Arbeit von Plachetka[17] hat gezeigt, dass eine Verteilung aneinander liegender Pixel auf unterschiedliche Berechnungsprozesse bei Raytracern von Vorteil ist. Dies liegt daran, dass jeder Pixel unabhängig von allen anderen Pixeln berechnet werden kann und Geometrien in der Regel durch mehre Pixel angezeigt werden. Durch die gleichmäßige Verteilung dieser Pixel wird auch automatisch die Aufgabenverteilung ausgeglichen. Dieser Ansatz wird im Bildaufteiler für Pixelspalten genutzt. Dieser teilt jede einzelne Pixelspalte einem Slave zu. Die Spalten werden von links anfangend nacheinander den Slaves zugeteilt. Wenn jedem Slave eine Spalte zugewiesen wurde, geht die nächste Spalte wieder an den ersten Slave usw. Diese Aufteilung ist für vier Slaves in Abbildung 9 zu sehen. In diesem ist jedes rot umrandete Kästchen ein einzelner Pixel. Die vier Objekte in dem 16x9 Bild verteilen sich durch die Aufteilung automatisch auf die einzelnen Slaves.
Für die Aufteilung müssen für jeden Slave sieben Integer-Werte beschrieben werden. Für jeden implementierten Bildaufteiler wird der gleiche Funktionsaufruf von MPI

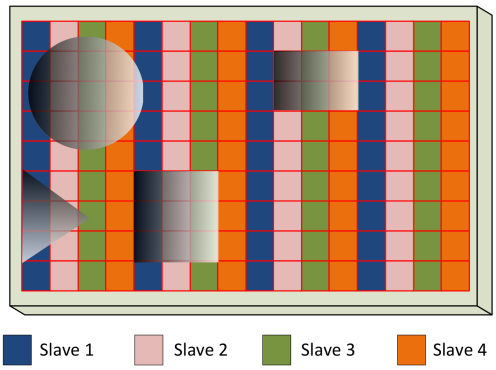

Abbildung 9 – Aufteilen eines 16x9 Bildes in Pixelstreifen für 4 Slaves

genutzt. In dieser muss die Länge der Nachricht festgelegt sein. Deshalb muss die maximale Anzahl der Werte von allen implementierten Bildaufteilern genutzt werden.
Für den Aufteiler in einzelne Pixelspalten steht der erste Wert für die Slavenummer und die zweite für die Anzahl an verwendeten Slaves. Dadurch weiß der Slave, welche Spalten er berechnen muss. Im 5. Wert steht die Höhe der Bildauflösung und im 6. die Weite. Der 7. Wert ist die ID des Aufteilers, in diesem Fall 1. Da nicht mehr Werte für diesen Aufteiler benötigt werden, sind der 3. und 4. Wert ungenutzt.

4.2.2 Aufteilung in statische Rechtecke

Obwohl die oben beschriebene Aufteilung eine gleichmäßige Verteilung ermöglicht, wird kein sehr gutes Gesamtergebnis erwartet. Dies liegt an der internen Optimierung von OptiX, welche für eine bessere Parallelisierung Strahlen, die fast identisch sind, bündelt. Dies betrifft also aneinander liegende Pixel. Aus diesem Grund ist es sinnvoll, im Gegensatz zu den Aufteiler in Pixelspalten, mehr zusammenhängende Pixel einem Slave zuzuteilen. Dieses Verhalten wurde im Aufteiler mit statischen Rechtecken implementiert.
Dieser Aufteiler teilt die Breite des Bildes in genau so viele Abschnitte ein, wie Slaves vorhanden sind. Jedem Slave wird dann ein Rechteck mit der Breite dieser

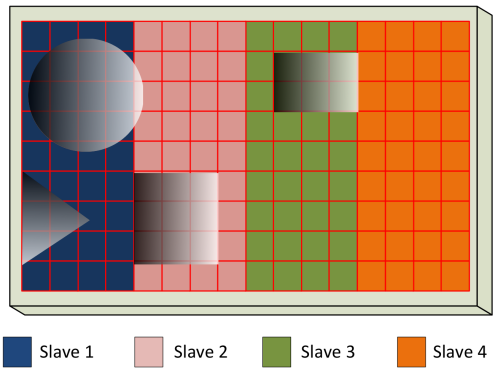

Abbildung 10 – Aufteilen eines 16x9 Bildes in statische Rechtecke für 4 Slaves

Abschnitte und der Höhe des Gesamten Bildes zugeteilt. Eine solche Aufteilung bei vier Slaves und einem 16x9 Bild ist in Abbildung 10 zu sehen. Die vier Geometrien des Bildes liegen identisch zu denen in Abbildung 10. Sie verteilen sich diesmal jedoch sehr ungleichmäßig auf die einzelnen Slaves: Slave 1 hat einen großen Anteil an den Geometrien während Slave 4 gar keine Geometrien in seinm Ausschnitt hat. Wie im vorherigen Kapitel gesagt, müssen für die Aufteilung sieben Werte pro Slave beschrieben werden. Die ersten zwei Werte stehen für die obere linke Ecke des zu berechnenden Rechtecks, wobei der 1. Wert die Breite und der 2. Wert die Höhe angibt. Die nächsten beiden Werte stehen für die untere rechte Ecke des Rechtecks, geordnet in gleicher Weise wie die erste Ecke. Der 5. Wert bestimmt die Höhe des gesamten Bildes und der 6. die Breite. Der 7. Wert ist die ID des Aufteiler, in diesem Fall 2.

