FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

SPEX: Visualisieren von 3D-Brillenmodellen im Browser mittels WebGL

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Luca Maestri
Matrikelnummer 0926939

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Wien, 30.06.2013

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Luca Maestri
Stöbergasse 17, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Kurzfassung

Diese Arbeit dokumentiert den Online-Teil der Spex Applikation, ein Projekt, welches durch die Kollaboration von JFPartners und der Technischen Universität Wien entstanden ist. Spex soll die Möglichkeit bieten, virtuelle 3-dimensionale Brillenmodelle auf Fotos von Kunden zu rendern. Dadurch können Kunden Brillenmodelle aus Onlinekatalogen anprobieren, ohne dass der Verkäufer alle Brillen auf Lager haben muss. Die Technologie, die zum Rendern der 3-dimensionalen Szene im Browser verwendet wurde, heißt WebGL, ein neuer Standard der Khronos-Gruppe, der dem Browser Zugriff auf die Grafikkarte anbietet. Vorteile des WebGL sind, dass er mittlerweile von den meisten Browsern unterstützt wird und dass die gesamte Berechnung der Szene auf der Grafikkarte des Clients läuft. Dadurch entfällt Last am Server.

Inhaltsverzeichnis

KAPITEL 1

Introduction

1.1 Spex

Die Idee des Projekts mit dem Namen Spex ist es, den Kunden der Optiker die Möglichkeit anzubieten, virtuelle Brillenmodelle anzuziehen. Dadurch können die Kunden Brillen anprobieren, die der Verkäufer nicht auf Lager hat oder die er erst nach dem Kauf bestellen wird. Der Vorgang der Bildaufnahme und Brillenauswahl soll dabei so schnell und einfach wie möglich gestaltet werden. Aus diesem Grund wird die Kinect zur Bildaufnahme verwendet. Die Kinect bietet die Möglichkeit, auch die Tiefe der fotografierten Szene zu berechnen. Dadurch kann man die Position und Drehung des Gesichts der fotografierten Person verfolgen und ihr eine virtuelle Brille aufsetzen. Das Projekt kann auch für Online-Kunden erweitert werden, die dann Brillen aus einem Online-Katalog aussuchen können. Das gesamte Projekt wird deswegen in zwei Teile aufgeteilt: der erste Teil ist der Offline-Viewer, mit dem man das Bild des Kunden aufnehmen kann. Der Offline-Viewer berechnet dabei auch die Position der Brille für jedes Bild und speichert die Resultate in eine Datei. Der zweite Teil des Projekts ist der Online-Viewer, der sich die Information aus dieser Datei zu Nutze macht, um die Visualisierung der 3-dimensionalen Brillenmodelle auch in Browsern zu berechnen. In dieser Arbeit wird der Online-Viewer dokumentiert.

1.2 Verwendete Technologien

Um die Anwendung zu realisieren wurden verschiedene Technologien verwendet. In diesem Kapitel werden diese Technologien und ihre Anforderungen erläutert. Als erstes wird WebGL angesprochen, welches zum Rendern von 3D-Szenen in Browsern verwendet wird. Darauf folgt Three.js, eine Javascript-Bibliothek, die eine vereinfachte Schnittstelle zum WebGL anbietet. Als nächstes wird Zip.js eklärt, welches ebenfalls eine Javascript-Bibliothek ist und zum Entpacken von Zip-Archiven am Browser dient. Als letztes wird IEWebGL erklärt, welches ein Plugin für den Internet Explorer ist und die Verwendung von WebGL im Internet Explorer er-

möglicht. Die Komplexität und Größe des Projekts erforderte die Kapselung des Quellcodes, doch Javascript bietet ürsprünglich nicht diese Möglichkeit an, deswegen wurde Require.js zu Modularisierung des Codes verwendet.

WebGL (http://www.khronos.org/webgl/)

WebGL ist eine Schnittstelle für die 3D-Grafik Programmierung in Browsern. Die Spezifikation für die Web Grafics Library 1.0 (WebGL 1.0) wurde 2011 von Khronos Group veröffentlicht und wurde daraufhin zum Standard für Firefox, Chrome, Opera und Safari. Microsoft hat die offizielle Unterstützung für WebGL abgelehnt, dafür gibt es mehrere öffentliche Projekte, die in Form von Plugins die Verwendung von WebGL im Internet Explorer ermöglichen. Das IE-WebGL Plugin ist ein Beispiel dafür.

Three.js (http://threejs.org/)

Three.js ist ein öffentliches Projekt und eine Javascript-Bibliothek, die einen vereinfachten Zugang für Programmierer zum WebGL bietet. Three.js übernimmt die korrekte Initialisierung des WebGL auf den jeweiligen Browser angepasst und stellt dem Programmierer schon einige Standard-Shader zur Verfügung. Da Three.js noch in Entwicklung ist, wurde die Revision 'dev54' von Three.js als Standard für unser Projekt gewählt. In der gewählten Revision gab es einige Bugs, die man selbst entfernen musste.

Zip.js (http://gildas-lormeau.github.io/zip.js/)

Zip.js ist eine Javascript-Bibliothek, die zum Entpacken von Zip-Archiven am Browser dient. In diesem Fall wurden die 3D-Modelle der Brillen und die dazugehörige Material-Information (*.mat) im Object Datei Format (*.obj) abgespeichert. Um das Problem der Größe dieser Dateien und den damit verbundenen Ladezeiten zu umgehen, wurden die Brillenmodelle in Zip-Archiven am Server komprimiert. Durch Zip.js können die Zip-Archive vom Client geladen und dort dann entpackt werden. Zip.js verwendet Datenformate aus WebGL und ist somit davon abhängig.

IEWebGL (http://iewebgl.com/)

Das Internet Explorer WebGL Plugin (IEWebGL) ermöglicht die Verwendung vom WebGL Standard im Internet Explorer. Das Plugin wird auf Windows Vista/7/8 32-Bit und 64-Bit mit dem Internet Explorer ab Version 8 unterstützt.

Require.js (http://www.requirejs.org/)

Require.js ist ein Javascript Modul-Loader. Der gesamte Online-Viewer wurde in eine Modulstruktur aufgeteilt, um die Kapselung, Codeübersicht und Erweiterbarkeit des Projekts beizubehalten. Ein Problem, dass dabei entsteht, sind die Abhängigkeiten der einzelnen Module untereinander und die Reihenfolge, in denen sie vom Client geladen werden. Require.js löst diese Probleme und bietet dabei gleichzeitig die Möglichkeit an, die Größe des Quellcodes und damit die Ladezeiten weiter zu minimieren.

