Institut für Informatik
AG Medieninformatik

Bachelorarbeit

Plattformunabhängige Darstellung von High-Quality-3D-Grafiken mittels Remote Rendering

Christoph Eichler

Juli 2014

Erstgutachter: Prof. Dr. Oliver Vornberger
Zweitgutachterin: Prof. Dr. Elke Pulvermüller

Danksagung

Hiermit möchte ich allen Personen danken, die mich bei der Fertigstellung dieser Arbeit unterstützten:

- Herrn Prof. Dr. Oliver Vornberger für seine Tätigkeit als Erstgutachter.
- Frau Prof. Dr. Elke Pulvermüller, die sich als Zweitgutachter zur Verfügung gestellt hat.
- Henning Wenke, der mich durch sein konstruktives Feedback innerhalb zahlreicher Gespräche sehr unterstützte und mir dieses interessante Thema vorschlug.
- Sascha Kolodzey für die Bereitstellung seiner Grafik-Engine und die Unterstützung bei der Einbindung der RR-API. Sascha war auch bei Problemstellungen anderer Art stets ansprechbar und hat mir durch sein präzises Feedback sehr geholfen. Vielen Dank!
- Timo Bourdon und Erik Wittkorn für die Korrektur dieser Arbeit und Ismael Bergmann für das Bereitstellen seines Rechners zu zahlreichen Latenzmessungen.
- Meiner Freundin Lina, die mir während der Bearbeitungsphase den Rücken freigehalten hat und diese Arbeit ebenfalls korrigierte.
- Meinen Eltern, die mich immer unterstützen und mir dieses Studium ermöglicht haben.

Zusammenfassung

Unter Remote Rendering versteht man das Auslagern hardwareintensiver Grafikberechnungen auf externe Server. Über das Internet können Nutzer mit den Rechenservern kommunizieren und mögliche grafische Szenen durch ihre Eingabe manipulieren. Das Bild der manipulierten Szene wird zum Nutzer zurück geschickt und dort angezeigt.
Durch diese Technik soll es in Zukunft möglich sein, qualitativ hochwertige Szenen auf mobilen Endgeräten, wie z.B. Smartphones, Tablets oder Notebooks anzuzeigen. Das Ziel dieser Arbeit ist die Implementierung einer Remote Rendering API, die es Entwicklern grafischer Desktop-Applikationen erlauben soll ihre Programme um Remote Rendering Funktionalitäten zu erweitern. Weiterhin soll das Verhalten der Remote Rendering API mit den Lösungen und Resultaten von Nvidia verglichen werden.

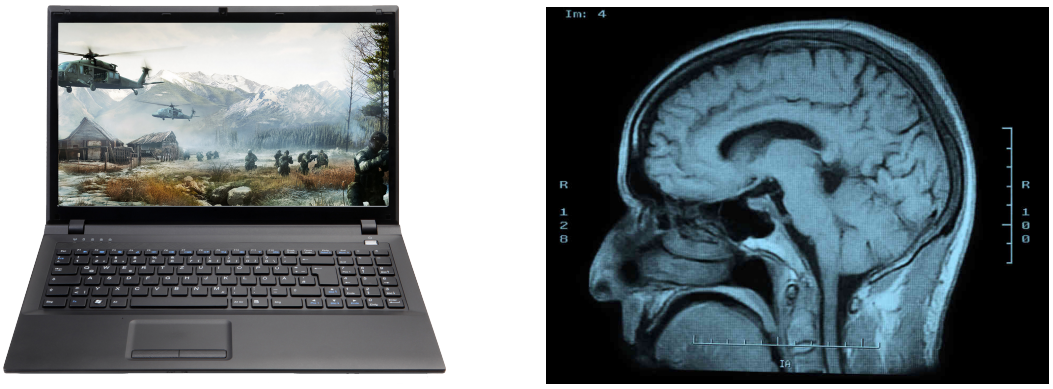
Abstract

The term Remote Rendering describes the outsourcing of hardwareintensive graphics computation to external servers. Users can communicate via Internet with these and, due to their input, manipulate possible graphical scenes. A „Screenshot“ of the manipulated Scene is sent back and displayed to the user.
In the future, this technology shall allow the rendering of high quality scenes on mobile devices, such as Smartphones, Tablets or Notebooks. The aim of this work is to implement a Remote Rendering API, that extends the functionalities of a common desktop application with Remote Rendering capabilities. Furthermore the properties of this application shall be compared to the solution of Nvidia.

Inhalt

1 Einleitung

Grafische Inhalte gewinnen immer mehr an Bedeutung. Im Bereich der Spieleentwicklung sind grafisch anspruchsvolle Szenen heute beliebter denn je. Auch im Bereich von Smartphones und Tablets sind sie auf dem Vormarsch, können jedoch qualitativ nicht mit der Anzeige eines leistungsstarken Desktop-PCs konkurrieren, denn die hohe Mobilität dieser Geräte wird durch kompakte Gehäuse und damit nur wenig Platz für leistungsstarke Hardware ermöglicht.
Mit Hilfe des Remote Renderings soll es in Zukunft möglich, sein anspruchsvolle grafische Inhalte, unabhängig von der zu Grunde liegenden Hardware, darzustellen und mit ihnen zu interagieren. Dafür werden komplexe Berechnungen auf externe Server ausgelagert und lediglich das berechnete Bild auf dem Endgerät, als eine Art interaktives Video, angezeigt. Prinzipiell kann jedes Gerät für Remote Rendering verwendet werden, das ein kodiertes Video abspielen kann, wie sie zum Beispiel vom Online-Portal Youtube bereitgestellt werden.
Eine große Rolle bei dieser Technik spielt die Latenz. Unter diesem Terminus versteht man die Zeitdifferenz zwischen einer Nutzereingabe und dem audiovisuellen Feedback. Abhängig von der Höhe der Latenz entstehen zwei Anwendungsszenarien:



(a) latenzsensibel **(b)** wenig latenzsensibel

Abbildung 1: Beispiele für zwei mögliche Anwendungen von Remote Rendering

In Abbildung 1a wird ein modernes Computerspiel gezeigt. Es steht stellvertretend für eine Anwendung, die eine möglichst geringe Latenz erfordert, um ein schnelles Feedback zu gewährleisten. Der Fokus dieser Arbeit liegt insbesondere auf Anwendungen dieses Typs. Statische Anwendungen, wie z.B. die Visualisierung eines Magnetresonanztomographen-Scans (s. Abb 1b) werden auch von einer höheren Latenz nicht in ihrer Funktionalität beeinträchtigt. Diese Gruppe von Anwendungen profitiert insbesondere davon, dass Ergebnisse ungebunden von stationären Systemen einsehbar und manipulierbar wären.

1.1 Motivation

Der Sektor der mobilen Devices boomt: Hohe Mobilität, geringe Anschaffungskosten und ausreichend multimediale Fähigkeiten motivieren den Kauf eines solchen Gerätes. Unter dieser

Entwicklung leiden die Absatzmärkte der Hersteller von Desktop-PC- und Server-Hardware[1]. In Abbildung 2 wird gezeigt, dass der Absatz von Notebook-PCs und Desktop-PCs seit 2010 leicht zurückgeht, der von Tablets und Smartphones dagegen geradezu explodiert. Weiterhin wächst der Markt von Smartphone- und Tablet-Spielen stetig. So betrug der kombinierte Jahresumsatz von Android Spielen im Jahr 2013 beispielsweise 1,77 Billionen US-Dollar[2].
In diese Entwicklung stößt die Technik des Remote Renderings, durch die die vergleichsweise schwache Hardware mobiler Endgeräte durch den Einsatz externer Rechenserver kompensiert werden soll.

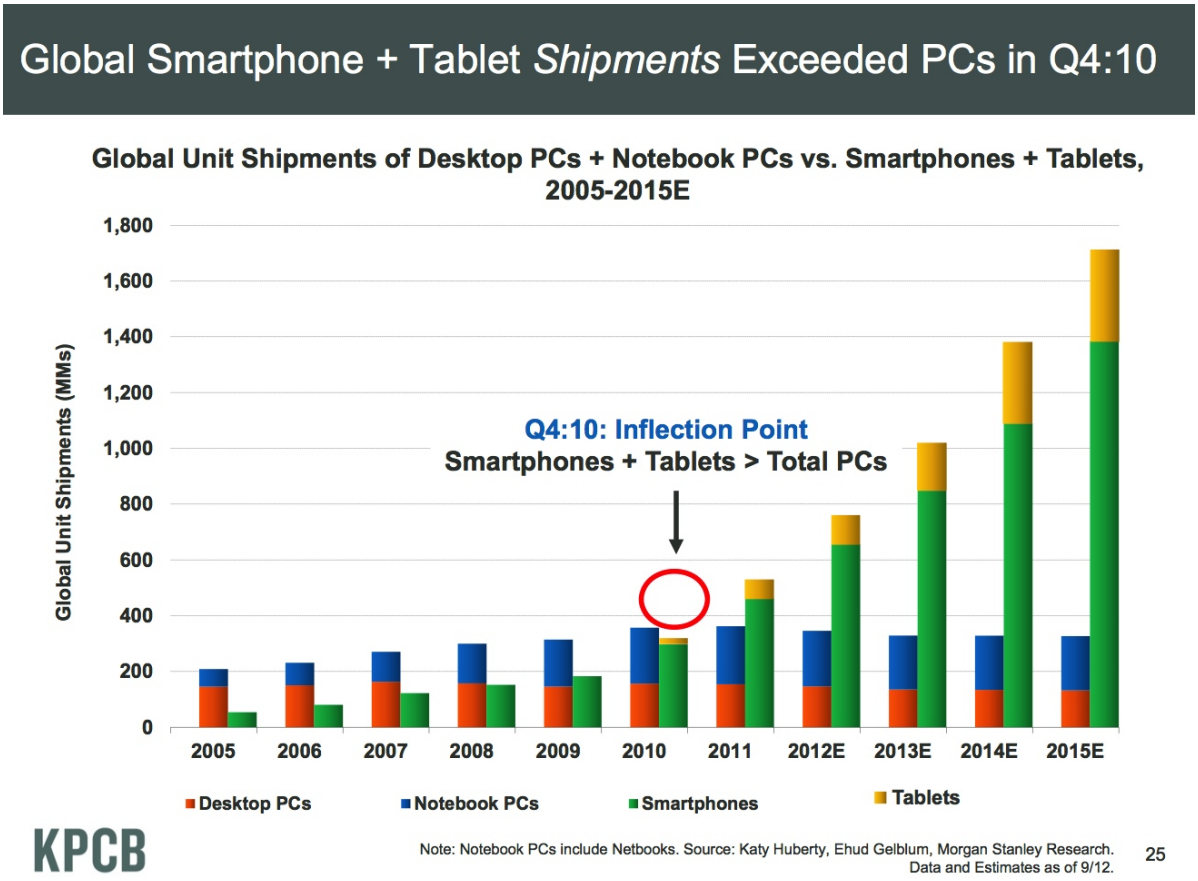


Abbildung 2: Verkaufte Devices der letzten zehn Jahre [3]

Aktuell sollte bei der Entwicklung einer grafischen Desktop-Applikation eine möglichst große Skalierbarkeit erreicht werden, das heißt, dass die Applikation auf möglichst vielen, unterschiedlich leistungsstarken PCs, Smartphones oder Tablets ausführbar sein sollte. Abhängig von der jeweiligen Leistungsfähigkeit werden qualitativ angepasste Grafiken synthetisiert. Um dies zu erreichen werden gleiche Inhalte verschiedener Qualität, wie zum Beispiel Texturen unterschiedlicher Auflösung, mitgeliefert und so die Größe der Applikation erhöht. Zusätzlich muss die Applikation auf möglichst vielen verschiedenen Hardwarekonfigurationen getestet

[1] http://techhive.de/intel-entlaesst-diesem-jahr-5000-mitarbeiter-205324546/ (02.04.2014)
[2] http://www.gamezone.de/screenshots/667x375/2013/07/Slide1.PNG (22.05.2014)
[3] http://digitalintelligencetoday.com/wp-content/uploads/2013/05/screenshot_1866.jpg (02.04.2014)

werden, um Fehlerquellen auszuschließen. Durch die Einführung von Remote Rendering würde diese Notwendigkeit entfallen. Grafische Applikationen müssten nur auf die Leistungsfähigkeit der eingesetzten Server optimiert und gegen deren Hardwarespezifikationen getestet werden. Die verbundenen Endgeräte könnten die Inhalte dann hardwareunabhängig in hoher Qualität wiedergeben.
Mit Nvidia ist ein großer Hersteller von Grafikkarten in die Entwicklung eines Remote Rendering Dienstes eingestiegen und präsentiert seinen *Nvidia GRID* genannten Dienst als eine Lösung zur verzögerungsfreien Anzeige grafischer Inhalte.

1.2 Zielsetzung

Ziel dieser Arbeit ist die Implementierung eines Remote Renderers, bestehend aus einem Grafikserver und zwei plattformspezifischen Clienten. Dabei sollen insbesondere moderne Techniken, wie massiv parallele Bildkompression mittels Cuda nach H264-Standard, Bilderzeugung durch Interaktion zwischen OpenGL und Cuda, sowie die Entwicklung eines schlanken Kommunikationsprotokolls Einzug finden.
Ein besonderer Schwerpunkt liegt dabei auf der Entwicklung einer API, die andere Entwickler in ihr Projekt einbinden können, um grafische Desktop Applikationen um Remote Rendering Funktionalitäten zu erweitern. Der Vorteil dieses Vorgehens liegt darin, dass viele schon bestehende Anwendungen einfach erweitert werden können. Die Clienten, die in dieser Arbeit entwickelt werden, sollen alle, von der API versandten, grafischen Inhalte darstellen können. Somit wird ein Softwarepaket angeboten, dass eine vollständige Durchführung von Remote Rendering erlaubt.
Einen ersten denkbaren Anwendungsfall dieser API stellt die Masterarbeit von Timo Bourdon dar. In ihr wird ein Automobilkonfigurator (siehe [8]) unter anderem um eine Remote Rendering Komponente erweitert.

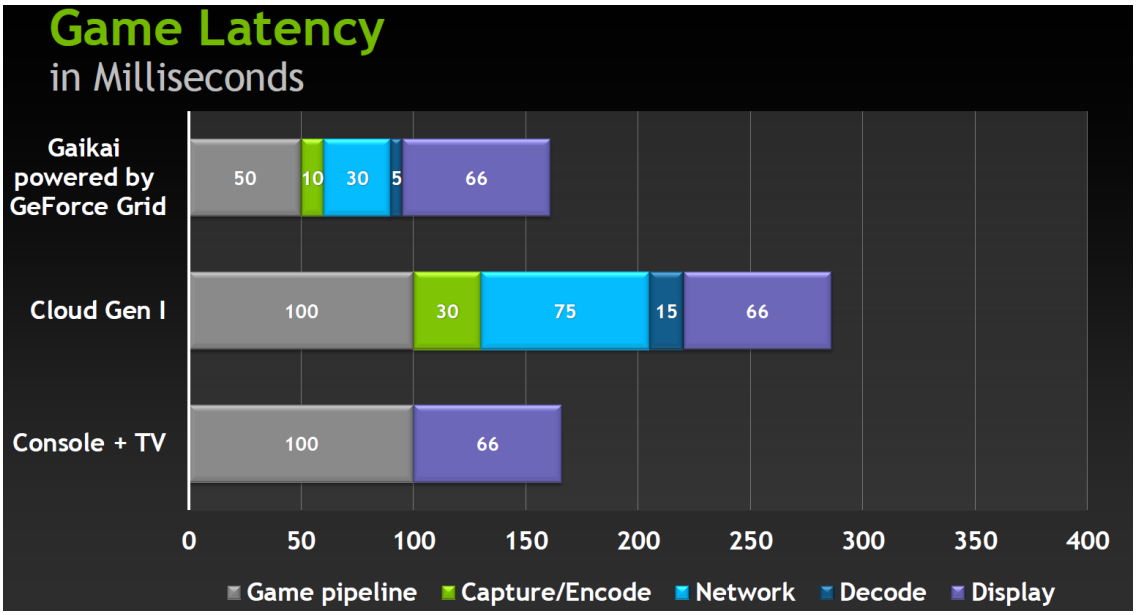


Abbildung 3: Mögliche Latenzzeiten nach Nvidia [3]

Ein weiteres Ziel ist die Untersuchung der so entstehenden Latenz. Da der Spielemarkt nur erschlossen werden kann, wenn die Latenz unter einer für Menschen wahrnehmbaren Schwelle liegt, sind diese Messungen von besonderer Bedeutung. Als Vergleichswert soll Abbildung 3 dienen, in der die Latenz von Nvidias *GRID* gezeigt wird.

2 Kommerzielle Anbieter

2.1 OnLive

OnLive ist eine US-amerikanische Firma, die 2004 von Steve Perlman in Palo Alto (USA) gegründet wurde. Das Unternehmen spezialisiert sich auf Cloud Gaming, worunter das Streamen von, auf Servern berechneten, Spielen (s. Abb. 1a, S. 1) verstanden wird. 2009, nach insgesamt sieben Jahren Entwicklungsdauer, wurde die Technologie der Öffentlichkeit auf der *Game Developers Conference* in San Francisco vorgestellt. Schon damals wurde der Dienst von einigen Vertretern der Branche als „Revolution“ hervorgehoben: Der Zwang zum Kauf moderner Hardware zur Wiedergabe aktueller Computerspiele unterbleibt, Patches werden zentral durch den Dienstleister verteilt und außerdem ist das Betrügen bei Onlinespielen, durch das Vorhalten der Spieledateien auf den Servern, deutlich erschwert.
Den Kunden stehen zur Nutzung des Dienstes einige Bezahlmodelle zur Verfügung: Es gibt die Option einen Spielepass für 3 Tage, 5 Tage oder 3 Jahre zu erwerben. Weiterhin kann eine Flatrate für alle Spiele erworben werden. Davon abgesehen fallen jedoch keine weiteren Kosten zur Nutzung des Dienstes an. Aktuell ist OnLive nur in den USA und Großbritannien verfügbar. Dies ist der Technik OnLives geschuldet. Von den Entwicklern wird eine maximale Distanz von 625km zum nächsten Rechenzentrum empfohlen, damit die Latenz den erwünschten Maximalwert nicht übersteigt. Somit hat die Expansion in weitere Länder den Bau neuer Serverfarmen zur Folge.
Eine Besonderheit OnLives ist der eigens entwickelte Kompressionsalgorithmus[4]. Dieser erlaubt das Streamen von Spielen der Auflösung 1280 x 720 bei einem Bandbreitenverbrauch von nur 5Mbit pro Sekunde. Nach Herstellerangaben soll somit eine Grafikqualität, vergleichbar mit der von aktuellen Konsolen und High-End PCs, erreicht werden. Letztendlich stimmt dies jedoch nur bedingt, da eine Auflösung von 1280x720 nicht mit der heute verwendeten Full HD Auflösung 1920x1080 konkurrieren kann. Weiterhin soll sich nach Angaben der Zeitschrift GamePro[5] ein „Unschärfeschleier über das Bild legen“. Dieser Nachteil ergibt sich aus der Bildkompression.

2.2 Gaikai

Genau wie OnLive, stammt das Unternehmen Gaikai aus der Branche des Cloud Gamings. 2008 von Andrew Gault und Rui Pereira gegründet, hat es seinen Hauptsitz in Aliso Viejo (USA). Im Gegensatz zu OnLive tritt Gaikai nicht direkt an den Kunden heran, sondern kooperiert mit anderen Unternehmen. 2012 verkündeten unter anderem Samsung und LG, dass deren Smart-TVs in Zukunft das Angebot von Gaikai unterstützen werden. Käufern

[4] http://www.ign.com/articles/2009/03/24/gdc-09-onlive-introduces-the-future-of-gaming (24.05.2014)
[5] http://www.gamepro.de/artikel/spiele-specials/1966283/onlive_erster_test_p3.html (28.04.2014)

dieser TVs soll es durch ein einfaches Softwareupgrade ermöglicht werden, auf dem Fernseher gestreamte Spiele zu spielen. Lediglich ein Gamepad, sowie eine permanente Internetverbindung mit mindestens 5MBit/s Datenrate sollen die Vorraussetzung sein.
In den medialen Fokus gelang das Unternehmen erneut 2012, als Sony bekannt gab, dass Gaikai für etwa 380 Millionen US-Dollar übernommen werde. Ziel dieser Übernahme war unter anderem die Integration der Technologie Gaikais in die neue Spielekonsole von Sony: Der Playstation 4. Deren Hardware unterstützt, anders als die Vorgänger, nicht mehr das Spielen von Titeln älterer Playstation Modelle. Zum Ausgleich sollen Spiele älterer Generationen mit Hilfe des Gaikai Dienstes auf die Playstation 4 gestreamed werden. Auf der CES (*Consumer Electronics Show*) 2014 in Las Vegas wurde dieser Dienst nun erstmals als *Playstation Now* vorgestellt. Im Sommer 2014 soll der Dienst in den USA starten, in Europa dagegen voraussichtlich erst 2015. Grund hierfür ist die schlechte Breitbandversorgung[6].