4.2.3 Aufteilung in dynamische Rechtecke

Die Aufteilung in Abbildung 10 zeigt deutlich, dass eine statische Aufteilung in ungünstigen Fällen für nicht optimale Ergebnisse sorgen wird. Aus diesem Grund wurde ein dynamischer Bildaufteiler implementiert. Im Gegensatz zu den Algorithmen, die in [16] vorgestellt werden, verteilt dieser Bildaufteiler jeweils das gesamte Bild. Da jedoch die Aufteilung der Geometrien immer erst nach der Berechnung des aktuellen Bildes zu Verfügung steht, kann nur reaktiv die Aufteilung

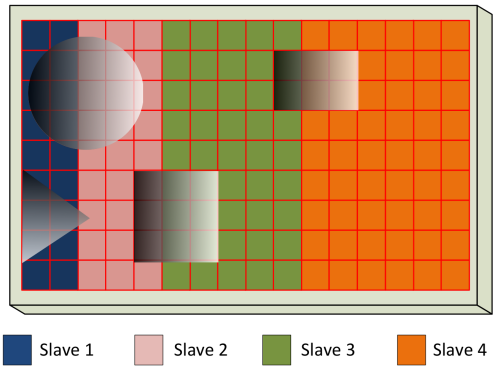

Abbildung 11 – Aufteilen eines 16x9 Bildes in dynamische Rechtecke für 4 Slaves

angepasst werden. Da sich dann jedoch schon die Verteilung der Geometrien geändert haben kann, muss dieser Prozess nach jedem Bild wiederholt werden.
Die Verteilung erfolgt in Rechtecken, die jedoch unterschiedlich breit sein können. Die Slaves senden dem Master die Zeit, die sie für die Berechnung des ihnen zugewiesenen letzten Teilbildes benötigt haben. Darauf aufbauend kann der Aufteiler entscheiden, ob und wie das Bild neu aufgeteilt werden muss: Dafür berechnet er zuerst den Durchschnitt aller Zeiten. Davon ausgehend findet er den Slave, der die längste Berechnungszeit benötigte und wie viel länger er als der Durchschnitt gerechnet hat. Wenn er eine bestimmte Prozentzahl länger gerechnet hat als der Durchschnitt, wird neu aufgeteilt. Dieser Parameter kann später nachjustiert werden, um die Rate der Neuverteilung anzupassen.
Wenn das Bild neu aufgeteilt werden soll, wird die Breite des Rechtecks des langsamsten Slaves durch die von ihm benötigte Zeit geteilt. Durch Multiplizieren mit der Differenz aus benötiger Zeit und Durchschnittszeit aller Slaves wird die Anzahl an Pixelspalten berechnet, die diesem Slave abgezogen werden müssen, sodass er die Durchschnittszeit erreicht. Diese Spalten werden dann gleichmäßig auf alle Slaves verteilt, die schneller waren als die Durchschnittszeit. In Abbildung 11 ist die gleiche Szene wie in den Abbildungen für 9 und 10 zu sehen. Da die Geometrien sich an den linken Rand drängen, wird Slave 1 am längsten brauchen, wodurch er nach einer Neuaufteilung ein kleineres Rechteck berechnen muss. Nach

mehreren Durchläufen werden die Zeiten in einer statischen Szene so ausgeglichen sein, dass keine Neuverteilung mehr stattfinden muss.
Der Algorithmus verteilt die Pixel gleichmäßig auf die entsprechenden Slaves, anstatt dem schnellsten Slave die meisten Pixel zu geben, um den Algorithmus nicht zu komplex werden zu lassen. In einer VR-Umgebung wird selten eine statische Szene angezeigt, sondern die Verteilung der Geometrien in der Szene verändert sich dauerhaft. Die Aufteilung beruht jedoch immer auf den Berechnungszeiten der letzten Szene, wodurch sie nicht perfekt sein kann. Der Kompromiss durch einen einfachen Algorithmus hat also mehr Vor- als Nachteile.
Die an die Slaves zu sendenen Aufteilungen sehen genauso aus, wie bei statischen Rechtecken. Nur der 7. Wert, also die ID des Aufteilers, ist diesmal 3.

4.3 Nachrichtenformate

Durch die Verteilung des Gesamtsystems auf mehrere Prozesse müssen Nachrichten zwischen diesen ausgetauscht werden, da nicht direkt auf den Speicher der anderen Prozesse zugegriffen werden kann. Diese Nachrichten werden mit MPI und Zero-MQ übertragen. Für jeden Zweck gibt es eine eigene Nachricht mit bestimmten Format. Diese sollen im folgenden vorgestellt werden.

4.3.1 VR-OOS Updates

Der Master empfängt über sein VR-OOS Modul Updates der Szene. Diese müssen an die Slaves weitergeleitet werden. Da dies innerhalb des Clusters geschieht, wird dafür MPI benutzt. Da jeder Slave die gleiche Nachricht benötigt, wird ein Broadcast verwendet (vgl. Kapitel 2.4.4). Die Nachricht setzt sich dabei aus zwei Teilen zusammen, die in einer eigenen Struktur zusammengefasst sind. Diese Struktur beinhaltet einen `Integer`, der die ID der veränderten Geometrie beinhaltet. Desweiteren enthält die Sturktur ein Array aus 12 `Floats`. In diesem Array ist die Transformationsmatrix enthalten, die von VR-OOS gesendet wurde.

4.3.2 Verteilung der Aufteilung

Der Master berechnet die Aufteilung des Bildes. Diese müssen den Slaves bekannt gegeben werden. Diese Nachrichten werden, wie die VR-OOS Updates, im Cluster verschickt. Es wird also MPI dafür benutzt. Da jeder Slave eine andere Aufteilung benötigt wird die Funktion Scatter verwendet (vgl. Kapitel 2.4.2). Es werden sieben `Integer` übertragen. Diese werden von dem jeweils benutzen Bildaufteiler beschrieben.

4.3.3 Einsammeln der Bildteile

Nach dem Berechnen der Teilbilder müssen diese an den Master gesendet werden, sodass dieser das Gesamtbild zusammen setzten kann. Dies geschieht innerhalb des Cluster, weshalb MPI verwendet wird. Da von mehreren Prozessen die Daten in einem Prozess gesammelt werden sollen, erscheint die Funktion Gather sinnvoll (vgl. Kapitel 2.4.3). Die benutzte Bibiliothek MVAPICH hat jedoch nur die blockierende Variante dieser Funktion implementiert. Da die Bilddaten relativ groß sind, ist es jedoch sinnvoll nicht-blockierende Aufrufe zu verwenden, um MPI die Möglichkeit zu geben, von allen Prozessen gleichzeitig zu kopieren. Aus diesem Grund wurden die nicht-blockierenden Varianten von Receive und Send benutzt (vgl. Kapitel 2.4.1).
Im Master werden die Receive-Aufrufe in einer Schleife für jeden Slave gestartet. Um die Längen der einzelnen Datensätze zu wissen, benutzt er die entsprechende Funktion des ausgewählten Bildaufteilers. Da alle Datensätze hintereinander in einen Speicherbereich geschrieben werden, benötigt der Master auch die jeweilige Startposition, welche er auch vom Bildaufteiler bekommt. Die Slaves benötigen nur jeweils einen Funktionsaufruf von Send. Die Länge ihres Datensatzes bekommen sie auch vom Bildaufteiler. Übertragen werden nur die berechneten Pixelwerte als `Unsigned Character`, wobei sie zeilenweise von links oben nach recht unten sortiert sind.