KAPITEL 2

Typographic Design

2.1 Ordnerstruktur

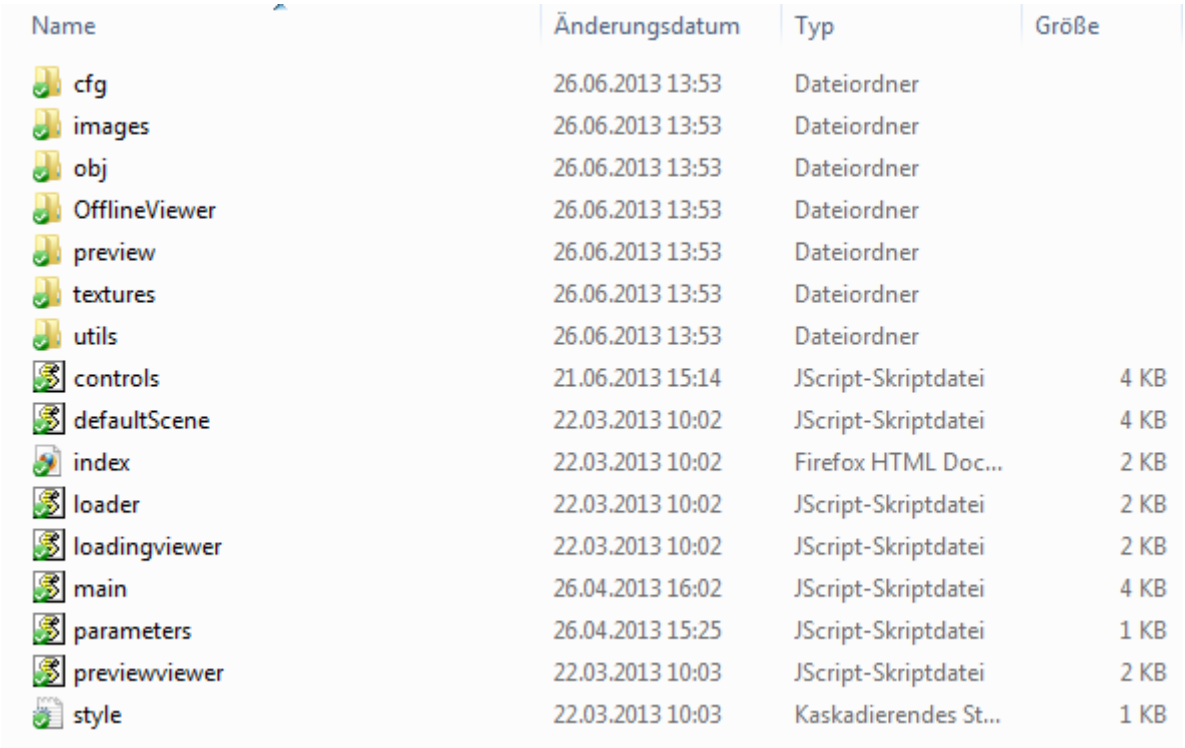
In diesem Abschnitt wird die Ordnerstruktur des Online Viewers erklärt. Die Wurzel des Projekts liegt im Ordner, in dem sich die 'index.html'-Datei befindet. Hier wird nur die Bedeutung der Ordner eingegangen. Ihr Inhalt wird nur kurz angesprochen, für eine genauere Beschreibung der einzelnen Dateien wird auf die Code-Dokumentation verwiesen. In Abbildung 2.1 kann man ein Bild der Ordnerstruktur des Online-Viewers sehen. Die folgenden Abschnitte beschreiben die Ordner in der Reihenfolge, die man auch im genannten Bild sehen kann.

cfg

In diesem Ordner befindet sich die Datei, die als Kommunikationsschnittstelle zwischen Online- und Offline-Viewer dient. Der Online-Viewer liest die vom Offline Viewer berechnete Position und Rotation der Brillenmodelle aus dieser Datei. Für jedes mit dem Offline-Viewer aufgenommene Bild wird die Position und Rotation der Brillenmodelle so berechnet, dass sie dem fotografierten Kunden wie normale Brillen auf der Nase sitzen.

images

Im Images-Ordner befanden sich ursprünglich die aufgenommenen Farbbilder des Kunden und weitere Bilder für die HTML-Struktur des Online-Viewers. In der aktuellen Version des Online-Viewers befinden sich die Farbbilder des Kunden im Offline-Viewer. So wird es ermöglicht, den gesamten Arbeitsvorgang von der Entstehung des Bildes bis hin zur Visualisierung im Browser schneller darzustellen, da der Online-Viewer direkt in den Ordnern des Offline-Viewers nach den Farbbildern sucht.



Name	Änderungsdatum	Typ	Größe
cfg	26.06.2013 13:53	Dateiordner	
images	26.06.2013 13:53	Dateiordner	
obj	26.06.2013 13:53	Dateiordner	
OfflineViewer	26.06.2013 13:53	Dateiordner	
preview	26.06.2013 13:53	Dateiordner	
textures	26.06.2013 13:53	Dateiordner	
utils	26.06.2013 13:53	Dateiordner	
controls	21.06.2013 15:14	JScript-Skriptdatei	4 KB
defaultScene	22.03.2013 10:02	JScript-Skriptdatei	4 KB
index	22.03.2013 10:02	Firefox HTML Doc...	2 KB
loader	22.03.2013 10:02	JScript-Skriptdatei	2 KB
loadingviewer	22.03.2013 10:02	JScript-Skriptdatei	2 KB
main	26.04.2013 16:02	JScript-Skriptdatei	4 KB
parameters	26.04.2013 15:25	JScript-Skriptdatei	1 KB
previewviewer	22.03.2013 10:03	JScript-Skriptdatei	2 KB
style	22.03.2013 10:03	Kaskadierendes St...	1 KB

Abbildung 2.1: Screenshot der beschriebenen Ordnerstruktur.

obj

Im Obj-Ordner befinden sich die 3-dimensionalen Brillenmodelle im 'obj'-Format abgespeichert. Dieses Format wurde gewählt, weil es leicht lesbar und übersichtlich ist, was das Debuggen vereinfacht hat. Ein Problem, das im Laufe des Projekts aufgetaucht ist, war die Größe dieser Brillenmodelle und die damit verbundenen Ladezeiten am Client. Um diesem Problem entgegenzuwirken wurden die Modelle ('obj'-Format) samt ihren Materialeigenschaften ('mat'-Format) als Zip-Archiv komprimiert. Diese Archive werden vom Server versendet und dann am Client über Zip.js entpackt und an die Grafikkarte weitergeschickt.

OfflineViewer

Der Offline-Viewer wurde in diesem Ordner abgelegt, weil man so mit dem Online-Viewer alle Dateien direkt aus dem Offline-Viewer auslesen kann und sich während einer eventuellen Vorstellung des Projekts unnötiges Kopieren der Dateien von einem Ordner zum anderen erspart. Der Offline-Viewer muss sich dabei im Wurzelordner des Online-Viewers befinden, weil der Online-Viewer nicht im Browser des Clients läuft und deswegen seine eigene Ordnerstruktur nicht verlassen darf. Dieser Ansatz vereinfacht zwar die Präsentation des Projekts, bringt aber nicht nur Vorteile mit sich. Gleichzeitiger Betrieb beider Viewer kann zu Fehlern führen, da beide gleichzeitig auf Ressourcen zugreifen. Es wird empfohlen, beim Fortsetzen des Projekts

den Offline-Viewer komplett vom Online-Viewer zu trennen, um eventuelle Zugriffsprobleme zu vermeiden.

preview

In diesem Ordner befinden sich die Vorschaubilder der Brillenmodelle. Diese werden im Online-Viewer für den interaktiven Brillenkatalog verwendet. Der Kunde kann die Vorschaubilder anklicken, um die entsprechenden Brillen anzuprobieren.

texture

Im Texture-Ordner befindet sich eine transparente Textur, die verwendet werden kann um Objekte durchsichtig werden zu lassen. In der Szene befinden sich nämlich nicht nur die 3-dimensionalen Brillenmodelle, sondern auch ein Kopfmodell. Auf diesem Kopf sitzen eigentlich die Brillenmodelle, doch da der Kopf durchsichtig texturiert ist sieht man ihn nur teilweise.

utils

In diesem Ordner befinden sich alle Bibliotheken, die verwendet wurden.