2.3 Valve

Auch Valve, der Entwickler der großen Internet-Vertriebsplattform Steam, arbeitet mittlerweile an einer Variante des Remote Renderings. Ziel ist dabei allerdings nicht das Bereitstellen von Serverhardware auf der aufwändige Spiele berechnet werden können, vielmehr bietet dieser Dienst sogenanntes In-Home Streaming an. Ein schon vorhandener Desktop PC kann zum Beispiel zur Berechnung, ein Fernseher zur Anzeige und ein Gamepad zur Steuerung des Spiels verwendet werden. Dargestellt ist das Prinzip in Abbildung 4. Nach Beendigung der Testphase soll der Dienst in den Steam-Clienten integriert werden.

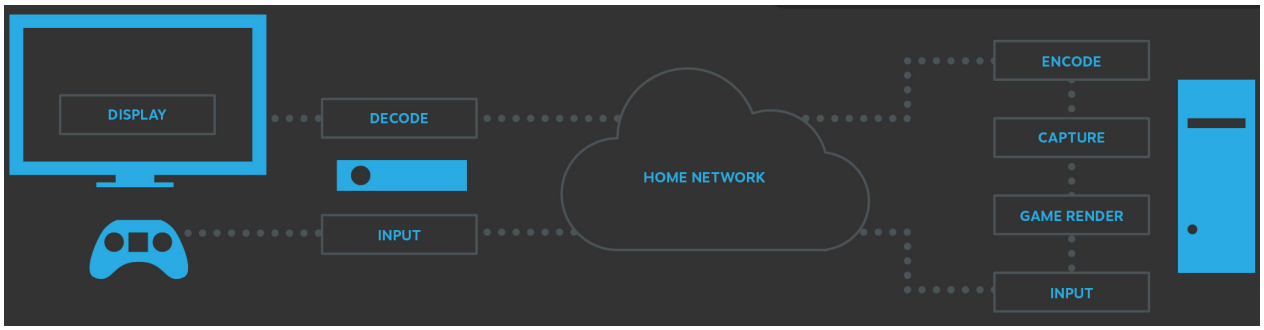


Abbildung 4: Valves In-Home Streaming[7]

[6] http://www.videogamer.com/news/ps4_gaikai_coming_to_europe_in_2015.html (24.05.2014)
[7] http://media.steampowered.com/apps/steamhardware/InHomeStreamingDiagram.jpg (05.05.2014)

3 Eigener Ansatz

Remote Rendering zeichnet sich durch eine strikte Aufgabenverteilung zwischen Client und Server aus. Abbildung 5 zeigt die Client-Server-Architektur von Nvidias *GRID*. Um vergleichbare Messergebnisse zu erzeugen, wird eine ähnliche Architektur gewählt. Spezifische Funktionalitäten des Remote Renderings werden jedoch in einer API gekapselt, wie in Abbildung 6 zu sehen.

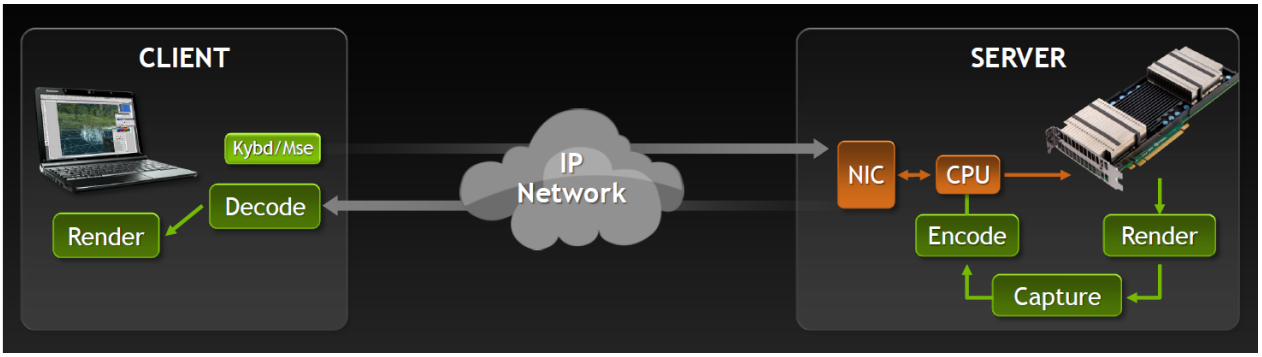


Abbildung 5: Programmarchitektur von Nvidia (siehe [3])

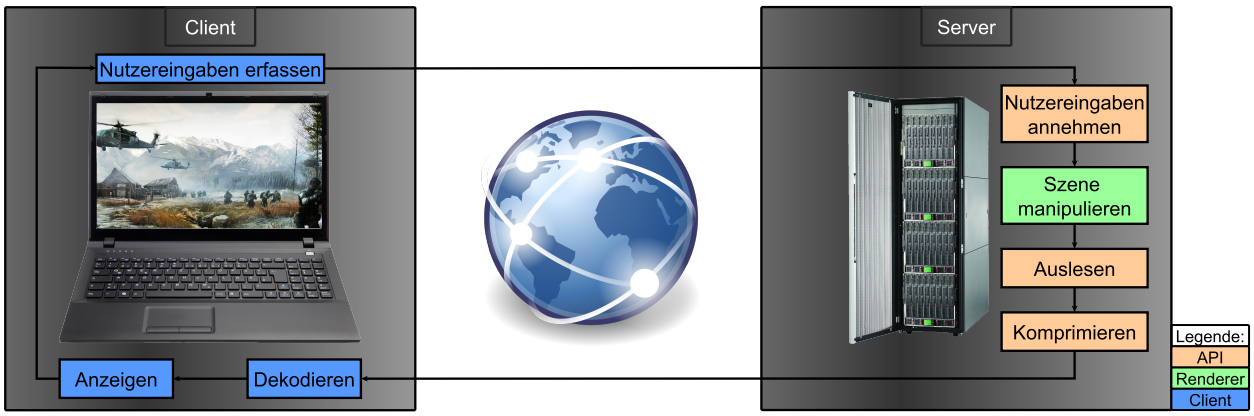


Abbildung 6: Geplante Client-Server-Architektur

Die gewählte Architektur sieht vor, dass die Szene durch einen austauschbaren Renderer auf dem Server berechnet wird. Danach folgt die Kompression und der Versand der Szenenbilder. Diese sollen dann von einem Clienten empfangen und angezeigt werden können. Da auf Seiten des Servers einige Funktionalitäten unabhängig von der Anwendung immer gleich ablaufen, bietet sich die Kapselung in einer API an. Folglich sind alle Remote Rendering Funktionalitäten für Anbieter grafischer Inhalte unter einer einfach zu bedienenden Schnittstelle zusammengefasst, die wenig Kenntnisse über interne Funktionalitäten erfordert. Somit soll eine Erweiterung eines jeden Grafikprogramms um Remote Rendering Funktionalitäten erfolgen können.
Der Client soll so schlank wie möglich gehalten werden (*Thin Client*). Das heißt, dass zwingend ein weiterer Rechner erforderlich ist, der den Clienten bei seinen Aufgaben unterstützt.

Clientseitig erfolgt deswegen lediglich die Dekompression und die Anzeige des Video-Streams, sowie die Übermittlung von Nutzereingaben. Diese Designentscheidung birgt den Vorteil, dass das Client-Programm wenig Rechenleistung benötigt, wodurch eine breite Verwendbarkeit garantiert wird. Weiterhin kann der Client, nur durch einen Serverwechsel, beliebige Szenen ohne weitere Konfiguration anzeigen.

3.1 Remote Rendering API

Die Remote Rendering API soll alle Funktionalitäten der Technologie kapseln und einen „einfachen“ Renderer zu einem netzwerkfähigen Remote Rendering Server erweitern. Folgende Anforderungen sind festzuhalten:

API-unabhängiges Auslesen der Szene
: Aktuell sind zwei große Grafik-APIs verbreitet: Zum einen Microsofts Direct3D, zum anderen OpenGL. Der Zugriff auf die berechneten Bilddaten unterscheidet sich je nach verwendeter Grafik-API erheblich. Die Remote Rendering API soll sowohl Direct3D-Szenen, als auch OpenGL-Szenen unterstützen und die Bilddaten entsprechend auslesen.

Kompression der Bildinhalte
: Zur Kompression des Bildes soll der hocheffiziente H.264/MPEG-4 AVC Standard verwendet werden. Dieser verspricht hohe Kompressionsraten bei guter Bildqualität. Da die Bildinformationen im späteren Programm via Internet an Clienten versendet werden sollen, besteht ein besonderes Interesse daran, die entstehende Datenmenge so klein wie möglich zu halten.

Konvertierung der Bildinhalte
: Zur Kompression müssen die Bilddaten zuvor in einen anderen Farbraum konvertiert werden. Dieser Schritt soll ebenfalls auf der Grafikkarte durchgeführt werden, da die Bilddaten nach der Synthese in ihrem Speicher liegen. Dieses Vorgehen verspricht einen großen Geschwindigkeitszuwachs.

Versenden der komprimierten Bildinformation
: Die Remote Rendering API soll bei ihrer Initialisierung automatisch Berkeley-Sockets anlegen, über die der entstehende Video-Stream an die Clienten versendet wird. Weiterhin wird ein Protokoll spezifiziert, das Art und Größe der Bilddaten und Nutzereingaben kennzeichnet.

Erfassung von Nutzereingaben
: Zusätzlich soll auch der Empfang von clientseitigen Nutzereingaben durch die RR-API abgewickelt werden. Im Detail können Entwickler mit Hilfe von Callback-Funktionen festlegen, wie die Szene bei welcher Eingabe manipuliert werden soll. Diese werden dann von der RR-API aufgerufen.

3.2 Renderer

Initial wird ein einfaches Grafikprogramm mit einer ersten Testszene entstehen. Dieses soll eine Bewegung innerhalb der Szene durch eine Veränderung der Kameraposition erlauben. Die Implementierung erfolgt in der Programmiersprache C++ und der Grafikschnittstelle OpenGL. Die Szene zeigt eine schwarze Kugel auf einem beliebig farbigen Hintergrund.
Nach der Fertigstellung der Remote Rendering API soll eine Grafikengine eingebunden werden, die von Sascha Kolodzey zur Verfügung gestellt worden ist. Die Engine ist in C++ implementiert und verwendet Microsofts Direct3D API zur Darstellung großer, komplexer Szenen. Durch diesen Schritt soll die Eigenschaft von Remote Rendering, komplexe Szenen auf hardwareschwachen Geräten darzustellen, noch einmal hervorgehoben werden. Abbildung 7 veranschaulicht, wie der Renderer in die API integriert wird.

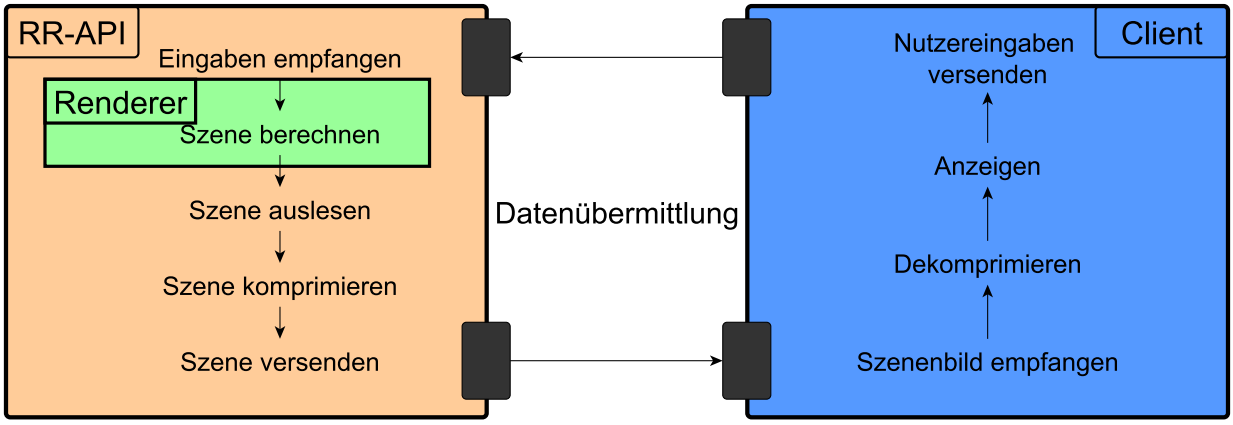


Abbildung 7: Einbettung eines Renderers in die RR-API

3.3 Client

Zusätzlich soll jeweils ein Client für das Betriebssystem Windows und ein Client für das Betriebssystem Android entwickelt werden, um Remote Rendering sowohl auf herkömmlichen PCs und Notebooks, als auch auf Smartphones und Tablets testen zu können. Die Spezifikationen beider Umsetzungen sind die Gleichen: Sowohl unter Android-Smartphones, als auch bei Notebooks gibt es eine Vielzahl möglicher Hardwarekonfigurationen. Daher soll die implementierte Fassung so ressourcenschonend wie möglich agieren, um eine hohe Abwärtskompatibilität zu gewährleisten. Die Folgenden Anforderungen lassen sich, unabhängig von der gewählten Fassung, festhalten:

Erfassung und Weitergabe von Nutzereingaben

Um die Idee der ausgelagerten Szenenberechnung umsetzen zu können, ist es von großer Bedeutung, dass der Nutzer mit dieser interagieren kann. Um dies zu verwirklichen müssen die Befehle, wie z.B. Tastatureingaben oder Touch-Gesten, vom Clienten erfasst und in geeigneter Form an den Server weitergegeben werden.

Dekompression des Video-Streams

Nachdem das Szenenbild vom Server komprimiert und versandt wurde, empfängt es der Client. Dieser soll das Bild dekomprimieren und zur Anzeige vorbereiten.

Anzeige des Szenenbildes

Auf Grund der unterschiedlichen Programmierschnittstellen von Android und Windows, kommen zwei verschiedene Verfahren zur Anzeige des Bildes zum Einsatz.

4 Verwendete Technologien

4.1 OpenGL

OpenGL ist eine plattformunabhängige Programmierschnittstelle zur Entwicklung von hardwarebeschleunigten Rastergrafikanwendungen. Ursprünglich von *Silicon Graphics* unter dem Namen *Iris GL* entwickelt, musste dieses auf Grund finanzieller Schwierigkeiten zu einem Open-Source-Projekt gewandelt werden. Heutzutage wird OpenGL von der Khronos Group betreut und liegt in der Version 4.4 vor.

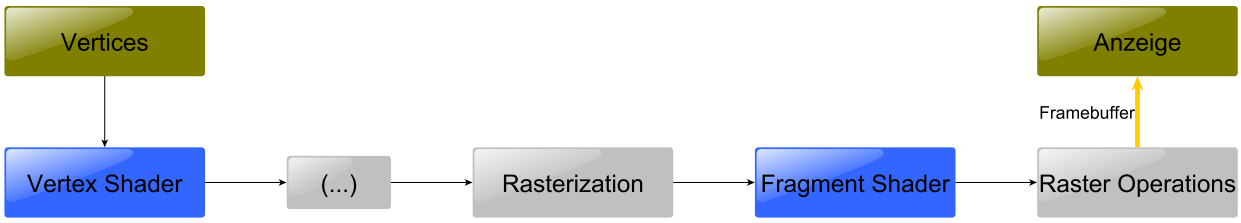


Abbildung 8: OpenGLs Grafikpipeline (Ab OpenGL Version 3.1, vereinfacht)

OpenGLs Schwerpunkt liegt darin die Grafikkarte zur Manipulation und Berechnung von Vektorgrafiken einzusetzen und diese dann, durch Rasterung, anzuzeigen. Die Grafik-Pipeline von OpenGL sieht sowohl programmierbare Abschnitte, die Shader, als auch fest installierte Bereiche, wie z.B. *Primitive Assembly* und *Rasterization*, vor. Die blau markierten Bereiche in Abbildung 8 zeigen die programmierbaren Abschnitte OpenGLs. Mit Hilfe des Vertex Shaders lässt sich zum Beispiel festlegen, wie Geometrien transformiert werden sollen.
Im Zuge dieser Arbeit ist der Framebuffer OpenGLs besonders interessant. In ihm werden die fertig gerenderten Bilder zwischengespeichert, bis sie an ein Anzeigegerät weitergeleitet werden. (s. Abb. 8).
Weiterhin unterstützt OpenGL die Verwendung von Texturen. Das sind Datenstrukturen, die ein- bis dreidimensionale Rastergrafiken beinhalten können und Funktionen für einen schnellen lesenden Zugriff bieten. Im Zuge dieser Arbeit werden sie zur Anzeige der, vom Clienten empfangenen, Bilder verwendet. Für mehr Informationen kann die OpenGL „Superbible“ [6] von Wright, Haemel, Sellers und Lipchak eingesehen werden.

4.2 Cuda

Bei Cuda (*Compute Unified Device Architecture*) handelt es sich um eine Technologie, die seit 2007 von Nvidia entwickelt wird. Während OpenGL die Grafikkarte zur Berechnung und Darstellung von 2D- und 3D-Vektorgrafiken nutzt, ermöglicht Cuda die Nutzung der Grafikkarte als Koprozessor. Das bedeutet, dass der Entwickler dedizierte Programmteile zur Berechnung durch die GPU abstellen kann. Auf Grund der Hardwarearchitektur einer Grafikkarte empfehlen sich hier vor allem Algorithmen und Programmteile, die hoch parallelisiert werden können. Je nach Parallelisierungsgrad ist eine signifikante Beschleunigung zu erwarten. Mögliche Anwendungsszenarien, die von einer Berechnung durch die GPU profitieren, sind zum Beispiel Partikelsimulationen, Physikberechnungen oder die Bildbearbeitung. Programmteile oder Algorithmen, die von der Grafikkarte ausgeführt werden sollen, sind als sogenannter Kernel (Dateiendung .cu) zu verfassen. Dieser stellt ein eigenes Programm dar und kann, neben dem eigentlichen Algorithmus, weitere Hilfsfunktionen enthalten. Cuda Kernel werden in *C for Cuda* verfasst. Es handelt sich dabei um die Sprache C, die um einige Funktionen von Nvidia erweitert wurde. Als Grundlage der folgenden Kapitel dienten die Bücher von Kirk und Hwu [1] und Sanders und Kandrot [2].

Die Cuda-API ist in C verfasst, daher ist Verwendung in Projekten der gleichen Sprache oder C++ unproblematisch. Es genügt die Einbindung zweier Bibliotheken: *cuda.lib* und *cuda-rt.lib*. Soll Cuda in Projekten anderer Programmiersprachen zum Einsatz kommen, müssen Wrapper verwendet werden. Aktuell existieren unter anderem Wrapper für die Sprachen Perl, Python, Java, Fortran und .NET. Allerdings sind ausschließlich Nvidia-Grafikkarten Cuda-kompatibel.

Generell spricht man im Kontext von GPGPU oft von Host und Device(s). Unter dem Terminus Host versteht man die CPU. Sie steuert den Programmfluss und weist einem oder mehreren Devices (in diesem Fall einer GPU) Aufgaben zu. Die GPU steht nach deren Bearbeitung für weitere Aufgaben zur Verfügung. Ein wichtiger Aspekt dieser Aufgabenverteilung ist der Flaschenhals, der durch die PCI-E-Schnittstelle entsteht. Abbildung 9 zeigt die Aufteilung zwischen Host und Device. Die CPU hält ihre Daten sowohl in extrem schnellen Caches, als auch im Hauptspeicher vor. Gerade bei großen Datenmengen, z.B. den Farbinformationen eines Bildes, reichen die CPU-internen Register und Caches jedoch nicht aus, sodass große Teile der Daten im Arbeitsspeicher vorgehalten werden müssen. Dieser besteht im Beispiel von Abbildung 9 aus DDR3-Riegeln, die mit 1600Mhz getaktet sind. Der daraus resultierende Speicherdurchsatz beträgt 25,6 Gigabyte pro Sekunde. Die GPU wiederum ist über eine PCI-Express Schnittstelle mit der CPU verbunden. Diese weist einen maximalen Datendurchsatz von 8 Gigabyte pro Sekunde auf und stellt damit im System Host-Device den besagten Flaschenhals dar. Daraus resultiert, dass häufige Kopiervorgänge großer Datensätze zwischen Host und Device unbedingt vermieden werden sollten. Es bietet sich an, die benötigten Daten zu Beginn von der CPU auf die GPU zu kopieren und erst nach Abschluss aller Berechnungen wieder herunter zu laden. Gerade im Austausch zwischen OpenGL und Cuda, die beide Speicher auf der Grafikkarte vorhalten können, ist es vorteilhaft Daten zwischen diesem zu kopieren. Da moderner Grafikkartenspeicher extrem hohe Datendurchsatzraten ab 160 Gigabyte pro Sekunde erreicht und Kopiervorgänge sehr schnell und günstig sind wird

so der langsame Umweg über die CPU vermieden. Neuere Grafikkarten erlauben sogar das direkte Ansprechen vom OpenGL-Speicher aus Cuda.

