Entwicklung einer WebGL-Applikation zur kundenspezifischen Konfiguration von Automobilen

Timo Bourdon

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science

Universität Osnabrück

Fachbereich 6: Mathematik und Informatik

Erstgutachter: Prof. Dr. rer. nat. Oliver Vornberger

Zweitgutachter: Dr. rer. nat. Jutta Göers

Danksagung

Mein herzlicher Dank geht an dieser Stelle an Herrn ***Prof. Dr. Oliver Vornberger*** und dies nicht nur wegen seiner sofortigen Zusage zur Betreuung und Erstkorrektur der Arbeit. Ebenso war seine Kooperation mit Volkswagen und die immerwährende Unterstützung eine große Hilfe und Motivation über den gesamten Zeitraum. Darüber hinaus möchte ich auch Frau **Dr. Jutta Göers** für ihre sofortige Bereiterklärung zur Zweitkorrektur der Arbeit danken.

Ein besonderer Dank gilt ***Henning Wenke***, der als Betreuer jederzeit ansprechbar, konstruktiv und motivierend wirkte. Ab der gemeinsamen Ausformulierung des Themas war er ein nicht mehr weg zudenkender Bestandteil meiner Arbeitsumgebung, nicht zuletzt auch auf Grund seiner umfassenden Hilfsbereitschaft, die ich zu keinem Zeitpunkt als eine Selbstverständlichkeit sah.

Des Weiteren möchte ich ***Erik Wittkorn*** danken, mit dem ich Dienstag um Dienstag über meine Arbeit im Diplomandenraum sprechen konnte und der stets wertvolle Tipps und Tricks beisteuerte.

Ebenso möchte ich meiner Familie sowie meiner Freundin ***Vera***, danken, die meine Teils sehr überehrgeizigen Momente und freudige Überschwänglichkeit in die richtigen Bahnen gelenkt hat und trotz der eigenen universitären Belastungen immer für mich da war.

INHALT

DANKSAGUNG **2**

1. EINLEITUNG UND MOTIVATION **5**

2. GESCHICHTE **6**

2.1. 3D IM INTERNET 6
2.2. WEBGL 7

3. AKTUELLER STAND VON ONLINE-AUTOKONFIGURATOREN **9**

3.1. FLASH – VW KONFIGURATOR 9
3.2. UNITY – PORSCHE KONFIGURATOR 11
3.3. ALLGEMEINER ÜBERBLICK 13

4. HTML5 .... **14**

5. JAVASCRIPT **15**

6. COMPUTERGRAFIK-GRUNDLAGEN **15**

6.1. GEOMETRIE-INFORMATIONEN 15
6.2. PRIMITIVE-ERZEUGUNG 17
6.3. PROJEKTIONS-MATRIZEN 18
6.4. GRAPHICS-PIPELINE 21
6.5. PHONG-LIGHTING 23
6.6. LICHTQUELLEN 25

7. WEBGL **27**

7.1. SYSTEM-VORAUSSETZUNGEN 27
7.2. STRUKTUR EINER WEBGL APPLIKATION 28
7.3. DER WEBGL-KONTEXT 29
7.4. INITIALISIERUNG 30
7.5. VERGLEICH OPENGL – WEBGL 32

8. 3D MODELLDATEN **40**

8.1. AUTODESK MAYA 40
8.2. BLENDER 40
8.3. AUSGANGSDATEN 41
8.4. OBJ, MTL UND JSON 42

9. DIE CLIENT-SERVER-ARCHITEKTUR **48**

10. CLIENT - DER AUTOMOBIL-KONFIGURATOR **49**

11. SERVER **50**

11.1. DIE SZENE 51
11.2. BLENDER 3D MODELLE 51
11.2.1 Export von Blender nach OBJ 52
11.2.2 Parsen von OBJ und MTL nach JSON 53
11.3. RESSOURCEN-ORIENTIERTES RENDERING 54

12. IMPLEMENTATIONSDETAILS 56
12.1. HTML 57
12.2. SKRIPTE 57
12.3. DIE ANWENDUNG 61
13. PERFORMANZ ANALYSE 71
14. ERGEBNISSE 74
15. VERGLEICH 76
15.1. WERKZEUGVERGLEICH - FLASH VS. UNITY VS. WEBGL 76
15.2. OPTIK UND PERFORMANCE VON KONFIGURATOREN 78
16. AUSBLICK 79
17. FAZIT 80
LITERATURVERZEICHNIS 81
ABBILDUNGSVERZEICHNIS 85
FAHRZEUGDATEN 86
TABELLENVERZEICHNIS 86

1. Einleitung und Motivation

Abbildung 1: VW Coupé Original
Quelle: [Web 1]

Abbildung 2: VW Coupé bearbeitet in Photoshop

Von den ersten Design-Entwürfen, über die Fertigung, bis hin zur Präsentation eines Automobils sind CAD-Daten fundamentaler Bestandteil dieses Prozesses. Während in der Konzeption und Konstruktion die Fahrzeugdaten in Modellierungsprogrammen generiert werden, liegen die Daten zum Zeitpunkt der Präsentation und Vermarktung in standardisierten Formaten vor, die die Grundlage für weitere Verarbeitungsschritte im Marketing bilden. Um dem Kunden die Gestaltung seines eigenen, potentiellen Fahrzeugs zu ermöglichen, bieten namenhafte Automobilhersteller heutzutage Automobilkonfiguratoren im Internet an. Die der Implementierungsansatz kann dabei sehr verschieden sein, wie an den Varianten von Porsche und Volkswagen zu sehen ist. Während der Sportwagenhersteller auf in Echtzeit gerenderte 3D Modelle via UnityEngine setzt und dem Kunden ein interaktives Betrachtungserlebnis des Fahrzeugs ermöglicht, wählt VW den Weg über Flash und präsentiert seine Fahrzeuge im JPEG-Format. In Konkurrenz zu diesen beiden Standards steht WebGL, eine standardisierte Low-Level-API, die es in Kombination mit HTML5 schafft, hardwarebeschleunigt 3D-Inhalte im Internet zu rendern. Da der WebGL-Standard noch sehr jung ist und nur wenige komplexe Anwendungen existieren, entstand neben der persönlichen Motivation für das Thema, die Idee, einen Automobilkonfigurator in WebGL zu implementieren, der es dem Kunden ermöglicht, browser- und hardwareunabhängig sein persönliches Fahrzeug zu konfigurieren. Inhaltlich lehnt sich der Konfigurator stark an anderen Konfiguratoren an und bietet dahingehend ebenfalls eine 360°-Betrachtung, sowie autospezifische Modifikationen (Felgen, Lackierungen). Einer der Kerngedanken der Arbeit ist es zudem, nicht nur mit 3D-Daten aus dritter Hand zu arbeiten, sondern von einem

Originaldatensatz der Firma Volkwagen auszugehen. Durch seine Komplexität und Qualität bietet er so die Möglichkeit, eine stärkere Nähe zur industriellen Realität zu schaffen. Darüber hinaus ist es ein weiteres Ziel, eine Art Vergleichsstudie zwischen der weitverbreiteten Variante eines Flash-Automobilkonfigurators und dem in Bezug auf WebGL stärksten Konkurrenten, der UnityEngine, zu ziehen. Hiermit sollen fundierte Kenntnisse über deren Leistungsstand gewonnen und in Vergleich zu WebGL gesetzt werden.

2. Geschichte

2.1. 3D im Internet

Kaum ein anderes Medium dieser Welt hat in den vergangenen 20 Jahren derart viel an Bedeutung gewonnen, wie das Internet. Was ursprünglich eine Initiative des US-Verteidigungsministeriums zur Wahrung der Kommunikation nach einem atomaren Erstschlag war, entwickelte sich über ein viertel Jahrhundert zur bedeutendsten multimedialen Plattform der Welt. Dabei waren ursprünglich Browser lediglich in der Lage, Texte anzuzeigen und alles andere als multimedial. Aus diesem Grund bedurfte es den ersten Schritten von Tim Berners-Lee (HTML, Webserver, Webbrowser) 1989 [Web 2], sowie dem Aufkommen des ersten grafikfähigen Webbrowser Mosaic im Jahre 1993 [Web 3], um das Fundament für das gegenwärtige, moderne Internet zu legen. Mit der Gründung von Youtube im Jahr 2005 [Web 4] verstärkte sich das Interesse an Video-Inhalten im Internet, was dazu geführt hat, dass bis heute die größte Menge des Internettraffics durch Videomaterial entsteht.

Eine weitere Entwicklung ist bei der Form von Webseiten zu erkennen. Waren anfängliche Webpräsenzen hauptsächlich von informativer bzw. darstellender Natur, so sind sie heute kaum von modernen Desktop-Anwendungen zu unterscheiden [VoAJ07]. Dieser Wandel ist vor allem auf das Transport-Konzept AJAX zurück zu führen, das es ermöglicht, asynchrones JavaScript und XML zu verbinden. So bietet beispielsweise Google mit Docs einen Webservice an, der der Microsoft Office Palette sehr ähnlich ist [Web 5].

Ein weiterer Schritt in dieser Entwicklung ist neben der Darstellung von 2D Inhalten auch die Implementierung von 3D Anwendungen im Internet. Auf der einen Seite ist dieses Bestreben schon so alt wie das Internet selbst, doch bieten erst die jüngsten Errungenschaften der

Webentwicklung wie AJAX und die 4. Generation der Programmiersprachen (PHP, JavaScript, HTML) das dafür notwendige Fundament. Einen ersten Ansatz dreidimensionale Inhalte darzustellen, stellt im Jahr 1995 VRML 1.0 (Virtual Reality Markup Language) dar. Hierbei handelt es sich um eine Beschreibungssprache für 3D-Szenen, sowie die damit verbundenen Geometrien, Beleuchtungen, Animationen und Interaktionsmöglichkeiten. Sein Nachfolger VRML97 (Version 2.0) wurde mit X3D bekannt gegeben und 2001 vom W3C-Konsortium als offizieller Standard für 3D-Inhalte im Internet verabschiedet [Web 6]. Durch die rasanten technologischen Fortschritte im Hardwarebereich in Bezug auf Grafikkarten und Prozessoren, sind heute die Türen geöffnet für komplexe und datenintensive 3D-Anwendungen. Einmal mehr hat Google auch hier ein Zeichen gesetzt, als im Jahr 2004, Google Earth den Weg ins Internet wie auch auf die heimischen Desktop-PCs gefunden hat. Zur Optimierung der Performance stand schon damals der OpenGL-Standard [PT3D] zur Verfügung und bereitete so nachfolgenden Projekten wie WebGL den Weg.

2.2. WebGL

WebGL ist ein Webstandard für eine Softwarebibliothek, die betriebssystemunabhängig und ohne zusätzliche Plug-Ins im Webbrowser läuft [M&C11]. Im Detail handelt es sich um eine sogenannte Low-Level API, die zur Entwicklung von 3D-Grafiken unter dem Aspekt der Hardwarebeschleunigung dient. Hardwarebeschleunigt bedeutet, dass rechenintensive Operationen auf dem Grafikprozessor (GPU), statt auf der CPU ausgeführt werden, um so die enorme Rechenleistung moderner Grafikkarten auszunutzen. Diese Entlastung der CPU sorgt für eine bessere Performance und spart Ressourcen. Im Webbrowser werden WebGL-Inhalte über das in HTML5 eingeführte Canvas-Element dargestellt. Auf Grund des Zusammenspiels mit der Programmiersprache JavaScript kann so direkt auf Elemente im Document Object Model zugegriffen und WebGL-Anwendungen ohne weiteres mit anderen JavaScript konformen Bibliotheken wie JQuery kombiniert werden.

WebGL basiert als eine Low-Level 3D Grafik-API auf OpenGL ES 2.0, nutzt die OpenGL Shading Language (GLSL) und wird von der Khronos Group sowie Mozilla als lizenzfreier Standard entwickelt. *ES* steht dabei für „embedded systems“, da diese Version für Geräte wie das iPhone, iPad und Android Smartphones konzipiert ist. Erste konkrete Arbeiten am

Projekt WebGL begannen im April 2009 [Web 7]. Im Mai 2010 wurde seitens der Khronos Group bekannt gegeben, dass auch Google den Standard unterstützt. Weitere namenhafte Unternehmen unter dem Dach der Khronos Group sind die IT-Unternehmen AMD, Nvidia, Apple und Opera [Web 8]. Dies zeigt das starke Interesse großer Unternehmen an dem neuen Standard.

ABBILDUNG 3: ÜBERBLICK ÜBER DIE UNTERNEHMEN DER KHRONOS GROUP
QUELLE [WEB 9]

Der größte Vorteil von WebGL ist, dass es nativ im Browser läuft, sofern dieser natürlich den Webstandard unterstützt. Darüber hinaus sind im Allgemeinen 3D-Anwendungen sehr rechenintensiv. Hier ist es wichtig, dass der Standard mit Hardwarebeschleunigung ausgerüstet ist, um eine effiziente Darstellung im Webbrowser zu gewährleisten. Die Technik des direkten Hardwarezugriffs und die damit verbundene Leistungsverbesserung war bis zu diesem Zeitpunkt nur Desktopanwendungen vorbehalten. Um den Einstieg und Zugang zur Programmierung in WebGL zu erleichtern, wurden im Lauf der Jahre viele JavaScript Bibliotheken entwickelt, die durch eine eigens konzipierte API auf die WebGL API zugreifen können (Three.js, etc.).

Trotz der vielen Vorteile, die der Standard mit sich bringt, finden im Internet auch alternative Technologien ihre Anwendung, die in direkter Konkurrenz auf dem Feld der dritten Dimension im Web zu WebGL stehen. Darunter sind Flash als Plattform zur Programmierung und Darstellung multimedialer und interaktiver Inhalte im Web, oder die Unity Game Engine, um 3D Rendering in Echtzeit im Browser zu ermöglichen. Im Folgenden wird auf diese beiden alternativen Technologien eingegangen, wobei stets der Fokus auf deren Verwendung in modernen Automobilkonfiguratoren gerichtet ist.

3. Aktueller Stand von Online-Autokonfiguratoren

3.1. Flash – VW Konfigurator

Abbildung 4: Flash Logo
Quelle [Web 10]

Bei Flash handelt es sich um eine Plattform zur Programmierung und Darstellung multimedialer und interaktiver Inhalte im Web, die ursprünglich von der Firma Macromedia ins Leben gerufen wurde. Im Jahr 2005 wurde Macromedia von Adobe aufgekauft, womit auch sämtliche Rechte an Adobe übergingen. Wie Eingangs beschrieben, sind Flash-Inhalte im Browser meist in Form von Bannern, Video- und Slideshow-Elementen, sowie interaktiven Schaltflächen (Spiele, Buttons, Menüs, etc.) gegeben und benötigen zur reibungslosen Funktion das Flash-Player-Plugin.

Über die Flash-Technologie agiert ebenso der deutsche Automobilkonzern, um seine Fahrzeugpalette einem breiten Publikum in Form eines Automobilkonfigurators zu präsentieren. Sämtliche Fahrzeuge liegen dabei im JPEG-Format vor, ebenso die verschiedenen Modellvarianten und ihre damit verbundenen Lackierungen und Felgen-Option. Dem Anwender werden jedoch unmittelbar bei der Nutzung von Flash dessen erste Einschränkungen deutlich. Beispielsweise hängen die Perspektiven von den gewählten Bildern des Fahrzeugs ab, sodass am Beispiel Volkswagen lediglich eine Vorder- und Rückansicht möglich ist.

Abbildung 5: VW Touareg vorne
Quelle [Web 11]

Abbildung 6: VW Touareg hinten
Quelle[Web 12]

Etwas mehr Perspektiven bietet der Konfigurator der Marke Audi, doch auch hier stößt man auf dieselben Limitierungen. Darüber hinaus resultiert aus den von Flash vorgeschriebenen Richtlinien der Tastaturbelegungen, ein eingeschränkter Bewegungsradius innerhalb der Anwendung. Lediglich die Pfeil- und Leertasten sind für die Interaktion vorgesehen.

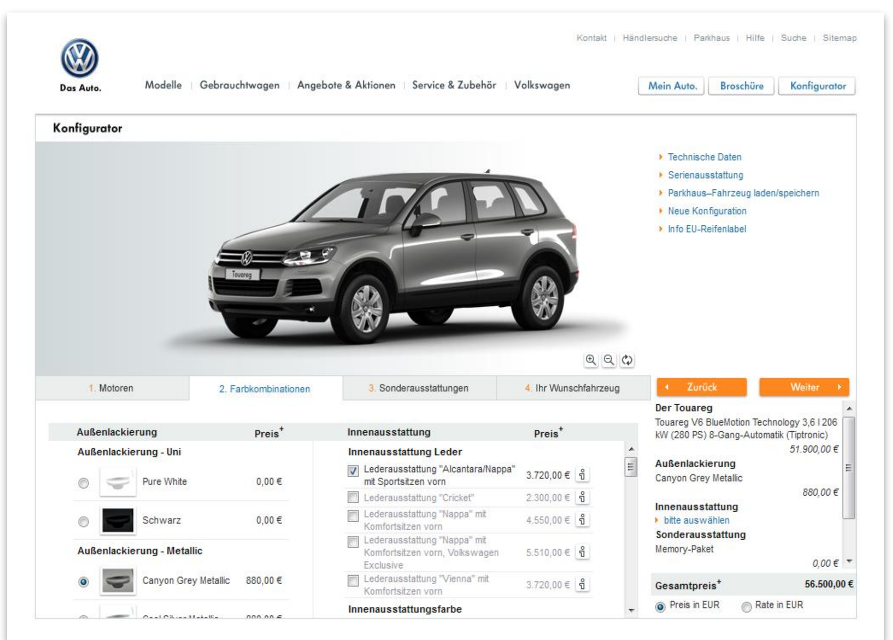


ABBILDUNG 7: VW AUTOKONFIGURATOR
QUELLE [WEB 13]

3.2. Unity – Porsche Konfigurator

Abbildung 8: Unity Logo
Quelle [Web 14]

Einen weiteren Weg, um multimediale Inhalte sowie in Echtzeit gerenderte 3D-Inhalte im speziellen, in Webbrowsern darzustellen, bietet die Unity Game Engine der Firma Unity Technologies. Das Unternehmen wurde 2004 in Kopenhagen (Dänemark) gegründet und legte früh den Fokus auf die Entwicklung einer Cross-Plattform bestehend aus einer Game-Engine sowie einer speziellen IDE (Integrated Development Environment).

Die Entwicklungsumgebung ist vom Aufbau stark angelehnt an etablierte 3D-Modelling-Programme und besteht zentral aus einem Szenefenster, das über verschiedenste Werkzeugleisten angesprochen wird. Darüber hinaus lässt sich die 3D-Szene auch über Programmierbefehle modifizieren – JavaScript, C# und Boo kommen hier zum Einsatz. Der Unity Editor charakterisiert sich als eine sehr intuitive Entwicklungsumgebung, da viele Elemente wie Sounds, Animationen und Texturen einfach via Drag&Drop importiert werden können und so den Einstieg in einen sehr komplexen Bereich der Medieninformatik enorm vereinfachen.

Mit dem wirtschaftlichen und gesellschaftlichen Aufstieg des Apple Iphone um 2008 [Web 15] erkannte auch Unity Technologies das Potential an der Einbettung von 3D-Inhalten in mobilen Webbrowsern [Web 16]. Somit entstand im Lauf der Folgejahre das Plugin „Unity-Web-Player“, das die generierten Inhalte für mobile wie auch standardmäßige Webbrowser verfügbar macht. Alle Inhalte werden dabei via OpenGL-Befehle hardwarebeschleunigt über die Grafikkarte berechnet, was optimale Effizienz der Anwendung und Performance leistet. Unity Technologies bietet für seine Nutzer eine freie Basis-Version der Software, vergrößert so sein Publikum an freien Entwicklern und stärkt die Community der Game-Engine, welche nach dem Magazin „Game Developers Magazine“ im Jahre 2011 neben der Unreal-Engine die beliebteste Game-Engine war [GDM11]. Zur kommerziellen Nutzung von Unternehmen bietet Unity die kostenpflichte Pro-Version an, die im Gegenzug einen stark erweiterten Funktionsumfang bereitstellt.

Neben dem italienischen Sportwagenhersteller Ferrari präsentiert auch die Volkswagen-Tochter Porsche einen Automobilkonfigurator auf ihrer Website, der durch die Unity Game

Engine entstanden ist. Die neben einer herkömmlichen 2-dimensionalen Variante vorhandene 3D-Ansicht, schöpft aus dem vollen Potential der Game-Engine. Jedes Fahrzeugmodell wird hardwarebeschleunigt in einer einfachen Showroom-Szene gerendert und ist über Menü-Punkte, ebenso wie über die Maus, im höchsten Maße modifizierbar. Sogenanntes Picking ermöglicht es interaktive Elemente des 3D-Models direkt anzuklicken, um beispielsweise Türen, Scheinwerfer oder Blinker zu betätigen. Ein erweiterter Szenerie-Modus versetzt das Auto in eine fiktive Umgebung mit Lichtern, Schatten und Spiegelungen.

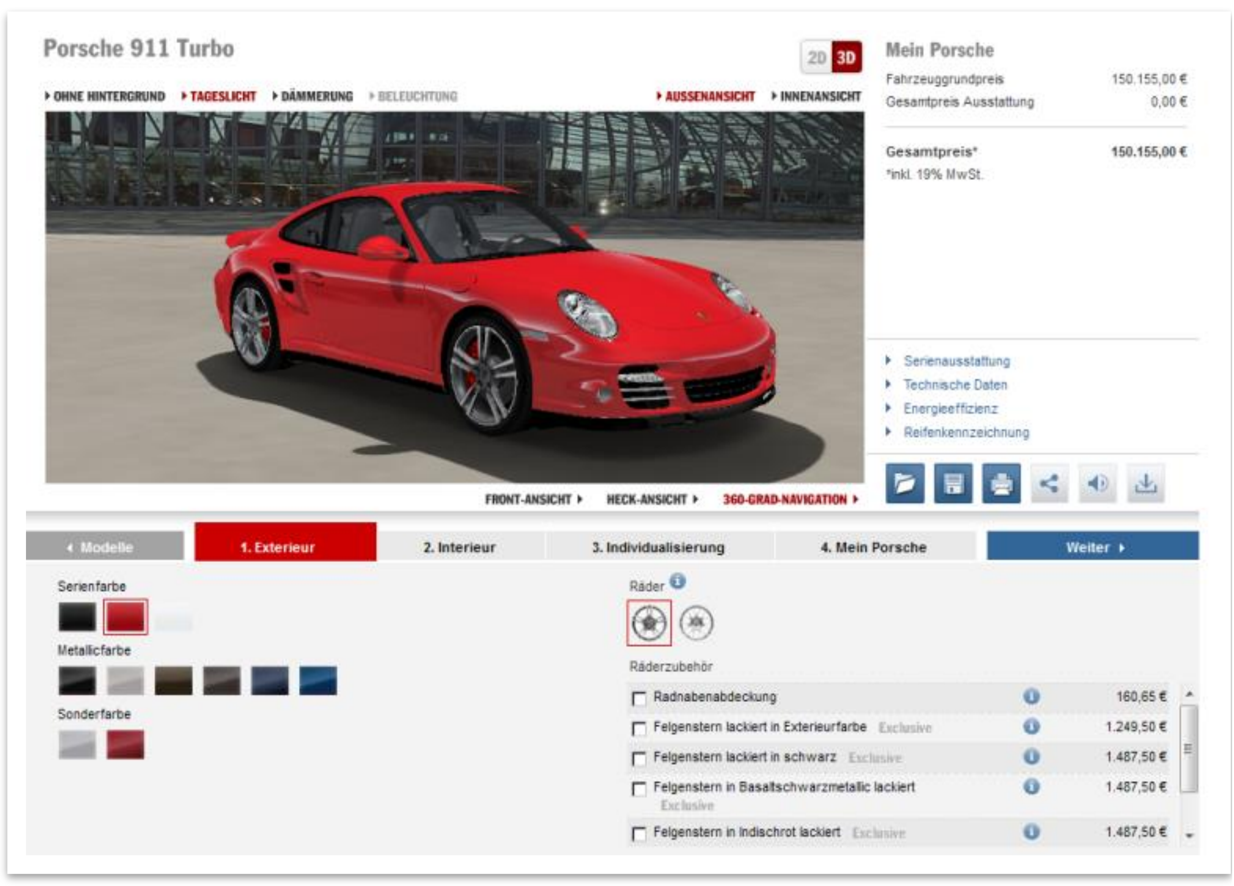


ABBILDUNG 9: PORSCHE AUTOKONFIGURATOR
QUELLE [WEB 17]

3.3. Allgemeiner Überblick

Die nachfolgende Tabelle zeigt eine Übersicht über namenhafte Automobilhersteller und deren technische Umsetzung ihrer Web-Konfiguratoren. Dabei ist in erster Linie festzustellen, dass eine hohe Nutzung der Flash-Variante vorherrscht und lediglich Porsche die Unity-Engine zur Realisierung wählt. Eine etwas interaktivere Gestaltung findet sich bei den Konfiguratoren der Marken Ferrari und Toyota wieder. Hier wird neben einer Picking-Funktion (Informationsausgabe per Click auf Fahrzeugteile) auch ein animierter Wechsel der Perspektiven ermöglicht.

Hersteller	Umsetzung
Audi	Flash mit JPEG
BMW	Flash mit JPEG
Ferrari	Flash mit animiertem Perspektivwechsel
Honda	Flash mit JPEG
Mercedes-Benz	Flash mit JPEG
Peugeot	Flash mit JPEG
Porsche	Unity Engine
Renault	Flash mit JPEG
Skoda	Flash mit JPEG
Toyota	Flash mit animiertem Perspektivwechsel
Volkswagen	Flash mit JPEG
Volvo	Flash mit JPEG

Tabelle 1: Konfiguratoren führender Automobilhersteller

4. HTML5

Bei HTML handelt es sich um eine Auszeichnungssprache zur Strukturierung und semantischen Auszeichnungen 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, die es ermöglicht, Bilder und 3D-Inhalte darzustellen. Bei letzterem handelt es sich um das Canvas-Element, das essentiell für das Rendering in WebGL ist.

Das Canvas-Element

Wichtigstes Element, gerade in Bezug auf 3D-Rendering im Browser, ist das mit HTML5 neu eingeführte Canvas-Element. Hierbei handelt es sich um eine zwei-dimensionale Zeichenfläche, die über JavaScript-Befehle angesprochen wird. Bereits im Jahr 2004 wurde die Entwicklung des Elements durch Apple als eine wichtige Komponente im MAC OS X Webkit, zur Verstärkung hauseigener Anwendungen wie dem „Dashboard" bzw. dem Safari Webbrowser abgeschlossen [Anyu12]. 2006 erweiterte Opera ihren Browser um diese Funktion, gefolgt von Mozilla (Firefox) im Jahre 2009 [Web 18]. Erzeugt wird das Canvas-Element über HTML-Code und ist somit ein Bestandteil der Website und dem zugrundeliegenden DOM-Tree (Document Object Model) wie jedes andere Website-Element auch.

```
<canvas id="mein_canvas" width="200" height="200"></canvas>
```

Das Canvas-Element ist durch ein ganzes Set an Funktionen ansprechbar, wodurch es möglich wird, dynamische generierte Grafiken zu realisieren. Im Groben sind folgende Funktionen grundlegend in der Canvas API:

- Kreisbögen
- Bézierkurven (quadratisch und kubisch)
- Farbverläufe
- Grafiken (Formate: PNG, GIF, JPEG)
- Transparenz (mit mehreren Abstufungen)
- 3D Rendering (über WebGL)

5. JavaScript

JavaScript ist eine objektorientierte, eigenständige Programmiersprache, die das Web 2.0 entscheidend geprägt hat. Durch JavaScript wurden unter anderem dynamische Anwendungen möglich, allen voran trug es aber zur Erweiterung von HTML mit optischen Effekten und Events bei. Komplexere Anwendungen traten erst wieder mit dem Aufkommen von AJAX (Asynchronous Java and XML) in den Fokus. Des Weiteren wurde es möglich, Quelltext clientseitig auszuführen, was die Server der entsprechenden Webseiten enorm entlastete. Syntaktisch ist JavaScript, wie der Name unmittelbar vermuten lässt, an Java angelehnt, distanziert sich aber in einigen Punkten sehr deutlich. Während Java selbst sehr streng typisiert ist, gibt es in JavaScript keine Variablentypen [Web2.0]. Bei immer wiederkehrende Methoden, Variablen oder ganzen Objekt-Klassen ist es empfohlen, diese in eigene Dateien, sog. JavaScript-Files, auszulagern. Dies sorgt neben einer separaten Sicherheitshandhabung der JavaScript -Dateien auch für einen besseren Überblick innerhalb des HTML-Dokuments. Darüber hinaus wird JavaScript allen objektorientierten Paradigmen gerecht. Dazu wird nicht wie in Java mit Klassen, sondern mit Prototypen gearbeitet. Im Rahmen dieser Arbeit dient JavaScript neben der herkömmlichen Verwendung auch als Mittel zur Kommunikation mit WebGL, sowie der Eventbehandlung der Mausinteraktion durch den Nutzer.