2.2 Anwendungsfaelle

Man kann das System zum Präsentieren auf einem Computer lokal laufen lassen. Als erstes verbindet man die Kinect mit dem Computer. Nachdem der Computer die Kinect erfolgreich erkannt hat, kann man den Offline-Viewer starten und ein Bild des Kunden damit aufnehmen. Der Offline Viewer berechnet die Position der Brillen in Echtzeit, sodass man das Ergebnis direkt sehen kann. Wird keine Brille am Offline-Viewer angezeigt, muss der Kunde den Kopf bewegen, um dem Offline-Viewer die Möglichkeit zu geben, das Gesicht zu erkennen. Sobald man ein Bild aufgenommen hat entstehen im Ordner des Offline-Viewers zwei neue Dateien. In einer Datei wird das Farbbild des Kunden abgespeichert, in der anderen die Tiefeninformation der Szene. Die zwei Dateien besitzen dann jeweils die entsprechenden Endungen 'color.jpg' für das Farbbild und 'depth.jpg' für die Tiefeninformation der Szene. Weiters entsteht in der faces.cfg Datei ein neuer Eintrag für jedes aufgenommene Bild. In dieser Datei werden die Position und Rotation der Brille abgespeichert. Für den Online-Viewer sind nur die Farbbilder und die faces.cfg Datei relevant. Da die Brillenposition schon im Offline-Viewer berechnet wird, muss der Online-Viewer nur die faces.cfg Datei auslesen und die darin enthaltene Information anwenden. Dabei gibt es zwei verschiedene Möglichkeiten.

Anwendungsfall 1

Dieser Anwendungsfall eignet sich für Präsentationen am besten, da Online- und Offline-Viewer beide lokal laufen können. Hier wird der Ordner des Offline-Viewers direkt in den Wurzelordner des Online-Viewers kopiert und laufen gelassen. Die Pfade des Online-Viewers werden weiters so angepasst, dass er nach aufgenommen Bildern und der faces.cfg Datei im Ordner des

Offline-Viewers sucht. Diese Vorgehensweise bietet den Vorteil, dass man sich das Kopieren der notwendigen Dateien vom Offline-Viewer in die entsprechenden Ordner des Online-Viewers erspart. Man kann den Offline-Viewer starten und die Bilder aufnehmen und diese dann direkt im Online-Viewer sehen, indem man den Offline-Viewer schließt und den Online-Viewer startet. Man empfiehlt für die lokale Verwendung des Online-Viewers den Firefox-Browser der als einziger standarmäßig den Zugriff von Webandwendungen auf lokale Dateien zulässt. Da der Online-Viewer in diesem Fall nicht von einem Server geladen wird, sondern lokal läuft, ist es unerlässlich, dass der Browser der Anwendung Zugriff auf lokale Dateien gewährt.

Abbildung 2.2: Abbildung beschreibt den 1. Anwedungsfall.

Anwendungsfall 2

In diesem Anwendungsfall läuft der Offline-Viewer auf einem Computer, während der Online-Viewer auf einem Server abgelegt wurde. Das bietet den Vorteil, dass der Online-Viewer über alle Computer, die einen kompatiblen Browser installiert haben, online erreichbar ist. Da jetzt Offline- und Online-Viewer getrennt sind, muss man die für den Online-Viewer notwendigen Dateien vomm Offline-Viewer in die entsprechenden Ordner des Online-Viewers kopieren. Die faces.cfg Datei muss in den cfg-Ordner kopiert werden, während die Farbbilder in den entsprechenden images-Ordner kopiert werden müssen. Weiters müssen die Pfade in der main.js-Datei auf diese Ordner umgestellt werden, falls das noch nicht gemacht wurde. Erst dann kann man die Bilder, die mit dem Offline-Viewer aufgenommen wurden, auch im Online-Viewer sehen.

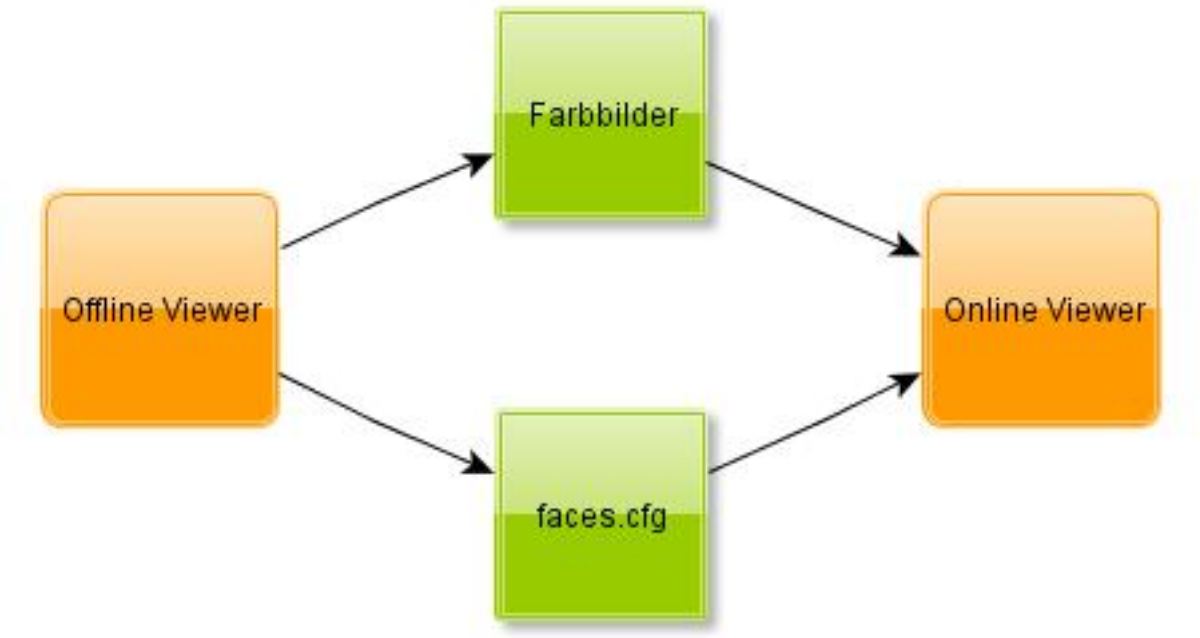


Abbildung 2.3: Abbildung beschreibt den 2. Anwedungsfall.

2.3 Gebrauchsanleitung

Sobald die Seite geladen wurde, kann man durch den Online-Viewer mittels Maus und Tastatur navigieren. Entweder durch Klicken auf die Brillenmodelle, die auf der linken Seite des Fensters zu sehen sind, oder durch Betätigen der Aufwärts- und Abwärts-Pfeiltasten können verschiedene Brillenmodelle anprobiert werden. Ähnlich werden die verschiedene Fotos durch Klicken auf die Pfeile im unteren Bereich des Fensters oder durch Betätigen der linken und rechten Maustaste ausgewählt. Das Menü auf der rechten Seite des Fensters dient manuellen Anpassungen der Brillen, sollte man nicht mit der berechneten Position zufrieden sein. Die Positionierung der Brillen erfolgt im 3-dimensionalen Raum. Dementsprechend kann man die Position der Brillen waagrecht (x-Achse), senkrecht (y-Achse) und in Sichtrichtung (z-Achse) verändern. Zusätzlich kann man die Rotation um die waagrechte (x-Achse), senkrechte (y-Achse) und die Achse in Sichtrichtung (z-Achse) anpassen. Diese Einstellungen können durch Bewegen der entsprechenden Balken mit der Maus oder durch manuelles Eintippen spezifischer Werte im entsprechenden Menüeintrag verändert werden. In Abbildung 2.4 kann man eine Zusammenfassung der Tastaturbefehle und der wichtigsten Module sehen.

