



Bachelorarbeit

Darstellung interaktiver 3D-Grafik im Webbrowser

**Zur Erlangung des akademischen Grades eines
Bachelor of Science**

**angefertigt von
Jonathan Gruber**

**Fakultät: Elektrotechnik und Informatik
Studiengang: Informatik**

**Erstprüfer Prof. Dr. rer. nat. Franz Regensburger
Zweitprüfer Prof. Dr. rer. nat. Thomas Grauschopf**

**ausgegeben am 01. Juli 2014
abgegeben am 04. August 2014**

Inhaltsverzeichnis

1 Das World Wide Web als Anwendungsplattform der Zukunft	1
2 Motivation und Aufbau der Arbeit	3
3 Begrifflichkeit und computergrafische Grundlagen	4
3.1 Grundbegriffe	4
3.2 Standard-Grafikpipeline	7
3.2.1 Modellierung	8
3.2.2 Sicht-Transformation	8
3.2.3 Projektions- und Sichtvolumen-Transformation	9
3.2.4 Clipping	10
3.2.5 Rasterisierung	10
4 Ziel- und Anforderungsanalyse	12
4.1 Thematische Eingrenzung	12
4.2 Charakterisierung der Anwendungsdomäne	13
4.3 Technische Anforderungen	14
4.3.1 Browserunterstützung und Plattformunabhängigkeit	14
4.3.2 Vermeidung von Browser-Plugins	15
4.3.3 Vergleich der Hardware-Anforderungen	16
4.3.4 Import bestehender 3D-Modelle	17
4.3.5 Entwicklungsaufwand	18
4.4 Anforderungen bezüglich des Nutzungserlebnisses	18
4.4.1 Benutzerinteraktion	18
4.4.2 Realistische Grafikdarstellung	19
5 Betrachtung aktueller 3D-Technologien im Webbrowser	20
5.1 Ursprung von Web3D: Virtual Reality Modeling Language	20
5.2 Extensible 3D und X3DOM	22
5.2.1 Aufbau von X3D	22
5.2.2 Das Framework X3DOM	23
5.2.3 Beispiel: Rotierende Pyramide	25
5.3 Web Graphics Library	29
5.3.1 Grundarchitektur	30
5.3.2 Bedeutung und Funktion der Shader	32

5.3.3 Mathematische Bibliotheken	33
5.3.4 Beispiel: Rotierende Pyramide	33
5.3.5 Three.js als Vertreter eines WebGL-Frameworks	41
6 Evaluation der 3D-Technologien	43
6.1 Architektur der Testumgebung	43
6.2 Methodik und Testdurchführung	45
6.2.1 Browserunterstützung und Plattformunabhängigkeit	45
6.2.2 Vergleich der Hardware-Anforderungen	48
6.2.3 Import von 3D-Modellen	50
6.2.4 Benutzerinteraktion	51
6.2.5 Realistische Grafikdarstellung	53
6.2.6 Entwicklungs-Aufwand	54
6.3 Bewertung der Testergebnisse	54
7 Zusammenfassung	56
Quellenverzeichnis	I
Abbildungsverzeichnis	VI
Tabellenverzeichnis	VI
Eidesstattliche Erklärung	VIII
Anhang A: Quelltext der Beispiel-Anwendung	IX
A.1 X3DOM	IX
A.2 WebGL	X
Anhang B: Quelltext der Testumgebung	XV

1 Das World Wide Web als Anwendungsplattform der Zukunft

Durch die zunehmend größere Verbreitung moderner Webstandards wie HTML5 und damit assoziierter Technologien hat sich das World Wide Web (WWW) von einem ehemals simplen Dokumentenbetrachtungssystem zu einer Plattform für vielfältige *Rich Internet Applications* (RIA) entwickelt [24]. Anttonen et al. [1] unterteilen diese Entwicklung in drei grobe Phasen:

Während das Web zu Anfang aus statischen, textorientierten HTML-Dokumente bestand, leitet das Aufkommen JavaScripts 1995 die zweite Phase mit immer mehr interaktiven Elementen ein. Proprietäre, weit verbreitete Browser-Plugins wie *Flash*, *Shockwave* und *Quicktime* ermöglichen zusätzlich zu unbewegten Bildern nun auch Animationen und die Einbettung von Video- und Audio-Inhalten.

Die dritte Stufe stellt schließlich den derzeitigen Übergang zu hochinteraktiven Web-Anwendungen dar, die klassischen Desktop-Applikationen in ihrem Nutzererlebnis (*User Experience*) immer mehr ähneln. Taivalsaari et al. [25] vertreten in ihrer Publikation „*The Death of Binary Software: End User Software Moves to the Web*“ die These, dass konventionelle Software zukünftig in immer größerem Maße durch webbasierte Applikationen ersetzt werde. Googles Linux-basiertes Betriebssystem *Chrome OS* ist dafür ein gutes Beispiel. Nahezu alle Anwendungsprogramme werden hier im Webbrowser Chrome ausgeführt. Zahlreiche in den letzten Jahren aufgekommene Standards wie *Web Storage*, *Web Sockets* und *Media Capture* unterstreichen diesen Bedeutungszuwachs von Web-Anwendungen und stützen die Prognose Taivalsaaris. Ein Bereich, der sich bisher noch kaum im Web etablieren konnte und nach wie vor in erster Linie konventioneller Client-Software vorbehalten ist, ist die hardwarebeschleunigte Darstellung von 3D-Grafik.

In seinem Artikel „*Is 3D Finally Ready for the Web?*“ aus dem Jahr 2010 behandelt Sixto Ortiz Jr. die Fragestellung, ob das Web nach zahlreichen gescheiterten Anläufen inzwischen gerüstet sei, auch diese „letzte Bastion“ [25] traditioneller Software abzubilden.

Der Autor analysiert hierfür den damaligen Stand webbasierter 3D-Grafik und kommt zum Schluß, dass es immer noch viele Hürden gibt, die einen Durchbruch dieser Technik verhindert. Er sah hierbei die Notwendigkeit von Browser-Plugins und den Mangel an Standardisierung als hauptsächliche Probleme.

2 Motivation und Aufbau der Arbeit

Mit der Veröffentlichung des Internet Explorers 11 im Herbst 2013 [45] ist der freie Grafikstandard *Web Graphics Library* (WebGL) erst kürzlich in den Webbrowser von Microsoft eingezogen. Hierdurch sind die Grundvoraussetzungen für die Plugin-freie Darstellung hardwarebeschleunigter 3D-Grafik nun in nahezu allen aktuellen Desktop-Browsern gegeben. Auch im mobilen Bereich ist derzeit eine neue Dynamik zu erkennen. So kündigte Apple im Juni 2014 die Integration von WebGL in ihre mobile Plattform *iOS 8* an [42]. Der Einflussbereich dieser Technologie nimmt damit weiter zu und die Etablierung von 3D-Grafik im *World Wide Web* scheint in greifbare Nähe gerückt.

Vier Jahre nach Ortiz Artikel soll dessen Grundfrage, ob das WWW bereit sei für die dritte Dimension, erneut aufgegriffen werden. Gegenstand dieser Arbeit ist davon ausgehend die Evaluation der zwei offenen Grafikstandards WebGL und X3D bezüglich ihrer Eignung für die Realisierung interaktiver 3D-Grafik im Webbrowser. Hierfür werden die zwei Technologien hinsichtlich verschiedener Kriterien gegenüber gestellt.

Die Arbeit ist wie folgt strukturiert: Zunächst werden einige elementare Grundbegriffe und Prinzipien der Computergrafik erläutert, um das Verständnis der nachfolgenden Ausführungen zu erleichtern. Daraufhin wird die behandelte Anwendungsdomäne der betrachteten Klasse von 3D-Anwendungen konkretisiert und der thematische Umfang der Untersuchung eingegrenzt. Ausgehend von exemplarischen Anwendungsfällen werden anschließend Ziele der Evaluation und Anforderungen einer solchen 3D-Anwendung spezifiziert. Um den grundsätzlichen Aufbau und die Funktionsweise der zwei betrachteten Technologien aufzuzeigen, werden diese im Anschluss daran im Detail beleuchtet. Dasselbe Beispiel verdeutlicht jeweils die paradigmatischen Besonderheiten beider Ansätze. Schließlich wird die Evaluation ausgehend von den zuvor spezifizierten Kriterien durchgeführt. Hierfür werden zum einen die Ergebnisse automatisierter Tests betrachtet und zum anderen solche Aspekte untersucht, die nur schwer quantifiziert werden können. Zuletzt werden die Ergebnisse hinsichtlich der Zielvorgabe bewertet und ein Fazit gezogen.

3 Begrifflichkeit und computergrafische Grundlagen

Bevor die Bewertungskriterien der späteren Evaluation spezifiziert werden, sollen zunächst einige häufig vorkommende Fachbegriffe und elementare Konzepte der Computergrafik erläutert werden, um das Verständnis der weiteren Ausführungen zu erleichtern.

3.1 Grundbegriffe

Ein durch das *World Wide Web Consortium* (W3C) definierter Standard für den Zugriff und die Manipulation von HTML- und XML-Dokumenten ist das **Document Object Model** (DOM). Die durch diese Spezifikation definierte Programmierschnittstelle ermöglicht das Traversieren, Hinzufügen, Ändern und Entfernen von DOM-Elementen, sodass ein Dokument vollständig dynamisch bearbeitet werden kann [16]. Das World Wide Web Consortium ist ein internationales Gremium zur Standardisierung von Techniken rund um das WWW [29].

Ein Sammelbegriff für sämtliche Technologien rund um die Darstellung von 3D-Computergrafik im World Wide Web ist **Web3D**. Dieses beinhaltet sowohl traditionelle Ansätze auf Basis von Browser-Plugins als auch jüngst aufgekommene native Umsetzungen. Die Bildsynthese – oder gängiger: das **Rendering** – bezeichnet die Erzeugung eines zweidimensionalen Einzelbilds, welches ausgehend von einer virtuellen 3D-Darstellung berechnet wird. Dieses Bild kann im Anschluss auf dem Bildschirm oder einem anderen Ausgabemedium angezeigt werden.

Abbildung 3.1 zeigt verschiedene Darstellungen eines **3D-Modells**. Ein solches Modell ist die mathematische Repräsentation eines dreidimensionalen Objekts im Raum, de-

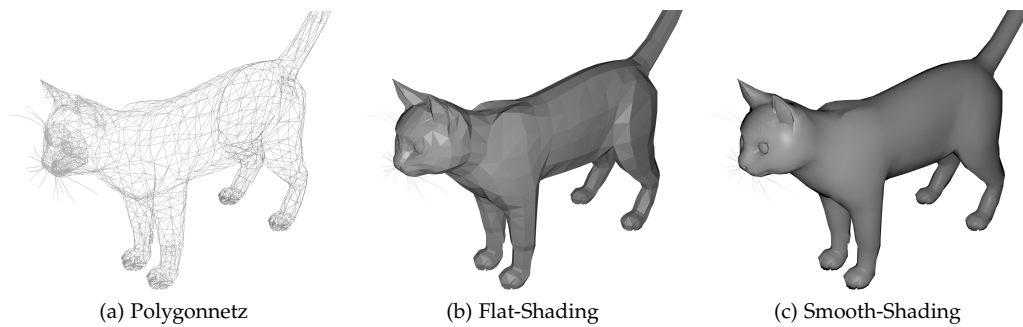


Abb. 3.1: Verschiedene Darstellungen eines 3D-Modells.

finiert durch eine Menge von Punkten (**Vertices**¹) Kanten. Wie die linke Figur zeigt, verbinden Kanten die Vertices zu einem Polygonnetz, welches die Seiten (*Faces*) des Objekts formt. Als Polygone dienen in der Computergrafik typischerweise Dreiecke. Diese sind aufgrund der Eigenschaft dass sie stets planar vorliegen, besonders gut für grafische Berechnungen geeignet [41]. Weiterhin wird die Orientierung der Polygone durch deren Normalenvektor festgelegt. Diese ist zum einen für Sichtbarkeitsentscheide (*Culling*) wichtig und dient zum anderen der Berechnung von Beleuchtungsmodellen. Unter **Culling** wird eine Reihe von Algorithmen verstanden, die darüber entscheiden, welche Objekte einer 3D-Szene zu zeichnen sind und welche nicht. Elemente, die komplett außerhalb des sichtbaren Bereichs liegen, müssen beispielsweise nicht betrachtet werden und verursachen so keinen unnötigen Rechenaufwand.

Ein **Beleuchtungsmodell** stellt das Verfahren dar, welches den Einfluss von Licht in einer 3D-Szene simuliert [26, S. 190 f.]. Hierbei wird zwischen dem lokalen und globalen Beleuchtungsmodell unterschieden. Während Ersteres lediglich den Einfluss der Lichtquellen in einem Punkt der Objektoberfläche berücksichtigt, werden beim globalen Modell auch Reflexion, Transparenzeffekte und Lichtbrechung mit einbezogen.

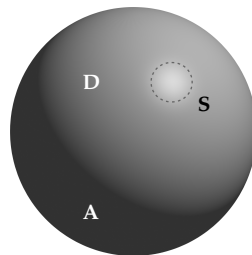


Abb. 3.2: Ambiente, diffuse und spekulare Beleuchtung.

¹Singular: Vertex.

Abbildung 3.2 zeigt wichtige Licht- und Reflexionsmodelle der Computergrafik [12, S. 722 f.]. Ambientes Licht (A) kommt aus allen Richtungen und hat somit auf alle Objekte denselben Einfluss, unabhängig von deren Position im Raum. Es sorgt für die Grundhelligkeit einer Szene, sodass selbst im Schatten liegende Objekte nicht tief-schwarz sind, sondern eine Grauschattierung aufweisen. Die diffuse Reflexion (D) führt zu einer gleichmäßigen Reflexion des Lichts in alle Richtungen. Da der Einfallswinkel des Lichts hierbei berücksichtigt wird, entsteht ein plastischer Eindruck. Die spekula-re oder spiegelnde Reflexion (S) sorgt schließlich für Glanzeffekte (*Highlights*) auf der Materialoberfläche.

Die Abbildungen 3.1b und 3.1c auf der vorherigen Seite zeigen verschiedene Schattie-rungsverfahren (**Shading**). Der weit gefasste Begriff *Shading* bezeichnet die Berechnung der Beleuchtung und Materialeigenschaften von Objektoberflächen. Durch Anwendung von *Flat-Shading* (Mitte) wird die Farbe jedes Polygons durch dessen Normale entspre-chend des lokalen Beleuchtungsmodells berechnet. Hieraus resultiert eine einheitliche Färbung des gesamten Polygons. Das *Smooth-Shading* (rechts) – genauer ein sogenanntes *Phong-Shading*² – führt zu einer glatten Oberfläche, indem die Normalen der Polygone interpoliert werden und die Farbe jedes Pixels hiervon ausgehend erneut durch An-wendung des lokalen Beleuchtungsmodells berechnet wird [26, S. 203 f.]. Das Phong-Verfahren erzeugt weiterhin Glanzeffekte durch spekulare Reflexion (siehe Kopf der Katze in Abbildung 3.1c).