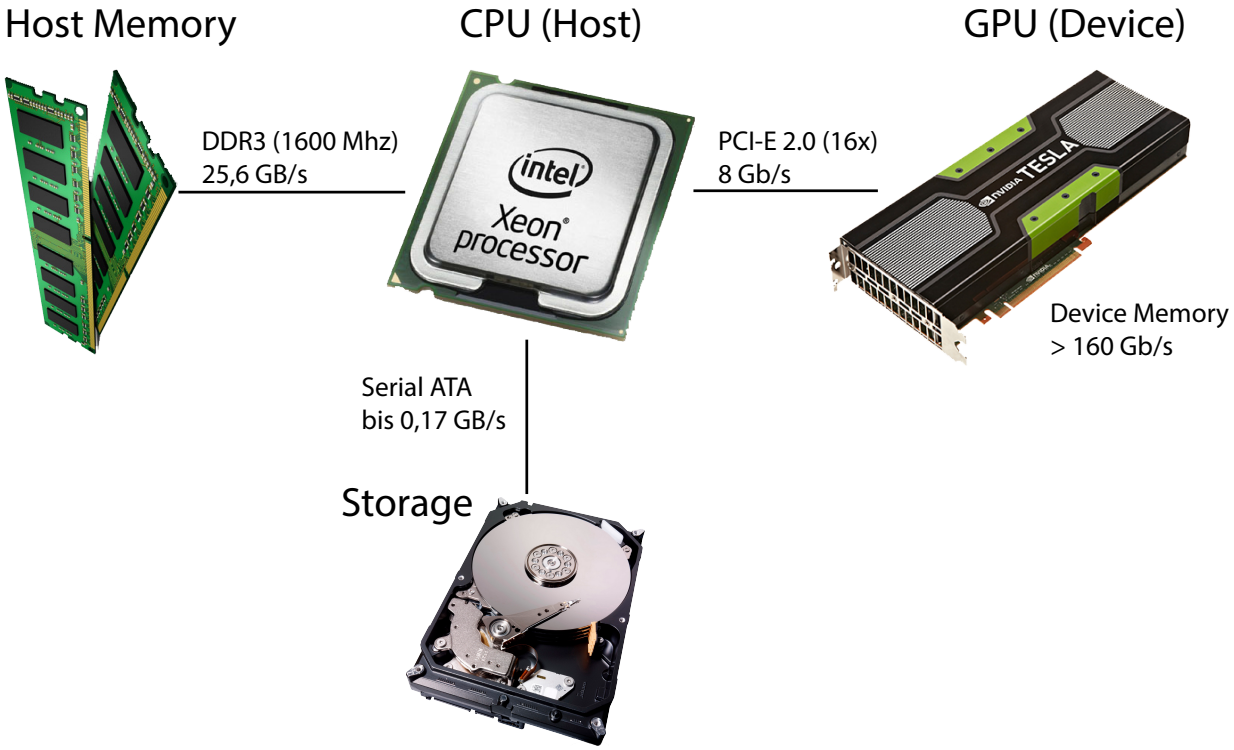


Abbildung 9: Datendurchsatz der verschiedenen Schnittstellen eines PC

4.2.1 Vektoraddition in Cuda

Das folgende Beispiel soll die Funktionsweise von Cuda erläutern. Das Programm hält zwei Integer Arrays vor. Diese stehen für zwei Vektoren der Dimension 1000, die mit Hilfe der Grafikkarte zeilenweise addiert werden sollen.

```
1  const int SIZE = 1000;
2  const int BLOCKSIZE = 100;
3
4  __global__ void vecAdd(int* a, int* b)
5  {
6      int ix = blockIdx.x * blockDim.x + threadIdx.x;
7      a[ix] += b[ix];
8  }
9
10 int main(void)
11 {
12     //Zwei Vektoren, die mit Daten gefüllt werden
13     int* a = new int[SIZE];
```

```
14      int* b = new int[SIZE];
15      for(int i = 0; i < SIZE; i++)
16      {
17          a[i] = i; b[i] = i;
18      }
19      //Speicher auf der GPU anlegen
20      int *deviceMemoryA, *deviceMemoryB;
21      cudaMalloc((void**) &deviceMemoryA, SIZE * sizeof(int));
22      cudaMalloc((void**) &deviceMemoryB, SIZE * sizeof(int));
23      //a und b auf GPU kopieren
24      cudaMemcpy(deviceMemoryA, a, SIZE * sizeof(int),
          ↪ cudaMemcpyHostToDevice);
25      cudaMemcpy(deviceMemoryB, b, SIZE * sizeof(int),
          ↪ cudaMemcpyHostToDevice);
26      //Kernel aufrufen
27      dim3 dimGrid(SIZE / BLOCKSIZE, 1); dim3
          ↪ dimBlock(BLOCKSIZE, 1);
28      vecAdd<<<dimGrid, dimBlock>>>(deviceMemoryA,
          ↪ deviceMemoryB);
29      //Daten zurück kopieren und ausgeben
30      cudaMemcpy(a, deviceMemoryA, SIZE * sizeof(int),
          ↪ cudaMemcpyDeviceToHost);
31      cudaFree(deviceMemoryA); cudaFree(deviceMemoryB);
32      //Gekürzt: Ausgabe oder Weiterverwendung
33  }
```

Listing 1: Vektoraddition mittels Cuda

Um diese Werte mit Cuda auf der Grafikkarte verarbeiten zu können, muss zuerst Speicher auf dieser allokiert werden. Dies geschieht mit Hilfe der Funktion *cudaMalloc()*. Durch einen Aufruf der Funktion *cudaMemcpy()* können die Arrays von der CPU in die frisch angelegten Speicherbereiche auf der GPU verschoben und mit Aufruf der Kernel-Funktion verarbeitet werden. Die Kernel-Funktion zeichnet sich durch den, dem Methodenkopf vorangestellten, Identifier „__global__“ aus. Über den Identifier „__device__“ ist es möglich zusätzliche Hilfsmethoden zur Ausführung auf der Grafikkarte zu implementieren, sie müssen jedoch aus der Kernelfunktion heraus aufgerufen werden. Die Kernel-Funktion entspricht damit in etwa einer herkömmlichen Main-Methode. Der Kernelaufruf (Listing 1, Zeile 23) birgt ebenfalls eine Besonderheit. Zwischen Methodennamen und Parameterliste befindet sich die Angabe, wie viele Threads zur Bearbeitung gestartet werden sollen.

Cuda verwaltet Threads in einem Grid. Dieses kann ein-, zwei- oder dreidimensional sein und enthält Blöcke der gleichen Dimension. In den Blöcken finden sich die Threads, die einzelne Aufgaben bearbeiten. In Abbildung 10 ist beispielsweise eine zweidimensionale Konfiguration dargestellt, wobei das Grid die Größe 4 x 4 - und jeder Block die Größe 2 x 2 hat. Folglich würden in dieser Konfiguration 64 Threads erzeugt werden. Innerhalb des Kernels kann jeder Thread seine globale Position im Grid mit Hilfe der Structs $blockIdx.\{x, y, z\}$,

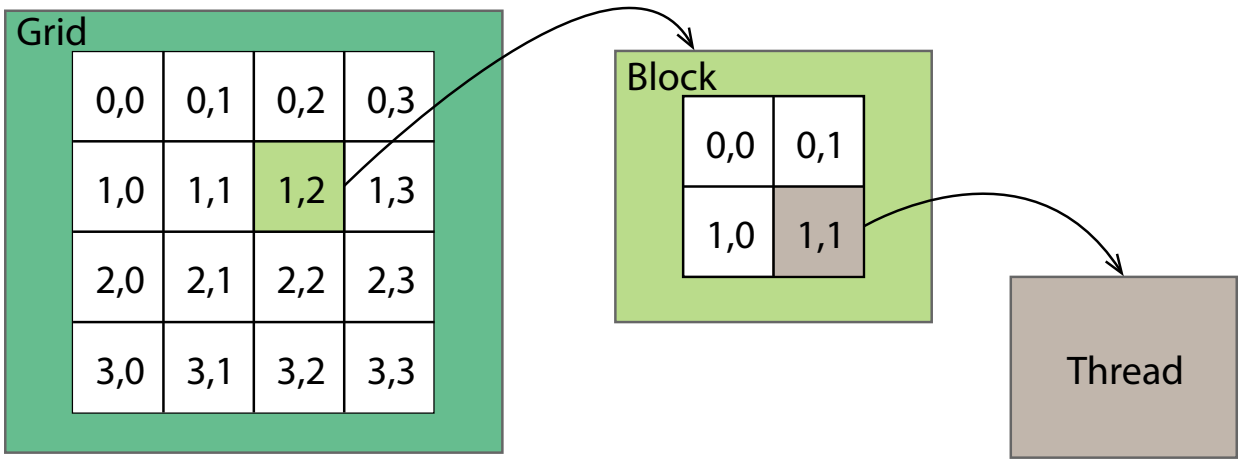


Abbildung 10: Threadverwaltung in Cuda

$blockDim.\{x, y, z\}$ und $threadIdx.\{x, y, z\}$ bestimmen. *BlockIdx* gibt die Position des Blocks im Grid an. Für Threads des grünen Blocks ist $blockIdx.x = 2$ und $blockIdx.y = 1$. Das Struct *threadIdx* verhält sich analog zu *blockIdx*. Es liefert Informationen über die Position des Threads im Block und *blockDim* die Größe des Blocks. Bezogen auf Abbildung 10, lässt sich die globale Position des hervorgehobenen Threads durch folgende Formel bestimmen:

$$globalX = blockIdx.x * blockDim.x + threadIdx.x = 2 * 2 + 1 = 5$$

$$globalY = blockIdx.y * blockDim.y + threadIdx.y = 1 * 2 + 1 = 3$$

Der Thread hat die globalen Koordinaten $[5, 3]$. Hinsichtlich des Anwendungsfalls Remote Rendering, könnte dieses Grid zur Bearbeitung eines Bildes der Auflösung 8 x 8 initialisiert worden sein. Jeder Thread entspräche dann einem Pixel des Bildes und könnte diesen manipulieren. Vor allem bei höheren Auflösungen wird deutlich, warum sich die Verwendung von Cuda bei der Bearbeitung von Bildern anbietet und deutliche Geschwindigkeitszuwächse gegenüber einer sequentiellen Verarbeitung durch die CPU verspricht.
In Listing 1 wird ein 10 x 1 großes Grid, das Blöcke der Größe 100 x 1 enthält, erzeugt. Dementsprechend entstehen 1000 Threads. Analog zum vorherigen Beispiel wird innerhalb des Kernels die globale Position eines jeden Threads ermittelt. Jeder Thread berechnet nun für „seine“ Zeile den neuen Wert des Vektors. Nach Abschluss der Berechnung, werden die Daten zurück auf die CPU kopiert und dort aus ausgegeben.

4.2.2 Interaktion mit OpenGL

Sowohl Cuda, als auch OpenGL stellen zwei eigenständige Schnittstellen dar, die die Grafikkarte zur Berechnung und Ausführung verschiedener Aufgaben einsetzen. Diese Aufgaben (OpenGL: Berechnung und Darstellung von 2D- und 3D-Computergrafikszenen; Cuda: Bildbearbeitung, Physiksimulation, Partikelsysteme, uvm.) gehen jedoch oftmals Hand in Hand. Ein gutes Beispiel stellt das, bei der Videobearbeitung und in vielen Computerspielen verwendete, Postprocessing dar. Postprocessing beschreibt die Weiterverarbeitung eines bereits

fertig gerenderten Bildes. Dadurch können dem Bild weitere Effekte, wie z.B. Bewegungsunschärfe (s. Abbildung 11) hinzugefügt werden.

(a) Mit Bewegungsunschärfe **(b)** Ohne Bewegungsunschärfe

Abbildung 11: Bewegungsunschärfe durch Postprocessing (siehe Straßenbelag)[8]

Um Postprocessing-Effekte zu realisieren, wird ein von OpenGL gerendertes Bild ausgelesen. Daraufhin wird es zur weiteren Bearbeitung an einen Cuda-Kernel übergeben und letztendlich wieder von OpenGL angezeigt. Es sind also mindestens zwei Kopierschritte zwischen beiden Schnittstellen notwendig. Da das Kopieren über die CPU sehr teuer ist (siehe Abb. 9, S. 11), sieht die Cuda-API eine Interaktion beider APIs direkt auf der Grafikkarte vor. So kann deren extrem schneller Speicher für das interne Übertragen der Daten verwendet werden. Die folgenden OpenGL-Ressourcen sind von Cuda ansprechbar:

Texturen
: Über den Befehl *cudaGraphicsGLRegisterImage()* kann eine OpenGL-Textur zur Bearbeitung registriert werden.

Buffer
: Durch die Methode *cudaGraphicsGLRegisterBuffer()* können OpenGL-Buffer, wie z.B. *GL_ARRAY_BUFFER*, für die Bearbeitung durch Cuda freigegeben werden.

Sind die benötigten OpenGL-Ressourcen registriert, können sie über entsprechende Funktionen gemappt werden. Das bedeutet, dass ein Zeiger auf das entsprechende Speicherareal der Ressource erstellt wird. Wird beispielsweise eine OpenGL Textur gemappt, wird ein Zeiger auf die Farbwerte der Textur zurückgeliefert.

4.3 Berkeley Sockets

Die Berkeley Socket API entstand 1983 als eine Netzwerkunterstützung für Unix-Systeme. Sie bietet die Möglichkeit UDP- und TCP-Verbindungen zu anderen Rechnern herzustellen und, obwohl ursprünglich für Unix-Systeme entwickelt, steht sie in ähnlicher Form unter Windows als Winsock-API zur Verfügung [7].
Im Fall von Remote Rendering liegt die Schwierigkeit in der Wahl zwischen TCP und UDP.

[8] http://farpeek.com/papers/MIG/mig2013.jpg (05.05.2014)

Daher sollen im Folgenden die Unterschiede, sowie einige Begrifflichkeiten erläutert werden: Zuverlässigkeit beschreibt die Fähigkeit Pakete in der korrekten Reihenfolge, ohne Datenverlust und doppelte Pakete übermitteln zu können. TCP und UDP fußen beide auf dem Internet Protokoll (IP). Dieses steuert den Versand von Datenpaketen zwischen zwei Rechnern, indem eine Adresse eingeführt wird, die einen Rechner eindeutig identifizierbar macht. UDP erweitert IP durch Ports, die aus dessen Rechner-zu-Rechner-Übertragung eine Prozess-zu-Prozess-Übertragung machen. UDP garantiert keine Sicherheit bezüglich Reihenfolge und Verlustfreiheit, es ist also nicht zuverlässig.
Die Leistung des TCP Protokolls besteht darin, die Paketorientierung des Internet Protokolls als einen bidirektionalen Datenstrom zwischen zwei Endpunkten zu kapseln. Dies wird durch zahlreiche Mechanismen erreicht, die die Zuverlässigkeit TCPs garantieren sollen. Unter anderem:

Sequenzierung
: Unter Sequenzierung versteht man eine fortlaufende Nummerierung der versandten Datenpakete. Mit ihrer Hilfe können die erhaltenen Pakete auf der Empfängerseite in der richtigen Reihenfolge und ohne Redundanzen zu einem Datenstrom zusammengesetzt werden.

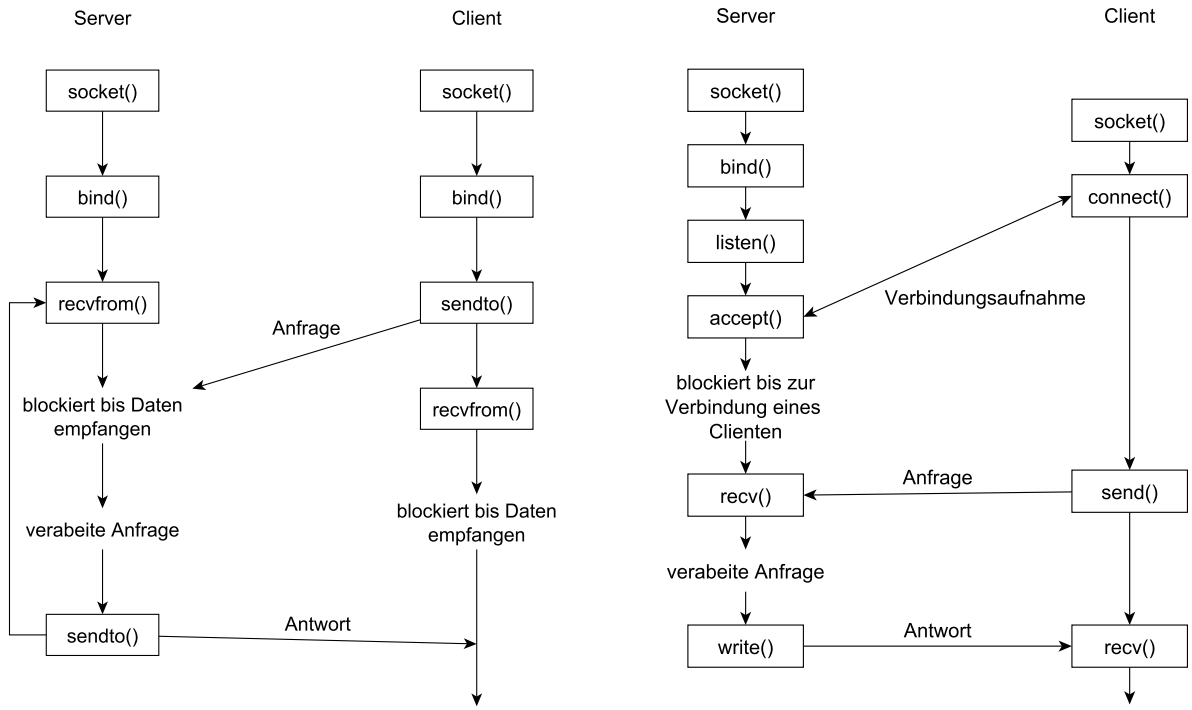
Neuübertragung
: Wird ein Paket empfangen, muss dem Versender eine Empfangsbestätigung übermittelt werden. Bleibt diese aus, wird das Paket nach einem festen Zeitintervall (*Timeout*) erneut übermittelt.

Flusskontrolle
: Unter Flusskontrolle versteht man die Fähigkeit zur Detektierung der aktuellen Netzwerklast. Sollte die TCP Implementierung feststellen, dass das Netzwerk zur Zeit stark ausgelastet ist, wird die Übertragungsrate rapide gesenkt, worauf eine Annäherung an die maximal mögliche Durchsatzrate durchgeführt wird.

Verbindungsorientierung
: Im Gegensatz zu UDP wird bei TCP eine feste Verbindung etabliert. Dieses wird durch den sogenannten Three-Way-Handshake erreicht. Möchte der Client A beispielsweise eine Verbindung zum Server B aufbauen, muss er ein initiales Verbindungsgesuch absetzen. B hat nun die Möglichkeit dieses zu bestätigen oder abzulehnen. Auf den Erhalt der Bestätigung von B sendet A eine erneute Bestätigung an B (*Forward Acknoledgement).*

Das Konzept von TCP gleicht damit einem Telefongespräch, während sich das Verhalten von UDP eher mit dem Versenden von Briefen beschreiben lässt. Zusammenfassend zeigt sich, dass TCP deutlich mehr Features als UDP bietet. Das hohe Maß an Sicherheit hat allerdings den Nachteil, dass Daten nicht immer direkt versendet werden können. Aus diesem Grund fußen fast alle Streaming-Protokolle auf UDP. Decoder sind in der Regel tolerant bezüglich fehlender Pakete, Anwender jedoch nicht gegenüber der durch TCP zusätzlich entstehenden Wartezeit. Im Zuge dieser Arbeit wurden für beide Protokolle Wrapper-Klassen geschrieben, die auf der Winsock-API basieren. Der Fokus liegt aus den genannten Punkten jedoch von vornherein auf der Verwendung von UDP.

Die folgenden Abbildungen 12a und 12b zeigen, wie der Verbindungsaufbau und die Kommunikation zwischen Client und Server mit Hilfe des Berkeley Sockets aufgebaut wird.



(a) UDP-Socket **(b)** TCP-Socket

Abbildung 12: Verbindungsaufbau und Kommunikation mit TCP und UDP Socket

Bei beiden Implementierungen muss vor dem Beginn der Kommunikation ein Socket-Handle erstellt und mit einer IP-Adresse, sowie mit einem Port initialisiert werden. Dieser Socket erlaubt es dann, über die Methoden *recvfrom()* und *sendto()* Daten zu versenden und zu empfangen. Die Kommunikation der TCP-Implementierung läuft ähnlich hab. Dem Datenaustausch ist jedoch der Verbindungsaufbau über die Methoden *listen()* (auf Verbindungsaufnahme warten), *connect()* (Verbindungsgesuch an einen Partner stellen) und *accept()* (Verbindungsgesuch annehmen) vorangestellt. Der eigentliche Datenaustausch läuft analog zu UDP über zwei Methoden: *recv()* und *send()*.

4.4 Android Developement Tools

Android ist ein Betriebssystem für Smartphones, Netbooks und Tablets. Es wird seit 2003 von der *Open Handset Alliance* entwickelt und wurde 2008 von Google als Version 1.0 freigegeben. Seitdem wird Android ständig um neue Softwarepakete und Features erweitert. Die *Android Developement Tools* (ADT) werden ebenfalls von Google entwickelt und stellen ein Plugin für die IDE Eclipse dar, das eine Entwicklung von Programmen für Android Endgeräte erlaubt. Das Paket enthält mit dem *Android Virtual-Device-Manager* einen Emulator,

der es erlaubt, Programme auf verschiedenen virtuellen Android Geräten zu testen. Ist ein eigenes Android Gerät vorhanden, wird eine Entwicklung auf diesem ebenfalls unterstützt. Dazu wird ein spezieller USB-Treiber mitgeliefert, mit dem ein Debugging des Programms auf dem jeweiligen Gerät möglich ist. Die Integration der ADT in Eclipse läuft über dessen Paketverwaltung.
Hauptsächlich wird die Programmiersprache Java zur Entwicklung für Android verwendet. Es besteht jedoch die Option in der Sprache C oder C++ zu programmieren. Dafür muss ein weiteres Toolkit eingebunden werden, das sogenannte Android NDK (Native Developement Kit).
Bevor für ein Android Endgerät entwickelt werden kann, sollten einige Überlegungen zu den benötigten Klassen und Softwarepaketen angestellt werden. Besteht das Ziel einer breiten Abwärtskompatibilität, muss auf einige Klassenbibliotheken verzichtet werden. Im Fall dieser Arbeit sollen z.B. die Klassen *MediaCodec* und *MediaFormat* zur Dekodierung eines Video-Streams verwendet werden. Diese wurden erst mit der Android-Version 4.1, alias Jelly Bean (API Level 16), eingeführt. Das heißt, dass die Applikation auf Geräten mit einer älteren Version nicht unterstützt wird und damit nicht ausführbar ist. Zu Beginn eines neuen Projektes kann ausgewählt werden, mit welchem API-Level das Projekt mindestens kompatibel sein soll. Nachträglich kann diese Entscheidung in der *AndroidManifest.xml*-Datei abgeändert werden (s. Abb.13).