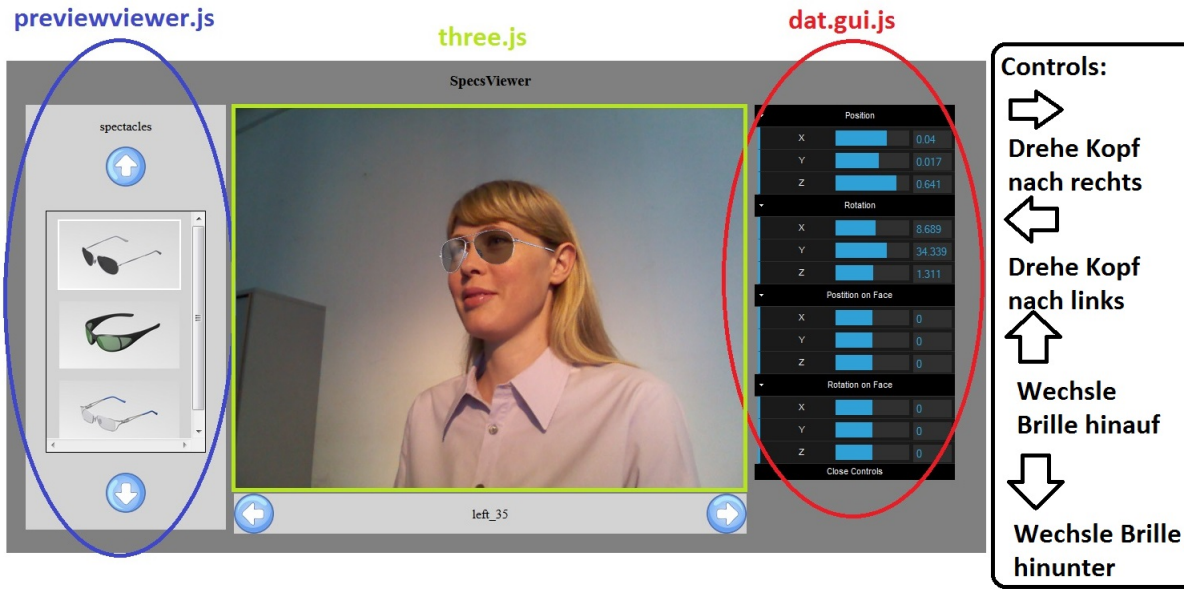


Abbildung 2.4: Kurze Gebrauchsanleitung der Tastaturbefehle und Beschreibung der wichtigsten Module.

2.4 Codedokumentation

In den folgenden Abschnitten wird spezifisch auf den Quellcode des Online-Viewers eingangen. Für jede relevante Datei wird ein entsprechender Eintrag erstellt, in dem ihre Funktion und Bedeutung für den Online-Viewer genauer erläutert werden.

ConfigLoader.js

Dieses Modul dient dem Laden der verschiedenen Szenenparameter in der '*.cfg'-Datei. Dabei wird die '*.cfg'-Datei durch einen HTML-Request am Client geladen, geparst und speichert die extrahierten Werte in die entsprechenden Variablen des Containerobjekts 'retlist'.

deflate.js/inflage.js

Dateien, die die Algorithmen zum Verpacken und Entpacken der zip-Archive und von Zip.js verwendet werden.

ZippedObjLoader.js

Um die Ladezeiten der Brillenobjekte zu verringern, wurden diese samt ihrer Materialinformation in Zip-Archiven verpackt. Deswegen wurde dieses Modul entwickelt, um das Archiv zu laden und zu entpacken. Die entpackten Informationen werden im DOM gespeichert und urls werden zurückgeliefert.

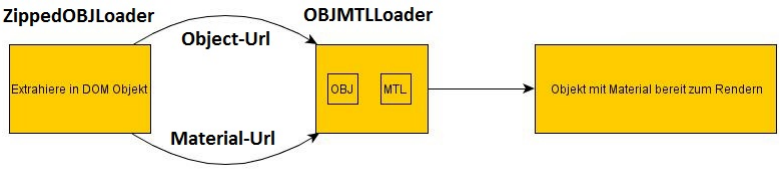


Abbildung 2.5: Ladeprozess vom Zip-Archiv bis zum Speicher des Clients.

Loader.js

Dieses Modul fasst alle notwendigen Loadermodule zusammen, instanziert und initialisiert sie falls notwendig. Dieses Modul wird letztendlich zum Laden der Dateien verwendet und bereitet die geladenen Objekte zur Verwendung auf.

LoadingViewer.js

Dieses Modul verwendet spin.js, um beim Laden von Dateien eine sich drehende Glyphe zu erzeugen. Diese visualisiert den Ladevorgang und verschwindet wieder, sobald die Dateien fertig geladen wurden.

Controls.js

In diesem Modul befindet sich die Instanzierung und Initialisierung von dat.gui.js. Das Modul enthält auch eine Kopie des momentan aktiven Listeneintrags aus dem Parameters Modul (parameters.js). Wenn man die Regler verschiebt, werden nur die Werte der Kopie verändert. Durch Aufrufen der Funktion updateParamsFromControls werden die Veränderungen zurück an das Main Modul (main.js) geschickt (siehe Abbildung 2.8). Die Regler können etweder per Drag and Drop oder durch manuelle Eingabe eines Wertes betätigt werden. In das GUI kann man jeweils drei Regler je Reiter sehen (siehe Abbildung 2.6. Der erste Reiter (Position) beschreibt die Position der Brille im 3-dimensionalen Raum. Eine Veränderung der Werte in diesem Reiter bewirkt also eine Translation der Brillen entlang der gewählten Achse. Dementsprechend bewirkt die Veränderung der Werte im zweiten Reiter (Rotation) eine Drehung der Brille entlang der gewählten Achse. Die Werte werden in diesem Reiter werden in Grad angezeigt. Die letzten Reiter (Position on Face/Rotation on Face) können vom Benutzer verwendert werden wenn er mit der Verdeckung der Brille durch den virtuellen Kopf nicht ganz zufrieden ist. Durch diese Regler kann er eine Verschiebung zwischen Brille und virtuellen Kopf hinzufügen. In anderen Worten wird die Brille wie mit den ersten sechs Reglern verschoben und rotiert, aber diesmal ohne dass sich der Kopf mitbewegt. Am Anfang ist die Verschiebung zwischen Brille und Kopf entlang aller Achse Null. Verändert der Benutzer zum Beispiel den Wert X im Regler 'Position on Face' wird der eingegebene Wert mit der Position des Kopfes addiert und die Brille wird sich um diesen Wert entlang der X-Achse bezüglich des virtuellen Kopfes verschoben haben.

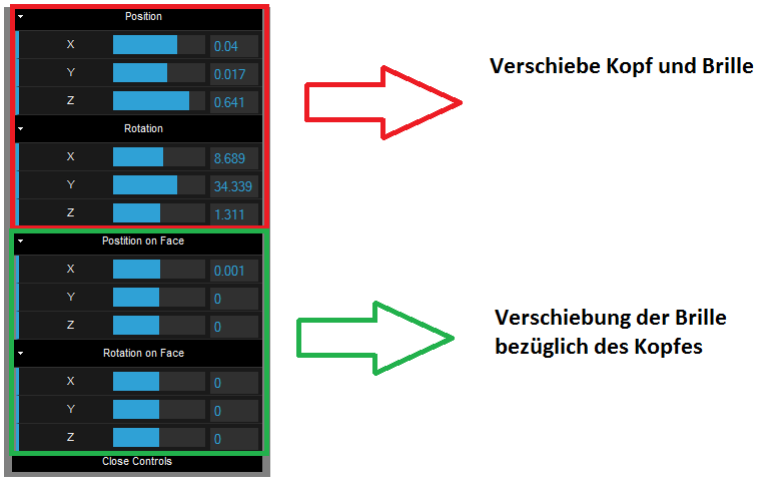


Abbildung 2.6: Zusammenfassung der GUI.

