UNIVERSITÄT OSNABRÜCK

Remote Rendering zur Device-unabhängigen Darstellung von hochaufgelösten 3D-Fahrzeugdaten im Web

Timo Bourdon

Masterarbeit

zur Erlangung des akademischen Grades

Master of Education

Universität Osnabrück

Fachbereich 6: Mathematik und Informatik

Erstgutachter: Prof. Dr. Oliver Vornberger
Zweitgutachter: Prof. Dr.-Ing. Elke Pulvermüller

Zusammenfassung

Unter Remote Rendering versteht man die Auslagerung rechenintensiver Grafikberechnungen auf externe Server. Ein Client kann über das Internet eine Verbindung zum Server aufnehmen und über Nutzereingaben beispielsweise eine 3D-Szene manipulieren. Der Server verarbeitet die Nutzerinteraktionen und sendet die berechneten Bilder im Rahmen einer Echtzeit-Darstellung an den Nutzer zurück. Durch die Auslagerung der Grafikberechnungen können insbesondere mobile Endgeräte, wie Smartphones, Tablets und Notebooks mit geringer Grafikrechenleistung auch hochauflösende 3D-Szenarien qualitativ hochwertig anzeigen. Zielsetzung dieser Arbeit ist es, einen 3D-Autokonfigurator um einen Remote-Renderer zu erweitern, sowie eine device-unabhängige Webschnittstelle für Endnutzer zu erstellen. Im Rahmen einer Anforderungsanalyse wird die Anwendung auf verschiedenen Endgeräten in unterschiedlichen Netzwerkszenarien getestet und analysiert. Eine weitere Evaluation wird die Integrierbarkeit der von AdaptVis bereitgestellten Remote-Renderer API untersuchen und Voraussetzungen an Anwendung und Hardware diskutieren.

Abstract

Remote Rendering is defined as the outsourcing of computationally intensive graphics calculations to external servers. A client connects to a server via the internet and is able manipulate a 3D scene by key-, mouse-, or touch inputs. The server processes the user interactions and sends the rendered frames back to the user as a real-time representation. By outsourcing the graphics computations mobile devices such as smartphones, tablets and laptops with low graphics computing power are able to display high-resolution 3D scenarios in high-quality. In this work, a 3D car configurator is to be extended by a remote renderer, and a device- and plattform-independent web interface needs to be created. As part of a requirement analysis the application will be tested on different devices in different network scenarios to show up the conditions for applications and hardware systems. A further evaluation will examine the integrability of the remote renderer API provided by AdaptVis.

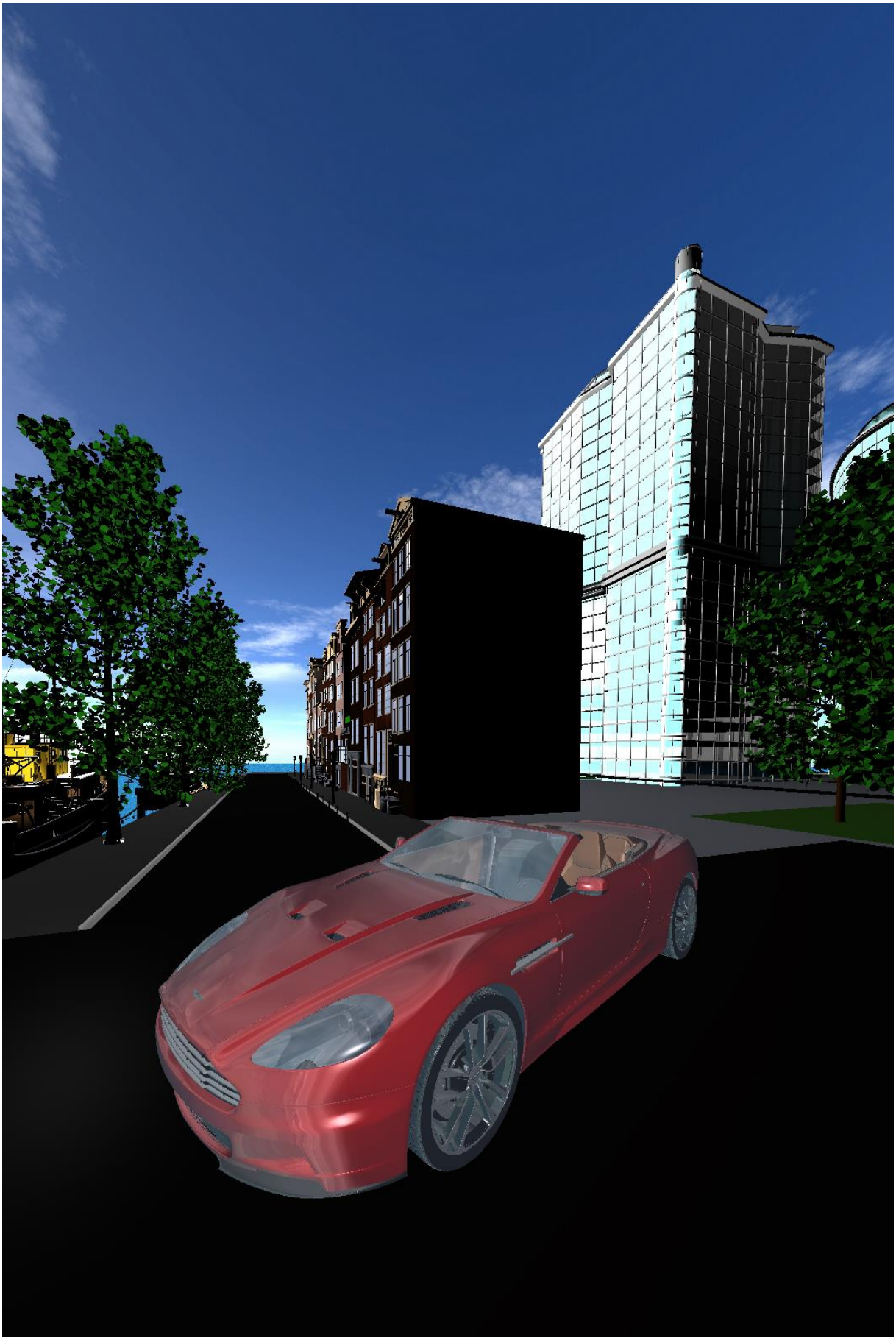


Abbildung 1: Fahrzeugansicht (Desktop Version)

Abbildung 2: Fahrzeugansicht (Web Version)

Inhaltsverzeichnis

Danksagung

Danken möchte ich an dieser Stelle meinen *Eltern* ohne die mein Studium insbesondere aus finanzieller Sicht sehr schwer geworden wäre. Sie standen immer mit Rat und Tat zur Seite und fanden zu jeder Zeit die richtigen Worte, um zu motivieren oder den Kurs zu ändern. Dieser Dank gilt auch meinen beiden *Geschwistern*.

Ebenfalls möchte ich meiner Freundin *Vera* danken, dass sie all die Jahre immer für mich da war und mich durch gute und schlechte Zeiten begleitet hat.

Ein großer Dank geht an *Henning Wenke*. Henning war zu jederzeit da und trug maßgeblich mit seiner ruhigen und konstruktiven Art zum Gelingen der Arbeit bei. Dies war insbesondere auch dann der Fall, als er mit seinen beiden Kollegen *Sascha Kolodzey* und *Erik Wittkorn* die AdaptVis GmbH gründete und neben seiner Promotion somit viele eigene Aufgaben in seiner Pflicht standen. Ein Betreuer wie Henning ist keine Selbstverständlichkeit und das Wort „Danke“ zu wenig um den Wert zu beschreiben.

An dieser Stelle gilt auch *Sascha* und *Ecki* großer Dank für die immerwährende Bereitschaft, technische Fragen und Problemstellungen zu klären. Danke ebenfalls an *Christoph Eichler* für seine Geduld und Unterstützung bei Fragen um C++.

Danke sagen möchte ich ebenfalls *Prof. Vornberger*. Zum einen natürlich für die Bereitschaft der Betreuung der Masterarbeit, zum anderen aber auch für seinen Einsatz und sein Vertrauen abseits der Abschlussarbeit. Ebenfalls möchte ich mich bei *Prof. Pulvermüller* für die Zweitkorrektur bedanken.

Zu guter Letzt sind es auch die *Freunde* die 14 Semester Studium in und abseits der Uni ungemein bereichert haben. Vielen Dank an Euch alle!

1 Motivation

1.1 Automobilkonfiguratoren

Mit Werbeslogans wie „Volkswagen, das Auto!“, „Audi, Fortschritt durch Technik!“ oder „Ford, immer eine Idee weiter!“ preisen zahlreiche Automobilhersteller ihre Fahrzeug-Palette im Fernsehen an und präsentieren sich einem breiten Publikum. Blickt man Jahre zurück funktionierte Autowerbung ausschließlich über Radio und Print-Medien. Wollte ein Kunde einen Eindruck über ein bestimmtes Fahrzeug gewinnen, so blieb diesem lediglich der Gang zum Autohändler. Vor Ort boten neben dem ausgestellten Fahrzeug, schriftliche Ratgeber zu jedem Fahrzeugmodell einen Einblick in die Leistungsvielfalt sowie die verschiedenen technischen und optischen Varianten (siehe Abbildung 3).

Abbildung 3: VW Karmann Ghia Typ 14 Modell-Prospekt (Blog, 2016)

Durch die Eroberung des Internets als multimediale Werbeplattform der Automobilhersteller erweiterte sich zunehmend das publizierte Online-Angebot. Im Jahre 2010 wurden von Automobilherstellern in Deutschland rund 1,46 Mrd. Euro in jegliche Werbe-Maßnahmen investiert, während im Jahr 2014 sogar 1,81 Mrd. Euro errechnet wurden (Nielsen, 2016). Als stärkste treibende Kraft der Online-Werbung stellte sich die Automobilindustrie heraus, die mit Investitionen in Höhe von ca. 188,7 Mio. Euro im Jahr 2014 an erster Stelle der beworbenen Produkte stand (Nielsen, 2014). Dass die

Webseiten vieler Automobilhersteller hohe Besucherzahlen aufweisen, liegt unter anderem daran, dass die Nutzer mit einer Vielzahl interaktiver Inhalte angesprochen werden. Ein Beispiel sind Automobilkonfiguratoren mit denen sich die aktuelle Fahrzeug-Palette nach Wunsch und Geschmack zusammenstellen lassen (Nielsen, 2014). Diese Konfiguratoren basieren oftmals auf Bildern der Fahrzeuge aus mehreren Perspektiven (siehe Abbildung 4), welche den Einstellungen des Nutzers gemäß angepasst werden.

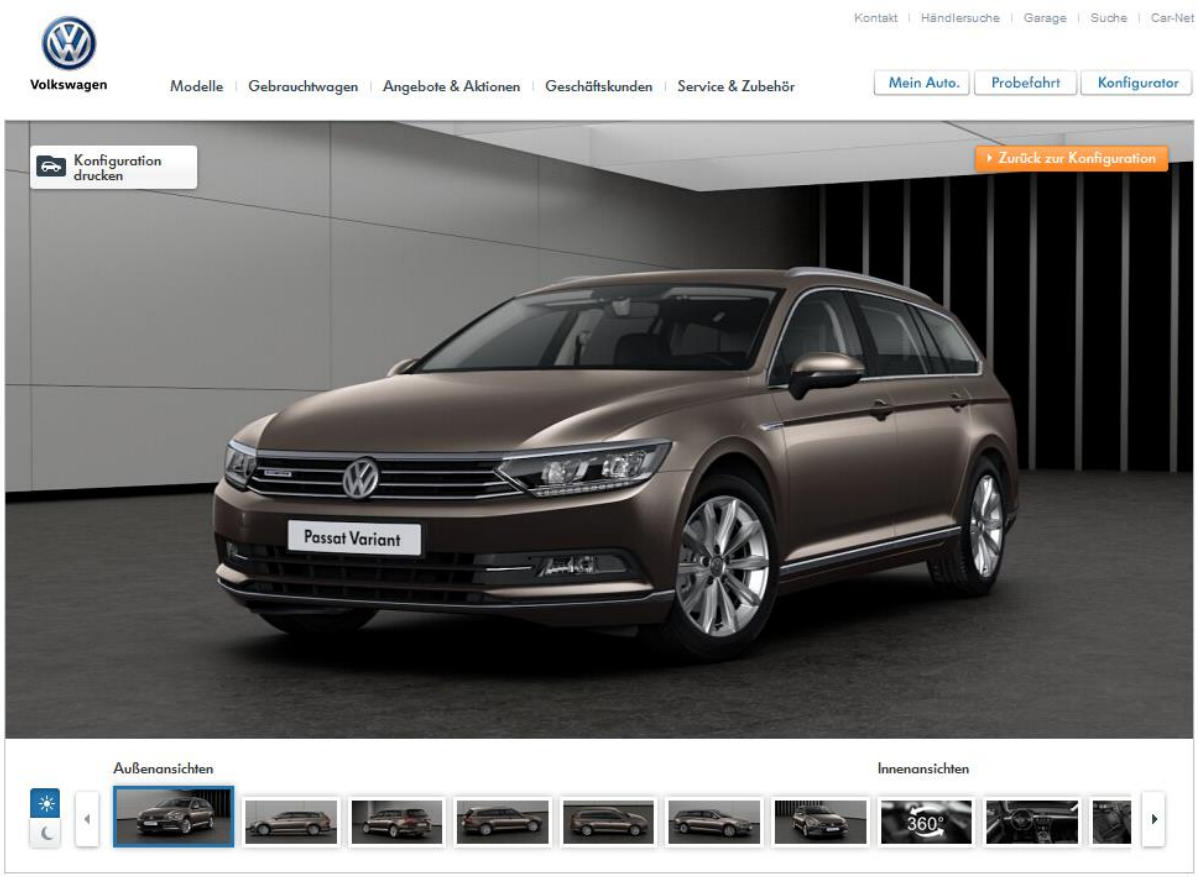


Abbildung 4: VW Automobil-Konfigurator (Volkswagen, 2016)

Porsche geht dabei einen Schritt weiter und bietet einen auf der Game-Engine Unity basierenden Autokonfigurator an, der in der Lage ist 3D-Modelle der Fahrzeuge darzustellen (siehe Abbildung 5).

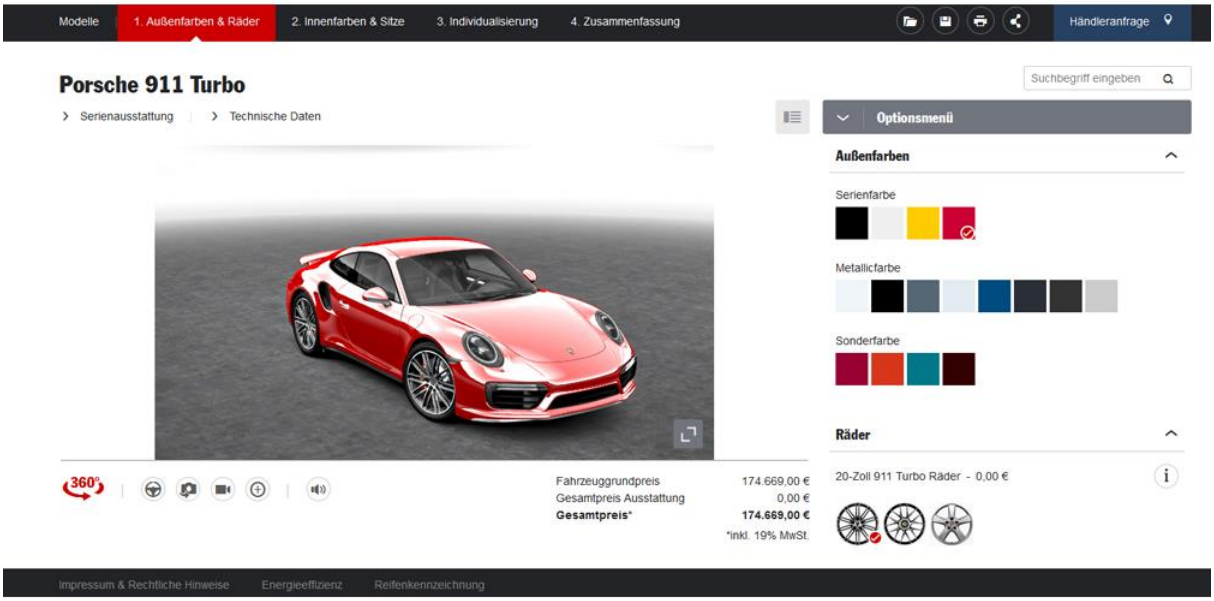


Abbildung 5: Porsche Automobil-Konfigurator (Porsche, 2016)

Dieses Vorgehen bietet, in Verbindung mit einem sehr anwenderfreundlichen Zugang, vollkommen neue Möglichkeiten der Interaktivität und Darstellung. wie 360°-Rundumsichten, fließende Kamerafahrten und einen erhöhten Detailgrad bei Nahansichten.

Repräsentative Umfragen zu Nutzerstatistiken der Webseiten von Automobilherstellern motivieren seit der Etablierung mobiler Endgeräte in den multimedialen Alltag ebenfalls die Ausweitung der Internet-Präsenz auf Smartphones und Tablets. An Hand der Daten einer Umfrage durch die Nielsen Media Research GmbH wird dies am Beispiel des Herstellers Audi sehr deutlich und erklärt den zunehmenden Fokus auf Automobil-Konfiguratoren und deren Wichtigkeit in Bezug auf das Erreichen neuer Nutzer und Vermarktung der Marke.

Die folgende Grafik zeigt einen Ausschnitt aus der veröffentlichten Nutzerstatistik. Diese beschreibt die Zahl der Besucher im April 2012 und deren Verhalten auf der Webseite von Audi. Insgesamt wurden 410.67 Nutzer gezählt, die im Mittel zirka 21 Minuten auf

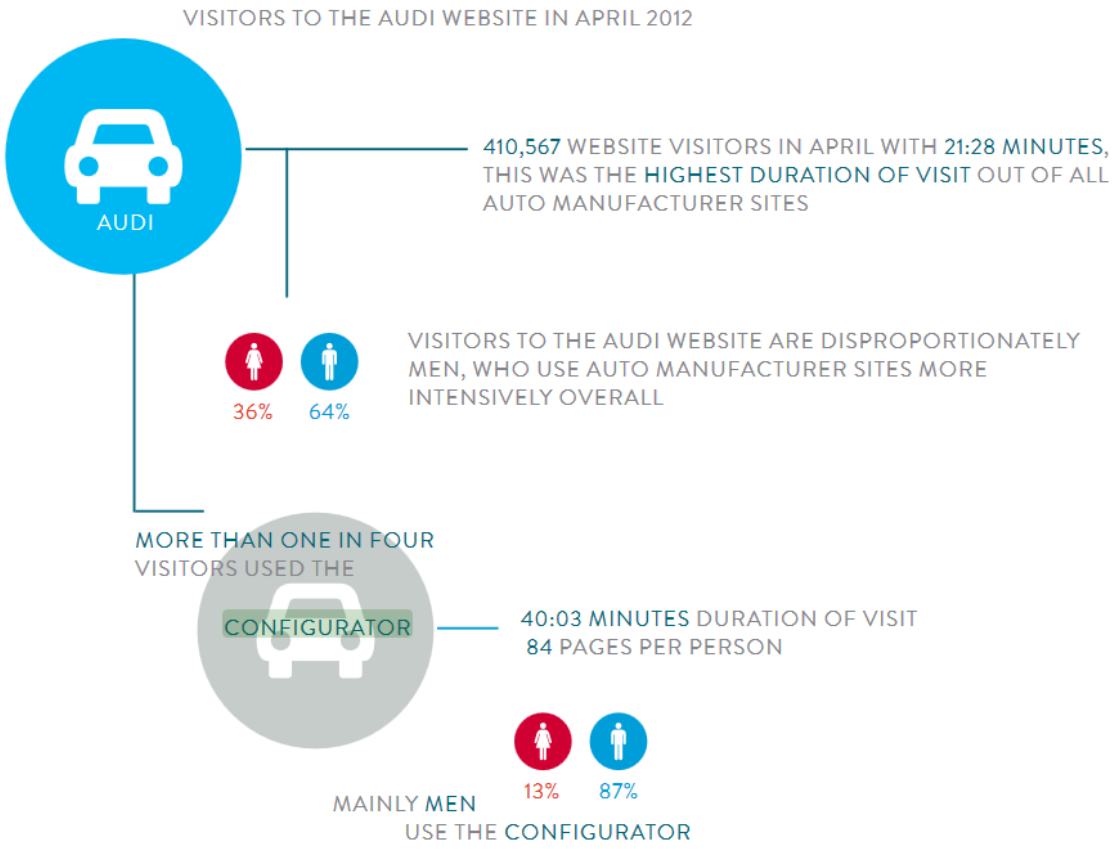


Abbildung 6: Nutzerstatistik Audi Webpräsenz (Nielsen, 2012)

der Internetpräsenz verbracht haben (Nielsen, 2012). Der Fokus dieser Statistik lag insbesondere auf der zeitlichen Erfassung der Verwendung des Automobilkonfigurators. Hier wurde bei jedem vierten Nutzer eine Besuchszeit von zirka 40 Minuten verzeichnet, wobei dieser Service insbesondere von männlichen Besuchern angenommen wurde (Nielsen, 2012). Diese Werte und weitere Beobachtungen der Entwicklung von Online-Präsenzen der Automobil-Branche zeigen die Attraktivität von Automobil-Konfiguratoren und rechtfertigen insbesondere den Vorstoß der Marke Audi mit einer Ausgliederung des bestehenden Konfigurators in eine eigene Standalone-App (Liedke, 2013) welche speziell für mobile Endgeräte entwickelt wird. Dass der Bereich der mobilen

Automobilkonfigurator-Versionen noch in den Kinderschuhen steckt, zeigen insbesondere die Web-Versionen bestehender Konfiguratoren.

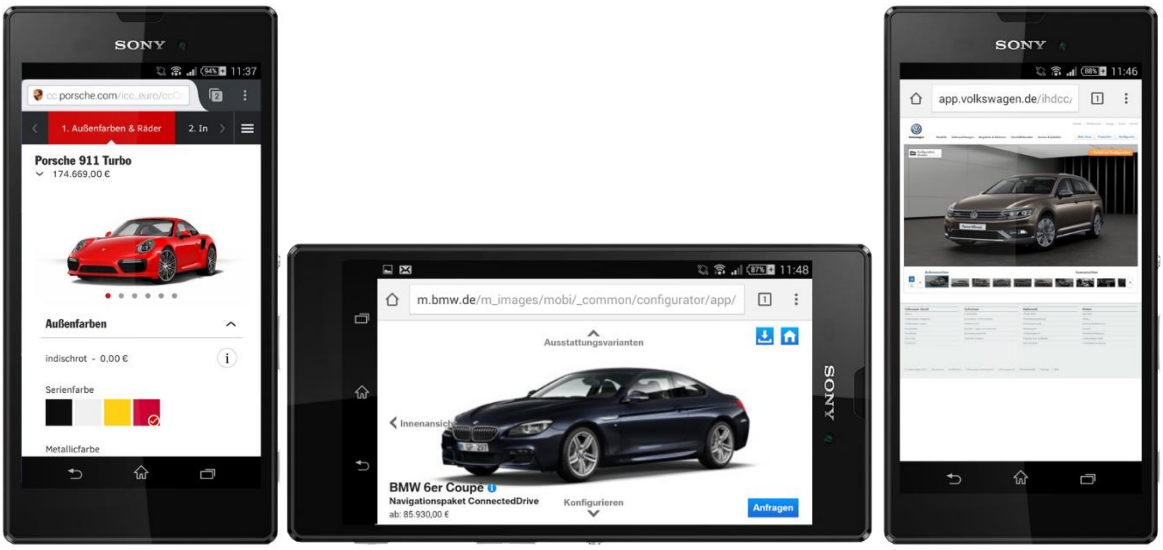


Abbildung 7: Mobile Web-Versionen von Automobilkonfiguratoren (Porsche, BMW, Volkswagen, 2016)

Die in Abbildung 7 aufgeführten Webversionen von Automobilkonfiguratoren deutscher Automobilhersteller zeigen zwar eine geeignete Anpassung auf jeweilige Ausgabeformat, doch wird bei der Interaktivität und Gestaltung des Interface deutlich, dass der Fokus nicht auf einer spezifischen Mobil-Version liegt. Hier gilt es die Entwicklungen der nächsten Monate und Jahre zu beobachten und in wie weit die Automobilbranche die positiven Nutzererfahrungen als Motivation zur Erschließung des mobilen Sektors für Automobilkonfiguratoren nutzen.