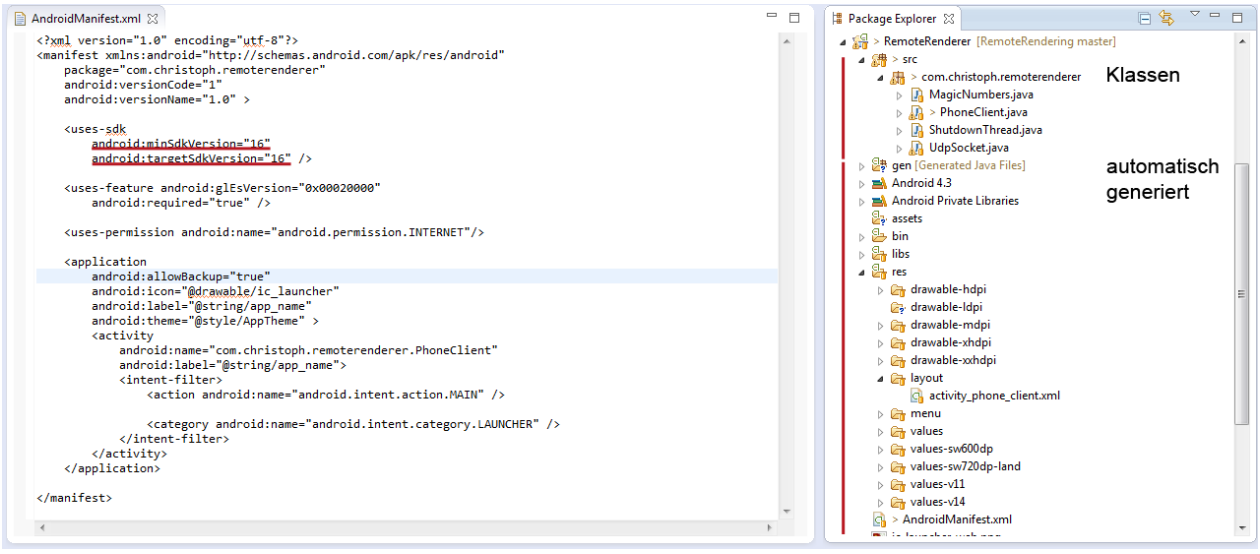


Abbildung 13: Manifest-Datei und Projektstruktur

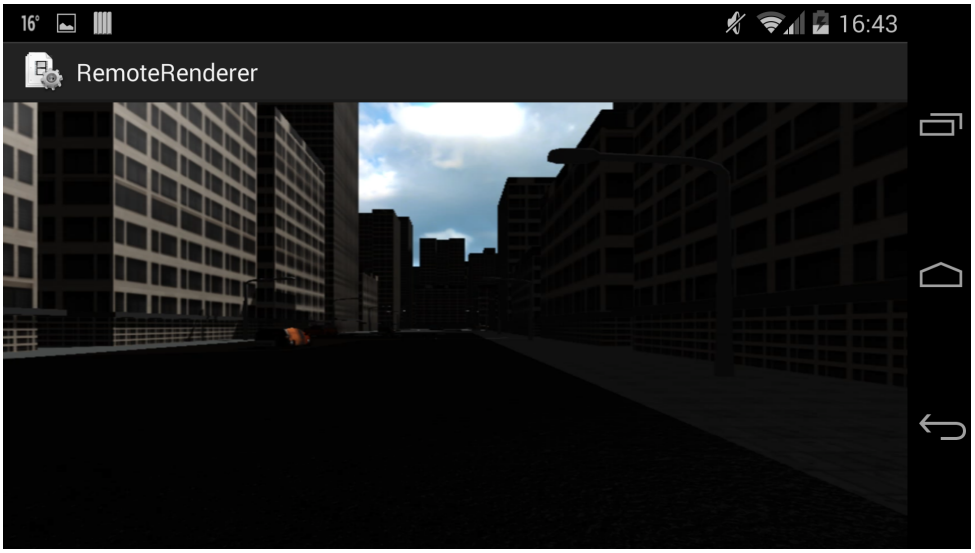
Nach der Erstellung des Projektes, liegt die komplette Ordnerstruktur der APP vor (ebenfalls zu sehen in Abbildung 13). Über diese ist es möglich dem Projekt eigene Dateien, Assets genannt, wie Bilder, Modelle, etc., hinzuzufügen.

4.4.1 Surfaces

Die Anzeige verschiedener Bildinhalte wird unter Android vom sogenannten Hardware Composer (HAL) gesteuert. Dieser wurde erstmals in der Android-Version 3.0 („Honeycomb“)

eingeführt. Die Implementierung des Hardware Composers liegt, abhängig vom Endgerät, in dessen Display-Hardware vor. Prinzipiell sind, je nach Anwendungen, verschiedene Elemente zur Anzeige vorgesehen. Unter anderem ist eine Anzeige der Applikation, sowie weiterer Elemente, wie der Statusbar und der Navigationsbar möglich. Diese Grafiken werden in verschiedenen Buffern vorgehalten. Die grundlegende Aufgabe des Hardware Composers besteht darin, viele Buffer mit unterschiedlichen Bildinhalten zu einem Gesamtbild zusammenzuführen. Im Fall von Statusbar, Navigationbar und Applikationsfenster wird der obere Teil des Bildes aus den Informationen der Statusbar, der rechte Teil aus denen der Navigationbar und der Rest des Bildes mit den Inhalten der Applikation gefüllt.

(a) Display-Elemente: Statusbar, Applikation, Navigationbar (v.l.n.r.)



(b) Zusammengesetztes Bild

Abbildung 14: Arbeit des Hardware Composers

Illustriert wird dieses Vorgehen in Abbildung 14. Die verschiedenen Buffer werden durch einen, SurfaceFlinger genannten Mechanismus bereitgestellt. Dieser erstellt beim Starten oder Maximieren einer App ein neues Surface, welches die Abstraktion der genannten Buffer darstellt und mit grafischen Inhalten gefüllt werden kann. Verschiedene Apps fungieren demnach

als Producer verschiedener Surfaces. Der SurfaceFlinger konsumiert diese, indem er sie für die Anzeige vorbereitet[9].
Die *MediaCodec*-Klasse bietet die Option, die dekomprimierten Bilder direkt in ein Surface zu speichern und so anzeigen zu lassen. Abbildung 15 demonstriert das grundlegende Vorgehen.

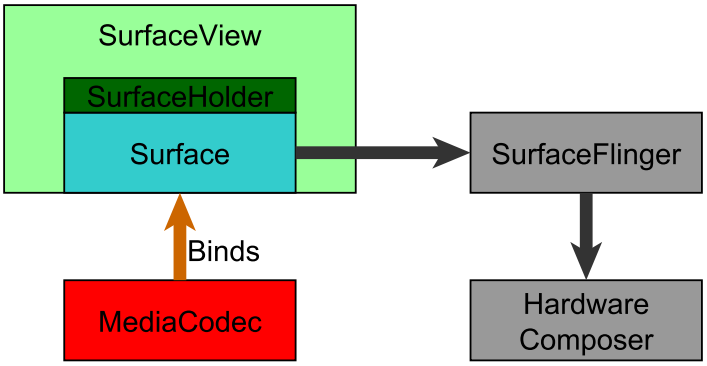


Abbildung 15: Interaktion zwischen MediaCodec, Surface und Framebuffer

Um ein Surface-Objekt zu erzeugen, empfiehlt sich das Anlegen einer *SurfaceView*. Sie erhält, über die Klasse SurfaceHolder, Zugriff auf ein Surface. Diese Variante birgt den Vorteil, dass das gehaltene Surface bereits für die Anzeige vorbereitet und korrekt auf die Displaygröße eingestellt ist.

5 Implementierung

Dieser Abschnitt soll sich mit der Implementierung des Remote Renderers befassen. Der Server, sowie der Client für das Betriebssystem Windows sind in C++ geschrieben. Die Wahl fiel auf diese Sprache, da viele Funktionen OpenGLs und CUDAs nativ in C vorliegen und somit ohne weiteres angesprochen werden können. Die Implementierung wurde mit Hilfe von Microsofts IDE *Visual Studio* vorgenommen.
Einzig der Client für Googles Betriebssystem Android ist in Java geschrieben, da so uneingeschränkter Zugriff auf die APIs von Android gewährleistet wird. Sprachen wie C und C++ werden zwar unterstützt[10], allerdings nicht in dem Funktionsumfang wie Java. Zur Implementierung des Android Clienten wurde die Open Source IDE *Eclipse* zusammen mit Googles ADT Plugin verwendet.

5.1 API

Die Kapselung der Remote Rendering Funktionalitäten in einer API ist essenziell. So wird sichergestellt, dass das fertige Programm kein statisches Konstrukt, bestehend aus einem Renderer und netzwerkfähiger Kompression, ist. Vielmehr ist es von Vorteil das Programm so

[9] http://source.android.com/devices/graphics/architecture.html (07.05.2014)
[10] Android NDK: http://developer.android.com/tools/sdk/ndk/index.html (24.04.2014)

zu modularisieren, dass auch andere Entwickler die hier erarbeitete Funktionalität einbinden und nutzen können. Dies geschieht mit Hilfe einer API. Sie ist, genau wie der Renderer, in C++ geschrieben. Der Renderer wird jedoch als ausführbare Datei (.EXE), die API dagegen als .DLL-Datei (*Dynamic Link Library*) kompiliert. Der Unterschied besteht darin, dass eine .DLL-Datei Funktionalitäten kapselt und diese nach außen über ein Interface bereitstellt, jedoch selber keine Hauptschleife enthält. Abbildung 16 soll die Struktur der RR-API, sowie deren Interaktion mit einem Renderer, verdeutlichen:

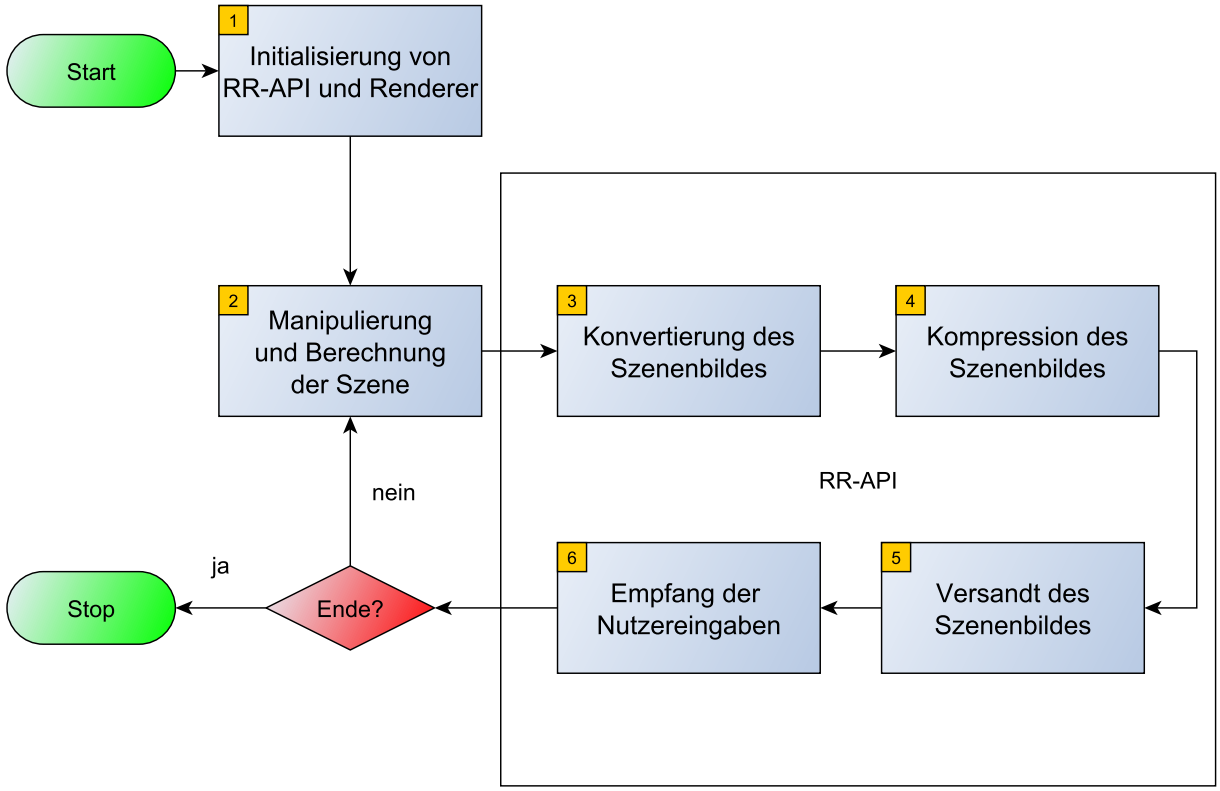


Abbildung 16: Programmablauf zwischen API und Renderer

Zuerst wird die Szene vom Grafikprogramm geladen. Weiterhin wird die RR-API mit den entsprechenden Parametern initialisiert. Lädt der Renderer beispielsweise eine Szene in der Auflösung 800 x 600, muss die RR-API zur Initialisierungszeit passend parametrisiert werden(**1**). Nachdem die Szene geladen wurde, muss diese konvertiert (**3**), komprimiert (**4**) und versandt (**5**) werden. Dann werden eventuell abgesetzte Nutzereingaben des Clienten abgefragt. Davon abhängend wird die Szene manipuliert (**2**) und die nächste Iteration beginnt. Das UML-Klassendiagramm (s. Abb. 17) zeigt den internen Aufbau der RR-API:

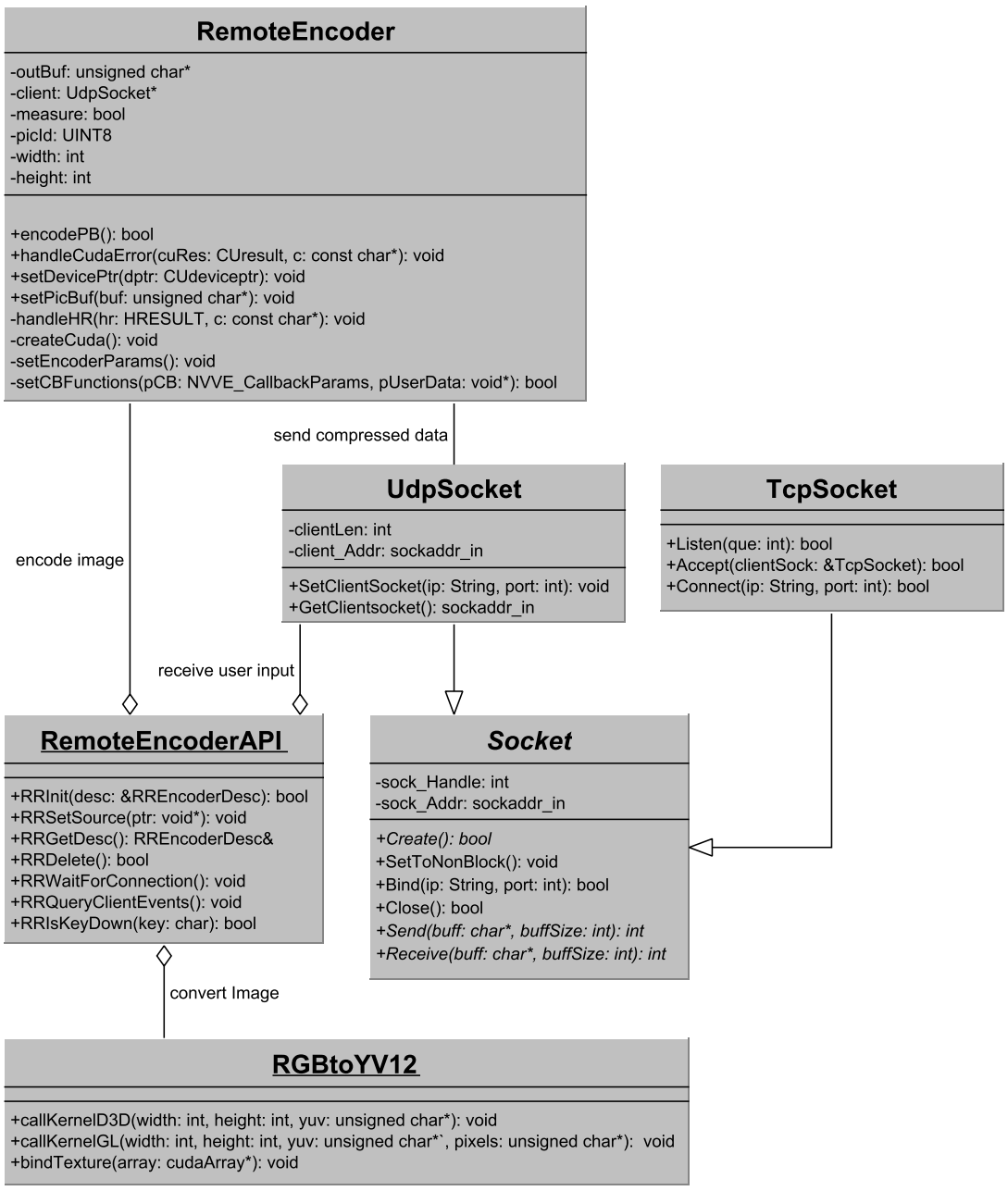


Abbildung 17: UML-Klassendiagramm der RR-API

Das Kernstück stellt das RemoteEncoderAPI-Modul dar, welches einen Encoder zur Kompression des Szenenbildes und einen Konverter einbindet. Die genaue Funktion dieser Klassen wird in den folgenden Abschnitten beleuchtet. Die Klassen UdpSocket und TcpSocket stellen Wrapper für das jeweilige Protokoll der Berkeley-Socket-Implementierung von Windows dar. Somit wird eine einfache Steuerung des Datenverkehrs ermöglicht.

5.1.1 Bildkonvertierung

Entsprechend Abbildung 16 besteht der erste Arbeitsschritt der RR-API darin, den Farbraum des Szenenbildes zu konvertieren. Der Farbraum, in dem die zwei großen Grafik-APIs OpenGL und Direct3D arbeiten, ist RGBA. In diesem additiven Farbmodell wird die Farbe eines Pixels durch das Verhältnis der drei Grundfarben Rot, Blau, Grün und deren Transparenz (Alpha-Kanal) bestimmt. In den meisten Fällen werden vier Bytes zur Kodierung der Farbe eines Pixels verwendet: Jeweils eines für den Rot- (**R**), Grün- (**G**), Blau- (**B**) und Alpha-Kanal (**A**), mit Werten im Intervall zwischen $[0, 255]$, oder in normalisierter Form 16 Bytes (4 Floats) mit Werten im Intervall von $[0, 1]$. Abbildung 18 zeigt, welche Farben durch Kombination der drei Farbkanäle entstehen können.

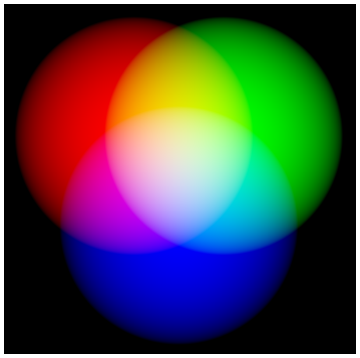
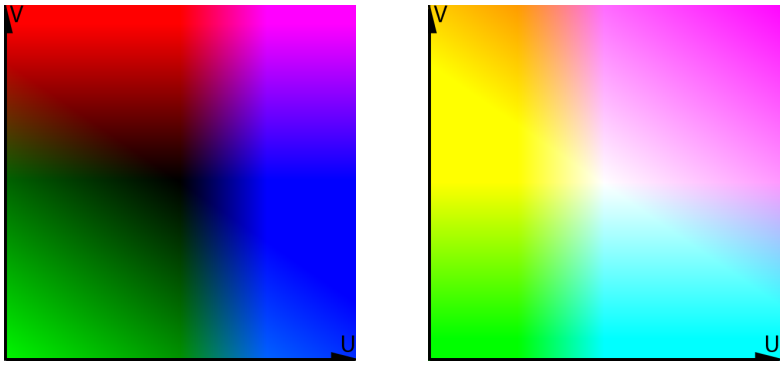


Abbildung 18: Mögliche Farbausprägungen im RGBA-Farbmodell

Die meisten Kompressionsbibliotheken erwarten jedoch Bildmaterial, das im YUV-Farbraum vorliegt. Dieses Farbmodell hat seinen Ursprung im Fernsehen. Bei der Umstellung von Schwarz-Weiß-Fernsehen auf Farb-Fernsehen wurde nach einem abwärtskompatiblem Farbformat gesucht, das sowohl den Betrieb von Schwarz-Weiß-, als auch von Farbfernsehern erlaubte. So entstand das YUV-Format, das im Gegensatz zum RGB-Format keine Farbkanäle für die Grundfarben, sondern einen Farbkanal für Helligkeiten (Luminanz, Y-Kanal) und zwei Kanäle für Farbwerte (Chrominanz, U- und V-Kanal), enthält. Mit diesem Format war es möglich, sowohl Schwarz-Weiß-Fernseher über den Luminanz-Kanal, als auch neuere Farbgeräte mit zusätzlichen Farbinformationen zu betreiben.