Parameters.js

Dieses Modul wird in main.js instanziert und mit allen Informationen aus der '*.cfg'-Datei vom Offline-Viewer gefüllt. Instanziert man das Modul erhaltet man eine Liste, wobei jeder Eintrag dieser Liste ein Bild, welches mit dem Offline Viewer aufgenommen wurde, repräsentiert (siehe Abbildung 2.7). Indem der Benutzer das Bild ändert (linke/rechte Pfeiltaste oder Maus siehe Abbildung 2.4) wird intern der aktive Listeneintrag verändert, die neuen Parameter werden geladen und dadurch verändert sich die Szene. Der Inhalt des aktiven Listeneintrags wird bei jedem Schleifendurchlauf an das Controls Modul (controls.js) geschickt, um das GUI zu aktualisieren. Führt der Benutzer Veränderungen an den Szenenparametern durch das GUI durch werden diese Veränderungen zurück an das Main Modul (main.js) geschickt und in den aktiven Listeneintrag der Parameterliste eingetragen. Dieser Synchronisierungsprozess ist nötig, weil Javascript den Zugang auf Instanzierungen eines Moduls von fremde Module einschränkt. Abbildung 2.8 veranschaulicht diesen Vorgang genauer.

PreviewViewer.js

Dieses Modul dient der Visualisierung der Brillenliste samt Vorschaubild der Brillenmodelle. Man kann auf die Vorschaubilder der Liste klicken, um eine Brille auszuwählen. Das gerade ausgewählte Brillenmodell besitzt einen helleren Rand.

Main.js

Initialisert und lädt alle notwendigen Module. Die Renderschleife und die Tastaturabfrage befindet sich hier.

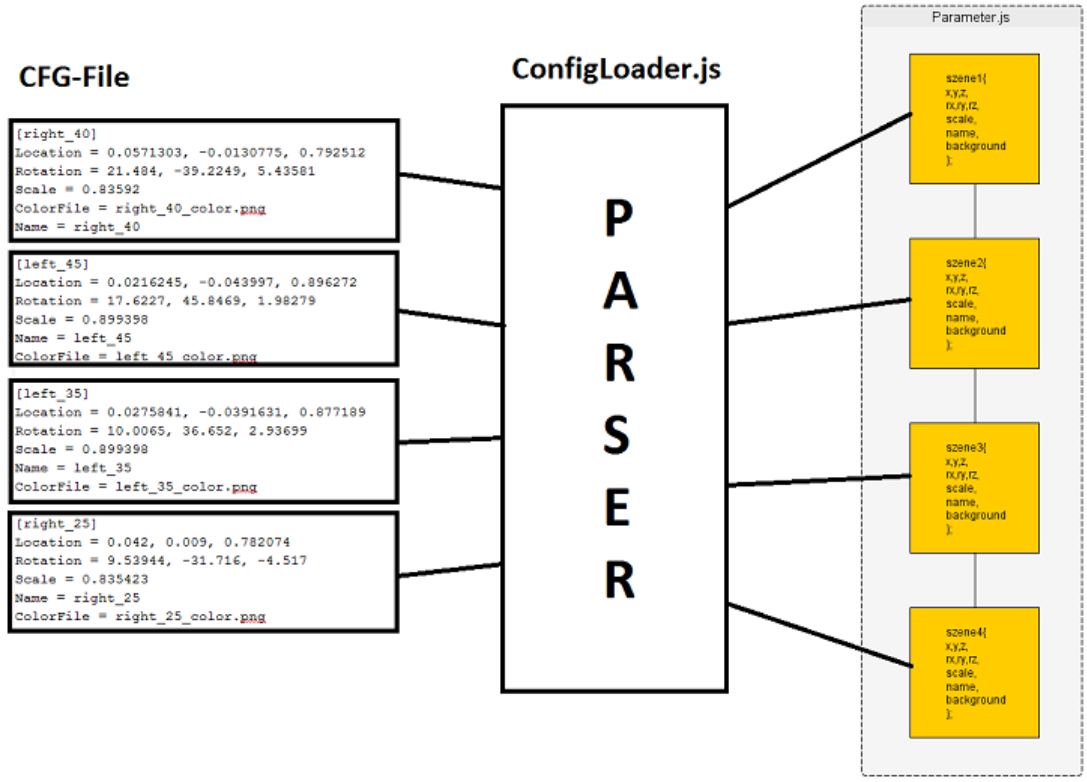


Abbildung 2.7: Von der CFG-Datei zur Parameterliste. Im Bild werden nur die relevanten Einträge von der CFG-Datei dargestellt.

MTLLoader.js

Ein aus dem three.js Framework kopiertes Modul, welches Materialeigenschaften über http-requests lädt und parst.

OBJLoader.js

Ein aus dem three.js Framework kopiertes Modul, welches Meshes im '*.obj'-Format über http-requests lädt und parst.

OBJMTLLoader.js

Ein aus dem three.js Framework kopiertes und angepasstes Modul, welches Meshes und ihre Materialeigenschaften durch OBJLoader.js und MTLLoader.js lädt, parst und zusätzlich zusammenfügt.

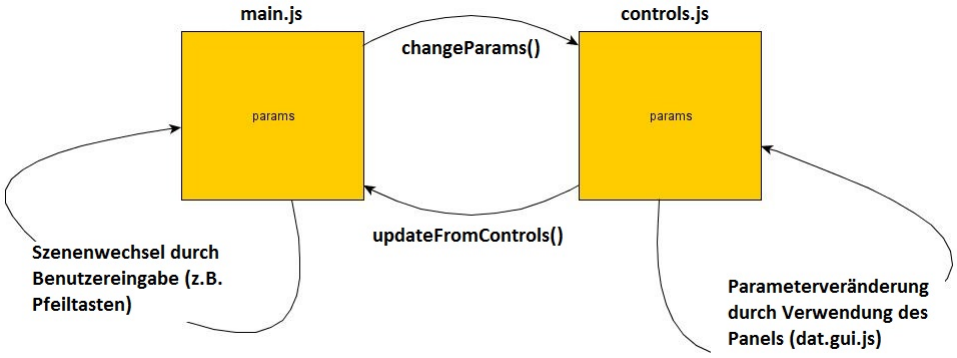


Abbildung 2.8: Synchronisierungsprozess der Parameter zwischen dat.gui.js und der Anwendung.

DefaultScene.js

In DefaultScene.js wird die Szene so initialisiert, dass sie der Szene im Offline Viewer entspricht. Am wichtigsten sind dabei die Kameraeinstellungen, damit die Projektions- und Viewmatrizen der zwei Viewer übereinstimmen und man dadurch die Transformationen aus dem Offline Viewer übernehmen kann.

```
1    //Scene center
2    this.center = new THREE.Vector3(0, 0, 0);
3    var camPos = new THREE.Vector3(0, 0, 1);
4    var lookat = new THREE.Vector3(0, 0, 0);
5    //Camera
6    this.camera = new THREE.PerspectiveCamera(48.6, this.width/this.height,
         0.001, 1000);
7    this.camera.position = camPos;
8    this.camera.lookAt(lookat);
9    this.sceneContainer.add(this.camera);
```

Listing 2.1: Verwendete Kameraeinstellungen für die Szene.