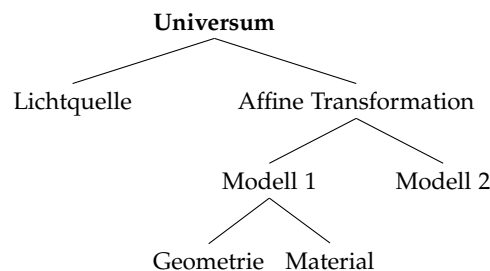


Abb. 3.3: Exemplarischer Szenengraph (unvollständig).

Eine in der Computergrafik häufig vorkommende Datenstruktur zur Repräsentation hierarchischer 3D-Szenen ist der sogenannte **Szenengraph**. Hierbei handelt es sich um einen gerichteten azyklischen Graph, dessen Wurzelknoten das *Universum*, also die voll-ständige Szene darstellt. Kindknoten der Wurzel sind Objekte der Szene und können wiederum weitere Elemente enthalten (vgl. Abbildung 3.3). Transformationen auf einen Knoten werden auf alle Kinder weitervererbt.

²Benannt nach Erfinder des Verfahrens, dem vietnamesischen Computergrafiker Bui Tuong Phong.

3.2 Standard-Grafikpipeline

Ein weiteres elementares Konzept nahezu aller heutigen Grafiksysteme ist die sogenannte Grafik- oder Rendering-Pipeline [12, S. 866-871]. Diese beschreibt die Gesamtheit der Einzelschritte, die nötig sind, um eine virtuelle 3D-Szene mittels mehrerer Transformationen und Berechnungen auf den zweidimensionalen Bildschirm abzubilden. Die späteren Beispiele der betrachteten Technologien werden diese Konzepte praktisch widerspiegeln.

Der Begriff „Pipeline“ kann analog zu klassischen UNIX-Pipelines gesehen werden und verdeutlicht das Prinzip des Verfahrens: Die Ausgabe einer Operation dient als Eingabe des nächsten Prozesses. Die Effizienz der grafischen Berechnungen wird hierbei wie in Abbildung 3.4 angedeutet durch Parallelisierung innerhalb des Grafikprozessors (*Graphics Processing Unit*), dessen Hardware hinsichtlich dieser Anforderung optimiert ist, enorm gesteigert. Vor jedem Schritt muss die Pipeline auf Fertigstellung der vorherigen Berechnung warten. Abbildung 3.4 zeigt den schematischen Aufbau der Pipeline:



Abb. 3.4: Schematische Darstellung der Standard-Grafikpipeline.

Während die Funktionalität der Pipeline zu Anfang der Hardware-Entwicklung statisch umgesetzt war (*Fixed-Function-Pipeline*) und nur durch wenige Parameter angepasst werden konnte, sind heutige Implementierungen durch sogenannte *Shader* vollständig programmierbar und weisen eine hohe Flexibilität auf³. Die Grafikpipeline kann sowohl in Hardware als auch in Software umgesetzt werden. Hardware-Implementierungen sind Software-basierten Realisierungen jedoch hinsichtlich der Effizienz und Geschwindigkeit deutlich überlegen und daher zu bevorzugen.

Im Folgenden werden die Einzelschritte der Pipeline erläutert. Die Details der konkreten mathematischen Berechnungen werden hierbei nicht weiter ausgeführt, da sie den Rahmen der vorliegenden Arbeit übersteigen.

³Shader-Programme werden in Abschnitt 5.3.2 näher behandelt.

3.2.1 Modellierung

Zu Beginn des Verfahrens steht das Laden beziehungsweise die Konstruktion der 3D-Modelle. Nach der Definition von Eckpunkten (*Vertices*) durch Angabe derer kartesischer Koordinaten werden die Modelle durch Polygone zusammengesetzt. Als Polygone dienen typischerweise Dreiecke aufgrund der bereits genannten Vorteile. Diese werden durch je drei Vertices definiert. Natürlich ist eine manuelle Definition der Geometrie in realen Anwendungsfällen bei komplexen Modellen mit zehntausenden von Vertices ausgeschlossen. Gemeinhin entstammen die geometrischen Daten spezieller 3D-Modellierungssoftware wie *Blender* [55], die die Erstellung aufwendiger Modelle erheblich erleichtern.

Die Koordinaten der Vertices liegen zunächst relativ zur „Modelmitte“ in einem lokalen Koordinatensystem vor. Mittels affiner Transformationen können die Modelle im Raum verschoben, skaliert, rotiert und verzerrt werden. Dies wird durch Multiplikation der entsprechenden, homogenen Transformationsmatrix mit allen Punkten des Modells realisiert. Weiterhin können mehrere Transformationen durch Multiplikationen aller beteiligten Matrizen durch eine einzige gleichwertige Modellmatrix \underline{M} ersetzt werden. Da die Matrizenmultiplikation nicht kommutativ ist, ist die Reihenfolge der Operation entscheidend.

Eine Überführung der Modelkoordinaten in sogenannte Weltkoordinaten wird somit durch folgende Berechnung vollzogen. Dabei sei \vec{v}_i der Spaltenvektor der Koordinaten des aktuell betrachteten Vertex.

$$\sum_{i=0}^n \underline{M} * \vec{v}_i$$

Abb. 3.5: Berechnung der Modellmatrix.

3.2.2 Sicht-Transformation

Anschließend wird die Ansicht der Szene mittels mehrerer Parameter festgelegt. Der Aug- oder Beobachtungspunkt, welcher der „Kamera“ der Szene entspricht, wird zunächst im Weltkoordinatensystem platziert. Des Weiteren wird das Zentrum der Be-

trachtung der (*Look-At-Punkt*) und der *Oben-Vektor* spezifiziert. Dieser legt die Orientierung der Ansicht fest, indem er die Richtung „oben“ definiert.

All diese Parameter werden innerhalb der Sichtmatrix \underline{V} abgebildet. Durch Multiplikation der Modellmatrix und der Vertices mit der Sichtmatrix erfolgt die Transformation in das sogenannte Sichtkoordinatensystem. Die Welt wird hierdurch relativ zur Kamera verschoben, welche nun im Ursprung liegt.

$$\sum_{i=0}^n \underline{V} * \underline{M} * \vec{v}_i$$

Abb. 3.6: Berechnung der MV-Matrix.

3.2.3 Projektions- und Sichtvolumen-Transformation

Der nächste Schritt ist die Projektion der Darstellung auf einen Einheitswürfel zur Vorbereitung der Bildschirm-Transformation. Hierfür muss zunächst ein Sichtvolumen definiert werden. Dieses grenzt den darzustellenden Bereichs des 3D-Raums durch Ebenen ein und wird für das spätere *Clipping* benötigt. Um die Darstellungseffizienz zu steigern, werden dabei Objekte außerhalb dieser Abgrenzung verworfen.

Es stehen verschiedene Projektions-Modelle zur Verfügung. Von besonderer Bedeutung sind die orthogonale Parallelenprojektion und die perspektive Projektion. Bei der Parallelenprojektion wird das Sichtvolumen durch einen Quader beschrieben, welcher mittels Angabe der Dimensionen für die Ausdehnung des Quaders in alle drei Achsen aufgespannt wird. Die spätere Transformation auf den Einheitswürfel ist in diesem Fall mittels einer Skalierung einfach umzusetzen. Zur Erzielung eines perspektivischen Effekts, also der Größendarstellung von Objekte in Abhängigkeit von deren Entfernung zum Beobachtungspunkt, wird bei der perspektivischen Projektion ein Pyradmidenstumpf (*Frustum*) verwendet. Durch die verzerrende Abbildung des Frustums auf den Einheitswürfel entsteht so die angestrebte kleinere Darstellung weiter entfernter Elemente.

$$\sum_{i=0}^n \underline{P} * \underline{V} * \underline{M} * \vec{v}_i$$

Abb. 3.7: Berechnung der MVP-Matrix.

Analog zu den vorherigen Schritten wird auch diese Berechnung durch eine Matrixmultiplikation realisiert. Durch Multiplikation der bisherigen Transformationen mit der Projektionsmatrix, welche die Parameter des gewählten Projektionsmodells enthält, wird hierbei die finale Transformation der Vertices erzielt.

Wie Formel 3.7 zeigt, kann die gesamte Modell-, Sicht- und Projektionstransformation durch das Matrizenprodukt von \underline{P} , \underline{V} und \underline{M} realisiert werden. Dieses Gesamtprodukt der Transformation wird als *MVP-Matrix* bezeichnet. Anstatt für jeden Vertex drei teure Matrizenmultiplikationen durchzuführen, kann die Berechnungseffizienz deutlich gesteigert werden, indem jeder der Punkte lediglich mit der MVP-Matrix multipliziert wird.

3.2.4 Clipping

Nachdem bereits Polygone, welche entweder abgewandt (*Backface-Culling*) oder komplett außerhalb des Sichtvolumens liegen (*Frustum-Culling*⁴), verworfen wurden, ist der nächste Schritt das sogenannte *Clipping*. Dieses Verfahren betrachtet Polygone, welche das Sichtvolumen schneiden und daher nur teilweise gezeichnet werden müssen. Zu diesem Zweck werden die Schnittpunkte des Polygons mit der Abgrenzung berechnet und anhand dessen ein entsprechendes neues Teilpolygon erstellt. Der Algorithmus von *Cohen-Sutherland* ist ein bekanntes Beispiel für die Realisierung dieses Vorgangs.

Häufig wird dieser Schritt auch der nachfolgenden Rasterisierung zugeordnet. Die Reihenfolge der einzelnen Berechnungen in der Pipeline kann je nach Grafiksystem und Implementierung variieren.

3.2.5 Rasterisierung

Der letzte Schritt der Grafikpipeline ist schließlich die Rasterisierung, welche der finalen Abbildung der 3D-Darstellung auf einzelne Bildpunkte (*Pixel*) dient. Aufgrund des durch die vorherigen Transformationen entstandenen Koordinatensystems in Form eines Einheitswürfels ist eine Abbildung auf die diskreten Abmessungen des Ausgabefensters einfach umzusetzen. Der Rasterisierungs-Prozess kann in zwei Unterpunkte gegliedert werden:

⁴Bei einer perspektivischen Projektion.

Zunächst wird mittels eines *Scanline*-Verfahrens eine Verdeckungsrechnung der Polygone durchgeführt. Hierdurch wird bestimmt, welche Punkte der Darstellung zu zeichnen sind und welche nicht, da sie durch Figuren im Vordergrund verdeckt werden. Ein Scanline-Verfahren ist ein Algorithmus, welcher Zeile für Zeile über ein gerastertes Bild iteriert und zu zeichnende Pixel ermittelt. Der zweite Schritt ist anschließend die Einfärbung dieser Pixel durch den in Abschnitt 3.1 kurz umrissenen Shading-Prozess. Hierbei stehen verschiedene Verfahren wie *Flat*- und *Smooth-Shading* zur Verfügung, welche zu verschiedenen Graden an Realismus und Glattheiten der Oberfläche führen. Ebenso wird hier die Texturierung der 3D-Modelle vorgenommen.

4 Ziel- und Anforderungsanalyse

Nachdem die Grundlagen für die weiteren Ausführungen geschaffen wurden, wird nachfolgend der thematische Umfang der Arbeit eingegrenzt. Anschließend werden die Ziele und Anforderungen der betrachteten Anwendungsdomäne ausgeführt.

4.1 Thematische Eingrenzung

Derzeitige Web3D-Technologien können anhand ihrer grundsätzlichen technischen Ausprägung grob in zwei Kategorien eingeteilt werden [15]. Vertreter des deklarativen Paradigmas bauen auf bestehende Webstandards auf und realisieren eine 3D-Darstellung mittels eines hierarchischen, direkt in HTML eingebundenen Szenengraphen. Kindelemente des Szenengraphen stellen hierbei die Objekte der 3D-Szene dar und werden in XML-ähnlicher Notation direkt innerhalb des *Document Object Models* integriert. Diesem Ansatz gegenüber stehen imperative Ansätze, welche auf einem deutlich niedrigeren Abstraktionsniveau angesiedelt sind und generell eine größere Hardwarenähe aufweisen. Ähnlich zu klassischen Grafikbibliotheken wie *OpenGL* und *Direct3D* wird die eigentliche Grafikprogrammierung hier durch explizite Einzelschritte auf Vertex- und Polygonebene umgesetzt.

Mit dem durch das Web3D Consortium entwickelten ISO-Standard *Extensible 3D* (X3D) und dem zugehörigen Framework *X3DOM* wird zunächst ein Repräsentant des deklarativen Modells betrachtet. Bei X3D handelt sich um eine XML-basierte Beschreibungssprache für die Darstellung webbasierter 3D-Grafik. X3DOM stellt ein JavaScript-basiertes Integrationsmodell für X3D in HTML dar.

Im Anschluss wird die *Web Graphics Library* (WebGL) als Vertreter einer imperativen Grafikbibliothek erörtert. WebGL basiert auf der *Open Graphics Library for Embedded Systems 2.0* (OpenGL ES), ein auf Mobilgeräte abgestimmter Dialekt von *OpenGL 2.0*

[19]. Die Bibliothek wurde hierbei von zahlreichen Altlasten wie Funktionen der Fixed-Rendering-Pipeline befreit. OpenGL ist eine weit verbreitende plattformunabhängige Spezifikation für 2D- und 3D-Computergrafikanwendungen [38]. Die Schnittstelle zwischen WebGL und dem Webbrowser wird dabei durch JavaScript realisiert.

Eine weiterer derzeitige Ansatz für 3D-Darstellung im Webbrowser ist die von Adobe als Bestandteil ihrer proprietären Multimedia-Plattform *Flash* entwickelte Technologie *Stage 3D*. Obgleich Flash im Desktop-Bereich äußerst verbreitet ist, ist eine Unterstützung auf den vorherrschenden mobilen Betriebssystemen *Android* und *iOS* gar nicht oder nur nach Installation entsprechender Software möglich [11]. Aufgrund dieses Umstands und des Fokus der Arbeit auf offene Standards ist Stage 3D daher nicht Bestandteil der Untersuchung.