(a) Y = 0 **(b)** Y = 1

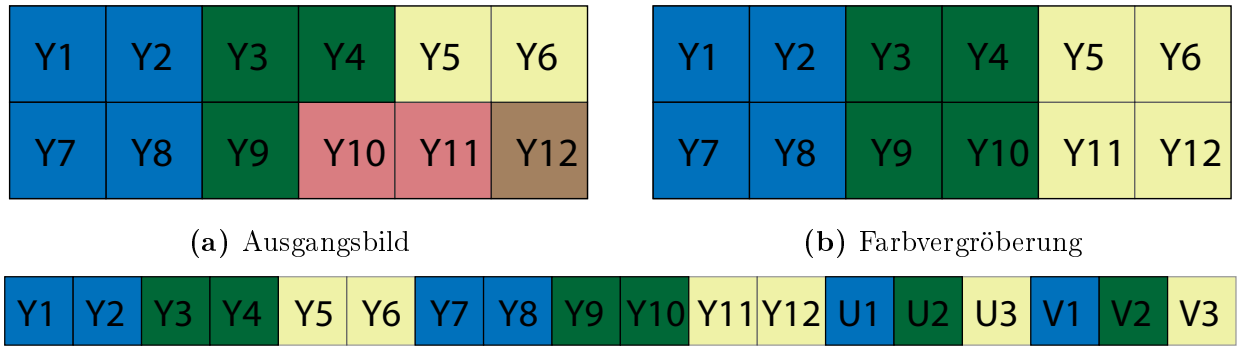
Abbildung 19: YUV-Ebene bei verschiedenen Helligkeiten

Abbildung 19 illustriert die Farbausprägungen des YUV-Farbmodells. Die Szenenbilder müssen in beide Richtungen konvertierbar sein: Auf der Serverseite muss das in RGBA vorliegende Bildmaterial zu YUV konvertiert werden. Auf der Clientseite muss das dekodierte Bild zurück in RGBA konvertiert werden, damit es angezeigt werden kann. Serverseitig werden die Farbwerte des Bildes mit der folgenden Matrix von RGBA nach YUV konvertiert:

RGBA zu YUV

$$Y = (66 \cdot R + 129 \cdot G + 25 \cdot B + 128) \div 256 + 16$$
$$U = (-38 \cdot R - 74 \cdot G + 112 \cdot B + 128) \div 256 + 128$$
$$V = (112 \cdot R - 94 \cdot G - 18 \cot B + 128) \div 256 + 128$$

Die Farbkonvertierung wird innerhalb der RR-API mit Hilfe von Nvidias CUDA durchgeführt. Wie in Abschnitt 4.2 bereits erwähnt, bietet CUDA die Möglichkeit Berechnungen massiv parallel auf der Grafikkarte auszuführen. Daraus resultiert, dass für jeden Pixel ein Thread auf der Grafikkarte gestartet wird, der die konvertierten Farbwerte berechnet. Bei einer Auflösung von 800 x 600 führt das zu 480.000 Threads, die auf der Grafikkarte gestartet werden. Auf Grund der parallelen Hardwarearchitektur von Grafikkarten kann die Konvertierung, verglichen mit einer zeitgemäßen, seriell arbeitenden CPU stark beschleunigt werden. Weiterhin birgt die Konvertierung mit Cuda den Vorteil, dass das gerenderte Bild nicht auf die CPU kopiert werden muss, sondern direkt als Grundlage der Konvertierung auf der GPU verweilen kann. Zusätzlich zur Farbraumkonvertierung wird innerhalb des Cuda-Kernels eine Vergröberung der Farbwerte ausgeführt: Vier Pixel werden zu einem Quadranten zusammengeführt. Jeder Pixel dieses Quadranten behält den Helligkeitswert. Die Farbwerte (U- und V-Kanal) werden von dem linken oberen Pixel des Quadranten bestimmt und die Farbwerte der drei restlichen Pixel verworfen.



(a) Ausgangsbild

(b) Farbvergröberung

(c) Speicher-Layout nach Vergröberung

Abbildung 20: Farbformat YV12

Abbildung 20 demonstriert, wie die Vergröberung der Farbinformationen abläuft: Gegeben sei das Bild 20a der Auflösung 6 x 2. Die Pixel Y3, Y4, Y9 und Y10 bilden einen Quadranten. Obgleich diese vier Pixel alle einen eigenen Farbwert besitzen, ist für die Konvertierung der Farbwert von Y3 entscheidend (man beachte den Pixel Y10). Die Farbinformationen der restlichen drei Pixel werden verworfen, jedoch nicht deren Helligkeitswerte (siehe Abbildung 20b). Da das menschliche Auge sehr viel empfindlicher auf Helligkeitsreize als auf Farbreize

reagiert, ist die Vergröberung eines YUV-Bildes zu einem YV12-Bild mit bloßem Auge kaum zu erkennen[11]. Das Layout der Farbwerte wird, wie in Abbildung 20c zu sehen, ebenfalls verändert. Anstatt pro Pixel einen Tupel aus Farbinformationen im Speicher vorzuhalten, werden zuerst die Helligkeitsinformationen abgespeichert, danach die, zu den Quadranten gehörenden, U-Werte, gefolgt von den V-Werten. Mit dem folgenden Cuda Kernel wird die Konvertierung auf der GPU durchgeführt.

```
__global__ void RGBtoYV12GL(unsigned char* yuv,
 ↪ unsigned char* pData)
{
   int ix = blockIdx.x * blockDim.x + threadIdx.x;
   int iy = blockIdx.y * blockDim.y + threadIdx.y;
   int gridWidth = blockDim.x * gridDim.x;
   int i = iy * gridWidth + ix;
   int globalBufferSize = gridDim.x * blockDim.x * gridDim.y
    ↪ * blockDim.y * 1.5;
   int rgbID = i * 4;
   int vpos = blockDim.x * gridDim.x * blockDim.y * gridDim.y;
   int upos = vpos + vpos / 4;
   int r = pData[rgbID], g = pData[rgbID+1],
         b = pData[rgbID+2];


   int y =  ((  66 * r + 129 * g +  25 * b + 128) >> 8) +  16;
   yuv[iy*gridWidth + ix] = y;

    if (!((i/gridWidth)%2) && !(i%2))
    {
       // U
      int u = ((-38 * r - 74 * g + 112 * b + 128) >> 8) + 128;
      yuv[upos + ix / 2 + iy / 2 * gridWidth / 2] = u;
        // V
      int v = ((112 * r - 94 * g - 18 * b + 128) >> 8) + 128;
      yuv[vpos + ix / 2 + iy / 2 * gridWidth / 2] = v;
    }
}
```

Listing 2: Kernel zur Konvertierung von RGB zu YV12

5.1.2 Bildkompression

Für die Kompression des Bildmaterials wird Nvidias *Nvcuvenc*-Bibliothek (Nvidia Cuda Video Encoder) verwendet [4]. Genau wie der „Konversion-Kernel“, verwendet diese zur Beschleunigung der Berechnungen ebenfalls Cuda. Die Wahl der Bibliothek wird dadurch motiviert, dass das konvertierte Bild weiter auf der GPU verarbeitet werden kann. Außerdem

[11]Siehe: http://forum.videohelp.com/images/guides/p1784363/yv12.png (19.04.2014)

eignet sich die Bildkompression aus den gleichen Gründen, wie die Bildkonvertierung, besonders zur Berechnung mit paralleler Hardware.
Zur besseren Nutzbarkeit der *Nvcuvenc*-Bibliothek wurde die Wrapper-Klasse *RemoteEncoder* geschrieben, die insbesondere die aufwändige Parametrisierung der Bibliothek mit den, auf das Anwendungsziel dieser Arbeit angepassten, Werten übernimmt (s. Abb. 17, S. 21). Der H264-Standard bietet zum Beispiel die Option eine Deltakompression hinzuzuschalten. Das bedeutet, dass das erste Bild als sogenanntes *Keyframe* mit vollen Bildinformationen komprimiert wird. Die darauf folgenden Bilder werden jedoch nur in der Differenz zu diesem *Keyframe* betrachtet und deren Abweichungen abgespeichert. Nach beliebig vielen Differenzbildern wird ein neues *Keyframe* generiert, um einen fehlerfreien Stand zu garantieren. Das Zuschalten der Deltakompression birgt den Vorteil, dass sehr viel Speicherplatz und damit, für den Versand benötigte, Bandbreite eingespart wird. Differenzbilder sind im Vergleich zu Keyframes, je nach vorherrschender Bildvarianz, bis zu zehn mal kleiner. Es muss jedoch untersucht werden, wie sich dieser Kompressionsmechanismus in einem Kontext vieler schneller, unvorhersehbarer Bewegungen verhält.

Die Klasse *RemoteEncoder* kapselt weiterhin die Funktionalitäten, die zur Bild-Enkodierung von der Bibliothek Nvidias bereitgestellt werden, sowie den Versand der fertig komprimierten Bilder. Durch Aufruf des Konstruktors werden sinnvolle Default-Werte gesetzt (s. [4], S. 25 ff.). Über die Methode *setDevicePtr(CUdeviceptr dptr)* kann das, vom Konversion-Kernel bearbeitete, Bild an den Encoder übergeben und durch die Methode *encodePB()* komprimiert werden. Während des Kompressionsprozesses werden durch Nvidias Kompressions-API vier Methoden aufgerufen, die vom Benutzer selbst zu implementieren sind. Diese Methoden heißen Callback-Funktionen. Sie sind bei der Initialisierung eines Encoders an die API zu übergeben. Innerhalb dieser Methoden kann der Entwickler Einfluss auf die Kompressionsarbeit nehmen. Die Funktionen, sowie deren Bedeutung werden im Folgenden erläutert. Eine vollständige Implementierung ist unter Appendix C.1 einzusehen:

OnBeginFrame und OnEndFrame
: Diese Methoden werden vor dem Start und nach der Beendigung der Kompression aufgerufen. Sie dienen hauptsächlich dem Debugging und müssen nicht implementiert werden.

AquireBitStream
: Innerhalb dieses Callbacks wird festgelegt, wie groß das zu kodierende Bild ist und durch *Call-by-Reference* an die Encoder-API weitergegeben. Diese Methode muss implementiert werden.

ReleaseBitStream
: Innerhalb dieser Methode erhält der Entwickler Zugriff auf das Ergebnis der Kompression. In dieser Arbeit wird diese Callback-Funktion zum Versand des fertigen Bildes verwendet.

5.1.3 Steuerung

Da der Nutzer im Fall von Remote Rendering nicht direkt den berechnenden Computer bedient und somit keine Befehle über klassische Eingabemedien, wie Tastatur und Maus absetzt, ist es notwendig für eine Übertragung der Eingaben vom Clienten zum Server zu sorgen. Dieses Feature wird ebenfalls von der Remote Rendering API abgedeckt, sodass bei ihrer Verwendung keine Notwendigkeit zu einer eigenen Implementierung der Steuerungsübertragung besteht. Um die „Steuerung aus der Ferne“ komfortabel zu realisieren, wurde ein einfaches Protokoll entwickelt, dass das Drücken und Loslassen von Tasten übermittelt. Drückt der Nutzer beispielsweise die Taste 'w', so wird in der zu übermittelnden Nachricht an erster Stelle ein 1 Byte großer Identifier abgelegt, der der RR-API mitteilt, dass eine Taste gedrückt wurde. Hinter diesem Identifier folgt, ebenfalls ein Byte groß, der ASCII-Code der gedrückten Taste. Zusätzlich gibt es einen weiteren Identifier für sogenannte *special Keys*, mit denen alle Tasten der Tastatur angesprochen sind, die nicht durch den 8-Bit ASCII-Code abgedeckt werden und einen Identifier für Mausbewegung. Nachdem die RR-API den clientseitigen Tastendruck registriert hat, wird eine entsprechende Boolesche Variable auf *true*- und erst bei entsprechender Übermittlung des Loslassens dieser Taste wieder auf *false* gesetzt. Dieses Vorgehen sorgt dafür, dass serverseitig eine gleichmäßige Umsetzung der Bewegung möglich ist, da in jeder Iteration die aktuellen Tastendrücke vorgehalten werden und deren entsprechende Bewegungen umgesetzt werden können. Die Wahrheitswerte der Tasten sind mit Hilfe von Arrays umgesetzt: Es existiert ein Boolesches Array der Größe 256 für ASCII-Tasten, sowie ein weiteres Array der Größe 246 für die restlichen möglichen Tasten der Tastatur. Der aktuelle Status der Taste 'w' aus dem obigen Beispiel würde somit an der Stelle *keyStates['w']* gefunden werden, wobei *keyStates* das Array zur Speicherung der ASCII-Tasten ist.
Die Weitergabe der Tastendrücke aus der RR-API an den Renderer geschieht über die C eigenen Funktionszeiger. Es handelt sich um Zeiger, die im Gegensatz zu Arrays nicht auf einen Datenspeicher, sondern auf ausführbaren Programmcode zeigen[12]. In diesem Anwendungsszenario bietet sich dieses Vorgehen an, da die Belegung und Bedeutung einzelner Tasten variiert und dem Entwickler somit freie Hand beim Implementieren des jeweiligen Verhaltens gelassen wird. Dabei sind die Methodenköpfe der, vom Entwickler zu implementierenden, Steuerungsmethoden die folgenden:

```
1 //Definition in der Header-Datei
2 typedef void (*KeyBoardHandler)(int key, bool pressed);
3 typedef void (*MouseHandler)(int dx, int dy, int button,
   ↪ int state);
4 //Anlage der Speicherfelder in der Cpp-Datei
5 KeyBoardHandler g_keyHandler;
6 MouseHandler g_mouseHandler;
```

Listing 3: Definition zweier Funktionszeiger zur Weitergabe clientseitiger Nutzereingaben

Die Funktion zur Verarbeitung der Tastaturdrücke muss nach Listing 3 also zwei Parameter erwarten: ein Integer, der angibt, welche Taste gerade gedrückt wurde und ein Boolean,

[12] http://openbook.galileocomputing.de/c_von_a_bis_z/012_c_zeiger_010.htm (24.04.2014)

mit dem zwischen Druck und Loslassen unterschieden wird. Unter Appendix C.2 wird eine beispielhafte Funktion gezeigt, wie sie vom hostenden Grafikprogramm implementiert werden könnte, um die Steuerung weiter zu verarbeiten.
Der Entwickler muss zur Initialisierungszeit seine Steuerungsfunktionen bei der RR-API anmelden. Diese hält dann die Zeiger auf die Funktionen vor (siehe Abbildung 3). Werden nun neue Benutzereingaben erfasst, so kann die, vom Entwickler geschriebene, Funktion aus der Remote Rendering API mit den entsprechenden Parametern aufgerufen werden. Listing 4 demonstriert den Aufruf der spezifischen Tastatursteuerungsfunktion über einen Funktionszeiger.

```
1    case KEY_PRESSED:
2       memcpy(&key, msg+sizeof(UINT8), sizeof(int));
3       if(key <= 256)
4       {
5          //Zeiger auf die Funktion. Sowohl die gedrückte Taste
6          // als auch der Status der Taste werden
            ↪ weitergereicht.
7          g_keyHandler(key, true);
8          //Intern wird der Status der Taste ebenfalls
            ↪ vorgehalten
9          g_keyStates[key] = true;
10      }
11      break;
```

Listing 4: Aufruf einer Funktion über einen Funktionszeiger

5.1.4 Entwicklerschnittstelle

Durch das Einbinden der Remote Rendering API werden folgende Funktionalitäten für Entwickler bereit gestellt:

```
1 CM_DLL_API bool CM_API RRInit(RREncoderDesc& desc);
2 CM_DLL_API void CM_API RRSetSource(void* ptr);
3 CM_DLL_API void CM_API RRWaitForConnection(void);
4 CM_DLL_API const RREncoderDesc& RRGetDesc(void);
5 CM_DLL_API void CM_API RREncode(void);
6 CM_DLL_API void CM_API RRQueryClientEvents(void);
7 CM_DLL_API bool CM_API RRIsKeyDown(char key);
8 CM_DLL_API bool CM_API RRDelete(void);
```

Listing 5: Interface der Remote Rendering API

RRInit

Initialisiert die RR-API. Erwartet ein Struct, das Informationen zur Bildgröße, der verwendeten Grafik-API (Direct3D oder OpenGL), der IP-Adresse des Servers, sowie Methoden, die bei einer Nutzereingabe aufgerufen werden sollen, vorhält.

RRSetSource

Legt die Bildquelle fest. Je nach verwendeter Grafik-API unterscheidet sich das Auslesen des fertig berechneten Bildes. Unter D3D11 wird ein *ID3DResource-Objekt*, unter OpenGL ein *Pixel Buffer Object* (PBO) erwartet. Dieses muss vom Entwickler einmalig selber angelegt und in jeder Iteration aktualisiert werden. Unter OpenGL sieht die Initialisierung und Aktualisierung eines PBO wie folgt aus:

```
//Ein PBO soll erzeugt und als int pbo gespeichert werden
glGenBuffers(1, &pbo);
//int pbo als Pixel Pack Buffer binden
glBindBuffer(GL_PIXEL_PACK_BUFFER, pbo);
//Festlegen der Größe des PBO
glBufferData(GL_PIXEL_PACK_BUFFER, width * height * 4,
  ↪ NULL, GL_STATIC_DRAW);

//In Hauptschleife: Auslesen des Bildinhaltes
//Anwählen des "Bildbuffers" von OpenGL
glReadBuffer(GL_BACK);
//int pbo als Pixelbuffer binden
glBindBuffer(GL_PIXEL_PACK_BUFFER, pbo);
//Pixelinformationen kopieren
glReadPixels(0, 0, width, height, GL_RGBA,
  ↪ GL_UNSIGNED_BYTE, 0);
```

Listing 6: Anlage und Aktualisierung eines Pixel Buffer Objects

RRWaitForConnection

Wartet darauf, dass eine Verbindung hergestellt wird.

RRGetDesc

Liefert die zuvor gesetzten Initialisierungsinformationen.

RREncode

Konvertiert und komprimiert den aktuellen Bildinhalt des Renderers. Durch das Zusammenspiel mit den Callback-Funktionen des Encoders, wird das komprimierte Bild im Anschluss automatisch versandt. Wichtig: Vorher muss sich ein Client verbunden haben, außerdem muss die API initialisiert und die entsprechende Bildquelle gesetzt worden sein.

RRQueryClientEvents

Überprüft, ob der Client neue Nutzereingaben übermittelt hat. Falls ja, werden diese an den Renderer weitergeleitet.

RRIsKeyDown

Liefert zurück, ob eine bestimmte Taste aktuell von Nutzer gedrückt ist.

RRDelete

Beendet die API und gibt alle Ressourcen frei.

5.2 Integration eines Renderers

Dieser Abschnitt befasst sich mit der Implementierung einer einfachen grafischen Szene. Dieser Schritt ist notwendig, um die Funktionalität der Remote Rendering API testen zu können. Nach Beendigung der Arbeit soll dieser Teil des Programms beliebig durch grafische Inhalte und Szenen anderer Entwickler austauschbar sein.
Die Testszene wurde, wie auch die RR-API mit C++ implementiert. Als grafische Schnittstelle dient OpenGL. Der Schwerpunkt lag auf einer schnellen Umsetzung und weniger auf besonderer grafischer Finesse. Ein Anspruch muss jedoch auch an die Testszene gestellt werden: Es muss die Möglichkeit zur Manipulation durch Nutzereingaben geben.
Die Szene zeigt eine schwarze Kugel auf rotem Hintergrund (siehe Abbildung 21). Durch Druck der Tasten 'W' und 'S' kann sich innerhalb der Szene auf der Z-Achse bewegt werden. Die Tasten 'A' und 'D' erlauben eine Navigation entlang der X-Achse und 'C' sowie die Leertaste entlang der Y-Achse.

(a) Komprimiertes Bild der Größe 2615 Byte **(b)** Komprimiertes Bild der Größe 8623 Byte

Abbildung 21: Einfache Testszene, mit OpenGL umgesetzt

Durch die Einfachheit der Testszene ergibt sich weiterhin der Vorteil, dass sich die Kompressionsarbeit des H.264-Encoders gut beobachten lässt. Der Vergleich aus Abbildung 21 zeigt, dass die Größe des komprimierten Bildes stark variieren kann. Dem liegt der höhere Anteil einer Farbe in Abbildung 21a zu Grunde, denn große Areale einer Farbinformation lassen sich stärker komprimieren als Areale mehrerer Farben. Dagegen sind die Häufigkeiten der Farben rot und schwarz in Abbildung 21b etwa gleich, woraus folgt, dass sich weniger Areale unter der Information 'rot' oder 'schwarz' zusammenfassen lassen. So wird das komprimierte Bild größer.
Nach der Fertigstellung der RR-API wird die Engine von Sascha Kolodzey, stellvertretend für einen beliebigen fremden Renderer, integriert. Da diese Microsofts Direct3D-API verwendet konnte neben der Integrierbarkeit eines fremden Renderers auch das Auslesen von Direct3D-Szenenbildern getestet werden. Das Resultat wird in Abbildung 22 demonstriert. Dieses Bild wurde von der Engine berechnet, der RR-API komprimiert und zum Versand vorbereitet und letztendlich durch den PC-Clienten angezeigt. Durch Hinzufügen der benötigten Steuer-

Callbacks (siehe Abschnitt 5.1.3) konnte die Steuerung ebenfalls mit geringem Aufwand auf eine Client-Server-Struktur umgestellt werden.