4.3.4 Einsammeln der Berechnungszeiten

Der dynamische Bildaufteiler benötigt die Rechenzeiten der einzelnen Slaves. Da die Bilddaten jedoch in einen zusammenhängenden Speicherbereich geschrieben werden sollen, ist es einfacher, die Zeiten nicht in der gleichen Nachricht zu übermitteln sondern eine eigene Nachricht dafür zu nutzen. Für diese wird MPI genutzt. Um die Struktur des Quellcodes auszunutzen, wurde dafür wieder Send und Receive genutzt. Der Master kann diese Funktionen in der gleichen Schleife wie die Funktionen zum Einsammeln der Bildteile aufrufen. Die Slaves müssen jeweils wieder nur einen Send-Aufruf starten. Der übertragende Wert ist jeweils eine Zeit in Millisekunden. Diese wird als einzelner `Integer` übertragen.

4.3.5 Streamen des Gesamtbildes an das Frontend

Nachdem das Bild berechnet wurde, muss es an das Frontend übertragen werden. Dies geschieht über ein normales Ethernet-Netzwerk. Zu diesem Zweck werden TCP-Socketsmittels Zero-MQ eingesetzt. Das Frontend stellt eine Verbindung zu der Netzwerkadresse des Backends her, welche ihm per Startparameter mitgeteilt werden muss. Jedes Bild wird in einer einzelnen Nachricht verschickt, wobei alle Daten als `Unsigned Character` interpretiert werden.
Die Nachricht besteht aus zwei Teilen. Der vordere Teil besteht aus den Bilddaten. Diese werden durch den eingesetzten Encoder bestimmt. Danach folgen Metainformationen für das Frontend. Da sich die Auflösung zwischendurch ändern kann, muss dies dem Frontend mitgeteilt werden. Dafür sind die ersten vier angehängten Werte da. Die ersten zwei davon bestimmten die Höhe des Bildes und die letzten zwei die Breite. Es sind jeweils zwei Werte, da der Bereich von `Unsinged Characters` nur von 0 bis 255 reicht, was jedoch für die benutzten Auflösungen nicht genug ist. Deshalb ist der jeweils erste Wert die eigentliche Höhe respektive Breite geteilt durch 256. Der jeweils zweite Wert ist der Rest dieser Ganzzahldivision. Der 5. Wert, der übertragen wird ist der aktuelle Bildzähler geteilt durch 256. Dieser wird nur für Debug-Zwecke gebraucht, z.B. ob das Frontend einen zu großen Abstand

zum aktuellen berechneten Bild auf dem Backend erreicht oder ob einzelne Bilder gar nicht ankommen.

5 Evaluierung

Nach der Implementierung eines funktionsfähigen verteilten, parallelen Raytracers ist es wichtig festzustellen, ob er im Hinblick auf den Stand-Alone Raytracer eine Verbesserung bringt oder nicht. Zu diesem Zweck werden Tests der Bildwiederholraten (engl.: Frames Per Second (FPS)) durchgeführt, um sie mit den Ergebnissen der vorangegangen Arbeit des Autors[3] zu vergleichen. Anschließend wird die Effizienz des Encoders und der implementierten Bildaufteiler untersucht.

5.1 Vergleich mit der Stand-Alone-Anwendung

Der verteilte, parallele Raytracer sollte aufgrund der höheren Rechenleistung, welche durch das benutzte Computercluster bereitgestellt wird, bessere Bildwiederholraten als der Stand-Alone Raytracer erreichen. Um dies zu überprüfen, wurden die gleichen vier Testszenen genutzt, welche in [3] untersucht wurden.

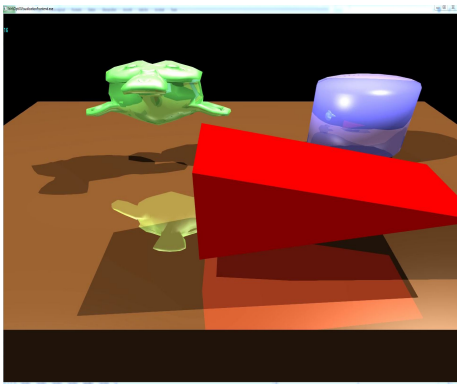

Abbildung 12 – Szene A: Standardtestszene aus 589 Dreiecken

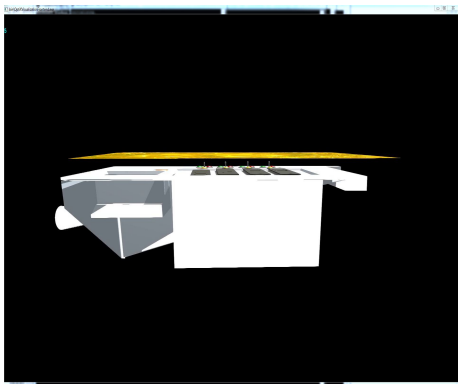

Abbildung 13 – Szene B: Satelliten-Mockup aus 7798 Dreiecken

In Abbildung 12 wird dabei Standardtestszene dargestellt. Abbildung 13 zeigt eine Szene mit einer mittleren Anzahl an Dreiecken. Diese wurde aufgrund einer ungünstigen Anordnung der Geometrien bereits früher mit einer geringeren Frequenz berechnet als die Szene mit den meisten Dreiecken, welche in Abbildung 14 gezeigt wird. Die vierte Szene, welche in Abbildung 15 gezeigt wird, hat 12