1.2 Remote Rendering

Der Computerspielemarkt ist sehr dynamisch und schnell wachsend. Über die Jahre wurde nicht nur die Interaktion zwischen Mensch und virtueller Welt weiterentwickelt, sondern auch der grafische Realismus. Moderne Videospiele fordern heutzutage leistungsfähige Grafikkarten, die lediglich in High-End Computern und Konsolen verbaut werden. Mobile Endgeräte wie Smartphones, Tablets und Notebooks sind auf Grund der integrierten Hardware (Onboard-Grafikkarten) nicht in der Lage, diese Spiele in der maximalen Qualität performant darzustellen weswegen diese allenfalls stark reduziert werden muss. An diesem Problem setzt Remote Rendering an. Unter Remote Rendering versteht man die Auslagerung rechenintensiver Grafikberechnungen auf externe Server in denen leistungsfähige GPUs verbaut sind. Ein Nutzer kann über das Internet eine Verbindung zum Server aufnehmen und über Nutzereingaben (Maus, Tastatur, Touchpad) beispielsweise eine 3D-Szene manipulieren. Der Server verarbeitet die Nutzerinteraktionen und sendet die berechneten und enkodierten Bilder im Rahmen einer Echtzeit-Vorschau an den Nutzer zurück. Durch die Auslagerung der Grafikberechnungen können insbesondere mobile Endgeräte, wie Smartphones, Tablets und Notebooks mit geringer Grafikrechenleistung auch hochauflösende 3D-Szenarien qualitativ anzeigen.

Der Grafikkarten-Hersteller NVIDIA bietet bereits mit seinem Projekt „Shield“ (Übersicht in Abbildung 8) ein derartiges Konzept an wobei NVIDIA GRID das Kern-Element des Konzepts ist (NVIDIA, 2016). GRID rendert 3D-Spiele auf Cloud-Servern, kodiert jeden Frame unmittelbar und streamt das Ergebnis auf verschiedenste Endgeräte (NVIDIA, 2016). Mit dieser Entwicklung ebnet NVIDIA den Weg, wie das Spielen von Videospielen der Zukunft aussehen kann.

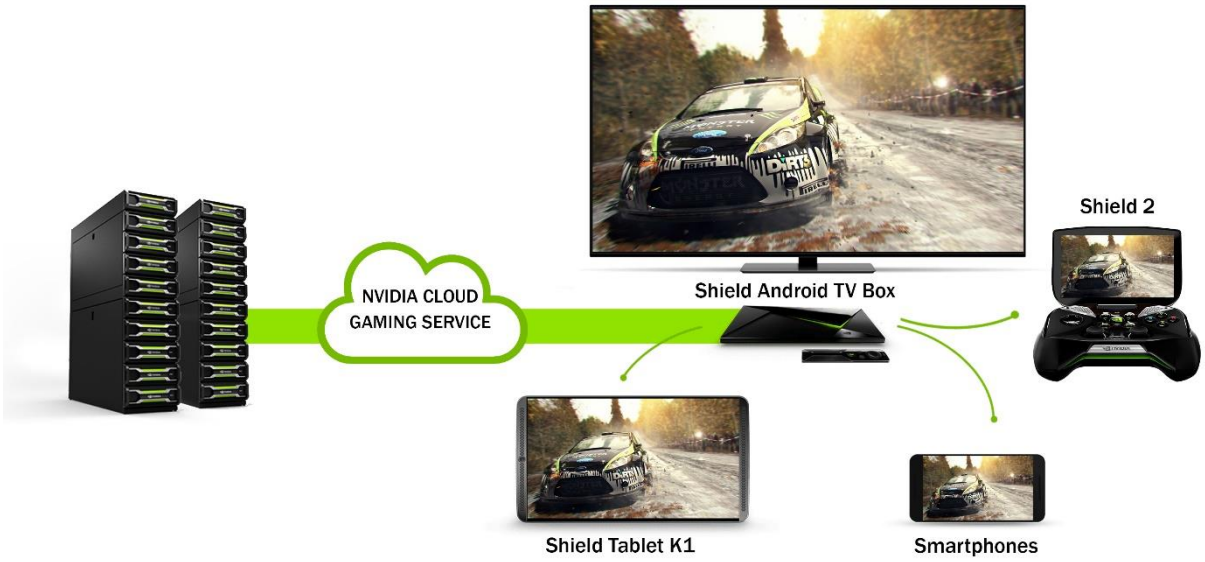


Abbildung 8 : NVIDIA Cloud Gaming Service "Shield" (NVIDIA, 2016)

Die strikte Trennung zwischen Server und Client birgt einen weiteren Vorteil. Der Nutzer erhält lediglich eine Echtzeit-Vorschau der 3D-Daten über einen Stream. Das heißt, dass die Daten der 3D-Anwendung, welche auf dem Server ausgeführt wird, nur dort vorliegen und dem Endnutzer nicht zugänglich sind. Dieser Aspekt ist insbesondere essentiell für rechtlich geschützte und sensible Daten, deren Geheimhaltung vor Dritten eine hohe Priorität hat.

Aus den genannten Aspekten folgt, dass Remote Rendering ebenfalls für Automobilkonfiguratoren eine interessante Technologie ist. 3D-Modelle von Fahrzeugen stellen im Detaillierungsgrad enorme Anforderungen an den Rendering-Prozess sowie die Übertragung der Daten im Internet. Auch hier ist es der Anspruch der Automobilhersteller, ein breites Publikum optimal zu erreichen, die mit unterschiedlichsten Endgeräten und damit verbundener Hardwareleistung und Internet-Bandbreite aufwarten. Ebenso unterliegen die 3D-Daten der Fahrzeuge strengen Datenschutzbestimmungen, weswegen ein direkter Zugriff auf diese durch Endnutzer ausgeschlossen werden sollte.

2 Stand der Bachelorarbeit – Ziele der Masterarbeit

Abbildung 9: Gegenüberstellung Render-Ergebnis
(links: Endergebnis der Masterarbeit; rechts: Endergebnis der Bachelorarbeit

Diese Arbeit baut auf dem Stand meiner Bachelorarbeit auf, in der es um die Entwicklung einer WebGL-Applikation zur kundenspezifischen Konfigurierung von Automobilen ging. Die Implementierung erfolgte mit JavaScript unter Verwendung des WebGL Standards zur Darstellung von 3D-Daten im Web. Das Rendering der relativ einfach gehaltenen Szene, sowie der Fahrzeugdaten findet dabei vollständig auf der Client-Seite statt, wodurch die Qualität der Anzeige an die client-seitig verwendete Hardware gebunden ist (Bourdon, 2013).

Durch die Integration eines Remote Renderers als ein wesentliches Ziel dieser Arbeit, sollen alle rechenintensiven Grafikberechnungen auf einen Server ausgelagert werden. Dadurch können die 3D-Daten unabhängig von der Hardware-Leistung des client-seitig verwendeten Endgeräts qualitativ dargestellt werden. Zunächst wurde der Versuch unternommen den in C++ geschriebenen Remote Renderer von Christoph Eichler (Eichler, 2014) zu integrieren. Im Zuge dessen ist der WebGL-Autokonfigurator in C++ übersetzt worden. Das Vorgehen musste jedoch abgebrochen werden, da die Encoder-

Bibliothek NVCUVENC nicht mehr von aktuellen NVIDIA Grafikkartentreibern unterstützt wurde und von NVIDIA seit August 2014 als „depricated“ d.h. als veraltet ausgeschrieben wurde.

Ein weiterer Remote Renderer wurde von der EXIST-Ausgründung AdaptVis (Wenke H.) bereitgestellt. Die Bestrebungen von AdaptVis zielen darauf ab, dass der in C++ implementierte Remote Renderer auch in Anwendungen integriert werden kann, die nicht in C++ geschrieben sind. Aus diesem Grund fand eine weitere Übersetzung der Autokonfigurator-Anwendung von C++ nach Java statt, um darüber hinaus eine Evaluation zu Integrierbarkeit und Performance zu realisieren. Damit sich die Anwendung neben den technischen Erweiterungen auch optisch weiterentwickelt, soll eine aufwändigere Szene modelliert werden in welche die Fahrzeuge geladen werden. Des Weiteren kommt Environment-Mapping zum Einsatz, um die Spiegelung der Umgebung auf die Fahrzeug-Modelle zu simulieren. Im Folgenden werden einige Impressionen der Arbeit gezeigt, welche die Szene darstellen und den Fokus insbesondere auf einen optischen Vergleich zwischen Bachelor- und Masterarbeit legen.

Szene der Masterarbeit

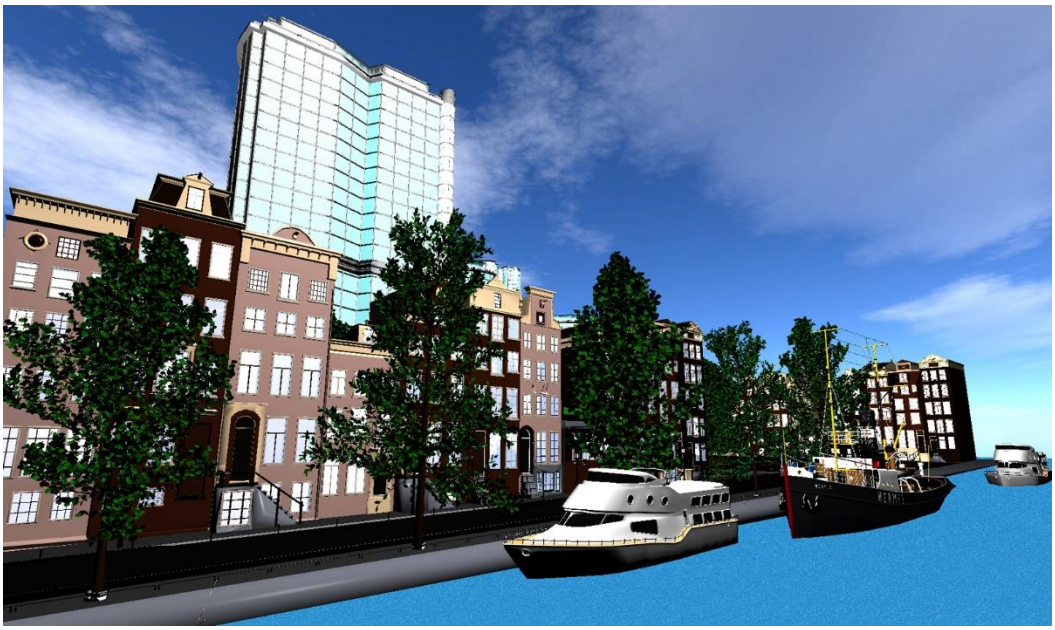


Abbildung 10: Ansicht auf Schiffe und alte Gebäude

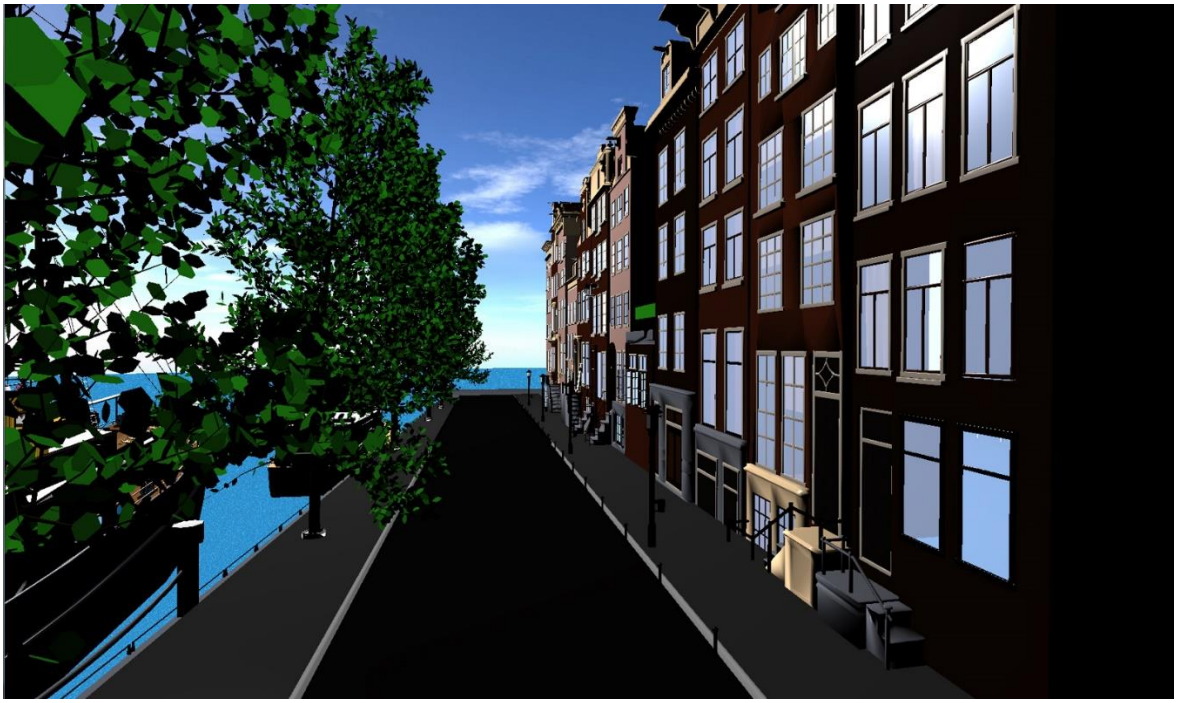


Abbildung 11: Traditionelle Gebäude am Meer

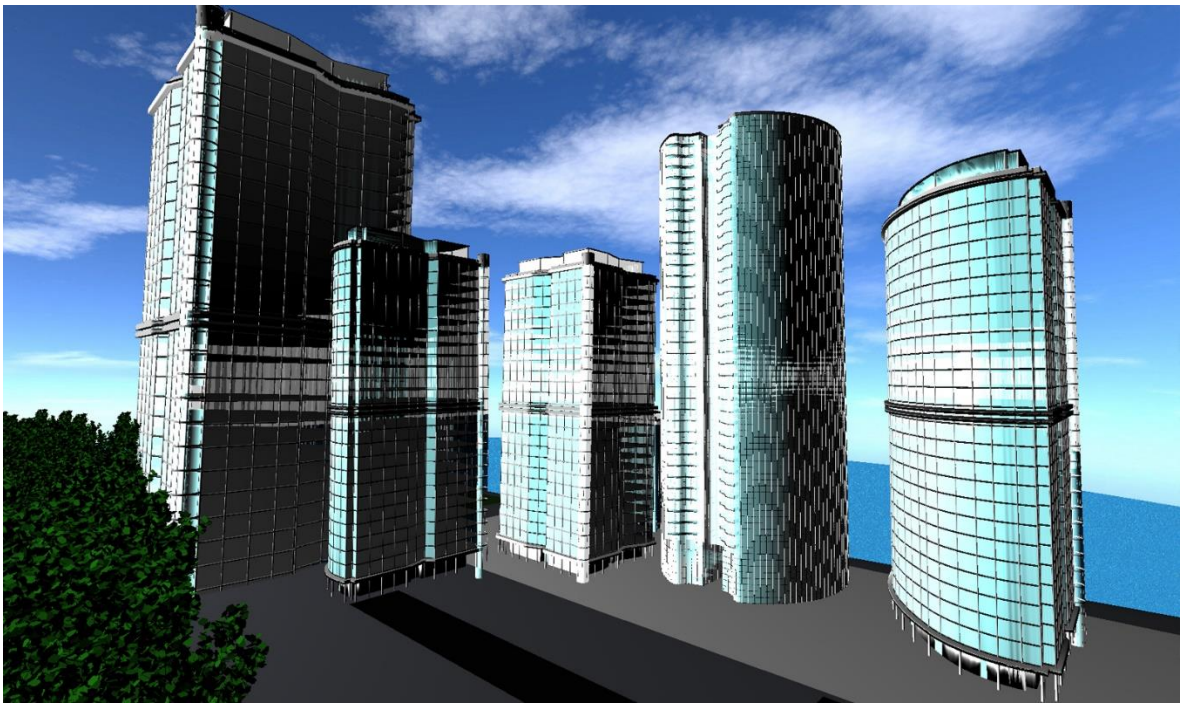


Abbildung 12: Ansicht auf Hochhaus-Komplex

Abbildung 13: Fahrzeuge in der Szene

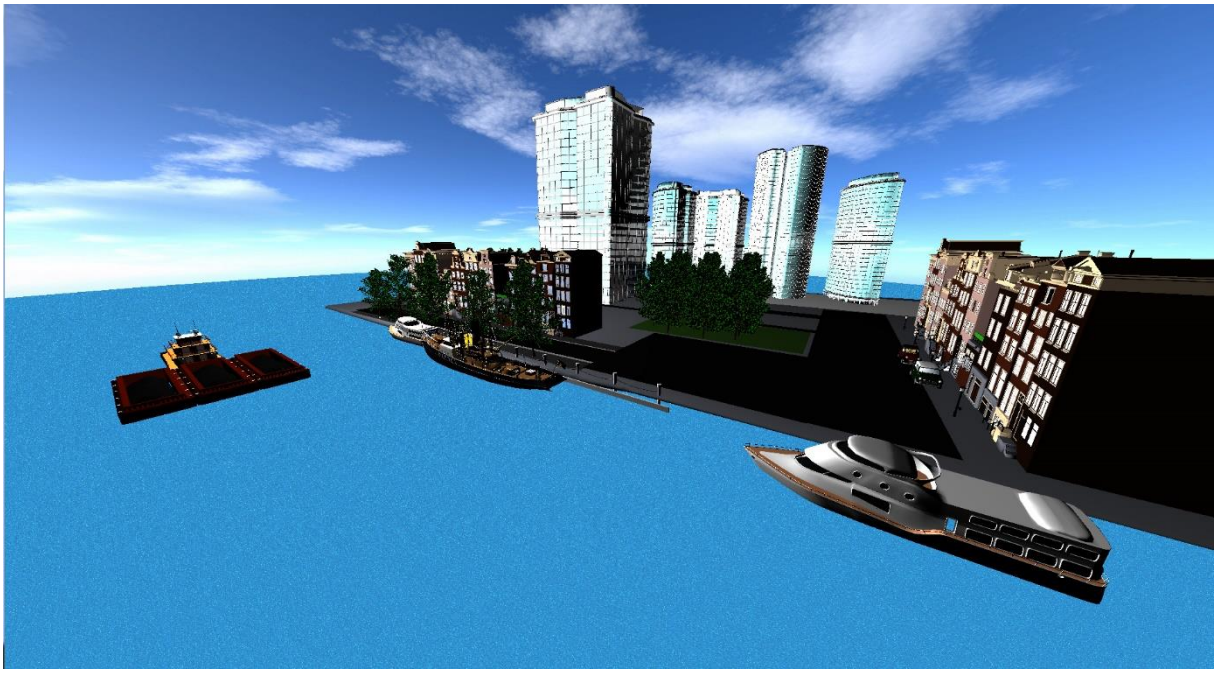


Abbildung 14: Ansicht der gesamten Szene

Vergleichende Ansichten (rechts Bachelorarbeit; links Masterarbeit)