Dabei wird im obigen Codeabschnitt eine perspektivische Kamera mit Öffnungswinkeln von 48.6 Grad initialisiert. Dieser Öffnungswinkel entspricht dem Standardöffnungswinkel der Kinect. Ändert man den Öffnungswinkeln der Kinect manuell im Offline Viewer muss die Kamera im Online Viewer dementsprechend angepasst werden. Die Kamera befindet sich entlang der Z-Achse um eine Einheit verschoben und schaut Richtung Ursprung. Zusätzlich werden der Szene drei weiße Lichtquellen die Richtung Ursprung leuchten hinzugefügt.

```
1  //Lights
2  var directionalLight = new THREE.DirectionalLight( 0xFFFFFF);
3  directionalLight.position.set( 2, 0, 0 ).normalize();
4  directionalLight.target.position.set(0, 0, 0);
5  this.sceneContainer.add( directionalLight );
6
7  var directionalLight = new THREE.DirectionalLight( 0xFFFFFF );
8  directionalLight.position.set( -2, 0, 0 ).normalize();
```

```
9   directionalLight.target.position.set(0, 0, 0);
10  this.sceneContainer.add( directionalLight );
11
12  var directionalLight = new THREE.DirectionalLight( 0xFFFFFF );
13  directionalLight.position.set( 0, 0, 2 ).normalize();
14  directionalLight.target.position.set(0, 0, 0);
15  this.sceneContainer.add( directionalLight );
```

Listing 2.2: Szenenbeleuchtung in DefaultScene.js.

Als letztes wird noch der Renderer initialisiert. Für die Erzeugung des Canvas wurde das WebGLHelper.js Skript verwendet. Dieses Skript übernimmt die Initialisierung des Canvas und ermöglicht gleichzeitig die Verwendung von WebGL im erzeugten Canvas auch im Internet Explorer. Nachdem das Skript den Canvas initialisiert hat kann man das Objekt im DOM-Objekt finden.

```
1   //Renderer
2   var externalCanvas = document.getElementById('renderCanvas');
3   this.renderer = new THREE.WebGLRenderer({
4     'canvas' : externalCanvas,
5     'antialias' : true
6   });
7   this.renderer.setSize( this.width, this.height );
```

Listing 2.3: Initialisierung des Renderers mit dem Canvas-Objekt.

Die Größe/Auflösung des Renderers kann frei gewählt werden solang man das Seitenverhältnis der Kinectkamera beibehaltet. Der Renderer passt sich außerdem der Größe des Canvas an, d.h. man kann die Auflösung unabhängig der benötigten Fenstergröße wählen (mit Augenmerk auf das Seitenverhältnis). Die Kinect nimmt Bilder mit einer Auflösung von 640x480 Pixel auf. Daraus folgt ein Seitenverhältnis von 4:3, deswegen wurde für den Renderer eine Auflösung von 1280x960 Pixel gewählt. Eine Auflösung von 640x480 Pixel für den Renderer würde zu bemerkbar schlechteren Brillenvisualisierungen führen, deswegen wurde beschlossen die Auflösung des Renderers zu erhöhen.

KAPITEL 3

Technische Details

3.1 Initialiserung

Die Initialisierung von WebGL im Browser ist sehr aufwändig und würde außerdem noch die Programmierung der Shader vorsehen. Im OnlineViewer wird für die Initialisierung von WebGL deswegen Three.js verwendet. Three.js bietet die Möglichkeit in wenigen Zeilen eine Szene, eine Kamera und einen Renderer zu initialisieren. Leider unterstützt der Internet Explorer WebGL nicht ohne die Installation eines Plugins. Zusätzlich zum Plugin wird ein Skript zur Initialisierung und Erkennung des Plugins zur Verfügung gestellt. Dieses Skript heißt WebGLHelper.js und wird in der index.html aufgerufen.

```
1
2 <script id="WebGLCanvasCreationScript" type="text/javascript">
3
4 WebGLHelper.CreateGLCanvasInline('renderCanvas');
5
6 </script>
```

Listing 3.1: Verwendung von WebGLHelper.js in der index.html

Nachdem der WebGLHelper.js den Kontext für WebGL im Internet Explorer erstellt hat, kann man WebGL ganz normal wie in der Dokumentation beschrieben initialisieren. Im folgenden Codeabschnitt kann man die Initialisierung einer Szene mithilfe von Three.js sehen.

```
1
2 var scene = new THREE.Scene();
3
4 var camera = new THREE.PerspectiveCamera( 75, window.innerWidth / window.
      innerHeight, 0.1, 1000 );
5
6 var renderer = new THREE.WebGLRenderer();
7
8 renderer.setSize( window.innerWidth, window.innerHeight );
9
```

```
10 document.body.appendChild( renderer.domElement );
```

Listing 3.2: Grundlegende Initialisierungsschritte für Three.js

Im OnlineViewer wird eine leere Szene von Three.js in DefaultScene.js initialisiert. Zusätzlich werden dort auch alle Parameter der Szene so angepasst, dass sie der Szene im OfflineViewer entsprechen. Dadurch kann man in Online- und OfflineViewer mit den gleichen Modellen arbeiten und sie ohne zusätzlichen Aufwand von einer Anwendung in die andere übertragen. Besonders wichtig sind hier

die Kameraparameter. Wenn diese nicht mit den Kameraparametern aus dem OfflineViewer übereinstimmen, werden die Brillen an der falschen Position gerendert. Das kommt daher, dass sich die Ansicht und Perspektive des Beobachters verändern und die Position der Brille für die neue Perspektive neu berechnet werden müsste.

```
1
2  var camPos = new THREE.Vector3(0, 0, 1);
3
4  var lookat = new THREE.Vector3(0, 0, 0);
5
6  this.camera = new THREE.PerspectiveCamera(48.6, this.width/this.height,
       0.001, 1000);
7
8  this.camera.position = camPos;
9
10 this.camera.lookAt(lookat);
11
12 this.sceneContainer.add(this.camera);
```

Listing 3.3: Initialisierung der Kamera in DefaultScene.js

Die Szenenmitte befindet sich hier am Punkt (0,0,0) im dreidimensionalen Raum. Die Kamera ist entlang der Z-Achse um eine Einheit verschoben und schaut in Richtung Szenenmitte. In diesem Fall wurde eine perspektivische Kamera mit einem Öffnungswinkel von 48.6 Grad initialisiert, um der Kamera aus dem OfflineViewer zu entsprechen. Im OfflineViewer kann man aus Testgründen den Öffnungswinkel der Kamera verändern. Eine solche Veränderung kann zu Fehlern in der Positionierung der Brillemodelle führen, wenn man den Winkel nicht auch im OnlineViewer anpasst. Nachdem man mithilfe von Three.js die Szene initialisiert hat, muss man nur noch die Brillenmodelle an der richtigen Position rendern.

3.2 Modularisierung und Synchronisierung

Ein Problem von Javascript ist, dass es als eine einfache Skriptsprache konzipiert wurde. Heutzutage werden Webseiten immer mehr durch Webanwendungen ersetzt. So steigt auch ihre Komplexität und die Länge des Codes. Wie es auch beim OnlineViewer der Fall ist, wird die Möglichkeit benötigt, den Code zu modularisieren, um die Übersicht und Erweiterbarkeit des gesamten Projekts zu bewahren. Um dieses Problem zu beseitigen wird Require.js verwendet. Ein weiterer Vorteil von Require.js ist die Definition von Abhängigkeiten. Man kann den Code so definieren, dass bestimmte Codeabschnitte erst nach dem Laden der entsprechenden Abhängigkeiten durchgeführt werden.

```
1
2  require(["defaultScene", "loader", "parameters", "controls", "previewviewer",
       "loadingviewer"],
3
4  function(DefaultScene, Loader, Parameters, Controls, PreviewViewer,
       LoadingViewer){
5
6  init();
7
8  });
```

Listing 3.4: Verwendung von Require.js um Module zu laden und sie nach dem Laden zu initialisieren.