Lichtquellen, um so den Raytracer besonders zu fordern, da für jede Lichtquelle von jedem anzuzeigenden Punkt der Geometrien ein neuer Strahl generiert und berechnet werden muss. Diese vier Szenen wurden jeweils mit drei Auflösungen getestet: 640x480(typische Videoauflösung), 1024x786(typische Monitorauflösung) und 1920x1080(volle HD-Auflösung). Um gleichzeitig den Einfluss von unterschiedlich vielen Slaves zu messen wurden jede Auflösung und Szene mit 1, 4, 12 und 20 Slaves getestet. Während bei einem Slave ähnliche Raten wie beim Stand-Alone Raytracer zu erwarten sind und bei 20 Slaves aufgrund der dann zu teilenden Grafikkarten wahrscheinlich ein Einbruch der Raten eintritt, ist es vor allem interessant, ob OptiX alle in jedem einzelnen Computer vorhandenen Grafikkarten von sich aus nutzt. In diesem Fall müssten bei 4 Slaves ähnliche Raten wie bei 12 auftreten, da dann auch bei 4 Slaves alle 12 Grafikkarten benutzt werden. Da sich bei der Entwicklung bereits gezeigt hat, dass der H.264-Encoder und die dynamische Bildaufteilung am effektivsten sind, wurde diese Kombination für die Testreihe genutzt.

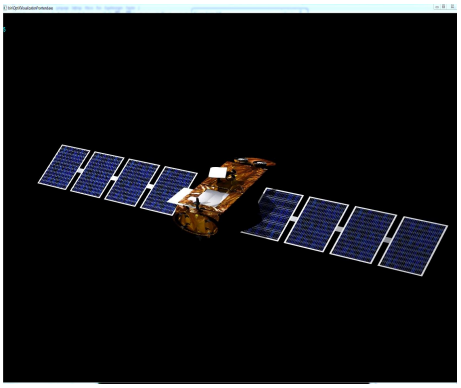

Abbildung 14 – Szene C: Satellit aus 44510 Dreiecken

Abbildung 15 – Szene D: 12 Lichter und 11916 Dreiecke

Alle Ergebnisse sind im Anhang in der Abbildung 23 aufgeführt. Die Ergebnisse aus der vorherigen Arbeit[3] sind im Anhang in der Abbildung 24 noch einmal dargestellt, wobei da auch der Einfluss verschiedener Effekte getestet wurde. Beim Test des parallelen Raytracers wurden jedoch immer beide Effekte (Schatten und

	Auflösung		
Prozesse	640x480	1024x786	1920x1080
Stand-Alone	25 FPS	10 FPS	5 FPS
1 Slave	25 FPS	13 FPS	8 FPS
4 Slaves	54 FPS	33 FPS	16 FPS
12 Slaves	40 FPS	30 FPS	17 FPS
20 Slaves	31 FPS	24 FPS	15 FPS

Tabelle 1 – Bildraten pro Sekunde der Szene C im Vergleich von Stand-Alone und unterschiedlicher Slaveanzahl

Reflektion) berechnet. In der Tabelle 1 sind die Bildraten für die Satellitenszene, bestehend aus 44510 Dreiecken, dargestellt. Sie zeigen, dass der entwickelte parallele Raytracer einen großen Forschritt für HD-Bilder hin zu interaktiven Bildraten darstellt. Die Verbesserung durch die Benutzung eines Clusters zeigt, dass der Ansatz eines parallelen Raytracers erfolgreich ist. Da Fluktuationen der Bildraten in einem produktiven Einsatz zu erwarten sind, werden 30 Bilder pro Sekunde angestrebt, sodass die Simulation durchgehen interaktiv sind. Die Optimierungen des Systems, um diese Leistung zu erreichen, sind jedoch wahrscheinlich kurzfristig möglich.

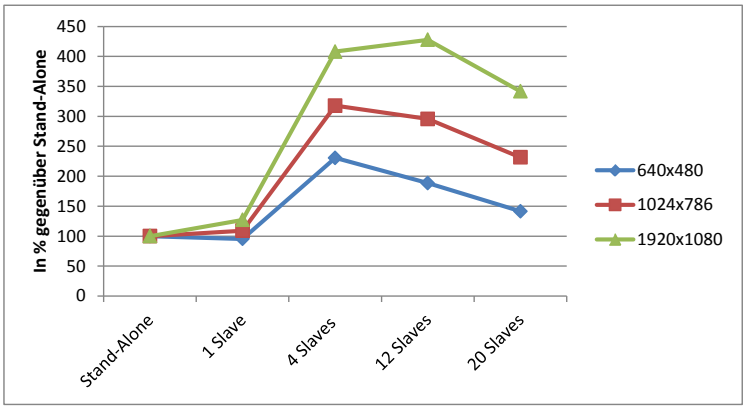

Abbildung 16 – Vergleich der durchschnittlichen Bildraten in Prozent gegenüber dem Stand-Alone Raytracer

Die Abbildung 16 zeigt die Verbesserung der durchschnittlichen Bildraten gegenüber

dem Stand-Alone Raytracer in Prozent, aufgeteilt nach Auflösung und Anzahl der benutzten Slaves. Es lässt sich erkennen, dass ein einzelner Slave ungefähr die Bildraten des Stand-Alone Raytracers erreicht. Der zeitliche Aufwand im verteilten, parallelen Raytracer für die Kommunikation und Synchronisierung ist also im Vergleich gering. Außerdem werden beim Einsatz von mehr Slaves als verfügbaren Grafikkarten nicht mehr die höchstmöglichen Bildwiederholraten erreicht. Die Frage, ob bei 4 Computern und 12 Grafikkarten 4 oder 12 Slaves benutzt werden sollten, kann nicht eindeutig geklärt werden. Bei geringer Auflösung ist der Kommunikationsaufwand im Vergleich zur eigentlichen Visualisierungsberechnung größer, was für eine geringere Anzahl an Slaves spricht. Bei einer höheren Auflösung hingegen ist der Anteil der Visualisierungsberechnung größer und mehr Slaves sind von Vorteil.