6. Computergrafik-Grundlagen

In den folgenden Kapiteln wird ein allgemeiner Überblick über die Grundlagen der Computergrafik gegeben. Die zur Verdeutlichung dienenden Beispiele stehen dabei in enger Verknüpfung mit dem zu implementierenden Automobilkonfigurator.

6.1. Geometrie-Informationen

WebGL behandelt Geometrien jeglicher Form unabhängig von ihrer Komplexität und der Anzahl der Vertices, aus denen eine Oberfläche bestehen kann. Fundamental sind zwei Datentypen, die die Geometrie jedes 3D-Objekts beschreiben, Vertices und Indices.

Vertices sind dabei die Punkte eines 3D-Objekts, welche die Eckpunkte der Geometrie darstellen. Jeder Vertex wird durch ein Koordinaten-Tripel beschrieben, bestehend aus drei Gleitkommazahlen (floating point), welches die x-, y- und z-Position des Punkts im Raum beschreibt. Gegebenenfalls ist eine vierte Komponente vorhanden, die als homogene Koordinate bezeichnet wird. Diese wird für spätere Transformationen mit 4x4-Matrizen wichtig.

Um Vertices an die Rendering-Pipeline zu übergeben, werden alle Vertices in ein JavaScript Array geschrieben, aus dem dann ein WebGL Vertex Buffer Object erstellt wird (Abbildung 10). Gleichzeitig ist dies auch ein entscheidender Nachteil, denn sehr große JavaScript Arrays können je nach Auslastung des Systems zu Problemen in der Darstellung und Verarbeitung führen. Unter **Indices** versteht man eine numerische Beschriftung jedes einzelnen Vertex in einer 3D-Szene. Über die Indices kann WebGL vermittelt werden, in welcher Reihenfolge die einzelnen Punkte verbunden werden sollen, um letztlich die Dreiecksoberflächen der Geometrie (Faces) zu erzeugen. Die Arten dieser Verbindungen werden im nachfolgenden Abschnitt „Primitive-Erzeugung“ beschrieben. Ebenso wie die Vertices, werden auch alle Indices in ein JavaScript Array gespeichert, bevor sie in einem WebGL Index Buffer an die WebGL Rendering-Pipeline übergeben werden.

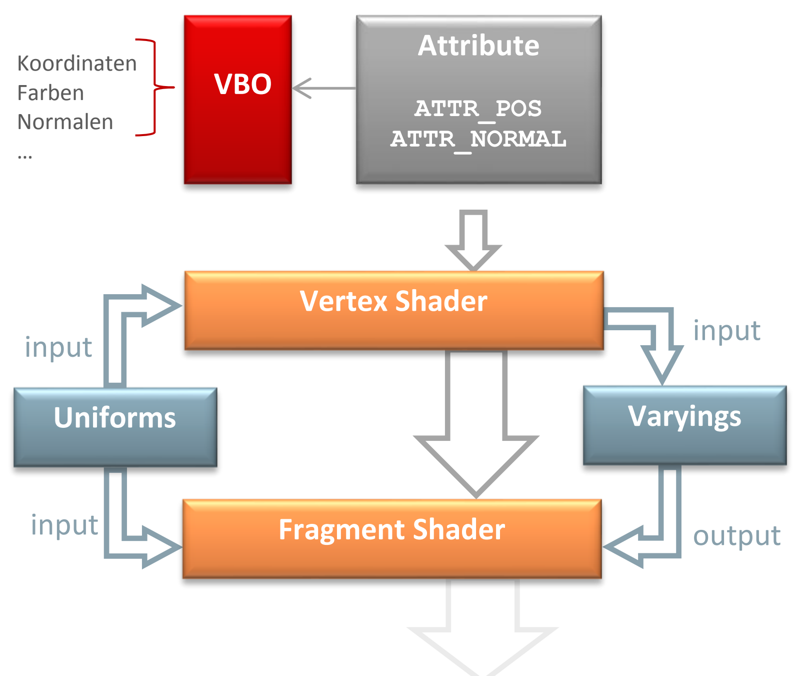


Abbildung 10: Ausschnitt der WebGL-Pipeline

6.2. Primitive-Erzeugung

Um nun die zuvor definierten Punkte zu einem Polygon bzw. komplexen Flächen zu verbinden, bedarf es der Erzeugung von sogenannten Primitives. Dies sind Punkte, Linien und Dreiecken, also einfachste geometrische Gebilde, aus denen höchst komplexe Polygone konstruiert werden können.

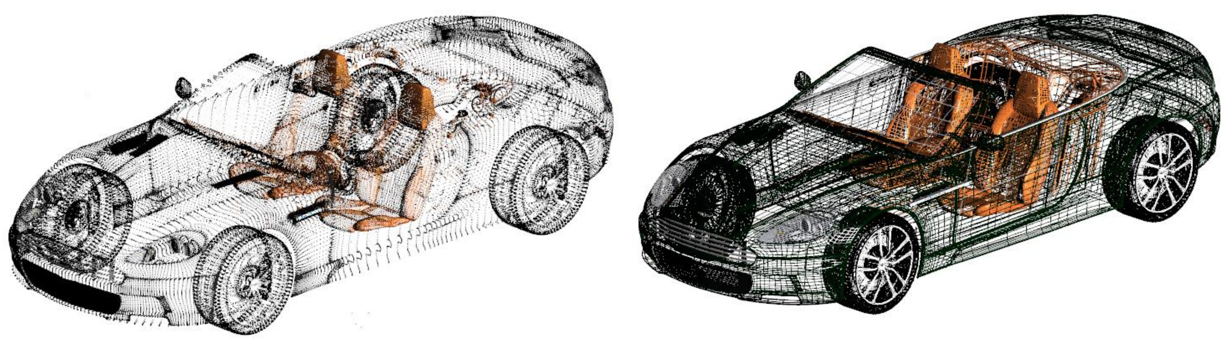


Abbildung 11: gl.POINTS   Abbildung 12: gl.LINES

GL.POINTS, GL.LINES

Grundständige Primitives sind Punkte und Linien, die über die WebGL-Bibliothek mit den Ausdrücken `gl.POINTS` und `gl.LINES` ausgeführt werden. Während wie in Abbildung 11 dargestellt jeder Vertex als Punkt gerendert wird, lässt sich über den LINES-Befehl ein Drahtgitter-Modell (Wireframe) der Geometrie erzeugen. Die Liniendarstellung einer Geometrie findet insbesondere bei der CAD-Konstruktionsabbildung Verwendung, eignet sich aber auch ausgesprochen gut zur Fehleranalyse bei defekten Indexbuffern, da diese durch falsche Geometrie-Verbindungen schnell auffallen.

GL.TRIANGLES

Abbildung 13: gl.TRIANGLES

Die einfachste aller geometrischen Flächen ist das Dreieck. Dies beruht auf der Tatsache, dass jede Ebene, beginnend beim Viereck, auf eine Zerlegung in Dreiecke zurückzuführen ist. Speziell bei der Modellierung von CAD-Modellen kommen komplexe Formen wie Nurbs zum Einsatz, die später zu Quadraten bzw. Dreiecken konvertiert werden. Über entsprechende

Befehle werden diese Erzeugnisse an Hand ihrer Indizierung später im Rahmen der Primitive-Assembly (Stufe der Graphics Pipeline) verbunden. Entsprechend der Vertex-Farben wird die Farbe des aufgespannten Dreiecks interpoliert und gerendert. Bei der Verwendung von 3D-Modellierungsprogrammen zur Generierung eines Fahrzeug-Modells ist darauf zu achten, dass das Gebilde triangulisiert werden muss, da sich an dieser Konvention die Indizierung orientiert. Dieser Zwischenschritt erspart entscheidende Arbeit, da andernfalls die Indizes manuell sortiert werden müssten, was in Anbetracht der Größenordnungen komplexer 3D-Modelle eine enorme Herausforderung darstellt.

6.3. Projektions-Matrizen

Im nächsten Schritt muss die 3D-Szene auf die 2-dimensionale Bildschirmfläche abgebildet werden. Für diesen Vorgang werden sogenannte Projektionsmatrizen verwendet. Ein wichtiger Aspekt der Projektion ist, dass bei der Abbildung eines Raums in eine Ebene die räumliche Dimension wegfällt. Dies muss bei der Erstellung der Projektionsmatrizen berücksichtigt werden, da trotzdem eine räumliche Verjüngung erzielt werden soll, damit der räumliche Effekt erhalten bleibt und Geometrien entsprechend perspektivisch verzerrt werden. Außerdem darf die z-Koordinate des Raumes nicht vollends vernachlässigt werden, da sie die entscheidende Komponente der Verdeckungsberechnung sein wird. Die in dieser Arbeit zum Einsatz kommenden Projektionsmatrizen werden im Folgenden näher erklärt.

Orthogonale Projektion

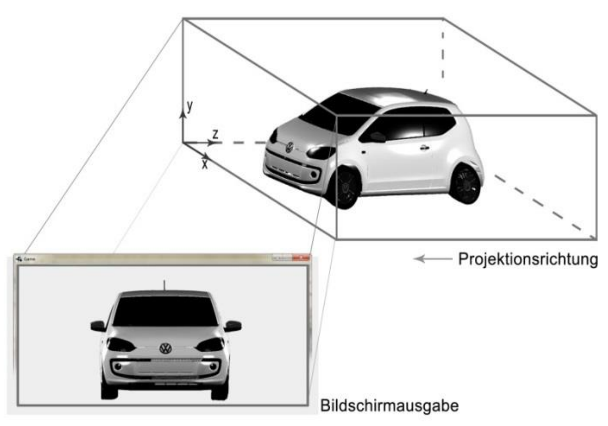


Abbildung 14: Orthogonale Projektion

Die orthogonale Projektion entsteht durch parallele Strahlen, die von jedem Objekt im Raum auf die Projektionsfläche geschossen werden. Durch dieses Vorgehen bleiben sämtliche Größen und Winkel erhalten, unabhängig von ihrer Entfernung zur Projektionsfläche. Dies stellt einen wesentlichen Grund für die Verwendung der Projektionsform,

sowohl im CAD-Bereich (*computer aided design*) als auch bei technischen Zeichnungen für die Erzeugung der Vorder-, Seiten- und Draufsicht, dar. In Bezug auf den sichtbaren Bereich der 3D-Szene, der letztenendes auf dem Bildschirm des Betrachters zu sehen ist, wird ein Würfel (*view volume*) aus der Gesamtszene herausgeschnitten (siehe Abbildung 14), wobei sämtliche Elemente außerhalb dieses Kubus nicht zu sehen sind. Die Grenzflächen sind die sogenannten c*lipping planes*.

Um im Folgenden die Matrix der orthogonalen Projektion erstellen zu können, werden die Punkte der Begrenzungsflächen wichtig, die den sichtbaren Würfelbereich darstellen. Hierbei handelt es sich um ein 6-Tupel mit den Werten

`left, right, bottom, up, near, far`

welches bei der Erzeugung der Matrix übergeben wird. An dieser Stelle ist darauf zu achten, dass `(left-right) / (top-bottom)` dem Seitenverhältnis des Ausgabegeräts entspricht, um unvorhergesehenen Verzerrungen zu vermeiden. Die Projektionsmatrix selbst ist das Produkt der Translation, sowie der Skalierung des sichtbaren Bereichs in das kanonische View Volume. Die Projektionsmatrix ist definiert durch:

Elemente der Skalierungsmatrix

$$\begin{pmatrix} 2/r-l & 0 & 0 & -l+r/r-l \\ 0 & 2/t-b & 0 & -t+b/t-b \\ 0 & 0 & -2/f-n & -f+n/f-n \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Elemente der Translationsmatrix

Für die genaue Zusammensetzung der Matrizen sei an dieser Stelle auf die Literatur-Quelle [CG12] verwiesen.

Perspektivische Projektion

Entgegen der Verwendung von parallelen Strahlen, die jedes Objekt auf die Bildfläche projizieren, ist die perspektivische Projektion eine Betrachter-zentrierte Perspektive, deren Strahlen einer gewissen Fluchtung unterliegen. Somit wird ein unserer natürlichen Wahrnehmung entsprechendes räumliches Abbild einer Szene ermöglicht. Dies liegt nicht zuletzt auch daran, dass die Bildentstehung auf der Netzhaut des menschlichen Auges oder

in einer Fotokamera nach denselben Regeln der perspektivischen Projektion funktioniert [CGuB1].

Die folgende Abbildung 15 vereinfacht die Systematik hinter der Projektion:

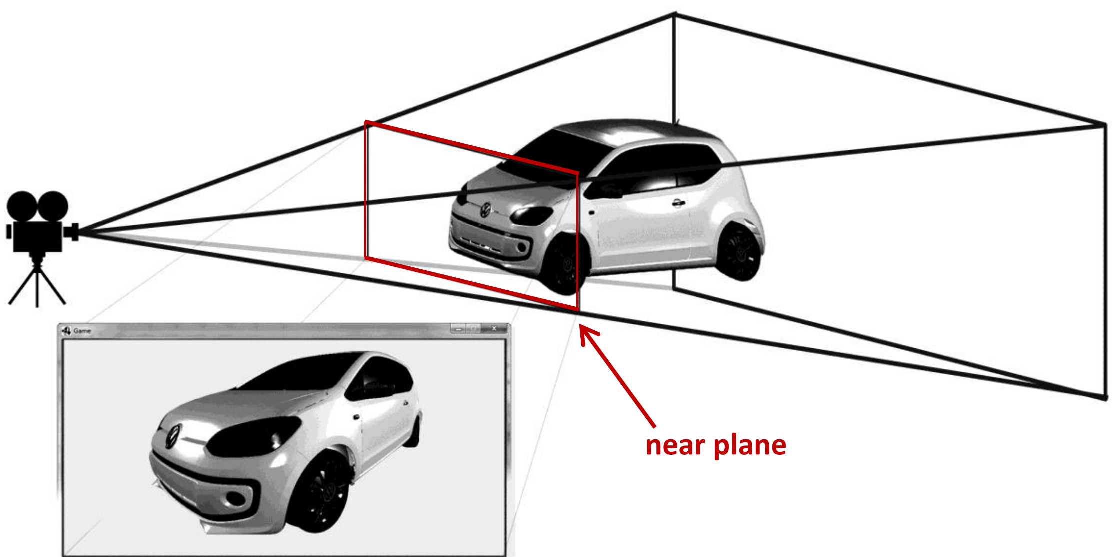


ABBILDUNG 15: PERSPEKTIVISCHE PROJEKTION

Während bei der orthogonalen Projektion ein Würfel zur Abgrenzung des sichtbaren Bereichs aus dem Gesamtraum herausgeschnitten wurde, entsteht bei der perspektivischen Projektion eine liegende Pyramide, dessen Spitze das Auge des Betrachters darstellt. Die sogenannte *near plane* legt mit ihrem Abstand zum Betrachter den Öffnungswinkel fest und kann bei entsprechender Wahl eher den Effekt eines Teleobjektivs (großer *near* Wert) oder eines Weitwinkelobjektivs (kleiner *near* Wert).

In Übereinstimmung mit der Matrix der orthogonalen Projektion wird gleichermaßen bei der Matrix der perspektivischen Projektion das bereits erwähnte 6-Tupel übergeben und in die folgende Matrix eingepflegt:

$$\begin{pmatrix} 2n/r-l & 0 & r+l/r-l & 0 \\ 0 & 2n/t-b & t+b/t-b & 0 \\ 0 & 0 & -f+n/f-n & -2fn/f-n \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Für weitere Details zur Zusammensetzung der Matrix sei an dieser Stelle ebenfalls [CG12] verwiesen.

6.4. Graphics-Pipeline

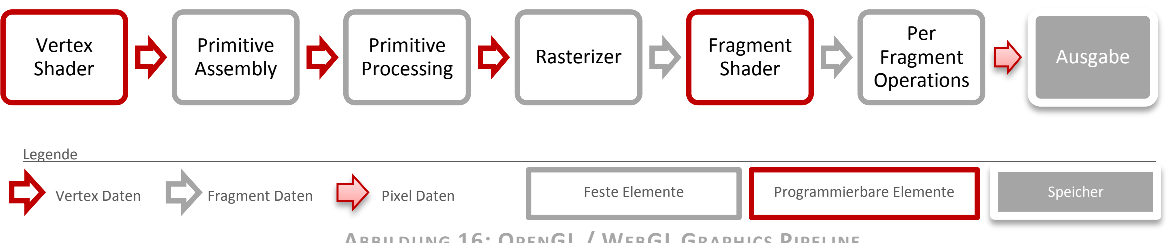


Abbildung 16: OpenGL / WebGL Graphics Pipeline
Quelle: [CG12]

Die Graphics-Pipeline stellt die Reihenfolge der Operationen dar, die benötigt werden, um eine mathematisch beschriebene Szene in ein Bild umzusetzen. Zunächst bilden alle Vertices zusammen eine komplexe Geometrie in Modell-Koordinaten, welche über den **Vertex Shader** entsprechend im Raum positioniert, verändert und so in die Welt-Koordinaten überführt wird. Der Vertex Shader als Teil der Shadereinheiten auf dem Grafikprozessor einer Grafikkarte, kann lediglich Geometrien manipulieren, nicht aber Neue hinzufügen bzw. entfernen. Dafür wurde der Geometry Shader (OpenGL 3.2) eingeführt.

Nachdem die Vertices im Vertex Shader vollkommen unabhängig voneinander verarbeitet wurden, kommen nun im Bereich der **Primitive-Assembly** die topologischen Informationen hinzu. Entsprechend der Identifier `gl.POINTS`, `gl.LINES`, `gl.TRIANGLES`, etc. werden so die übergebenen Vertices in Abhängigkeit ihrer Indices zu Primitives verbunden um im nächsten Schritt, dem **Primitive-Processing**, weiter verarbeitet zu werden. Hier kommt unter anderem das Clipping (engl.: *to clip* = „abschneiden“, „kappen“) der Polygone zur Anwendung. Die Clipping-Ebenen (nahe und ferne Ebene) stellen das sichtbare Teilvolumen in Blickrichtung dar, wobei lediglich die Polygone innerhalb dieses Volumens vom Betrachter erfassbar sind. Des Weiteren kommt das Culling (engl.: to cull = „wegschneiden“, „Abfall“) zum Einsatz. Dieser Algorithmus prüft Polygone, ob sie mit der Vorderseite zum Betrachter zeigen. Dies kann über die Orientierung der Vertices eines Primitives analysiert werden, da die Konvention vorsieht, dass die Orientierung von Seiten, die auf den Betrachter gerichtet sind, gegen den Uhrzeigersinn initialisiert sind. Hierbei ist auf eine konsistente

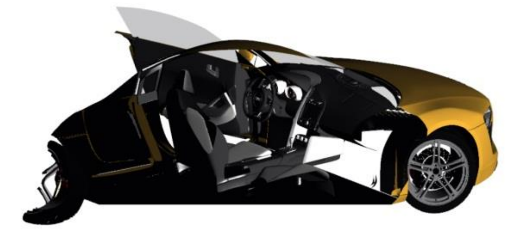


Abbildung 17: Clipping Ebene in 3D Modell

Polygon-Orientierung zu achten, damit alle Außenflächen eines Objektes auch außen dargestellt werden.

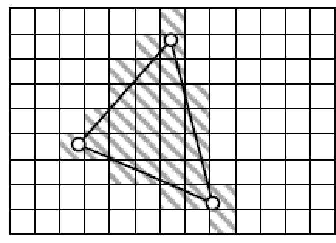


ABBILDUNG 18: RASTERISIERUNG
QUELLE: [CG12]

Der **Rasterizer** als Teil der Graphics- Pipeline sorgt für die Generierung der Fragments, die für den weiteren Verlauf zur Erzeugung von Rastergrafiken notwendig sind. Fragments sind diejenigen Bereiche, die von Primitives überlappt und später zu Bildpixeln werden.

Der letzte programmierbare Teil der Graphics-Pipeline ist der **Fragment-Shader**. Je nach Implementation ist es die Aufgabe des Fragment-Shaders alle rasterisierten Fragments hinsichtlich ihrer Oberflächen- und Materialeigenschaften oder Texturen zu modifizieren. Häufige Anwendungsfälle sind unter anderem die Implementation des Phong-Lighting Modells, auf das in Kapitel 6.5 eingegangen wird, sowie Spiegelungen, Bloom- oder Lens Flare Effekte (Post Processing). Es kann der Fall eintreten, dass ein Pixel des gerenderten Endergebnisses unter Umständen aus mehreren Fragments besteht. Dieser Zustand wird durch die parallele Existenz verschiedener Transparenzebenen möglich, um räumlich verteilte Objekte gleichzeitig sichtbar werden zu lassen.

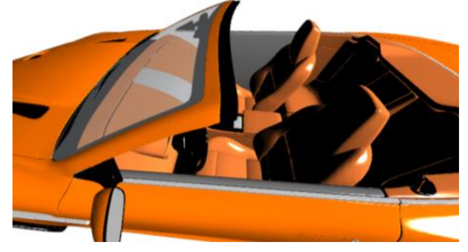


ABBILDUNG 19: BLENDING

Im letzten Teil der Pipeline, den **Per Fragment Operations**, werden die generierten Fragments einem Tiefentest (Depth Test) unterzogen. Dieses Verfahren wird pro Pixel sequentiell für alle Fragments ausgewertet und ermöglicht so das Ausblenden von verdeckten Objekten. Im Anschluss werden alle Fragments, die diesen Tiefentest passiert haben dem Blending unterzogen, um korrekte Farbwerte bei gegebenen Transparenzen zu berechnen. Die Funktionsweise des Blendings beruht darauf, dass eine vom Programmierer gewählte Blending-Funktion mit Quell- sowie Zielfarbwerten gefüllt und ausgewertet wird. Die Quellfarbe ist die bereits für einen Pixel eingetragene Farbe (destination). Diese wird entsprechend der Blending-Funktion mit dem neu hereinkommenden Farbwert (source) vermischt und ermöglicht so beispielsweise die Darstellung von Objekten die hinter einer Glasscheibe liegen (siehe Abbildung 19).

Am Ende der Pipeline steht der Framebuffer, auch Bildspeicher genannt, in den alle Farbwerte pro Pixel geladen werden, die letztlich im Ausgabefenster auf dem Bildschirm zu sehen sind. Für genauere Details bezüglich dem Aufbau und der Funktionsweise, sei an dieser Stelle auf das Kapitel *6.4.4.4 Frame Buffer Objects*, des Buches *„Computergrafik und Bildverarbeitung Band I* von Alfred Nischwitz verwiesen, in dem diese Thematik entsprechend erläutert wird. Wie im Kapitel der Graphics-Pipeline erwähnt, folgen im nächsten Abschnitt die Grundlagen des Phong Shading Models, das im Fragment Shader des Automobilkonfigurators verwendet wird.

6.5. PHONG-LIGHTING

Das Phong Beleuchtungsmodell beschreibt die Art und Weise, wie die Oberflächen-Normalen, Materialien von Objekten und in die Szene integrierte Lichter kombiniert werden. In dieser Beleuchtungserrechnung finden sich die physikalischen Prinzipien wieder, weshalb man oft auch von Lichtreflektions-Modellen spricht. Das Phong-Beleuchtungsmodell beschreibt somit die Reflektion des Lichtes auf der Oberfläche eines Gegenstandes in Abhängigkeit seiner Materialien sowie dem einfallenden Licht, als Summe dreier verschiedener Reflektionskomponenten [CGuB1] (siehe Abbildung 20).

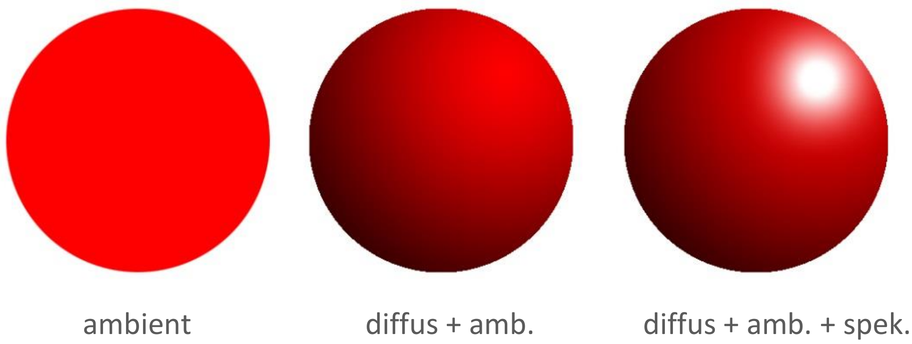


ABBILDUNG 20: PHONG KOMPONENTEN

Ambienter Anteil

Der ambiente Lichtanteil, stellt das gestreute Licht der Umgebung dar und bildet eine Art globale Beleuchtungskomponente, da sie unabhängig von allen Einflussgrößen wie Lichtquellen, Materialkonstanten und Einfallswinkeln ist.

$$Farbe \mathrel{+}= Licht_{ambient} \cdot Material_{ambient}$$

Diffuser Anteil (Abbildung 21)

Die diffuse Komponente spiegelt den Anteil des Lichts wieder, der gleichmäßig in alle Richtungen reflektiert wird, unabhängig vom Standpunkt des Betrachters. Jedoch ist die Intensität der Reflektion abhängig von der Position des Lichts und dem damit verbundenen Einfallswinkel der Punktlichtquelle. Ebenso liegt eine Abhängigkeit von den Materialkonstanten (Reflektionsfaktoren) vor, die beschreiben wieviel Licht absorbiert bzw. reflektiert wird. Die mathematische Berechnung der diffusen Komponente macht sich dabei die Eigenschaften des Skalarprodukts von Vektoren zu nutze. Licht wird umso stärker reflektiert, je paralleler ein Lichtstrahl konträr zur Richtung der Normalen verläuft. Durch Einschränkung auf das Intervall [0,1] wird ein Faktor zwischen Parallelität (0) und Orthogonalität (1) ermittelt der die Intensität der diffusen Reflektion an der jeweiligen Stelle bestimmt.