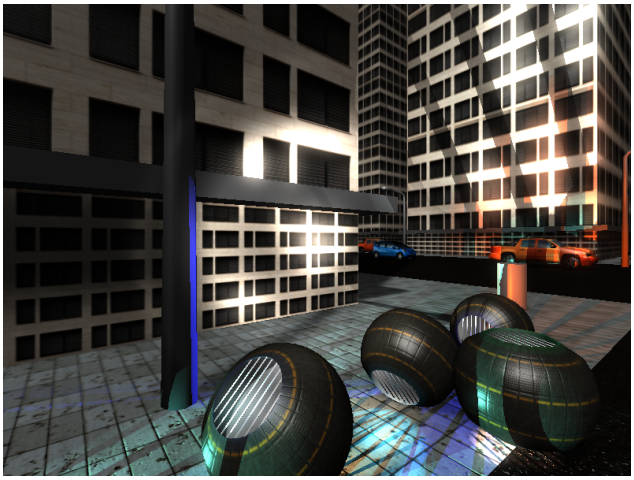


Abbildung 22: Übertragenes Bild der Engine von Sascha Kolodzey

5.2.1 Einbindung der Testszene

Abbildung 23 zeigt den stark vereinfachten Ablauf eines herkömmlichen Grafikprogramms.

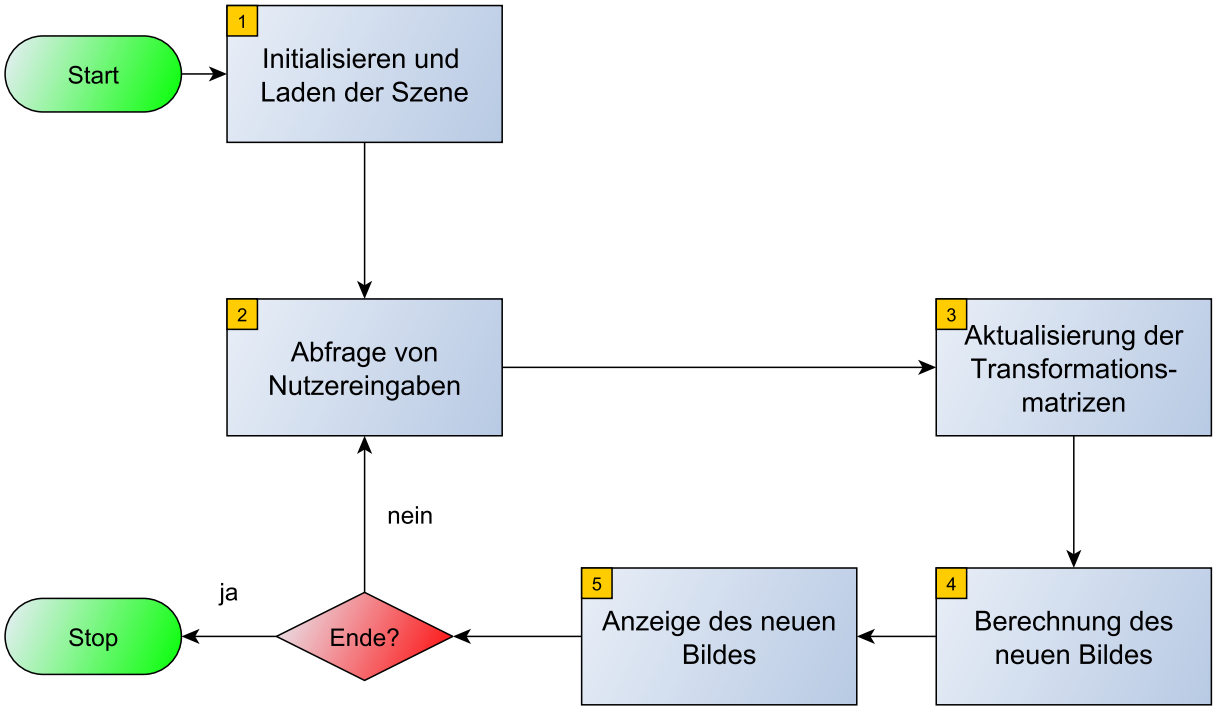


Abbildung 23: Schematischer Ablauf eines herkömmlichen Grafikprogramms

Zur Startzeit muss eine Szene geladen, sowie benötigte Datenfelder initialisiert werden (**1**). Danach wird überprüft, ob der Nutzer Befehle abgesetzt hat, die eine Manipulation der Sze-

ne zur Folge haben (**2**). Ist dies der Fall, werden die notwendigen Transformationsmatrizen verändert (**3**) und an den Vertex Shader der Grafikpipeline weitergereicht. Durch diesen wird dann ein neuer Szenenausschnitt berechnet. Es entsteht Bewegung. Weiterhin kann durch nachfolgende Shader, sowie Postprocessing das Bild weiter bearbeitet werden (**4**). Ist dieser Schritt abgeschlossen, wird das Bild zur Anzeige freigegeben.
Im Folgenden werden die nötigen Schritte zur Einbindung der RR-API, in Bezug auf Abbildung 23, aufgezeigt.

Initialisieren (1)

Die RR-API ist mit Hilfe eines *RREncoderDesc*-Structs zu initialisieren. Listing 7 zeigt exemplarisch eine mögliche Parametrisierung der RR-API.

```
1  RREncoderDesc rdesc; //Anlage des Structs
2  rdesc.gfxapi = GL; //Wahl der GrafikAPI: GL oder D3D
3  rdesc.w = 800; //Auflösung des Bildes (Breite)
4  rdesc.h = 600; //Auflösung des Bildes (Höhe)
5  rdesc.ip = 127.0.0.1); //Ip auf die der Server sich
     ↪ binden soll
6  rdesc.port = 1337; //Port des Servers
7  rdesc.keyHandler = RRMouseDummy; //Tastatur-Callback
8  rdesc.mouseHandler = RRMouseDummy; //Maus-Callback
9  //Aufruf der Methode zur Initialisierung
10 RRInit(rdesc);
```

Listing 7: Anlage eines Description-Structs zur Initialisierung der RR-API

Danach ist mittels der Methode *RRWaitForConnection* auf die Verbindung eines Clienten zu warten. Außerdem muss die verwendete Bildquelle über *RRSetSource* registriert werden. Unter OpenGL ist dies ein Pixelbuffer Objekt, das zuvor angelegt worden sein muss (siehe Listing 6, S. 28), unter Direct3D eine ID3D11Resource[13].

Abfrage von Nutzereingaben (2)

Im nächsten Schritt sind mögliche Eingaben des Clienten abzufragen. Dies geschieht durch die Verwendung der Methode *RRQueryClientEvents*. Sind die Callbacks zur Steuerung korrekt implementiert und registriert, kann die Eingabe so direkt weitergegeben werden.

Anzeige des neuen Bildes (5)

Nachdem das Bild fertig berechnet ist, muss es über das Internet an einen Clienten versendet werden. Dazu muss mit jedem Frame der Pixelbuffer über die Methode *glReadPixels()*, bzw. die ID3D11Resource über die *getBuffer()*-Methode der IDXGISwapChain, aktualisiert werden. Diese Ressourcen sind der RR-API bekannt. Daher kann nach Aktualisierung die Methode *RREncode* aufgerufen werden, die Kompression und Versand des Bildes startet.

[13]Siehe: http://msdn.microsoft.com/de-de/library/windows/desktop/ff476584%28v=vs.85%29.aspx (25.04.2014)

Beendung (6)

Zur Beendung der RR-API ist die Methode *RRDelete* aufzurufen, die für eine korrekte Freigabe der verwendeten Ressourcen sorgt.

5.3 PC-Client

Der PC-Client ist für einen Betrieb unter Microsofts Betriebssystem Windows konzipiert. Abbildung 24 zeigt den schematischen Programmablauf des Clienten.

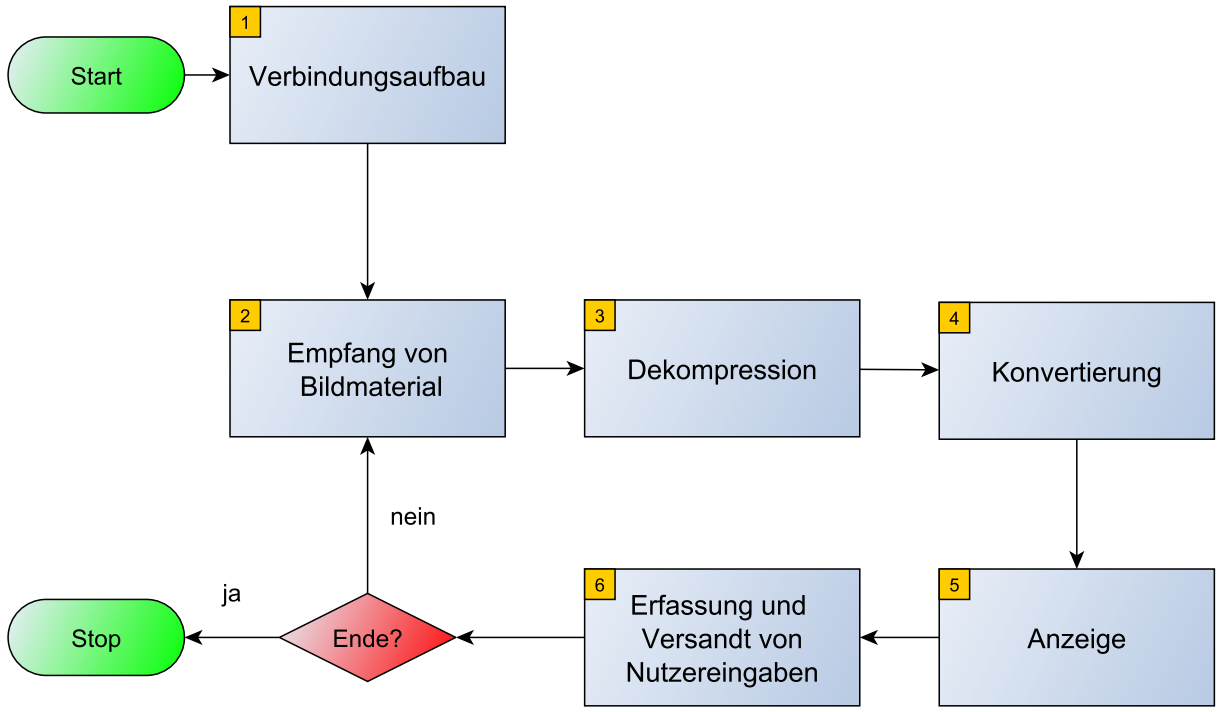


Abbildung 24: Programmablauf des PC-Clienten

Nach einer initialen Herstellung der Verbindung zum RR-Server (**1**), wird auf den Empfang seines Bildmaterials gewartet (**2**). Danach werden die inversen Arbeitsschritte des Servers ausgeführt, d.h. das Bild wird zuerst dekomprimiert (**3**), dann konvertiert (**4**) und dann mit Hilfe von OpenGL angezeigt (**5**). Am Ende einer Iteration werden mögliche Nutzereingaben erfasst und an den Server versandt. Einen Überblick über die interne Modul- und Klassenstruktur soll Abbildung 25 geben. Es werden wieder UDP- und TCP-Wrapper verwendet. Dazu kommen Klassen, die das Dekomprimieren und Konvertieren steuern. Die genaue Funktion dieser Klassen wird in den folgenden Abschnitten erläutert.

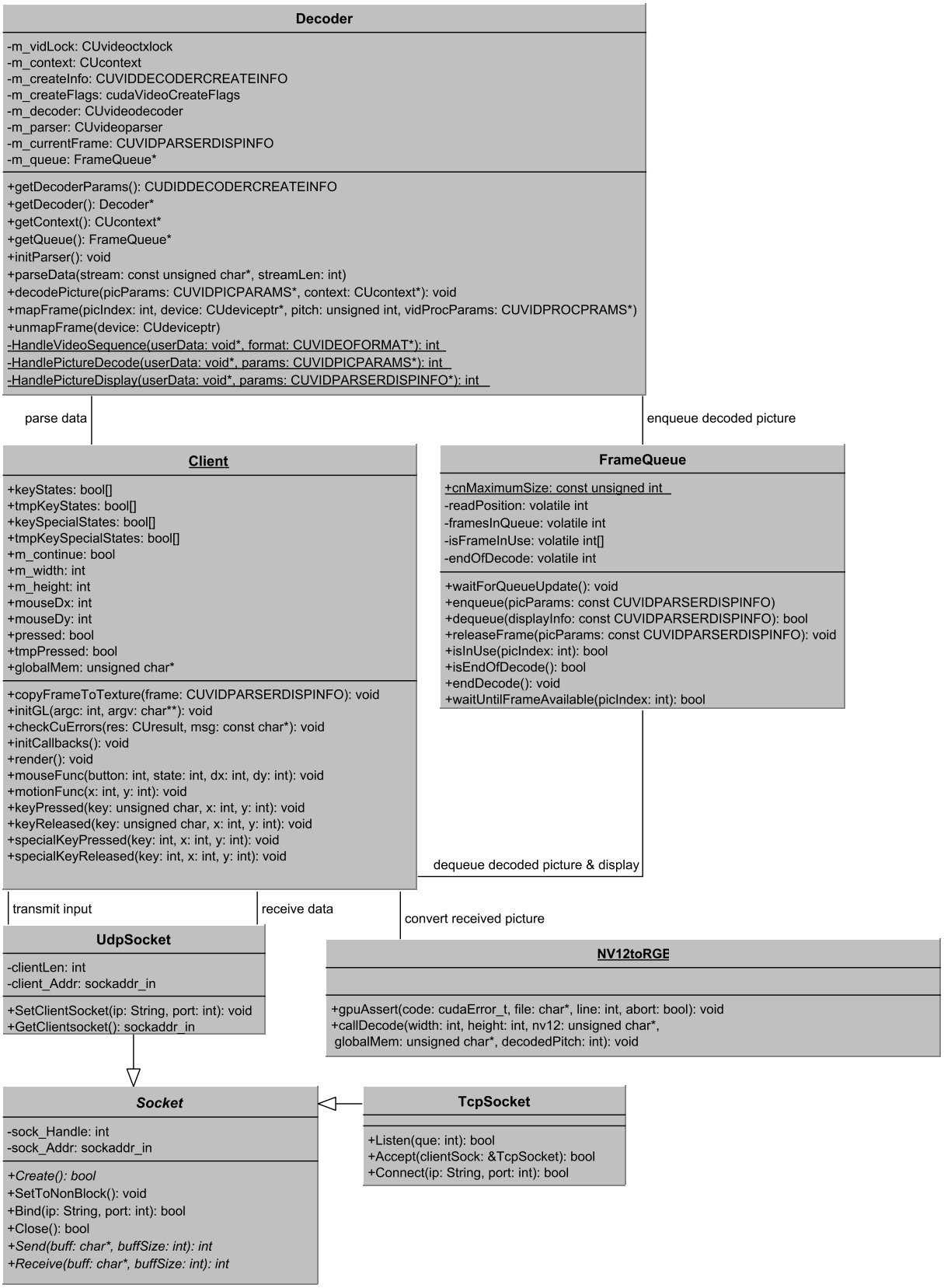


Abbildung 25: UML-Klassendiagramm des PC-Clienten

5.3.1 Nvcuvid

Zur Dekompression des Video-Streams wird Nvidias *Nvcuvid*-Bibliothek (Nvidia Cuda Video Decoder) verwendet [5]. Diese nutzt, gleich der serverseitigen Enkodierungs-Bibliothek, die hoch parallele Hardwarearchitektur der Grafikkarte zur Beschleunigung der Bildverarbeitung. Die Dekompression des Bildes wird in dieser Bibliothek durch das Zusammenspiel zweier Module geregelt: dem Decoder und dem Parser. Der Parser ist dabei als übergeordnetes Verwaltungskonstrukt zu verstehen, aus dem die Dekompression angestoßen und die Verwendung der fertig bearbeiteten Bilder geregelt wird. Beide Module wurden zur besseren Ansprechbarkeit in der Klasse Decoder.cpp zusammengefasst. Abbildung 25 zeigt die, von der Klasse angebotenen, Funktionalitäten.

Über die Methode *parseData* können die Bildinformationen, die vom Server komprimiert wurden, an den Parser übergeben werden. Dieser steuert die Verarbeitung der Bilder durch Callback-Funktionen, die vom Entwickler zu implementieren sind:

HandleSequenceChange
: Diese Methode wird vom Parser bei detektierten Wechseln im Bildformat aufgerufen.

DecodePicture
: Aus dieser Methode soll das Dekodieren des Bildes gesteuert werden.

DisplayPicture
: Nach Beendigung der Dekompression wird diese Methode aufgerufen. Das fertig dekomprimierte Bild kann dann weiter bearbeitet, oder zur Anzeige vorbereitet werden.

Bis auf den *HandleSequenceChange*-Callback werden alle Methoden bei jedem Aufruf von *parseData()* aufgerufen. Der *DisplayPicture*-Callback erlaubt eine Übergabe des dekodierten Bildes an den Anzeige-Thread. Da sowohl Anzeige als auch Dekompression in verschiedenen Threads laufen empfiehlt sich die Verwendung einer geeigneten Datenstruktur. In diesem Fall wurde eine Queue (*FrameQueue.cpp*, s. Abb. 25) verwendet, in die fertig dekomprimierte Bilder eingehängt werden können. Der Anzeige-Thread kann diese dann zur Weiterverarbeitung entnehmen. Sollte einer der beiden Threads (Producer oder Consumer) zu schnell für den anderen arbeiten, können diese über Mechanismen in der Queue kurzzeitig angehalten werden. Die Datenstruktur wurde aus Nvidias CudaSamples bezogen[14].

[14] http://www.ecse.rpi.edu/wrf/wiki/ParallelComputingSpring2014/cuda-samples/samples/3_Imaging/cudaDecodeGL/FrameQueue.cpp (29.04.2014)

5.3.2 Bildkonvertierung und Bildanzeige

Nachdem das Bild fertig dekomprimiert ist, liegt es wieder in einem Farbraum vor, der nicht von OpenGL anzeigbar ist. Diesmal handelt es sich um das sogenannte NV12-Format. Dieses fußt, genau so wie YV12 (siehe Abschnitt 5.1.1), auf dem YUV-Farbraum, besitzt jedoch ein leicht abgewandeltes Speicherlayout. Dieses ist in Abbildung 26 (Bezogen auf Abbildung 20b S. 23) dargestellt.

Abbildung 26: Speicherlayout von NV12

Das Farbformat NV12 ist YV12 relativ ähnlich. Im vorderen Teil des Speicherblocks findet sich wieder eine Liste aller Y-Werte der Pixel. Danach folgen im Wechsel die U- und V-Werte der Pixel. Zum Vergleich: Bei YV12 stand an erster Stelle eine Liste aller Y-Werte der Pixel, danach kamen alle U-Werte und zum Schluss alle V-Werte. Bei beiden Formaten teilt sich ein Zweier-Pixelquadrant ein UV-Wertepaar. Die Adressierung der Farbwerte im Cuda-Kernel ist jedoch unterschiedlich. Die Formeln, die zur Konvertierung von YUV nach RGB im Kernel verwendet wurden, sind die Folgenden:

$$R = (298 \cdot (Y - 16) + 409 \cdot (V - 128) + 128) \div 256$$
$$G = (298 \cdot (Y - 16) - 100 \cdot (U - 128) - 208 \cdot (V - 128) + 128) \div 256$$
$$B = (298 \cdot (Y - 16) + 516 \cdot (U - 128) + 128) \div 256$$

Die entstehenden Werte müssen sich in einem Intervall von $[0, 255]$ bewegen, d.h. größere, bzw. kleinere Werte sind an die Grenzen anzupassen.

Das fertig konvertierte Bild wird vom Cuda-Kernel in ein globales Speicherareal geschrieben. Dieses kann dann mit nur einem Kopierschritt in ein, bei OpenGL als Textur registriertes, Areal überführt werden. So werden unperformante Kopierschritte zwischen CPU und GPU vermieden.
Mit Hilfe der so gewonnenen Textur wird ein Viereck texturiert, das genau die Größe des Applikationsfensters hat. Das Viereck dient sozusagen als „Leinwand“ für die, vom Server berechnete, Szene. Abbildung 27 veranschaulicht dieses Vorgehen. Die Ausdehnung des Vierecks wurde verringert, sodass der blaue Hintergrund zu sehen ist. Auf dem Viereck selber läuft weiterhin die empfangene Szene.