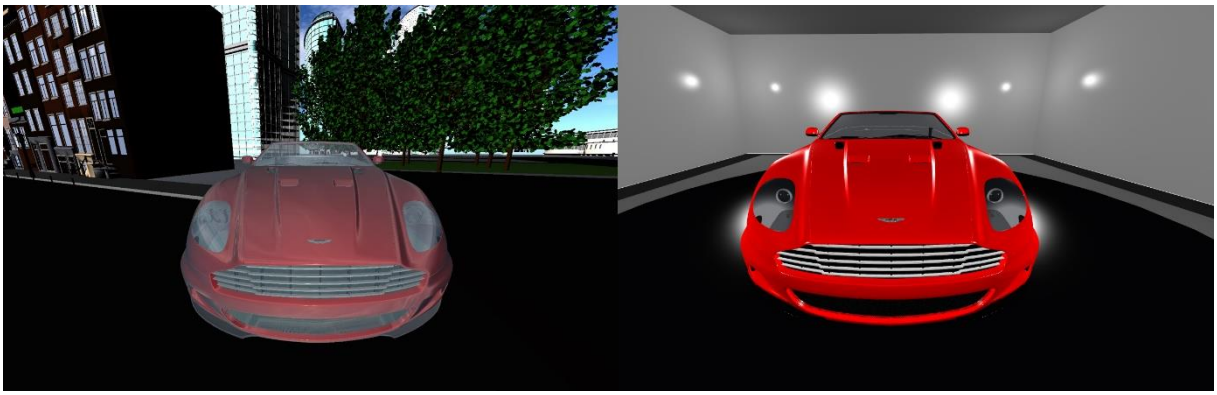


Abbildung 15: Vergleich - Frontansicht

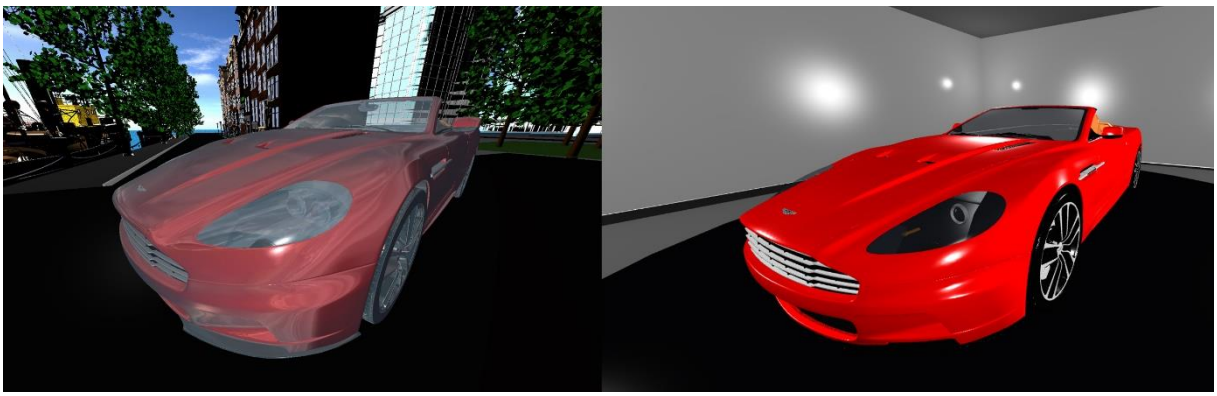


Abbildung 16: Vergleich - Front-Seitenansicht

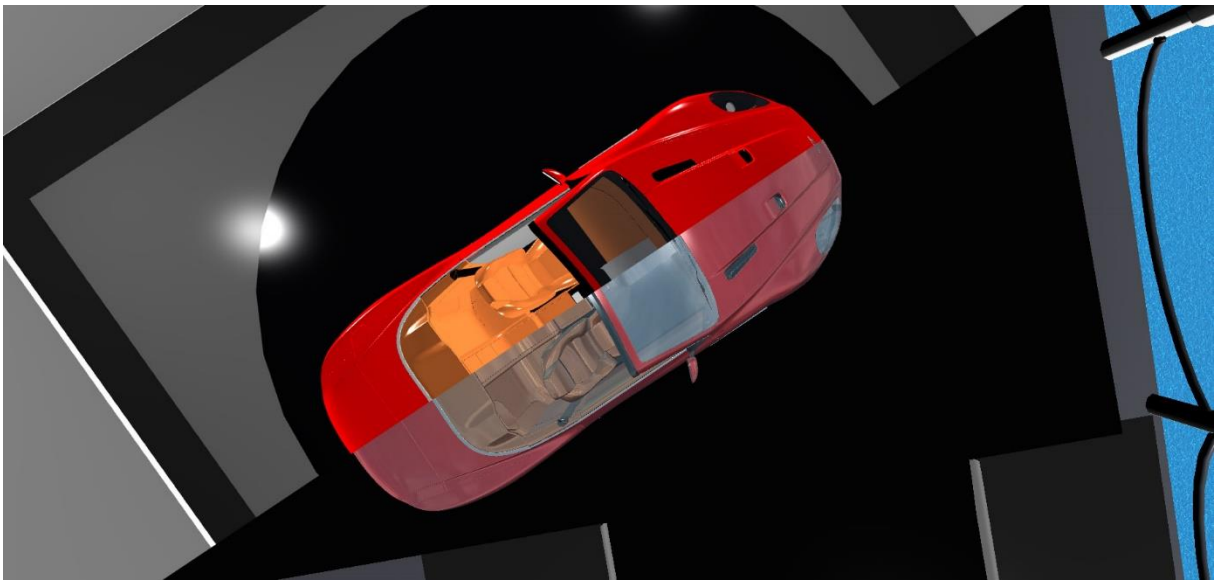


Abbildung 17: Vergleich – Draufsicht

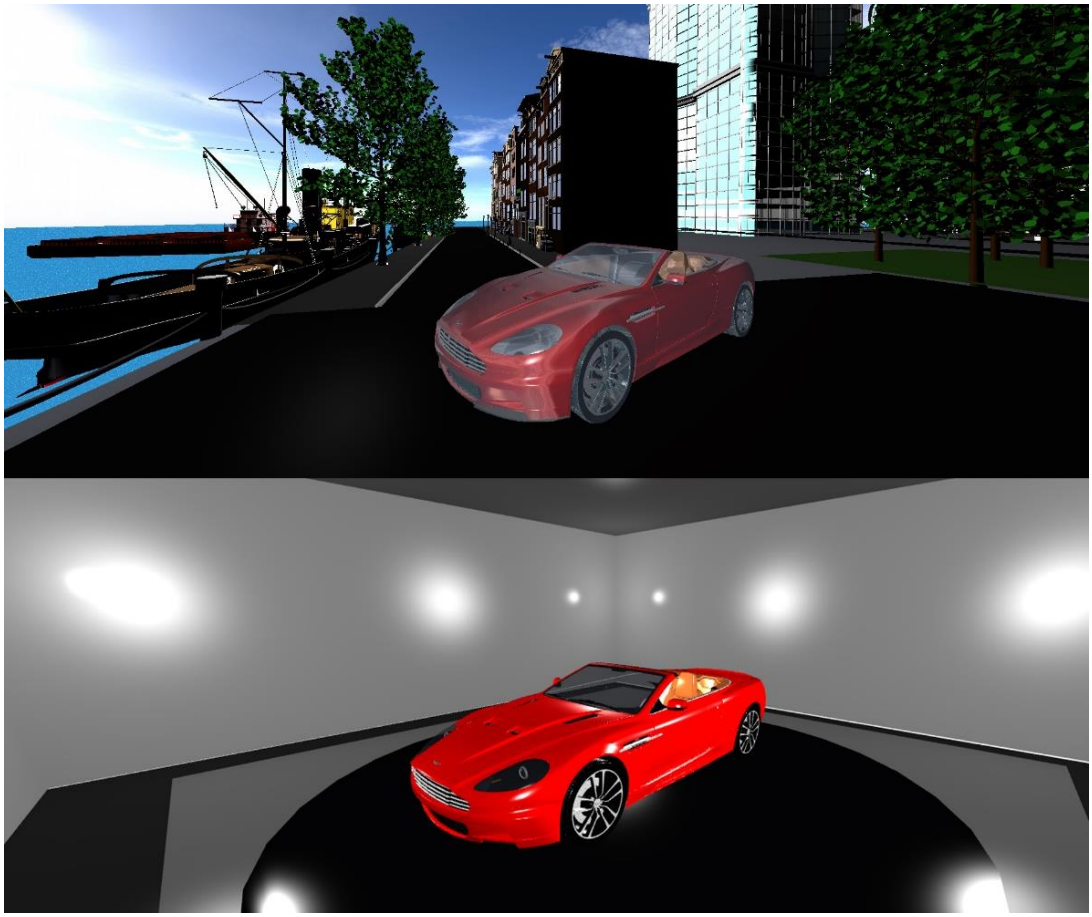


Abbildung 18: Vergleich - Fahrzeug Gesamtansicht

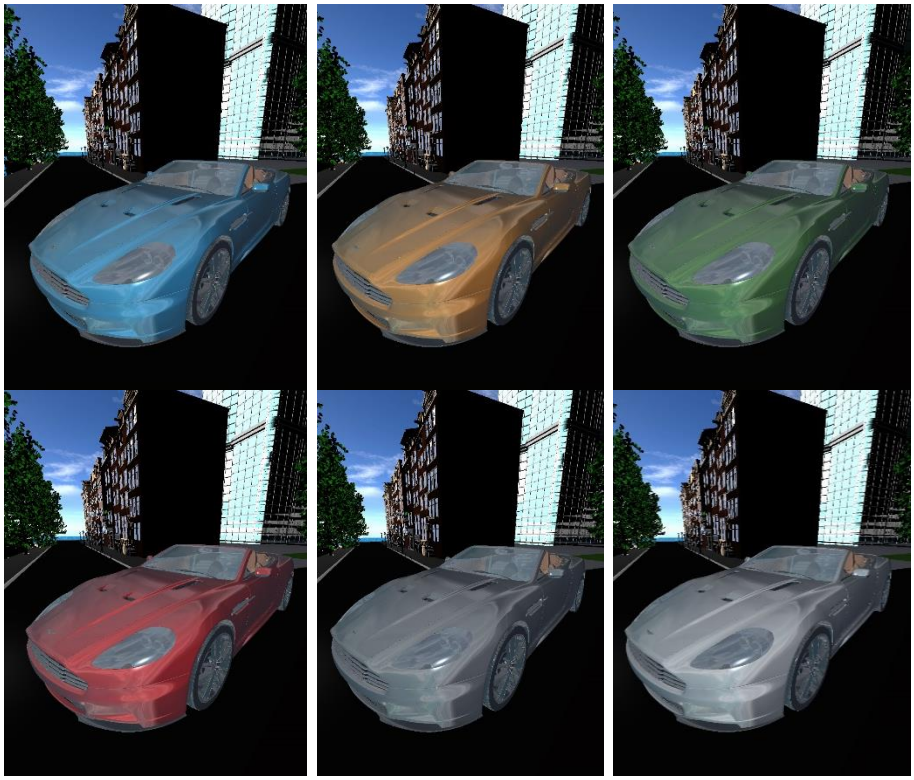


Abbildung 19: Auswahl der Lackfarben

3 Grundlagen

3.1 HTML5

Bei HTML handelt es sich um eine Auszeichnungssprache zur Strukturierung und semantischen Auszeichnung von Inhalten im Internet. Die Entwicklung der 1997 eingeführten Version 4.0 wurde lange Zeit nicht fortgeführt und erst mit HTML5 auf die aktuellen Anforderungen an Web-Anwendungen weiterentwickelt. So wurden hier unter anderem neue Elemente für die Einbettung von Audio- und Videodaten integriert, sowie eine interaktive Zeichenfläche (`canvas`-Element), die es ermöglicht, Bilder und 3D-Inhalte darzustellen. Noch viel essentieller für diese Masterarbeit ist jedoch die Technologie der Websockets, die erstmals in HTML5 ermöglicht wird. Dabei handelt es sich um eine bidirektionale und persistente Verbindung zwischen Browser auf Client-Seite und einem Webserver der ebenfalls Websockets implementiert. Ein detaillierter Einblick in die Thematik der Websockets wird in Kapitel 3.3 gegeben.

3.2 TCP vs. UDP

TCP und UDP sind Protokolle, die im OSI-Schichtmodell auf der IP Ebene aufsetzen und dazu verwendet werden, Daten (auch Pakete genannt) über das Internet zu versenden. Das heißt unabhängig davon ob die Daten über TCP oder UDP versendet werden, ist das Ziel über eine IP Adresse bekannt. Manche Fachliteratur verwendet den Ausdruck TCP/IP was jedoch nichts anderes als „TCP over IP“ bedeutet. TCP und UDP sind nicht die einzigen Protokolle die auf der IP Ebene arbeiten, jedoch die meist verbreitetsten. TCP steht für transmission control protocol und ist ein zuverlässiges, verbindungsorientiertes Protokoll, das Daten als Bytestrom versendet. Verbindungsorientiert bedeutet hier, dass bevor Datenpakete ausgetauscht werden können, eine von beiden Seiten (Client und Server) gesicherte Verbindung hergestellt sein muss. Diese Verbindung arbeitet im Fullduplex-Betrieb, d.h. beide Parteien können unabhängig voneinander Daten senden. TCP baut auf dem IP-Protokoll auf welches paketorientiert arbeitet. Datenpakete können unter Umständen verloren gehen, doppelt beim Empfänger ankommen oder in falscher Reihenfolge. Um diese Unsicherheiten seitens IP zu vermeiden, wurde TCP mit entsprechenden Mechanismen wie Prüfsummen in Datenpaketköpfen und Sequenznummern zur Sicherstellung der Paket-Reihenfolge ausgestattet. Beim Empfänger wird ein Puffer eingerichtet, der die Datenpakete in der richtigen Reihenfolge vorhält, verlorene Datenpakete erneut anfordert und doppelte Datenpakete entsprechend verwirft (Grigorik, 2013).

UDP steht für user datagram protocoll und ist ein nicht-zuverlässiges, verbindungsloses ungesichertes und ungeschütztes Übertragungsprotokoll, das Daten in Form von Paketen (Datagrammen) versendet. Das heißt, dass durch UDP keine Garantie gewährleistet wird, dass versendete Pakete auch ankommen, die Reihenfolge erhalten bleibt, oder Pakete mehrfach beim Empfänger ankommen können. Darüber hinaus finden keine Maßnahmen zur Sicherung der Daten statt (Grigorik, 2013). Dies bedeutet, dass es keine Gewähr bezüglich Verfälschung und Zugänglichkeit für Dritte, für gesendete Daten gibt.

3.3 Websockets

Das Websocket Protokoll ist ein auf TCP basierendes Netzwerkprotokoll (Abts, 2007) und ermöglicht eine bidirektionale Kommunikation zwischen einem Client und einem Webserver. Als Sicherheitsmodell wird das „origin-based security model“ verwendet (Force, 2011), dass jeder Webbrowser implementiert. Zum Aufbau der Verbindung wird, wie bei TCP Verbindungen auch, ein sogenannter „opening handshake“ durchgeführt, der eine sichere Verbindung gewährleistet und nach Eröffnung der TCP Verbindung diese auch aufrecht erhält (Force, 2011). Bei einer reinen HTTP -Verbindung wird jeder HTTP-Request mit einer HTTP-Response abgeschlossen, wodurch ein erneuter Verbindungsaufbau zwischen Client und Server notwendig ist. Darüber hinaus werden bei jedem Request und Response HTTP-Header-Informationen mitgesendet, die das zu sendende Datenpaket vergrößern. Mit dem Websocket Protokoll entfallen einerseits der ständige Neuaufbau der Verbindung, sowie die zusätzlichen Daten durch die Header-Informationen.

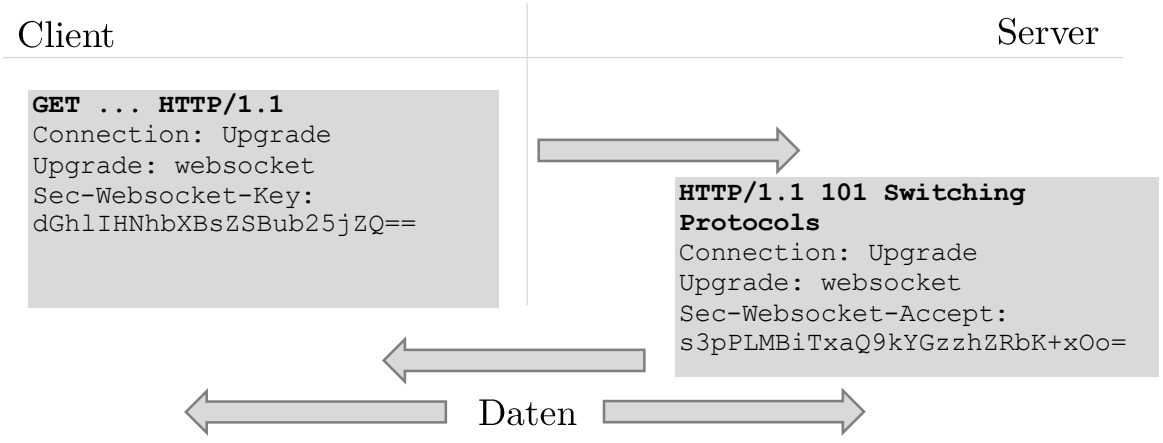


Abbildung 20: Ablauf des Websocket-Protokoll Handshakes (Force, 2011)

Der Verbindungsaufbau wird vom Client begonnen in dem dieser einen sogenannten Protokollwechsel (upgrade) initiiert. Die dabei erzeugte Zeichenkette Sec-Websocket-Key dient zur Überprüfung, ob beide Seiten dem upgrade zustimmen. Im Beispiel von Abbildung 20, wie es dem RFC6455 (Force, 2011) zu entnehmen ist, sendet der Client

den Key „dGhlIHNhbXBsZSBub25jZQ==" an den Server. Auf diesen Protokollwechsel reagiert der Server mit dem Wert `Sec-Websocket-Accept`. Um zu diesem Zustand zu kommen, konkateniert der Server den vom Client erhaltenen Key mit der GUID (Globally Unique Identifier, [RFC4122]) `"258EAFA5-E914-47DA-95CA-C5AB0DC85B11"`, die nicht von Netzwerk-Endpunkten genutzt wird, welche das Websocket Protokoll nicht verstehen (Force, 2011). Auf dieser Zeichenkette wird ein SHA-1 Hash (160 Bits, [FIPS.180-3]) ausgeführt, sowie eine Base64-Codierung bevor das Resultat `"s3pPLMBiTxaQ9kYGzzhZRbK+xOo="` mit dem Status `101 Switching Protocols` an den Client zurückgesandt wird (Force, 2011).
Ab diesem Zeitpunkt steht die Verbindung und Daten können in sogenannten Frames transportiert werden.

Seitdem HTML5 Einzug in moderne Browser erhalten hat, ist auch die relativ einfache Implementierung der Websocket API möglich. Im Folgenden wird beschrieben, welche grundlegenden Bestandteile dazu auf einer Webseite integriert sein müssen.

Soll eine Websocket-Verbindung geöffnet werden, so muss zunächst der Websocket-Konstruktor aufgerufen werden:

```
var connection = new WebSocket('ws://127.0.0.1:12345');
```

In diesem Fall wird eine Verbindung zu einem lokalen Websocket aufgebaut mit IP 127.0.0.1 sowie dem Port 12345. An dieser Verbindung können nun eine Reihe von Events abgefangen werden, die je nach Zustand automatisch angestoßen werden:

```
connection.onopen = function(){
      connection.send('Die Verbindung steht!');
};
connection.onerror = function(error){
      connection.log('WebSocket Error: ' + error);
};
connection.onmessage = function(e){
      connection.send('Server: '+ e.data);
};
```

Ruft ein Client die Webseite auf, so wird die Verbindung angefordert. Bei erfolgreichem `upgrade` auf das Websocket-Protokoll wird die Funktion `onopen` angestoßen und vom Server die Meldeung „`Die Verbindung steht`“ an den Client gesendet. Entstand beim Verbindungsaufbau ein Fehler, übernimmt die `onerror` Funktion automatisch den weiteren Ablauf. Empfängt der Client Daten die der Server gesendet hat, wird die Funktion `onmessage` ausgeführt. Diese gibt über das Event `e`, die Daten aus die der Server gesendet hat (Malte Ubl, 2010).

Mit der `send`-Funktion kann der Client Daten an den Server senden. Seit der aktuellen Spezifikation der Websocket API können auch Binärdaten wie `BLOB`-Objekte und ArrayBuffer Objekte versendet werden. Bei `BLOB`-Objekten (Binary large Objects) handelt es sich in der Regeln um Bilder, Video- und Audiodateien (Malte Ubl, 2010).

3.4 H.264 Codec (Encoding – Decoding)

H.264/AVC (advanced video coding) ist ein Standard, eingeführt von der ITU-T (International Telecommunication Union) sowie der ISO/IEC (International Organisation for Standardisation/International Electrotechnical Commisuion) (Overview of the scalable video coding extrension of the H.264/AVC standard, 2007). Der H.264/AVC Standard wurde erstmals im Jahr 2003 veröffentlicht (Richardson, 2010), wobei seit diesem Zeitpunkt mehrere Berichtigungen und Updates herausgegeben wurden. Der Standard baut auf früheren Standards wie MPEG-2 und MPEG-4 auf und bietet das Potential für eine effizientere und bessere Kompression d.h. komprimierte Videos in höherer Qualität und größere Flexibilität beim Komprimieren, übertragen und Speichern von Videos (Richardson, 2010). Videokompression ist gerade in Bezug auf die Darstellung auf mobilen Endgeräten und begrenzte Bandbreiten zur Datenübertragung ein relevantes Thema, denn Kompression bedeutet Platzeinsparung. Der H.264-Standard vermindert die Menge an Informationen basierend auf Redundanzen zwischen den

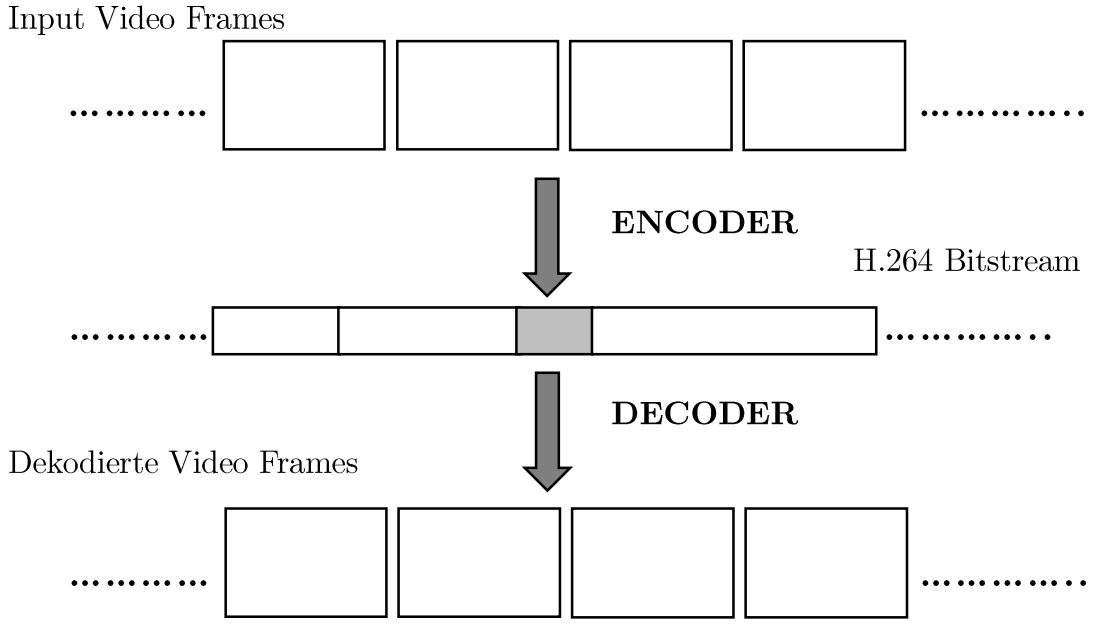


Abbildung 21: Videokodierung

einzelnen Bildern, die zur Wiedergabe eines Videos benötigt werden. Dazu werden ein Encoder sowie dein Decoder benötigt. Wie in Abbildung 21 (Richardson, 2010) zu erkennen ist, wird die Sequenz der originalen Video-Frames in das H.264 Format encodiert. Dabei greift H.264 auf das Keyframe-Konzept zurück, dass im Folgenden näher erläutert wird. In Sequenzen einzelner Video-Frames in denen wenige Veränderungen stattfinden kann mit höherer Qualität encodiert werden, als in solchen, in denen sehr viele Bewegungsabläufe stattfinden (Ozer, 2009). Das ist der Grund, warum das Kompressionsverfahren des H.264-Codecs wie eingangs bereits erwähnt, auf der Ermittlung von Redundanzen zwischen einzelnen Video-Frames arbeitet. Je mehr Redundanz, desto höher die Bildqualität bei jeglicher Bitrate. Um die Redundanzen optimal nutzen zu können, unterscheidet das Keyframe-Konzept drei Arten von Frames - I, P, und B-Frames:

I-Frames: Auch bekannt als „*key frames*“ sind I-Frames, vollkommen unabhängig in ihrer Information von anderen Frames und beziehen sich lediglich auf sich selbst. In Bezug auf den Speicherbedarf sind I-Frames die größten der drei, besitzen somit aber auch die höchste Qualität und sind folglich am wenigsten effizient aus für eine Kompression (Ozer, 2009).

P-Frames: Diese Frames stehen für die sogenannten „*predicted frames*“. Wird ein solcher Frame produziert, kann der Encoder auf I-Frames oder bereits erstellte P-Frames mit redundanter Bildinformation zurückgreifen. P-Frames sind weit aus effizienter als I-Frames, jedoch ineffizienter gegenüber B-Frames (Ozer, 2009).

B-Frames: Im Vergleich zu P-Frames sind B-Frames „*bi-direktional vorhersagend*“. Wie in Abbildung 22 zu sehen ist, bedeutet dies, dass der Encoder beim Erstellen eines B-Frames sowohl vorwärts als auch rückwärts nach redundanter Bildinformationen suchen kann. Das macht B-Frames zum effizientesten aller drei Frame-Typen.