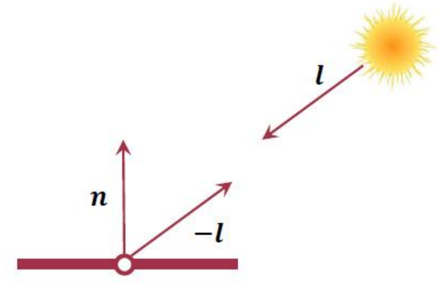


ABBILDUNG 21: DIFFUSE REFLEXIONSKOMPONENTE QUELLE: [CG12]

$$Farbe += Licht_{diffuse} \cdot Material_{diffuse} \cdot clamp(dot(n, -l), 0,1)$$

Spekularer Anteil (Abbildung 22)

Die spekulare Komponente ermöglicht die Darstellung einer spiegelnden Lichtreflektion auf einem Gegenstand. Innerhalb einer kleinen Umgebung (in Abbildung 22 liegt *v* darin) um den idealen Reflektionswinkel (Einfallswinkel = Ausfallswinkel) wird diese spekulare Reflektion „r“ in voller Intensität vom Betrachter wahrgenommen. Somit ist sie abhängig von der Position der Lichtquelle, dem Einfallswinkel des Lichtstrahls „l“ und der Position des Betrachters „v“, der einen gewissen Blickwinkel auf das Objekt hat. Zudem hat die Oberflächenbeschaffenheit einen entscheidenden Einfluss auf eine „störungsfreie“ Spiegelung. Betrachtet man beispielsweise die spekulare Reflektion der Sonne auf dem Meer, so sorgen Wellen für eine störende

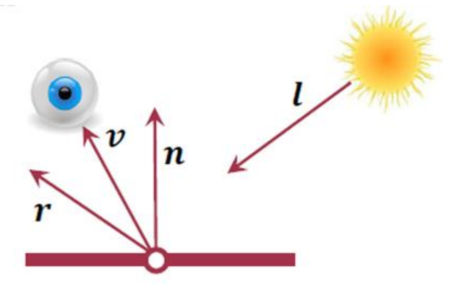


ABBILDUNG 22: SPEKULARE REFLEXIONSKOMPONENTE QUELLE: [CG12]

Wirkung, ebenso wie eine Lampe, die auf nasses Mauerwerk leuchtet (siehe Abbildung 23). Betrachtet man die Formel zur Berechnung wird auch hier deutlich, dass äquivalent zur diffusen Komponente, ebenfalls mit dem Skalarprodukt der Anteil des Winkels zwischen reflektiertem Lichtstrahl und Augenposition des Betrachters berechnet wird. Neu hinzu kommt der Glanzfaktor (*engl.: shininess*) der angibt, wie glänzend bzw. matt die Oberfläche wirkt. Ein höherer Glanzfaktor erzeugt den Eindruck einer plastikartigen Oberfläche, je niedriger dieser Wert ist, desto diffuser ist die Reflektion an dieser Stelle.

ABBILDUNG 23: SPEKULARE REFLEKTIONEN

$$Farbe \mathrel{+}= Licht_{specular} \cdot Mat_{specular} \cdot (dot(reflect(L,N), Pos_{Eye}))^{Mat_{shininess}}$$

Die Summe aller beschriebenen Komponenten stellt den Farbwert dar, der dem Fragment an der entsprechenden Stelle zukommt. Kommen mehrere Punktlichtquellen in der Szene vor, werden alle drei Komponenten separat für jede Lampe berechnet und am Ende aufsummiert.

6.6. LICHTQUELLEN

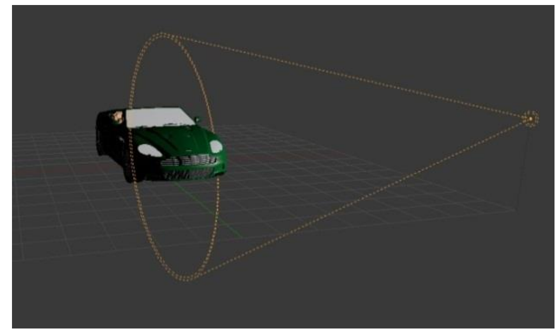


ABBILDUNG 24: PUNKTLICHT

Ohne die Reflektion von Licht auf den Oberflächen von Gegenständen würde man praktisch nichts sehen. Dabei wird das Licht je nach Farbe, Form und Beschaffenheit der Oberfläche unterschiedlich zurückgeworfen und für das menschliche Auge sichtbar. Wie an der Lichtreflektion des Fahrzeugs in Abbildung 24 zu erkennen ist, wird ein Teil stärker beleuchtet als die restlichen Teilbereiche. Betrachtet man beispielsweise das Licht der Sonne, so stellt man fest, dass sämtliche Lichtstrahlen parallel auf der Erde auftreffen und somit eine gleichmäßig Ausleuchtung stattfindet. Man

spricht von gerichtetem Licht (*engl.: directional light*). Anders ist es nun bei Punktlichtquellen (*engl.: point lights*), die auch in der 3D-Szene des Automobilkonfigurators verwendet werden. Hier geht man von einer Lichtquelle aus, die in einer festen Entfernung ihre Strahlen, abhängig von einem Öffnungswinkel, in die Szene schießt. Einer der ausgesendeten Lichtstrahlen trifft dabei genau senkrecht auf einen Flächenbereich und leuchtet diesen vollständig aus. Viele Strahlen treffen annähernd senkrecht auf und abzählbar viele verteilt auf der gesamten Fläche des beleuchteten Objekts. Dieses Lichtkonzept, dessen Lichtintensität jenach Auftreffwinkel der Strahlen variiert (siehe Abbildung 24), wird durch die Normalen der Dreiecksflächen gesteuert, aus denen jedes polygonale Objekt besteht. Normalen charakterisieren zum einen die Orientierung einer Fläche, womit sichergestellt werden kann, ob eine Fläche zum Betrachter zeigt oder nicht. Zum anderen kann sie zur Berechnung des Winkels zwischen reflektiertem Strahl und Oberfläche herangezogen werden. Ist dieser 90°, so wird der getroffene Punkt zu 100% ausgeleuchtet.

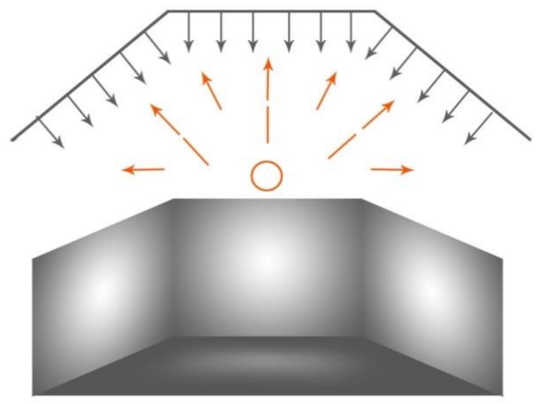


ABBILDUNG 25: POINT LIGHT
QUELLE: [TB13]

Die geometrische Deutung sowie das gerenderte Endergebnis wird in der nebenstehenden Abbildung 25 dargestellt. Deutlich zu erkennen ist die hohe Lichtintensität in den Mitten der Wände, da dort die Lichtstrahlen senkrecht auftreffen. Im näheren Umfeld herrscht ein gedämmter Eindruck, da jegliche Orthogonalität der einfallenden Lichtstrahlen aufgehoben ist.

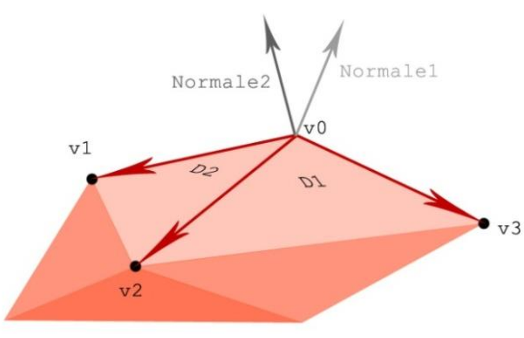


ABBILDUNG 26: NORMALEN AUF OBERFLÄCHE

Die Normale, also der Vektor der orthogonal auf der Fläche steht, lässt sich über das Kreuzprodukt der zwei Vektoren berechnen, die das jeweilige Dreieck aufspannen. Auf mathematische Hintergründe des Kreuzprodukts wird nicht näher eingegangen. Das Ergebnis spiegelt sich in Abb. 26 wieder in dem für die Dreiecke die entsprechenden Normalenvektoren gebildet worden sind.

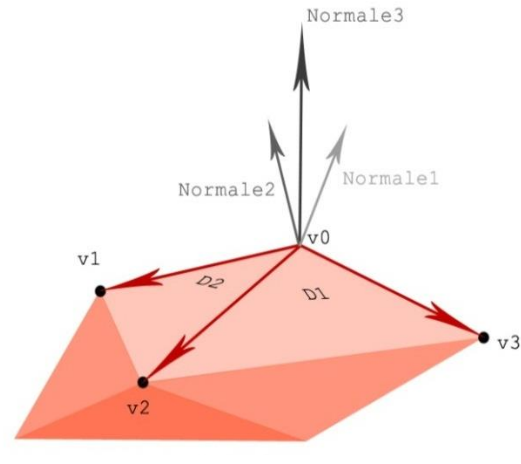


ABBILDUNG 27: INTERPOLIERTE NORMALE

Ein wichtiger Punkt der nicht außer Acht gelassen werden darf behandelt die Vertices, die von Dreiecken gemeinsam verwendet werden. Hier ist im allgemeinen nicht klar, welche Normalen von welcher Dreiecksfläche (z.B. der Punkt v0 in Abbildung 27) verwendet werden soll. In diesen Fällen wird eine interpolierte Normale (Normale3 in Abbildung 27) ermittelt, die durch Addition der Normalen, der beiden anliegenden Flächen ermittelt wird.

7. WEBGL

7.1. SYSTEM-VORAUSSETZUNGEN

Neben einer Grafikkarte die OpenGL (> Version 2.0) unterstützt sind keine weiteren Systemvoraussetzungen notwendig. WebGL hat als web-basierte 3D Grafik API lediglich spezielle Anforderungen an den verwendeten Browser. So ist darauf zu achten, dass Firefox ab der Version 4.0, Google Chrome ab Version 11, Safari ab OSX 10.6, oder Opera ab Version 12 verwendet wird. Zudem müssen Mac-Nutzer in ihren Browser-Einstellung das entsprechende WebGL-Flag in der Konfiguration auf `enabled` schalten, da dies standardmäßig `disabled` ist.

Für eine aktuelle Auflistung aller WebGL-fähigen Browser sei an dieser Stelle auf die Webseite der Khronos Gruppe verwiesen:

http://www.khronos.org/webgl/wiki/Getting_a_WebGL_Implementation

Für die Überprüfung der WebGL-Funktionalitäten des eigenen Systems ist der folgende Link hilfreich.

http://get.webgl.org/

7.2. Struktur einer WebGL Applikation

Um in den folgenden Kapiteln der Arbeit näher auf die „Anatomie“ des Automobilkonfigurators eingehen zu können wird an dieser Stelle eine kurze Übersicht geboten, die die wesentlichen Bestandteile der Struktur einer WebGL-Applikation zusammenfasst.

- **Canvas Element**
  Wie in den Einführungskapiteln erwähnt ist das Canvas-Element die Zeichenfläche, in der die zu rendernde Szene dargestellt wird. Mit HTML5 fand dieses Element erstmals im Internet Verwendung und kann über das Document Object Model via JavaScript angesprochen werden.

- **3D Objekte**
  Eine Szene ist charakterisiert durch ihre Objekte. Der Automobilkonfigurator beinhaltet dabei einen Showroom, konstruiert durch einfache geometrische Formen (Quader, Zylinder) in den verschiedene 3D-Fahrzeugmodelle geladen und konfiguriert werden. Im Kapitel 11.1 – Die Szene, sowie der Beschreibung der Funktionen auf Seite 65 werden diese Objekte näher erläutert.

- **Lichter**
  Ohne die Interaktion mit Lichtquellen wären sämtliche Geometrien in der 3D-Szene unsichtbar. Je nach Beleuchtungsmodell verändert sich die Beleuchtung der 3D Objekte in Abhängigkeit ihrer mitgeführten Materialien, welche gerade durch das Phong-Lighting Modell sehr stark in den Fokus gerückt werden.

- **Kamera**

  Speziell in WebGL wird das Canvas Element mit dem Viewport (Sichtfläche) gleichgesetzt. Das Canvas Element ist also die sichtbare Fläche die durch ein Objektiv einer Kamera zu sehen ist und unterliegt somit gewissen perspektivischen Regeln und Einstellungen (siehe Abbildung 28).

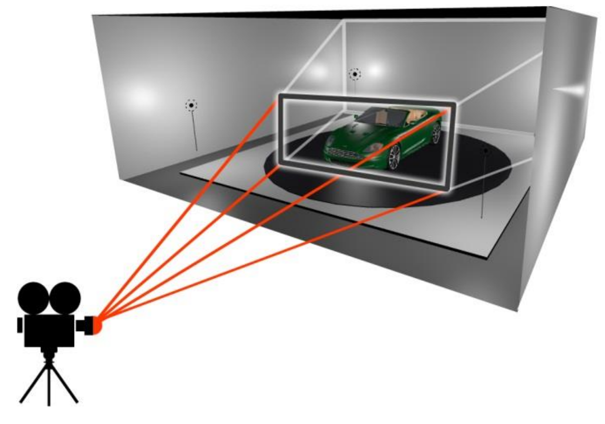


ABBILDUNG 28: KAMERA FRUSTUM

7.3. DER WEBGL-KONTEXT

Eingebettet in den HTML Code einer Website, muss der WebGL-Kontext über JavaScript explizit deklariert werden, damit dieser im Canvas Element korrekt dargestellt und verarbeitet werden kann. Neben den nötigen Skripten, die im Verlauf der Arbeit näher erläutert werden (siehe Kapitel 12.2) und für die Initialisierung von WebGL von großer Bedeutung sind, muss zunächst das `<canvas>` Element im `<body>` der HTML-Seite definiert werden.

```
<body onload="main()">

    <canvas id="mein_canvas" width="640" height="480">
    Dieser Text wird angezeigt wenn der Browser
    kein Canvas-Element unterstützt!
    </canvas>

</body>
```

In Verknüpfung mit dem JavaScript Event `onload` wird die Beispielmethode `main()` ausgeführt, die im weiteren Verlauf alle Funktionen der WebGL-Anwendung aufruft, u.a. die `initGL()` Funktion zur Initialisierung des WebGL-Kontexts.

7.4. Initialisierung

```
function main(){

     var canvas = document.getElementById(„mein_canvas“);
     initGL(canvas);

}
```

Neben weiteren wichtigen Funktionen die essentiell für die Main-Methode des Automobilkonfigurator sind, wird sich hier lediglich auf die wichtigen Initialisierungs-Methoden beschränkt. Offensichtlich ist, dass eine Variable `canvas` angelegt wird, der das im HTML-Code definierte Canvas-Element zugewiesen wird. Sollte im weiteren Verlauf auf die Variable `canvas` zugegriffen werden, muss die Variable an dieser Stelle global angelegt werden. Für eine höhere Codesicherheit ist davon an dieser Stelle jedoch abzuraten, da in der Regel dies die einzige Verwendung der Variable `canvas` ist.

```
function initGL(canvas){
     var gl = null;
     try{

          gl = canvas.getContext(“experimental-webgl”) ||

               canvas.getContext(“webgl”);

          gl.viewport(0, 0, canvas.width, canvas.height);
          gl.clearColor(0.0, 0.0, 0.0, 1.0);

     }catch(e){}

     if(!gl){
          alert(“WebGL konnte nicht initialisiert werden!”);

     }

}
```

Entscheidend für die Initialisierung ist anschließend der oben angeführte Quellcode der `initGL()` Methode. Wie bereits in den einführenden OpenGL Kapiteln erwähnt, beginnen sämtliche Funktionen und Parameter mit „`GL`". Um die Ähnlichkeit dieser Syntax sicherzustellen, wird hier eine Variable `gl` deklariert, die standardmäßig mit `null` belegt ist. Über die `getContext` Methode der `canvas` Variable, die als Übergabeparameter einen `DOMString` erhält, wird aus den im Repertoire vorhandenen Kontexten mit `webgl` bzw. `experimental-webgl` der WebGL-Kontext ausgewählt. Je nach Parameter wird ein anderes Funktionensortiment freigeschaltet und für das Canvas Element verwendbar. Speziell für den WebGL-Kontext existieren zwei Parameter, welche jedoch auf `webgl` reduziert werden, sobald die WebGL Spezifikation ihren finalen Stand erreicht hat. Das Resultat dieser Initialisierungs-funktionen spiegelt sich auf der HMTL-Seite, entsprechend der Abbildung 29, wieder.

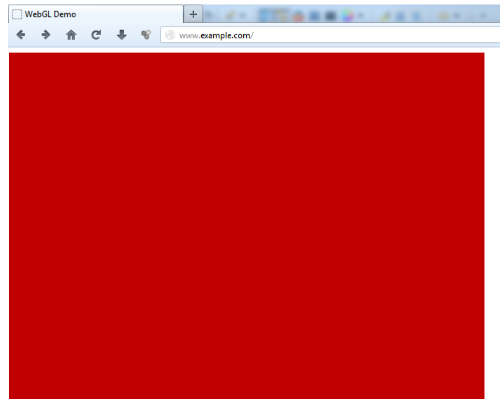


ABBILDUNG 29: WEBGL KONTEXT

7.5. VERGLEICH OPENGL – WEBGL

Um die Grundlagen der vorherigen Kapitel zu konkretisieren, liegt im Folgenden der Fokus auf einem direkten Vergleich der Geometrie-Erzeugung mittels WebGL und OpenGL 3.0. Grundlegend existiert für fast jede Programmiersprache ein sogenanntes „Hello-World"-Programm, das einen kurzen, prägnanten Einblick in die Syntax und Semantik einer Programmiersprache gewähren soll. Unter dem Aspekt, dass der Fokus sowohl in WebGL als auch OpenGL, auf dem Rendering von 3D-Modellen liegt, wird für den folgenden Funktionscodevergleich das „Hello-Triangles" Programm implementiert. In diesem werden alle grundlegenden Funktionen, die auch komplexere Programme benötigen, eingearbeitet.

Wichtig ist hierbei zu wissen, dass die Implementation des OpenGL-Programms auf der Programmiersprache Java unter Einbezug der Wrapperklasse LWJGL basiert. Deshalb können speziell Funktionsnamen in ihrem Ausdruck von Funktionen in C oder C++ abweichen. OpenGL-Funktionen beginnen mit „gl" vor dem Funktionsnamen, während WebGL-Funktionen entsprechend nach dem Variablennamen des Canvas-Elements genannt werden. Für einen vergleichbareren Code wurde dieses hier mit „gl." deklariert.

Die in den Shadern verwendeten Variablen unterscheiden sich ebenfalls in der Bezeichnung – in Vertex Shadern unter OpenGL finden `in`- bzw. `out`-Variablen Verwendung, in WebGL (Shader-Versionen nur bis 1.10 möglich) entsprechend `attributes` und `varying`-Variablen. Zudem werden die `in`-Variablen des Fragment-Shaders unter WebGL ebenfalls als `varying` bezeichnet. Um einen einfachen, auf das wesentliche reduzierten Code zu ermöglichen, werden in diesem Beispiel weder eine Matrix für eine dynamische Kamera (View-Matrix) noch eine Matrix für die perspektivische Projektion (keine *perspective division*) verwendet. Ebenso entfallen die Modelling-Transformations und somit eine dafür nötige Model-Matrix.

Die „Hello Triangles“-Anwendung

Imports	
WebGL	**OpenGL**
Sämtliche **imports** werden über die Aktivierung des WebGL-Kontextes im Canvas-Element geregelt und durchgeführt.	**Import static** org.lwjgl.BufferUtils; **import static** org.lwjgl.LWJGLException; **import static** org.lwjgl.opengl.Display; **import static** java.nio.ByteBuffer; **import static** java.nio.FloatBuffer; **import static** org.lwjgl.opengl.PixelFormat; **import static** org.lwjgl.opengl.ContextAttribs; **import static** org.lwjgl.opengl.DisplayMode; **import static** org.lwjgl.opengl.GL11*; **import static** org.lwjgl.opengl.GL15*; **import static** org.lwjgl.opengl.GL20*; **import static** org.lwjgl.opengl.GL30.*;

Globale Variablen	
WebGL	**OpenGL**
var vboId; var nboId; var iboId;	**private static int** *vaoId* = 0; **private static int** *vboId* = 0; **private static int** *iboId* = 0; **private static int** *naoId* = 0; **private static int** *nboId* = 0;
var programID; var vs; var fs; var ATTR_POS; var ATTR_NORMAL;	**private static int** *programID*; **private static** String *vs*; **private static** String *fs*; **private static int** *ATTR_POS*; **private static int** *ATTR_NORMAL*;
var width = 600; var height = 600;	**private static int width = 600;** **private static int height = 600;**

Vertex Shader

OpenGL (in, uniform, out) – WebGL (attribute, uniform, varying)

```
vs =
   "#version 100                                                                      \n"+
   "in vec3 vs_in_normal;                                                               "+
   "in vec3 vs_in_pos;                                                                  "+

   "// Licht:                                                                         \n"+
   "uniform vec3 uAmbientColor;                                                         "+
   "uniform vec3 uLightingDirection;                                                    "+
   "uniform vec3 uDirectionalColor;                                                     "+
   "out vec3 vLightWeighting;                                                           "+

   "void main(void)                                                                   \n"+
   "{                                                                                   "+
   "gl_Position = vec4(vs_in_pos, 1.0);                                                 "+
   "//BEGINN BELEUCHTUNG                                                              \n"+
   "float fDirectionalLightWeighting = max(dot(vs_in_pos.xyz, -uLightingDirection), 0.0);    "+
   "vLightWeighting = uAmbientColor + uDirectionalColor * fDirectionalLightWeighting;       "+
   "//ENDE BELEUCHTUNG                                                                \n"+
    "}                                                                                  ";
```

Fragment Shader
OpenGL (in, uniform, out) – WebGL (varying, uniform)

```
fs =
   "#version 100                                                                     \n"+
   "//this ifdef is a temporary work-around for the (upcoming) strict shader validator   \n"+
   "#ifdef GL_ES                                                                     \n"+
   "precision highp float;                                                           \n"+
   "#endif                                                                           \n"+
   "in vec3 vLightWeighting;                                                         \n"+
   "void main(void)                                                                  \n"+
   "{                                                                                  "+
   "gl_FragColor = vec4(vLightWeighting,1.0);                                          "+
   "}                                                                                  ";
```

main()

Diese Methode ist die ausführende Schleife der Hauptanwendungen. In einer festen Prozesskette werden hier alle Funktionen aufgerufen, die für den korrekten Ablauf der Anwendung nötig sind.

WebGL

```
function main(){

  setupWebGL();
  programID = createShaderProgram(vs, fs);
  gl.useProgram(programID);

  setupTriangles();

  gl.enableVertexAttribArray(ATTR_POS);
  gl.enableVertexAttribArray(ATTR_NORMAL);
  gl.clearColor(0.4, 0.4, 0.4, 1.0);
  gl.clearDepth(1.0);
  gl.enable(gl.DEPTH_TEST);
  glViewPort(0, 0, width, height);

  drawScene();

}
```

OpenGL

```
public static void main(String[] args) {

  setupOpenGL();
  programID = createShaderProgram(vs, fs);
  glUseProgram(programID);

  setupTriangles();

  glEnableVertexAttribArray(ATTR_POS);
  glEnableVertexAttribArray(ATTR_NORMAL);
  glClearColor(0.4f, 0.4f, 0.4f, 1.0f);
  glClearDepth(1.0);
  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, width, height);

  drawScene();
  Display.update();
}
```

setupWeb / OpenGL

Initialisierung des Viewport sowie des OpenGL/WebGL-Kontextes. Des Weiteren liegen hier die Shader in Quelltextform vor und werden in einer Variablen gespeichert.

WebGL

```
function setupWebGL(){
vs = “Vertex Shader als String”;
fs = „Fragment Shader als String“;

// WebGL spezifisch
var canvas =
document.getElementById("WebGL-canvas");

gl = WebGLUtils.setupWebGL(canvas);

 if(gl == null){
  alert ("kein WebGL-Kontext gefunden.
");
  }


}
```

OpenGL

```
public static void setupOpenGL() {
vs = “Vertex Shader als String”;
fs = „Fragment Shader als String“;
//LWJGL spezifisch
 try {
  PixelFormat pixelFormat = new PixelFormat();
  ContextAttribs contextAtrributes =
  new ContextAttribs(3, 2).withProfileCore(true)
                  .withForwardCompatible(true);

 Display.setDisplayMode(new DisplayMode(width, height));
 Display.create(pixelFormat, contextAtrributes);
 } catch (LWJGLException e) {
       e.printStackTrace();
       System.exit(-1);
  }
}
```

setupTriangles()

Initialisierung und Befüllen des Vertex-, Normalen- und Indexbuffers mit den Dreieck-Daten. Explizit sei hier auf die Verwendung von VertexArray-Objects in OpenGL hingewiesen. Dies ist in WebGL nicht möglich, da das Konstrukt dort nicht existiert und allein über WebGL Buffer realisiert wird.

WebGL

```
function setupTriangles()  {

  var vertices = [
     0.747306, -0.009661, 0.361111,
    -0.782882, -0.005750, -0.210892,
    -0.282495, 0.535817, 0.786934,
     0.283156, 0.363219, -0.727404
];

  vboId = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vboId);

  gl.bufferData(gl.ARRAY_BUFFER,
                new Float32Array(vertices),
                gl.STATIC_DRAW);

  var normals = [
   -0.425167, 0.867653, -0.257704,
    0.493840, 0.865305, 0.085841,
   -0.425167, 0.867653, -0.257704,
    0.493840, 0.865305, 0.085841
];

nboId = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, nboId);

gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(normals), gl.STATIC_DRAW);

var indices = [
     1, 2, 3,
     0, 3, 2
  ];

  iboId = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER,
                                     iboId);

gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new
               Uint16Array(indices),
               gl.STATIC_DRAW);
}
```

OpenGL

```
public static void setupTriangles() {

float[] vertices = {
   0.747306f, -0.009661f, 0.361111f,
  -0.782882f, -0.005750f, -0.210892f,
  -0.282495f, 0.535817f, 0.786934f,
   0.283156f, 0.363219f, -0.727404f
};

  FloatBuffer verticesBuffer =
  BufferUtils.createFloatBuffer(vertices.length);

  verticesBuffer.put(vertices);
  verticesBuffer.flip();

  vaoId = glGenVertexArrays();
  glBindVertexArray(vaoId);
  vboId = glGenBuffers();
  glBindBuffer(GL_ARRAY_BUFFER, vboId);
  glBufferData(GL_ARRAY_BUFFER, verticesBuffer,
                                GL_STATIC_DRAW);
float[] normals = {
    -0.425167f, 0.867653f, -0.257704f,
     0.493840f, 0.865305f, 0.085841f,
    -0.425167f, 0.867653f, -0.257704f,
     0.493840f, 0.865305f, 0.085841f
};

  FloatBuffer normalBuffer =
  BufferUtils.createFloatBuffer(normals.length);

  normalBuffer.put(normals);
  normalBuffer.flip();

  naoId = glGenVertexArrays();
  glBindVertexArray(naoId);
  nboId = glGenBuffers();
  glBindBuffer(GL_ARRAY_BUFFER, nboId);
  glBufferData(GL_ARRAY_BUFFER, normalBuffer,
                                GL_STATIC_DRAW);

  byte[] indices = {
        1, 2, 3,
        0, 3, 2
  };

  ByteBuffer indicesBuffer =
  BufferUtils.createByteBuffer(6);
  indicesBuffer.put(indices);
  indicesBuffer.flip();

 iboId = glGenBuffers();
 glBindBuffer(GL15.GL_ELEMENT_ARRAY_BUFFER, iboId);

 glBufferData(GL_ELEMENT_ARRAY_BUFFER,
                   indicesBuffer, GL_STATIC_DRAW);

}
```

createShaderProgram()

Erzeugung, Bindung und Verlinkung des Shaderprogramms. Der Shader-Quellcode wird in beiden Beispielen direkt von einer String-Variable kompiliert und an das Shader-Programm gebunden.