In main.js wird zum Beispiel Require.js dazu verwendet, um alle für die Initialisierung der Webanwendung notwendigen Module zu laden und erst wenn diese Module fertig geladen sind mit der Initialisierung durch init() fortzufahren. Die Definition eines Moduls ist einer Klassendefinition aus objektorientierten Sprachen sehr ähnlich und kann auch mithilfe von Require.js erfolgen.

```
1
2  define(['utils/OBJMTLLoader.js', 'utils/OBJLoader.js', 'utils/ConfigLoader.js
       ', 'utils/ZippedOBJLoader.js' ], function(){
3
4  Loader = function(){
5
6  this.OBJMTLLoader = new THREE.OBJMTLLoader();
7
8  this.OBJLoader = new THREE.OBJLoader();
9
10 this.CFGLoader = new THREE.ConfigLoader();
11
12 this.ZippedOBJLoader = new THREE.ZippedOBJLoader();
13
14 }
15
16 ...
```

Listing 3.5: Definition des Loader Moduls im OnlineViewer.

Loader wird hier eine Konstruktorfunktion zugewiesen, um das Modul zu instanzieren. Dem Modul können dann weitere Funktionen durch Zugriff auf die Prototype-Eigenschaft hinzugefügt werden.

```
1
2  ...
3
4  Loader.prototype = {
5
6  loadHead : function(objurl){
7
8  this.OBJLoader = new THREE.OBJLoader();
9
10 this.OBJLoader.addEventListener( 'load', function ( event ) {
```

```
11
12  var object = event.content;
13
14  object.traverse( function ( child ) {
15
16  if ( child instanceof THREE.Mesh ) {
17
18  child.material = new THREE.MeshLambertMaterial({
19
20  opacity: 0.0,
21
22  color: 0x000000,
23
24  transparent : false
25
26  });
27
28  }
29
30  });
31
32  object.receiveShadow = true;
33
34  scene.addHead( object );
35
36  });
37
38  this.OBJLoader.load(objurl);
39
40  },
41
42  loadZippedSpecs : function(zippedobjname){
43
44  this.ZippedOBJLoader = new THREE.ZippedOBJLoader();
45
46  var zippedobjurl = 'obj/' + zippedobjname + '.zip';
47
48  this.ZippedOBJLoader.load(zippedobjurl, function(objurl, mtlurl){
49
50  this.OBJMTLLoader = new THREE.OBJMTLLoader();
51
52  this.OBJMTLLoader.addEventListener( 'load', function ( event ) {
53
54  var object = event.content;
55
56  scene.addSpecs( object );
57
58  });
59
60  this.OBJMTLLoader.load(objurl, mtlurl);
61
62  });
63
```

```
64 }
65
66 ...
```

Listing 3.6: Zugriff auf Prototype-Eigenschaft des Loader Moduls.

Aus Javascript-Perspektive hat man nichts anderes getan, als verschiedene Funktionen und Variablen unter einer Referenz namens Loader zu speichern. Diese Referenz ist der Rückgabewert der Callback-Funktion, die dem Define-Befehl übergeben wurde. Beim Laden des Moduls übernimmt Require.js das Warten auf die Abhängigkeiten des Moduls und die korrekte Rückgabe der Referenz.

Hier wurden nur Codeabschnitte aus dem Modul Loader verwendet, um die Funktionalität von Require.js im OnlineViewer näher zu beschreiben. Um den gesamten Code zu sehen, wird auf die Datei Loader.js verwiesen.

3.3 Laden der Brillenmodelle

Nachdem die Objekt- und Materialinformationen wie schon beschrieben durch Zip.js entpackt wurden, werden sie geparst und in Objekte aus Three.js übertragen. Dieser Vorgang erfolgt in OBJMTLLoader.js, OBJLoader.js und MTLLoader.js. Dabei fügt OBJMTLLoader nur die Funktionalität von OBJLoader.js und MTLLoader.js zusammen, um direkt ein Modell samt seiner Texturen und Materialeigenschaften zu liefern. Dabei wird in Three.js so vorgegangen, dass die Vertices gelesen und zu Faces zusammengefügt werden. Vertices werden in einem Geometry-Objekt gespeichert.

```
1
2 var geometry = new THREE.Geometry();
```

Listing 3.7: Geometry Instanzierung.

Weiters werden durch Index-Referenzen der Vertices Faces innerhalb des Geometry-Objekts erstellt. Geometry-Objekte mit gleichen Materialeigenschaften werden in einem Mesh-Objekt zusammengefasst. Das Mesh-Objekt wird letztendlich einer Gruppe von Meshes hinzugefügt. Dieses Gruppen-Objekt repräsentiert das fertige Brillenmodell.

```
1
2 function addMesh(){
3
4 if(geometry.vertices.length > 0){
5
6 geometry.mergeVertices();
7
8 geometry.computeCentroids();
9
10 geometry.computeFaceNormals();
11
12 geometry.computeBoundingSphere();
13
14 group.add(mesh);
15
```

```
16 geometry = new THREE.Geometry();
17
18 material = new THREE.MeshLambertMaterial();
19
20 mesh = new THREE.Mesh( geometry, material );
21
22 verticesCount = 0;
23
24 }
25
26 }
```

Listing 3.8: Mesh wird dem Gruppen-Objekt hinzugefügt.

Bis jetzt wurde jeder Mesh nur ein Standard-Lambertmaterial hinzugefügt. Während die Vertexinformation geladen wurde, wurde aus dem zweiten File im Zip-Archiv auch schon die Materialinformation extrahiert. Nachdem Material und Objekt-Files fertig geladen wurden, wird die obj3d-Variable nach allen Meshes durchlaufen und jeder Mesh wird ihr gleichnamiges Material hinzugefügt.

```
1
2 if ( mtlDone && obj3d ) {
3
4 // MTL file is loaded and OBJ file is loaded
5
6 // Apply materials to model
7
8 if ( materialsCreator ) {
9
10 obj3d.traverse( function( object ) {
11
12 if ( object instanceof THREE.Mesh ) {
13
14 if ( object.material.name ) {
15
16 var material = materialsCreator.create( object.material.name );
17
18 if ( material ) {
19
20 object.material = material;
21
22 }
23
24 }
25
26 }
27
28 } );
29
30 }
```

Listing 3.9: Gruppenobjekt (obj3d) wird durchlaufen um jeder entahltenen Mesh das richtige Material hinzuzufügen.

Das Modell ist jetzt samt Materialinformation fertig geladen und bereit, der Szene zum Rendern hinzugefügt zu werden.

3.4 Renderschleife

Die Renderschleife wird mithilfe der Funktion requestAnimationFrame( ) erzeugt. Diese Funktion bietet der setInterval-Funktion gegenüber den Vorteil, dass sie eigens für das Zeichnen vom Canvas erzeugt wurde. Beim Aufrufen der Funktion kann man ihr eine Callback-Funktion übergeben und

dadurch eine Schleife erzeugen. Die Funktion wird bestenfalls 60 mal pro Sekunden aufgerufen, was zu einer flüssigen Darstellung von Animationen führt. Ein weiterer Vorteil der Funktion ist, dass sie merkt, ob die Anwendung grade im Vorder- oder Hintergrund ist und die Aufrufe pro Sekunde dementsprechend drosselt. Dadurch spart man Prozessorleistung, während man sich andere Tabs im Browser anschaut oder der Browser minimiert wurde.