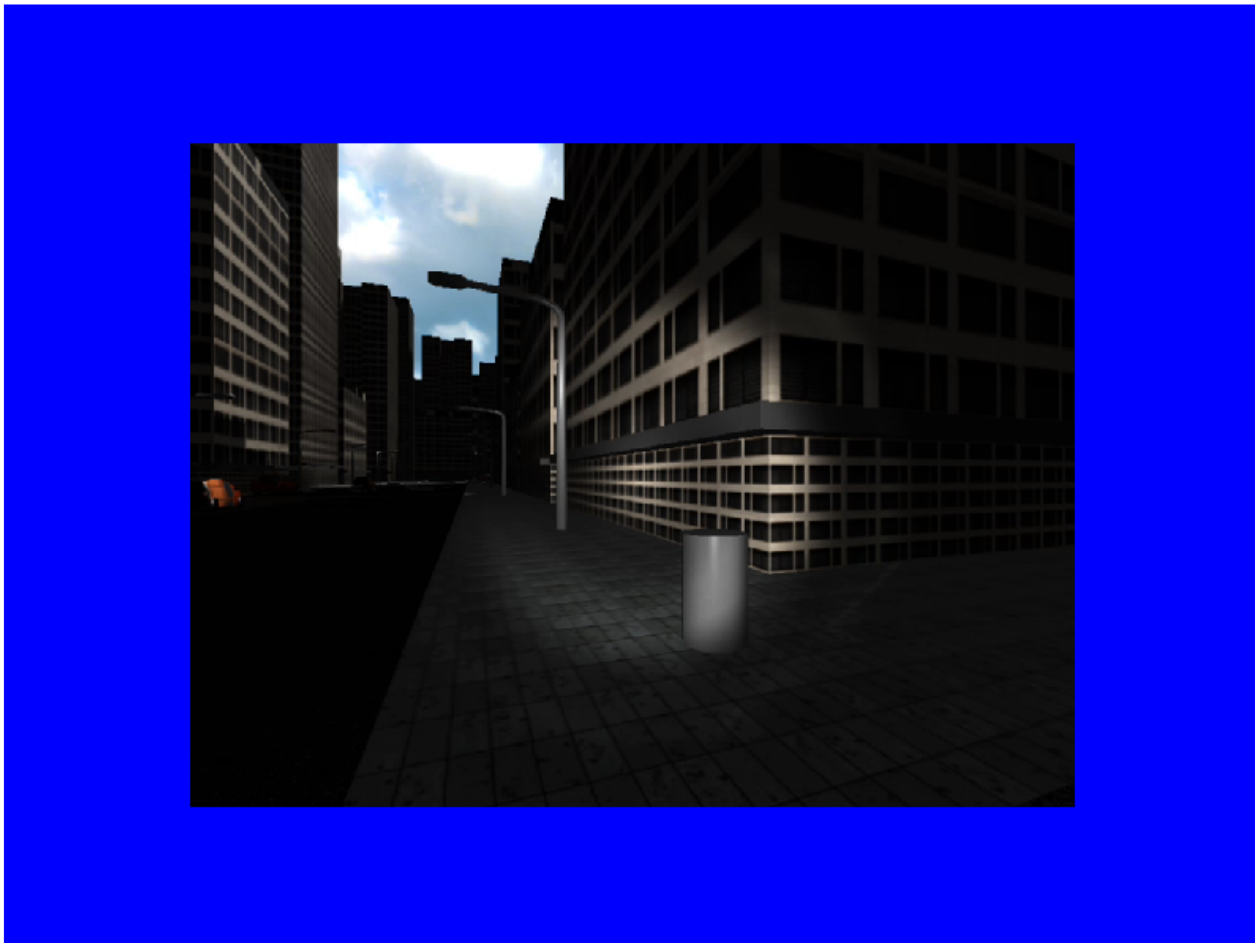


Abbildung 27: Texturierter „Screen Quad“

5.4 Android-Client

Um das Potential des Remote Renderings weiter zu unterstreichen, wurde ein zweiter Client für Smartphones und Tablets mit Android-Betriebssystem entwickelt. Das Resultat wird in Abbildung 29 gezeigt. Zur Ausführung des Clienten wird Android 4.1 benötigt (API Level 16). Diese Version ist auch unter dem Namen *Jelly Bean* bekannt. Die Applikation wurde auf dem Google Nexus 5 getestet, zu sehen in Abbildung 28.

Abbildung 28: Google Nexus 5

Der Programmablauf gleicht dem des PC-Clienten (siehe Abbildung 24 S. 32). Kernstück der Dekompressionsarbeit stellt diesmal die *MediaCodec*-Klasse von Google dar. Vor der Benutzung dieser Klasse muss eine Angabe über die Auflösung der Szenenbilder erfolgen. Weiterhin muss angegeben werden, von welchem Format der eingehende Video-Stream ist. Unter anderem werden die Codecs MPEG-4 und VP-9 unterstützt. In diesem Szenario ist der Codec H.264/AVC die richtige Wahl, da dieser bereits vom Server verwendet wird. Nach der initialen Konfiguration hat der Entwickler die Möglichkeit den

eingehenden Datenstrom in sogenannte Input-Buffer einzuhängen. Durch das Einhängen werden die Datenpakete zur Dekompression durch die *MediaCodec*-Klasse freigegeben und sind nach abgeschlossener Bearbeitung über eine Reihe an Output-Buffern zu erreichen. Alternativ kann der Entwickler festlegen, dass das Ergebnis der Dekompression direkt auf ein Surface gerendert wird (Siehe Abschnitt 4.4.1). Diese Technik wurde zur Implementierung des Android-Clienten verwendet.

Abbildung 29: Android-Client auf Googles Nexus 5. Renderer: Sascha Kolodzey + RR-API

Da es sich bei diesem zweiten Clienten um eine Machbarkeitsstudie handelt, wurde auf eine aufwändige Steuerung verzichtet. Aktuell ist eine Navigation innerhalb der Szene durch Vorwärtsbewegung und Rotation möglich. Die Steuerung kann in weiteren Arbeiten erweitert werden. Zum Beispiel wäre ein Einführen von Multitouch-Gesten denkbar. Dennoch reichen die aktuellen Steuermöglichkeiten aus, um auch unter Android das Funktionsprinzip und Potential von Remote Rendering zu zeigen. Die existierende Steuerung kann durch das Erweitern der Methode *onTouchEvent(MotionEvent e)* weiter angepasst werden. Verschiedene Touch-Gesten werden dabei auf ein Format gemappt, das der RR-API verständlich ist. Im aktuellen Entwicklungsstand wird die Berührung des linken Bildschirmrandes z.B. auf einen Druck der Taste 'q' gemappt. Dieses führt dann zu einer Rotation der Kamera nach links.

6 Resultate

In dem folgenden Abschnitt sollen einige Messresultate diskutiert werden. Zuerst werden die entstehenden Latenzzeiten innerhalb des LAN und WAN untersucht, danach die benötigte Bandbreite der Anwendung.

6.1 Latenz im LAN

Dieser Abschnitt behandelt die Latenz zwischen Clienten und Server innerhalb eines lokalen Netzwerks (LAN). Wie eingangs beschrieben, bestimmt die Latenz maßgeblich das Nutzerempfinden. Eine geringe Latenz ist immer zu bevorzugen.

Messmethodik und Vorüberlegung:

Aus der Methode *handleReleaseBitStream()* (s. Abschnitt 5.1.2) besteht Zugriff auf die komprimierten Bilder. Innerhalb dieser Methode ist jedoch nicht festzustellen um welches Bild es sich genau handelt. Im Bezug auf Latenzmessungen ist dieser Punkt problematisch, da keine Aussagen über die „Auslastung“ des Encoders gemacht werden können. Das bedeutet, dass nicht festgestellt werden kann, wie viele Bilder aktuell vom Encoder verarbeitet werden.

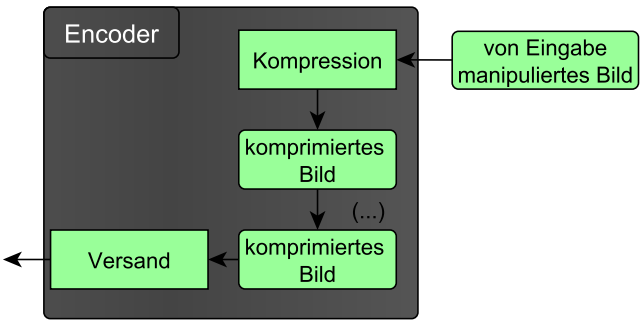


Abbildung 30: Bilder, die auf Versand warten

Abbildung 30 soll die Problematik veranschaulichen. Wenn vom Clienten ein Befehl abgesetzt wird, muss der eingebundene Renderer auf diesen reagieren und die Szene entsprechend verändern. Das neue, vom Renderer synthetisierte, Bild ist für die Latenzmessung von großem Interesse, denn es stellt das unmittelbare Feedback der Nutzereingabe dar. Nun wird dieses Bild an den Encoder übergeben, um die Kompression anzustoßen. Zu dem Zeitpunkt der Übergabe lässt sich allerdings keine Aussage darüber treffen, wie viele Bilder sich aktuell in der Kompression befinden oder auf den Versand warten. Die Identifikation des „Feedback-Bildes“ ist nach der Übergabe an den Encoder unmöglich. Es wird lediglich sichergestellt, dass die Bilder in der korrekten Reihenfolge ausgegeben werden.
Um dennoch eindeutige Ergebnisse zu erzielen, wird eine weitere Version der Programme PC-Client, RR-API und Android-Client erstellt. Diese zeichnen sich durch einen Mechanismus aus, der die „Feedback-Bilder“ mit einem „Wasserzeichen“ versieht. Das heißt, dass die ersten 100 Byte des Bildes mit 0 überschrieben werden. Dieses Bild wird dann konvertiert, komprimiert und an den Clienten zurück gesendet. Registriert der Client das „Wasserzeichen“, wird

nach der Anzeige die Differenzzeit zum vorausgegangenen Tastendruck berechnet. In Bezug auf die eingangs eingeführten Messungen von Nvidia (s. Abb 3, S. 3) werden so sämtliche Posten von der Game-Pipeline bis zur Anzeige gemessen. Abbildung 31 soll den Mechanismus zur Messung der Latenz verdeutlichen.

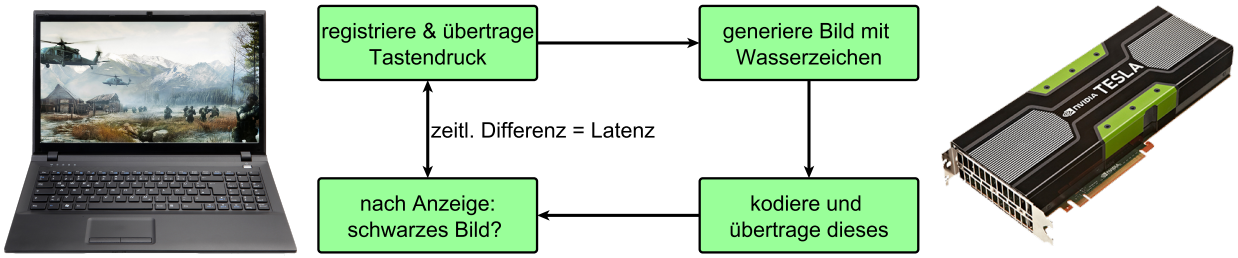
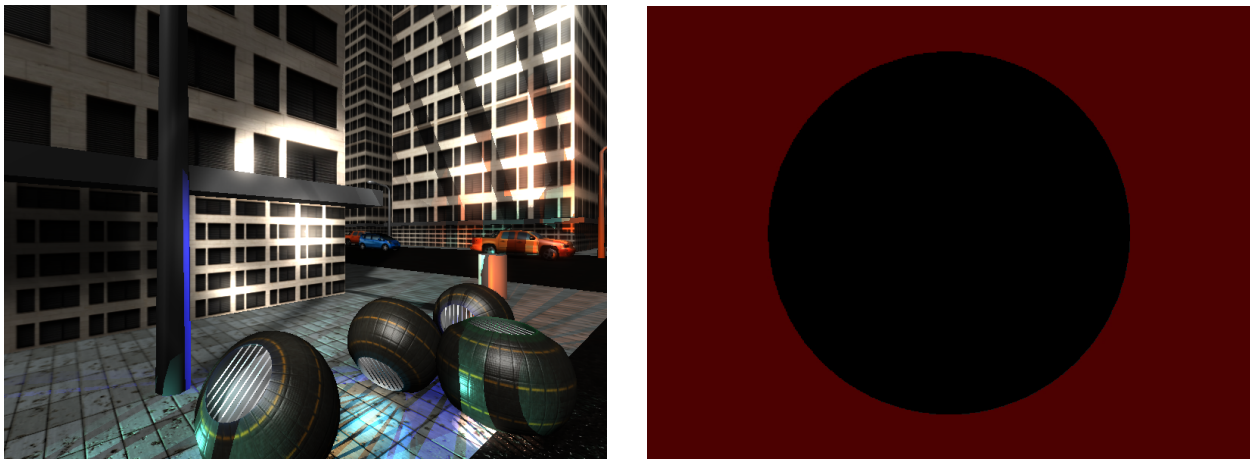


Abbildung 31: Schematische Darstellung der Latenzmessung

Zuerst wird innerhalb eines LAN gemessen. Das heißt die hier präsentierten Messergebnisse orientieren an In-Home Streaming-Diensten, wie unter anderem in Abschnitt 2.3 vorgestellt. Der folgende Abschnitt 6.2 befasst sich dann mit der zusätzlichen Latenz, die durch das Internet entsteht. Im Folgenden wird das UDP-Protokoll zur Übermittlung der Daten verwendet. Es wird zwischen PC-Client und RR-Server und zwischen Android-Clienten und RR-Server gemessen. Weiterhin wird innerhalb jedes Aufbaus mit zwei verschiedenen Renderern gemessen. Zum einen mit dem Renderer von Sascha Kolodzey (s. Abb. 32a), als Stellvertreter einer rechenintensiven Szene, zum anderen mit einem Renderer der weniger Rechenleistung benötigt (s. Abb. 32b). Beide Renderer stellen die Szenen in einer Auflösung von 800 x 600 dar.



(a) Renderer A **(b)** Renderer B

Abbildung 32: Angeführte Renderer zur Bestimmung der Latenz

Die folgende Tabelle zeigt die Eingabelatenz abhängig von dem Clienten und dem verwendeten Renderer.

Tabelle 1: Latenz zwischen Server und PC-Client

	Renderer A	Renderer B
PC-Client	$140ms$	$140ms$
Android-Client	$161,8ms \pm 46,4ms$	$160ms \pm 28,9ms$

Bei dem Messen mit dem Android-Clienten fielen immer wiederkehrende Latenzspitzen, also zwischenzeitlich große Wertausprägungen auf. Würde man diese aus der Messreihe herausnehmen, wäre die Latenz nahezu identisch mit der, des Systems PC-Client - RR-API. Unter Einbezug der Latenzspitzen, steigt die durchschnittliche Latenz jedoch um ca. 20 Millisekunden. Weiterhin resultiert aus den Schwankungen die hohe Abweichung. Die genauen Hardwarespezifikationen der Messgeräte werden in der folgenden Tabelle 2 aufgelistet:

Tabelle 2: Hardwarespezifikationen der Messgeräte (LAN)

Modell Funktion	Desktop-PC Server	Lenovo Thinkpad Edge 540 PC-/Windows-Client	Google Nexus 5 Android-Client
CPU	Intel Core i7-2700K	Intel Core i5-4200M	Qualcomm Krait 400
GPU	Nvidia Geforce GTX 580	Nvidia Geforce GT 740M	Qualcomm Adreno 330
RAM	8192MB	8192MB	2048MB

6.2 Latenz im WAN

In diesem Abschnitt wird die Latenzmessung um die Komponente Internet erweitert. Somit ist mit einer höheren Latenz zu rechnen, da die Pakete eine größere Wegstrecke durchlaufen müssen. Die Gesamtverzögerung eines Paketes, das über das Internet versendet wird, setzt sich aus den aufsummierten Latenzen der einzelnen Router des Netzwerkes zusammen. Jeder Router sorgt durch spezifische Arbeitsschritte für eine Verzögerung des Pakets. Man unterscheidet zwischen den folgenden Verzögerungsquellen [9]:

processing delay
: Die Zeit, die ein Router zur Verarbeitung eines Pakets benötigt.

queueing delay
: In der Regel wird ein Paket pro Zeiteinheit von einem Router verarbeitet. Sollten mehrere Pakete in kurzer Abfolge am Router eintreffen, müssen sie auf Verarbeitung warten.

transmission delay
: Diese Verzögerung wird von der Datenrate der jeweiligen Internetleitung verursacht. So dauert es z.B. doppelt so lange ein Paket an eine 8MBit/s-Leitung zu übergeben wie an eine 16MBit/s-Leitung.

propagation delay

Hierunter wird die Zeit verstanden, die das Paket benötigt, um vom Versender zum Empfänger zu gelangen. Die Verzögerung ist durch den Term $d_{propagation} = d \div s$ gegeben, wobei d die Länge des Kabels und s die Geschwindigkeit innerhalb des Mediums ist.

Es ergibt sich pro Router eine Latenz von:

$$d_{Router} = d_{processing} + d_{queue} + d_{transmission} + d_{propagation}$$

Daraus folgt, dass die Latenz in viel stärkerem Maße von der Menge an dazwischengeschalteten Routern abhängig ist, als von der physikalischen Entfernung. Jeder weitere Router trägt mit einem neuen Summanden zur Gesamtlatenz bei.
Die Methodik der Latenzmessung ist identisch mit der aus Abschnitt 6.1. Es wird jedoch lediglich mit Renderer A (s. Abb 32a) gemessen. Tabelle 3 führt die Hardwarespezifikationen der Messgeräte auf. Abbildung 33 zeigt den Messaufbau, der sich durch eine Distanz (Luftlinie) von 71km zwischen Server und Client auszeichnet.

Tabelle 3: Hardwarespezifikationen der Messgeräte (WAN)

Modellname Funktion	Desktop-PC Server	Lenovo Thinkpad Edge 540 PC-/Windows-Client
CPU	Intel Core i7-2600	Intel Core i5-4200M
GPU	Nvidia Geforce 660 Ti	Nvidia Geforce GT 740M
RAM	8192MB	8192MB
Internetanbindung	VDSL @ 50MBit/s	DOCSIS @ 32MBit/s

Abbildung 33: Distanz zwischen Client und Server

Die gemessene Latenz beträgt bei einem Paketrouting über das Internet

$$188,5ms \pm 8,37ms$$

Dieses Ergebnis deckt sich mit der eingangs getroffenen Vermutung, dass die Latenz steigt. Vergleicht man den gemessenen Wert mit den Werten der Messung innerhalb des LAN(s. Tabelle 1), so ergibt sich ein Wachstum von etwa 45 Millisekunden. Dieser Wert deckt sich mit denen, die von Nvidia für den Internetversand veranschlagt wurden (vgl. Abb. 3, S. 3). Nvidia selbst geht von 30 zusätzlich entstehenden Millisekunden für ein Internetrouting aus, der *Cloud Gen I*-Dienst brauchte dagegen noch 75 Millisekunden.

6.3 Bandbreite

Die Größe des zu sendenden Datenstroms spielt beim Remote Rendering eine fast genau so große Rolle, wie die Höhe der Latenz. Es ist zwar von einer weiter steigenden Versorgung der Haushalte mit Internetanschlüssen höherer Bandbreite auszugehen, doch um auch heute schon Remote Rendering anbieten zu können, muss untersucht werden, wie „teuer“ qualitativ ansprechende Bilder sind. Ziel ist es, einen möglichst guten *Tradeoff* aus Bandbreite und dazugehöriger Bildqualität zu finden.
Im Grunde genommen fallen bei der Videokompression vier Methoden zur Reduzierung des übermittelten Datenstroms ins Auge. Erstens: Ein Video besteht aus einer konsekutiven Folge von Bildern, die sich in ihrer Information jeweils leicht unterscheiden. Erst durch die Trägheit des menschlichen Gehirns wird der Unterschied der Bilder zueinander als Bewegung interpretiert. Der Schwellenwert ab dem Menschen eine Folge von Bildern als bewegtes Bild wahrnehmen, liegt bei 18Hz bis 24Hz[15]. Es sind also mindestens 18 Bilder pro Sekunde nötig, um den Eindruck einer Bewegung zu erwecken. Die erste Option zur Reduzierung der Größe des Datenstroms besteht darin, die Anzahl der, pro Sekunde übermittelten, Bilder (FPS) zu reduzieren. Der Zusammenhang zwischen FPS und der Größe des Datenstroms ist linear, d.h. sollen doppelt so viele Bilder pro Sekunde übertragen und angezeigt werden, ist der resultierende Datenstrom auch doppelt so groß.
Die zweite Variante stellt eine Verringerung der Auflösung dar. Halbiert man beispielsweise die Auflösung eines HD-Bildes (1920 x 1080), resultiert dieses in einer viermal geringeren Pixelmenge. Der so entstehende Datenstrom ist ebenfalls etwa viermal kleiner.
Über die Veränderung der Bitrate lässt sich die Kompressionsrate ebenfalls regulieren. Unter dem Terminus Bitrate versteht man die Anzahl verwendeter Bits pro Zeiteinheit, um ein kontinuierliches Medium zu beschreiben. Die Bitrate eines Videos ließe sich z.B. durch den folgenden Term ermitteln: Größe des Videos $\cdot\, 8 \div$ Länge des Videos.
Die angeführten Abbildungen (Abb. 34 und Abb. 35) zeigen die Auswirkung der Bitrate auf die Bildqualität. Links wird die komplette Aufnahme der Szene gezeigt, rechts der vergrößerte Inhalt des roten Kästchens. Die Bitrate wurde zwischen beiden Aufnahmen verzehnfacht. Die Unterschiede sind deutlich an den Fronten der Gebäude auszumachen. Abbildung 34 wirkt insgesamt deutlich unruhiger, besonders Farbverläufe werden nicht korrekt dargestellt. Betrachtet man die vergrößerten Ausschnitte so fallen weitere Farbverfälschungen auf, die sich bei der Wahl einer zu geringen Bitrate einstellen.
Zu guter Letzt lässt sich über das Hinzuschalten der bereits in Abschnitt 5.1.2 angesprochenen Deltakompression benötigte Bandbreite einsparen. Innerhalb dieses Anwendungsgebietes ist diese Variante jedoch nicht praktikabel, da der Video-Stream in Echtzeit kodiert wird und unvorhersehbare Nutzereingaben zu starken Störungen des Bildes führen können. Abbildung 36 zeigt, wie sich die Verwendung von Deltakompression im Kontext des Remote Renderings auf die Bildqualität auswirken kann.