4.2 Charakterisierung der Anwendungsdomäne

Bevor die angestrebten Ziele und Testkriterien der Evaluation formuliert werden können, muss zunächst die grundsätzliche Natur der untersuchten Klasse von 3D-Anwendung charakterisiert werden, um eine Vorstellung für deren spezifische Anforderungen zu schaffen. Mittels exemplarischer Anwendungsfälle wird die Anwendungsdomäne konkretisiert und erleichtert die nachfolgende Analyse:

1. Darstellung von Produkten in Web-Shops und auf Hersteller-Websites mittels eingebetteter 3D-Modelle: Interessierte Kunden können die Ansicht interaktiv durch Klicken und Ziehen anpassen und das Produkt so von allen Seiten detailgenau betrachten. Falls es verschiedene Ausführungen des Artikels wie Farbe oder Material gibt, so kann dies dynamisch verändert werden.
2. Virtuelle Repräsentation von Ausstellungsstücken im Kulturbereich: Einige Museen und andere kulturelle Einrichtungen nutzen heutzutage 3D-Scanner zur Katalogisierung ihrer Exponate. Diese 3D-Modelle können auf Websites eingebunden werden und anderen auf diesem Gebiet forschenden Experten so zur Verfügung gestellt werden.
3. Visualisierung wissenschaftlicher Modelle im akademischen und schulischen Umfeld wie beispielsweise die Darstellung von Anatomie in der Medizin oder die Abbildung komplexer Moleküle in der Chemie: Studenten und Schüler können durch Erforschen der 3D-Darstellung die Zusammenhänge und räumlichen Verhältnisse des gezeigten Sachverhalts selbstständig erfassen. Darüber hinaus kön-

nen Konzepte mittels Animationen und der Möglichkeit, einzelne Darstellungsebenen auszublenden, sehr gut veranschaulicht werden.

Explizit nicht Teil der weiteren Betrachtung sind browserbasierte Computerspiele, da diese für sich ein sehr weitläufiges Thema darstellen und den Umfang der Arbeit überschreiten.

4.3 Technische Anforderungen

4.3.1 Browserunterstützung und Plattformunabhängigkeit

Um eine webbasierte 3D-Anwendung einer möglichst großen Zielgruppe zugänglich zu machen, ist die grundsätzliche Unterstützung der 3D-Technologie innerhalb des Webbrowsers und des Betriebssystems notwendig. Abbildung 4.1 zeigt die Entwicklung der weltweiten Marktanteile⁵ von Webbrowsern auf Desktop- und Mobilgeräten innerhalb des Zeitraums von Januar 2009 bis April 2014.

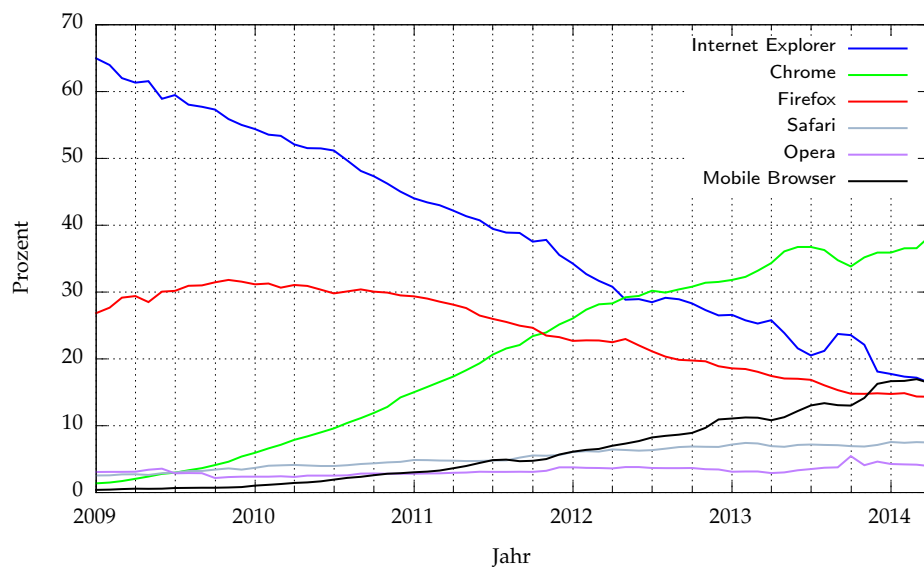


Abb. 4.1: Marktanteil von Webbrowsern weltweit [50].

⁵Aufgrund der Schnellebigkeit des World Wide Webs sind Aussagen über die tatsächlichen Marktanteile von Webbrowsern grundsätzlich schwierig zu treffen und stets mit Unschärfe versehen.

Deutlich erkennbar ist ein kontinuierlich starker Rückgang des Microsoft Internet Explorers von ehemals 65,0% zu Anfang des Jahres 2009 auf nur noch 16,4% im April 2014. Gleichzeitig stieg der Marktanteil von Google Chrome seit dessen Erscheinungsjahr 2008 stetig auf 38,3% an. Seit 2011 ist Firefoxs Anteil rückläufig und reduzierte sich annähernd um die Hälfte von 26,7% auf 14,3%. Während sich Safari auf niedrigem Niveau von 2,6% auf 7,5% leicht steigern konnte, weist Operas Kurve nur geringfügige Veränderungen auf. Innerhalb des betrachteten Zeitraums stieg der Anteil hier von 3,1% auf 4,0%. Das Diagramm zeigt zudem eine signifikante Zunahme von mobilen Webbrowsers, die einen klaren Trend hin zur mobilen Nutzung des Webs belegen. So stieg dieser Anteil von 0,4% stetig auf 16,4%, was fast einem Sechstel aller Zugriffe entspricht. Eine Erhebung des Statistischen Bundesamts aus dem Jahr 2013 stützt diese These: 2013 nutzten 51% aller Internetbenutzer im Alter ab 10 Jahren in Deutschland das mobile Internet [33].

Auf Basis dieser Daten definiert sich die erste Zielvorgabe: Die nähere Überprüfung der Unterstützung der betrachteten 3D-Technologie innerhalb der fünf meistverwendesten Desktop-Browsern, also Chrome, Internet Explorer, Firefox, Safari und Opera. Je nach Verfügbarkeit der Browser werden diese auf den drei verbreitesten Betriebssystemen, Microsoft Windows, Apple Mac OS X und GNU/Linux getestet.

Aufgrund des wie in Abbildung 4.1 gezeigten, zunehmend stärkeren Einflusses mobiler Geräte stellt deren Untersuchung hinsichtlich dieses Kriteriums ebenso eine Anforderung dar. Betrachtet werden hierbei das hauptsächlich durch Google entwickelte freie Betriebssystem *Android* und Apples mobile Plattform *iOS*.

4.3.2 Vermeidung von Browser-Plugins

Eine weitere Anforderung stellt die Darstellung der 3D-Szene unter Vermeidung zusätzlicher Browser-Plugins von Drittherstellern dar. In der langjährigen Geschichte von Web3D gab es zahlreiche Plugin-basierte Ansätze, die sich allesamt langfristig nicht behaupten konnten [11].

Sofern eine explizite Aktion seitens des Benutzers, etwa das Installieren eines Browser-Plugins nötig ist, um eine Webseite mit eingebetteter 3D-Grafik zu betrachten, stellt dies eine Einstiegshürde für den Webauftritt dar und kann dessen Benutzerakzeptanz senken. Viele Benutzer stehen Plugins generell eher skeptisch gegenüber, da sie unbekannte

Software darstellen, was häufig von Laien als eine potentielle Gefährdung ihres Computers erachtet wird. Zudem sind Plugins oftmals mühsam zu verwalten sind [21]. Das Installieren von Software verlangt zudem eine gewisse Fachkompetenz des Benutzers, die nicht immer gegeben ist. Dies ist in Anwendungsdomänen wie Schulen, Universitäten und Firmen darüber hinaus aufgrund mangelnder Administrationsrechte grundsätzlich nicht möglich.

Veraltete oder wenig verbreitete Browser-Plugins können tatsächlich ein nicht unerhebliches Sicherheitsrisiko innerhalb des Browsers darstellen, da unentdeckte Schwachstellen innerhalb des Plugin-Codes von Angreifern ausgenutzt werden können (*Exploits*) [21]. Weiterhin können Browser-Plugins zu Abstürzen der Software führen. Sowohl Google Chrome als auch Mozilla Firefox führen aufgrund dieser möglichen Sicherheits- und Stabilitätsprobleme zahlreiche als unsicher erachtete Plugins erst nach expliziter Autorisierung des Benutzers aus [40] [46].

Aufgrund all dieser Unzulänglichkeiten von Browser-Plugins sind diese als Abhängigkeit einer Web3D-Anwendung zu vermeiden.

4.3.3 Vergleich der Hardware-Anforderungen

Ebenso entscheidend für die Alltagstauglichkeit und Benutzerakzeptanz der 3D-Applikation ist deren Hardware-Anforderung und die generelle *Performance* bei der Darstellung. Da letztere in erster Linie von der verfügbaren Grafik-Hardware abhängt, ist eine absolute Bewertung der Leistungsfähigkeit von X3DOM und WebGL schwierig vorzunehmen. Für die Bewertung der zwei Technologien müssen diese einander daher auf demselben Testsystem gegenüber gestellt werden, um vergleichbare Aussagen bezüglich ihrer Performance treffen zu können. In der Evaluation soll diese auf verschiedenen Betriebssystemen und Webbrowsern verglichen werden.

Weiterhin besitzt die Bildschirmauflösung großen Einfluss auf die Hardware-Anforderung der 3D-Anwendung, da sich der Berechnungsaufwand während des Renderings durch die höhere Pixelanzahl vergrößert. Abbildung 4.2 zeigt die zehn häufigsten Bildschirmauflösungen innerhalb der Spieleplattform Steam des amerikanischen Spieleentwicklers Valve Cooperation im April 2014. Die Statistik entstammt einer nicht repräsentativen monatlich durchgeführten Hardware-Umfrage innerhalb dieses Netzwerks. Die mit Abstand am häufigsten gemessenen Bildschirmauflösungen sind 1920 x 1080 Pixel

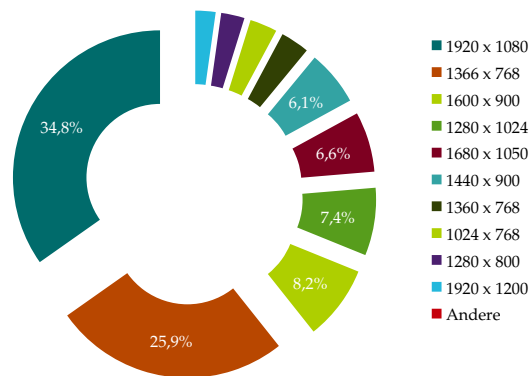


Abb. 4.2: Zehn meistgenutzte Bildschirmauflösungen auf Steam [32].

(*FullHD*) und 1366 x 768 Pixel. Als weiteres Kriterium der Evaluation wird daher die Untersuchung des Einflusses der nachfolgend aufgelisteten fünf Bildschirm-Auflösungen auf die Performance spezifiziert.

- 1024 x 768
- 1280 x 1024
- 1376 x 768
- 1600 x 900
- 1920 x 1080

4.3.4 Import bestehender 3D-Modelle

Bei der Umsetzung anspruchsvoller 3D-Szenen ist die Verwendung von 3D-Modellierungssoftware wie *Blender* [55], *3ds Max* [53] oder *Maya* [52] für die Erstellung komplexer Modelle aufgrund der hohen Zahl von Polygonen und komplizierter Texturierungstechniken unabdingbar. Das einfache und performante Laden solcher im Vorfeld erstellten 3D-Daten ist daher für Anwendungen, die über die Darstellung simpler geometrischer Körper hinausgeht, wesentlich. Um die initiale Ladezeit zu reduzieren, ist das asynchrone Laden von Inhalten von ebenso großer Bedeutung.

Um die Tauglichkeit der verschiedenen Technologien diesbezüglich zu erproben, wird innerhalb der Testumgebung die generelle Unterstützung für das Laden externer 3D-Modelle erprobt.

4.3.5 Entwicklungsaufwand

Ein ebenso zu beachtender Aspekt ist der Aufwand bei der Entwicklung einer Web3D-Applikation. Ein typischer Webentwickler besitzt Expertenwissen bezüglich der klassischen Websprachen HTML, CSS und JavaScript. Weiterhin kann davon ausgegangen werden, dass er mit mindestens einer der derzeit vorherrschenden serverseitigen Programmiersprachen PHP, Python oder Ruby vertraut ist. Systemnahe, statisch typisierte Sprachen wie C und C++ können dagegen tendenziell nicht in das Repertoire eines solchen Programmierers gezählt werden. Paulson [23] führt im *Computer Magazine* aus, dass in den letzten Jahren generell ein Trend hin zu dynamischen Sprachen beobachtet werden könne. Hardwarenahe Grafikbibliotheken, die klassischen systemnahen Programmiersprachen wie C ähneln, können daher ein Hindernis bei der Erstellung einer Web3D-Anwendung darstellen.

Ausgehend von diesen Annahmen muss also auch das Vorwissen des Software-Entwicklers bei der Bewertung der betrachteten 3D-Techniken mit einfließen. Sofern neue und bis dato unbekannte Konzepte vorliegen, kann dies die Einarbeitungszeit deutlich verlängern und somit die Entwicklungskosten erhöhen.

4.4 Anforderungen bezüglich des Nutzungserlebnisses

4.4.1 Benutzerinteraktion

Erst durch die Möglichkeit des Benutzers, mit den Objekten innerhalb der 3D-Welt zu interagieren, kann sich das Potential einer 3D-Anwendung voll entfalten. Im Gegensatz zu einer statischen, fixen Darstellung von Gegenständen kann durch die benutzergesteuerte Navigation im Raum ein reichhaltigeres, lebendigeres Benutzererlebnis erzielt werden, da der Anwender aktiv Einfluss auf die dargestellte Welt nimmt.

Eine mögliche Navigations-Methode, im Folgenden als Orbit-Navigation bezeichnet, erlaubt das Verschieben, Skalieren und Rotieren von Objekten mit der Maus, wodurch diese aus beliebigen Betrachtungswinkeln untersucht werden können. Darüber hinaus ist eine freie Navigation, in welcher durch die Szenerie „geflogen“ werden kann, gängig. Die Steuerung wird hierbei ebenfalls mit der Maus und zusätzlich der Tastatur

realisiert. Die betrachteten 3D-Technologien sollen hinsichtlich dieser und weiterer Navigationsmodelle untersucht werden.

Außerdem soll eine mögliche Umsetzungen für sogenanntes *Picking* betrachtet werden. Picking beschreibt die Auswahl einzelner Objekte im Raum durch den Mauszeiger, um beispielsweise mehr Informationen zu diesem Gegenstand zu erhalten. Durch die dritte Dimension gestaltet sich dieser Prozess dabei weitaus weniger einfach als in einer zweidimensionalen Darstellung, da die Grafikprojektion umgekehrt werden muss.