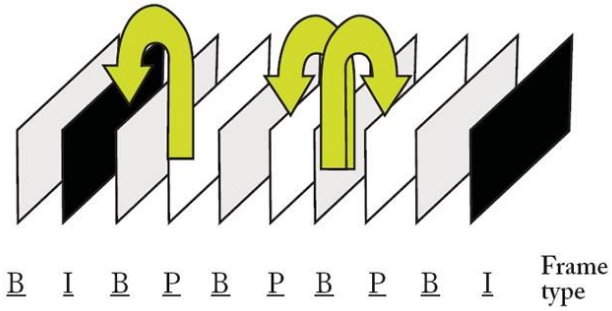


Abbildung 22: I,P und B-Frames in einem H.264-enkodierten Bit-Stream (VIDEOAKTIV, 2013)

Für eine Visualisierung der einzelnen Frames steht die Abbildung 23. Hier ist vereinfacht zu erkennen, welche Bildinformationen als redundant anzusehen sind und welche als I-Frame vorliegen müssen. Beispielsweise ist es erforderlich, dass der erste Frame einer Video-Sequenz ein I-Frame ist, da zu Beginn keine vorhersagbaren Frames existieren können. Je mehr P- und B-Frames vorliegen, umso geringer ist die resultierende Datenrate des Videos (Ozer, 2009).

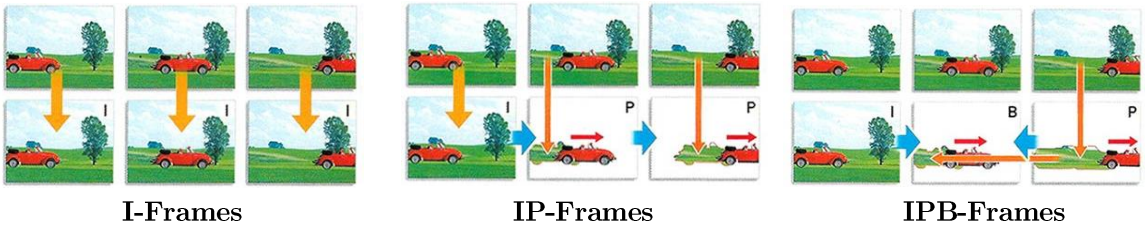
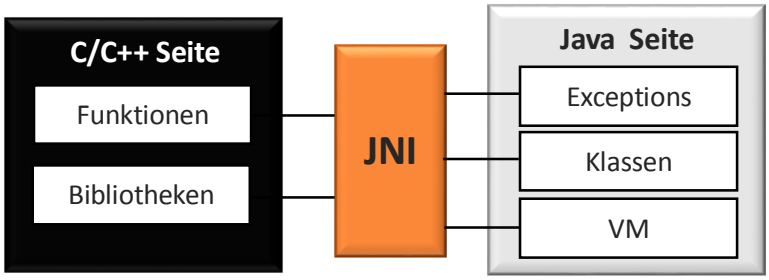


Abbildung 23: Beispiel-Bildsequenz mit I-,P- und B-Frames (VIDEOAKTIV, 2013)

3.5 JNI Bridge

Das Java Native Interface (JNI) ist ein natives Programmier-Interface für die Programmiersprache Java welches fester Bestandteil des JDK (Java Development Kit) ist. Implementiert man Programme mit dem JNI, stellt der Entwickler sicher, dass der Code portabel zwischen allen Betriebssystemen bleibt. Das Java Native Interface erlaubt es Java Code, welcher innerhalb einer Java Virtual Machine (VM) ausgeführt wird, mit anderen Anwendungen und Bibliotheken in Sprachen wie C und C++ geschrieben sind zu operieren. Die verbindende Rolle welche das JNI einnimmt, wird in der nachfolgenden Abbildung schematisch an Hand von Java und C++ visualisiert.



Während es möglich ist, eine Anwendung vollständig in Java zu schreiben, können Situationen entstehen, in denen Java gewissen Ansprüchen einer Anwendung nicht mehr genügt. Solche Ansprüche können unter anderem sein:

- Plattform-abhängige Features, welche von der Anwendung benötigt werden, können nicht durch die standardmäßigen Java Class Bibliotheken abgedeckt werden (Oracle, 2016).
- Eine Bibliothek wurde bereits in einer anderen Programmiersprache implementiert und soll in die bestehende Java Anwendung integriert werden.

Der zweite Punkt spielt insbesondere für diese Arbeit eine große Rolle, da der von AdaptVis bereitgestellte RemoteEncoder in Form einer Windows-DLL

(RE_JNI_Bridge.dll) vorliegt und in C++ geschriebene Funktionen bereit hält. Für diese Fälle wurde das JNI (Java Native Interface) konzipiert. Bei JNI handelt es sich um eine Schnittstelle die den wechselseitigen Zugriff zwischen Java-Anwendungen auf der einen Seite und plattformspezifischen (nativen) Bibliotheken und Anwendungen auf der anderen Seite ermöglicht (Ofterdinger, 2005). Um Zugriff auf die Funktionalitäten der in C++ geschriebenen RE_JNI_Bridge.dll zu erhalten, muss diese aus der Java-Anwendung heraus als Klasse instanziiert werden. (Ofterdinger, 2005)

3.6 OpenGL – Ein Überblick

Im Folgenden wird ein kurzer Überblick über OpenGL (OpenGraphics Library) sowie die OpenGL Shading Language (GLSL) gegeben, da im weiteren Verlauf der Arbeit auf spezielle Befehle eingegangen wird.

Bei OpenGL handelt es sich um eine prozedural arbeitende und hardwarenahe API für 3D-Rastergrafiken die darüber hinaus Plattform-, Betriebssystem- und sprachunabhängig ist. Die Spezifikationen variieren dabei pro Version stark im Funktionsumfang, wobei die Anwendung in dieser Masterarbeit auf der Version 4.0 Core Profile aufbaut. Die Aufgabe von OpenGL besteht hauptsächlich aus der Verwaltung der Graphics-Pipeline, die sich aus freien Stages welche vom Entwickler programmiert werden und festen Stages zusammensetzt.

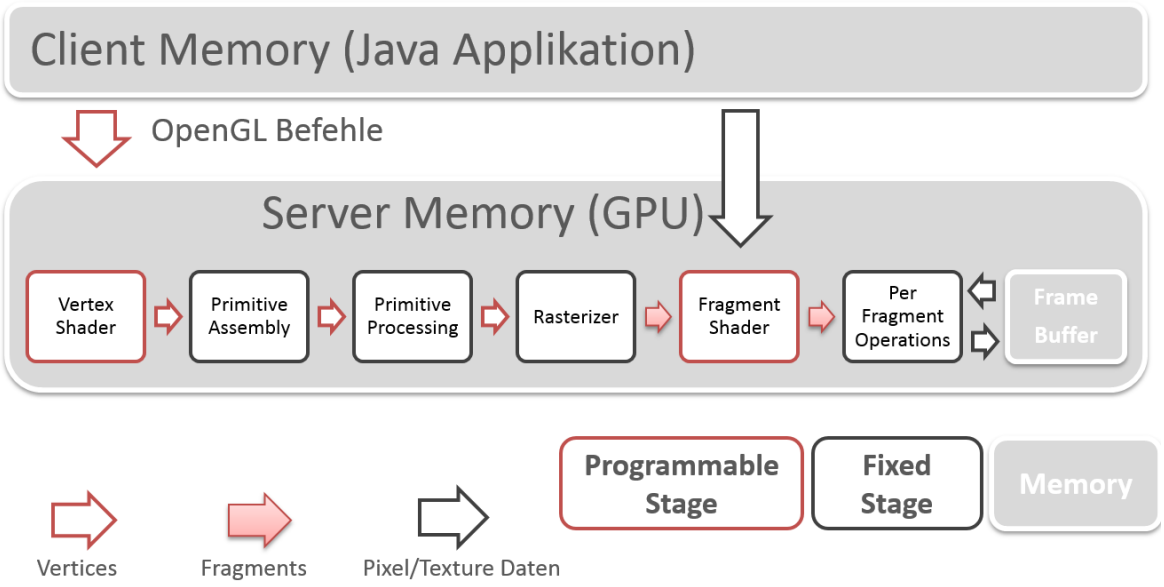


Abbildung 24: OpenGL Graphics Pipeline (Bourdon, 2013)

Die Darstellung der Graphics-Pipeline wie sie in Abbildung 24 gezeigt wird, repräsentiert eine reduzierte Version der eigentlichen Pipeline, zu der seit OpenGL 4.0 noch weit mehr Stages gehören (beispielsweise Geometry Shader und Tesselation Shader). Die ausschließliche Verwendung von Vertex- und Fragment Shader begründet die Abbildung in diesem Zusammenhang.

Des Weiteren erzeugt und übersetzt OpenGL angebundene Shader-Programme und kontrolliert den Datenfluss, welcher in Abbildung 25 in vereinfachter Art und Weise zusammengefasst dargestellt ist.

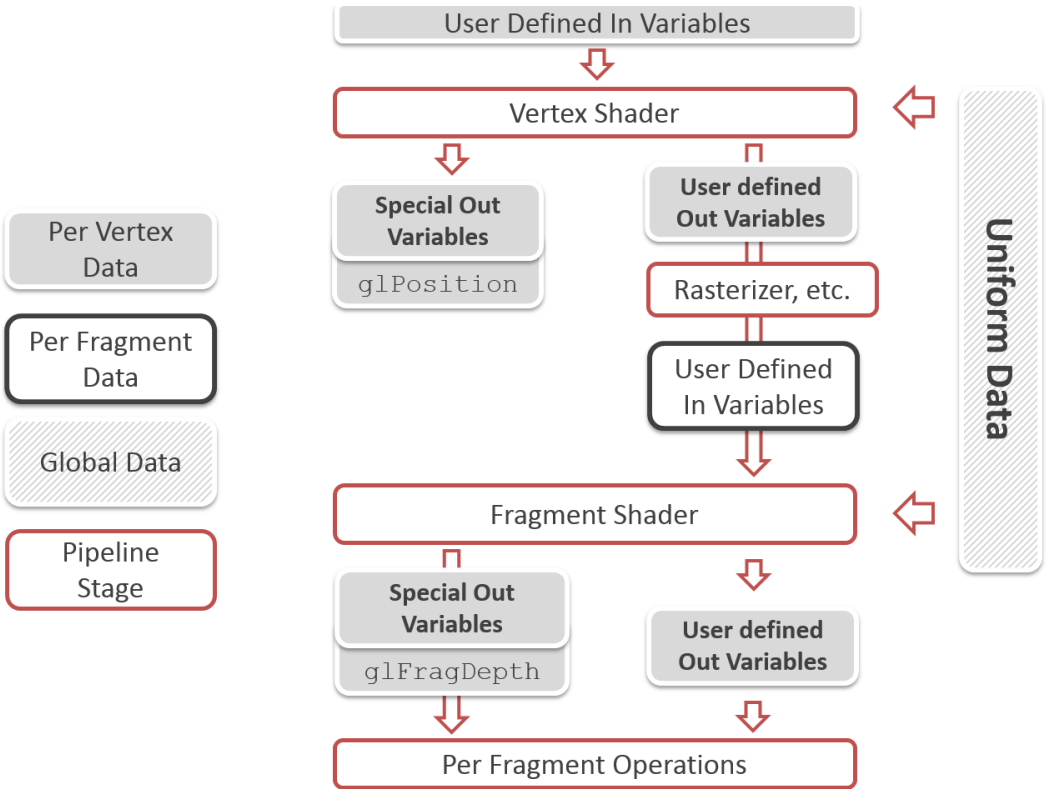


Abbildung 25: Datenfluss einer OpenGL-Anwendung (Bourdon, 2013)

Grundsätzlich wird zwischen drei verschiedenen Arten von Daten unterschieden. Per Vertex Daten sind jene Daten, über die jeder einzelne Vertex verfügt. Das sind in der Regel die Position eines Vertex und seine Normale, unter Umständen aber auch eine Farbe. Per Fragment Daten beziehen sich ausschließlich auf ein Fragment und beschreiben meist dessen Farb- und Tiefeninformationen. Globale Daten werden im Programmverlauf erzeugt und als sogenannte Uniform-Daten in Vertex- und Fragment-Shader übergeben. Im Rahmen der OpenGL-Graphics-Pipeline werden insbesondere model-, view- und projection-Matrix als Uniform-Variablen an die Shader übergeben, da diese sich zur Laufzeit meist abhängig von Nutzer-Eingaben dynamisch verändern.

Grundsätzlich basiert OpenGL auf dem Client-Server-Schema, wobei Client und Server in diesem Fall in einem Endgerät vereint sind (siehe Abbildung 26). Die Anwendung repräsentiert in diesem Szenario den Client, welche durch das Absetzen von OpenGL-Befehlen mit der GPU als Server kommuniziert. Die OpenGL Implementation der Grafikkarte setzt diese um und rendert die Daten.

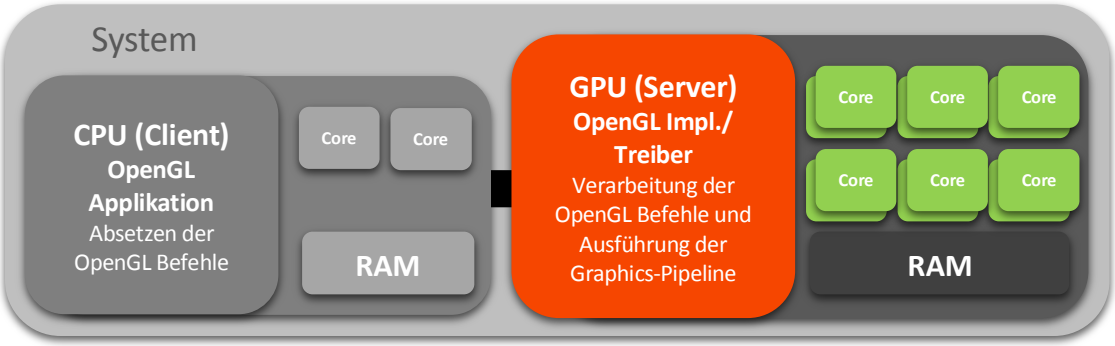


Abbildung 26: OpenGL als Client-Server Modell

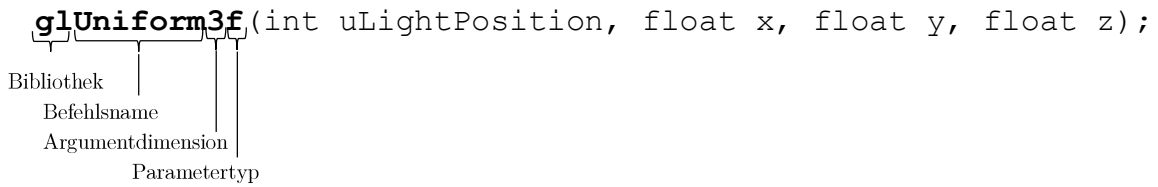
Dieses Schema wird im Rahmen dieser Masterarbeit bewusst für die Zwecke des Remote-Renderings aufgebrochen. Der Client kommuniziert über eine Webschnittstelle mit dem Server auf dem sowohl die Anwendung, als auch das Rendering abläuft.

OpenGL-Befehle sind in ihrer Struktur oft sehr ähnlich aufgebaut und können wie folgt in allgemeiner Weise beschrieben werden:

```
gl<Root Command>[arg count][arg type][v]
```

Syntaktisch beginnen Befehle immer mit "`gl`" was den Hinweis auf die OpenGL-Bibliothek gibt. Das „`root command`" stellt den Befehlsnamen dar, wobei „`arg count`" die Anzahl an Übergabeparameter und „`arg type`" den Datentyp der Parameter beschreibt. Der Zusatz „`v`" gibt lediglich darüber Auskunft ob ein „vektorieller Datentyp" (Arrays oder Pointer) zu Grunde liegt und taucht nur bei speziellen OpenGL-Befehlen auf.

Diese allgemeine Beschreibung wird an einem spezielleren Befehl nochmals verdeutlicht:



Der Befehl `glUniform3f` verändert den Wert der Uniform-Variablen, welche durch den Integer-Parameter `uLightPosition` sowie die drei Übergabeparameter vom Datentyp `float` spezifiziert wird. Uniform-Variablen stellen zugleich eine der wichtigsten Komponenten in OpenGL dar. Sie sind das Bindeglied zwischen der Hauptanwendung und den Shadern. Wird eine Uniform-Variable im Rahmen des Programmablaufs gesetzt bzw. verändert, so wird dieser auch beim nächsten Aufruf des Shaders der sie verwendet berücksichtigt. In der Regel sind Transformations-Matrizen, Materialeigenschaften und Daten für Licht prädestiniert für Uniform-Daten. Neben GL-Befehlen ist eine weitere Eigenschaft der OpenGL-API die Verwendung von speziellen Konstanten, was insbesondere den Charakter als zustandsorientierte API unterstreicht. Konstanten wie beispielsweise `GL_DEPTH_TEST`, `GL_CULL_FACE` oder `GL_PRIMITIVE_RESTART` werden über die Befehle `glEnable(...)` bzw. `glDisable(...)` gesetzt bzw. zurückgesetzt. Das Zustandsprinzip behält den Status nach der Aktivierung so lange bestehen, bis eine Deaktivierung erfolgt.

3.7 GLSL – Ein Überblick

Bei GLSL handelt es sich um eine Shadersprache in der die programmierbaren Teile der OpenGL Graphics-Pipeline individuell programmiert werden können. Die in GLSL geschriebenen Programme werden Shader genannt und werden auf Vertices (Vertex Shader) und Fragments (Fragment Shader) angewendet. Spezielle Shader können auch ganze Geometrien erzeugen und werden Geometry Shader genannt. Diese Art wird in dieser Masterarbeit jedoch nicht verwendet. GLSL basiert auf ANSI C und wurde mit vektoriellen und Matrix-Datentypen sowie einigen Built-In Funktionen erweitert (Programming, 2014). In den Shadern wird unabhängig von den Datentypen zwischen drei Variablen unterschieden – `in`-, `out`- und `uniform`-Variablen. Der folgende Quellcode-Auschnitt beschreibt den grundlegenden Aufbau eines Shaders.

```
#version 330 core
uniform     <varType><varName>;      // Uniform Variablen
in          <varType><varName>;      // In Variablen
out         <varType><varName>;      // Out Variablen

void main(){
   ...                               // Main Funktion des Shaders
}
```

Zunächst wird die Version des Shaders gesetzt von der unter anderem der Funktionsumfang abhängig ist. Darauf folgt die Deklaration der Variablen so wie die `main`-Methode in der die Logik des Programms abgelegt ist. Neben Rechen- und Logikoperatoren sind in GLSL auch Flusskontrollen (if, if-else und switch-case) auch einige mathematische und vektorielle Funktionen implementiert. Beispiele dafür sind

```
float   sin(float radians)      Trigonometrische Funktionen
float   pow(float x, float y)   Berechnet x^y
float   sqrt(float x)           Berechnet die Wurzel aus x
float   length(vec4 x)          Betrag eines Vektors
float   dot(vec4 a, vec4 b)     Skalarprodukt aus a und b
vec4    normalize(vec4 x)       Liefert normalisierten Vektor
vec3    cross(vec3 a, vec3 b)   Kreuzprodukt a x b (nur für vec3)
mat4    inverse(mat4 a)         Inverse Matrix, ab GLSL 1.50 / GL 3.2
```

3.8 Environment-Mapping

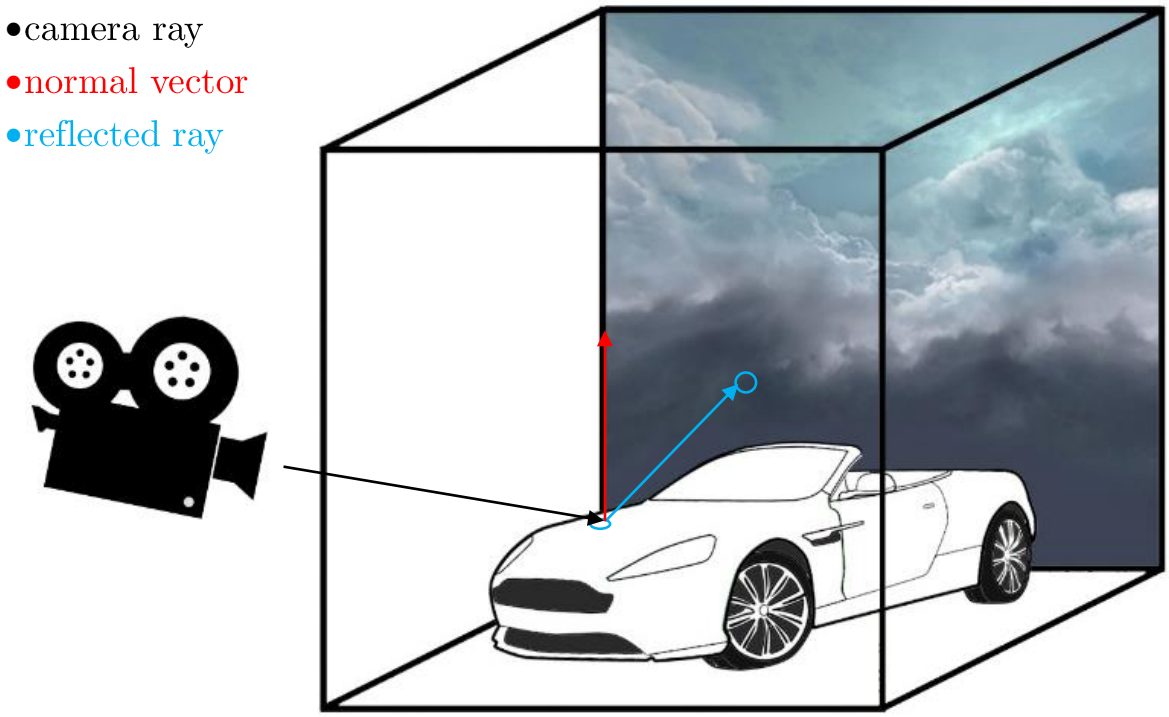


Abbildung 27: Cubemaps für Environment Mapping

Betrachtet man ein Objekt, dass seine Umgebung sehr stark reflektiert (Spiegel, Chrom, Lack, etc.) so gewinnt man den Eindruck nicht das Objekt selbst zu sehen, sondern ausschließlich die reflektierte Umgebung. Würde man einen Strahl von der Augen-Position des Betrachters zum Objekt verfolgen, so sieht man an diesem Punkt auf dem Objekt die Umgebung, wie sie an der Stelle des reflektierten Strahls aussieht. Abbildung 27 verdeutlicht diese Situation etwas genauer und führt so das Konzept des Environment Mapping mittels Cubemaps ein. Ein Strahl aus der Kamera, bzw. des Augenpunktes des Betrachters auf ein Objekt in der Szene wird entlang des Normalenvektors in diesem Punkt reflektiert und mit der umgebenden Skybox geschnitten um die Farbe des getroffenen Pixels zu ermitteln. Die Skybox ist ein großer Environment-Mapping Würfel, der die gesamte Szene umgibt und ist mit einer Textur (Cubemap) belegt.