5.2 Vergleich Encoder - Rohdaten

Der Nutzen des implementierten Encoders im Vergleich zur Übertragung der Rohdaten an das Frontend wird durch eine eigene Testreihe untersucht. Um dies zu testen, wird eine Szene benötigt, bei welcher der Raytracer genügend hohe Bildwiederholraten erzeugt, um das Netzwerk besonders zu belasten. Aus diesem Grund wurde die Standardtestszene A genutzt. Diese wurden in den drei bereits in Kapitel 5.1 genannten Auflösungen jeweils mit und ohne Encoder getestet. Weiterdem wird der dynamische Bildaufteiler und 4 Slaves verwendet.

	Auflösung		
Komprimierung	640x480	7024x786	1920x1080
Rohdaten	32 FPS	14 FPS	5 FPS
H.264 Encoder	62 FPS	36 FPS	19 FPS

Tabelle 2 – Bildraten pro Sekunde bei verschiedenen Auflösungen

Die Ergebnisse dieser sechs Testdurchläufe sind in der Tabelle 2 aufgeführt. Dabei wird deutlich, dass das Netzwerk vom Cluster zum Frontend zum Flaschenhals wird. In Kapitel 2.6 wurde gezeigt, dass bei einer HD-Auflösung theoretisch maximal 15

Bilder pro Sekunde als Rohdaten über das Netzwerk transportiert werden können und das auch nur, wenn es keinen Protokollaufwand gäbe. In der Realität werden nur fünf Bilder pro Sekunde erreicht. Dieses Problem konnte durch den Einsatz des Encoders jedoch gelöst werden, mit dem fast viermal so viele Bilder pro Sekunde möglich sind, was zur Zeit den Berechnungsmöglichkeiten des Raytracers entspricht. Somit ist nicht das Netzwerk sondern die Geschwindigkeit des Raytracers die begrenzende Komponente des Systems.

5.3 Vergleich der Bildaufteiler

Eine Herausforderung in der Entwicklung des verteilten, parallelen Raytracers war die Aufteilung des zu berechnenden Bildes auf die einzelnen Slaves. Da der jeweils langsamste Slave die Gesamtbildrate bestimmt, ist es wichtig, die Berechnungszeiten der Slaves durch eine sinnvolle Aufteilung aneinander anzupassen. Dafür wurden drei Aufteiler implementiert, deren Effizienz durch Tests ermittelt wurden.

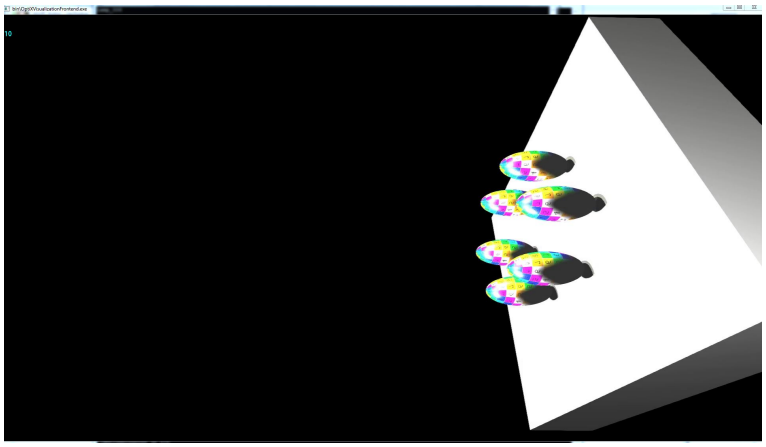

Abbildung 17 – An den Rand der Szene geschobene Geometrien

Da für aussagekräftige Ergebnisse lange Berechnungszeiten erforderlich sind, wurde die Szene D mit maximaler Komplexität genutzt, welche in Abbildung 15 gezeigt wird. Als Auflösung wurde 1920x1080 verwendet. Für den Transport der Daten wurde der Encoder eingeschaltet, um den Einfluss des Netzwerkes auf das Ergeb-

	Bildaufteiler		
Szene	Pixelspalten	statische Rechtecke	dynamische Rechtecke
Gleich verteilt	2 FPS	7 FPS	7 FPS
Ungleich verteilt	5 FPS	8 FPS	10 FPS

Tabelle 3 – Bildratenvergleich der Bildaufteiler bei gleich/ungleich verteilter Szene

nis gering zu halten. Die Szene wurde außerdem mit 4 Slaves getestet. Um die Aufteiler vergleichen zu können wurde die Szene D zuerst gleichmäßig über den Bildschirm verteilt und beim zweiten Durchlauf an den rechten Rand gesetzt, wie in Abbildung 17 gezeigt wird. Bei diesem zweiten Durchlauf sind bei der Aufteilung in statische Rechtecke nur die beiden rechten Rechtecke für die Berechnung der Geometrien verantwortlich und die beiden linken Rechtecke zeigen nur das Schwarz des Hintergrundes. Es sind also unterschiedliche Rechenzeiten der einzelnen Slaves zu erwarten.
Die Tabelle 3 zeigt die erreichten Bilder pro Sekunde für die einzelnen Konfigurationen an. Die Bildraten der gleich verteilten Szene dürfen dabei nicht direkt mit den Bildraten der ungleich verteilten Szene verglichen werden. Da die gleichverteilte Szene durch viel mehr Pixel als die ungleich verteilte Szene dargestellt wird, benötigt sie generell mehr Rechenzeit.
Aus dem Ergebnis lassen sich zwei Aussagen ableiten: Zum einen ist die Aufteilung in Pixelspalten der Breite 1 nicht geeignet. Dies lässt sich auf die Optimierungen von OptiX zurückführen. Das Framework bündelt ähnliche Strahlen, um sie performanter zu berechnen. Dies ist mit dieser Aufteilung jedoch nicht mehr möglich. Die zweite Aussage bezieht sich auf die Aufteilugn in Rechtecke: Bei einer Szene, in der die Geometrien über das gesamte Bild annähernd gleich verteilt sind, sind die statischen Rechtecke so leistungsfähig wie die dynamischen. Bei einer ungleichen Verteilung hingegen haben die dynamischen Rechtecke einen Vorteil und bringen 25% mehr Bilder pro Sekunde.
Da die Bildaufteiler vor allem Einfluss auf die einzelnen Berechnungszyklen der Slaves haben, wurden diese noch genauer untersucht. In der Abbildung 18 sind die Durchschnittszeiten für einen Berechnungsdurchlauf aller Slaves für jeweils eine