4.4.2 Realistische Grafikdarstellung

Ein weiterer wichtiger Punkt ist der Grad an Realismus der gezeigten 3D-Szene. Bezugnehmend auf die in Abschnitt 4.2 umrissenen Anwendungsfälle sollte beispielsweise die Darstellung eines Produkts in einem Web-Shop möglichst nahe an das reale Vorbild heranreichen. Um eine wirklichkeitsgetreue Darstellung der 3D-Szene zu erzielen, ist insbesondere deren Beleuchtung entscheidend. Dabei ist das ambiente Licht sowie die diffuse und spekulare Reflexion ausschlaggebend. Durch Anwendung aufwendiger Schattierungsverfahren wie Phong-Shading können Materialeigenschaften mit realistischen Reflexionen erzeugt werden. Auch der Schattenwurf ist für ein rundes Gesamtbild der 3D-Szene wichtig. Die Texturierung der Modelle stellt schließlich eine verhältnismäßig billige Möglichkeit dar, diese realistischer abzubilden.

Globale Beleuchtungsmodelle wie *Raytracing* und *Radiosity* sind in der Lage, nahezu fotorealistische Bilder durch sehr aufwendige Berechnungsverfahren zu erzielen. Aufgrund der hohen Anforderungen an die Hardware und der langen Rechenzeiten sind diese Modelle jedoch für Echtzeit-Grafikdarstellungen im Webbrowser ungeeignet und werden daher nicht näher behandelt.

Um X3DOM und WebGL bezüglich der realistischen Grafikdarstellung zu erproben, wird ein texturiertes 3D-Modell in jeder Technologie mit den ausgeführten Beleuchtungstechniken dargestellt und im Anschluss verglichen.

5 Betrachtung aktueller 3D-Technologien im Webbrowser

5.1 Ursprung von Web3D: Virtual Reality Modeling Language

Vor der näheren Betrachtung von X3D und WebGL soll zunächst deren Ursprung dargestellt werden, um die grundlegenden Konzepte dieser heutigen Technologien zu erläutern. Die Geschichte von Web3D reicht über zwei Jahrzehnte zurück, in denen es eine Vielzahl von Bestrebungen verschiedenster Institutionen gab, 3D-Grafik im World Wide Web (WWW) zu etablieren. Die überwiegende Mehrheit dieser Ansätze scheiterte jedoch und konnte nie eine erwähnenswerte Bedeutung im WWW erlangen [11]. Erst innerhalb der letzten Jahre entwickelte sich durch das Aufkommen WebGLs eine neue Dynamik innerhalb dieses Felds.

Die ersten Anfänge der webbasierten 3D-Computergrafik gehen bis ins Frühjahr 1994 zurück, als der Erfinder des noch jungen World Wide Webs, Tim Berners-Lee, öffentlich dazu aufrief eine Spezifikation für virtuelle Realität zu entwickeln [22]. Der Informatiker Dave Raggett, der seit 1992 an der Entwicklung zahlreicher Web-Standards richtungsweisend beteiligt war [30], reichte daraufhin seine Publikation „*Extending WWW to support Platform Independent Virtual Reality*“ für die im selben Jahr stattfindende erste internationale World Wide Web Konferenz (WWW1) in Genf ein. Raggett prägte innerhalb dieser Arbeit den Begriff *Virtual Reality Markup Language* (VRML⁶) und beschrieb seine Vision von webbasierter virtueller Realität [49]: Eine Auszeichnungssprache auf Basis der *Standard Generalized Markup Language* (SGML) soll es ermöglichen, hierarchische 3D-Szenen auf einer abstrakten Ebene deklarativ zu beschreiben.

⁶Gesprochen *vermal*.

Auf der WWW1 im Mai 1994 bildete sich daraufhin auf Initiative von Ragget und Berners-Lee hin die sogenannte VRML BOF⁷. Ziel dieser Expertenrunde war die Diskussion bereits bestehender Ansätze von 3D-Visualisierung mit Fokus auf der Interaktion mit dem WWW. Der auf diesem Gebiet Pionierarbeit leistende Ingenieur Mark Pesce stellte hierbei einen in Zusammenarbeit mit dem Softwareentwickler Tony Parisi entwickelten einfachen Prototypen vor [22]. Die Demonstration zeigte eine einfache, dreidimensionale Darstellung eines Würfels und ermöglichte die Interaktion mit dem WWW, indem eine URL aufgerufen wurde, wenn man auf die Geometrie klickte. Der Grundstein für die weitere Entwicklung der Virtual Reality Modeling Language im Besonderen und der webbasierten Computergrafik im Allgemeinen war somit gelegt. Um den Bezug zur Computergrafik zu verdeutlichen, wurde die Sprache kurz darauf in *Virtual Reality Modeling Language* umbenannt [4].

Die daraufhin gegründete, schnell wachsende Arbeitsgruppe rund um die *www-vrml* Mailing-Liste einigte sich auf das ehrgeizige Ziel, eine erste Spezifikation noch innerhalb des Jahres 1994 fertigzustellen [4]. Aufgrund der guten Basis, die das *Inventor Format* des amerikanischen Computerherstellers *Silicon Graphics* bot, entschied man sich für dieses als Ausgangspunkt für die weitere Entwicklung der Sprache. Tatsächlich konnte VRML 1.0 im November vollendet werden.

Zwei Jahre später wurde deren deutlich ausgereifere Version 2.0 der Spezifikation fertiggestellt und 1997 zu einem ISO-Standard⁸ [14] erhoben. Zur stetigen Weiterentwicklung und Schutz dieser neuen, als *VRML97* bekannt gewordenen, offenen Spezifikation wurde kurze Zeit später das *VRML Consortium* gegründet. Die Non-Profit-Organisation, die inzwischen den Namen *Web3D Consortium* trägt, betreut bis heute die Entwicklung des VRML-Nachfolgers *X3D* und treibt dessen Verbreitung voran. Sie setzt sich aus Unternehmen, akademischen Institutionen und Einzelpersonen zusammen [28].

Trotz der hohen Erwartungen konnte sich die VRML nie in dem Maße im WWW durchsetzen, wie es sich deren Entwickler zunächst erwartet hatten. Die Gründe hierfür sind vielfältig: Zunächst war die Installation eines Browser-Plugins zwingend notwendig, um das Format innerhalb des Webbrowsers darstellen zu können. Die Hardware des durchschnittlichen damaligen Personal Computers wies darüber hinaus in vielen Fällen eine unzureichende Leistung, sodass rechenaufwändige 3D-Szenarien nicht zu realisieren waren. Weiterhin war die Geschwindigkeit damaliger Internetanschlüsse zu gering,

⁷Eine sogenannte *Birds of a feather* (BOF) stellt eine informelle Diskussionsrunde von Experten zu einem spezifischen Thema dar. Diese finden insbesondere im Umfeld der Internet Engineering Task Force häufig Anwendung [20] [35].

⁸ISO/IEC 14772-1:1997.

um die relativ großen Datenmengen, die bei Computergrafik typischerweise anfallen, bewältigen zu können.

5.2 Extensible 3D und X3DOM

5.2.1 Aufbau von X3D

Zwei Jahre nach Veröffentlichung des VRML97-Standards begann eine Arbeitsgruppe des Web3D Consortiums 1999 mit der Entwicklung des Nachfolgers von VRML, *Extensible 3D* [9]. Dieser X3D abgekürzte, freie Standard wurde 2004 ebenso wie zuvor VRML zu einem ISO-Standard⁹ [7]. Bei der Entwicklung X3Ds wurde auf eine Abwärtskompatibilität zu VRML97 geachtet. Das „X“ im Namen impliziert bereits die Nutzung der *Extensible Markup Language (XML)* als Grundlage dieses neuen Grafikformats. Hierdurch wurde eine größere Konvergenz zu bestehenden Webtechnologien wie HTML und XHTML erzielt.

X3D nutzt einen gerichteten, azyklischen Szenengraph zur Repräsentation einer 3D-Szene (vgl. Abbildung 3.3). Knoten unterhalb der Wurzel des Graphen (dem *Univsum*) stellen dabei sämtliche Objekte der Welt dar. Die X3D-Spezifikation definiert eine große Zahl von Knoten, welche alle benötigten Elemente einer Grafikdarstellung wie Geometrien, Lichtquellen, Kameras, Texturen, Transformationen et cetera abbilden. Der Szenengraph kann in drei Formaten abgespeichert werden. Zur Verfügung stehen ein XML-Format, die klassische VRML-Notation und schließlich ein komprimiertes Binärformat [9]. Für die Darstellung des Formats ist ein spezieller *X3D-Viewer* notwendig. Dieser erzeugt aus dem X3D-Szenengraph eine 3D-Darstellung, die frei von allen Seiten betrachtet werden kann.

Da die X3D-Spezifikation sehr umfangreich ist, existieren sogenannte *Profile*, welche eine Menge von Knoten für eine bestimmte Anwendungsdomäne bündeln. Hierdurch ist es möglich, nur tatsächlich benötigte Knoten des Standards zu laden.

⁹ISO/IEC 19775-1.

5.2.2 Das Framework X3DOM

Entstehung

Während der Ausarbeitung des neuen HTML5-Standards stellte X3D den vom World Wide Web Consortium favorisierten Standard für die Realisierung von Web3D-Inhalten dar. Der Arbeitsentwurf der HTML5-Spezifikationen aus dem Jahr 2009 vermerkt diesbezüglich:

„Embedding 3D imagery into XHTML documents is the domain of X3D, or technologies based on X3D that are namespace-aware.“ [13]

Diese äußerst knappe Formulierung definiert jedoch in keinsten Weise näher, wie eine Plugin-freie Integration von X3D in HTML aussehen könnte. Behr et al. vom Fraunhofer Institut für graphische Datenverarbeitung Darmstadt publizierten daraufhin im selben Jahr ihren Ansatz, wie X3D-Inhalte direkt in HTML eingebettet werden könnten [3]. Ziel des vorgeschlagenen Integrations-Modells *X3DOM* ist eine wechselseitige Verknüpfung zwischen bestehenden Webtechnologien wie dem *Document Object Model* und X3D. Die Autoren erhoffen sich, langfristig eine ähnliche Entwicklung anzustoßen, wie sie das Vektorgrafikformat *Scalable Vector Graphics* (SVG) im WWW durchlief [3]. Während SVG zunächst nur in einigen wenigen Browsern unterstützt wurde, ist dieser Standard inzwischen überall nativ implementiert [34]. Zum Zeitpunkt der Veröffentlichung dieser Arbeit lag X3DOM noch in einer frühen *Pre-Alpha*-Version vor und zahlreiche Details der Implementierung blieben offen.

Architektur

Im darauffolgenden Jahr legten Behr et al. [2] in einer weiteren Arbeit die zuvor abstrakt dargestellten Ideen ihres Modells mittels der inzwischen gereiften Architektur des X3DOM-Frameworks konkret dar. Grundgedanke von X3DOM ist die Bereitstellung einer einheitlichen Schnittstelle zur deklarativen Programmierung von 3D-Szenen. Hierbei werden die direkt innerhalb des HTML-Dokuments eingebetteten X3D-Knoten wechselseitig mit dem X3D-System synchronisiert. Durch das Hinzufügen, Löschen und Abändern dieser DOM-Elemente kann der Anwendungs-Entwickler die 3D-Szene mittels einfacher JavaScript-Aufrufe dynamisch anpassen. Zusätzlich zu den im X3D-

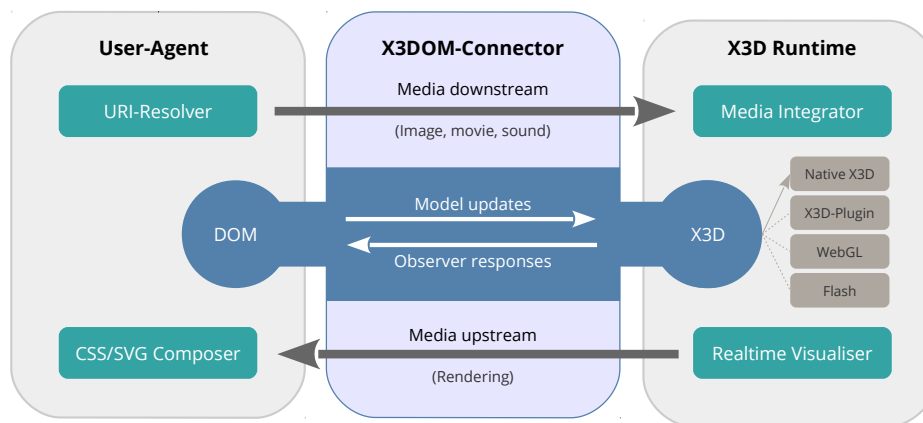


Abb. 5.1: Architektonischer Aufbau von X3DOM. Geringfügig adaptiert nach Behr et al. [2].

Standard spezifizierten Attributen fügt X3DOM vielen Knoten weitere Eigenschaften hinzu, die deren Benutzung weiter vereinfachen.

Die in Abbildung 5.1 gezeigte Architektur X3DOMs besteht im Wesentlichen aus drei Komponenten: Dem *User Agent*, dem X3DOM-Connector und dem X3D-Runtime.

Ersteres entspricht hier dem Webbrowser. Dieser enthält das Document Object Model und die hierin integrierten X3D-Knoten. Der Browser ermöglicht es weiterhin Multimedia-Dateien wie Grafiken, Video und Audio durch seinen *URI-Resolver* von einem Server herunterzuladen, um diese X3D bereit zu stellen.

Das X3D-Runtime-Objekt auf der andern Seite des Schemas ist für das eigentliche Rendering der Szene und die Verarbeitung von Benutzereingaben verantwortlich. Die Architektur erlaubt hierbei mehrere sogenannter *Backends*. Ein Backend stellt die Technologie dar, mit welcher die computergrafischen Berechnungen vollzogen werden. Mittels eines Fallback-Systems wird das Rendering-Backend anhand einer Priorisierung der Technologien ausgewählt. Hierbei wird zunächst überprüft, ob der Webbrowser X3D nativ darstellen kann oder ob alternativ ein X3D-Plugin vorhanden ist. Schlägt dies fehl, so wird das Rendering durch WebGL realisiert. Ist auch diese Technik nicht verfügbar, wird auf das proprietäre Flash-Plugin ausgewichen. Wenn dies ebenfalls scheitert, kann keine 3D-Darstellung erfolgen [2]. Die X3D-Runtime-Komponente beinhaltet weiterhin einen sogenannten *Media Integrator*, der die vom User Agent geladenen Medien wie beispielsweise Texturen verarbeitet.

Der *X3DOM Connector* fungiert schließlich als Mittelsmann der zwei obigen Komponenten. Er stellt die Verbindung zwischen den X3D-Knoten im DOM des HTML-Dokuments und dem X3D-System her und synchronisiert diese. Wird das DOM durch JavaScript manipuliert, so wird dies an das Runtime-Objekt propagiert, welches die Szene entsprechend neu rendert und das Bild durch den *Media Upstream* an das HTML-Dokument weiterleitet. Gleichzeitig reagiert das System durch ein *Observer*-Entwurfsmuster auf Benutzereingaben, welche durch JavaScript-Events wie *onClick* abgebildet werden.