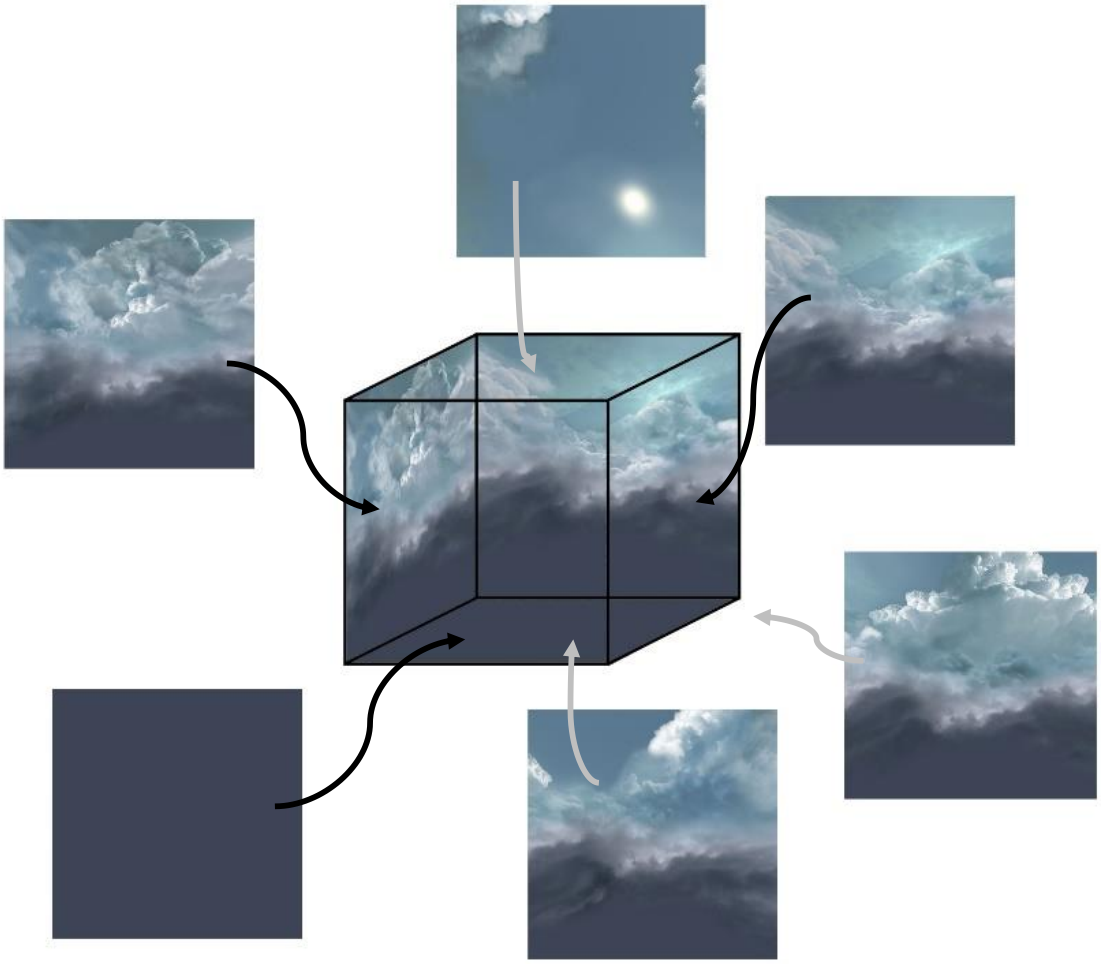


Abbildung 28: Cubemap Beispiel

Unterstützt werden Cubemaps auf NVIDIA Grafikkarten seit dem Modell GeForce 256 (NVIDIA Corporation, 1999). Während gewöhnliche Texturen in der Regel aus einer Bilddatei bestehen, enthält eine Cubemap sechs quadratische Texturen, die flüssig ineinander übergehen. Die in Abbildung 28 verwendete Textur ist ein Teil der in dieser Arbeit unter anderem verwendeten Cubemap-Textur und findet sich in Abbildung 28 als vollständige Darstellung wieder. In der Regel bildet die Mitte jeder einzelnen Textur die Horizontlinie, wobei der untere Teil in den Boden der Szene übergeht, der obere Teil in den Himmel der Szene. Die Shader für die Integration von Environment-Mapping sind relativ einfach zu erstellen und benötigen nur wenige Komponenten:

Vertex Shader

Parameter	*Variablen Name*	*Type*
View Projection Matrix	`viewProj`	`mat4`
Model Matrix	`modelMatrix`	`mat4`
Model Vertex Position	`vs_in_pos`	`vec3`
Model Vertex Normal	`vs_in_normal`	`vec3`
World Vertex Position	`world` `viewDirection`	`vec4` `vec3`
World Normal Position	`normalWC` `normalDirection`	`vec3` `vec3`

```
uniform mat4 viewProj;          // View-Projection Matrix Kamera
uniform mat4 modelMatrix;       // Matrix der Modelling-Transformations

in vec3 vs_in_pos;              // Geometrie-Vertices
in vec3 vs_in_normal;           // Geometrie-Normalen

out vec3 viewDirection;         // für EV-Mapping: Blickrichtung
out vec3 normalDirection;       // für EV-Mapping: Normalenrichtung


void main()
{
   // Transfer der Modell-Koordinaten in Welt-Koordinaten
   vec4 world = modelMatrix * vec4(vs_in_pos, 1.0);
   viewDirection = world.xyz;

   // Berechnung der Normalen im Welt-Koordinatensystem
   vec3 normalWC = (modelMatrix * vec4(vs_in_normal, 0.0)).xyz;
   normalDirection = normalWC;

   // Ausgabe
   gl_Position = viewProj * world;

}
```

Fragment Shader

Parameter	*Variablen Name*	*Type*
CubeMap als uniform	`cubeMap`	`samplerCube`
Blickrichtung auf Objekt	`viewDirection`	`vec3`
Normalen-Richtung	`normalDirection`	`vec3`
Reflektionsvektor der reflektierten Blickrichtung	`reflectedDirection`	`vec3`

```
in vec3 viewDirection;          // Blickrichtung
in vec3 normalDirection;        // Normalenrichtung
uniform samplerCube cubeMap;    // Uniform der Cube-Map-Texturen

out vec4 finalcolor;            // Ausgabe Farbwert des Fragments

void main()
{
   // Reflektionsvektor der Blickrichtung an der Normalen in einem Punkt
   vec3 reflectedDirection = reflect(viewDirection, normalize(normalDirection));

   // Farbausgabe, durch Berechnung der Texturkoordinate mit Reflektionsvektor
   finalcolor = textureCube(cubeMap, vec3(reflectedDirection.x,
                                          -reflectedDirection.y,
                                           reflectedDirection.z));

 }
```

4 Gesamtüberblick der Anwendung

In den folgenden Kapiteln wird ein Gesamtüberblick (siehe Abbildung 29) über die Anwendung gegeben und sowohl auf die Server- wie auch die Client-Seite expliziter eingegangen.

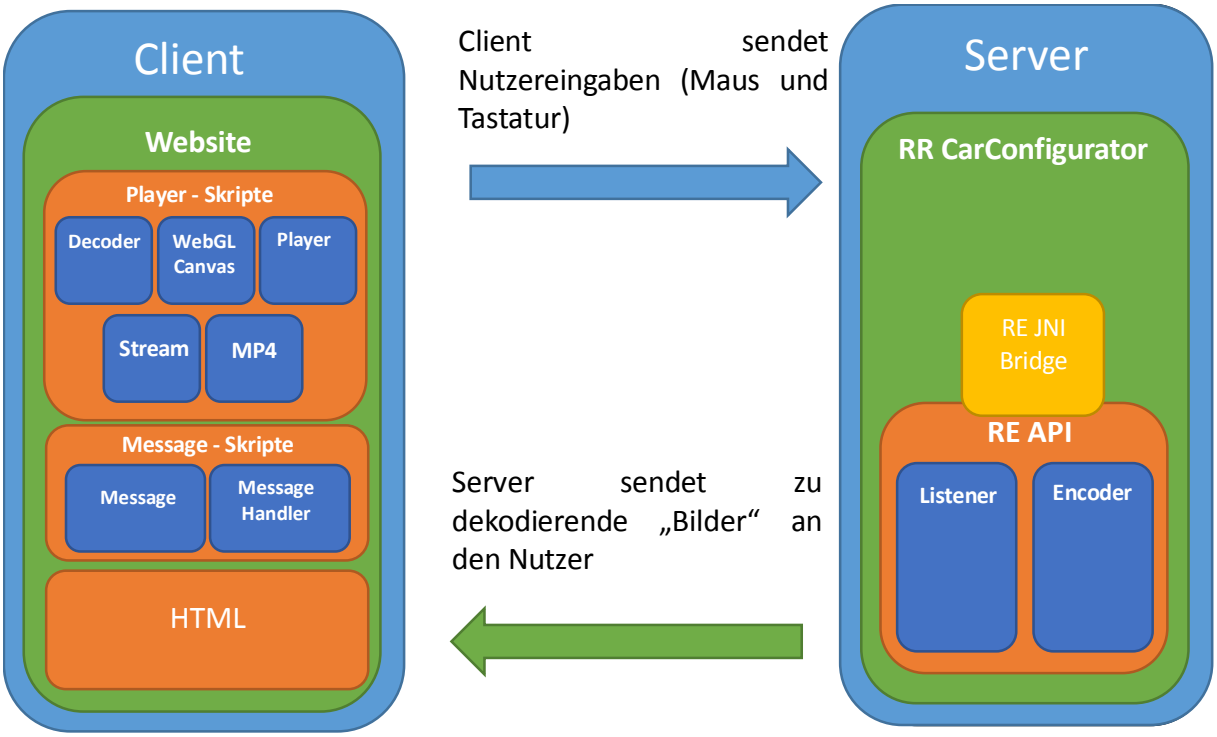
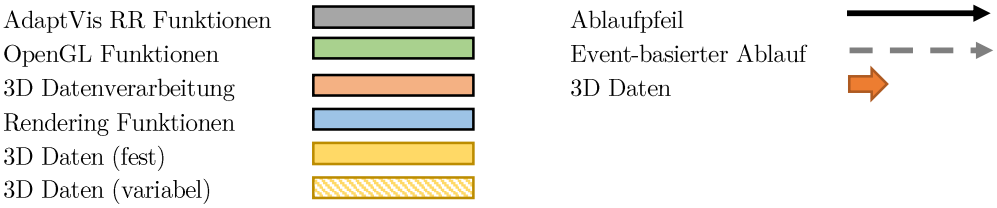


Abbildung 29: Gesamtüberblick der Anwendung

Entwickelt wurde die Anwendung in der Entwicklungsumgebung Eclipse und Sublime Text-Editor. Die 3D-Modelle wurden in Blender modelliert und modifiziert.

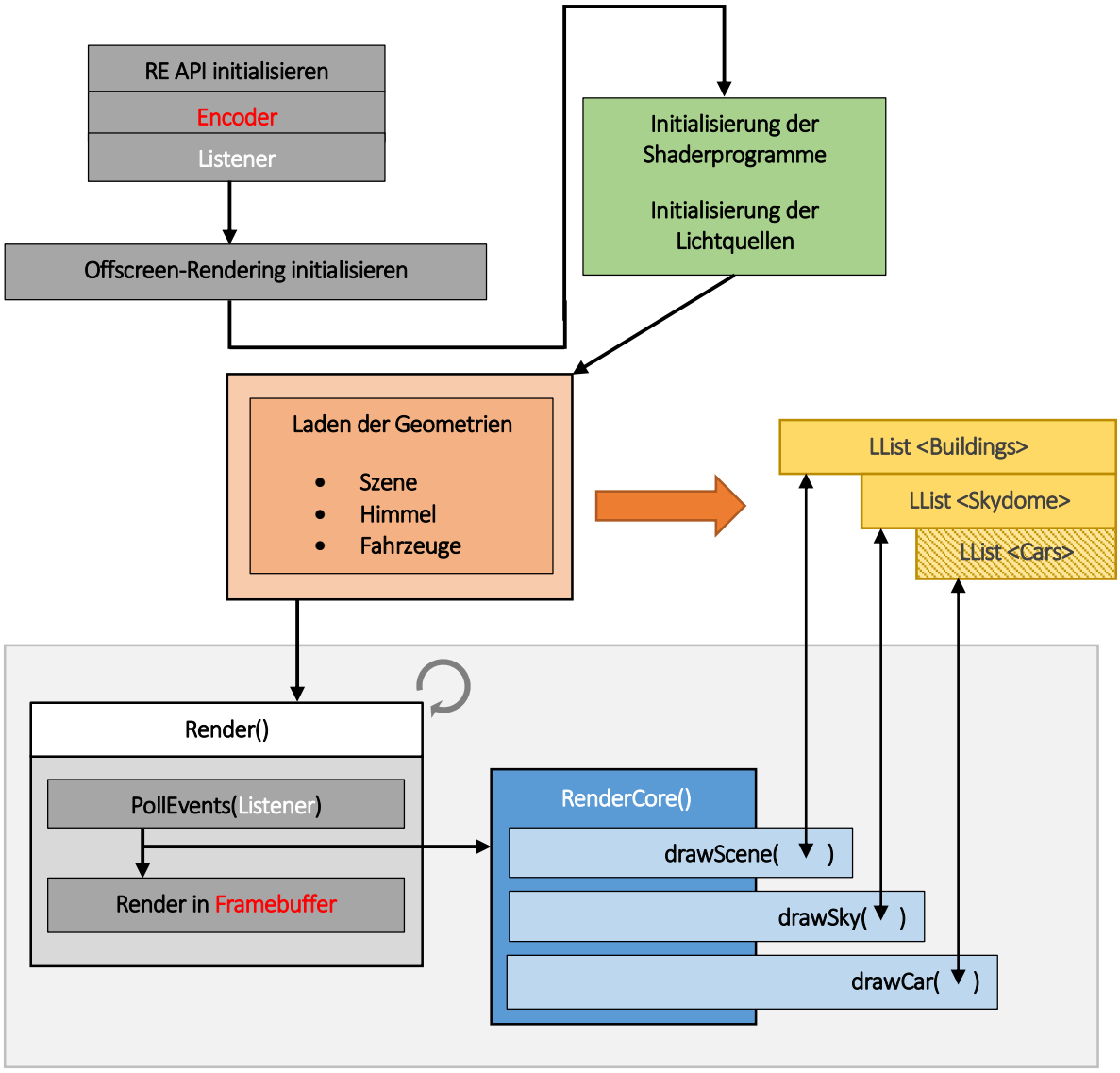
4.1 Server Seite

Im folgenden Kapitel liegt der Fokus auf einem detaillierten Einblick in den Aufbau und die Implementation der Server Seite der Anwendung. Dabei wird zunächst eine Gesamtübersicht gegeben, wobei ablaufbedingt zusammenhängende Module farblich unterschieden werden. Die folgende Legende verdeutlicht das Ablaufdiagramm:



4.2 RR CarConfigurator - Überblick

Implementierung der Anwendung (Autokonfigurator).



4.3 Remote Encoder API

In diesem Abschnitt wird näher auf die eingebettete Remote Encoder API der AdaptVis GmbH eingegangen und ist aufgeteilt in die Vorstellung der Klassen repräsentiert als UML-Diagramme, sowie einer Ablaufbeschreibung des Initialisierungsprozesses.

4.3.1 Klassen

Die Klasse der RE API verwaltet alle Funktionalitäten um den Remote Renderer und hält weitere Klassen für die Beschreibung des Encoders sowie für den Event-Listener bereit, welcher die Interaktionen des Clients annimmt und weiter verarbeitet (siehe Abbildung 30).

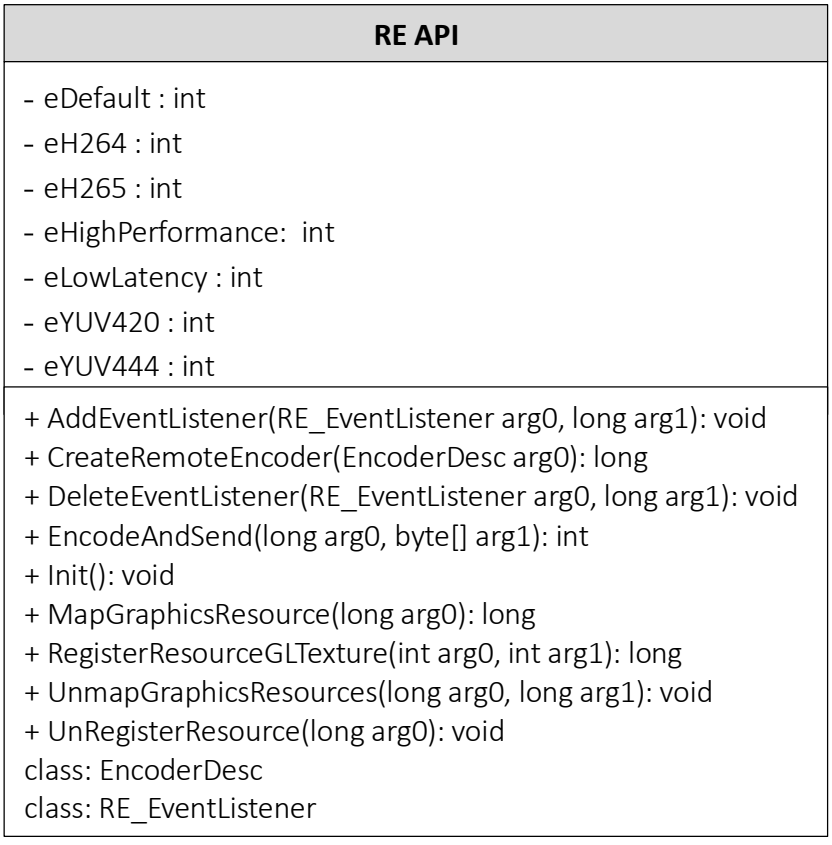


Abbildung 30: UML-Diagramm RE-API

Damit ein reibungsloser Initialisierungsvorgang gewährleistet werden kann ist es nötig, die JNI-Bridge in Form der zur Verfügung gestellten DLL zu importieren. Diese stellt das Bindeglied zwischen der Hauptanwendung (in Java) und der in C++ implementierten API dar.

Das in Abbildung 31 dargestellte UML-Diagramm beschreibt den Event-Listener der Remote Encoder API. In der aktuellen Version kann dieser grundlegende Maus- und Tastatur-Events erkennen, sowie einfache „character“ als Nachricht empfangen und senden.

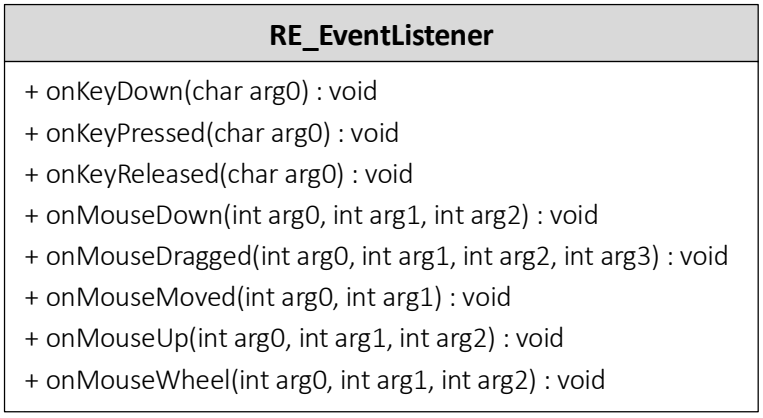


Abbildung 31: UML-Diagramm RE-EventListener

4.3.2 Dokumentation der Methoden und Konstanten

RE-API

Parameter

int	eDefault	
int	eH264	*Initialisierungs-Parameter für H.264*
int	eH265	*Initialisierungs-Parameter für H.265*
int	eHighPerformance	
int	eLowLatency	*Initialisierungs-Parameter für das Bildformat YUV420*
int	eYUV420	*Initialisierungs-Parameter für das Bildformat YUV444*
int	eYUV444	

Methoden

`AddEventListener(RE_EventListener arg0, long arg1): void`

Die Methode `AddEventlisener` wird statisch aus der `RE_API` aufgerufen und erwartet als Übergabeparameter zum einen eine Instanz eines `RE_EventListeners`, sowie als weiteren Übergabeparameter einen `RemoteEncoder` welcher als ID vom Datentyp `long` vorliegt.

`CreateRemoteEncoder(EncoderDesc arg0): long`

Diese Methode legt eine neue Instanz eines `RemoteEncoders` an. Sie erwartet als Übergabeparameter eine Instanz vom Datentyp `EncoderDesc` welche im weiteren Programmverlauf mit allen nötigen Parametern gesetzt wird (siehe EncoderDesc im weiteren Verlauf).

`DeleteEventListener(RE_EventListener arg0, long arg1): void`

Die Methode vollzieht das Löschen eines Eventlisteners und benötigt dazu als Übergabeparameter entsprechend den `EventListener` sowie den `Encoder` an welchen dieser gebunden ist.

`EncodeAndSendSurface(long arg0, byte[] arg1): int`

Mittels dieser Methode wird der Enkodier- und Sendevorgang eingeleitet. Dazu wird zunächst der entsprechende Encoder übergeben, sowie der „Pointer“ vom Datentyp `long` welcher die gemappte Resource (hier die Textur mit dem Inhalt des Framebuffers) enthält.

`Init(): void`

Initialisierung der API

`MapGraphicsResource(long arg0): long`

Dieser Methode wird eine Grafik-Resource, hier in Form einer GL-Texture übergeben welche entsprechend gemapped wird. Der simulierte Pointer mit Datentyp `long` kann entsprechend für weitere Funktionen die mit der Resource arbeiten verwendet werden.