WebGL

```
function createShaderProgram(vs, fs){


var programID = gl.createProgram();
var vsID = gl.createShader(gl.VERTEX_SHADER);
var fsID = gl.createShader(gl.FRAGMENT_SHADER);

  gl.attachShader(programID, vsID);
  gl.attachShader(programID, fsID);

  gl.shaderSource(vsID, vs);
  gl.shaderSource(fsID, fs);

  gl.compileShader(vsID);
  gl.compileShader(fsID);

ATTR_POS = gl.getAttribLocation(programID,
                                "vs_in_pos");

ATTR_NORMAL = gl.getAttribLocation(programID,
                             "vs_in_normal");

 gl.linkProgram(programID);

 return programID;
}
```

OpenGL

```
public static int createShaderProgram(
                     String vs, String fs) {

int programID = glCreateProgram();
int vsID = glCreateShader(GL_VERTEX_SHADER);
int fsID = glCreateShader(GL_FRAGMENT_SHADER);

 glAttachShader(programID, vsID);
 glAttachShader(programID, fsID);

 glShaderSource(vsID, vs);
 glShaderSource(fsID, fs);

 glCompileShader(vsID);
 glCompileShader(fsID);

ATTR_POS = glGetAttribLocation(programID,
                                "vs_in_pos");

ATTR_NORMAL = glGetAttribLocation(programID,
                             "vs_in_normal");

glLinkProgram(programID);

 return programID;
}
```

updateUniforms()

Aktualisierung sämtlicher Uniformvariablen.

WebGL

```
function updateUniforms(){

//Beleuchtung
gl.uniform3f(gl.getUniformLocation(programID,
                               "uAmbientColor"),
                                  0.7, 0.7,0.7);

var vLightDir = [-0.267,-0.267,-0.267];

gl.uniform3f(gl.getUniformLocation(programID,
"uLightingDirection"),   vLightDir [0],
 vLightDir [1], vAdjLDir[2]);


gl.uniform3f(gl.getUniformLocation(programID,
         "uDirectionalColor"), 0.8, 0.8, 0.8);
}
```

OpenGL

```
public static void updateUniforms() {

//Beleuchtung
 glUniform3f(glGetUniformLocation(programID,
                               "uAmbientColor"),
                               0.7f, 0.7f,0.7f);

float[] vLightDir= {-0.267f, -0.267f, -0.267f };


glUniform3f(glGetUniformLocation(programID,
 "uLightingDirection"),  vLightDir [0],
vLightDir[1], vLightDir [2]);


glUniform3f(glGetUniformLocation(programID,
      "uDirectionalColor"), 0.8f, 0.8f, 0.8f);
}
```

drawScene()	
Methode die das Rendering ausführt und in einer Schleife aufgerufen wird.	
WebGL	OpenGL

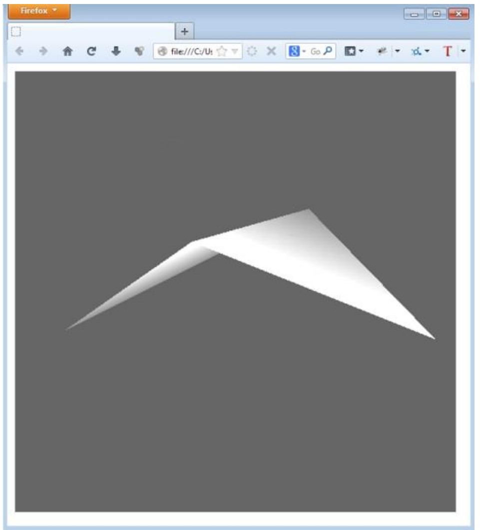
WebGL:

```
function drawScene()  {

 gl.clear(gl.COLOR_BUFFER_BIT |
         gl.DEPTH_BUFFER_BIT);

 updateUniforms();

 gl.bindBuffer(gl.ARRAY_BUFFER,vboId);
 gl.vertexAttribPointer(ATTR_POS, 3, gl.FLOAT,
                                    false, 0,0);

 gl.bindBuffer(gl.ARRAY_BUFFER, nboId);
 gl.vertexAttribPointer(ATTR_NORMAL, 3, gl.FLOAT,
                                    false, 0, 0);

 gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, iboId);
 gl.drawElements(gl.TRIANGLES, 6,
                           gl.UNSIGNED_SHORT, 0);

}
```

OpenGL:

```
public static void drawScene() {

  glClear(GL_COLOR_BUFFER_BIT |
          GL_DEPTH_BUFFER_BIT);

  updateUniforms();

  glBindBuffer(GL_ARRAY_BUFFER, vboId);
  glVertexAttribPointer(ATTR_POS, 3, GL_FLOAT,
                                 false, 0, 0);

  glBindBuffer(GL_ARRAY_BUFFER, nboId);
  glVertexAttribPointer(ATTR_NORMAL, 3,
                       GL_FLOAT, false, 0, 0);

  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, iboId);
  glDrawElements(GL_TRIANGLES, 6,
                        GL_UNSIGNED_BYTE, 0);

}
```

Ausgabe der Dreiecke	
Die Ausgabe der Dreiecke geschieht im eingebettetem Canvas-Element im Browser (links) mit den zuvor gesetzten Größenangaben von 600 Pixeln in Höhe und Breite, wie auch im OpenGL-Fenster (rechts) der Java Application mit den gleichen Maßen.	
WebGL	OpenGL



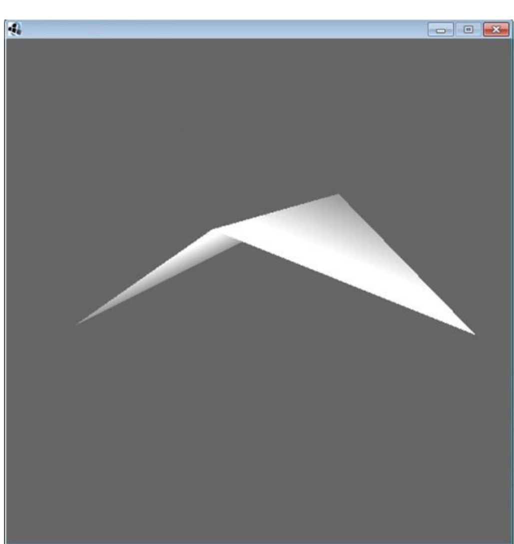


TABELLE 2: CODE-VERGLEICH WEBGL - OPENGL

Allgemeiner Vergleich

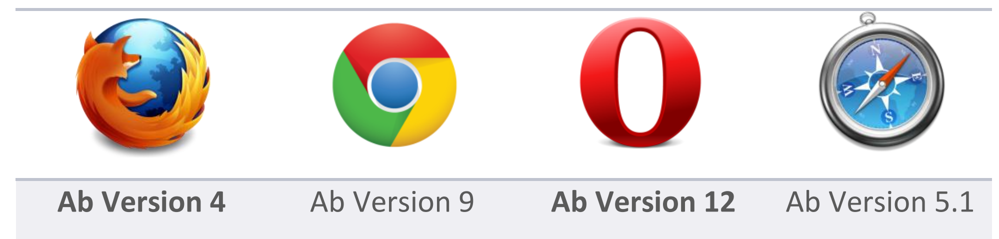
Alle Funktionen, die in der OpenGL Bibliothek vorhanden sind, basieren auf den Programmiersprachen C und C++. Im Lauf der Jahre entstanden Frameworks und Wrapperklassen mit denen es möglich wurde, OpenGL auch in Java zu programmieren was vor allem Bildungsinstitutionen (Hochschulen und Universitäten) zu Gute kam, da Java häufig als Lehrsprache in der Informatik zum Einsatz kommt. Im Gegensatz dazu werden die Funktionen der WebGL API, deren Basis die Version OpenGL ES 2.0 ist, vollständig über JavaScript angesprochen, eine Skript-Sprache die in allen gängigen Browsern unterstützt wird und zum dynamischen Web beiträgt. Die Daten, die am häufigsten in OpenGL verarbeitet werden, sind Vertices, Indices, Normalen, Texturkoordinaten, sowie Texturen. Da bereits Eingangs (siehe Kapitel 7) eine detaillierte Beschreibung der Begriffe gegeben wurde, wird an dieser Stelle nicht näher darauf eingegangen. Dieselben Daten finden auch in WebGL Verwendung.

Wie bereits erwähnt basiert WebGL weitestgehend auf OpenGL ES 2.0. Da dieser Versionsstand vergleichbar mit OpenGL 2.0 (2004) ist, kann daraus geschlossen werden, dass der Funktionsumfang noch nicht so weit entwickelt ist wie jener, der aktuellen OpenGL Version. Diese wurde im August 2012 als Version 4.3 verabschiedet und beinhaltet neben dem Vertex- und Fragment-Shader zusätzlich einen Geometry Shader, zwei Tesselation Shader sowie einen Compute Shader. Während der Vertex- und Fragmentshader die beiden Shader darstellen, die seit Version 2.0 vorhanden und zur Verarbeitung der Vertices sowie Erzeugung der Fragments notwendig sind, ist der Geometry Shader ein „mögliches“ Element der Graphics Pipeline. Durch ihn können aus vorhanden Primitives neue primitive Geometrien erzeugt werden. Zur Verfeinerung bestehender Geometrien bietet sich der in OpenGL 4.0 eingeführte Tesselation Shader. Da neben dem Interesse 3D-Modelle in Echtzeit im Webbrowser zu rendern, ebenfalls auch Bedarf an parallelen Algorithmen besteht, befindet sich aktuell der Standard WebCL durch die Khronos Group in der Erarbeitungsphase. Dieser basiert auf der sehr jungen OpenCL Technologie und nutzt CPU, GPU bzw. ganz allgemein parallele Hardware, um die Parallelisierung von Algorithmen, beispielsweise für Partikelsysteme, Physik-Engines oder browserorientierte Video- und Bildverarbeitungs-Anwendungen, zu ermöglichen [Web 19].

Ein weiterer Aspekt, der direkten Einfluss auf die Vergleichbarkeit der Funktionen hat, betrifft die Verwendung von Vertex Array Objects [Web 20]. Sie bilden einen elementaren Bestandteil in OpenGL und ermöglichen eine verbesserte Performance. Dieser „Datentyp“ hat in WebGL bislang noch keinen Einzug erhalten und muss daher via `WebGL Vertex Buffer Objekten` und `JavaScript Arrays` implementiert werden. Beim Vergleich der beiden „Hello-Triangle“ Programmcodes wird dies deutlich.

Eine erhebliche Vereinfachung der Programmierung in WebGL, wird durch das automatische Speicher-Management erzielt. OpenGL Programme, die meist in der Programmiersprache C implementiert werden, sind darauf angewiesen, dass die Allokierung und Deallokierung des Speichers manuell vom Programmierer vorgenommen wird. Hierfür werden zwar entsprechende Methoden bereitgestellt, welche den intuitiven Umgang jedoch sehr erschweren. JavaScript hingegen arbeitet strikt nach den Regeln des Variablen Scoping und reserviert bzw. löscht entsprechende Speicherstellen, wenn sie nicht länger benötigt werden.

Folgende Browser, bis auf den Internet Explorer unterstützen WebGL (siehe Tabelle 2).



Ab Version 4	Ab Version 9	**Ab Version 12**	Ab Version 5.1

TABELLE 3: BROWSERKOMPATIBILITÄT FÜR WEBGL
QUELLE: [WEB 21]

Microsoft erhob in seiner Stellungnahme, bzgl. der Verweigerung von WebGL im Internet Explorer, Sicherheitsbedenken auf Grund der Kommunikation zwischen Browser und Grafikkarte, da somit ein direkter Zugriff auf die Hardware im Endsystem stattfindet [Web 22]. Bislang sind jedoch keine kriminellen Fälle in Verbindung mit dieser Sicherheitslücke bekannt. Im Übrigen muss WebGL in Chrome und Safari zunächst aktiviert werden, da die Funktion standardmäßig deaktiviert ist.

8. 3D Modelldaten

8.1. Autodesk Maya

Abbildung 30: Maya Logo
Quelle: [Web 23]

Autodesk Maya ist eine professionelle Software zur 3D-Visualisierung und Animation. Die Anfänge von Maya wurden von den Entwickler-Firmen Alias und Wavefront Technologies begründet. Im Jahre 2006 wurde die gesamte Softwarepalette (u.a. Maya), wegen der Übernahme durch den stärksten Konkurrenten Autodesk, weiter vorangetrieben und fortgeführt [Web 24]. Maya hat sich in der Film- und Fernsehindustrie, aber auch allgemein in der Industrie und Architektur zu einem der meistgenutzten und namenhaftesten Modellierungs- und Animationswerkzeugen entwickelt [TMAP]. Berühmte Ergebnisse die ihren Ursprung der Maya-Software verdanken sind unter anderem die fiktive Figur „Gollum“ aus dem Herr der Ringe Epos, Star Wars Episode III [Web 25] oder restaurierte Filmsequenzen (meist Außenaufnahmen von Raumschiffen) in der alten Star-Trek Serie. [Web 26]

8.2. Blender

Abbildung 31: Blender Logo
Quelle: [Web 27]

Bei Blender handelt es sich ebenfalls um eine 3D-Modellierungs- und Animationssoftware, dessen Umfang weitaus geringer als der von Autodesk Maya ist, jedoch in der Bedienung intuitiver und für Einsteiger geeigneter erscheint. Blender war anfangs ein betriebsinternes Entwicklertool des niederländischen Animationsstudios NeoGeo. Chefentwickler Ton Roosendaal gründete 1998 seine eigene Firma Not-a-Number Technologies (NaN), um Blender weiter zu entwickeln und zu vermarkten. Das Vorhaben endete mit der Insolvenz der Firma im Jahre 2002. Fortan waren die Rechte an Blender unter die freie Software-Lizenz GNU General Public License (GPL) gestellt und wurden unter dem Dach der neu gegründeten Blender Foundation weiter geführt, die ihre Kosten aus Spendengeldern deckt [Web 28]. Da der gesamte Quellcode des Programms öffentlich zugänglich ist, lässt sich Blender auf nahezu jedes Rechnersystem bzw. Rechnerplattform übersetzen und ausführen. Vorteilhaft ist der

niedrige Speicherbedarf trotz des hohen Funktionsumfangs. Die Größe der aktuellen Version 2.65a beläuft sich auf knapp 35MB [Web 29]. Dabei unterstützt Blender die wesentlichen Funktionen wie Modellierung, Texturierung, Lichtmodelle, Animation und Rendering und verfügt zudem über mehrere Rendering-Systeme, eine Spiele-Engine sowie einen Videoschnitt-Editor. Neben dem internen Blender-Renderer existiert der sog. „Cycles-Renderer“ der in Kombination mit CUDA oder OpenCL auf der GPU Echtzeit-Rendering bietet.

8.3. AUSGANGSDATEN

3-dimensionale Fahrzeugmodelle sind in ihrer inneren Struktur, beginnend bei der Anzahl der Vertices bis hin zu Materialeigenschaften, derart komplex, dass eine präzise manuelle Definition kaum noch möglich ist. Entsprechend den zu Verfügung stehenden Möglichkeiten, die auch in der Industrie genutzt werden, wurde auch bei den verwendeten Fahrzeugmodellen in dieser Arbeit auf 3D-Modellierungsprogramme zurückgegriffen, um in relativ kurzer Zeit einen ausreichend großen Pool an Fahrzeugdaten zu generieren, die ein effizientes Arbeit an der WebGL-Anwendung erst möglich machen. Die ersten Modelle, die in frühen Stadien der Entwicklung zur Verwendung kamen, entstammen der Online-Plattform *blendswap.com* für freie Blender-Modellierer. Auf Grund des Open-Source Hintergrunds fand der meiste Teil der 3D- und Konvertierungsarbeiten im Rahmen der Arbeit mit der Software Blender statt. In Kooperation mit VW Wolfsburg konnte zusätzlich auf ein 3D-Modell des jüngst erschienenen VW Up! zugegriffen werden. Ab diesem Zeitpunkt konnten hochauflösende CAD-Daten in die Anwendung einfließen, die die qualitativen Anforderungen der Industrie (Konzeption und Fertigung) erfüllen und daher keines Wegs vergleichbar sind mit den Modellen freier Modellierer. Zur Bearbeitung der VW-Rohdaten kam Maya vom Software-Hersteller Autodesk zum Einsatz, welches als Studentenversion frei zugänglich ist, und für die Zwecke dieser Arbeit in Bezug auf den Funktionsumfang vollkommen ausreichend war.

Zusammengefasst lagen für ein 3D-Modell, basierend auf den verwendeten Programmen, die folgenden Datenformate vor:

- *.obj OBJ-Format (Blender)
- *.mtl MTL-Format (Blender)
- *.blend Blender-Format (Blender)
- *.my Maya-Format (Autodesk Maya)
- *.mb Maya Binary Format (Autodesk Maya)
- *.json JSON Format (JSON)

8.4. OBJ, MTL UND JSON

OBJ

Auf dem aktuellen Software-Markt für 3D-Modellierungsprogramme existiert eine große Bandbreite an Produkten. Bekannte Vertreter sind Cinema 4D, Autodesk Maya, Blender, 3DS Max. Die Firma ALIAS | Wavefront Technologies, heute Teil der Firma Autodesk, erkannte bereits sehr früh den Nutzen eines offenen 3D-Formats, das von vielen Modellierungsprogrammen gemeinsam unterstützt wird, um das Importieren sowie das Exportieren zwischen den Programmen zu erleichtern. Für diesen Zweck, dem zum damaligen Zeitpunkt in der Entwicklung stehenden Programm Maya, sowie dem eigenen Advanced Visualizer Projekt [Web 30], wurde das OBJ-Format eingeführt.

Das OBJ-Format speichert im reinen ASCII-Text die wichtigsten Elemente eines dreidimensionalen Objekts. Dazu zählen neben den Vertices, die das Mesh (Netz aus Polygonen) beschreiben, ebenso die Indices, Normalen und Texturkoordinaten. Um optische Eigenschaften (Farben, Spiegelungen, Transparenz, Glanzlichter, usw.) zu integrieren, wird das MTL-Dateiformat (Material Template Library) angeboten, welches über einen eigenen Begriff in der OBJ-Datei referenziert werden kann.

Die grundlegende Syntax des OBJ-Formats wird in der folgenden Tabelle beschrieben und steht in enger Verknüpfung mit Abbildung 32 auf der folgenden Seite:

`#`	Einleitung eines Kommentars
`o`	Name des erzeugten 3D-Objekts
`v`	Definition eines Vertex mit (x,y,z)-Koordinate
`vn`	Definition einer Normalen mit (x,y,z)-Koordinate
`vt`	Defintion der Texturkoordinate (s,t)
`f`	Face (hier Dreiecksfläche) `f i1 // n1  i2 // n2  i3 // n3` Face bestehend aus den Indices i1, i2 und i3 sowie der Flächennormalen n1, n2 und n3
`g`	Deklaration eines Objekts (wie „o“) als Polygongruppe
`s`	Smoothing der Vertices, falls durch 3D-Programm gesetzt
`mtllib`	Verweis zur verwendeten Material-Datei (MTL)
`usemtl`	Schlüsselwort zur Zuweisung eines expliziten Materials zu einer Polygongruppe

TABELLE 4: OBJ SYNTAX

Im Rahmen der Arbeit kam das 3D-Modellierungsprogramm Blender zur Anwendung, das neben vielen anderen Import/Export-Formaten auch das OBJ/MTL-Format nutzt. Um zu verdeutlichen, wie sich eine in Blender modellierte Geometrie im OBJ-Format darstellt, zeigt Abbildung 32 die wesentlichen Elemente und Verknüpfungen.

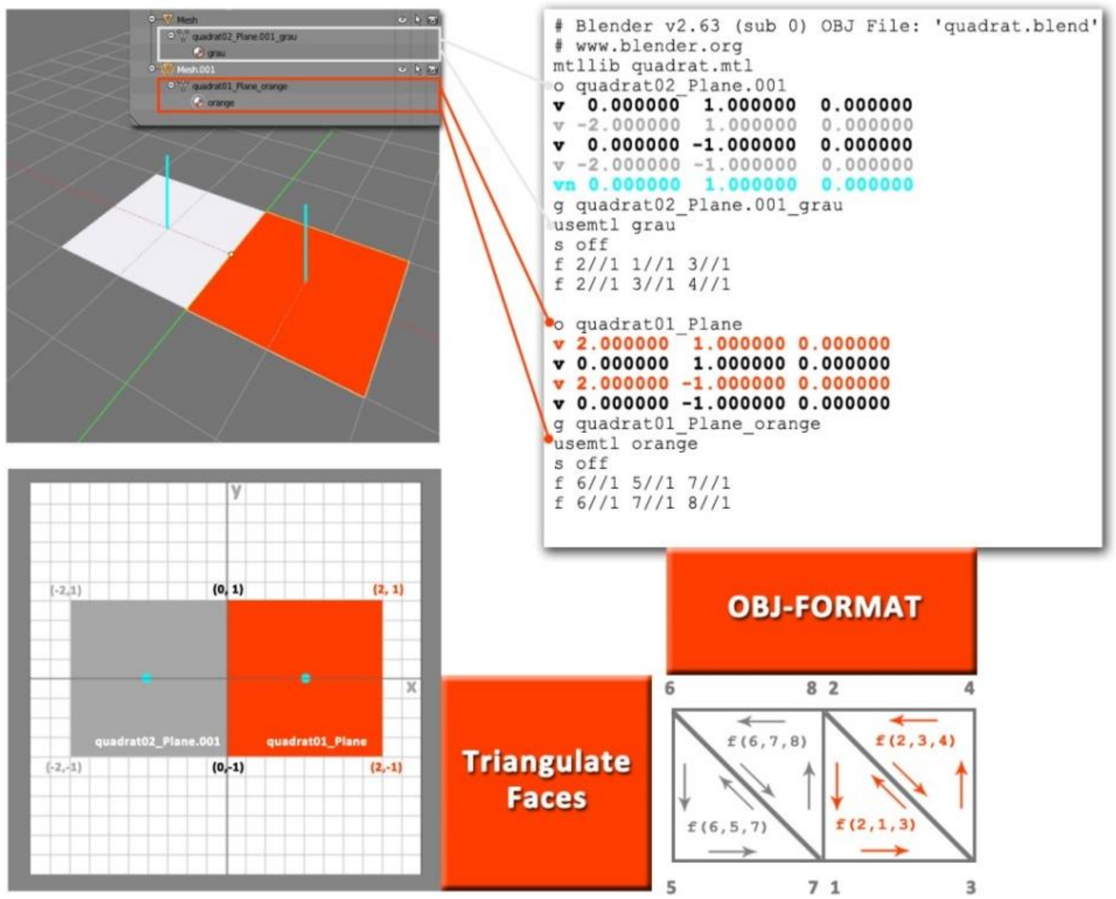


Abbildung 32: Blender in OBJ

Links oben sind das Modellierungsfenster in Blender, sowie ein Ausschnitt der Geometrieliste zu sehen. Wird für jedes Mesh die Unterebene geöffnet, werden die Polygongruppen (in OBJ mit „g“ bezeichnet) aufgeführt und die darauf definierten Materialien. Für ein besseres Verständnis, ist darunter das Grid mit der Geometrie in einem 2-dimensionalen Koordinatensystem auf der xy-Ebene dargestellt, zum Ablesen der Vertex-Koordinaten. Um kostenintensive Indexbuffer-Berechnungen zu vermeiden, unterzieht man das gesamte Mesh einem Triangulierungs-Algorithmus, der die gesamte Geometrie in Dreiecke transformiert. Damit sind alle Vorkehrungen getroffen mit jedem 3D-Modellierungsprogramm einen OBJ-Export durchzuführen, der kompatibel für weitere Schritte (Import, Export, Konvertierungen, etc.) ist.

MTL

Als Erweiterung des OBJ-Formats, um die optischen Eigenschaften eines 3D-Objekts zu beschreiben, kann das MTL-Format angesehen werden, das parallel zur Generierung der OBJ-Datei erzeugt wird. In einer .mtl-Datei befindet sich zu Beginn ein Verweis auf die Ursprungsdatei der Geometrie, sowie die Gesamtanzahl der folgenden Materialien. Jedes neue Material beginnt mit dem Identifier `newmtl` und dem Namen. Die nachfolgende Tabelle 5 stellt die verschiedenen Komponenten des verwendeten Beleuchtungsmodells dar, welche auch in Abbildung 33 aufgezeigt werden:

#	**Einleitung eines Kommentars**
material count	Anzahl aller Materialien in der MTL-Datei
newmtl	Anlegen eines neuen Materials
Ns	Glanz-Exponent der spekularen Reflektion
Ka	Ambienter Farbanteil des Materials
Kd	Diffuser Farbanteil des Materials
Ks	Spekularer Farbanteil des Materials
Ni	Optische Dichte eines Mediums (Lichtdurchlässigkeit)
d	Transparenzwert (0 ≙ durchsichtig, 1 ≙ undurchsichtig)

TABELLE 5: MTL SYNTAX

Abbildung 33 dient dazu, wie zuvor beim OBJ-Format die Parallelen zum Blender-Format und dem Modellierungsprogramm zu ziehen. Hier liegt der Fokus allerdings weniger auf dem Modellierungsbereich, sondern auf dem Materialbereich der gewählten Geometrie. Die in der MTL-Datei resultierenden Werte für die Materialkoeffizienten (Ns, Ka, Kd, Ks, Ni, d) basieren alle auf den benutzerspezifischen Eingaben im Programm. Beim Export werden alle jemals angelegten Materialien zusammengeführt und ausgegeben.