5.2.3 Beispiel: Rotierende Pyramide

Anhand eines einfachen Beispiels wird im Folgenden der strukturelle Aufbau und einige grundlegenden Konzepte einer X3DOM-basierten Grafikdarstellung aufgezeigt, indem der Quelltext konkret Schritt für Schritt erläutert werden. Das Beispiel und sein Gegenstück in WebGL ist auf der beigefügten CD enthalten.

Die exemplarische Anwendung stellt eine Pyramide dar, welche um die Y-Achse rotiert. Durch eine Auswahlliste kann aus einer perspektivischen oder einer orthogonalen Parallelenprojektion gewählt werden. Abbildung 5.2 zeigt diese zwei verschiedenen Ansichten. Die Rotation der Figur kann zudem mittels eines Kontrollkästchens (*Checkbox*) jederzeit pausiert werden.

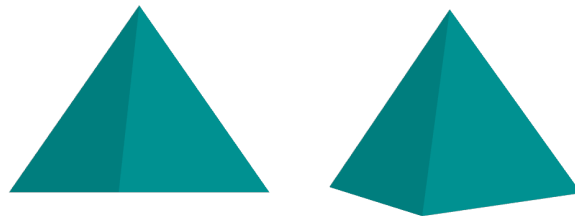


Abb. 5.2: Orthogonale und perspektivische Darstellung der Pyramide.

Eine große Stärke X3Ds im Hinblick auf die Nutzung im World Wide Web liegt in der großen Nähe zu HTML. Da beide Sprachen Anwendungen der Standard Generalized Markup Language (SGML) sind, ist ihr syntaktischer Aufbau sehr ähnlich. Schlüsselwörter in eckigen Klammern, sogenannte „*Tags*“, dienen der logischen und hierarchischen Strukturierung der Inhalte. Ein Knoten kann zusätzlich durch verschiedene Attribute parametrisiert werden.

```

1 <head>
2   <script src='x3dom.js'></script>
3 </head>
4 <body>
5   <h1>Konventioneller HTML-Code</h1>
6   <X3D width='800px' height='600px'>
7     <Scene>
8       <!-- Elemente des Szenengraphs -->
9     </Scene>
10   </X3D>
11 </body>

```

Listing 1: Einbettung des X3D-Szenengraphs in HTML

Nach dem Laden des X3DOM-Quelltexts im Kopfbereich eines HTML-Dokuments kann X3D direkt in dieses eingebunden werden. Listing 1 zeigt diese Einbettung mittels des speziellen X3D-Tags. Durch Angabe der *width*- und *height*-Attribute wird die benötigte Größe der Zeichenfläche spezifiziert. Dem X3D-Tag direkt untergeordnet liegt die Wurzel des Szenengraphen, welche sämtlichen weiteren Knoten der Anwendung enthält. Um die Lesbarkeit der folgenden Ausführungen zu gewährleisten, wurde der komplette Quelltext der Szene in Listing 2 auf der nächsten Seite abgedruckt.

Das kleine Programm kann im Wesentlichen in drei Schritte gegliedert werden:

1. Festlegen der Ansicht und des Navigationsmodus.
2. Erstellung und Transformation der Pyramiden-Geometrie.
3. Animation der Figur.

Zunächst wird die Ansicht der Szene mittels des *Viewpoint*-Elements in den Zeilen 21 bis 23 festgelegt. Der Darstellung können beliebig viele solcher Viewpoints hinzugefügt werden. Durch JavaScript können diese anschließend angesteuert und dynamisch aktiviert werden. Standardmäßig wird hierbei eine perspektivische Projektion verwendet. Für eine orthogonale Parallelenprojektion steht der *OrthoViewpoint* zur Verfügung. Das *NavigationInfo*-Element definiert daraufhin das Navigationsmodell der Szene, also die Art und Weise, wie die Darstellung durch den Benutzer interaktiv verändert werden kann. X3DOM bietet hierbei eine Vielzahl verschiedener vorgefertigter Modi [10]: Während die Kamera bei manchen Modelle frei bewegt werden kann, sind andere auf die nähere Betrachtung eines einzelnen Objekts zugeschnitten. Im Beispiel ist eine solche Navigation deaktiviert.

3D-Modelle werden in X3D mittels des *Shape*¹⁰-Knotens umgesetzt. Das Format bietet für zahlreiche geometrische Primitive wie Kugeln, Quader, Zylinder et cetera bereits

¹⁰Engl. für Form oder Gestalt.

```

19 <X3D width='800px' height='600px'>
20   <Scene>
21     <Viewpoint id='persp-vp' position='0 0 5'></Viewpoint>
22     <OrthoViewpoint id='ortho-vp' position='0 0 10'
23       fieldOfView='[-2, -2, 2, 2]'></OrthoViewpoint>
24     <NavigationInfo type='none'></NavigationInfo>
25
26     <Transform DEF='pyramid' translation='0 -0.75 0'>
27       <Shape>
28         <IndexedFaceSet coordIndex='
29           4 0 1 -1
30           4 1 2 -1
31           4 2 3 -1
32           4 3 0 -1
33           3 2 1 0 -1
34         '>
35         <Coordinate point='
36           1 0 -1
37           -1 0 -1
38           -1 0 1
39           1 0 1
40           0 2 0
41         '></Coordinate>
42       </IndexedFaceSet>
43       <Appearance>
44         <Material diffuseColor='0 0.66 0.66'></Material>
45       </Appearance>
46     </Shape>
47   </Transform>
48
49   <TimeSensor id='time' DEF='time' cycleInterval='4' loop='true'></TimeSensor>
50   <OrientationInterpolator DEF='move' key='0 1' keyValue='
51     0 1 0 0
52     0 1 0 3.14
53   '></OrientationInterpolator>
54
55   <Route fromNode='time' fromField='fraction_changed'
56     toNode='move' toField='set_fraction'></Route>
57
58   <Route fromNode='move' fromField='value_changed'
59     toNode='pyramid' toField='set_rotation'></Route>
60 </Scene>

```

Listing 2: Gesamter Szenengraph des Beispiels.

vordefinierte Elemente, die die Erstellung dieser Basisfiguren sehr einfach gestalten. Listing 3 zeigt dies exemplarisch durch Deklaration eines Würfels mittels des *Box*-Tags in Zeile 5. Dem Shape-Knoten ist weiterhin das *Appearance*-Element untergeordnet, welches wiederum einen *Material*-Knoten enthält (vgl. Listing 2, Zeilen 43 - 45). Dieser legt zahlreiche Attribute hinsichtlich der Beleuchtung des Objekts fest: etwa die Reflexionsintensität des ambienten Lichts, die Farbe diffusen Lichts, den Grad spekularer Glanzeffekte und so weiter fest.

```

1 <Shape>
2   <Appearance>
3     <Material diffuseColor='0 0.66 0.66'></Material>
4   </Appearance>
5   <Box></Box>
6 </Shape>

```

Listing 3: Erstellung einfacher Primitive durch den Shape-Knoten.

Sofern das Modell durch kein geometrisches Primitiv dargestellt werden kann, bietet X3D durch das *IndexedFaceSet*-Element einen Mechanismus, wie die topologische Struktur eines 3D-Modells durch explizite Angabe der Vertices und Seiten definiert werden kann. Unter der Topologie wird in diesem Zusammenhang die Nachbarschaftsbeziehung der Knotenpunkte, Kanten und Flächenstücke einer Geometrie verstanden. Der *IndexedFaceSet*-Knoten befindet sich unmittelbar unterhalb des Shape-Elements. Die Seiten der Figur werden dabei mittels des *CoordIndex*-Elements spezifiziert. Durch Angabe dreier oder mehr Indices, der im Kindknoten *Coordinate* gegebenen Vertices, wird ein entsprechendes Polygon erstellt (Zeile 28 ff.). Die Zahl -1 dient dabei als Begrenzungszeichen der verschiedenen Seiten. Zu beachten gilt zudem, dass die Indexierung der Vertices bei 0 beginnt. Der so kreierten Pyramide wird mittels des *diffuseColor*-Attributs des Material-Knotens die Farbe Türkis zugeordnet (Zeile 44). Farben werden in X3D durch das RGB-Farbmodell angegeben, wobei 1 der maximalen Intensität eines Farbkannels entspricht.

Die gesamte Geometrie wird durch ein Transformations-Element umhüllt. Dieses ermöglicht die affine Transformation untergeordneter Elemente. In diesem Fall wird die Pyramide in die negative Y-Richtung translatiert (Zeile 26). Das Attribut *DEF* dient der eindeutigen Bezeichnung des Knotens und wird im finalen Animations-Schritt benötigt.

Animationen werden in X3D durch sogenannte Schlüsselbilder (*Keyframes*) realisiert. Hierbei werden verschiedene diskrete Zustände der Animation durch die Keyframes definiert und die Bewegung anschließend durch lineare Interpolation zwischen diesen Werten erzielt. Innerhalb X3Ds sind hierfür mehrere Komponenten nötig: Zunächst

wird ein sogenannter *TimeSensor* definiert, welcher die Interpolation der Werte anstößt und die Länge der Animation durch das Attribut *cycleInterval* festlegt (Zeile 49). Die Schlüsselbilder werden innerhalb eines *Interpolator*-Knotens spezifiziert (Zeile 50 f.). Da die Rotation der Figur verändert werden soll, wird ein *OrientationInterpolator* benötigt. Um eine vollständige Drehung umzusetzen wird das Bogenmaß des Winkel im Intervall von 0 bis π interpoliert. Um diese zwei Komponenten nun zu verknüpfen, führt X3DOM das Konzept von *Routes* ein. Durch dieses Pipeline-artige Konstrukt kann ein automatischer Datenaustausch zwischen verschiedenen Knoten angestoßen werden, sofern eine bestimmte Bedingung zutrifft (vgl. Zeile 55 ff.). Der Zeitsensor im Beispiel initiiert in regelmäßigen Abständen eine zeitabhängige Neuberechnung des aktuellen Rotationswinkels im *OrientationInterpolator*. Sobald sich der Rotationswinkel entsprechend verändert hat, wird dies an den Transformationsknoten propagiert, welcher nun die Rotation der Pyramide anpasst.

Bei Betrachtung des gesamten Quelltexts des Beispiels in Listing 2 wird die Kompaktheit X3Ds deutlich. Bereits wenige XML-Knoten waren ausreichend, um die gewünschte rotierende Pyramide im Webbrowser darzustellen. Sofern eine manuelle Definition der Geometrie nicht notwendig gewesen wäre, wäre das Programm sogar noch deutlich kürzer ausgefallen. Dies unterstreicht die Vorteile des deklarativen Ansatzes von X3D.

Die eingangs zitierte Nennung X3Ds in der HTML5-Spezifikation ist inzwischen nicht mehr in der aktuellen Revision des Arbeitsentwurfs zu finden. Somit muss X3D nicht mehr zwingend als der offizielle, durch das W3C angestrebte Standard für Web3D betrachtet werden. Mit der Web Graphics Library ist eine starke Alternative für einen solchen offiziell unterstützen Grafikstandards entstanden. Im Folgenden soll dieser näher beleuchtet werden.

5.3 Web Graphics Library

Der Ursprung der Web Graphics Library liegt in der Arbeit des inzwischen bei Mozilla angestellten Software-Entwicklers Vladimir Vukićević. Dieser präsentierte 2006 einen ersten experimentellen Prototypen einer Schnittstelle des HTML5-Canvas-Elements¹¹ zur Grafikbibliothek OpenGL [51]. Dieser neue Ansatz sollte die native Darstellung von 3D-Grafik im Webbrowser ermöglichen. Hierin liegt der fundamentale Unterschied und Vorteil WebGLs gegenüber allen vorherigen Web3D-Ansätzen: Durch die direkt in den

¹¹Canvas: engl. für „Leinwand“.

Browser integrierte Schnittstelle zu OpenGL sind Plugins für die Berechnung von 3D-Grafik nicht länger notwendig.

Sowohl Firefox als auch Opera realisierten ein Jahr nach Vukićevićs Vorstoß frühe, eigenständige Implementierungen dieses Ansatzes innerhalb ihrer Browser [51] [43]. Während Mozillas Umsetzung von Canvas-3D eine nahezu direktes *Mapping* von JavaScript zu OpenGL darstellte, war Operas Ansatz etwas abstrahierter, um eine bessere Plattformerunterstützung zu erzielen.

Zur Standardisierung dieses neuen Vorstoßes für 3D-Grafik im Web bildete sich 2009 daraufhin die WebGL-Arbeitsgruppe innerhalb der *Khronos Group*. Zwei Jahre später wurde Version 1.0 der Spezifikation auf Basis von *OpenGL ES 2.0* im Februar 2011 fertig gestellt [17]. Die Khronos Group ist ein seit 2000 bestehendes internationales Industriekonsortium, welches die Entwicklung zahlreicher lizenzfreier und offener Standards im Multimedia-Bereich vorantreibt [36]. Zahlreiche namhafte Organisationen und IT-Unternehmen wie Apple, Google, Mozilla und Opera beteiligen sich seither aktiv bei der Entwicklung von WebGL [37].

5.3.1 Grundarchitektur

Um die grundlegende Architektur einer WebGL basierten Webanwendung zu verstehen, werden zu Anfang die beteiligten Technologien und deren Zusammenspiel erörtert. Die schematische Abbildung 5.3 veranschaulicht die Beziehung der einzelnen Komponenten untereinander:

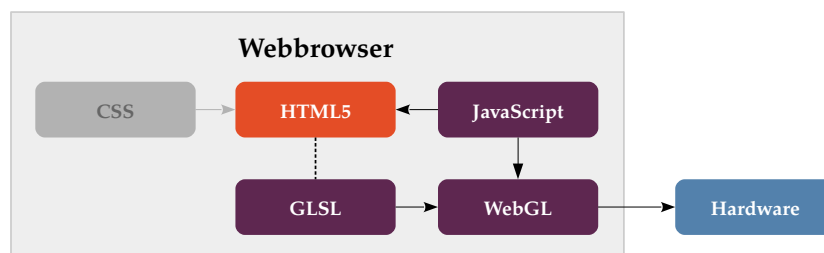


Abb. 5.3: Grundarchitektur einer WebGL-basierten Web3D-Anwendung.

Zur Darstellung der 3D-Grafik ist zunächst ein konventionelles HTML-Dokument nötig. Wichtig ist hierbei eine Deklaration des Dokumenttyps als HTML5, da WebGL das

erst mit HTML5 eingeführte Canvas-Element benötigt. Obgleich dieser neue Standard derzeit noch nicht den höchsten Reifegrad einer *W3C Recommendation* erlangt hat, ist dessen Entwicklung weit vorangeschritten. So wurde bereits im Mai 2011 der „*Last Call*“ ausgerufen, also die Aufforderung, letzte Änderungsvorschläge für den Entwurf einzureichen [31]. Weiterhin wurde in selbiger Ankündigung das Jahr 2014 als Zeitpunkt für den Statuswechsel der Spezifikation zu einer W3C-Empfehlung anvisiert.