`RegisterResourceGLTexture(int arg0, int arg1): long`

Zunächst muss dieser Methode eine GL-Texture übergeben werden, sowie deren Art. Dies kann beispielsweise `GL_TEXTURE_2D` sein. Zurück gegeben wird dann eine Grafik-Resource, die unter anderem mittels CUDA weiterverarbeitet werden kann

`UnmapGraphicsResources(long arg0, long arg1): void`

Unmapping der Grafik-Resource, sodass eine neuer Inhalt an die Resource gemapped werden kann

`UnRegisterResource(long arg0): void`

Rücksetzen der Registrierung der Resource, sodass ein neuer Register-Vorgang begonnen werden kann

`class: EncoderDesc`

`class: RE_EventListener`

EncoderDesc

Parameter

int	codec	Parameter für den Video-Codec (hier H.264, H.265)
int	format	*Initialisierungs-Parameter für H.264 oder H.265*
int	fps	*Anzahl der Frames pro Sekunde*
int	h	*Höhe des zu enkodierenden Videos in Pixel*
int	w	*Breite des zu enkodierenden Videos in Pixel*
int	maxBitrate	*Initialisierungs-Parameter für das Bildformat YUV420*
int	port	*Initialisierungs-Parameter für das Bildformat YUV444*
int	profile	*Profil der Enkodierung (HighPerformance, LowLatency)*
int	sendBufferSize	
boolean	useNVENC	Nutzung der NVENC Bibliothek
booean	useOpenCL	Nutzung von OpenCL
boolean	useWebSocket	Nutzung von WebSockets
int	varBitrate	Parameter für Bitrate

RE_EventListener

Methoden

```
onKeyDown(char arg0) : void
onKeyPressed(char arg0) : void
onKeyReleased(char arg0) : void
onMouseDown(int arg0, int arg1, int arg2) : void
onMouseDragged(int arg0, int arg1, int arg2, int arg3) : void
onMouseMoved(int arg0, int arg1) : void
onMouseUp(int arg0, int arg1, int arg2) : void
onMouseWheel(int arg0, int arg1, int arg2) : void
```

4.3.3 Abläufe

Die ersten Schritte nach Ausführung der Anwendung beinhalten zunächst das Erstellen einer neuen Instanz und Initialisieren des Remote Encoders. Dazu ist das Initialisieren der folgenden Attribute (siehe auch Seite 46 – Encoder Description) erforderlich:

```
RE_API.Init();                          // Initialisierung der RE-API

boolean IHaveNVIDIA = true;             // Server: Nvidia GPU?

// Setzen der Encoder-Parameter
RE_API.EncoderDesc desc = new RE_API.EncoderDesc();

desc.w = init.width;                            // Höhe Ausgabefenster in Pixel
desc.h = init.height;                           // Breite Ausgabefenster in Pixel
desc.fps = 30;                                  // Angestrebte FPS Zahl
desc.useNVENC = IHaveNVIDIA;                    // Nvidia GPU?
desc.useOpenCL = false;                         // OpenCL nötig in der Anwendung?
desc.varBitrate = false;                        // false wenn keine Bitrate-Limit
desc.useWebSocket = true;                       // Websockets?
desc.codec = RE_API.eH264;                      // Codec zur Enkodierung
desc.format = RE_API.eYUV420;                   // Farbformat des Streams
desc.port = 12345;                              // Port
desc.profile = RE_API.eHighPerformance;         // H264 Performance Modus
desc.sendBufferSize = -1;                       // Limitierung der Buffersize
desc.maxBitrate = 5000000;                      // Maximale Bitrate

encoder = RE_API.CreateRemoteEncoder(desc);
```

Im Rahmen der Anwendungen werden für die Auflösung der angezeigten Bilder 720p bzw. 1080p verwendet, aus denen sich die entsprechenden Angaben zu Höhe und Breite ergeben. Zur Enkodierung wird hier NVENC verwendet. NVENC steht seit der aktuellen Generation von NVIDIA GPUs zur Verfügung, d.h. jene die auf der Pascal, Kepler oder Maxwell Architektur basieren. Vor NVENC basierte die Hardware-Enkodierung von NVIDIA auf CUDA, welche sowohl die CPU als auch die GPU zur Video-Enkodierung nutzte und somit entsprechend Rechenleistung von beiden Komponenten beanspruchte (Okunev, 2014). NVENC hingegen nutzt einen speziell dafür vorgesehenen H.264 Enkodierungs-Chip womit die meiste Rechenleistung der GPU für andere Zwecke zur Verfügung steht. Es ist anzumerken, dass Grafikkarten in Bezug auf die simultanen Enkodierungs-Vorgänge limitiert sind. Nach der NVENC Spezifikation sind GeForce Karten auf 2 Streams und Quadro Karten auf 6 Streams limitiert (Okunev, 2014).

Im Anschluss an die Erzeugung des Encoders, wird der Listener implementiert der auf die vom Client initiierten Maus- und Tastatur-Events reagiert und diese passend an die Hauptanwendung delegiert:

```
RE_EventListener listener = new RE_EventListener() {

 public void onMouseWheel(int arg0, int arg1, int arg2) {
 }

 public void onMouseUp(int arg0, int arg1, int arg2) {
 }

 public void onMouseMoved(int arg0, int arg1) {
 }

 public void onMouseDragged(int arg0, int arg1, int arg2, int arg3, int arg4) {
 }

 public void onMouseDown(int arg0, int arg1, int arg2) {
 }

 public void onKeyReleased(char arg0) {
 }

 public void onKeyPressed(char arg0) {
 }

 public void onKeyDown(char arg0) {
 }

};

RE_API.AddEventListener(listener, encoder);
```

Zum Abschluss wird dem Encoder der zuvor erzeugte Event-Listener angehängt, womit die wesentlichen Bestandteile der Remote-Encoder API erfüllt sind. Damit Bilder versendet werden können, müssen Bilder erstellt werden. Zu diesem Zweck wird vor dem Rendering-Prozess eine Textur `color_tex` erstellt, deren Inhalt später das jeweils

```
color_tex = glGenTextures();
glBindTexture(GL_TEXTURE_2D, color_tex);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, w, h, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                                                          (ByteBuffer)null);
```

Um Offscreen-Rendering realisieren zu können, wird das Konzept der Framebuffer-Objekte benötigt. Diese werden ähnlich wie eine Textur erzeugt und müssen sofort gebunden werden, damit sich alles weitere auf dieses Objekt bezieht. Zum Framebuffer-Objekt wird jedoch noch ein Renderbuffer-Objekt als Depthbuffer benötigt, da der Framebuffer kein zugehöriges Texturformat besitzt. An dieser Stelle ist darauf zu achten, dass sowohl für die Textur als auch der Renderbuffer gleich viel Speicherplatz (entsprechend `width` · `height` des Ausgabefensters) angelegt wird.

```
fb = glGenFramebuffers();
glBindFramebuffer(GL_FRAMEBUFFER, fb);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
                                                                color_tex, 0);

depth_buffer = glGenRenderbuffers();
glBindRenderbuffer(GL_RENDERBUFFER, depth_buffer);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT24, w, h);

glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                                            GL_RENDERBUFFER, depth_buffer);

resource = RE_API.RegisterResourceGLTexture(color_tex, GL_TEXTURE_2D)
```

Im letzten Schritt wird der Depthbuffer dem Framebuffer-Objekt hinzugefügt.

4.4 3D Modelle

Alle Objekte der Szene wurden mit der 3D-Modellierungs-Software Blender erstellt und für die weitere Verarbeitung in das textbasierte OBJ-Format exportiert.

Das Vorgehen gleicht dem, welches bereits in meiner Bachelorarbeit zur Anwendung gekommen ist (Bourdon, 2013). Im Folgenden wird ein Überblick über die Eckdaten der Modelle gegeben um diese einordnen zu können:

Objekt		**Geometrie**		**Speicherbedarf (in MB)**		
Name	*Typ*	*Vertices*	*Faces*	*OBJ*	*JSON*	*# JSON*
Stadtszene	`Building`	3.208.098	2.268.962	406	293	2928
Aston Martin	`Car`	445.076	642.565	74,3	22,7	163
Audi R8	`Car`	997.761	906.004	165	56,4	182
VW Up	`Car`	3.980.103	2.620.010	380	148	396

(**Anmerkung**: Das „#“-Symbol wird im Folgenden für „Anzahl“ verwendet.)

Die verwendeten 3D-Modelle lagen im Rahmen dieser Arbeit komponentenweise vor. Das heißt, dass jede Fahrzeugkomponente (beispielsweise Fronstpoiler, Heckspoiler, Sitze, Glasscheiben, etc.) als unabhängiges 3D-Objekte existiert mit zugewiesenen Materialdaten (diffuse, ambiente und spekulare Farbe, Transparenz, etc.). An dieser Stelle konnten Daten durch Gruppierung von 3D-Objekten in Blender eingespart werden, da der verwendete OBJ/JSON-Parser für jedes in Blender vorliegende 3D-Objekt eine JSON-Datei anlegt, inklusive der Materialdaten.

Die Gruppierung der 3D-Objekte fand somit basierend auf den Materialdaten statt, sodass Objekte mit denselben Materialien entsprechend zu einem Objekt zusammengefasst wurden. Abbildung 32 verdeutlicht dies am Beispiel eines Fahrzeug-Modells. Sämtliche Karosserieteile mit gleicher Materialeigenschaft wurden entsprechend zu einer Geometrie zusammengefasst.

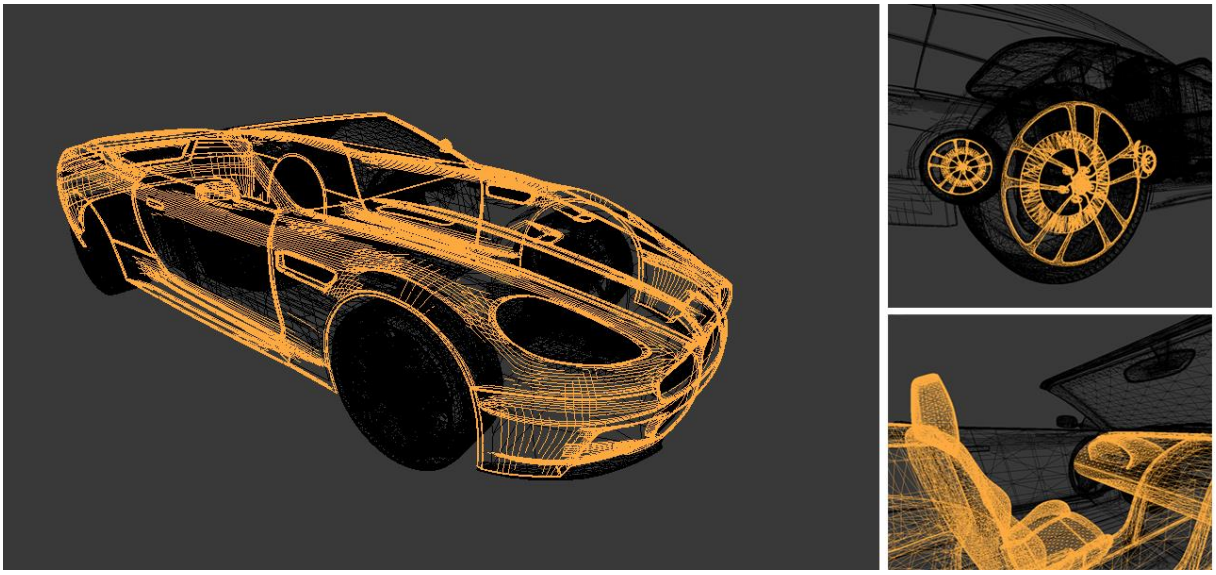


Abbildung 32 : Gruppierte 3D-Fahrzeugkomponenten

Die nachfolgende Tabelle zeigt den, durch die Zusammenfassung reduzierten Bedarf an JSON-Dateien:

Objekt		**Speicherbedarf (in MB)**			
Name	*Typ*	*OBJ*	*JSON*	*# JSON*	***Einsparung***
Stadtszene	`Building`	251	176	1757	**40 %**
Aston Martin	`Car`	72,3	20,7	32	**81 %**
Audi R8	`Car`	176	58	39	**79 %**
VW Up	`Car`	380	153	68	**83 %**

In diesem Zusammenhang fällt auf, dass bei der Stadtszene im Vergleich zu den Fahrzeugen weitaus weniger JSON-Dateien eingespart werden konnten. Dies lag insbesondere daran, dass viele Häuser der 3D-Szene mehrere Materialien besitzen und es so zu Komplikationen beim Gruppieren der Geometrien gab.

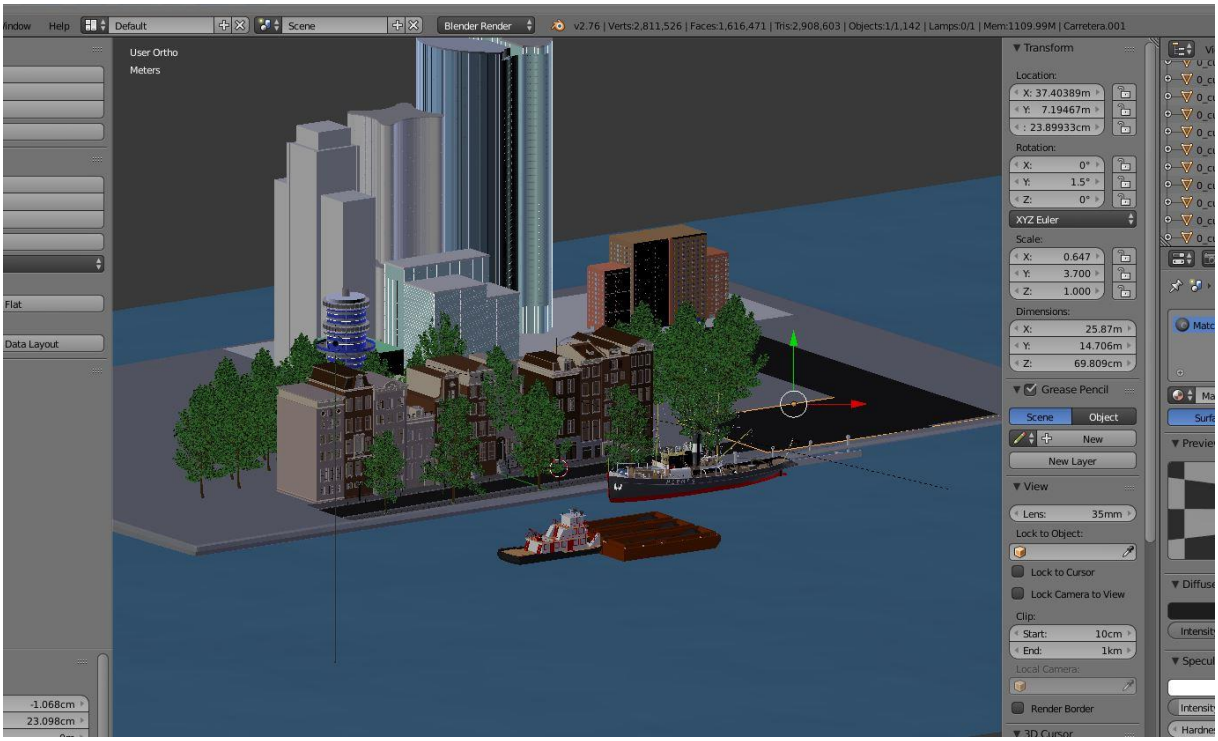


Abbildung 33: Screenshot Blender

Abbildung 33 zeigt einen fortgeschrittenen Zwischenstand der Stadt-Szene im Visualisierungsbereich des 3D-Modellierungsprogramms Blender. Bei den verwendeten 3D-Modellen handelt es sich um freie Daten, die unter der *Creative Commons* lizensiert sind. Die beteiligten Urheber sind am Ende der Arbeit entsprechend genannt.

4.4.1 Klassen

Bevor auf den Ablaufprozess des Ladens von 3D-Modellen eingegangen wird, bietet dieses Kapitel einen kurzen Überblick über die essentiellen Klassen. Dabei liegt der Fokus zum einen auf der Klasse „Part“ sowie der Klasse „Car“ als allgemeiner Repräsentant für alle 3D-Modelle.

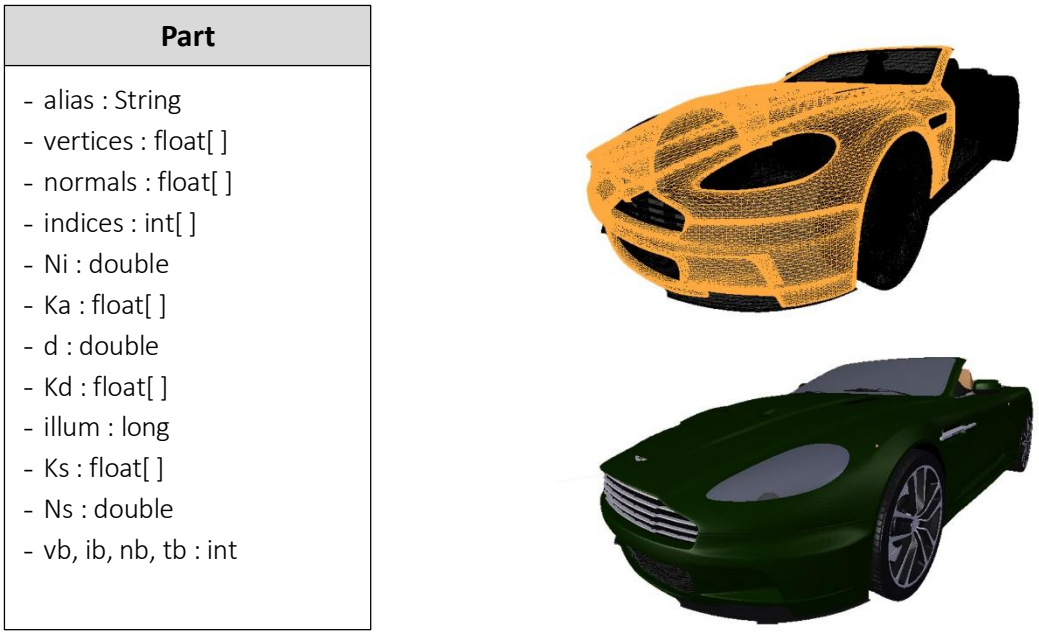


Abbildung 34: UML-Klassendiagramm für Buildings und Part

Alle 3D-Modelle werden, wie eingangs in Kapitel 4.4. beschreiben wurde, in Komponenten unterteilt welche als JSON-Datei vorliegt. Die Struktur dieser Datei ist äquivalent zur Klasse „Part“ (siehe Abbildung 34), sodass ein Anlegen einer Geometrie als Objekt vereinfacht wird. Der „alias“ gibt den eindeutigen Namen der 3D-Komponente an. Die eigentlichen Geometriedaten liegen in den drei Arrays für Vertices, Normalen und Indices. Für die in der MTL-Datei mitgelieferten Materialeigenschaften eines 3D-Modells werden weitere Variablen vorgehalten, wie der Anteil der diffusen Farbe (Kd), ambienten Farbe (Ka), spekularen Farbe (Ks) oder der Transparenzgrad (d). Da sich 3D-Objekte vom Typ „Gebäude“ analog – bis auf geringfügige Abweichung – darstellen lassen, werden die wesentlichen Methoden, Parameter und Eigenschaften an Hand der

Klasse „Car“ vorgestellt. Diese lässt sich über das UML-Diagramm in Abbildung 35 zusammengefasst darstellen:

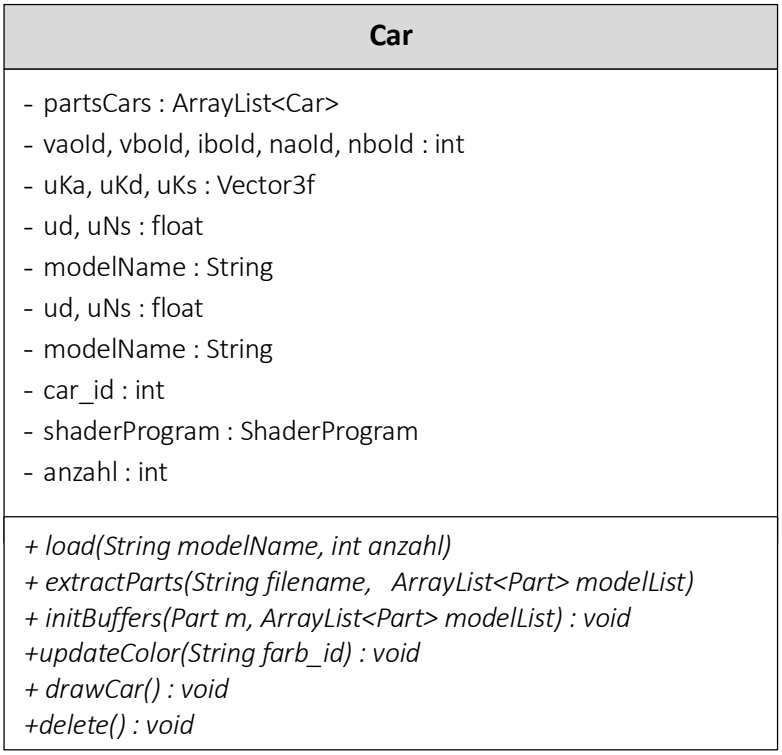


Abbildung 35: UML-Klassendiagramm für Buildings und Part

4.4.2 Abläufe

Für die verschiedenen 3D Objekte welche in der Szene platziert sind existieren drei verschiedene Klassen – Building, Cars und Skydome. Diese sind sehr ähnlich aufgebaut und unterstützen die Modularisierung der Anwendung. Exemplarisch am Generieren eines Objekts vom Typ Fahrzeug soll hier der Ablauf geschildert werden, der sich beim Laden von Objekten der Umgebung (Häuser, Bäume, etc.) oder der Himmelskugel analog beschreiben lässt. Zunächst findet der Aufruf der Methode `loadCars(String fahrzeug)`

statt, welche hier exemplarisch mit dem Fahrzeug-Modell `aston_martin` durchgeführt wird und einer gegebenen Anzahl von 163 Fahrzeug-Komponenten:

```
loadCars("aston_martin");
```

Im Rahmen dieser Funktion wird zunächst ein neues Objekt vom Typ `Cars` angelegt, wobei die Variable `modelname` der Name des Verzeichnisses ist, in welchem die JSON-Dateien der einzelnen Fahrzeugkomponenten liegen. Auf eine genauere Erläuterung des Aufbaus der JSON-Dateien für Fahrzeugkomponenten wird nicht näher eingegangen und auf den Grundlagen-Teil der Bachelorarbeit verwiesen (Bourdon, 2013). Im letzten Parameter wird das jeweilige Shader-Programm angegeben, mit welchem das Fahrzeug gerendert wird. Die Angabe eines speziellen Shader-Programms ist für die Fahrzeug-Modelle essentiell, da hier das Environment-Mapping implementiert ist und lediglich auf Auto-Geometrien angewendet wird. Die Variable `anzahl_parts` bezieht sich auf die Anzahl der JSON-Dateien, welche im Verzeichnis des Fahrzeugs liegen. Diese Zahl wird durch eine entsprechende Methode ermittelt. Im Hauptprogramm wird ein Objekt vom Typ `Car` wie folgt angelegt:

```
aston_martin_db9 = new Cars(modelname, carShaderProg);
carList.add(aston_martin_db9);
aston_martin_db9.load(modelname, anzahl_parts);
```

Nachdem ein Fahrzeug vollständig geladen wurde, liegen alle Geometrien des Fahrzeugs in einer am Objekt gespeicherten Liste vor, welche über eine klasseninterne `draw`-Funktion gerendert werden können. Diese Methode wird für alle Fahrzeuge aufgerufen,

```
glUseProgram(carShaderProg.getId());
updateTransformation(carShaderProg);
for (Cars c : carList) {
      c.drawCar();
}
```

Die Organisation aller Fahrzeuge in dieser Liste hat den Vorteil, dass die Daten zu jederzeit vollständig vorhanden sind. Das heißt auch nach dem Löschen eines Fahrzeugs aus der Szene, ist das Objekt noch vorhanden und muss lediglich erneut in die Liste eingefügt werden. Im Hinblick auf eine spätere Nutzung im Web können so lästige Wartezeiten, welche die Fahrzeuge zum Initialisieren und Laden benötigen, minimiert werden.

5 Client Seite

5.1 Aufbau

Abbildung 36: Web-Version des Autokonfigurators

In Abbildung 36 wird die Webversion des Automobilkonfigurators präsentiert. Dieser orientiert sich im groben Aufbau und Funktionsumfang den herkömmlichen Konfiguratoren. Zunächst ersetzen noch Richtungspfeile und ein Button für eine 360°-Drehung die Möglichkeit, auf Touch-Events von mobilen Endgeräten wie Smartphones und Tablets zu reagieren. Darüber hinaus existieren drei Button für die Auswahl der jeweiligen Fahrzeugmodelle sowie eine Farbpalette für entsprechende Änderungen an der Lackfarbe. Inwieweit der Funktionsumfang denkbar ausgeweitet werden kann, sei an dieser Stelle auf das ausblickende Kapitel 8 hingewiesen.

5.2 H.264 Decoder

Der in dieser Arbeit verwendete H.264 Decoder basiert in seiner Implementation auf WebGL und JavaScript und wurde von der AdaptVis GmbH bereitgestellt. Auf weitere Details bezüglich Implementation und Abläufe wird an dieser Stelle nicht eingegangen, da es sich hierbei um das Produkt der AdaptVis GmbH handelt und es somit als Blackbox fungiert.