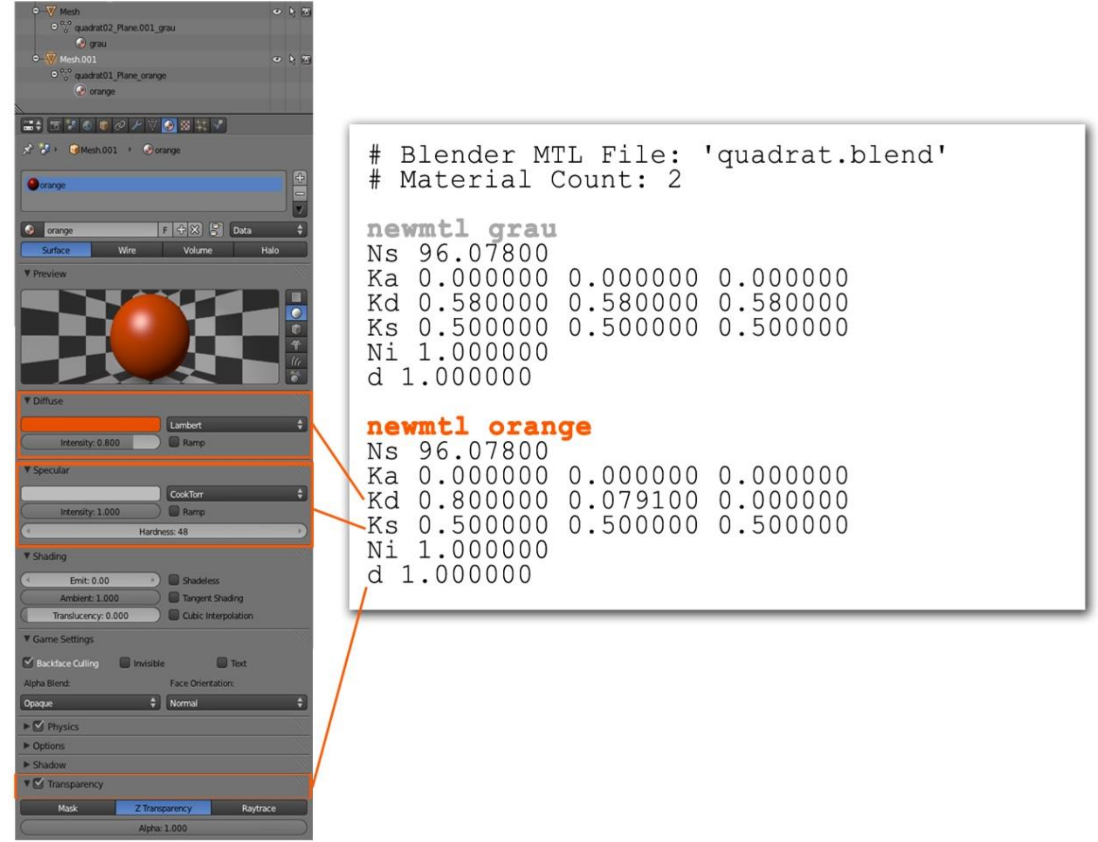


ABBILDUNG 33: BLENDER IN MTL

JSON

Bei JSON (JavaScript Object Notation) handelt es sich um ein leichtgewichtiges Datenaustauschformat, das insbesondere bei Webservices wie SOAP und REST verwendet wird. Leichtgewichtig ist es deshalb, weil es die Probleme des ähnlichen Formats XML bezüglich des hohen Overheads (höherer Speicherbedarf) über eine Syntax-Teilmenge mit JavaScript umgeht. Während in XML viele eigene Tags und Leerzeichen den Speicherbedarf wachsen lassen, beschränkt sich JSON auf eine minimalistische Syntax, die vollständig aus JavaScript Code besteht. Somit können Evaluierungs- und Parsing-Vorgänge sehr einfach durchgeführt werden, da JavaScript speziell dafür vorgesehene Funktionen besitzt (`eval()`

und `JSON.parse()`). Die Struktur des Formats baut auf zwei Grundstrukturen auf – `Arrays` und `Name/Wert` Paare. Die folgende Tabelle 6 beschreibt die Konstrukte der JSON Syntax:

object	*members*	*pair*	*array*	*elements*	*value*
{} {*members*}	*pair* *pair, members*	*string : value*	*[]* *[elements]*	*value* *value, elements*	*string*(*char*) *number*(*int*) *object* *array* true false *null*

Tabelle 6: JSON Konstrukte
Quelle:[Web 31]

Um 3D-Formate wie OBJ und MTL in das JSON Format zu konvertieren, bedarf es der Verwendung von sogenannten Parsern. Aufgrund des hohen Interesses an der Entwicklung von Konvertierungsskripten, um möglichst vielen Formaten den Weg ins Internet zu ermöglichen, existiert eine Vielzahl an frei verfügbaren Varianten. Auf die Logik des hier verwendeten Parsers wird nicht näher eingegangen. Die Ausgabe des OBJ-JSON-Parsers in Bezug auf die im Kapitel 8.4 eingeführte Quadrat-Geometrie (siehe Abbildung 28), zeigt der nachfolgende Quellcode-Ausschnitt:

Sie verdeutlicht noch einmal die Syntax von JSON und rundet das Kapitel der Ausgangsdaten als letztes wichtiges Dateiformat für den Automobilkonfigurator ab.

```
{
  "alias" : "quadrat01_Plane_orange",
  "vertices" : [2.0,1.0,0.0,0.0,1.0,0.0,2.0,-1.0,0.0,0.0,-1.0,0.0],
  "indices" : [2,1,3,2,3,4],
  "Ni" : 1.00000,
  "Ka" : [0.00000,0.00000,0.00000],
  "d" : 1.00000,
  "Kd" : [0.80000,0.07910,0.00000],
  "Ks" : [0.50000,0.50000,0.50000],
  "Ns" : 96.07800
}
```

9. Die Client-Server-Architektur

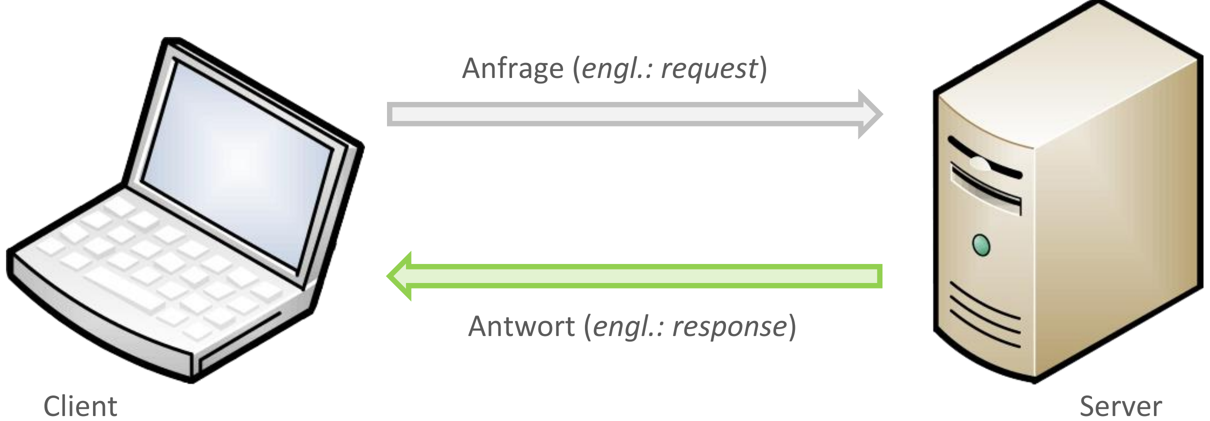


Abbildung 34: Client-Server Modell

Das Client-Server Modell stellt eines der grundlegenden Netzwerkmodelle dar. Man geht dabei von einem Netzwerk aus, bestehend aus einem oder mehreren Clients, sowie einem oder mehreren Servern. Im Rahmen dieses Modells wird die Datenverarbeitung eines Anwendungsprogramms zwischen dem Client- und dem Server-Teil aufgeteilt, da in der Regel der Client lediglich für die Darstellung und Funktionen der Benutzerschnittstelle zuständig ist, der Server für die Bereitstellung und Verwaltung der Daten [WinfEE09].

Um den Blick von einer eher hardware-orientierten Sicht des Modells auf die theoretische Sicht zu lenken, wie sie auch in den meisten informatischen Ausführungen präferiert wird, wird im Folgenden der Client als Anwendungsprogramm betrachtet, das von einem anderen Anwendungsprogramm (Server) einen Dienst anfordert (request). Der Server antwortet (response) dem Client mit der Bereitstellung des Dienstes und ist in der Lage gleichzeitig auch mehrere Clients zu bedienen. Dies ist abhängig von den Ressourcen des Servers.

Neben komplexen Client-Server-Ketten, in denen Server durch Dienstanforderungen bei anderen Servern zu einem Client werden, existieren auch lokale Systeme. Hier befinden sich beide Parteien in einem Rechner wie beispielsweise bei einem Webserver, der Dienste eines Datenbankservers in Anspruch nimmt, welcher ebenfalls auf diesem Rechner stationiert ist. Alle genannten Szenarien werden in der folgenden Abbildung 35 zusammenfassend dargestellt.

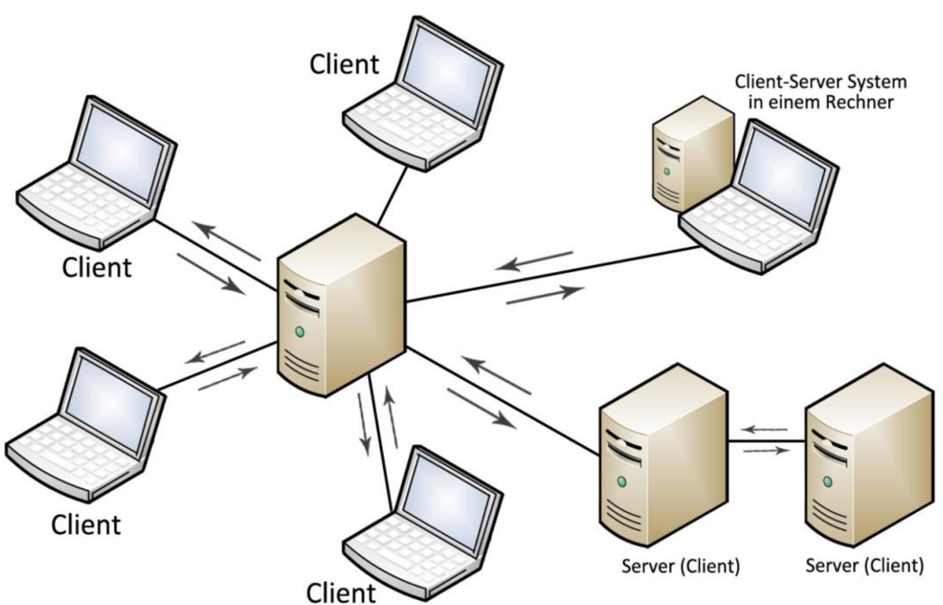


ABBILDUNG 35: CLIENT-SERVER NETZWERK

Die Architektur hinter dem WebGL-Automobilkonfigurator basiert ebenfalls auf dem Client-Server-System, da sich diese Modell (sehr) dazu anbietet. So ist es möglich, die große Menge an Daten, die durch die 3D-Fahrzeugmodelle entsteht, auf einen leistungsfähigen Server auszulagern und sie über gezielte Anforderung seitens des Clients abzurufen. Der Endnutzer benötigt zur ordnungsgemäßen Ausführung der Anwendung nichts weiter als die GUI (Graphical User Interface) der WebGL-Applikation als Schnittstelle zwischen Client und Server, um die entsprechenden Dienste anzufordern und im Canvas-Fenster zu rendern.

10. CLIENT - DER AUTOMOBIL-KONFIGURATOR

Die Benutzeroberfläche

Die grundlegende Benutzeroberfläche der Anwendung beschränkt sich auf ein sehr funktionales Layout. Eingebettet in eine Hintergrundgrafik befindet sich im linken Drittel der Menübereich, in dem die interaktiven Parameter liegen. Im rechten Teil der Website ist das WebGL Canvas-Element, sowie der Platzhalter für den Ladebalken, der später mit JQuery implementiert wird, verankert. Durch relative Größenangaben kann das Layout auf jedes beliebige Format skaliert werden. Ebenso variabel ändert sich der Viewport in Abhängigkeit von der Größe des Canvas-Elements, sodass bei beliebiger Auflösung, die entsprechenden Seitenverhältnisse der 3D-Szene erhalten bleiben.

ABBILDUNG 36: WEBSITE LAYOUT

ABBILDUNG 37: INTERAKTIVE SCHALTFLÄCHEN

Entsprechend der allgemeinen Funktionen von Automobilkonfiguratoren, sind auch hier entsprechende, interaktive Benutzerschaltflächen (Abbildung 36) vorgesehen und implementiert, die das Interface zur Programmlogik herstellen. Folgende Funktionen existieren in der WebGL Variante des Automobilkonfigurators: Se sollen im weiteren Verlauf näher erläutert werden.

- Fahrzeugauswahl
- Felgenauswahl
- Lackierung
- Perspektiven
- Mausaktion zur Kamerasteuerung

11. SERVER

In diesem Kapitel besitzen sämtliche Komponenten serverseitige Berührungspunkte d.h. der Speicherort liegt auf dem Server und sämtliche Anfragen, bis auf wenige Ausnahmen durch den Client bezüglich dieser Komponenten, sind an den Server gerichtet.

11.1. DIE SZENE

Showroom

Die initiale Szene des Automobilkonfigurators startet mit der Frontperspektive des Showrooms, in dem später die 3D-Modelle geladen werden. Das Gebilde selbst wurde in Blender modelliert und besteht aus einfachen geometrischen Grundelementen (Quader, Zylinder), um die Anzahl der generierten Dreiecke sowie den damit verbundenen Datenübertragungsaufwand möglichst gering zu halten. Die Punkt-Lichter der Szene, wie sie in Abbildung 36 zu sehen sind, stellen hier nur repräsentativ den Ort der Lichtquellen dar, die später aus dem Programmcode heraus generiert und platziert werden. Die Arbeit mit Beleuchtung war in Blender bereits nötig, um die entsprechenden Material-Charakteristika zu definieren.

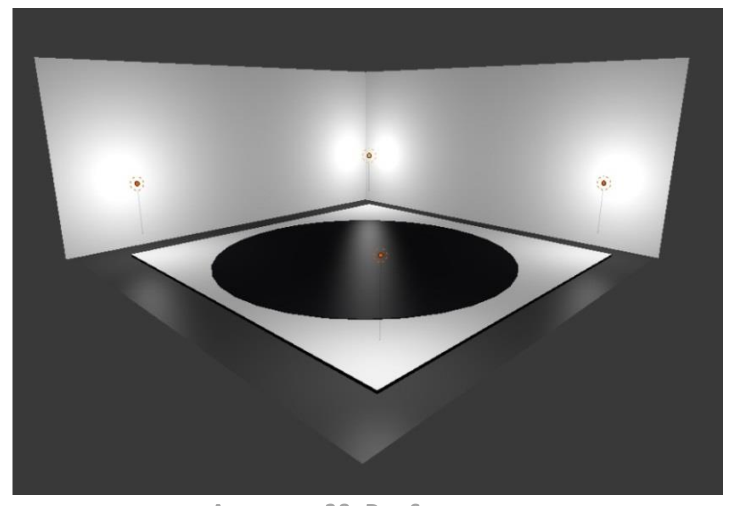


ABBILDUNG 38: DER SHOWROOM

11.2. BLENDER 3D MODELLE

Alle im Automobilkonfigurator verwendeten Fahrzeuge wurden nicht selbst modelliert, sondern von der Website `www.blendswap.org` entnommen. Sämtliche urheberrechtlich geschützten Materialien sind mit ihrem Autor am Ende der Arbeit genannt, da die meisten Werke unter einer Creative Common Lizenz liegen. Eine Ausnahme betrifft den VW Up! der durch die technische Entwicklung von Volkswagen Wolfsburg bereitgestellt wurde. Folgende Tabelle soll einen Einblick in die Größenordnungen der Fahrzeuge sowie den Speicherbedarf geben:

Fahrzeug	OBJ	JSON	Vertices	Faces	Materialien
Audi R8	176 MB	56,4 MB	1.020.059	1.847.758	38
Aston Martin DBS	39 MB	20,7 MB	445.076	642.525	17
VW Up!	323 MB	126 MB	3.502.866	2.074.492	43

TABELLE 7: DATENUMFANG

11.2.1 Export von Blender nach OBJ

Eine detaillierte Einführung zum Export des Blender-Formats in das offene OBJ-Format gab es bereits im Grundlagenkapitel 8.4, weshalb an dieser Stelle lediglich auf ein paar wichtige Modalitäten mit Blender eingegangen wird. Nach dem ein Fahrzeugmodell fertig modelliert, auf den Showroom skaliert wurde und sämtliche Materialien gesetzt sind, kann der Export nach OBJ durchgeführt werden (siehe Abbildung 39).

Abbildung 39: Blender Oberfläche

Dazu folgt man der Anweisung `File -> Export -> Wavefront (*.obj)`

Im Exportmenü selbst sind folgende Parameter (siehe Abbildung 39) zu setzen:

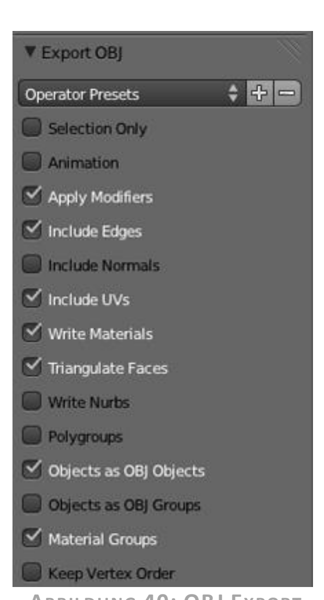


Abbildung 40: OBJ Export Parameter

Apply Modifiers Rendering sämtlicher Vertices, die durch mathematische Operationen entstanden sind (bspw.: Reduzierung der Vertex-Anzahl von Meshes). Die Operationen werden speziell in Blender als „Modifier" bezeichnet und sollten gesetzt werden, da ansonsten ein unvollständiges 3D-Modell in der Darstellung die Folge sein kann.

Include Edges Falls Kanten (engl.: edges) existieren, die nicht zu einer Dreiecksfläche (face) gehören, werden diese hierdurch ebenfalls exportiert.

Include UVs Falls Texturen vorhanden sind, werden die entsprechenden Texturkoordinaten somit exportiert.

Write Materials	Generierung der MTL-Datei, die sämtliche Materialien beinhaltet.
Triangulate Faces	Aktivierung des Triangulisierungs-Algorithmus, der das gesamte Mesh mit Dreiecken darstellt. Entsprechend werden die Indices jedes OBJ-Objekts gesetzt und in ein Array gespeichert. Die Rendermethode `gl.TRIANGLES` kann so ohne vorherige Manipulationen am Indexbuffer durchgeführt werden
Objects as OBJ Objects	Sämtliche Objekte in Blender werden als Objekte in OBJ angelegt
Material Groups	Besteht ein Blender-Objekt aus mehreren Materialien (bspw.: ein Reifen der aus Gummi und Aluminium besteht), dann wird das Objekt in ein Objekt pro Material aufgeteilt. Somit würde es ein Objekt `Reifen_gummi` und `Reifen_aluminium` geben.

11.2.2 PARSEN VON OBJ UND MTL NACH JSON

Der letzte Konvertierungsschritt beschäftigt sich mit dem Parsing der generierten OBJ- und MTL-Dateien nach JSON. Ein entscheidender Vorteil - bedingt durch das strukturierte Textformat von OBJ, MTL und JSON - liegt in der relativ einfachen Gestaltung des Parsers. Der verwendete Parser wurde dabei nicht selbst entwickelt, sondern auf einer Webpräsenz zum Thema WebGL entnommen und umgestaltet [Web 32].

Die für den Automobilkonfigurator relevanten Daten sind:

- Objekt-Name (OBJ-Geometriegruppe)
- Geometriedaten: Vertices und Indices
- Materialname pro OBJ-Geometriegruppe
- Materialkonstanten (Ni, Ka, Kd, Ks, d, Ns)

Diese werden gezielt aus der OBJ – und MTL-Datei ausgelesen und pro neuer OBJ-Geometriegruppe in eine eigene Datei geschrieben, sodass am Ende jede

Fahrzeugkomponente eine eigene JSON-Datei besitzt. Diese kann später separat manipuliert werden. Der gesamte Vorgang wird in Abbildung 41 verdeutlicht.

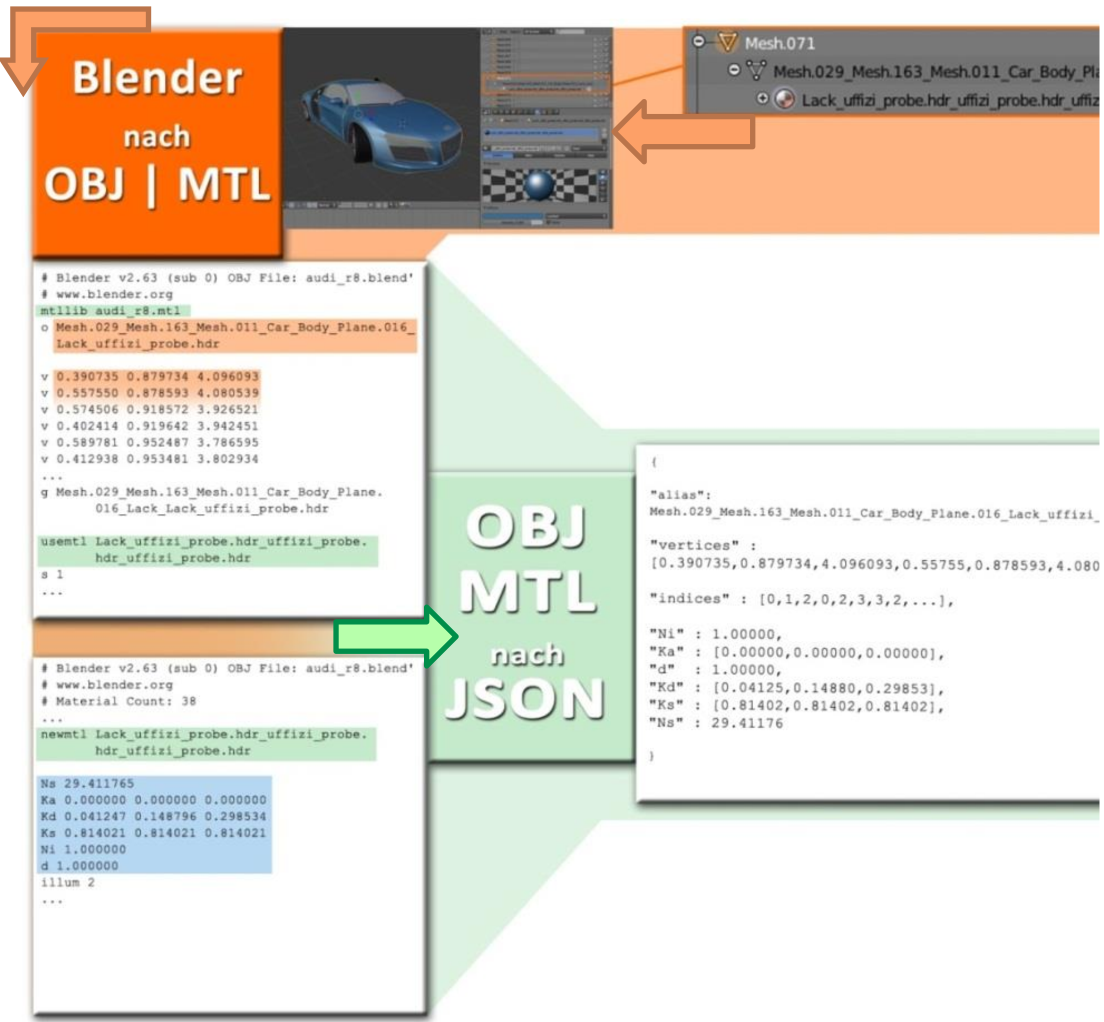


ABBILDUNG 41: PARSING OBJ NACH JSON

11.3. RESSOURCEN-ORIENTIERTES RENDERING

Die RequestAnimFrame Funktion

Die Bildrate einer gerenderten Animation im Web ist abhängig von der CPU- und GPU-Leistung sowie der Auslastung und Konnektivität der Internetverbindung. Um trotz dieser sensiblen Einflussfaktoren ein qualitativ hochwertiges und ressourcen-orientiertes Rendering zu ermöglichen, steht das Thema „Timing“ im Fokus. Hier kamen bis vor kurzem die JavaScript-Methoden `setTimeout()` und `setInterval()` zum Zuge, um durch ein fest vorgegebenes Zeitintervall automatisierte Rendering-Aufrufe zu starten [Web 33].

Dieses beschränkte Intervall hindert allerdings den Animationsprozess auf die Auslastung der Hardware zu reagieren, da das Verhalten der Hardware-Auslastung alles andere als konstant ist, sondern im höchsten Maße dynamisch, je mehr Prozesse es zu verarbeiten gilt. Bereits bei der Ausführung mehrerer Animationen im Browser, stößt man mit diesen Methoden schnell an die Grenzen der Belastbarkeit.

Abhilfe wurde mit der in HTML5 eingeführten `requestAnimationFrame()` Methode geschafft [Web 34], die im Folgenden konkretisiert wird. Die Funktion wird im Automobilkonfigurator über die `renderLoop()` Methode aufgerufen und gestaltet sich wie folgt: Die rAF-Methode befindet sich selbst in der WebGL-API und bekommt neben dem Element, auf das die Wiederholungsrate angewendet werden soll, zusätzlich die Callback-Funktion übergeben, welche in einer vom Browser vorgegeben Zeitspanne zyklisch aufgerufen wird:

```
WebGLUtils.requestAnimationFrame(dom_element, callback_function)
```

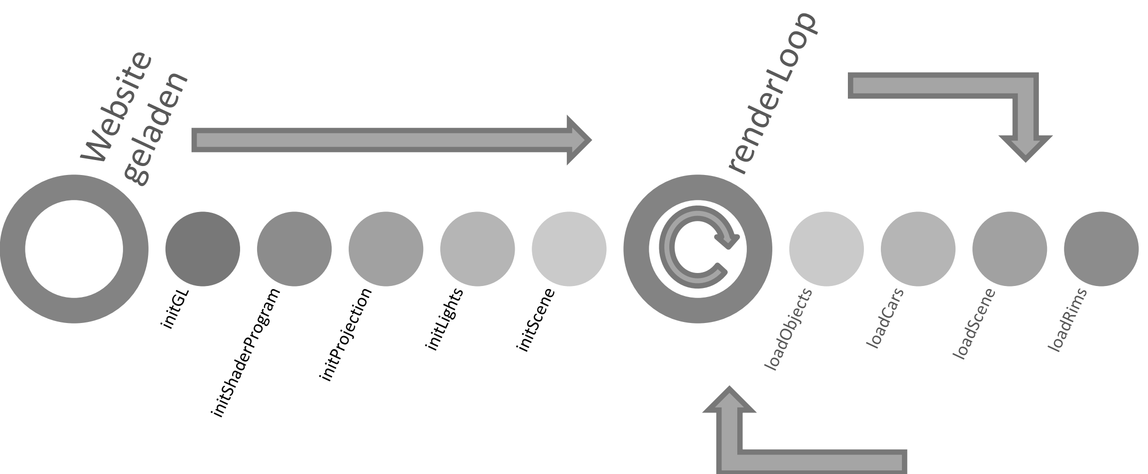
Die Methode birgt den Vorteil, dass sie nur dann aufgerufen wird, wenn das betroffene Browser-Tab im Sichtbereich des Nutzers liegt. Somit wird sichergestellt, dass Hardware-Ressourcen nur dann beansprucht werden, wenn sie gemäß „Call-by-need“ wirklich benötigt werden. Im Funktions-Code signalisiert der `window`-Identifier den Bezug auf das aktuell sichtbare Browser-Fenster (Tab). Die unten beschriebenen Strings geben die möglichen Funktionsnamen für die `requestAnimationFrame` Methode an, da sie abhängig vom Browser unterschiedlich in der API benannt wird.

```
var functionNames = [
 "requestAnimationFrame","webkitRequestAnimationFrame",
 "mozRequestAnimationFrame", "oRequestAnimationFrame", "msRequestAnimationFrame"
];
```

Dieses String-Array wird entsprechend der Gültigkeit überprüft und falls die Methode im Quellcode der Website, die im aktuellen Fenster (`window`) angezeigt wird, vorhanden ist, auch ausgeführt. Dabei ist auf die `setTimeout()` Methode zu achten, die hier mit dem Ziel, eine gewünschte Framerate von 60fps zu gewährleisten, gesetzt ist. Der folgende Link leitet zu einem Echtzeit-Vergleichstest weiter, der die Leistungsfähigkeit der beiden Timer Varianten `setTimeout()` und `requestAnimationFrame()` darstellt [Web 35].