Das Canvas-Element stellt eine rechteckige Zeichenfläche dar, welche durch JavaScript angesteuert wird und der dynamischen Generierung von Grafiken dient [6]. Das Element verfügt über mehrere sogenannte *Kontexte*. Ein Kontext stellt die Schnittstelle zu einer Grafik-Implementierung dar und erlaubt es, JavaScript auf der Leinwand zu zeichnen. Während der Kontext mit der Bezeichnung 2d zweidimensionale Grafiken ermöglicht, kann das Element ebenso als Projektionsfläche dreidimensionaler Grafik dienen. In diesem Fall wird der *WebGLRenderingContext* durch Angabe des Schlüsselworts *webgl* verwendet. Ein Kontext mit dem Namen 3d existiert nicht und liefert kein Ergebnis.

Wie Listing 4 zeigt, besitzt ein HTML-Dokument für die Darstellung von WebGL in seiner minimalen Form einen sehr kompakten Aufbau. Dies verdeutlicht bereits einen paradigmatischen Unterschied zu X3DOM: Die Elemente der 3D-Szene sind nicht Teil von HTML, sondern dieses dient nur der letztendlichen Anzeige der berechneten 3D-Grafik durch das Canvas-Element. Dieses ist mit einem eindeutigen Bezeichner (*ID*) versehen, um es in JavaScript einfach ansprechen zu können (Zeile 8). Weiter sind die Größenangaben der Zeichenfläche für die Initialisierung von WebGL wichtig. In den Zeilen 9 und 10 wird der Quelltext der Grafikshader innerhalb von Script-Tags eingebunden. Um die Übersichtlichkeit zu wahren, wird deren Inhalt jedoch zunächst ausgelassen. Die Bedeutung und Funktion der Shader wird in Abschnitt 5.3.2 erörtert. Zuletzt wird der ausgelagerte, selbst implementierte WebGL-Code in Zeile 11 geladen (*main.js*).

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset='utf-8'>
5     <title>Minimales HTML-Dokument</title>
6   </head>
7   <body>
8     <canvas id='glcanvas' width='800' height='600'></canvas>
9     <script id='shader-fs' type='x-shader/x-fragment'></script>
10    <script id='shader-vs' type='x-shader/x-vertex'></script>
11    <script src='main.js'></script>
12  </body>
13 </html>
```

Listing 4: Aufbau eines minimalen HTML-Dokuments für WebGL.

Der nächste Baustein der Architektur JavaScript stellt die zentrale Komponenten des Systems dar. Mittels Abfrage des WebGL-Kontexts durch das Canvas-Elements in HTML wird die Schnittstelle zu WebGL und damit der Grafik-Hardware aufgebaut. Das *Application Programming Interface* (API) der Grafikkbibliothek ist durch die Funktionen des Kontext-Objekts abgebildet und ermöglicht so die eigentliche Grafikprogrammierung. In Abschnitt 5.3.4 werden die konkreten Einzelschritte, die nötig sind, um eine 3D-Darstellung mit WebGL zu erzielen, anhand des Pyramiden-Beispiels erläutert.

Die dritte Komponente der Architektur stellen schließlich die eingangs erwähnten Grafikshader dar. Diese werden im JavaScript-Code eingelesen und innerhalb des WebGL-Systems weiterverarbeitet. Wie die gestrichelte Linie innerhalb des Schemas in Abbildung 5.3 andeutet, ist der C-ähnliche Shader-Code oftmals innerhalb des HTML-Dokuments eingebettet (vgl. Listing 4). Grundsätzlich kann dieser Quelltext in einer beliebigen Form als String abgespeichert werden. Er könnte also beispielsweise ebenso aus einer Datenbank stammen. Auch eine direkte Einbettung innerhalb von JavaScripts ist möglich, aber aufgrund der fehlenden Unterstützung der Sprache für mehrzeilige Strings mühsam.

Das letzte Element *Cascading Stylesheets* (CSS) sind keine Notwendigkeit für WebGL per se. Da sie jedoch den De-Facto-Standard für die Gestaltung von Websites und damit auch von Benutzeroberflächen von Webanwendungen darstellen, ist diese Technologie der Vollständigkeit halber ebenso innerhalb des Schemas aufgeführt.

5.3.2 Bedeutung und Funktion der Shader

Shader sind elementarer Bestandteil heutiger Grafikpipelines und die Ursache derer enormen Flexibilität [12, S. 722 f.]. Es handelt sich dabei um kleine Programme, die verschiedenste Effekte beim Rendern von Computergrafik erzielen und direkt innerhalb der *Graphics Processing Unit* auf der Grafikkarte ausgeführt werden. Sie werden in einer eigenen sogenannten Shader-Sprache programmiert. In WebGL wird dabei aufgrund dessen Ursprung auf die *OpenGL ES Shading Language* (GLSL ES) zurückgegriffen.

„At the heart of the shading calculations is the simulation of the way light interacts with objects.“ – Cook [8]

WebGL kennt zwei Arten von Shadern [18]: Den Vertex- und den Fragment-Shader. Ersterer berechnet die letztendliche Position der Vertices im Raum und reicht Vertex-Attribute wie deren Farbe an den Fragment-Shader weiter. Dieser wird innerhalb der Rasterisierungs-Phase der Rendering-Pipeline zu einem späteren Zeitpunkt angewandt. Der Fragment-Shader dient der Berechnung der Farben der einzelnen Rasterpunkte. Hierfür wird der Einfluss der Lichtquellen auf die Fragmente anhand der verschiedenen Beleuchtungsmodelle und Schattierungsverfahren bestimmt. Ein Fragment beschreibt ein Pixel zusammen mit weiteren Informationen wie der zugehörige Farbe, der Z-Koordinate des Tiefenpuffers und dem Alpha-Wert.

5.3.3 Mathematische Bibliotheken

Bei computergrafischen Berechnungen sind mathematische Vektoren- und Matrizenoperationen aufgrund ihres sehr häufigen Vorkommens von elementarer Bedeutung. Da weder JavaScript innerhalb seiner schmalen Mathematik-Standardbibliothek noch WebGL, im Gegensatz zu OpenGL ES, eine solche Funktionalität bieten, muss diese durch den Programmierer selbst implementiert werden.

Um diesen immer wiederkehrenden Anforderungen gerecht zu werden, haben sich im WebGL-Umfeld eine Vielzahl von performanten JavaScript-Bibliotheken entwickelt, die diese mathematischen Operationen und Computergrafik-spezifischen Berechnungsroutinen bereitstellen. In den nachfolgenden Ausführungen wird die populäre Bibliothek *glMatrix* [56] des Softwareentwicklers Brandon Jones von Google verwendet, um diese Basisfunktionalität zu realisieren.

5.3.4 Beispiel: Rotierende Pyramide

Analog zu dem in Abschnitt 5.2.3 gezeigten X3DOM-Beispiel soll im Folgenden die Schritte dargelegt werden, die nötig sind, um die rotierende Pyramide mit WebGL darzustellen. Hierdurch werden die unterschiedlichen paradigmatischen Vorgehensweisen der zwei Ansätze deutlich.

Zusätzlich zu den vorherigen Einstellungsmöglichkeiten bezüglich Projektionsart und Animation kann die Ansicht nun wie in Abbildung 5.4 dargestellt zwischen dem Drahtgittermodell der Figur und gefüllten Flächen umgeschaltet werden.

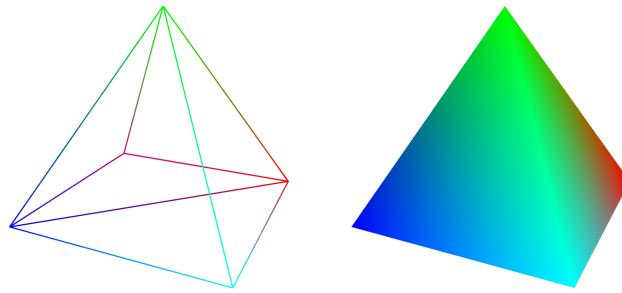


Abb. 5.4: Verschiedene Ansichten der Beispiel-Anwendung.

Der Programmablauf kann wie folgt skizziert werden:

1. Initialisierung des WebGL-Kontexts und Setzen grundlegender Einstellungen.
2. Erstellung des Fragment- und Vertex-Shaders.
3. Erstellung der *Buffer Objects* und Schreiben der Vertex-Attribute.
4. Optional: Animation der geometrischen Figur.
5. Berechnung und Darstellung des Einzelbilds (Rendering).

Als Einstiegspunkt der Anwendung dient die `main`-Funktion, welche obigen Ablauf durch mehrere Funktionsaufrufe abbildet. Der Sinn und Zweck der verschiedenen Funktionen wird in den folgenden Abschnitten erklärt.

```

15 function main() {
16     var glOptions = {antialias: true};
17     try {
18         canvas = document.getElementById('gl-canvas');
19         if (canvas && Boolean(window.WebGLRenderingContext)) {
20             gl = canvas.getContext('webgl', glOptions) ||
21                 canvas.getContext('experimental-webgl', glOptions);
22         }
23     } catch (e) {
24         console.error('Your webbrowser does not seem to support WebGL.')
25     }
26
27     if (gl) {
28         initGl();
29         initShaders();
30         initBuffers();
31         tick();
32     } else {
33         console.error('WebGL could not be initialized.');
```

Listing 5: Main-Methode.

Initialisierung des WebGL-Kontexts

In den Zeilen 16 bis 25 (vgl. Listing 5) wird zunächst der WebGL-Kontext mittels des Canvas-DOM-Elements abgefragt und in der Variable `gl` gespeichert. In manchen Browser-Implementierungen von WebGL trägt der Kontext noch die Bezeichnung `experimental-webgl`, welche auf die noch experimentelle Unterstützung der Technologie hinweist. Um eine möglichst breite Browserunterstützung zu erzielen, wird daher auch auf diesen Bezeichner hin geprüft. Die Option `antialias` aktiviert das Anti-Aliasing und sorgt so für eine glattere Kantendarstellung.

Sofern die Schnittstelle erfolgreich initialisiert werden konnte, werden die weiteren Funktionen `initGl`, `initShaders`, `initBuffers`, und `tick` aufgerufen. Die `initGl`-Routine (vgl. Listing 6) legt einige grundsätzliche Einstellungen fest: Als erstes wird der Tiefenpuffer aktiviert (*Z-Buffering*). Hierdurch werden Gegenstände, die sich auf der Z-Achse hinter einem weiteren Objekt befinden, im Rasterisierungsschritt verdeckt, so wie es der natürlichen Wahrnehmung entspricht. Weiterhin wird die Breite der zu zeichnenden Linien und die Hintergrundfarbe der Projektionsfläche festgelegt. Der Funktionsaufruf `gl.clear` in Zeile 50 bewirkt das Zurücksetzen des Farb- und Tiefenpuffers und führt so zu einer komplett weißen, leeren Zeichenfläche.

```
47 gl.enable(gl.DEPTH_TEST);
48 gl.lineWidth(2.0);
49 gl.clearColor(1.0, 1.0, 1.0, 1.0);
50 gl.clear(gl.COLOR_BUFFER_BIT|gl.DEPTH_BUFFER_BIT);
51 vpMatrix = calcViewProjMatrix();
```

Listing 6: Initialisierung des WebGL-Kontexts.

```
62 function calcViewProjMatrix() {
63     var viewMatrix = mat4.create();
64     var projMatrix = mat4.create();
65     mat4.lookAt(viewMatrix, [0, 0.5, -2], [0, -0.25, 0], [0, 1, 0]);
66
67     if (cfg.perspective) {
68         var aspectRatio = canvas.width / canvas.height;
69         mat4.perspective(projMatrix, degToRad(45), aspectRatio, 0.5, 10);
70     } else {
71         mat4.ortho(projMatrix, -1, 1, -1, 1, 0.5, 3.5);
72     }
73
74     return mat4.multiply(vpMatrix, projMatrix, viewMatrix);
75 }
```

Listing 7: Berechnung der Sicht-Projektions-Matrix.

Zur Berechnung der Sicht- und Projektionsmatrix wird die Hilfsfunktion `calcViewProjMatrix` aufgerufen (vgl. Listing 7). Innerhalb dieser Methode wird zunächst die Sicht-

matrix durch Aufruf der `mat4.lookAt`-Funktion der *glmMatrix*-Bibliothek erstellt. Hierbei fließen Augpunkt, das Zentrum der Beobachtung (*Look-at-Punkt*) und Oben-Vektor ein. Je nach aktueller Einstellung wird anschließend entweder eine orthogonale Projektionsmatrix durch Angabe der Ausdehnung des Sichtvolumens erstellt oder die Projektionsmatrix für eine perspektivische Darstellung erzeugt. In letzterem Fall wird ebenso die Abmessung des *Frustums* und die Distanz der *Front*- und *Backplane* als Parameter benötigt.

Die Berechnung dieses Matrizenprodukts dient der Einsparung von Rechenzeit. Während sich die Modellmatrix durch die Animation kontinuierlich ändert, bleibt die Sicht-Projektions-Matrix größtenteils gleich.

Initialisierung der Shader

Im nächsten Schritt werden die im HTML-Dokument eingebetten Shader durch die Hilfsmethode `getShader` eingelesen und verarbeitet. Hierbei wird je nach Typ ein entsprechendes Shader-Objekt erstellt, der Quelltext aus den Script-Tags eingefügt und dieser schließlich kompiliert. Listing 8 zeigt den relevanten Abschnitt der Funktion.

```
110  switch (shaderScript.type) {
111      case 'x-shader/x-fragment':
112          shader = gl.createShader(gl.FRAGMENT_SHADER);
113          break;
114      case 'x-shader/x-vertex':
115          shader = gl.createShader(gl.VERTEX_SHADER);
116          break;
117  }
118
119  gl.shaderSource(shader, shaderScript.textContent);
120  gl.compileShader(shader);
```

Listing 8: Auslesen der Shader-Quelltexte.

Anschließend wird das Shader-Programm-Objekt erstellt und der kompilierte Shader-Code angehängt. Durch Aufruf der `linkProgram`-Methode wird das Programm zu einer für die GPU verarbeitbaren Binärdatei gebunden (*Linking*). Sofern hierbei kein Fehler aufgetreten ist, wird WebGL angewiesen, die so gepackten Shader beim Rendering-Vorgang für die Berechnung der Darstellung zu verwenden (vgl. Listing 9, Zeile 92). Zuletzt wird die Speicherposition der Uniform-Variable `uMVPMatrix` innerhalb des Vertex-Shaders abgefragt. Dies ist notwendig, um deren Wert während der späteren Animation aktualisieren zu können. Eine Uniform-Variable zeichnet sich dadurch aus, dass sie sowohl im Vertex- als auch im Fragment-Shader zur Verfügung steht [18].