6 Leistungsdiagnose

Im Rahmen der Leistungsdiagnose wird der Fokus der Messungen insbesondere auf der serverseitigen Enkodierungszeit sowie der client-seitigen Dekodierungszeit liegen. Darüber hinaus wird unter Einbezug der entsprechenden Auflösung des Client-Endgeräts der Datendurchsatz pro Sekunde analysiert (Kilobyte pro Sekunde), um Richtwerte für die minimalen Anforderungen der Datenrate der Internetverbindung auf Server- und Clientseite zu ermitteln. Dabei werden die folgenden Mess-Szenarien beobachtet und analysiert.

Mess-Szenario 1 (ruhige Szene)

Im ersten Szenario wird die Kamera für 10 Sekunden in der Ausgangsposition bleiben (siehe Abbildung 37), sodass der Datendurchsatz bei ruhiger Szene gemessen wird. Hier wird ein geringes Datenaufkommen erwartet, da H.264 insbesondere bei Standbildern eine hochgradige Kompressionseffizienz erreicht und den Datendurchsatz reduziert.

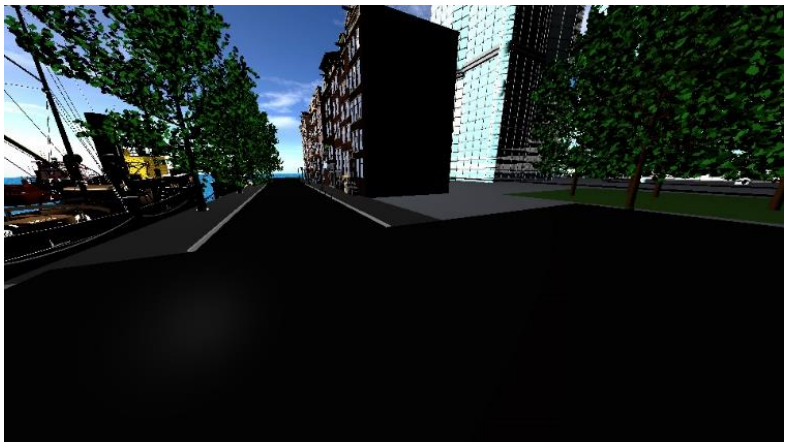


Abbildung 37 : Standbild der Szene

Messwerte

- *Zeit* in Sekunden
- *Frames per Second*
- *Datendurchsatz* in Kilobyte (kB) pro Sekunde
- *Gesamtes Datenvolumen* in MegaByte (MB)

Mess-Szenario 2 (bewegte Szene)

Im zweiten Szenario steht die Kamera zunächst für 5 Sekunden still und fokussiert den Bereich auf den diese zu Beginn gerichtet ist. Im Anschluss daran findet eine vom Nutzer auf dem jeweiligen Endgerät initiierte 360°-Drehung (siehe Abbildung 38) der Kamera statt. Durch diese wird der Datendurchsatz erhöht. Abschließend kommt die Kamera wieder zum Stillstand für zirka 2 Sekunden.

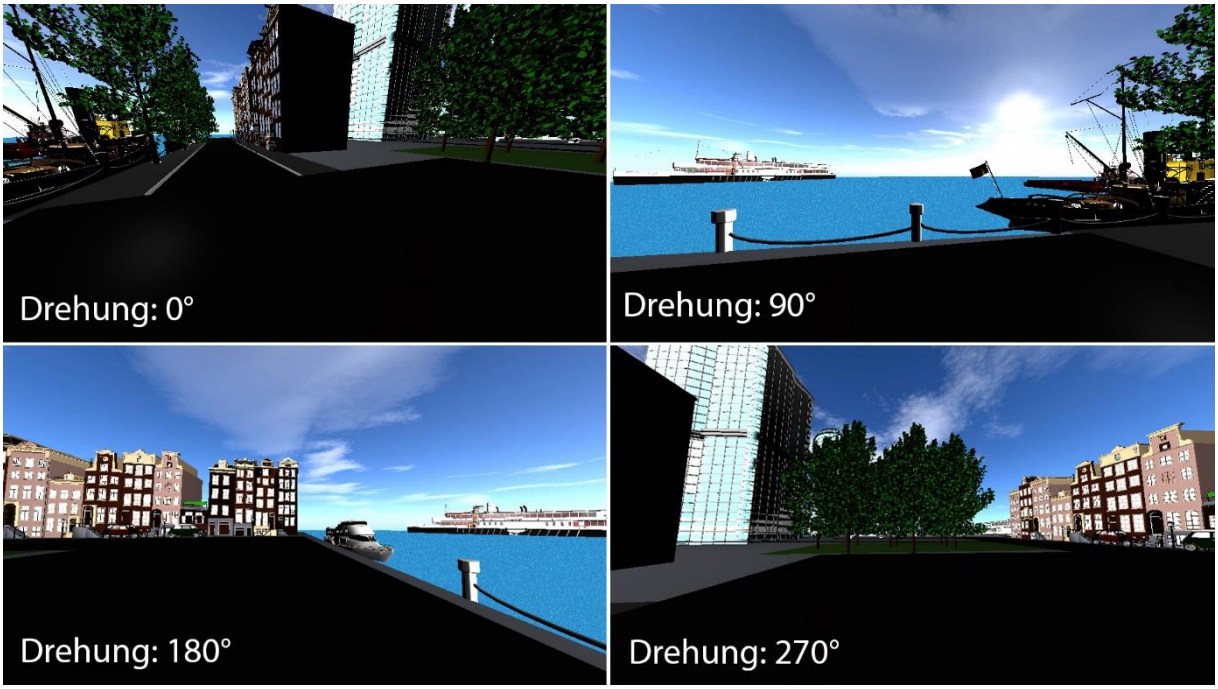


Abbildung 38 : Einzelansichten der Drehung

Messwerte

- *Zeit* in Sekunden
- *Frames per Second*
- *Datendurchsatz* in Kilobyte (kB) pro Sekunde
- *Gesamtes Datenvolumen* in MegaByte (MB)

6.1 Endgeräte (serverseitig)

Abbildung 39: Serverseitige Endgeräte

Modell **Funktion**	**Home Desktop-PC** **Server**
CPU RAM (in MB.)	Intel Core i3-4130 8192
GPU RAM (in MB.)	NVIDIA GeForce GTX 750 4041

6.2 Endgeräte (clientseitig)

Modell **Funktion**	**HP ProBook 4730s** **Web-Client**	**Apple Iphone 6** **Web-Client**
CPU RAM (in MB.)	Intel Core i5-2410LM 4000	A8 Dual-Core 1,4 MHz 1000
GPU RAM (in MB.)	AMD Radeon HD6490M 512	PowerVR GX6650

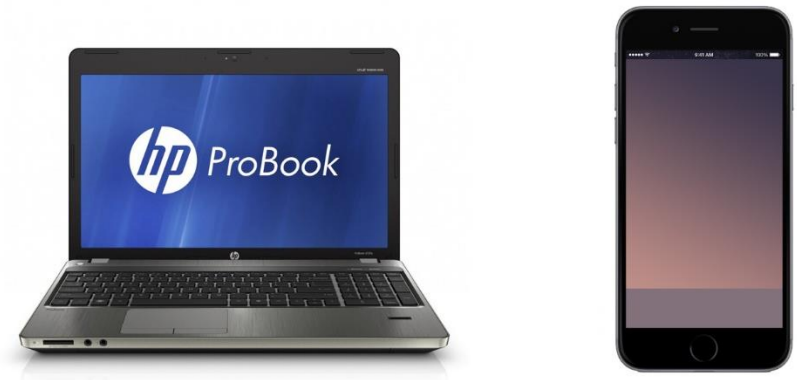


Abbildung 40: Clientseitige Endgeräte

6.3 Netzwerk-Szenarien der Datenraten-Messung

Client			Server: **Home PC** / **LAN**
Endgerät	**Standort**	**Netzwerk**	
Smartphone	Home	WLAN	
Notebook	Home	WLAN	

	Heim-Netzwerk	
	LAN	*WLAN*
Übertragungsstandard	DSL 50.000	DSL 50.000
Übertragungsrate	1 GBit/s	65 Mbit/s

Anmerkung: Für die Messungen werden die mobilen Endgeräte in einem WLAN-Netzwerk registriert sein, da dies eher der Realität entspricht. Zum einen sind aktuelle Datentarife für große Datenmengen jenseits der 2 GB nicht ausgelegt, zum anderen besteht selten die Möglichkeit mobile Endgeräte wie Smartphones an das Ethernet anzubinden.

6.4 Auswertung der Daten

6.4.1 Standbild 10 Sekunden, Limit: 30FPS, 960 x 540 Pixel

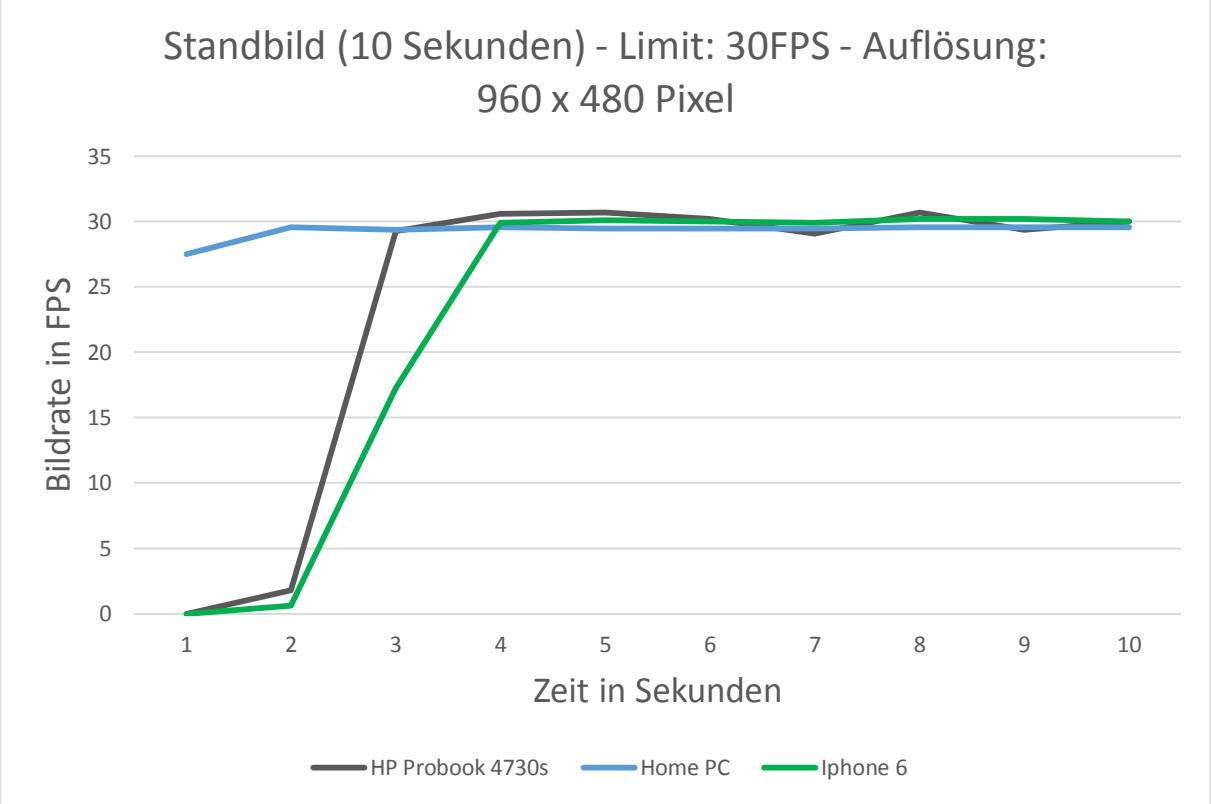


Diagramm 1: FPS - Standbild

Zunächst sei an dieser Stelle erwähnt, dass die Entscheidung gegen Full-HD und für eine Auflösung von 960 x 540 Pixel daraus resultiert, dass diese insbesondere für die Darstellung von Inhalten auf Smartphones gut geeignet ist. Aus dem Diagramm 1 der Messung mit dem PC als Server wird für die „Frames per Second“ deutlich, dass nach einer kurzen Verweilphase auf niedriger Bildrate sowohl das Notebook als auch das Smartphone auf Augenhöhe waren. Die kurze Tiefphase ist darin begründet, dass zunächst initial das erste Bild empfangen und dekodiert werden muss. Anschließend wird die Stärke der Vorhersage-Frames des H.264 Codecs beim Standbild sehr deutlich, da die Bildrate fortwährend der des Servers entspricht.

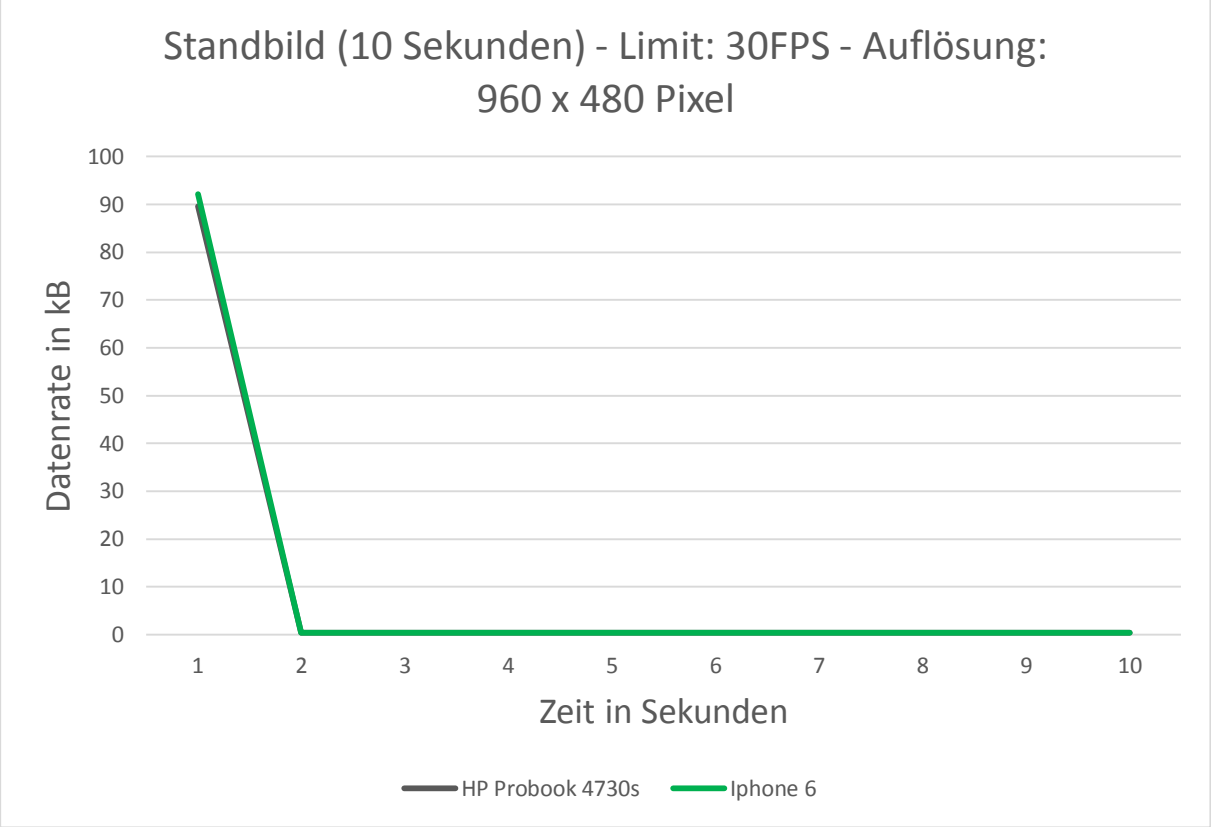


Diagramm 2: Kilobyte/Sek. - Standbild

Sehr gut ist im Diagramm 2 ebenfalls der in Kapitel 3.4 beschriebene Aspekt der „Vorhersage“ des H.264 Codecs zu erkennen. Zeigt der Nutzer die Szene erstmals an so ist der erste Frame ein „I-Frame“. Dieser muss vollständig geladen werden und liegt hier mit einer Größe von ca. 90 Kilobyte vor. Danach wird über die „Vorhersage“-Algorithmen jedes weitere Bild interpretiert, was eine Reduktion der Datenrate auf ein Minimum von ca. 0,03 Kilobyte zur Folge hat.

6.4.2 Drehung (360 Grad), Limit: 30 FPS, 960 x 540 Pixel

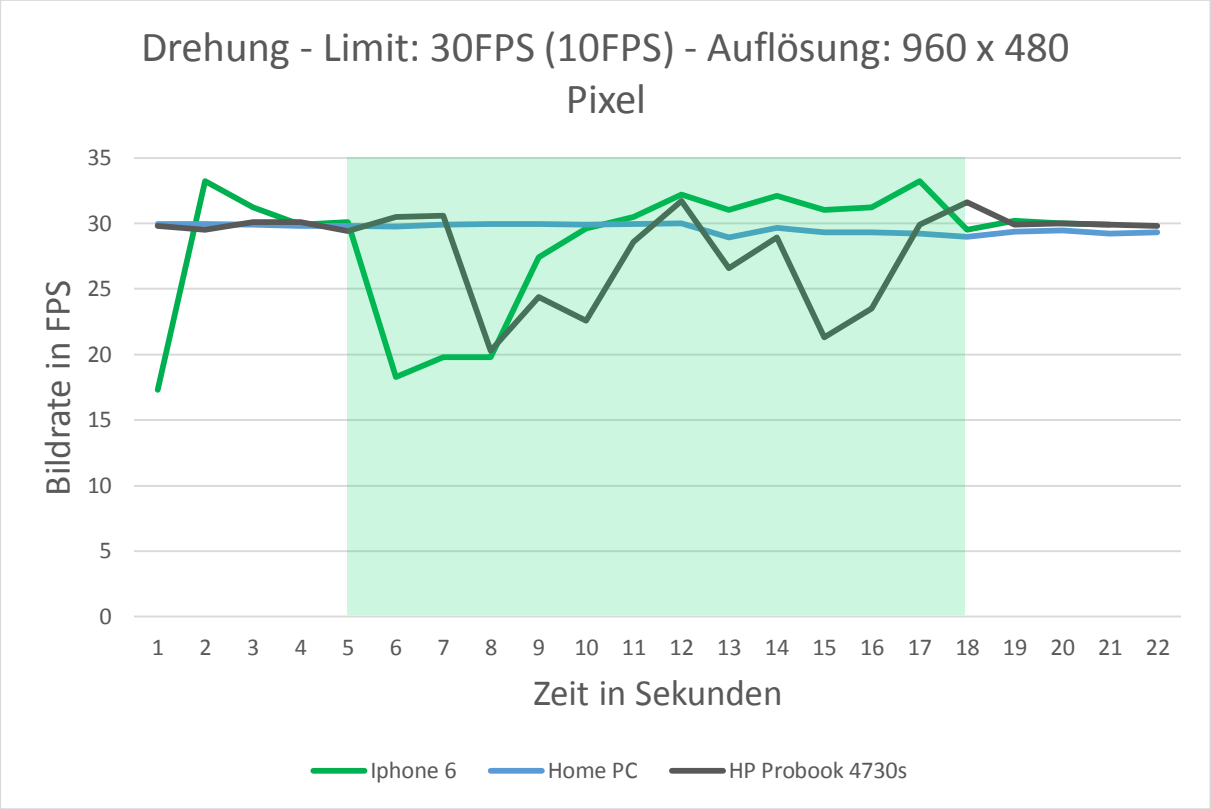


Diagramm 3: FPS - Drehung

Der Zeitraum der Drehung wird im Diagramm 3 mit einem grün-transparenten Kasten versehen, um die Zusammenhänge besser zu erkennen auf die im Folgenden eingegangen wird. Nach einem Standbild von ca. 5 Sekunden beginnt die Drehung, welche durch einen Einbruch der FPS-Zahl sowie der Erhöhung der Datenrate einhergeht und auf beiden Endgeräten deutlich zu erkennen ist. Warum sich im Verlauf der Drehung die FPS-Zahl des Smartphones schneller erholt, als die des Notebooks ist unklar.

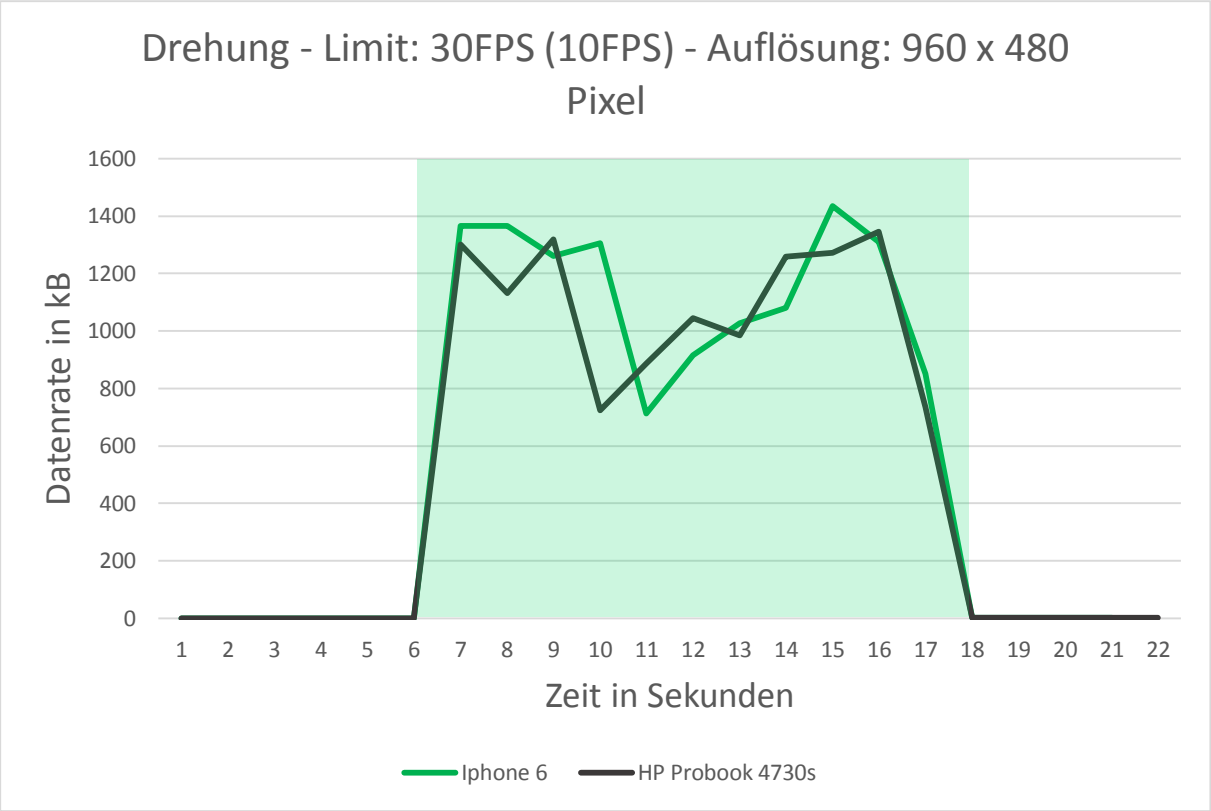


Diagramm 4: Kilobyte/Sek. - Drehung

Das kurzzeitige „Erholen der FPS-Zahl“ sowie die Schwankungen der Datenrate sind auf die jeweils betrachtete Komplexität der Szene zurückzuführen. Wie in Abbildung 38 zu erkennen ist, gibt es Szenenbereiche mit viel Geometrie im sichtbaren Feld aber auch deutlich geringer bestückte Areale. Jenach Komplexität des Bildausschnitts benötigt ein zu übertragendes Bild mehr oder weniger Datenvolumen, was sich ebenfalls in den Graphenverläufen in Diagramm 3 zeigt. Während die Datenrate auf Grund eines wenig komplexen Ausschnitts abnimmt, erhöht sich gleichermaßen die FPS-Zahl für diesen Moment.