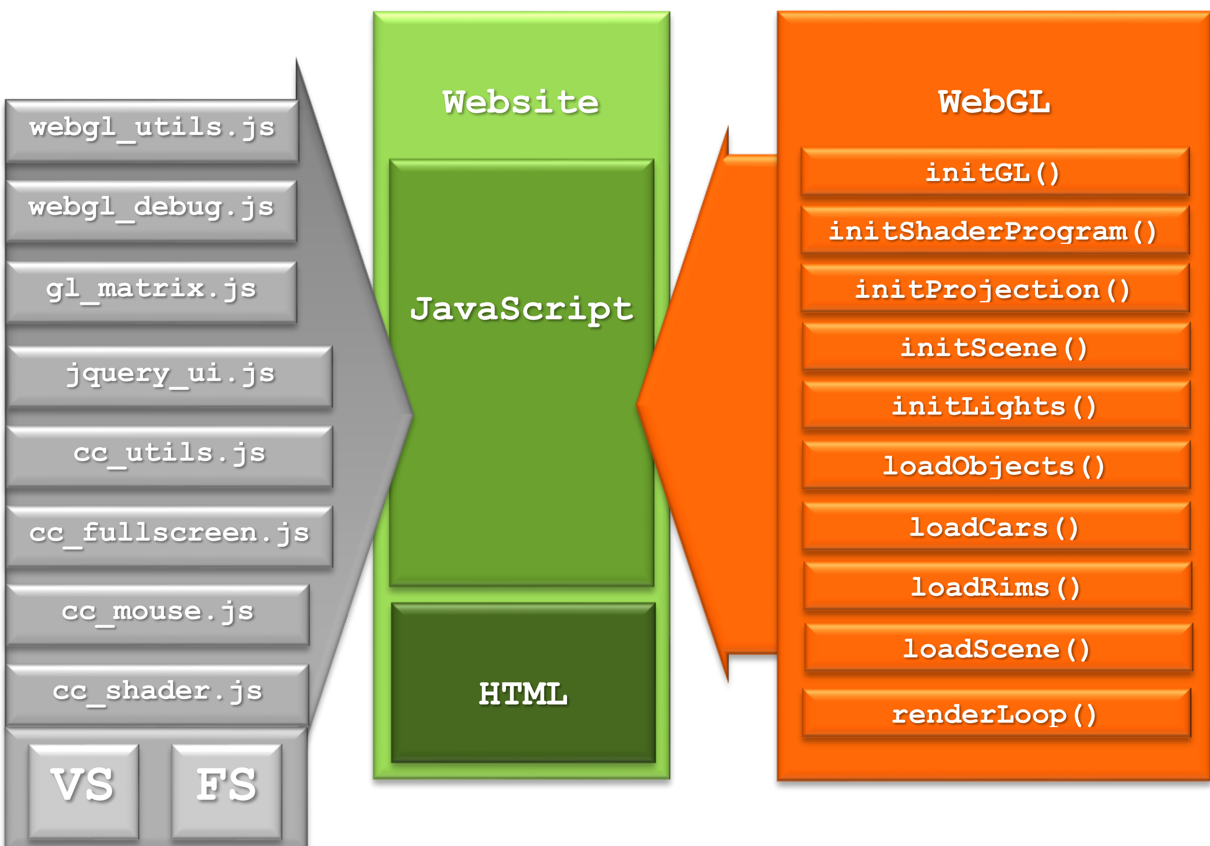
12. IMPLEMENTATIONSDETAILS

Der Ablauf sowie ein detaillierter Blick in einige Implementationen des WebGL-Konfigurators, bildet das letzte große Kapitel dieser Arbeit und beschäftigt sich dahingehend mit einer funktionsorientierten Betrachtung der Anwendung, sowie dem Prozess, den die Applikation während der Ausführung im Browser durchläuft. Durch die Anfrage des Nutzers wird die Website geladen. Der Initialisierungsprozess, bestehend aus den Methoden `initGL`, `initShaderProgram`, `initProjection`, `initLights`, `initScene`, wird automatisch gestartet und erzeugt den Showroom. Nachfolgend wird die `renderLoop` in einer Dauerschleife ausgeführt und registriert innerhalb dieser Methode sämtliche Eingaben des Nutzers. Neben den Mauseingaben, welche sich auf den Blickwinkel des Betrachters auswirken, werden hier ebenfalls Fahrzeugwahl (`loadObjects` / `loadCars`), Felgenwahl (`loadRims`) und Lackierungswechsel registriert. Nach der vollständigen Ausführung einer Methode wird die `renderLoop` weiter ausgeführt. Diese Prozesskette gestaltet sich entsprechend des nachfolgenden Ablaufdiagramms.



Neben der Betrachtung der einzelnen Funktionen, die zum erfolgreichen Ablauf der Anwendung nötig sind, liegt der Fokus dieses Kapitels auch auf dem inneren Aufbau, der Programmstruktur der Anwendung. Grundsätzlich ist die Anwendung in drei Bereiche aufgeteilt, die selbst aus mehreren Bausteinen bestehen. So gibt es die Website selbst, die neben dem HTML-Code auch den JavaScript Code beinhaltet, welcher den WebGL-Bereich darstellt. Darüber hinaus existieren die Skripte, die für sich genommen einen weiteren

Sektor darstellen, der den JavaScript-Code unterstützt bzw. modularisiert. Das nächste Struktogramm beschreibt die Programmstruktur des WebGL-Konfigurators in übersichtlicher Form:



12.1. HTML

Den geringsten Teil der Anwendung macht der HTML-Code aus, der lediglich das Rahmenwerk des Layouts der Webseite bildet, in dem darüber hinaus der JavaScript-Code der WebGL-Anwendung sowie die einzelnen interaktiven Komponenten eingebettet ist. Auf das Layout und die implementierten Elemente wurde bereits in Kapitel 11.1 - Die Benutzeroberfläche detailliert eingegangen.

12.2. SKRIPTE

Wie in Kapitel 7 - JavaScript erwähnt, werden stets wiederkehrende oder sicherheitstechnisch sensible Funktionen und Daten in eigene JavaScript-Dateien

ausgelagert, nicht zuletzt auch aus Gründen der Modularisierung und Übersichtlichkeit des Codes der Hauptanwendung. Um einen detaillierten Überblick über die unterstützenden Skripte zu geben, werden die wichtigsten hier im Folgenden erläutert. Dabei wurden Skripte eigens konzipiert und implementiert, aber auch bestehende verwendet bzw. Teile von bestehenden Skripten übernommen.

Ein grundlegendes Skript stellt die `WebGL-Utils.js` Datei dar. Dieses ist für jede WebGL-Anwendung notwendig, da sämtliche Initialisierungsfunktionen, die zum Erstellen eines WebGL-Kontextes im Canvas-Element nötig sind, in diesem enthalten sind. Des Weiteren werden Browserkompatibilität und Initialisierungsprobleme bezüglich der WebGL-Konformität frühzeitig überprüft und dem Anwender über Konsolenausgaben signalisiert. Bereitgestellt wird dieses Skript auf der Webpräsenz der Khronos-Group unter dem folgenden Link [Web 36].

WebGL bietet selbst kein automatisches Debugging an, solange kein externes Debugging-Skript diesen Part übernimmt. An dieser Stelle kommt das `WebGL-Debug` Skript zum Einsatz, das über eine Funktion in der Lage ist, sämtliche OpenGL Funktionsaufrufe in der Konsole des Browsers auszugeben. Hierzu muss lediglich die Initialisierung des WebGL-Kontextes mit dem Debug-Kontext umschlossen werden (*wrappen*), was sich wie folgt gestalten lässt:

```
gl = WebGLDebugUtils.makeDebugContext(

                     canvas.getContext("experimental-webgl"),
                     throwOnGLError,
                     logGLCall

);
```

Allerdings sei explizit darauf hingewiesen, dass dieser Kontext bei der Veröffentlichung des Programms im Web entfernt werden sollte, da sämtliche Logging-Ausgaben in den Arbeitsspeicher geschrieben werden und dies zu enormen Systemauslastungen und Performance-Einbußen führt. Das Debugging-Skript wird ebenfalls von der Khronos-Group weiterentwickelt und zur Nutzung unter dem folgendem Link [Web 37] freigegeben. Sämtliche mathematischen Funktionen befinden sich im `GL-Matrix` Skript. Dadurch wird dieses Skript unerlässlich bei der Implementierung von Kamera- und Lichtmodellen, aber auch für ganz einfache Matrix-Transformationen. Sämtliche Transformationen liegen für Vektoren mit vier Koordinaten und Matrizen bis zur Dimension 4x4 vor. Die Entwicklung

sowie der öffentliche Download des Skripts werden von Bryan Jones vorangetrieben. Darüber hinaus berichtet er auf seinem Blog ausführlich über Neuigkeiten rundum das Projekt GL-Matrix. Der Blog ist unter dem folgenden Link zu finden [Web 38].

Die `JQuery UI` (*UI = user interface*) ist eine erweiternde Bibliothek für JQuery zur Gestaltung und Funktionserweiterung einer Website. Neben den bekannten Funktionen, um Objekte *drag*- bzw. d*ropable* zu setzen oder sogenannte *Widgets* (Kalender, Terminplaner, etc.) einzubinden, wird beim Automobilkonfigurator von der *„Progressbar“* Gebrauch gemacht. Diese ist im Programm an die `loadCars()` Methode gebunden, die über einen Zähler im Verhältnis zur Gesamtanzahl der Fahrzeugteile die entsprechende Prozentzahl dem Ladebalken übergibt. So wird dem Anwender während der Ausführung der Anwendung bereits der Lade-Fortschritt seiner Konfiguration entsprechend angezeigt. Die folgende Abbildung zeigt den Ladebalken während des Ladeprozesses:

ABBILDUNG 42: LADEBALKEN

Das `cc_utils` Skript beinhaltet quelltextintensive Funktionen, die den Code der Hauptanwendung unnötig vergrößern. Funktionen für MouseOver-Effekte der Menüschaltflächen, sowie die immerwährende Aktualisierung der Transformationsmatrizen und Geometrievariablen der Szene sind elementare Funktionen dieses Skripts. Hinzu kommt die Lackierungsfunktion. Hier werden nach Aktivierung der entsprechenden Events im Menübereich der Website (Farbauswahl), gezielt die Uniformvariablen der diffusen Materialkonstante farblich an die Wahl des Benutzers angepasst und dem Hauptprogramm übergeben.

Um eine WebGL-Vollbildanzeige zu ermöglichen, wird die in HTML5 eingeführte Fullscreen API mit der `cc_fullscreen.js` in die Arbeit integriert. Ursprünglich war die Vollbildanzeige nur Flash-Anwendungen und Video-Elementen zugesprochen. Browser hielten sich lange aufgrund von Sicherheitsbedenken zurück, da eine Anwendung, die in den Vollbildmodus gebracht werden kann, dem Nutzer jegliche Kontrolle über Browserleisten und einige betriebssysteminterne Funktionen für den Moment entzieht [Web 39]. Mit HTML5 ist es allerdings möglich, gezielte Elemente im DOM-Tree in den Vollbildmodus zu

schalten, ohne dass dabei der Benutzer die Kontrolle über wichtige Systemfunktionen verliert. Unter diesen Gegebenheiten lässt sich auch das Canvas-Element, welches die WebGL-Renderingfläche darstellt, in den Vollbildmodus schalten. Die gesamte Fullscreen API ist auf der Entwicklerseite von Mozilla frei erhältlich [Web 40].

Die neueren Browserversionen unterstützen weitestgehend die API. Für weitere Kompatibilitätsinformationen soll der folgende Tabellenausschnitt Hilfestellung bieten:

Firefox	Chrome	Safari	Opera
19.0	25.0	5.1	12.1
20.0	26.0	6.0	12.5
21.0	27.0		

TABELLE 8: KOMPATIBILITÄT DER FULLSCREEN API
QUELLE: [WEB 41]

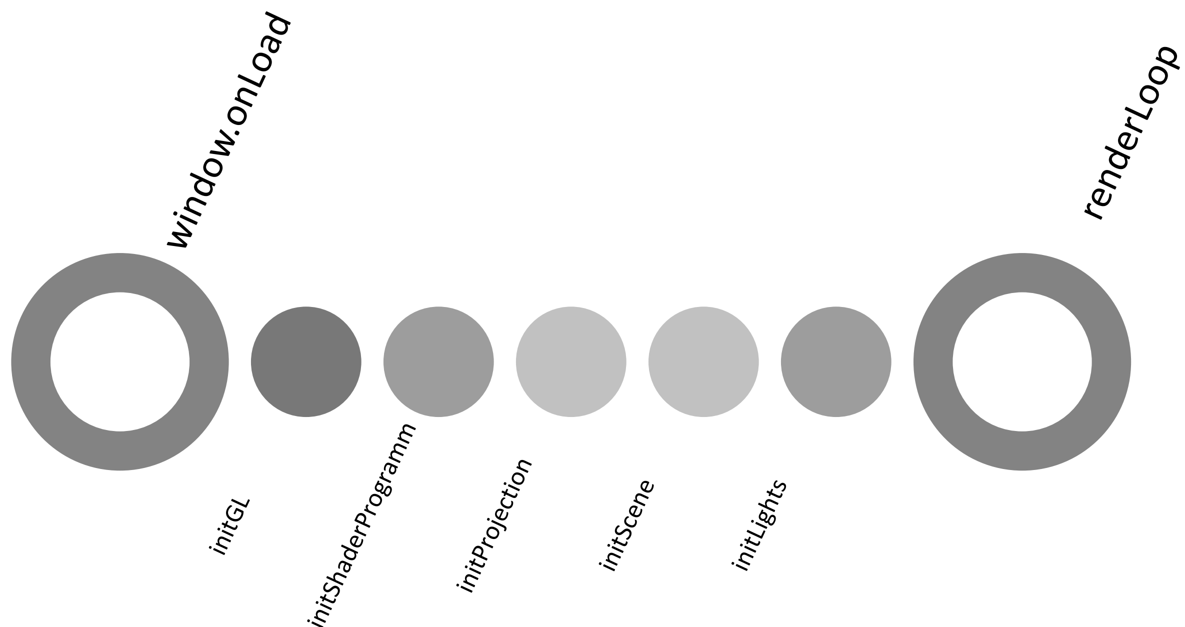
Um die Interaktion mit der Anwendung über die Maus als eigenes Modul des Programms zu betrachten und gesondert von anderen Bereichen handhaben zu können, wurde das `cc_mouse` Skript entwickelt. Mit dem Ziel explizit Mausinteraktionen nur auf dem Canvas-Element zu registrieren und behandeln zu können, wird in der Hauptfunktion des Programms initialisiert, welche Operationen auf dem Canvas-Element zulässig sind. Die Implementation, sowie der Zweck der jeweiligen Funktionen ist aufgrund der Namensgebung zu erahnen und kann im Detail im Quellcode nachgelesen werden. Eine Anmerkung gilt der `mouseMove()` Methode, welche für das Setzen einzelner Parameter der Modelmatrix verantwortlich ist. So beeinflusst beispielsweise diese Funktion je nach Bewegungsrichtung das Schwenken der virtuellen Kamera nach oben und unten (mit der Einschränkung, dass nicht unter das Fahrzeug gesehen werden kann), sowie nach links und rechts. Die Zoom-Möglichkeit ist mit der `mouseWheel()` Funktion verknüpft.

Das letzte nötige Skript verarbeitet die `Shader` und wurde ebenso wie das `Utils` Skript zur Reduzierung des Quelltextumfangs der Hauptanwendung erstellt. Darüber hinaus kommt speziell bei den Shadern der verstärkte Sicherheitsgedanke zu tragen. Wären hier Shader zum Einsatz gekommen, deren Logik vor der Öffentlichkeit unter Verschluss gestellt werden müsste, wäre diese Variante der Umsetzung ohnehin die einzig empfehlenswerte, da ein separates Skript für die Shader auch unter separate Sicherheitsmodalitäten gestellt werden

kann. Dieses Skript beinhaltet demnach den reinen Quelltext des Vertex- sowie des Fragment Shaders in GLSL und den Funktionscode zur Erstellung des Shader-Programms. In den Shadern selbst ist das Phong-Modell implementiert, dessen Funktionsweise in Kapitel 6.5 – Phong Lighting näher geschildert wurde.

12.3. Die Anwendung

Beim Aufruf der Website im Browser wird automatisch durch die `window.onLoad` Funktion die Main-Methode der WebGL-Anwendung gestartet und so die Abfolge der unten aufgeführten Funktionen angestoßen. Zu Beginn finden somit alle Initialisierungen, die für WebGL nötig sind, sowie jene zur Darstellung der Standard-Szene, statt. Diese Befehlskette läuft gemäß dem nachfolgenden Schema ab und bleibt am Ende offen für weitere Nutzereingaben hinsichtlich Fahrzeugwahl und Konfigurationen.

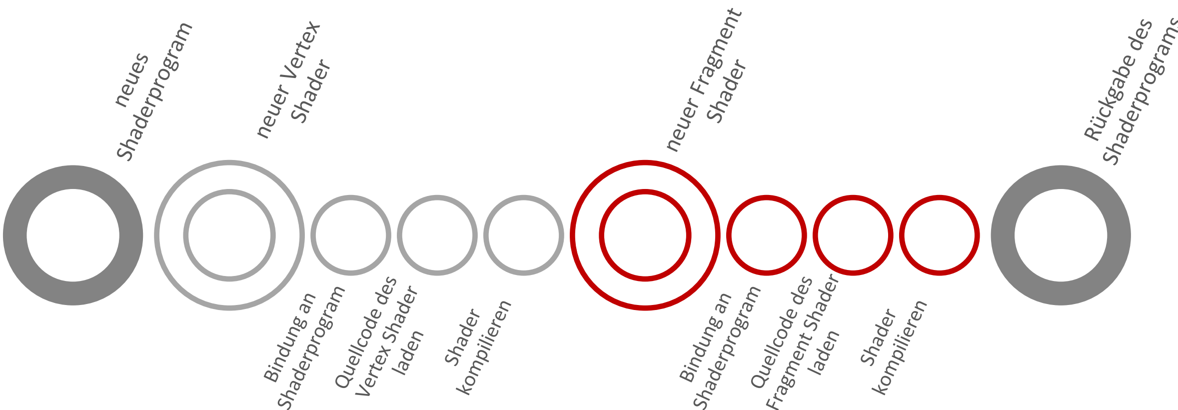


initGL

Die erste Methode, die mit Ausführung der Main-Funktion der Hauptanwendung gestartet wird, initialisiert das Canvas-Element als WebGL-Kontext gemäß dem Vorgehen in Kapitel 9.4 – Initialisierung des WebGL-Kontextes.

initShaderProgram

Diese Funktion initialisiert, analog zur Methode „createShaderProgram“ des HelloTriangle Beispiels auf Seite 33, das Shader Programm. Die einzelnen Stufen der Generierung des Shaderprogramms sollen durch das folgende Prozessdiagramm nochmals übersichtlich dargestellt werden:



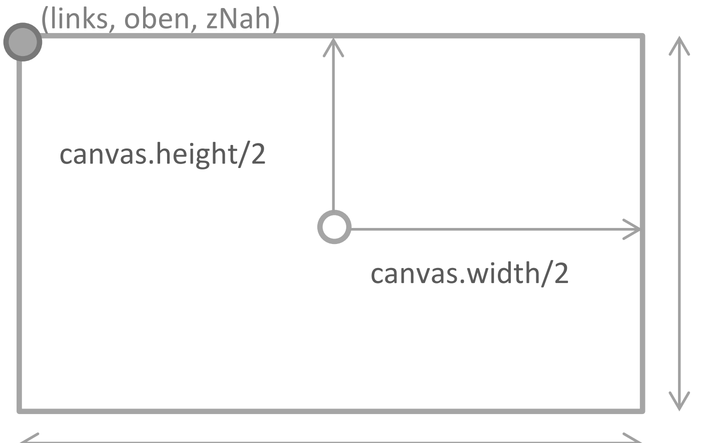
Auf Grund der globalen Deklaration liegt das Shaderprogramm ab diesem Zeitpunkt in der gesamten WebGL-Anwendung vor und kann von jeder Methode verwendet werden.

initProjection

Aus Kapitel 7.2 – Struktur einer WebGL-Applikation geht hervor, dass das Canvas-Element genau dem Viewport d.h. dem Betrachtungsfenster des Nutzers entspricht. Im Rahmen dieser WebGL-Anwendung werden zwei verschiedene Perspektiven (orthogonal und perspektivisch) angeboten, sodass mit Beginn dieser Funktion eine initiale Perspektive gesetzt wird. Die `initProjection` Methode bekommt durch die Belegung der im HTML-Code gesetzten Radio-Buttons die entsprechende ID der gewählten Perspektive und kann somit im Rahmen einer `switch-case` Struktur ermitteln, welche Perspektive geladen werden muss. Der nachfolgende Quellcode beschreibt einen der Radio-Buttons mit der für die Funktion wichtigen ID der gewählten Perspektive.

```
<input type    = "radio"
       id      = "persproj"
       name    = "Projektion"
       value   = "perspective"
       onclick = "initProjection(), drawScene()" checked>
```

Die optische Aufteilung in Bezug auf das Canvas-Element kann wie folgt dargestellt werden



canvas.height (in der Anwendung: 480px)

canvas.width (in der Anwendung: 818px)

Dabei finden die nachfolgenden Werte im WebGL-Konfigurator Verwendung und werden entsprechend in die Projektions-Matrizen des `gl-Matrix` Skripts geladen und zurückgegeben.

```
n = zNah      =   0.1
f = zFern     =   100
l = links     =   -(canvas.width/2)
r = rechts    =   (canvas.width/2)
t = oben      =   (canvas.height/2)
b = unten     =   -(canvas.height/2)
viewport      =   (canvas.width, canvas.height)
```

Für ein und dieselbe Betrachterposition ergeben sich somit die folgenden Perspektiven:

Perspektivische Projektion	**Orthogonale Projektion**

Ausgabe

ABBILDUNG 43: VERGLEICH DER PROJEKTIONEN

initLights

Im nächsten Schritt werden die Lichter an die Ecken der Bodenplattform gesetzt. Die dafür benötigten Koordinaten wurden in Blender bereits ermittelt. Im Programm werden diese in einem Array `lights` gespeichert, wobei jedes Licht seinerseits ebenfalls aus einem Array besteht, um die Übergabe als Uniform-Array an die Shader später zu vereinfachen. Die Struktur des Arrays ist am Beispiel - einer der vier Lampen - wie folgt zu verstehen: Wird ein erstes Licht deklariert, so werden seine Komponenten in das `light1[]` eingefügt.

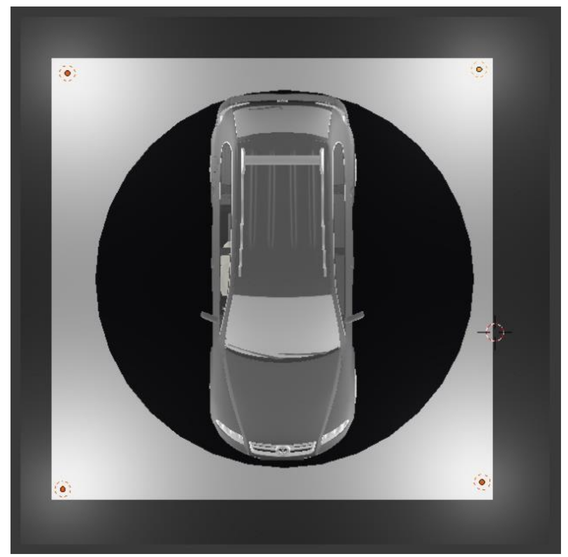


ABBILDUNG 44: LICHTER DER SZENE

```
var light1 = [];
var light1_pos = [-7.0, 3.0, -7.0];
var light1_ambient = [0.0, 0.0, 0.0];
var light1_diffuse = [0.9, 0.9, 0.9];
var light1_specular = [0.8, 0.8, 0.8];
```

Mit Blick auf die Gesamtszene (siehe Abbildung 44) befinden sich in den vier Ecken der Bodenplatte die vier Punktlichter, die die Szene ausleuchten.

initScene

Die Szene, die sich beim Laden der Website dem Betrachter präsentiert, wird als Teil der Funktionenkette automatisch über die `initScene()` Methode erzeugt. Hier wird zum ersten Mal in der Anwendung ein ins JSON-Format geparstes 3D-Modell aus dem entsprechenden Verzeichnis auf dem Server geladen und in die Buffer transferiert. Dies geschieht asynchron über einen Ajax-Request, der sich in der `loadScene()` Methode wie folgt gestaltet:

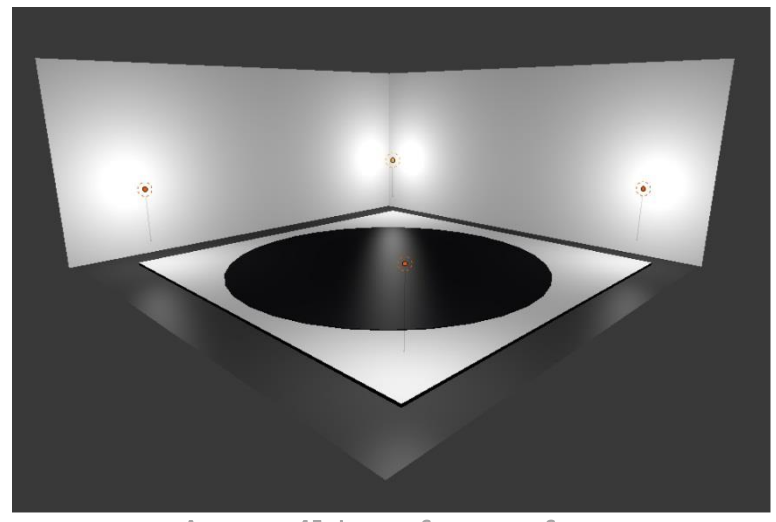


ABBILDUNG 45: INITIALE SHOWROOM-SZENE

Zunächst wird ein `XMLhttpRequest` Objekt erzeugt, welches im weiteren Verlauf die Anfrage an den Server darstellt. Dieses Objekt bringt die Funktion `open` mit, die über die beiden Parametern „`GET`“ und dem Dateipfad die Anfrage zusammenbaut und an den Server übermittelt. Ebenfalls ist das `request`-Objekt mit einer State-Variablen ausgestattet, welche Auskunft über den aktuellen Zustand der Datenanforderung und –übertragung geben kann.

Neben den möglichen Zuständen

(0) – `XMLhttpRequest`-Objekt instanziiert,
(1) – Zustand nach Aufruf der `open`-Methode (Zusammenstellung des Requests)
(2) – Zustand nach Aufruf der `send`-Methode (Absetzen des Request)
(3) – Datenübertragung läuft

wird hier insbesondere der Zustand (4), ob die Daten des Requests erfolgreich vom Server geladen wurden, als Signal zur Befüllung der Buffer, abgefragt. Eine weitere Sicherheit über das Vorhandensein einer Datei auf dem Server kann über `request.status` abgefragt werden. Gibt der Server nach Anfrage durch den Nutzer den Fehlercode `404` zurück, existiert die Datei im gewählten Verzeichnis nicht. Signalisieren sämtliche Werte, dass eine erfolgreiche Übertragung der JSON-Datei stattgefunden hat, so können die Buffer über die `initBuffers(jsonData)` Methode verlustfrei befüllt werden.