```

80 function initShaders() {
81     var fragmentShader = getShader(gl, 'shader-fs');
82     var vertexShader = getShader(gl, 'shader-vs');
83
84     gl.program = gl.createProgram();
85     gl.attachShader(gl.program, vertexShader);
86     gl.attachShader(gl.program, fragmentShader);
87     gl.linkProgram(gl.program);
88
89     if (!gl.getProgramParameter(gl.program, gl.LINK_STATUS)) {
90         console.error('Unable to initialize the shader program.');
```

Listing 9: Erstellung und Binden des Shader-Programms.

Erstellung und Schreiben der Daten-Puffer

Um den Shadern innerhalb von WebGL Daten zu übergeben, werden sogenannte *Vertex Buffer Objects* (VBO) verwendet. Ein solches Buffer Object stellt einen Speicherbereich innerhalb der GPU dar, welcher verschiedene Vertex-Attribute aufnimmt. In der Beispiel-Anwendung beschreiben drei verschiedene solcher Puffer die Geometrie und Farben der Pyramide. Bei der Definition dieser Rohdaten kommen sogenannte *TypedArrays* zum Einsatz. Diese speziellen Arrays bieten aufgrund ihrer starken Typenbindung bei großen Datenmengen bessere Performance gegenüber konventiellen JavaScript-Feldern, da der Zugriff entsprechend optimiert werden kann [39].

```

134 var vertices = new Float32Array([
135     0.5,  -0.5, -0.5, // P0
136     -0.5, -0.5, -0.5, // P1
137     -0.5, -0.5,  0.5, // P2
138     0.5,  -0.5,  0.5, // P3
139     0.0,   0.4,  0.0 // P4
140 ]);
141
142 var indices = new Uint8Array([
143     4, 0, 1, // Back
144     4, 1, 2, // Left
145     4, 2, 3, // Back
146     4, 3, 0, // Front
147     0, 1, 2, // Bottom
148     0, 2, 3 // Bottom
149 ]);
```

Listing 10: Definition der Vertices und Seiten der Pyramide.

Im ersten Puffer werden die fünf Eckpunkte (*Vertices*) der Figur durch ihre kartesischen Koordinaten im Raum angegeben (vgl. Listing 10). Der zweite Puffer legt die Seiten der

Figur fest. Hierfür werden je drei der zuvor definierten Punkte anhand ihrer Indices zu einem Dreieck verbunden. Zu beachten ist, dass die Nummerierung der Indices bei 0 beginnt. Die Reihenfolge der Punkte ist aufgrund des *Backface Culling* wichtig. Dieses Verfahren dient der Verbesserung der Darstellungseffizienz, indem nur die durch den Normalenvektor definierte Vorderseite von Dreiecken gezeichnet wird. Das letzte hier nicht gezeigte Array spezifiziert schließlich die Farben der Pyramide, indem die RGB-Werte den fünf Punkten ihrer Reihenfolge entsprechend zugeordnet werden. Die Farbkanäle werden in WebGL analog zu X3D im Zahlenintervall $[0, 1]$ angegeben.

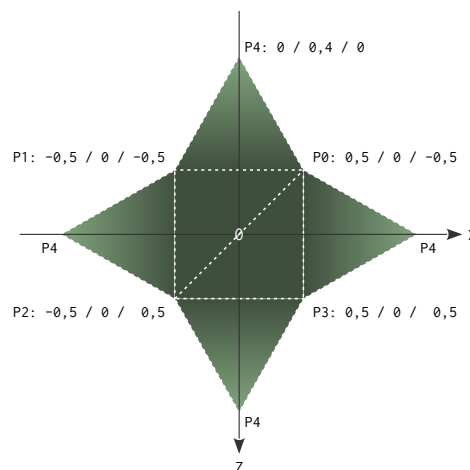


Abb. 5.5: Topologische Struktur der Pyramide.

Abbildung 5.5 veranschaulicht den topologischen Aufbau der Pyramide, indem deren Geometrie innerhalb der XZ-Ebene aufgespannt wurde. Dies erleichtert die Nachvollziehbarkeit der obigen Indices-Zuweisung.

```

168 buffer = gl.createBuffer();
169 gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
170 gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
171
172 FSIZE = vertices.BYTES_PER_ELEMENT;
173 attribute = gl.getAttribLocation(gl.program, 'aPosition');
174 gl.vertexAttribPointer(attribute, 3, gl.FLOAT, false, FSIZE * 3, 0);
175 gl.enableVertexAttribArray(attribute);

```

Listing 11: Schreiben der Puffer-Daten

Für das Schreiben der Puffer in den Speicher der GPU sind mehrere Schritte notwendig: Zunächst muss ein Buffer-Objekt durch die WebGL-API erstellt werden. Anschließend wird dieses durch die `bindBuffer`-Methode als derzeitiger Arbeitspuffer ausgewählt.

Die eigentlichen Daten – hier das Array mit den Vertices – können dann durch Aufruf von `bufferData` geschrieben werden (Zeile 168 ff.). Im nächsten Schritt wird die Speicherposition der Shader-Variable `aPosition` abgefragt, um diese mit dem Daten-Puffer zu verknüpfen (Zeile 173). Dies wird durch die Funktion `vertexAttribPointer` realisiert. Dabei gibt deren fünfter Parameter an wie viele Bytes zwischen Beginn und Ende der Definition einer Seite der Pyramide im Vertex-Buffer liegen. Da je drei Eckpunkte eine Seite definieren, wird dieser Wert mit der Länge eines Einzelements (`FSIZE`) multipliziert. Mit dem Aufruf von `enableVertexAttribArray` wird das Vertex-Attribut schließlich aktiviert. Nachdem der Farb- und Indexpuffer auf die selbe Weise an die Grafikhardware gereicht wurden, kann das eigentliche Rendering der Figur beginnen.

Animation der geometrischen Figur

Die Funktion `tick`, welche initial in der `main`-Routine aufgerufen wurde, ist für die Animation und Berechnung des Einzelbilds verantwortlich. Sofern die Bewegung der Pyramide mittels des Kontrollkästchens aktiviert ist, wird die Animation gestartet (vgl. Listing 12, Zeile 191). Im Anschluss wird die Rendering-Funktion aufgerufen und die erneute Ausführung von `tick` durch die `requestAnimationFrame`-Methode angefordert (Zeile 195 f.). `requestAnimationFrame` stellt einen vom Browser bereitgestellte Mechanismus zur Optimierung von JavaScript-Animationen dar [27]. Der nächste Animationsschritt wird hierbei vom Webbrowser unter Berücksichtigung der aktuellen CPU-Last eingeplant, anstatt diesen nach einem fixen Zeitinkrement auszuführen. Hierdurch wird eine flüssigere Animation erzielt. Ein weiterer Vorteil ist die Einsparung von Rechenzeit und Akkuverbrauch, da die Animationsschleife nur dann ausgeführt wird, wenn der entsprechende Browser-Tab aktiv ist. Dies ist insbesondere bei Mobilgeräten ein wichtiger Aspekt.

```
189 function tick() {
190     if (cfg.animation) {
191         animate();
192     } else {
193         lastTick = Date.now(); // Save rotation state
194     }
195     render();
196     requestAnimationFrame(tick);
197 }
```

Listing 12: Einzeliteration der Animation.

Die eigentliche Animation der Rotation um die Y-Achse erfolgt durch die Funktion `animate`. Da das Zeitintervall zwischen zwei Aufrufen der Animations-Methode vari-

ieren kann, kann eine ungleichmäßige Bewegung entstehen, sofern stets das gleiche Inkrement zum Rotationswinkel hinzuaddiert wird. Unter Berücksichtigung des Zeitintervalls zwischen zwei Iterationen kann dieses Defizit leicht behoben werden, indem der ermittelte Delta-Wert in die Berechnung dieses Inkrements einfließt (vgl. Listing 13, Zeile 215). Der Modulo-Operator sichert den Wertebereich von $[0, 360]$ Grad.

```

202 var now, lastTick, delta;
203 function animate() {
204     if (typeof lastTick === 'undefined')
205         lastTick = Date.now();
206     now = Date.now();
207     delta = now - lastTick;
208     lastTick = now;
209
210     rotAngle += (0.03 * delta) % 360;
211     mat4.rotateY(modelMatrix, mat4.create(), degToRad(rotAngle));
212 }

```

Listing 13: Animation der Rotation.

Da die Grundfläche der Pyramide genau mittig innerhalb der XZ-Ebene liegt und deren Normale auf die Spitze so genau mit der Y-Achse zusammenfällt, ist die Berechnung der Rotation hier sehr einfach. Die Modellmatrix muss lediglich auf die Rotationsmatrix um die Y-Achse für diesen Winkel gesetzt werden. Dies wird durch Aufruf der `mat4.rotateY`-Funktion aus der *glmMatrix*-Bibliothek in Zeile 211 erzielt.

Rendering des Einzelbilds

Schließlich kann der Rendering-Prozess der virtuellen Pyramidendarstellung angestoßen werden. Hierfür wird zunächst die MVP-Matrix durch Multiplikation der aktuellen Modellmatrix mit der Sicht-Projektionsmatrix berechnet (vgl. Listing 14 Zeile 219). Anschließend wird das Ergebnis mittels Aufruf der `uniformMatrix4fv`-Methode an den Vertex-Shader übergeben.

```

218 function render() {
219     mat4.multiply(mvpMatrix, vpMatrix, modelMatrix);
220     gl.uniformMatrix4fv(mvpMatrixUniform, false, mvpMatrix);
221
222     gl.clear(gl.COLOR_BUFFER_BIT|gl.DEPTH_BUFFER_BIT);
223     var drawMethode = (cfg.wireframe) ? gl.LINE_STRIP : gl.TRIANGLES;
224     gl.drawElements(drawMethode, numOfTris, gl.UNSIGNED_BYTE, 0);
225 }

```

Listing 14: Rendering der Darstellung.