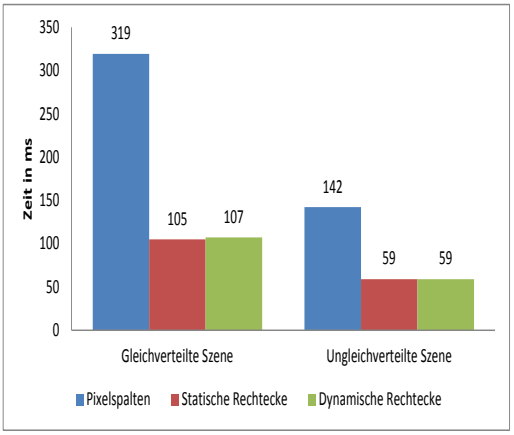

Abbildung 18 – Durchschnittszeiten der Renderzyklen der Slaves

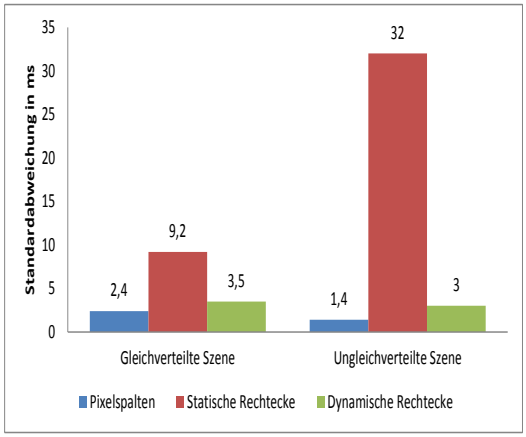

Abbildung 19 – Standardabweichung der einzelnen Renderzyklen

Konfiguration zu sehen. Darin wird wieder deutlich, dass die Aufteilung in Pixelspalten deutlich schlechter ist. Die statischen und dynamischen Rechtecke hingegen erreichen in etwa gleiche Durchschnittszeiten. Um die Differenz in den Bildraten zu erklären ist deshalb die Abbildung 19 interessant, in der die Standardabweichungen der einzelnen Berechnungszeiten getrennt nach Aufteiler dargestellt werden. Während die Pixelspalten und dynamischen Rechtecke eine geringe Standardabweichung aufweisen, gibt es bei der ungleich verteilten Szene mit den statischen Rechtecken große Abweichungen der einzelnen Berechnungsdurchläufe. Für ein genaueres Verständnis dieser Ergebnisse sind die durchschnittlichen Berechnungszeiten und die Standardabweichung von den einzelnen Slaves wichtig. Diese sind in Abbildung 20 für die Aufteilung mit Pixelspalten, in Abbildung 21 für die Aufteilung mit statischen Rechtecken und in Abbildung 22 für die Aufteilung mit dynamischen Rechtecken aufgeführt.
Dabei wird deutlich, dass die Pixelspalten und dynamische Aufteilung sehr ausgeglichene Zeiten über alle Slaves ermöglichen. Bei den statischen Rechtecken hingegen ist gut zu sehen, welche beiden Slaves in der ungleich verteilten Szene für den Ausschnitt mit den Geometrien verantwortlich sind. Selbst bei der annähernd gleich verteilten Szene sind Unterschiede aufgrund leicht unterschiedlicher Komplexität

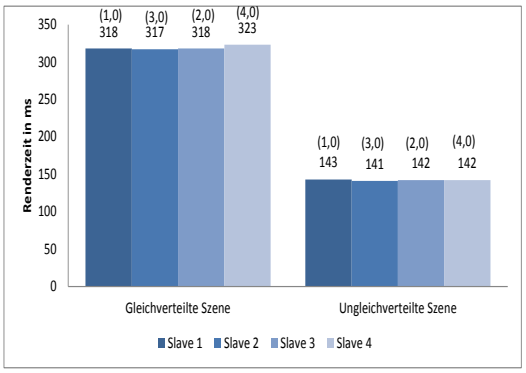

Abbildung 20 – Berechnungszeiten der einzelnen Slaves mit dem Pixelaufteiler sowie Standardabweichung in Klammern

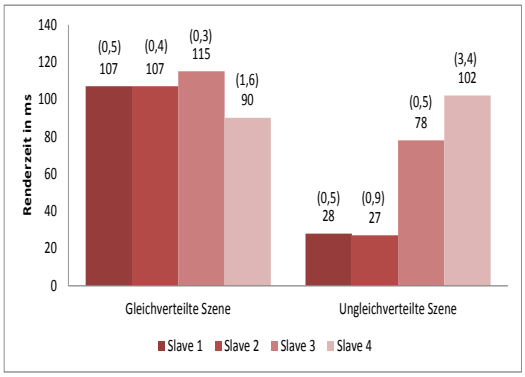

Abbildung 21 – Berechnungszeiten der einzelnen Slaves mit statischem Aufteiler sowie Standardabweichung in Klammern

der einzelnen Bildausschnitte erkennbar. Die Standardabweichungen sind bei der dynamischen Aufteilung am höchsten. Dies ist durch die Arbeitsweise des Aufteiler erklärbar: Erst wenn die einzelnen Slaves unterschiedlich lange brauchen und somit Abweichungen vom Durchschnitt entstehen, werden die Aufteilungen angepasst.