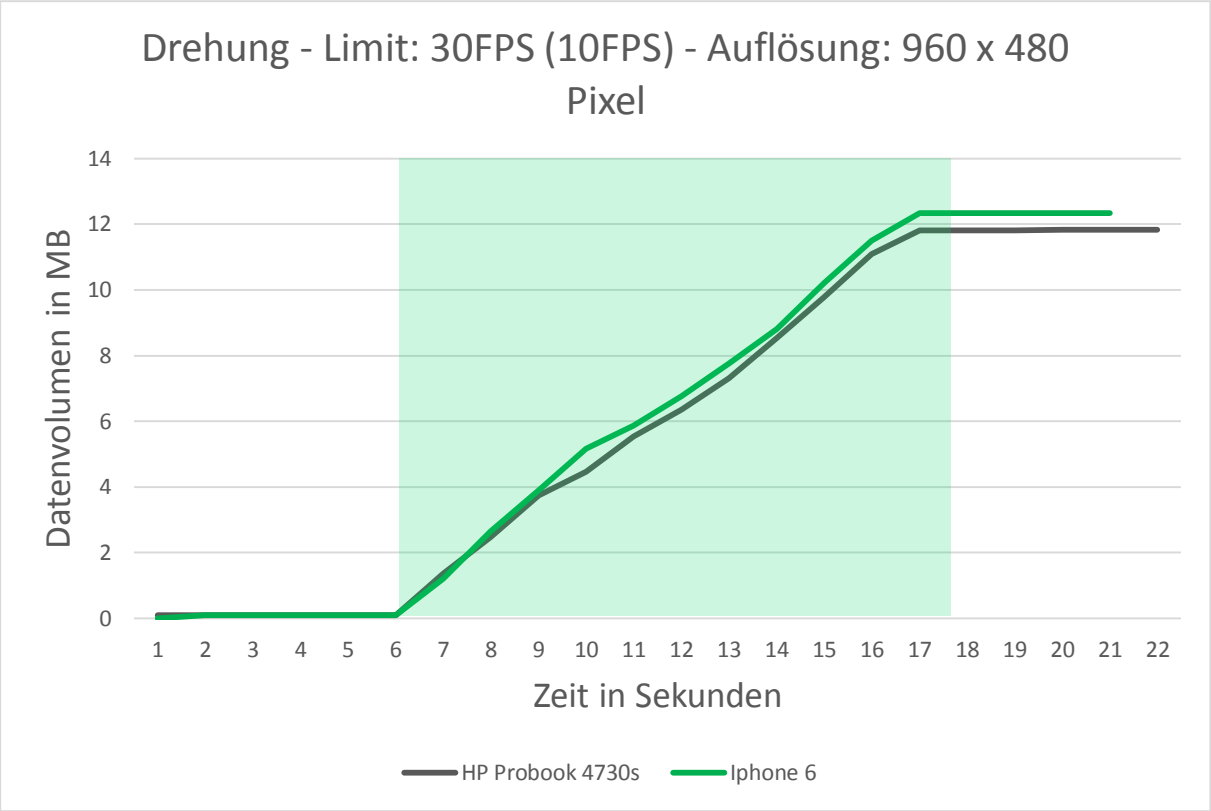


Diagramm 5: Datenvolumen - Drehung

Diagramm 5 beschreibt den Verlauf des Datenvolumens über die Zeit der Drehung. Der Anstieg der gesamten Daten verdeutlicht an dieser Stelle nochmals das Kern-Konzept des H.264 Codecs. In Ruhephasen bleiben Datenvolumen sowie Datenrate sehr niedrig und lediglich in Kamerafahrten mit dynamischen Szenenänderungen entsteht ein Anstieg.

6.5 Messung der Enkodierungs- und Dekodierungszeiten

Im Rahmen der Messungen wurden neben den client-seitigen Daten auch server-seitige Daten erfasst. Dies war insbesondere die Zeit, welche zum Dekodieren von 100 Frames benötigt wurde, um auch eine Aussage über einen länger andauernden Zeitraum treffen zu können. Zum Vergleich wurde ein leistungsstarker Desktop-PC (im Folgenden als Uni-PC bezeichnet) mit integrierter Nvidia GeForce Titan X herangezogen. Der folgende Code-Ausschnitt beschreibt die Messmethodik:

```
public class render{

// Rendering...

// 1. Messpunkt
re_last = System.nanoTime();

      // Enkodierungs-Resource an Encoder binden
      long mappedPtr = RE_API.MapGraphicsResource(resource);
      // Enkodierung und senden
      RE_API.EncodeAndSendSurface(encoder, mappedPtr);
      RE_API.UnmapGraphicsResources(resource, mappedPtr);

// 2. Messpunkt
re_now = System.nanoTime();

      nanos =re_now - re_last;
      re_last = re_now;


      // frames - Ausgabe pro Sekunde
      // nanos  - Ausgabe pro 100 Frames

}
```

Es wurde jeweils vor und nach dem Enkodierungs- und Sendevorgang ein Zeit-Stempel gesetzt, welcher in Nanosekunden einen Anhaltspunkt für die Enkodierungszeit liefert.

Serverseitige Enkodierung

Gerät	**FPS**	**Enkodierungszeit / 100 Frame**
Desktop PC (Uni)	30	5125986,6 Nanosekunden
		5,12 Millisekunden
Desktop PC (Home)	30	16809081,2 Nanosekunden
		16,8 Millisekunden

Dabei wurde deutlich, dass die Enkodierungs-Algorithmen im Uni-PC wesentlich näher an der „harten Echtzeit“ von ca. 5ms arbeiten als jene der schwächeren Hardware im Home-PC, welche zeitlich um mehr als das Dreifache variieren. Dennoch konnten jedoch kaum signifikante Unterschiede in erzielter FPS-Zahl und Darstellung auf Client-Seite sichtbar gemacht werden, was an dieser Stelle die Vermutung nahelegt, dass auch mit Grafikkarten der oberen Mittelklasse für Remote-Rendering gute Ergebnisse erzielt werden können.

7 Erfahrungsbericht der RE API Integration

Dieses Kapitel spiegelt einen persönlichen Erfahrungsbericht wieder, der sich wie folgt gliedert: Zunächst wird die Integrierbarkeit der Remote Rendering Software der AdaptVis GmbH in die Java-Anwendung des Automobil-Konfigurators beschrieben. Ergänzend werden ferner Richtlinien dargelegt, welche von Programmen, die den Remote Renderer integrieren möchten erfüllt sein müssen, damit eine reibungslose Einbettung gewährleistet werden kann. Schlussendlich werden Voraussetzungen an die Hardware auf der Server-Seite aufgezeigt, welche dazu beitragen, dass ein qualitativ hochwertiges Endergebnis auf der Client-Seite realisiert werden kann.

7.1 Integrierbarkeit

Durch AdaptVis werden die benötigten Software-Komponenten der Remote Encoder API in Form von DLLs und Archiven bereitgestellt. Diese folgenden Ausführungen erläutern dieses grundsätzlich (ohne ins Detail zu gehen) und behandeln die Integration in Java-Programme:

- *RE_JNI_Bridge.dll*: In dieser DLL liegt die in Kapitel 3.5 beschriebene JNI Bridge in ihrer Implementation vor. In dieser Ausführung ermöglicht das Java Native Interface die Kommunikation zwischen der in Java geschriebenen Hauptanwendung so wie der in C/C++ geschriebenen Remote Encoder API. Die DLL liegt im Grundverzeichnis der Anwendung und muss entsprechend im Build-Path der Anwendung integriert werden.
- *RemoteEncoderRelease.dll*: In dieser DLL liegt der RemoteEncoder in seiner Logik implementiert. Um dessen Funktionalitäten in der Java-Anwendungen nutzen zu können, muss dieser im Build-Path der Anwendung integriert sein.

- *jRemoteEncoder*: Hier liegen die Java Class-Dateien für den Remote-Encoder und den entsprechenden Event-Listener. Die Ordner des Archivs müssen im Programm und Java-Anwendungen im Allgemeinen über einen Import zu Beginn eingebettet werden.

  Dies geschieht wie folgt:

  ```
  import com.av.RE_API;
  import com.av.RE_API.RE_EventListener;
  ```

  Zusätzlich ist eine Integration der jRemoteEncoder.zip im Build-Path der jeweiligen Anwendung notwendig. Sind die Software-Komponenten korrekt eingebunden muss die `RE_API` aus der Anwendung heraus initialisiert werden, sowie jeweils ein Objekt des RemoteEncoders und `RE_EventListeners` angelegt werden. Im weiteren Programm-Verlauf muss ein Framebuffer angelegt werden, in welchen die aktuelle Szene gerendert wird, sodass dieser als Grundlage für die Enkodierung und Übertragung an den Client genutzt werden kann. Eine Möglichkeit der Umsetzung, wurde bereits in Kapitel 4.3.3 – Serverseitige Abläufe – beschrieben.

7.2 Voraussetzungen an die Anwendung

Eine Anwendung mit integrierter Remote Encoder API sollte in erster Linie aufwändige Grafikberechnungen ausführen, deren qualitative Darstellung sowohl auf GPU-leistungsfähigen, als auch insbesondere GPU-leistungsschwachen Endgeräten (Smartphones, Tablets und Notebooks) gewährleistet werden soll. Dabei erwartet das Programm stets neue Nutzerinformationen auf deren Basis der grafische Inhalt aktualisiert und an den Nutzer gestreamt wird.

7.3 Hardware-Voraussetzungen des Servers

Im Rahmen dieser Arbeit konnten einige Aspekte ermittelt werden, welche die Hardware auf Server-Seite maßgeblich bestimmen. Einerseits ist dies ganz allgemein die Rechenleistung der Grafikkarte, andererseits aber auch die Enkodiergeschwindigkeit. Der Vorgang des Encodierens kann dabei von einer Software durchgeführt werden oder aber wie in dieser Arbeit durch den Hardware-Encoder auf der Grafikkarte selbst.

Die Maxwell NVENC Hardware verdoppelt dabei die Encoding-Performanz im Vergleich zu Kepler-GPUs und schafft dabei 16-faches Echtzeit-Video-Encoding (1x HD = 1080p bei 30 FPS) (Nvidia, 2014). Das bedeutet, dass die Hardware bis zu 480 Frames von Videomaterial mit einer Auflösung von 1920x1080 Pixeln pro Sekunde encodiert, im höchsten Performance-Mode (HP preset) (Nvidia, 2014).

8 Fazit und offene Problemstellungen

Zusammenfassend zeigt diese Arbeit, dass Remote Rendering im Bereich der Darstellung von 3D-Inhalten auf mobilen Endgeräten ein vielversprechendes Konzept darstellt und mit entsprechender Soft- und Hardware ökonomisch und performant umgesetzt werden kann. Dass es sich bei Remote Rendering um eine zukunftsweisende Technologie handelt zeigt auch das Interesse anderer Branchen und Firmen wie NVIDIA. Wurden vor nicht allzu langer Zeit Musik und Filme noch über CDs, Kassetten und DVDs konsumiert, bilden immer mehr Streaming-Dienste die Grundlage für Entertainment in den heimischen vier Wänden. Mit NVIDIAs Cloud-Gaming Projekt „Shield“ wird ein weiterer Schritt in einer immer stärker vernetzten Welt geebnet. Videospiele müssen nicht mehr in Form eines Datenträgers beim Endnutzer vorliegen, sondern können unabhängig vom Standort und der clientseitigem Hardware über das Internet gestreamt werden. Im Rahmen dieser Arbeit wurde der in Java-Script und WebGL implementierte Autokonfigurator zunächst nach C++/OpenGL übersetzt und anschließend in Java/OpenGL. Die überarbeitete Klassenstruktur und die damit erzielte Modularisierung (inbesondere der Geometrien) des Programmaufbaus erwies sich als übersichtlich und verhalf den Remote Renderer der Firma AdaptVis erfolgreich zu integrieren und zu analysieren. Das Ergebnis verdeutlicht zum einen die Leistungsfähigkeit des Produkts, zum anderen aber auch die plattformunabhängige Vielseitigkeit. Unter Berufung auf alle durch die Arbeit gewonnenen Resultate können abschließend die folgenden offenen Problemstellungen formuliert werden:

1. **Multi-User**

   Ein wesentliches Kriterium an einen Online-Automobilkonfigurator ist eine Multi-User Organisation. Jeder Nutzer sollte eine eigene Session erhalten, sodass dieser unabhängig von anderen Kunden mit der Anwendung interagieren kann. Aktuell bietet die Anwendung die Möglichkeit, dass ein Nutzer ein Fahrzeug konfigurieren kann, wobei die in dieser Masterarbeit verwendete Server-Hardware (Nvidia

GeForce Titan X) auf bis zu 2 parallel laufende Encoder ausgelegt ist. Hier ist es jedoch an der RE API, diese entsprechend zu erweitern.

2. **Optimierung des Cube-Mapping-Shaders**

   Das implementierte Cube-Mapping befindet sich aktuell im Fragment-Shader, welcher speziell für Fahrzeuge verwendet wird. Dabei wird nicht zwischen spiegelnden und nicht-spiegelnden Oberflächen unterschieden. Das bedeutet, dass jede Fahrzeugkomponente zwar seine Grundfarbe (ambient, diffus, spekular) enthält, der Shader jedoch nicht bezüglich des Reflektionsgrades des jeweiligen Materials unterscheidet. Hier müsste eine zusätzliche Aufteilung aller Fahrzeug-Komponenten zwischen reflektierend und nicht reflektierend stattfinden. Im Rahmen des Renderings könnte sodann ein Phong-Shader beispielsweise alle nicht-reflektierenden Teile rendern, während ein um Cube-Mapping erweiterter Phong-Shader Reflektionen vornimmt.

3. **Ausweitung des Funktionsumfangs**

   Aufbauen auf Ansätzen der Bachelorarbeit (Bourdon, 2013) ist es denkbar, einzelne Fahrzeugkomponenten wie Felgen oder Spoiler durch alternative Versionen zu ersetzen. Auch ein Wechsel der Tageszeit könnte implementiert werden, um Lichter am Fahrzeug zu simulieren. Darüber hinaus wäre eine Steuerung auf mobilen Endgeräten via Touch-Gesten wünschenswert, was jedoch in der RE API noch nicht verankert ist. Ebenfalls ist es vorstellbar, den Trend der Virtual Reality aufzugreifen und eine Ansteuerung des Bewegungssensors sowie Gyroskop[1] zu implementieren. Über Brillen wie das Google Cardboard oder ähnliche Produkte könnte das Smartphone dann als virtuelle Bildfläche zur Konfiguration von Automobilen dienen.

[1] Gyroskop: Ein Gyrosensor ist ein Beschleunigungs- oder Lagesensor, der auf kleinste Beschleunigungen, Drehbewegungen oder Lageänderungen reagiert. Das Prinzip des Gyrosensors basiert auf der Massenträgheit und wird u.a. in Fliehkraftreglern eingesetzt. (ITWissen, 2016).

9 Literaturverzeichnis

Abts, Dietmar. 2007. *Masterkurs Client/Server Programmierung mit Java.* Wiesbaden : Springer, 2007.

Blog, Zwischengas. 2016. Zwischengas Blog. *Zwischengas Blog.* [Online] 04. 05 2016. http://www.zwischengas.com/de/blog/2013/03/29/Der-idealisierte-Volks-Sportwagen.html.

Bourdon, Timo. 2013. *Entwicklung einer WebGL-Applikation zur kundenspezifischen Konfigurierung von Automobilen.* Uni Osnabrück : Bachelorarbeit, 2013.

Eichler, Christoph. 2014. *Plattformunabhängige Darstellung von High-Quality-3D-Grafiken mittels Remote Rendering.* Uni Osnabrück : Bachelorarbeit, 2014.

Emscripten Dokumentation. [Online] [Zitat vom: 5. Mai 2016.] https://kripken.github.io/emscripten-site/docs/getting_started/FAQ.html.

Force, Internet Engineering Task. 2011. *RFC 6455 - The Websocket Protocol.* 2011.

Grigorik, Ilya. 2013. *High Performance Browser Networking.* Sebastopol, CA 95472 : O'Reilly, 2013.

ITWissen. 2016. ITWissen. *ITwissen.* [Online] 2016. [Zitat vom: 05. August 2016.] http://www.itwissen.info/definition/lexikon/Gyrosensor-gyro-sensor.html.

Liedke, Volker. 2013. Car Configurator als Standalone-App. Ein Erfolgskonzept!? [Online] Blogomotive.com, 2. Oktober 2013. [Zitat vom: 18. Juni 2016.] http://www.blogomotive.com/2013/10/car-configurator-als-standalone-app-ein-erfolgskonzept/.

Malte Ubl, Eiji Kitamura. 2010. html5rocks. [Online] 20. Oktober 2010. [Zitat vom: 11. Mai 2016.] http://www.html5rocks.com/de/tutorials/websockets/basics/.

McReynolds, T., Blythe, D. 2005. *Advanced Graphics Programming Using OpenGL.* San Francisco : Elsevier, 2005.

Nielsen. 2014. Auto-Werbung verstärkt Online. [Online] 20. 05 2014. http://www.nielsen.com/de/de/insights/reports/2014/auto-werbung-verstarkt-online.html.

Nielsen. 2016. Statista. *Entwicklung der Bruttoausgaben der Automobilhersteller für Werbung in Deutschland in den Jahren 2010 bis 2014 (in Milliarden Euro) .* [Online] 04. 05 2016. http://de.statista.com/statistik/daten/studie/74992/umfrage/werbeausgaben-der-automobilhersteller-in-deutschland/.

Nielsen, Media Research Company. 2012. Digital Facts Automobile. [Online] Juni 2012. [Zitat vom: 18. Juni 2016.] http://www.nielsen.com/content/dam/nielsenglobal/eu/nielseninsights/pdfs/Nielsen%20Digital%20Facts%20Automotive%20201206.pdf.

Nischwitz, A., Fischer, M., Haberäcker, P., Socher, G. 2013. *Computergrafik und Bildverarbeitung - Band 1: Computergrafik.* Wiesbaden : Vieweg+Teubner Verlag | Springer Fachmedien, 2013.

NVIDIA Corporation, 1999. 1999. Cube Map OpenGL Tutorial. [Online] 1999. [Zitat vom: 7. Mai 2016.] http://www.nvidia.de/object/cube_map_ogl_tutorial.html.

Nvidia. 2014. NVENC - NVIDIA HARDWARE VIDEO ENCODER. [Online] Juli 2014. [Zitat vom: 2. Juli 2016.] http://developer.download.nvidia.com/compute/nvenc/v4.0/NVENC_AppNote.pdf.

NVIDIA. 2016. NVIDIA GRID. *Cloud Gaming.* [Online] 5. Mai 2016. http://www.nvidia.de/object/nvidia-grid-cloud-gaming-de.html.

Ofterdinger, Adrian. 2005. Java Native Interface ab J2SE 1.4. *Java Spektrum.* Mai 2005, 05, S. http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2005/05/ofterdinger_JS_05_05.pdf.

Okunev, A., Bashlykov, A. 2014. NVIDIA's NVENC Hardware H.264 Video Encoder. [Online] Medialooks, 17. November 2014. [Zitat vom: 18. Mai 2016.] http://blog.medialooks.com/814EAo/.

Oracle. 2016. Java SE Documentation. [Online] Oracle, 2016. [Zitat vom: 14. 05 2016.] http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html#wp9502.

Overview of the scalable video coding extrension of the H.264/AVC standard. **H. Schwarz, D. Marpe und T. Wiegand. 2007.** September 2007, 2007, IEEE Transactions on Circuits and Systems for Video Technology, S. 1103-1120.

Ozer, Jan. 2009. Adobe Developer Connection. [Online] 16. März 2009. [Zitat vom: 10. Juni 2016.] http://www.adobe.com/devnet/adobe-media-server/articles/h264_encoding.html.

Paul, Ryan. 2011. Native JavaScript H.264 decoder offers compelling demo of JS performance. [Online] arstechnica - Technology Lab / Information Technology, 31. Oktober 2011. [Zitat vom: 05. Juni 2016.] http://arstechnica.com/information-technology/2011/10/native-javascript-h264-decoder-offers-compelling-demo-of-js-performance/.

Programming, GLSL. 2014. GLSL Programming. *GLSL Programming.*

[Online] 05. April 2014.
https://en.wikibooks.org/wiki/GLSL_Programming/Blender/Reflecting_Surfaces.

Richardson, Iain E. 2010. *The H.264 Advanced Video Compression Standard (Second Edition).* West Sussex : Jon Wiley & Sons, Ltd, 2010.

Using H264 Encoder on NVIDIA GPUs. [Online] Leadtools. [Zitat vom: 18. Mai 2016.] https://www.leadtools.com/help/v19/multimedia/filters/usingh264encodernvidiagpus.html.

VIDEOAKTIV. 2013. VIDEOAKTIV - Videobrechnung: Optimale Einstellung für YouTube. *VIDEOAKTIV.* [Online] VIDEOAKTIV, 5. Dezember 2013. [Zitat vom: 10. Juni 2016.] http://www.videoaktiv.de/praxistechnik/editing-hintergrundinfo/videoberechnung-optimale-einstellungen-fur-youtube.html.

Wenke H., Kolodzey S., Wittkorn E. Echtzeit-Datenvisualisierung / Parallele Programmierung . [Online] AdaptVis. [Zitat vom: 2016. Mai 17.] http://www.adaptvis.com/?content=services#remote_rendering.

10 Abbildungverzeichnis

11 Diagrammverzeichnis

12 Verzeichnis der verwendeten 3D Modelle

Datum	**Autor**	**Modellname (*Lizenz*) und Link**
14.05.2016	Unbekannt	Skyscraper (*Creative Commons Zero 1.0*)
		http://www.blendswap.com/blends/view/81908
16.05.2016	Unbekannt	Hermes Kutter (*Creative Commons Zero 1.0*)
		http://www.blendswap.com/blends/view/41826
16.05.2016	Thomaswornall	Kutter “Barge” (*Creative Commons Attribution 3.0*)
		http://www.blendswap.com/blends/view/39638
04.06.2016	Skaldy	Amsterdam Canals (*Creative Commons Attribution 3.0*)
		http://www.blendswap.com/blends/view/75091
17.06.2016	Seraphim10	Austin Cooper 1962 (*Creative Commons Zero 1.0*)
		http://www.blendswap.com/blends/view/74398
19.06.2016	Helijah	Citroen 2CV (*Creative Commons Zero 1.0*)
		http://www.blendswap.com/blends/view/66427
19.06.2016	TheDuckCow	Yacht (*Creative Commons Attribution 3.0*)
		http://www.blendswap.com/blends/view/26723

Erklärung zur selbstständigen Abfassung der Master-Arbeit

Name: _____

Geburtsdatum: _____

Matrikel-Nummer: _____

Titel der Master-Arbeit: _____

Ich versichere, dass ich die eingereichte Master-Arbeit selbständig 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 andrer Autoren entnommenen Stellen habe ich kenntlich gemacht

_____ _____

Ort, Datum Unterschrift