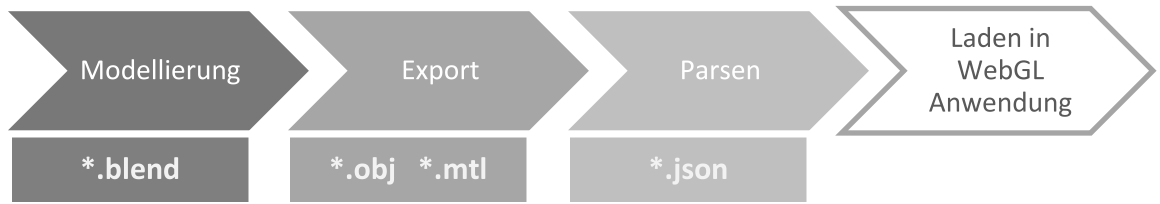
initBuffers

Die `initBuffers(jsonData)` Methode besitzt einen Übergabeparameter. Bei diesem Wert handelt es sich um eine zum JavaScript Objekt geparste JSON Datei. Dadurch kann im weiteren Programmablauf mit einer „JSON-Datei“ wie mit einem normalen JavaScript-Objekt gearbeitet werden, was die Syntax enorm vereinfacht. Beispielsweise kann auf die Vertices einer JSON Datei einfach mit `jsonData.vertices` zugegriffen werden, vorausgesetzt

das Array mit den Vertices heißt bereits entsprechend in der JSON-Datei. Der grundlegende Ablauf der Methode erzeugt pro übergebenen JSON-Datensatz einen Vertex-, Normalen- und Indexbuffer und lädt die entsprechenden Daten hinein. Neben der Erstellung der Buffer pro Datensatz, wird jedes JSON-Objekt im zu Beginn der WebGL-Anwendung global deklarierten `part`-Array gespeichert. Dieses Array wird später für das korrekte Setzen der Material-Uniform-Variablen benötigt. Darüber hinaus wird jedes Bufferobjekt in ein entsprechendes Array gespeichert, um später nacheinander sämtliche Datensätze rendern zu können.

loadObjects(id)

Wird die `loadObjects(id)` Methode aufgerufen, sind bereits alle Initialisierungsvorgänge abgeschlossen und die Showroom-Szene wird in der Frontalansicht präsentiert. Der Prozess, den die in Blender modellierten Fahrzeuge bis zu diesem Zeitpunkt durchlaufen haben, soll noch einmal an dem folgenden Ablaufdiagramm deutlich gemacht werden:



Die loadObjects(id) Methode hat nun die selektierende Aufgabe, aus den Anfragen des Nutzers die korrekten Datenpfade zu konstruieren, die dann an die eigentlichen Ladefunktionen übermittelt werden. Innerhalb einer `for`-Schleife werden sämtliche JSON-Datensätze eines angeforderten Objekts (Auto, Spoiler, Felge, etc.), dessen Gesamtanzahl dem Programm über die Variable `partAnz` bekannt ist, durchlaufen und der konstruierte Datenpfad der entsprechenden Lademethode übergeben. Die Variable `object_selection` besitzt dabei die ID des vom Nutzer gewählten Elements, welches namentlich gleichgesetzt ist mit der Bezeichnung des Verzeichnis in dem beispielsweise die Daten des VW Up! liegen.

loadCars

Die loadCars() Methode wird wie eben erwähnt immer dann aufgerufen, wenn der Nutzer ein Fahrzeug auswählt. Im Folgenden wird ein beispielhafter Aufruf abgehandelt, der neben der Systematik der Funktion auch die Handhabung der ID in Verbindung mit dem Dateiverzeichnis zeigt. Dazu betrachte man die folgende Abbildung:

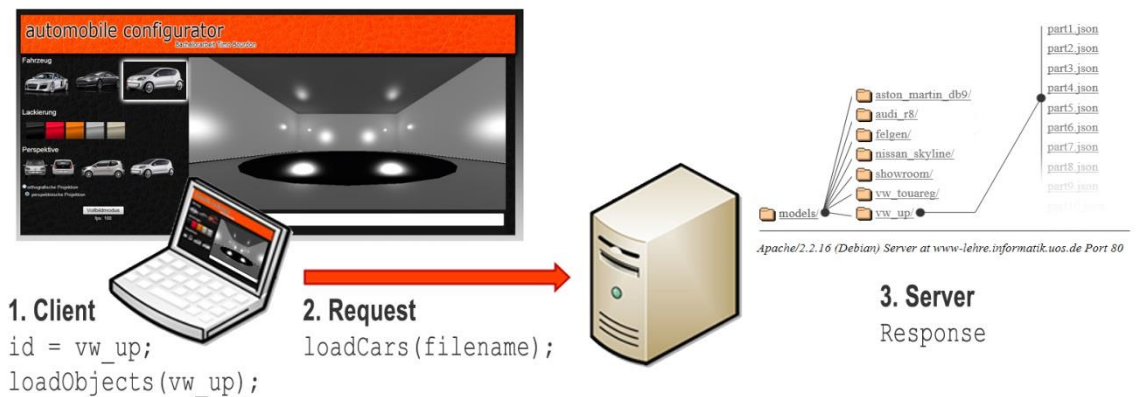


ABBILDUNG 46: LOADCARS() METHODE

Während die WebGL-Anwendung im Browser läuft, wählt der Nutzer hier den VW Up und stößt damit die `loadObjects()` Methode an, die wiederum als Übergabeparameter die ID `vw_up` besitzt. Diese ist im HTML-Code als die ID des Fahrzeugbildes deklariert. Mit ihr wird, wie bei der vorangegangenen loadObjects Methode erläutert, der Dateipfad erstellt, der angefordert wird. Der Request selbst sowie das anschließende Befüllen der Buffer läuft entsprechend der Umsetzung der `loadScene()` Methode ab und besitzt lediglich eine Ausnahme bezüglich der Buffer. Nachdem die Showroom-Szene, die aus insgesamt 8 JSON-Datensätzen (d.h. jeweils 8 Vertex-, Normalen- und Indexbufferobjekte sowie 8 Einträge im `part`-Array) besteht, geladen ist, findet sich folgender Zustand des Arrays:

Diese Daten dürfen aber keinesfalls überschrieben werden, da das vom Nutzer gewählte Fahrzeug zusätzlich in die Szene gerendert werden soll. Aus diesem Grund muss überprüft

werden, ob vor dem Laden eines neuen Fahrzeugs sämtliche Arrays mit der Szene bereits befüllt sind und entsprechend alle weiteren Datensätze ab Stelle [8] in die Arrays eingefügt werden. Die Struktur, die diesen Fall überprüft besitzt dabei die Form:

```
if(vbo.length > 0 && nbo.length > 0 && ibo.length > 0 && part.length > 0){
                         vbo.splice(8,vbo.length);
                         ibo.splice(8,ibo.length);
                         nbo.splice(8,nbo.length);
                         part.splice(8,part.length);
}
```

An dieser Stelle wird von der `splice` Funktion Gebrauch gemacht, die es ermöglicht, gezielte Stellen oder Bereiche in einem JavaScript Array zu löschen bzw. Stellen oder Bereiche zu füllen. In sämtlichen Arrays wird so der erste Fahrzeug-JSON-Datensatz an Stelle [8] eingefügt bzw., wenn bereits ein Fahrzeug geladen ist, sämtliche Dateninhalte der Arrays ab dieser Stelle gelöscht, so dass sicher gestellt werden kann, dass die Szene keine weiteren Fahrzeuge außer dem Gewählten enthält. Ist ein Fahrzeug vollständig geladen, ändert sich der Inhalt der Arrays wie folgt:

Um diese recht einfache Darstellung zu konkretisieren, kann es hilfreich sein Abbildung 47 auf der folgenden Seite zu betrachten. Hier wird mit Hilfe des Firefox-Addons *Firebug* (Werkzeug zur Webanalyse und Webentwicklung) ein detaillierter Einblick in den Inhalt verschiedenster Arrays gegeben - im speziellen dem `part`-Array an dieser Stelle.

Konsole HTML CSS Skript DOM Netzwerk Cookies

window

part	[Object { alias="Raum_Podest_Cube_waende", vertices=[24], indices=[36], mehr... }, Object { alias="wand_links_Plane.002_waende", vertices=[12], indices=[6], mehr... }, Object { alias="wand_rechts_Plane.003_waende", vertices=[12], indices=[6], mehr... }, 5 mehr...]
0	Object { alias="Raum_Podest_Cube_waende", vertices=[24], indices=[36], mehr... }
1	Object { alias="wand_links_Plane.002_waende", vertices=[12], indices=[6], mehr... }
2	Object { alias="wand_rechts_Plane.003_waende", vertices=[12], indices=[6], mehr... }
3	Object { alias="wand_hinten_Plane.008_waende", vertices=[12], indices=[6], mehr... }
4	Object { alias="wand_decke_Plane.007_waende", vertices=[12], indices=[6], mehr... }
5	Object { alias="wand_vorne_Plane.006_waende", vertices=[12], indices=[6], mehr... }
6	Object { alias="Raum_Boden_Plane.004_boden", vertices=[12], indices=[6], mehr... }
Ka	[0, 0, 0]
Kd	[0.06401, 0.06401, 0.06401]
Ks	[0.10486, 0.10486, 0.10486]
Ni	1
Ns	96.07843
alias	"Raum_Boden_Plane.004_boden"
d	1
illum	2
indices	[1, 0, 2, 3 mehr...]
vertices	[8.339585, -1.009243, 8.399589, 9 mehr...]
7	Object { alias="Cylinder_Cylinder.002_kreis", vertices=[576], indices=[1140], mehr... }
8	Object { alias="orange0_orange0_orange", vertices=[2910], indices=[4800], mehr... }
9	Object { alias="chrome05_chrome05_chrome", vertices=[10899], indices=[20640], mehr... }

ABBILDUNG 47: PART-ARRAY VIA FIREBUG

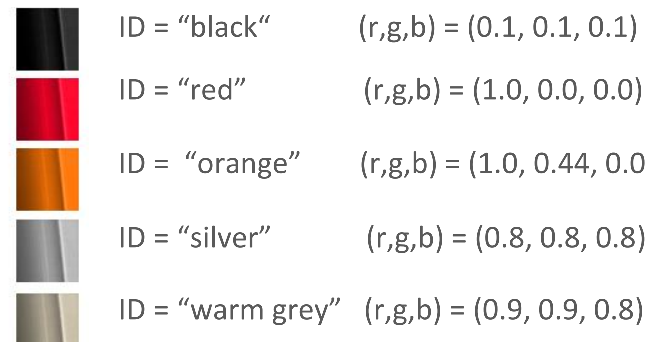
Zu erkennen ist die erwähnte Aufteilung zu Beginn des Arrays, welche die Stellen 0 bis 7 des Arrays für die Showroom-Szene vorsieht und alle weiteren Plätze, je nach Anzahl der Datensätze, mit denen des Fahrzeugs belegt. Insbesondere zeigt die Abbildung die weitere Verschachtelungssystematik bezüglich des Datensatzes an Stelle 6, welcher die Attribute des JSON-Objekts wiederspiegelt. Darunter finden sich unter anderem die Materialkonstanten (*ambient, diffuse, specular*) sowie die *Vertices* und *Indices* wieder.

loadRims

Zur Funktion der Felgenauswahl eines jeden Fahrzeugs ist es wichtig zu erwähnen, dass es sich lediglich um Pläne der Umsetzung handelt, da die entsprechende Logik noch nicht implementiert ist. Als erstes müssen, vergleichbar mit dem Vorgehen bei der später folgenden Lackierungsfunktion, sämtliche Datensätze, die Felgendaten enthalten, ermittelt und aus den vier Arrays gelöscht werden. Im Anschluss daran können die neuen Felgen geladen werden, wobei entsprechend ihrer ID ein Dateipfad erstellt wird, der auf den entsprechenden Felgensatz des zugehörigen Autos verweist. Die `loadRims()` Methode, die genau diesen Ablauf steuern soll, wird Teil des `cc_utils` Skript, da ebenso wie bei der `updateColor()` Methode zur Lackierung der Fahrzeuge, sehr viel Programmcode zur Beschreibung der zu löschenden Datensätze entstehen wird und dieser das Hauptprogramm nur unnötig vergrößert.

updateColor

Die Veränderung der Lackierung ist eine rein clientseitige Operation, die zu keinem Zeitpunkt mit dem Server oder den Daten darauf interagiert. In der WebGL-Anwendung findet sich diese Funktion auf Grund der Funktionslänge im `cc_utils` Skript wieder.



Wählt der Nutzer eine der aufgeführten Farben aus, startet er automatisch die Funktion, welche die ID der Farbe mitübergeben bekommt.

In der Methode wird eine nach den Farben modularisierte `switch-case` Struktur durchlaufen. Um ausschließlich die Fahrzeugteile der Karosserie zu lackieren, macht man sich an dieser Stelle den Namen des Materials zu Nutze, welches für jede Fahrzeugkomponente definiert ist. Jedes Fahrzeugteil existiert als eigenes Objekt im `part`-Array, wobei sich der zugehörige Name stets aus dem der Geometrie und dem daran verknüpften Material zusammensetzt. So kann gezielt mit der `indexOf` Funktion überprüft werden, ob der Material-Name ein Teilstring des Gesamtnamens ist. Wenn dies der Fall ist, wird die entsprechende Uniformvariable der diffusen Materialkonstanten mit dem neuen Farbwert überschrieben, welcher beim nächsten Rendering-Durchgang berücksichtigt wird.

drawScene

Die wichtigste Routine der WebGL-Anwendung ist die Rendering-Schleife, die die Szene im Canvas-Element ausgibt. Diese ist dabei so eingestellt, dass sie möglichst alle 60ms erneut aufgerufen wird. Somit werden im Mittel pro Sekunde ca. 16 Frames möglich, wobei dieser Wert insbesondere während des Ladeprozesses eines Fahrzeugs variieren kann. Zu Beginn der Operation werden die Transformationsmatrizen auf Zustandsveränderungen überprüft, die vom Nutzer via Mausbewegung initiiert werden. In Abhängigkeit davon, ob dieser das Fahrzeug dreht oder heranzoomt, verändern sich die Matrizen und wirken sich so direkt auf die Perspektive der Szene aus. Dieser Faktor muss bei jedem Rendervorgang berücksichtigt werden. Ebenso wird überprüft, ob sich der Nutzer im Fullscreen-Modus befindet oder nicht,

da entsprechend auch die Szene neu skaliert und ausgegeben werden muss um Unschärfe-Artefakte zu verhindern.

Innerhalb einer `for`-Schleife wird das `part`-Array durchlaufen und die Uniformvariablen für die Materialkonstanten gesetzt, sowie sämtliche Bufferobjekte gebunden und über den `drawElements` Befehl gerendert. Als Ergebnis erscheint das Fahrzeug im Showroom!

13. Performanz Analyse

Die in diesem Kapitel aufgeführten Resultate sind die Ergebnisse einer Leistungsstudie in Bezug auf Initialisierung der Anwendung, sowie der traffic-intensivsten Methode, der loadCars Funktion. Zur Analyse und Berechnung der Zeiten wurde das browserinterne Entwicklertool von Google Chrome mit Hinblick auf die folgende Hardware-Konfiguration des Smartphones, Notebooks und Desktop-PCs herangezogen:

Abbildung 48: Test Hardware

Smartphone	**Samsung Galaxy Wifi 4.0**	***Grafikkarte***	**PowerVR SGX 540**
Baujahr	2012	***Baujahr***	-
CPU	750 MHz	***GPU***	-
RAM	512 MB	***Memory Size***	-
Browser	Firefox Mobile		
Notebook	**HP Probook 4730s**	***Grafikkarte***	**ATI Radeon HD7470**
Baujahr	12/2011	***Baujahr***	12/2011
CPU	Intel Core i5-2410LM	***GPU***	Seymour XT
RAM	4 GB DDR3 SDRAM	***Memory Size***	1024MB
Browser	Chrome 25.0.1364.97 m		
	Firefox 19.0.2		

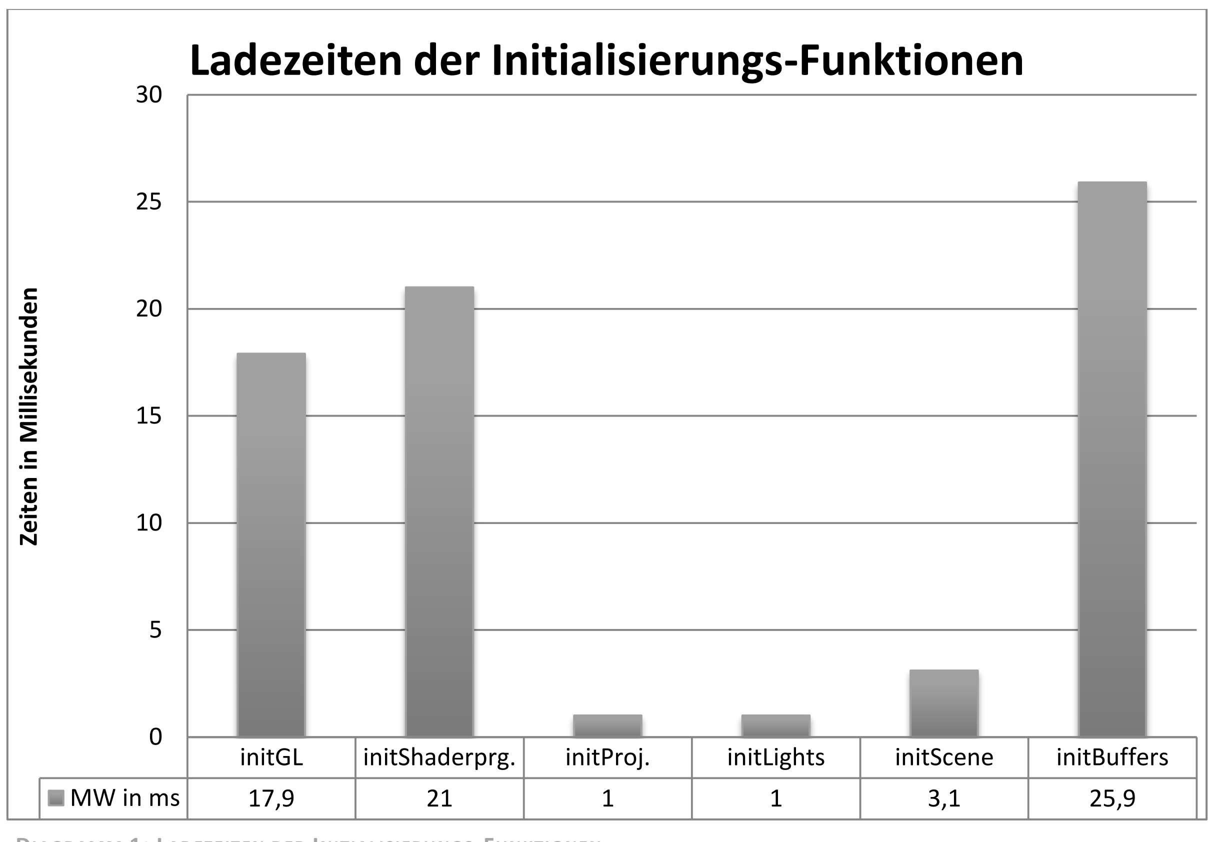
Desktop-PC		*Grafikkarte*	**Nvidea GTX 470**
Baujahr	-	*Baujahr*	04/2010
CPU	Intel Core i7 860	*GPU*	-
RAM	4 GB	*Memory Size*	1280 MB
Browser	Firefox 19.0.2		

Leistungsmerkmale der Netzwerke:

	Heim-Netzwerk		Uni-Netzwerk*	
	LAN	*WLAN*	*LAN*	*WLAN*
Übertragungsstandard	DSL 16.000	DSL 16.000	Breitband	Breitband
Übertragungsrate	65 Mbit/s	65 Mbit/s	100 Mbit/s	65 Mbit/s

*Daten vom Rechenzentrum AVZ: Außenanbindung von 600 MBit/s, realisiert über eine STM-4 Glasfaserleitung der Telekom. Anbindung an das Deutsche Forschungsnetz (DFN e.V.) , redundant an die Kernnetzknoten in Münster und Bielefeld. *Stand: 25.03.2013*

Im ersten Zeitexperiment wurde der gesamte Ablauf aller Initialisierungsfunktionen bis zur vollständigen Darstellung des Showrooms betrachtet und nach dem oben geschilderten Vorgehen analysiert. Die im Diagramm dargestellten Werte charakterisieren jeweils die mittlere Anzahl an Millisekunden die nach 6 Ladevorgängen der Website gemessen wurden. Die Ausführung fand auf dem HP Probook 4730s im WLAN-Betrieb statt.



	initGL	initShaderprg.	initProj.	initLights	initScene	initBuffers
MW in ms	17,9	21	1	1	3,1	25,9

DIAGRAMM 1: LADEZEITEN DER INITIALISIERUNGS-FUNKTIONEN

Im zweiten Zeitexperiment wurde der Ladevorgang der drei vorhandenen Fahrzeugmodelle auf den zuvor aufgeführten Endgeräten untersucht. Dabei wurden als Stichprobe 6 Zeiten gemessen, pro Fahrzeug und Endgerät. Die Gesamtzeit entspricht wieder dem arithmetischen Mittel der jeweiligen Ladezeiten. Während des Ladevorgangs wurde insbesondere auf eine Deaktivierung des browser-internen Cachings geachtet, sodass keine bereits geladenen Daten erneut verwendet wurden, was die Zeitmessung verfälscht hätte.

		loadCars (Ladezeit in Sekunden)		
		Aston Martin DBS	Audi R8	VW Up!
	Vertex-Anzahl	445.076	1.020.059	3.502.866
	Datenvolumen	20,7 MB	56,4 MB	126 MB
Smartphone	WLAN (Heim)	99,8	x	x
	WLAN (Uni)	103,42	221,76	x
Notebook	WLAN(Heim)	12,4	29,8	65,6
	WLAN(Uni)	7,9	14,7	30
	LAN(Heim)	13,1	27,7	61,7
Desktop-PC	LAN(Uni)	9,4	15,9	26,9

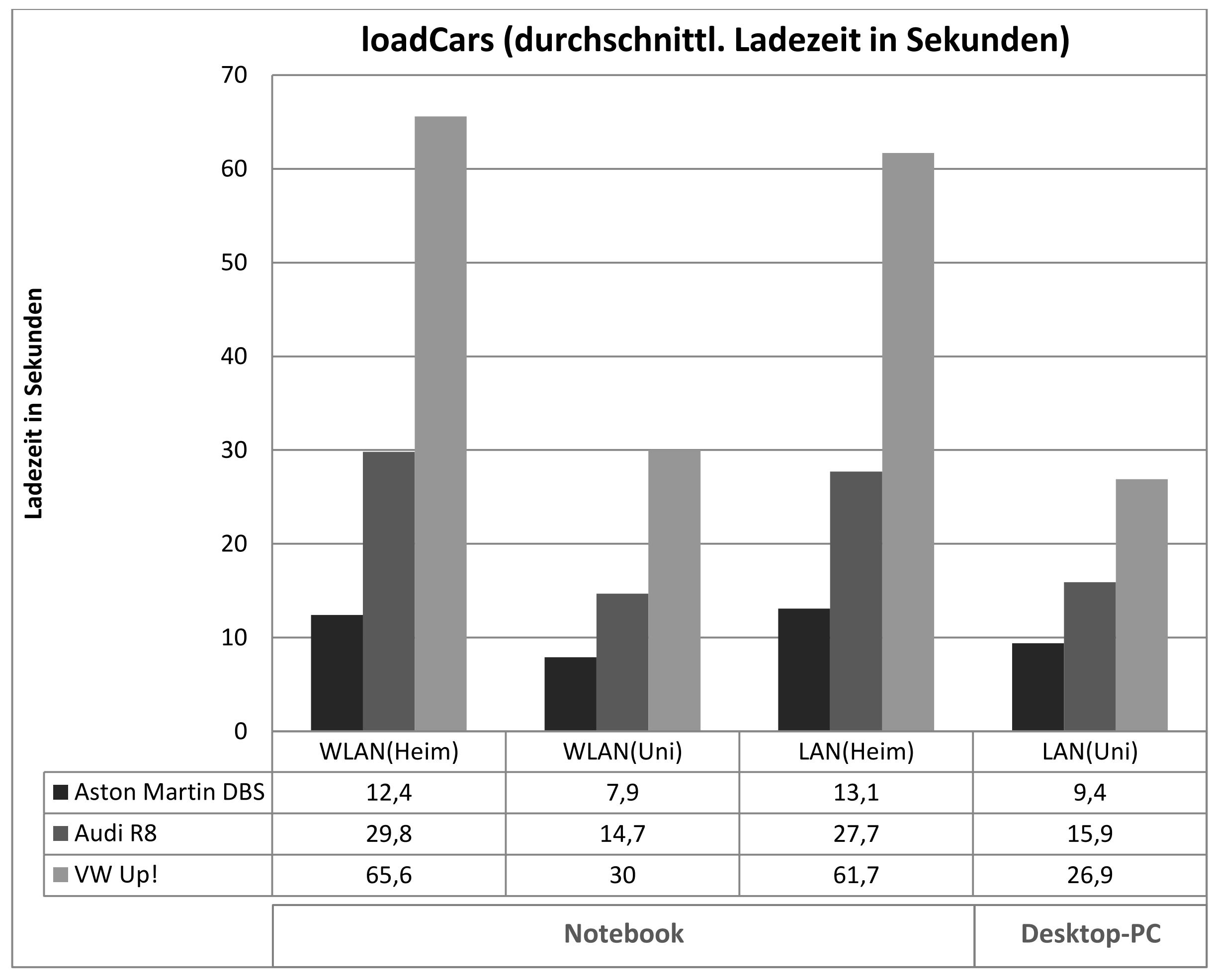


DIAGRAMM 2: LADEZEITEN DER LOADCARS-METHODE

14. Ergebnisse

Neben der vorangegangenen statistischen Analyse der Ergebnisse, folgen in diesem Kapitel Screenshots des WebGL-Konfigurators, welche die Ausgaben der implementierten Funktionalitäten zeigen. Dabei wird der Fokus insbesondere auf die Möglichkeiten der Fahrzeugauswahl und –lackierung gelegt, sowie ein Ausblick auf die konzeptionellen Vorstellungen des Felgenwechsels gegeben, da dieser wie auf Seite 68 beschrieben, noch nicht programmiert wurde.