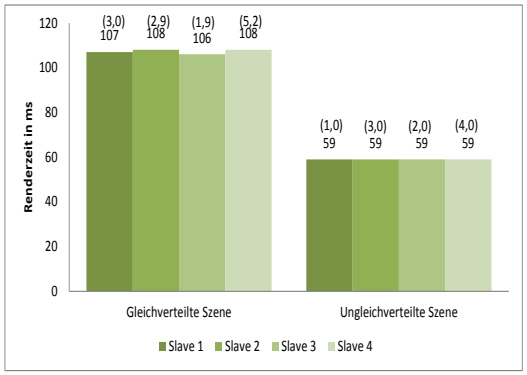

Abbildung 22 – Renderzeiten der einzelnen Slaves mit dem dynamischen Aufteiler sowie Standardabweichung in Klammern

6 Zusammenfassung und Ausblick

Die vorliegende Arbeit zeigt, wie ein Raytracer als verteiltes, paralleles System implementiert werden kann, um die Rechenleistung mehrerer Computer oder eines Clusters zu nutzen. Die Ergebnisse aus den einzelnen Tests des entwickelten Raytracers zeigen, dass dieser eine große Verbesserung gegenüber dem Stand-Alone Raytracers ist. Die Visualisierung ist nun im Schnitt vier mal so schnell und Szenen mit einer mittleren Komplexität lassen sich nun auch in HD-Auflösung mit annähernd interaktiven Bildwiederholraten berechnen. Das entworfene Design erlaubt eine einfache Implementierung von alternativen Encodern und Bildaufteilern, um die Leistung noch zu steigern. Bei den Bildaufteilern, einem zentralen Aspekt eines parallelen Raytracers, wurden verschiedene Ansätze implementiert und getestet. Die Aufteilung in einzelne Pixelspalten hat zwar, wie erhofft, zu einer sehr gleichmäßigen Rechenzeit der einzelnen Slaves geführt, jedoch wurden damit die schlechtesten Bildraten erreicht. Dies liegt an internen Optimierungen des Frameworks OptiX, welches für die Implementierung auf der Grafikkarte genutzt wurde. Eine Aufteilung in statische Rechtecke zeigte gute Bildraten, jedoch waren die einzelnen Berechnungszeiten der Slaves bei ungünstigen Szenen weit auseinander. Der implementierte Aufteiler mit dynamischen Rechtecken konnte dieses Problem lösen. Der eingesetzte H.264 Encoder ist in der Lage genug, die Bilder ohne Drosselung an das Frontend zu übertragen. Insgesamt wurde das Ziel erreicht, einen leistungsfähigen verteilten, parallelen Raytracer zu entwickeln.
Um die Leistung noch weiter zu steigern, gibt es verschiedene Ansatzmöglichkeiten, die in der Zukunft untersucht werden sollten. Zum einen können die Algorithmen zum Zusammensetzten der Teilbilder im Master so verbessert werden, dass sie durch Multithreading beschleunigtwerden können. Desweiteren erlaubt die plattformunabhängige Implementierung durch MPI die Möglichkeit, noch mehr Computer einzusetzten, die nicht im expliziten Cluster verbunden sind. Dadurch könnten auch gerade unbenutzte Workstations von Mitarbeitern benutzt werden. Die wahrscheinlich größte Verbesserung kann jedoch durch eine Änderung in den Slaves erreicht werden. Diese warten auf Anweisungen für die Aufteilung des Bildes vor jedem

Berechnungsdurchlauf. Wenn sie jedoch bereits das nächste Bild berechnen würden und dieses sofort an den Master schicken, wenn die Aufteilung wieder die gleiche wie beim letzten Bild ist, würden die Slaves weniger Zeit im Leerlauf verbringen. Dafür müsste dann auch der dynamische Bildaufteiler angepasst werden, sodass nur bei großen Laufzeitunterschieden eine neue Aufteilung berechnet wird. Durch diese Verbesserungen könnten bereits kurzfristig bemerkenswerte Verbesserungen der Bildwiederholraten erreicht werden. Langfristig verspricht auch die dauerhaft besser werdene Hardware im Grafikkartenbereich den Einsatz von Raytracern für die Berechnung von einer Virtuellen Realität in Echtzeit.

Literatur

[1] Phrood, Wikipedia: *Raytracing - Schattenstrahl*. http://commons.wikimedia.org/wiki/File:Raytracing-Schattenstrahl.svg, Abruf: 16.09.2013

[2] Wolff, Robin ; Preusche, Carsten ; Gerndt, Andreas: A Modular Architecture for an Interactive Real-Time Simulation and Training Environment for Satellite On-Orbit Servicing. In: *Journal of Simulation*, Operational Research Society, 2013, in press

[3] Berthold, Frieder: Erweiterung eines Echtzeit-Raytracing Moduls zur fotorealistischen Darstellung in VR Anwendungen / Deutsches Zentrum für Luft- und Raumfahrt e. V. in der Helmholtz-Gemeinschaft. 2013. – Forschungsbericht

[4] Wolff, Robin ; Preusche, Carsten ; Gerndt, Andreas: A Modular Architecture for an Interactive Real-Time Simulation and Training Environment for Satellite On-Orbit Servicing. In: *Proceedings of the 2011 IEEE/ACM 15th International Symposium on Distributed Simulation and Real Time Applications*. Washington, DC, USA : IEEE Computer Society, 2011 (DS-RT '11). – ISBN 978–0–7695–4553–0, 72–80

[5] NVIDIA: *OptiX*. http://www.nvidia.com/object/optix.html, Abruf: 20.04.2013

[6] Baker, Mark: Cluster Computing White Paper. In: *CoRR* cs.DC/0004014 (2000)

[7] Association, InfiniBand T.: *About InfiniBand*. http://www.infinibandta.org/content/pages.php?pg=about_us_infiniband, Abruf: 15.09.2013

[8] Mallón, DamiánA. ; Taboada, GuillermoL. ; Teijeiro, Carlos ; Touriño, Juan ; Fraguela, BasilioB. ; Gómez, Andrés ; Doallo, Ramón ; Mouriño, J.Carlos: Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. Version: 2009. http://dx.doi.org/10.1007/978-3-642-03770-2_24. In: Ropo, Matti (Hrsg.) ; Westerholm, Jan (Hrsg.) ; Dongarra, Jack (Hrsg.): *Recent Advances in Parallel Virtual Ma-*

chine and Message Passing Interface Bd. 5759. Springer Berlin Heidelberg, 2009. – DOI 10.1007/978–3–642–03770–2_24. – ISBN 978–3–642–03769–6, 174-184

[9] Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard, Version 3.0*. High-Performance Computing Center Stuttgart, 2012