3.5 Brillenposition

Die Brillenposition wird aus der Konfigurationsdatei (CFG-Datei), die vom Offlineviewer erstellt wurde, ausgelesen. In der CFG-Datei wird die Position der Brille durch einen 3 dimensionalen Translationsvektor (Location), einen 3 dimensionalen Ausrichtungvektor (Rotation) und einen Saklierungfaktor (Scale) definiert. Die Kinect liefert in der Z-Koordinate des Translationsvektors die Entfernung des Kofpes von der Kamera bei der Aufnahme des Bildes in Meter. Die Werte der Ausrichtung beschreiben jeweils die Drehung der Brillen um die X-,Y- und Z-Achse in Grad. Der Skalierungsfaktor wird benötigt, um die Brillen der Kopfgröße des Benutzers anzupassen. Denn nur weil, die Brille in die richtige Position und Ausrichtung gebracht wurde muss sie nicht notwendigerweise schon die richtige Größe haben, da jeder Kopf unterschiedlich groß ist. Der Skalierungsfaktor ist also gleich eins wenn der Kopf des Benutzers genau so groß ist wie der virtuelle Kopf. Alle drei Werte (Position, Ausrichtung und Skalierung) werden direkt vom Kinect SDK geliefert und vom Offline Viewer in die CFG-Datei geschrieben (siehe http://msdn.microsoft.com/en-us/library/microsoft.kinect.facetracking.iftresult.get3dpose.aspx stand 14.10.2013). Das auslesen der CFG-Datei wurde im ConfigLoader.js implementiert. Dort werden die Daten für jedes aufgenommene Bild ausgelesen und in einer Objektliste eingefügt.

```
1 retObj = {
2   x: xLocs[loc], y: yLocs[loc], z: zLocs[loc],
3   rx: xRots[loc], ry: yRots[loc], rz: zRots[loc],
4   scale: scales[loc],
5   name: names[loc],
6   background: backgrounds[loc]
7 };
```

Listing 3.10: Struktur des Objekts in dem die Brillenkoordinaten der Szene gespeichert werden.

Zusätzlich zur Brillenposition werden im Objekt noch der Name des aufgenommenen Bildes (background) und der Name der Szene (name) gespeichert. Das Bild wird dann geladen und als Hintergrund der Szene gerendert. Da die Ausrichtung der Brille im dreidimensionalen Raum

vom Offlineviewer in Grad angegeben wird, mussten die ausgelesenen Werte noch in Deg umgerechnet und die X- und Y-Achsen mussten gespiegelt werden, um die Daten im Onlineviewer zu visualisieren.

```
1 this.specs.rotation.x = -1 * parameters.rx * Math.PI/180;
2 this.specs.rotation.y = -1 parameters.ry * Math.PI/180;
3 this.specs.rotation.z = parameters.rz * Math.PI/180;
```

Listing 3.11: Umrechnung der Brillenausrichtung vom Offlineviewer in den Onlineviewer

3.6 Kopfposition

Um die Verdeckung der Brille zu berechnen, wird eine durchsichtige Textur auf dem virtuellen Kopf angewendet. Der durchsichtige Kopf dient dazu, die Verdeckungen der Brillen durch den Kopf des Trägers zu simulieren. Nehmen wir ein frontales Bild des Kunden auf, spielt der durchsichtige Kopf keine Rolle. Drehen wir aber den Kopf des Kunden, müssen Verdeckungen durch Nase und Stirn des Brillenträgers simuliert werden. Eine schnelle und effiziente Methode, um diese Verdeckungen zu simulieren, war es, die Brillen auf diesen durchsichtigen Kopf zu setzen, aber seine Position trotzdem in den Z-Buffer zu schreiben. Dadurch ist der Kopf zwar durchsichtig, aber die Pixel der Brillenmodelle, die sich hinter dem Kopf befinden, werden nicht gerendert, weil sie im Z-Buffer als verdeckt erscheinen (siehe Abbildung 3.1). Lässt man also die Position der Brillen unverändert sind sie so positioniert, dass sies auf dem virtuellen Kopf sitzen. Das bedeutet, dass man die gleichen Transformationen auf dem virtuellen Kopf anwenden kann, um diesen mit den Brillen zu bewegen; Kopf und Brillen besitzen die gleiche Modellmatrix.

3.7 Transformationen

Der Benutzer erhaltet im Online Viewer die Möglichkeit die Position der Brillen manuell anzupassen. Dabei werden die Veränderungen die er vornimmt in das Parameter Modul gespeichert (siehe Absatz parameter.js). In der Renderschleife in Main.js werden die im Parameter Modul enthaltenen Szeneninformationen (Position, Ausrichtung und Skalierung) dem DefaultScene Modul übergeben.

```
1 scene.updateSceneWithParams(parameters.getSceneParamsByIndex(scene_index));
```

Listing 3.12: Übertrage Szeneninformationen an das DefaultScene Modul.

Dort werden dann der Translations-, Rotations- und Skalierungsvektor aus dem Parametermodul ausgelesen und auf die Modellmatrix der Brillen- und Kopfmeshes angewendet. Die Matrixoperationen übernimmt dabei three.js.

```
1 scene.updateSceneWithParams(parameters.getSceneupdateSceneWithParams:
    function(parameters){
2   this.specs.scale.x = parameters.scale;
3   this.specs.scale.y = parameters.scale;
4   this.specs.scale.z = parameters.scale;
5   this.specs.position.x = parameters.x + parameters.dx;
6   this.specs.position.y = parameters.y + parameters.dy;
```

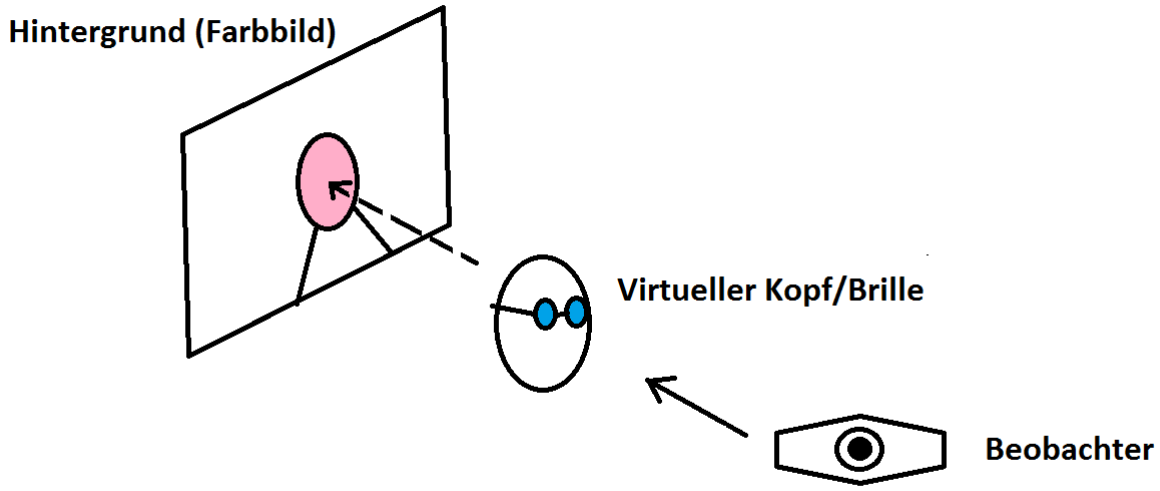


Abbildung 3.1: Skizze des Szenenaufbaus.

```
7    this.specs.position.z = 1 - parameters.z + parameters.dz;
8    this.specs.rotation.x = -1 * (parameters.rx + parameters.drx) * Math.PI
         /180;
9    this.specs.rotation.y = -1 * (parameters.ry + parameters.dry) * Math.PI
         /180;
10   this.specs.rotation.z = (parameters.rz + parameters.drz) * Math.PI/180;
11   this.head.scale = this.specs.scale;
12   this.head.position.x = parameters.x;
13   this.head.position.y = parameters.y;
14   this.head.position.z = 1 - parameters.z;
15   this.head.rotation.x = -1 * (parameters.rx) * Math.PI/180;
16   this.head.rotation.y = -1 * (parameters.ry) * Math.PI/180;
17   this.head.rotation.z = (parameters.rz) * Math.PI/180;
18 }ParamsByIndex(scene_index));
```

Listing 3.13: Übertrage Szeneninformationen an das DefaultScene Modul.