Fahrzeugauswahl

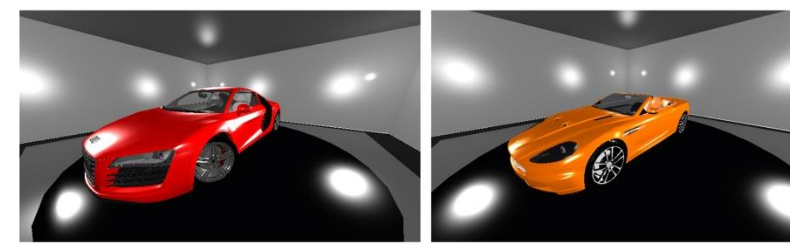


Abbildung 49: Fahrzeug-Auswahl

Fahrzeug-Lackierungen

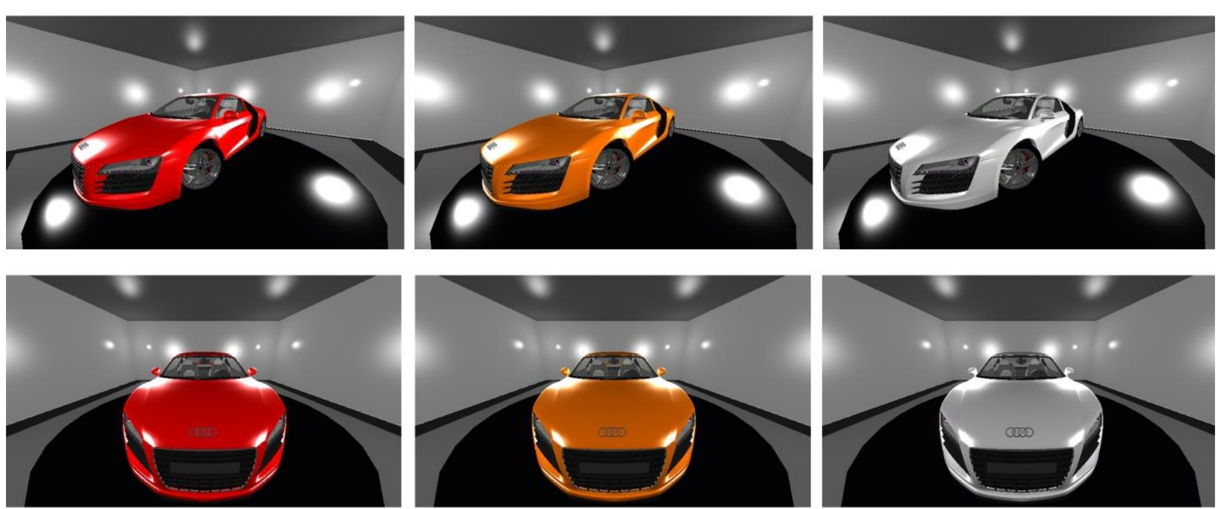


Abbildung 50: Audi R8 Lackierungen

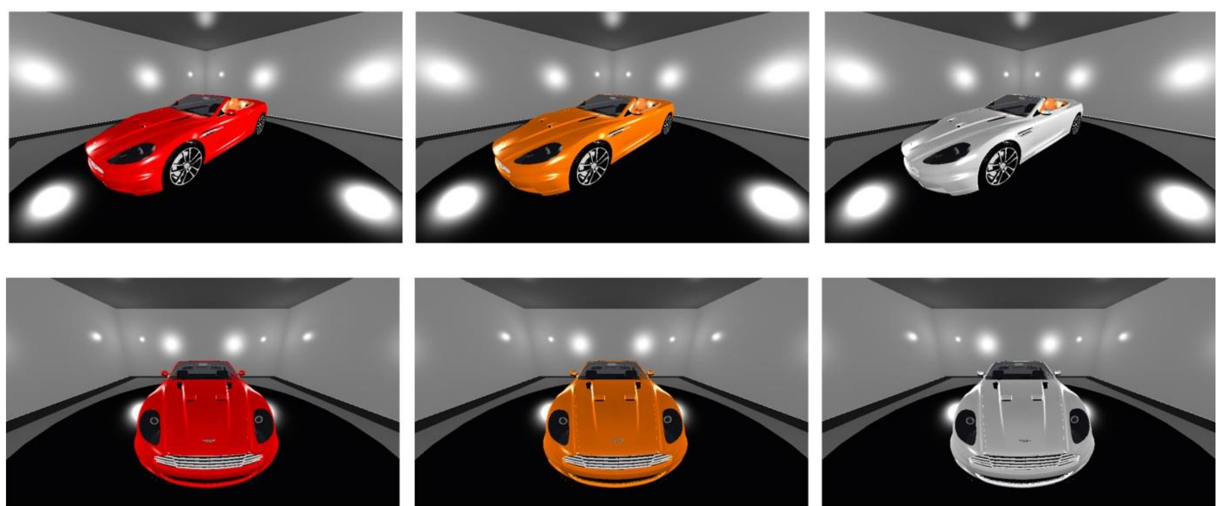


Abbildung 51: Aston Martin DBS Lackierungen

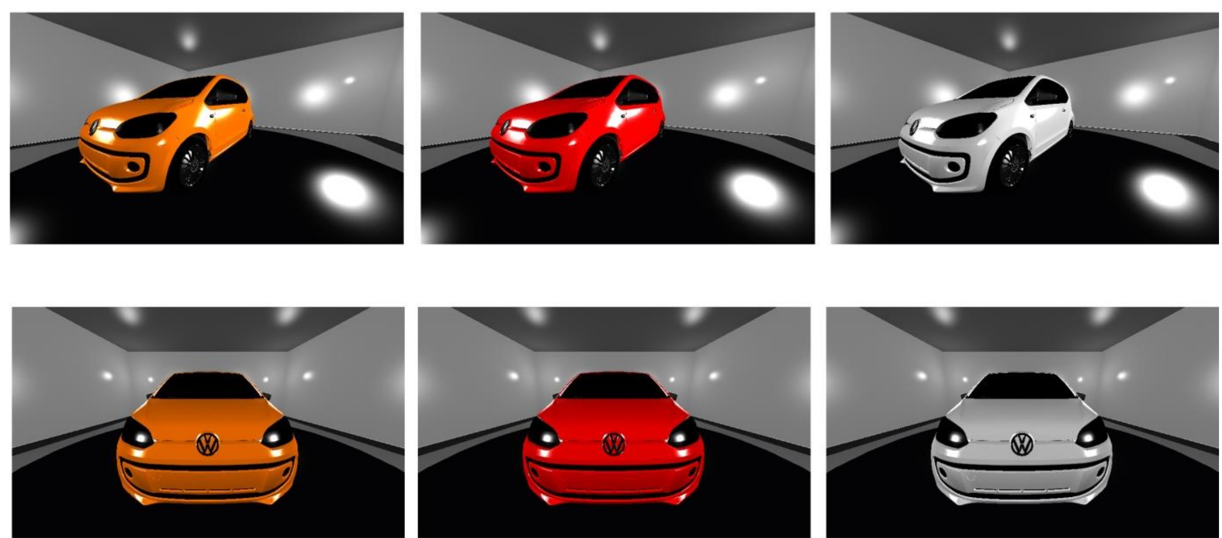


Abbildung 52: VW Up! Lackierungen

Felgenwechsel

Abbildung 53: Audi R8 Felgenwechsel

ABBILDUNG 54: ASTON MARTIN DBS FELGENWECHSEL

ABBILDUNG 55: VW UP! FELGENWECHSEL

15. VERGLEICH

15.1. WERKZEUGVERGLEICH - FLASH VS. UNITY VS. WEBGL

Nachdem die unterschiedlichen Werkzeuge bereits in Kapitel 8 erläutert wurden, wird an dieser Stelle als Ergänzung ein allgemein umfassender Vergleich der Technologien aufgeführt. Ein Kriterium dieses Vergleichs behandelt die PlugIns. Flash ist nicht nativ im Browser verfügbar und benötigt daher einen Zusatz, der es ermöglicht Flash-Inhalte darzustellen. Ebenso wie Flash ist auch Unity auf ein eigens konzipiertes PlugIn angewiesen, den Unity Web Player, um eigene Inhalte im Browser darzustellen. Die Tatsache, dass potentiell sicherheitsgefährdende PlugIns im Browser integriert werden, schreckt jedoch die Nutzer meist nicht ab derartige Technologien zu verwenden[Web 42] Hinzu kommt, dass Flash im Hintergrund Cookies anlegt (sog. Local Shared Objects), die benutzerbezogene Daten speichern und beim Wiederaufruf von bestimmten Websites verwendet werden. Dieses Risiko wird von dem Großteil der Internetnutzer allerdings als sehr gering eingestuft [Web 42] und es sonst nicht möglich wäre, dass Flash eines der verbreitetsten multimedialen Plattformen im Web ist.

Anders als bei WebGL welches frei von PlugIns, nativ in den meisten Browsern implementiert ist, wird Flash auch im Internet Explorer unterstützt, der auch nach neusten Analysen der am meisten genutzte Browser weltweit ist. Hier hat WebGL noch nicht Einzug gehalten, da

Microsoft ein zu hohes Sicherheitsrisiko beim Thema „Hardwarebeschleunigung“ sieht [Web 22]. Bei der Kompatibilität mit mobilen Endgeräten wird WebGL über kurz oder lang in den Vordergrund rücken, da Flash sämtliches Interesse an einem mobilen Engagement eingestellt hat. Gerade in der heutigen Zeit in der viele Nutzer über Smartphones und Tablets den Zugang ins Internet suchen kann dies ein entscheidender Vorteil für WebGL sein, welches beispielsweise in Firefox Mobile oder Opera Mobile bereits ohne Komplikationen funktioniert.

Um beim allgemeinen Werkzeugvergleich zu bleiben, wird als nächstes die Programmierung betrachtet. Flash wird mit der Skriptsprache ActionScript programmiert und behilft sich mit den Funktionen der Stage3D API. Im Gegenzug nutzt WebGL JavaScript. Im Hinblick auf die Unterstützung von Hardwarebeschleunigung können sowohl Unity und Flash als auch WebGL punkten, da alle drei diese Funktionalität ermöglichen. Speziell scheint Adobe Flash verstärkt, nach Einstellung des mobilen Sektors, an der Hardwarebeschleunigung zu arbeiten, da diese Funktionalität noch relativ neu ist und Adobe somit das Interesse auf die Innovation des Flash Players zu legen scheint.

Die relativ schwachen Möglichkeiten des Debuggings fielen einem erst nach eigener Programmierung in WebGL auf. Seitens der Entwicklungsumgebung konnte dies nicht eingerichtet werden. Erst unter zu Hilfenahme eines Debug-Skriptes, welches zur Laufzeit sämtliche OpenGL Befehle in die Konsole des Browsers ausgibt, konnte eine einigermaßen brauchbare Debug-Funktionalität erreicht werden. Dies geschieht allerdings stark auf Kosten der Systemressourcen, da die Ausgaben den Arbeitsspeicher rasant auslasten.

Zu guter Letzt ist anzumerken, dass mit der Arbeit in WebGL ein nicht unbeträchtlicher Aufwand an Einarbeitung in Kauf genommen werden muss, da ein hohes Maß an Wissen über OpenGL notwendig ist. Viele funktionsorientierte Bibliotheken vereinfachen zwar den Einstieg in WebGL und auch die stetig wachsende Community um den freien Standard trägt dazu bei, doch bieten alleine schon Programme wie Adobe Flash und die langjährige Etablierung im Web einen weitaus einfacheren und intuitiveren Zugang zur Materie.

15.2. Optik und Performance von Konfiguratoren

Um die WebGL Inhalte dieser Arbeit abzurunden werden nun einige Bereiche thematisiert, die den aktuellen Stand der Technik, Ästhetik und Umsetzung von Automobilkonfiguratoren mit Flash und WebGL wiederspiegeln. Während der mediale Output (Bilder, Videos) von flash-basierten Konfiguratoren in seiner Bildqualität auf Grund der festen Auflösung beschränkt ist, kann WebGL hiermit trumpfen. Die virtuelle Szene, in der das Fahrzeug zu sehen ist, kann beliebig groß skaliert werden, ohne dabei Qualitätsverluste zu erzeugen, da jedes Bild neu berechnet wird.

Abbildung 56: JPEG vs. WebGL Qualitätsvergleich

Bei Bildern, die meist wegen des Wunsches verkürzter Ladezeiten im komprimierten JPEG Format vorliegen, treten an dieser Stelle schnell Artefakte (Pixelkanten und Schlieren) auf, die auf das digitale Zoomen zurückzuführen sind. Die zuvor aufgeführte Abbildung 56 spiegelt dieses Phänomen sichtbar wieder. Darüber hinaus bleibt die Optik zu jedem Zeitpunkt auf eine gewisse Fotoästhetik beschränkt, da jegliche räumliche Integration und Interaktion fehlt. Flash erweiterte diesen Funktionsumfang nicht, da nach eigenen Ansichten ein erhöhtes Sicherheitsrisiko von einer größeren Tastenbelegung ausgeht, insbesondere bei der Schaltung in den Fullscreen-Modus, da so die dahinterliegende Ebene verdeckt wird. Im

Gegensatz dazu bietet WebGL die gesamte Tastenbandbreite der Tastatur inklusive Maus zur Steuerung der Anwendung an und schafft so viel komplexere Möglichkeiten der Interaktion – obgleich bei dem in dieser Bachelorarbeit implementierten Automobilkonfigurator lediglich die Maustasten zum Einsatz kommen. Der Ort des Renderings ist bei flash-basierten Anwendungen stets serverseitig, da Bilder und Videos offline vorgerendert dort gespeichert liegen und softwarebasiert (auf der CPU) beim Client dargestellt werden. Im Gegensatz dazu findet das Rendering mit WebGL auf Clientseite statt und geschieht online und hardwarebasiert über die Grafikkarte des Nutzers. Dabei ist dieses Vorgehen weitaus ressourcen-orientierter, da die GPU (*graphics processing unit*) weitaus mehr Rechenkapazität bietet als die CPU (*central processing unit*).

16. Ausblick

Der in dieser Arbeit erstellte WebGL-Konfigurator bietet ein breites Spektrum an Erweiterungspotential in Bezug auf Umfang der Anwendung bzw. spezielle Technologien. So könnte man sich beispielsweise die Anbindung an eine Datenbank vorstellen, die sämtliche Fahrzeug-Datenpakete in binärer Form vorliegen hat, um auf diesem Wege modularisierter als bisher vorzugehen. Des Weiteren ist der Umfang des Angebots an Fahrzeugen, Felgen und Lackierungen beliebig nach obenhin skalierbar, sodass ein WebGL-Konfigurator auch die große Produktpalette namenhafter Automobilhersteller bewältigen könnte. Eine weitere Idee für zukünftige Arbeiten ist die Optimierung des Konfigurators für mobile Endgeräte wie Smartphones und Tablets. Wie auf der folgenden Abbildung 57 zu sehen ist, lässt sich die Anwendung auch auf einem Smartphone starten und Fahrzeuge mit geringem Datenvolumen nach längerer Wartezeit laden, lackieren und in unterschiedlichen Perspektiven betrachten.

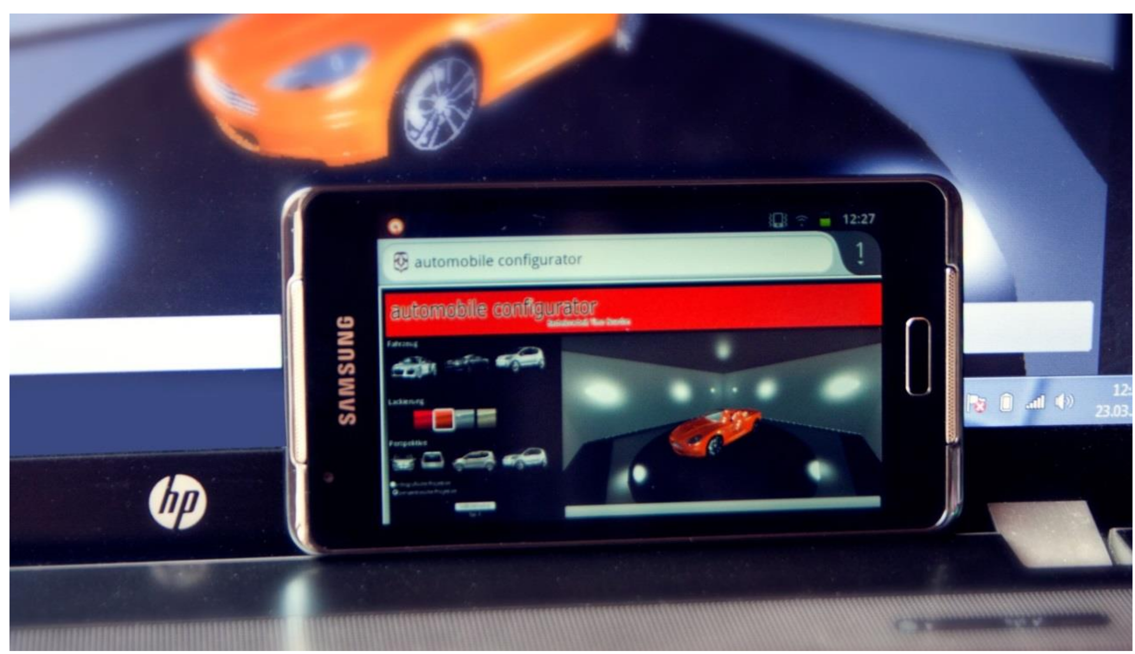


ABBILDUNG 57: MOBILER AUTOMOBIL-KONFIGURATOR

Gerade in Bezug auf komplexe und leistungsintensive Rendering-Prozesse sowie der Qualitätsoptimierung für eher schwache Endgeräte, könnte die Technologie des Remote-Renderings eine wichtige Rolle spielen. Rechenlastige Vorgänge werden nicht auf der clientseitigen Hardware gerendert, sondern auf leistungsstarken Server durchgeführt und dem Nutzer als interaktives Video angeboten. Dies geschieht damit unabhängig von den Leistungsmerkmalen der Hardware des Nutzers und kann sowohl optisch qualitativ als auch performant auf jedem Endgerät gleichermaßen dargestellt werden. Im Rahmen dieses Aspekts würde auch eine Vielzahl an verschiedenen Testreihen zur Optimierung der Darstellung auf unterschiedlichen Endgeräten wegfallen, da die Ausgabe wie eben beschrieben überall identisch ist.

17. FAZIT

Mit dem bis zum aktuellen Stadium entwickelten WebGL-Automobilkonfigurator, konnte der WebGL-Standard erfolgreich getestet und entdeckt werden. Sämtliche Funktionen, die ebenso bei Flash- und Unity-basierten Konfiguratoren implementiert sind, wurden auch hier erfolgreich umgesetzt. Mit dem Wunsch nach einer performanteren Implementation für mobile Endgeräte und der Optimierungsmöglichkeit durch Remote-Rendering, ist die persönliche Motivation zur Fortführung der WebGL Anwendung ungebrochen.

Literaturverzeichnis

[Anyu12] Andreas Anyuru, *Professional WebGL Programming: Developing 3D Graphics for the Web*, John Wiley & Sons, 2012

[CG12] Henning Wenke, *Computergrafik Skript*, Computergrafik-Vorlesung der Universität Osnabrück, Sommersemester 2012

[CGuB1] Alfred Nischwitz, Max Fischer, Peter Haberäcker, Gudrun Socher, *Computergrafik und Bildverarbeitung: Band I*, Vieweg + Teubner Verlag, 2011

[DMA06] Jürgen Hagler, *Digital Media for Artists – 3D-Grafik - Modul Schattierungen*, Archiv 2002-2011 Kunstuniversität Linz, 2006

[GDM11] Mark DeLoura*, Engine Game Survey, Game Developers Magazine*, Mai 2011

[M&C11] Maximilian Eibl, *Mensch & Computer 2011*, Oldenbourg Wissenschaftsverlag GmbH, 2011

[PT3D] Jacynthe Pouliot, *Progress and New Trends in 3D Geoinformation Science*, Springer-Verlag Berlin Heidelberg, 2013

[TMAP] Prof. Faxin Yu, *Three-Dimensional Model Analysis and Processing*, Springer Heidelberg, 2010

[VoAJ07] Sebastian Roth, *Vorteile der AJAX-Technologie für interaktive Internet-Anwendungen,* Hochschule Wismar, 2007

[Web2.0] Prof. Dr. O. Günther, *Informatik im Fokus – Web 2.0*, Springer-Verlag Berlin Heidelberg, 2008

[WinfEE09] Kenneth C. Laudon, Jane Price Laudon, Detlef Schoder, *Wirtschaftsinformatik: Eine Einführung*, Addison-Wesley Verlag; Auflage: 2., aktualisierte Auflage, 20. November 2009

Quelle: [Web 1] Internet, http://www.welt.de/img/motor/crop100110086/3200716838-ci3x2l-w580-aoriginal-h386-l0/VW-Konzept1-DW-Sonstiges-Wolfsburg.jpg, Stand: 02.02.13

[Web 2] Internet, http://www.spiegel.de/netzwelt/web/it-legenden-wie-tim-berners-lee-das-web-erfand-a-610257.html
Stand: 03.02.13

[Web 3] Internet, http://www.chip.de/bildergalerie/Von-Mosaic-bis-Chrome-Die-Geschichte-der-Web-Browser-Galerie_36291355.html?show=3
Stand: 03.02.13

[Web 4] Internet, http://www.webanalyticsworld.net/2010/05/2005-2010-historical-youtube.html
Stand: 03.02.13

[Web 5] Internet, http://www.heise.de/open/artikel/Web-statt-Desktop-Framework-Umstieg-mit-System-893435.html
Stand: 04.02.13

[Web 6] Internet, http://media2mult.uos.de/pmwiki/fields/cg-II-09/index.php?n=WebGraphics.VRMLAmpX3D
Stand: 04.02.13

[Web 7] Internet, http://www.golem.de/0903/66105.html
Stand: 05.02.13

[Web 8] Internet, http://www.golem.de/0909/69946.html
Stand: 05.02.13

[Web 9] Internet, https://www.khronos.org/assets/uploads/apis/members.png
Stand: 05.02.13

[Web 10] Internet, http://www.apfelmag.com/files/2009/02/flashlogo.jpg,
Stand: 02.02.13

[Web 11] Internet,
http://cc5.volkswagen.de/cc5/images/vehicle.ashx?height=235&rs=ExtraLarge&rt=Front&hash=-468246794&mi=images%2fde-4.0%2fmissing_big_stage_xlarge.gif,
Stand: 02.02.13

[Web 12] Internet,
http://cc5.volkswagen.de/cc5/images/vehicle.ashx?height=235&rs=ExtraLarge&rt=Back&hash=-468246794&mi=images%2fde-4.0%2fmissing_big_stage_xlarge.gif,
Stand: 02.02.13

[Web 13] Internet, http://www.volkswagen.de/de/CC5.html,
Stand: 02.02.13

[Web 14] Internet, http://www.indiegamemag.com/media/Unity_logo_big2-613x287.jpg, Stand: 02.02.13

[Web 15] Internet, https://www.apple.com/pr/library/2007/01/09Apple-Reinvents-the-Phone-with-iPhone.html
Stand: 02.02.13

[Web 16] Internet,
http://www.gamasutra.com/view/news/169846/Mobile_game_developer_survey_leans_heavily_toward_iOS_Unity.php#.URo6LGdfLoa
Stand: 12.02.13

[Web 17] Internet, http://www.porsche.com/germany,
Stand: 02.02.13

[Web 18] Internet, http://caniuse.com/canvas,
Stand: 01.02.13

[Web 19] Internet, http://www.zdnet.de/41554830/webcl-khronos-entwickelt-browser-variante-von-opencl/
Stand: 09.02.13

[Web 20] Internet, https://www.khronos.org/registry/webgl/specs/1.0/#6
Stand: 08.02.13

[Web 21] Internet, http://caniuse.com/webgl
Stand: 09.02.13

[Web 22] Internet, https://blogs.technet.com/b/srd/archive/2011/06/16/webgl-considered-harmful.aspx?Redirected=true
Stand: 09.02.13

[Web 23] Internet, http://www.sentinel-cgi.co.uk/images/layout/panel_info/logo_autodesk-maya.png
Stand: 19.02.13

[Web 24] Internet,
http://usa.autodesk.com/adsk/servlet/index?id=5970886&siteID=123112
Stand: 19.02.13

[Web 25] Internet,
http://usa.autodesk.com/adsk/servlet/item?siteID=123112&id=9730479
Stand: 19.02.13

[Web 26] Internet,
http://www.autodesk.de/adsk/servlet/item?siteID=403786&id=7261851&linkID=411015
Stand: 19.02.13

[Web 27] Internet,
http://programme.deutschedownloads.de/upload/program3053.jpg
Stand: 19.02.13

[Web 28] Internet, http://www.blender.org/blenderorg/blender-foundation
Stand: 19.02.13

[Web 29] Internet, http://www.chip.de/downloads/Blender-32-Bit_12993220.html
Stand: 19.02.13

[Web 30] Internet, http://paulbourke.net/dataformats/obj/
Stand: 25.02.13

[Web 31] Internet, http://www.json.org/json-de.html
Stand: 25.02.13

[Web 32] Internet, http://www.calvin.edu/~jev42/352/proj8/mrdoob-three.js-25dddd2/utils/exporters/convert_obj_three.py
Stand: 28.02.13

[Web 33] Internet, https://ie.microsoft.com/testdrive/Graphics/RequestAnimationFrame/Default.html
Stand: 03.04.13

[Web 34] Internet, http://www.drweb.de/magazin/requestanimationframe-performante-javascript-animationen-ohne-settimeout-und-setinterval-37205/
Stand: 28.02.13

[Web 35] Internet, http://www.sitepoint.com/html5-full-screen-api/
Stand: 01.03.13

[Web 36] Internet, https://www.khronos.org/registry/webgl/sdk/demos/common/webgl-utils.js
Stand: 03.04.13

[Web 37] Internet, https://www.khronos.org/registry/webgl/sdk/debug/webgl-debug.js
Stand: 03.04.13

[Web 38] Internet, http://blog.tojicode.com/
Stand: 03.04.13

[Web 39] Internet, http://drawlogic.com/2008/09/28/flash-10-changes-good-and-bad-mostly-good-full-screen-input-rtmfp-clipboard-local-save-and-load/
Stand: 20.03.13

[Web 40] Internet, https://developer.mozilla.org/enUS/docs/DOM/Using_fullscreen_mode
Stand: 03.04.13

[Web 41] Internet, http://caniuse.com/#search=fullscreen
Stand: 01.03.13

[Web 42] Internet, http://econsultancy.com/uk/blog/11001-majority-of-consumers-ignore-privacy-and-cookie-info-stats
Stand: 20.03.13

ABBILDUNGSVERZEICHNIS

Abbildung 1: VW Coupé Original 5
Abbildung 2: VW Coupé bearbeitet in Photoshop 5
Abbildung 3: Überblick über die Unternehmen der Khronos Group 8
Abbildung 4: Flash Logo 9
Abbildung 5: VW Touareg vorne 9
Abbildung 6: VW Touareg hinten 9
Abbildung 7: VW Autokonfigurator 10
Abbildung 8: Unity Logo 11
Abbildung 9: Porsche Autokonfigurator 12
Abbildung 10: Ausschnitt der WebGL-Pipeline 16
Abbildung 11: gl.POINTS 17
Abbildung 12: gl.LINES 17
Abbildung 13: gl.TRIANGLES 17
Abbildung 14: Orthogonale Projektion 18
Abbildung 15: Perspektivische Projektion 20
Abbildung 16: OpenGL / WebGL Graphics Pipeline 21
Abbildung 17: Clipping Ebene in 3D Modell 21
Abbildung 18: Rasterisierung 22
Abbildung 19: Blending 22
Abbildung 20: Phong Komponenten 23
Abbildung 21: Diffuse Reflexionskomponente 24
Abbildung 22: Spekulare Reflexionskomponente 24
Abbildung 23: Spekulare Reflektionen 25
Abbildung 24: Punktlicht 25
Abbildung 25: Point Light 26
Abbildung 26: Normalen auf Oberfläche 26
Abbildung 27: Interpolierte Normale 27
Abbildung 28: Kamera Frustum 29
Abbildung 29: WebGL Kontext 31
Abbildung 30: Maya Logo 40
Abbildung 31: Blender Logo 40
Abbildung 32: Blender in OBJ 44
Abbildung 33: Blender in MTL 46

Abbildung 34: Client-Server Modell 48
Abbildung 35: Client-Server Netzwerk 49
Abbildung 36: Website Layout 50
Abbildung 37: Interaktive 50
Abbildung 38: Der Showroom 51
Abbildung 39: Blender Oberfläche 52
Abbildung 40: OBJ Export Parameter 52
Abbildung 41: Parsing OBJ nach JSON 54
Abbildung 42: Ladebalken 59
Abbildung 43: Vergleich der Projektionen 63
Abbildung 44: Lichter der Szene 64
Abbildung 45: Initiale Showroom-Szene 65
Abbildung 46: loadCars() Methode 67
Abbildung 47: part-Array via Firebug 69
Abbildung 48: Test Hardware 71
Abbildung 49: Fahrzeug-Auswahl 74
Abbildung 50: Audi R8 Lackierungen 74
Abbildung 51: Aston Martin DBS Lackierungen 75
Abbildung 52: VW Up! Lackierungen 75
Abbildung 53: Audi R8 Felgenwechsel 75
Abbildung 54: Aston Martin DBS Felgenwechsel 76
Abbildung 55: VW Up! Felgenwechsel 76
Abbildung 56: JPEG vs. WebGL Qualitätsvergleich 78
Abbildung 57: Mobiler Automobil-Konfigurator 80

FAHRZEUGDATEN

Audi R8: http://www.blendswap.com/blends/view/3766
Aston Martin DBS: http://www.blendswap.com/blends/view/44901

TABELLENVERZEICHNIS

Tabelle 1: Konfiguratoren Führender Automobilhersteller 13
Tabelle 2: Code-Vergleich WebGL - OpenGL 37
Tabelle 3: Browserkompatibilität für WebGL 39
Tabelle 4: OBJ Syntax 43
Tabelle 5: MTL Syntax 45
Tabelle 6: JSON Konstrukte 47
Tabelle 7: Datenumfang 51
Tabelle 8: Kompatibilität der Fullscreen API 60

ERKLÄRUNG ZUR SELBSTSTÄNDIGEN ABFASSUNG DER BACHELORARBEIT

Ich versichere, dass ich die eingereichte Bachelorarbeit / die entsprechend gekennzeichneten Teile der eingereichten Bachelorarbeit selbständig und ohne unerlaubte Hilfe verfasst habe. Anderer als der von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlichen oder sinngemäß den Schriften andrer Autoren entnommenen Stellen habe ich kenntlich gemacht

....	....
Ort, Datum	Unterschrift