[15] http://de.wikipedia.org/wiki/Bewegte_Bilder (30.05.2014)

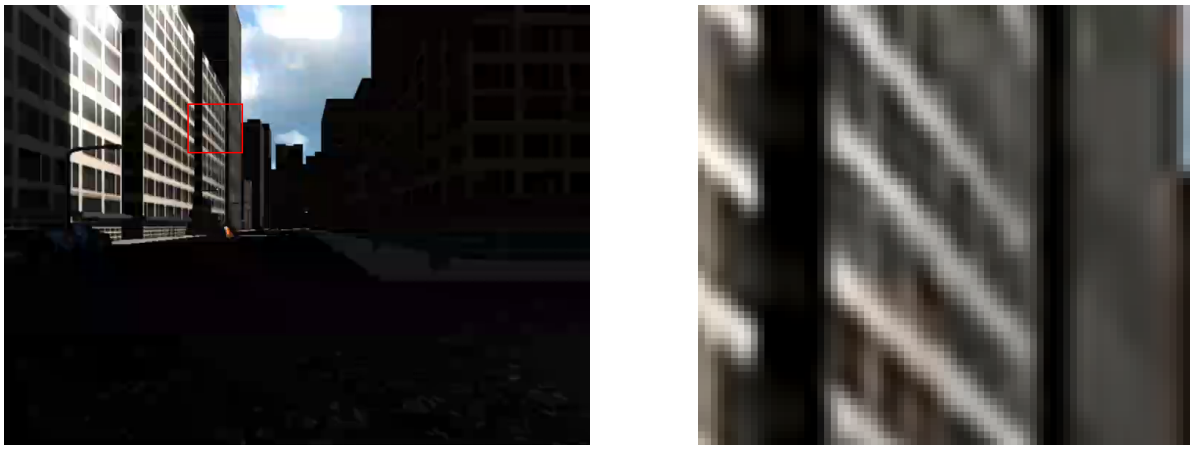


Abbildung 34: Bildqualität bei einer Bitrate von 50kBit

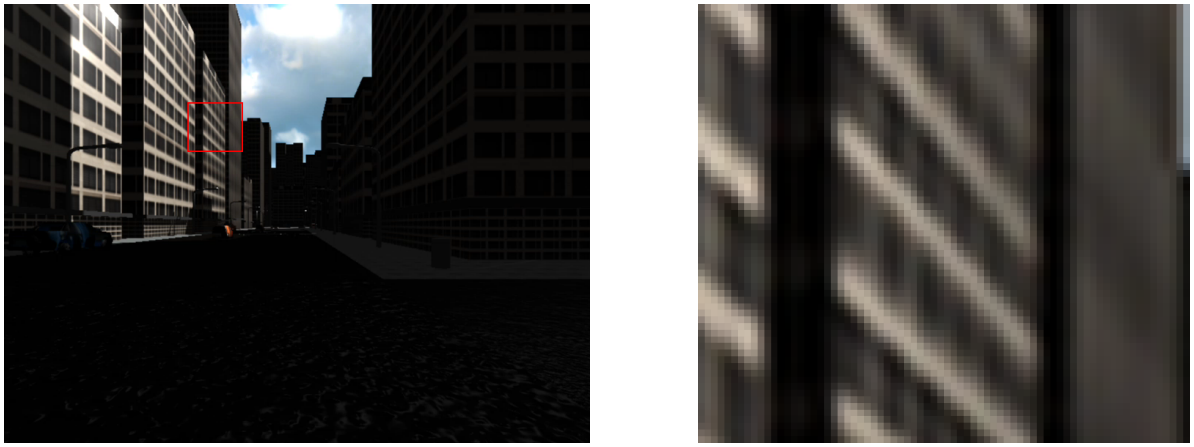
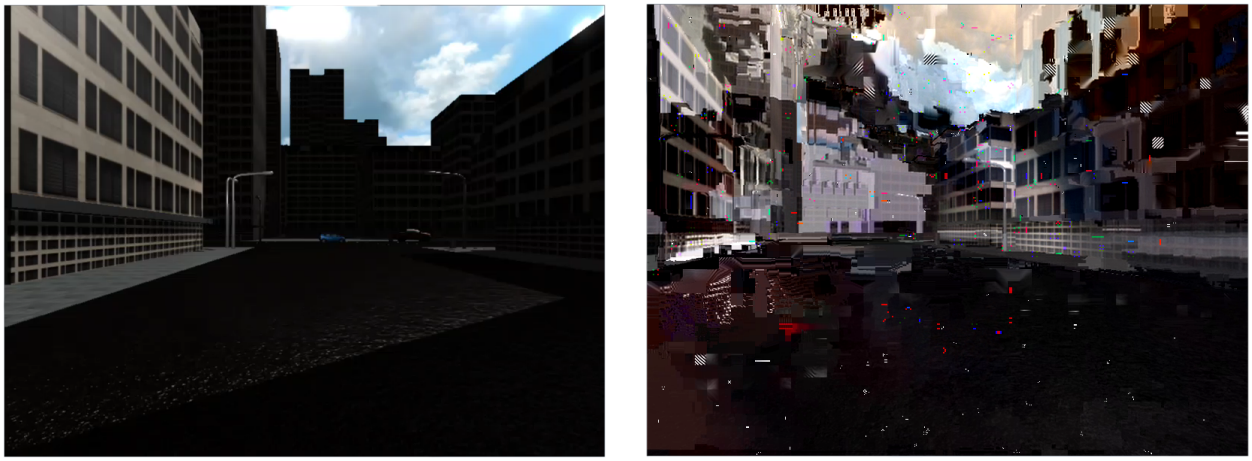


Abbildung 35: Bildqualität bei einer Bitrate von 500kBit



(a) Ohne Deltakompression **(b)** Mit Deltakompression

Abbildung 36: Auswirkung der Verwendung von Deltakompression im Kontext von Remote Rendering

Im Folgenden soll die Auswirkung verschiedener Bitraten, Auflösungen und Bildwiederholungsraten auf die Größe des Datenstroms diskutiert werden. Gemessen wurde zuerst mit Renderer B.

Tabelle 4: Datenstromgröße in Abhängigkeit von FPS, Bitrate und Auflösung, Renderer B

Auflösung und FPS	50kBit	300 kBit	500 kBit
800 x 600 @ 30fps	520kBit	520kBit	520kBit
800 x 600 @ 60fps	1mBit	1mBit	1mBit
1920 x 1080 @ 30fps	1,4mBit	1,5mBit	1,5mBit
1920 x 1080 @ 60fps	2,8mBit	3,1mBit	3,1mBit

Die Messergebnisse von Renderer A werden dem gegenüber gestellt. Die dargestellte Szene ist bezüglich der Farbvielfalt deutlich komplexer, daher führt die Veränderung der Bitrate in diesem Szenario zu einer deutlich größeren Veränderung des entstehenden Datenstroms. Da die vorliegende Version dieser Szene die Full HD Auflösung von 1920x1080 nicht unterstützt, können nur die Ergebnisse unter der Auflösung 800x600 angeführt werden.

Tabelle 5: Datenstromgröße in Abhängigkeit von FPS und Bitrate, Renderer A

FPS	50kBit	300 kBit	500 kBit
800 x 600 @ 30fps	1,42mBit	8,5mBit	14,3mBit
800 x 600 @ 60fps	2,84mBit	17mBit	28,6mBit

Da die Bitrate eine Größenbeschränkung des Datenstroms pro Zeiteinheit darstellt, wirkt sie sich linear auf die Größe aus. Ist das zu kodierende Bild jedoch so einfach gehalten, dass die Schranke nicht erreicht wird, bleibt auch der resultierende Datenstrom unbeeinflusst. Dieses Verhalten wird in Tabelle 4 besonders deutlich. Sind die zu kodierenden Bilder dagegen komplexer, ergibt sich ein ebenso linearer Zusammenhang zwischen Bitrate und Datenstromgröße wie bei Auflösung und FPS (s. Tabelle 5).

7 Fazit

Die Technologie des Remote Renderings knüpft an den aktuellen Trend der immer größer werdenden Vernetzung an. War es vor zehn Jahren üblich, Musik zum Hören auf ein Handy zu laden, wird dieses heute zu einem großen Teil von Diensten wie Spotify, SoundCloud oder LastFm übernommen. Der Vorteil dieser Dienste ist offensichtlich: Die Musik muss nicht lokal als Datei vorgehalten werden, sondern kann von überall und mit jedem Gerät abgerufen und angehört werden. Dasselbe Bild bietet sich auf dem Videomarkt. Anstatt DVD ist der neueste Trend auch hier das Streamen auf mobile Geräte. Wahlweise können Nutzer dieser Dienste Filme gegen Bezahlung, ähnlich einer globalen Videothek, anschauen oder Inhalte freier Videoplattformen, wie z.B. Youtube oder Dailymotion, kostenlos konsumieren. Der nächste logische Schritt dieser Entwicklung könnte das Remote Rendering sein. Schon heute setzen immer mehr Unternehmen auf diese Technik und werten ihre Produkte so auf. Remote Rendering birgt, genau wie andere Cloud-Dienste, den Vorteil, dass einmal gekaufte Inhalte überall abrufbar sind und das geräteübergreifend synchron.
Abschließend lässt sich festhalten, dass das Ziel dieser Arbeit erreicht wurde. Es ist gelungen, verschiedene, teilweise schon existierende Technologien so ineinander greifen zu lassen, dass ein plattformunabhängiges Grafikerlebnis geschaffen wird. Besonders der Aspekt von Latenz und entstehendem Datenvolumen konnte im Zuge der Implementierung untersucht und analysiert werden. Dabei sind die entstandenen Latenzzeiten der Anwendung in einem anwenderfreundlichen Bereich. Die Arbeit an der Remote Rendering API und den Clienten ist nach der Abgabe dieser Arbeit noch nicht abgeschlossen, jedoch gibt der aktuelle Stand bereits einen guten Überblick über die Möglichkeiten der Technologie.

8 Ausblick

Die folgenden Aspekte können nach der Abgabe dieser Arbeit weiter optimiert werden:

Verwendung eines Hardwareencoders
: Die Kompression der Szenenbilder durch einen dedizierten Hardwareencoder verspricht eine massive Beschleunigung der Kodierung. Verglichen mit der hier verwendeten GPU-unterstützten Kompression, kann die Bearbeitungszeit bis Faktor 5 beschleunigt werden[16]. Dadurch könnte die entstehende Latenz weiter reduziert werden.

Verwendung eines Hardwaredecoders
: Auch auf Seiten des Clienten gilt, dass eine Hardware-beschleunigte Dekompression weniger Zeit benötigen wird, als eine in Software. Auf der anderen Seite wird so jedoch die Menge der potentiellen Anwender reduziert, da ein Gerät mit Chip zur Hardwaredekompression erforderlich ist.

Multithreading
: Es besteht die Idee, Übertragung und Empfang von Steuerevents an separate Threads zu delegieren, um eine noch schnellere Reaktion des Systems auf Eingaben zu ermöglichen.

[16] http://www.tomshardware.com/reviews/sandy-bridge-core-i7-2600k-core-i5-2500k,2833-5.html (29.05.2014)

A Abkürzungen

API
Application Programming Interface: „Ein Programmteil, der von einem Softwaresystem anderen Programmen zur Anbindung an das System zur Verfügung gestellt wird.“[17]

APP
Applikation, zu deutsch Anwendungssoftware.

CPU
Central Processing Unit: Zentrale Verarbeitungseinheit, bzw. Hauptprozessor eines Computersystems.

DOCSIS
Data Over Cable Service Interface Specification: Spezifikation für Schnittstellen von Kabelmodems. Wird zur Datenübertragung innerhalb eines bestehenden Kabelfernsehnetzes verwendet.

FPS
Frames Per Second: Pro Sekunde angezeigte Bilder, normalerweise sollten mindestens 18 Bilder pro Sekunde angezeigt werden, um den Eindruck von Bewegung zu erzeugen.

GPU
Graphics Processisng Unit: Die Grafikkarte eines Computersystems. Im Gegensatz zur CPU liegt der Schwerpunkt der GPU nicht auf der Flusskontrolle, bzw. Steuerung von Prozessen, sondern auf der Berechnung und Anzeige verschiedenster Grafiken.

GPGPU
General-Purpose Computing on Graphics Processing Units: Unter GPGPU versteht man die Verwendung der Grafikkarte für andere Zwecke als Computergrafik.

LAN
Lokal Area Network: Ein lokales Netzwerk, das aus einigen wenigen Rechnern und anderen netzwerkfähigen Geräten besteht. Die geographische Ausdehnung eines LAN übersteigt in den meisten fällen keine 500 Meter.

RAM
Random-Access Memory: Arbeitsspeicher eines Computersystems. Im Gegensatz zur Festplatte nicht permanent, dafür deutlich schneller.

RR
Remote Rendering: Eine Technologie, die grafische Inhalte auf Servern berechnen lässt und diese dann, als „interaktives Video“ an Clienten verschickt.

TCP
Transmission Control Protocol: TCP ist ein verbindungsorientiertes Protokoll, das den Datenaustausch zwischen zwei Computern ermöglicht. Im Gegensatz zu UDP ist TCP zuverlässig.

[17] http://de.wikipedia.org/wiki/Programmierschnittstelle(08.05.2014)

UDP
: User Datagram Protocol: UDP ist ein minimales, verbindungsloses Protokoll, das den Datenaustausch zwischen mehreren Computern ermöglicht.

WAN
: Wide Area Network: Im Gegensatz zum LAN ist die Ausdehnung eines WAN deutlich größer und kann sogar Kontinente umfassen. Weitverkehrsnetze (WAN) haben dabei die Aufgabe verschiedene lokale Netze (LAN) zu verbinden. Um dies zu gewährleisten bedarf es einer eindeutigen Adressierung einzelner Rechner im WAN, sowie Zwischensystemen die die Pakete weitervermitteln.

VDSL
: Very high speed Digital Subscriber Line: Beschreibt Standards zur Datenübertragung über herkömmliche Telefonanschlussleitungen. Erreicht wesentlich höhere Übertragungsgeschwindigkeiten als herkömmliches DSL oder ADSL.

B Begriffserklärung

Client

Ein Client ist ein Programm, das von einem Nutzer ausgeführt werden kann. Es steht dabei in einer Client-Server-Verbindung. In diesem Gefüge besteht die Hauptaufgabe des Clienten darin, Informationen, die der Server lokal vorhält, anzuzeigen. Es gibt verschiedenste Arten von Clienten: Browser stellen HTML-Code dar, der von Webservern abgerufen werden kann; Email-Clienten verbinden sich zu Email-Servern und laden so neue E-Mails; mehrspielerfähige Computerspiele stellen oftmals auch einen Clienten dar, da sie sich zu einem Server verbinden müssen, der die globalen Positionen aller Spieler vorhält.

Server

Ein Server stellt die Gegenseite des Clienten dar. In den meisten fällen herrscht eine 1 zu n Beziehung zwischen Server und Clienten, d.h. ein Server kann durchaus mehrere Clienten mit Informationen versorgen. Ziel eines Servers ist es, Daten lokal vorzuhalten, sodass diese von verschiedenen Positionen abgerufen werden können.

RR-Client

Ein Client der mit einem Remote Rendering Server Verbindung aufnimmt, um dessen Daten zu visualisieren.

RR-Server

Ein Server, der eine Szene berechnet und Szenenbilder, mit Hilfe der Remote Rendering Technologie, zur Anzeige auf einem Clienten bereit stellt.

Producer, Consumer

Der Terminus Producer bezeichnet einen Prozess oder einen Thread, der eine Ressource bereitstellt. Diese Ressource wird nach Bereitstellung vom Consumer abgegriffen und weiterverarbeitet.

Bandbreite

Bezeichnet die Datenübertragungsrate von Internetleitungen. Eine höhere Bandbreite ist meistens besser.

Ethernet

Technologie, die den Austausch von Daten zwischen Geräten eines lokalen Datennetzes steuert.

C Code

C.1 Callback-Funktionen der Nvcuvenc-Bibliothek

```
1  //Optional, muss nicht implementiert werden
2  static void __stdcall HandleOnBeginFrame(
     ↪ const NVVE_BeginFrameInfo* frameInfo, void* pUserData){}
3  //Optional, muss nicht implementiert werden
4  static void __stdcall HandleOnEndFrame(                    const
     ↪ NVVE_EndFrameInfo* frameInfo, void* pUserData){}
5
6  static unsigned char* __stdcall HandleAquireBitStream(
     ↪ int* pBuffersize, void* pUserData)
7  {
8     RemoteEncoder *remo = NULL;
9     if(pUserData)
10    {
11       remo = (RemoteEncoder * )pUserData;
12       *pBuffersize = remo->getWidth()*remo->getHeight()*3/2;
13       return remo->GetCharBuf();
14    }
15 }
16 static void __stdcall HandleReleaseBitStream(
     ↪ int nBytesInBuffer, unsigned char* cb, void* pUserData)
17 {
18    RemoteEncoder *remo = NULL;
19    if(pUserData)
20    {
21       remo = (RemoteEncoder * )pUserData;
22       char* msg = new char[sizeof(UINT8) + sizeof(
           ↪ unsigned char) * nBytesInBuffer + sizeof(int)];
23       memcpy(msg, &FRAME_DATA, sizeof(UINT8));
24       memcpy(msg + sizeof(UINT8), &nBytesInBuffer,
           ↪ sizeof(int));
25       memcpy(msg + sizeof(UINT8) + sizeof(int), cb,
           ↪ sizeof(unsigned char) * nBytesInBuffer);
26
27       int numBytes = remo->getClient()->Send(msg,
           ↪ sizeof(UINT8) * nBytesInBuffer + sizeof(int));
28       delete [] msg;
29    }
30 }
```

Listing 8: Implementierung der Callback-Funktionen der Nvcuvenc-API

C.2 Funktion zur Verarbeitung von weitergereichten Nutzereingaben

```
1  //Beachte: Rückgabewert und Parameterliste
2  void RRKeyCallback(int key, bool pressed)
3  {
4     std::string tmp = pressed ? "pressed" : "released";
5     printf("Key %d %s \n", key, tmp);
6     if(pressed)
7        //Hier könnte z.B. eine Kamera bewegt werden oder
8        //anderweitig Steuerung realisiert werden
9        keyStates[key] = true;
10    else
11       keyStates[key] = false;
12 }
```

Listing 9: Beispielhafte Funktion des Grafikprogramms zur Verarbeitung der Tastatureingaben

D Literaturverzeichnis

[1] David B. Kirk, Wen-Mei W. Hwu:
Programming Massively Parallel Processors, hg. von Morgan Kaufman, 2010

[2] Jason Sanders, Edward Kandrot:
Cuda by Example: An Introduction to General-Purpose GPU Programming, hg. von Addison Wesley, 2010

[3] Grid Presentation: Geforce_Grid_Press_Presentation.pdf
URL: http://assets.sbnation.com/assets/1119979/GeForce_Grid _Press_Presentation.pdf, hg. von Nvidia, 2012

[4] Nvidia Cuda Video Encoder: nvcuvenc.pdf,
URL: http://docs.nvidia.com/cuda/samples/3_Imaging/cudaEncode/doc/nvcuvenc.pdf
hg. von Nvidia, 2010

[5] Nvidia Cuda Video Decoder: nvcuvid.pdf,
URL: http://docs.nvidia.com/cuda/samples/3_Imaging/cudaEncode/doc/nvcuvenc.pdf
hg. von Nvidia, 2008

[6] Wright, Haemel, Sellers, Lipchak:
OpenGL SuperBible, hg. von Addison Wesley, 2010

[7] Bob Quinn, Dave Shute:
Windows Sockets Network Programming, hg. von Prentice Hall, 2011

[8] Timo Bourdon:
Entwicklung einer WebGL-Applikation zur kundenspezifischen Konfiguration von Automobilen
http://www.inf.uos.de/prakt/pers/dipl/BourdonBachelorarbeit.pdf (24.05.2014)

[9] Jim Kurose, Keith Ross
Computer Networking: A Top Down Approach, hg. von Addison-Wesley, 2009

E Abbildungsverzeichnis

F Listing-Verzeichnis

G Tabellenverzeichnis

Erklärung zur selbstständigen Abfassung der Bachelorarbeit

Ich versichere, dass ich die eingereichte Bachelorarbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als der von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommenen Stellen habe ich kenntlich gemacht. Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und nicht veröffentlicht.

Osnabrück, den 18.06.2014

Christoph Eichler