[10] Dept of Computer Science and Engineering: *MVAPICH*. http://mvapich.cse.ohio-state.edu/overview/mvapich2/, Abruf: 10.09.2013

[11] Postel, J.: *User Datagram Protocol*. http://www.ietf.org/rfc/rfc768.txt, Abruf: 11.09.2013

[12] Postel, J. et al.: *Transmission Control Protocol*. http://tools.ietf.org/html/rfc793, Abruf: 11.09.2013

[13] Odom, ChristianN.S. ; Shetty, NikhilJ. ; Reiners, Dirk: Ray Traced Virtual Reality. Version: 2009. http://dx.doi.org/10.1007/978-3-642-10331-5_96. In: Bebis, George (Hrsg.) ; Boyle, Richard (Hrsg.) ; Parvin, Bahram (Hrsg.) ; Koracin, Darko (Hrsg.) ; Kuno, Yoshinori (Hrsg.) ; Wang, Junxian (Hrsg.) ; Wang, Jun-Xuan (Hrsg.) ; Wang, Junxian (Hrsg.) ; Pajarola, Renato (Hrsg.) ; Lindstrom, Peter (Hrsg.) ; Hinkenjann, André (Hrsg.) ; Encarnação, MiguelL. (Hrsg.) ; Silva, CláudioT. (Hrsg.) ; Coming, Daniel (Hrsg.): *Advances in Visual Computing* Bd. 5875. Springer Berlin Heidelberg, 2009. – DOI 10.1007/978–3–642–10331–5_96. – ISBN 978–3–642–10330–8, 1031-1042

[14] Sullivan, G.J. ; Wiegand, T.: Video Compression - From Concepts to the H.264/AVC Standard. In: *Proceedings of the IEEE* 93 (2005), Nr. 1, S. 18–31. http://dx.doi.org/10.1109/JPROC.2004.839617. – DOI 10.1109/JPROC.2004.839617. – ISSN 0018–9219

[15] VideoLAN: *x264*. http://www.videolan.org/developers/x264.html, Abruf: 19.09.2013

[16] Mantler, Stephan ; Schmalstieg, Dieter: Dynamic Load Balancing in Distributed Virtual Environments / Institute of Computer Graphics and Algo-

rithms, Vienna University of Technology. Version: April 1998. http://www.cg.tuwien.ac.at/research/publications/1998/Mantler-1998-DynX/. Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, April 1998 (TR-186-2-98-13). – Forschungsbericht. – human contact: technical-report@cg.tuwien.ac.at

[17] Plachetka, Tomas: Perfect Load Balancing for Demand- Driven Parallel Ray Tracing. Version: 2002. http://dx.doi.org/10.1007/3-540-45706-2_56. In: Monien, Burkhard (Hrsg.) ; Feldmann, Rainer (Hrsg.): *Euro-Par 2002 Parallel Processing* Bd. 2400. Springer Berlin Heidelberg, 2002. – DOI 10.1007/3–540–45706–2_56. – ISBN 978–3–540–44049–9, 410-419

A Anhang

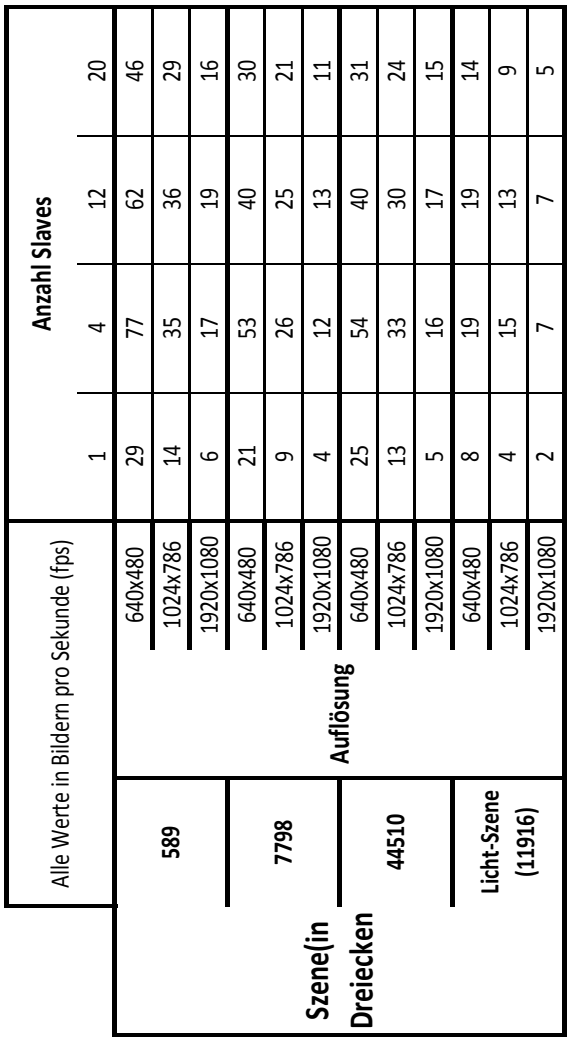

Abbildung 23 – Messungen der Bildraten des parallelen Raytracers

			PC							
			Student Geforce 460 GTX			Alienware Geforce 480 GTX				
			Auflösung							
	Alle Werte in Bildern pro Sekunde (fps)		640x480	1024x786	1920x1080	640x480	1024x786	1920x1080		
Szene(in Dreiecken	589	Effekte	Keine	53	22	9	59	33	14		
			Schatten	33	13	6	52	22	10		
			Reflektion	41	17	7	59	27	11		
			Beides	25	10	4	42	18	8		
	7798		Keine	13	5	2	25	11	5		
			Schatten	8	3	1	17	7	3		
			Reflektion	13	5	2	25	11	5		
			Beides	8	3	1	17	7	3		
	44510		Keine	23	10	4	40	17	7		
			Schatten	16	6	3	25	10	5		
			Reflektion	24	10	4	40	17	7		
			Beides	15	6	3	25	10	5		
Licht-Szene (11916)			1 Licht	27	11	5	41	19	8		
			3 Lichter	17	7	3	29	12	5		
			12 Lichter	5	2	1	9	4	1		

Abbildung 24 – Messungen der Bildraten des Stand-Alone Raytracers