```

1 void main(void) {
2     gl_Position = uMVPMatrix * a_Position;
3     v_Color = a_Color;
4 }

```

Listing 15: Hauptfunktion des Vertex-Shaders.

Der Vertex-Shader, gezeigt in Listing 15, ist äußerst simpel gehalten und entspricht dem gezeigten Verfahren bei Betrachtung der Standard-Grafikpipeline in Abschnitt 3.2. Die endgültige Position jedes Vertex wird durch Multiplikation der Koordinaten mit der MVP-Matrix realisiert (vgl. Listing 15, Zeile 3). Der Fragment-Shader ist ebenfalls einfach gehalten. Den einzelnen Fragmenten wird lediglich die zuvor spezifizierte Farbe zugewiesen. Durch die standardmäßige Interpolation der Farben der Eckpunkte bei OpenGL ergeben sich die Farbverläufe auf den Seiten der Figur.

Nachdem der Farb- und der Tiefenpuffer geleert wurde, kann das Bild schließlich mit der gewählten Methode gezeichnet und in den Framebuffer geschrieben werden (vgl. Listing 14, Zeile 222 ff.).

5.3.5 Three.js als Vertreter eines WebGL-Frameworks

Wie das vorherige Beispiel zeigt, ist aufgrund des niedrigen Abstraktionsniveaus der WebGL-API im Vergleich zu X3D ein sehr viel höherer Aufwand notwendig, um selbst einfachste dreidimensionale Szenen darzustellen. Da sich jedoch viele Teile des Programmes wie beispielsweise die Abfrage des WebGL-Kontexts oder die Initialisierung der Shader bei jeder WebGL-basierten Anwendung wiederholen, ist eine Generalisierung dieser immer wieder benötigten Funktionalität auf höherer Ebene naheliegend.

Innerhalb der letzten Jahre entstanden so zahlreiche, zum Teil sehr umfangreiche und ausgereifte Frameworks, welche die Umsetzung grafisch anspruchsvoller 3D-Anwendungen erheblich erleichtern. So können Basiselemente wie Geometrien, Texturen, Lichtquellen, Kameras et cetera mittels weniger Funktionsaufrufe komfortabel erstellt und einem Szenengraph hinzugefügt werden.

Eine Methode zur groben Beurteilung der Popularität von Software stellt die Anzahl von Favorisierungen (Sterne) auf dem Portal *GitHub* dar [11]. GitHub ist ein Hosting-Dienst für Repositories der Versionsverwaltung Git, welcher insbesondere bei Open-Source-Projekten beliebt ist. Das Framework *Three.js* des spanischen Webentwicklers Ricardo

Cabello¹² liegt hierbei mit einem Wert von 15.558 Sternen weit vor dem hinsichtlich dieses Kriteriums zweitpopulärsten Systems *PhiloGL*, welches mit 567 Stimmen lediglich einen Bruchteil dieses Werts verbuchen kann [57] [54]. Zusätzlich zu der generellen Vereinfachung bei der Erstellung einer 3D-Szene bietet Three.js ähnlich zu X3DOM verschiedene Rendering-Backends, die als Fallback-Lösungen dienen können. Neben WebGL kann die Darstellung auch innerhalb eines 2D-Canvas, durch SVG oder durch CSS3-Transformationen erfolgen. Aufgrund der deutlich schlechteren Performance dieser Techniken und der Fokussierung dieser Arbeit auf WebGL und X3D, werden diese Technologien jedoch nicht näher betrachtet.

Sämtliche weiteren Untersuchungen von WebGL innerhalb des Evaluationsteils werden aufgrund der genannten Vorteile eines abstrahierten WebGL-Frameworks auf Basis von Three.js realisiert.

¹²Besser bekannt als *Mr.doob*.

6 Evaluation der 3D-Technologien

Nachdem das Grundprinzip und die Funktionsweise von WebGL und X3DOM im vorherigen Kapitel ausführlich dargelegt wurden, werden diese im Folgenden anhand der in Kapitel 4 spezifizierten Kriterien evaluiert. Hierfür wird zunächst kurz die Testumgebung beschrieben, welche für die Erprobung der Technologien implementiert wurde, und die verwendete Methodik erläutert. Schließlich wird die eigentliche Evaluation mittels verschiedener Tests durchgeführt und die Ergebnisse im Bezug auf die Zielsetzung bewertet.

6.1 Architektur der Testumgebung

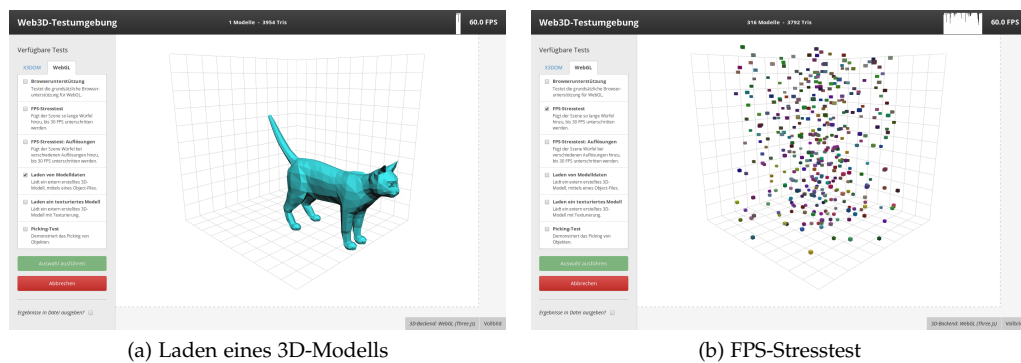


Abb. 6.1: Verschiedene Tests innerhalb der Testumgebung.

Zur automatisierten Erprobung von X3DOM und WebGL wurde eine webbasierte Testumgebung umgesetzt, welche die sequentielle Ausführung verschiedener Tests ermöglicht. Die Tests sind auf die spezifizierten Anforderungen zugeschnitten und erproben unterschiedliche Funktionalitäten des Webbrowsers beziehungsweise des Grafiksystems. Abbildung 6.2 zeigt den architektonischen Grundaufbau der Umgebung, die im Folgenden kurz erläutert werden soll.

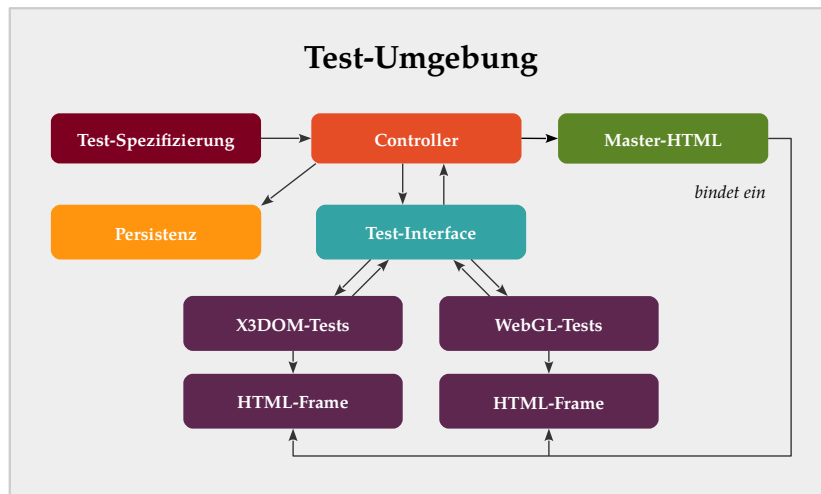


Abb. 6.2: Architektur der Testumgebung.

Als zentrales Bindeglied der Applikation koordiniert der *Controller* jederzeit den Programmablauf und vermittelt zwischen den einzelnen Komponenten. Zu Beginn des Ablaufs wird die Test-Spezifikation eingelesen. Diese liegt in Form eines *JSON*-Objekts vor und hat mehrere Funktionen: Zum einen enthält sie die Metadaten der Tests wie deren textuelle Beschreibung, und zum anderen werden die verschiedenen Testfälle mit einer zugehörigen Funktion verknüpft. Weiterhin werden hier die variierenden Parameter für den späteren Aufruf dieser Methode definiert.

Zur Darstellung der aktuell betrachteten 3D-Technologie dient ein sogenannter *iFrame*. Dieses *HTML*-Element ermöglicht das Einbetten eines externen *HTML*-Dokuments innerhalb eines rechteckigen Rahmens auf einer Webseite. Die so geladene *HTML*-Seite besitzt ein eigenes *Document Object Model* und operiert größtenteils unabhängig zum Eltern-Dokument. Diese Abschottung hat den Vorteil, dass die 3D-Darstellung innerhalb des *iFrames* in seiner Ausführung durch die Benutzeroberfläche der Testumgebung nicht gestört wird. Darüber hinaus kann das angezeigte *HTML*-Dokument jederzeit dynamisch durch den *Controller* mit einem anderen ausgetauscht werden. Dies erleichtert das Testen sowohl von X3D als auch von WebGL in jeweils eigenen Dateien.

Auch die Ausführung der Tests wird vom *Controller* überwacht. Dieser baut nach der Auswahl der auszuführenden Tests durch den Benutzer eine entsprechende Warteschlange auf und führt die Tests der Reihe nach aus. Diese sind gegen ein abstrahiertes

Test-Interface programmiert und somit unabhängig von ihrer konkreten Implementierung. Für jede der Technologien existiert ein klassenähnliches JavaScript-Objekt, das alle abstrakten Methoden des Test-Interface auf das entsprechende 3D-Framework zugeschnitten, realisiert. Auch das eigentliche Rendern der Szene wird hier umgesetzt.

Um die Messergebnisse der Tests schließlich dauerhaft zu speichern, interagiert der Controller nach Abschluss aller Tests mit der Persistenz-Komponente. Diese dient der Speicherung der volatilen JavaScript-Arrays in einer JSON-Datei, welche daraufhin heruntergeladen werden kann.

6.2 Methodik und Testdurchführung

6.2.1 Browserunterstützung und Plattformunabhängigkeit

Durch das in Abschnitt 5.2.2 beschriebene Fallback-System X3DOMs ist ein Test der Browserunterstützung für diese Technologie sehr einfach zu realisieren. Mit dem sogenannten Runtime-Objekt stellt X3DOM eine Schnittstelle zur dynamischen Ansteuerung des Systems bereit. Die Funktion `backendName` der API liefert den Namen des verwendeten Backends. Falls das Runtime-Objekt nicht existiert, kann X3DOM nicht initialisiert werden, weil keinerlei 3D-Backend zur Verfügung steht. In diesem Fall ist eine 3D-Darstellung durch X3DOM nicht möglich.

```
146 if (x3d.runtime) {  
147     return 'SUPPORTED_' + x3d.runtime.backendName().toUpperCase();  
148 } else {  
149     return 'UNSUPPORTED';  
150 }
```

Listing 16: Browser-Support-Test für X3DOM.

Auch die Überprüfung WebGLs ist leicht zu realisieren. Zunächst wird innerhalb von JavaScript das `window`-Objekt betrachtet. Dieses stellt das Fenster des Browsers dar und beinhaltet zusätzlich zu dessen Eigenschaften zahlreiche globale Variablen und Schnittstellen zu JavaScript-basierten Webtechnologien. Es wird überprüft, ob `window` das `WebGLRenderingContext`-Objekt enthält (vgl. Listing 17, Zeile 170 f.). Wenn dieses Element nicht existiert, ist WebGL in diesem Browser nicht implementiert und es besteht keinerlei Unterstützung für die Technologie. Andernfalls wird versucht den WebGL-Kontext mittels eines dynamisch erstellten Canvas-Elements auszulesen (Zeile 173 ff.).

Falls dies fehlschlägt, so ist anzunehmen dass der Webbrowser zwar eine grundsätzliche Unterstützung für WebGL besitzt, aber die Technologie Browser-intern deaktiviert ist.
































































```
208 if (Boolean(window.WebGLRenderingContext) === false) {
209     return 'UNSUPPORTED';
210 }
211 try {
212     var canvas = document.createElement('canvas');
213     var gl = canvas.getContext('webgl') ||
214             canvas.getContext('experimental-webgl');
215     if (!gl) {
216         return 'ERROR_INITIALIZING';
217     }
218 } catch (e) {
219     return 'ERROR_INITIALIZING';
220 }
221
222 return 'SUPPORTED';
```


Listing 17: Browser-Support-Test für WebGL.


Ursache hierfür kann ein instabiler Grafiktreiber beziehungsweise zu schwache Hardware sein. Der Browser unterbindet die Ausführung von WebGL dann anhand einer durch den Hersteller verwalteten *Blacklist*. Die Blacklist beinhaltet Hardwarekomponenten, die dafür bekannt sind, Probleme zu verursachen. Viele Browser erlauben jedoch das explizite Überschreiben dieser internen Deaktivierung WebGLs durch das Setzen sogenannter *Flags* in speziellen Einstellungsdialogen. In Chrome steht dieser beispielsweise bei Eingabe der Adresse „about:flags“ zur Verfügung. In einigen Browser-Versionen ist WebGL darüber hinaus standardmäßig deaktiviert, da die Technologie durch den Hersteller als noch zu experimentell erachtet wird.

Tabelle 6.1 auf der nächsten Seite zeigt die Testergebnisse dieser Überprüfung auf den betrachteten Plattformen. Wie in der Anforderungsanalyse spezifiziert, wurden hierbei die gängigen Desktop-Betriebssysteme (Windows, Mac OS X und GNU/Linux) untersucht. Weiterhin wurde die Unterstützung innerhalb der mobilen Plattformen Android 4.4 und der aktuellen und zukünftigen Version von iOS getestet (Version 7 und 8).


In den meisten Testfällen zeigt sich ein einheitliches Bild: WebGL wird auf fast allen Plattformen nativ unterstützt. Dadurch ist eine Darstellung von X3D durch X3DOM mit entsprechendem Backend möglich. Eine Ausnahme stellt die derzeitige Version 7 von iOS dar: Der Standard-Browser Safari von Apple bietet weder für X3DOM noch für WebGL eine Unterstützung. Aufgrund der fehlenden Unterstützung von iOS für Adobe Flash steht auch dieses Fallback-Model nicht zur Verfügung. Interessanterweise besteht bereits seit iOS 5 eine Implementierung WebGLs in Safari. Diese ist jedoch abgesehen


	Gerät	 36	 31	 9	 10	 11	 23*	 7
Windows 7	Desktop	 	 	 	 	 	 	-
Mac OS X 10.9	Notebook	 	 	-	-	-	 	 
Ubuntu 14.04	Desktop	 	 	-	-	-	 	-
Arch Linux	Desktop	 	 	-	-	-	 	-
Android 4.4	Nexus 7	 	 	-	-	-	 	-
iOS 7	iPad 3	 	-	-	-	-	-	 
iOS 8 (Beta)	Simulator	 	-	-	-	-	-	 
Android 4.4	Nexus 5	 	 	-	-	-	 	-
iOS 7	iPhone 5s	 	-	-	-	-	-	 


 X3DOM


 WebGL


- Browser nicht verfügbar


 Keine Unterstützung


 Flash-Backend


 WebGL-Backend

 X3D-Plugin

 Natives X3D

 Keine Unterstützung

 Teilweise unterstützt

 Unterstützt

* Anmerkung: Opera unter GNU/Linux in Version 12.

Tab. 6.1: Unterstützung von X3DOM und WebGL auf verschiedenen Testsystemen.

von der Werbeplattform *iAd* deaktiviert und kann nicht angesprochen werden [5]. Wie bereits im Motivationsteil erwähnt, wird mit der Veröffentlichung von iOS 8 jedoch auch dieses Hinderniss durch die Aktivierung WebGLs überwunden sein [42]. Auch in der Desktop-Variante von Safari unter Mac OS X ist WebGL bis dato nicht verfügbar. Es ist jedoch davon auszugehen, dass WebGL ähnlich zu iOS mit der für Herbst 2014 angesetzten Version 8 von Safari aktiviert sein wird. Sofern das Flash-Plugin installiert ist, bietet der Browser derzeit zumindest eine X3DOM-Unterstützung durch das Flash-Backend.

Ebenso aus dem Raster fallen die Linux-Vertreter Ubuntu 14.04 und Arch Linux. Unter GNU/Linux liegt Opera noch in Version 12 vor und hinkt der aktuellen Versionsnummer 23 nach. Die Ursache für diesen Unterschied ist die 2013 vollzogenen Umstrukturierung des Browsers. Nach einem Wechsel zur HTML-Rendering-Engine *Blink*, einer Abspaltung von *Webkit*, wurde Opera auf Basis von Chromium¹³ neu aufgesetzt [44]. Zwar existiert in Opera 12 eine WebGL-Implementierung, sie ist jedoch standardmäßig deaktiviert und muss durch die oben beschriebene *Flag* im Browser explizit aktiviert werden.

¹³Chromium ist das Open-Source-Projekt hinter Googles Webbrowser Chrome.

Da WebGL erst ab Internet Explorer 11 unterstützt wird, steht in den Versionen 9 und 10 lediglich bei X3DOM eine Unterstützung für 3D-Darstellung zu Verfügung, wenn ein Flash-Plugin installiert ist.

6.2.2 Vergleich der Hardware-Anforderungen

Um die Hardware-Anforderungen und die Performance der zwei Technologien gegenüberzustellen, wurde innerhalb der Testumgebung ein GPU-Stress-Test implementiert. Hierbei werden der 3D-Szene so lange Würfel hinzugefügt, bis die Bildwiederholrate (*Framerate*) wie in Abschnitt 4.3.3 spezifiziert unter 25 Bilder pro Sekunde (*FPS*) fällt. Die Zahl der grafischen Primitive und der Polygone (Dreiecke) wird dabei mit aufgezeichnet. Um eine Vergleichbarkeit der Betriebssysteme zu gewährleisten, wurde der Test auf allen Plattformen in Google Chrome ausgeführt.

Die Messergebnisse in Abbildung 6.3 auf der nächsten Seite zeigen eine enorme Differenz zwischen X3DOM und WebGL auf. WebGL schafft im Test unter Chrome auf jeder Plattform im Schnitt über sieben mal mehr Primitive als X3DOM darzustellen, ehe die Framerate unter 25 Bilder pro Sekunde fällt.

Da X3DOM in allen betrachteten Fällen das WebGL-Backend und damit die gleiche Technologie für das Rendering verwendete, ist dieses Ergebnis überraschend. Die Vermutung dass dieser deutlich schlechtere Wert auf die große Zahl von teuren DOM-Manipulationen durch X3DOM zurückzuführen ist, hat sich nicht bestätigt. Auch für verhältnismäßig große Zeitintervalle wie 1000 Millisekunden zwischen dem Hinzufügen eines weiteren Würfels bleiben die erzielten Werte etwa gleich. Der Grund für diesen Unterschied scheint daher in der grundsätzlichen Architektur der zwei Ansätze zu liegen. Die in Abschnitt 5.2.2 erörterte Zustandssynchronisierung der X3D-Knoten im DOM des HTML-Dokuments mit dem X3D-Backend führt wahrscheinlich zu einem größeren Rechenaufwand pro Einzelbild, als ihn Three.js als relativ schlanke Abstraktionsschicht zu WebGL verursacht.

Zum Vergleich der verschiedenen Webbrowser werden die Ergebnisse des Stress-Tests auch für Firefox in Abbildung 6.4 gezeigt. Während die X3DOM-Werte in etwa gleich bleiben, liegt die Performance von WebGL bei Firefox klar hinter der bei Google Chrome gemessenen. Weiterhin können bei WebGL mehr als zwei mal so viel Geometrien dargestellt werden als bei X3DOM, ehe die FPS-Untergrenze erreicht ist. Der Unterschied zu Google Chrome verdeutlicht die Varianz der Leistungsfähigkeit der WebGL-

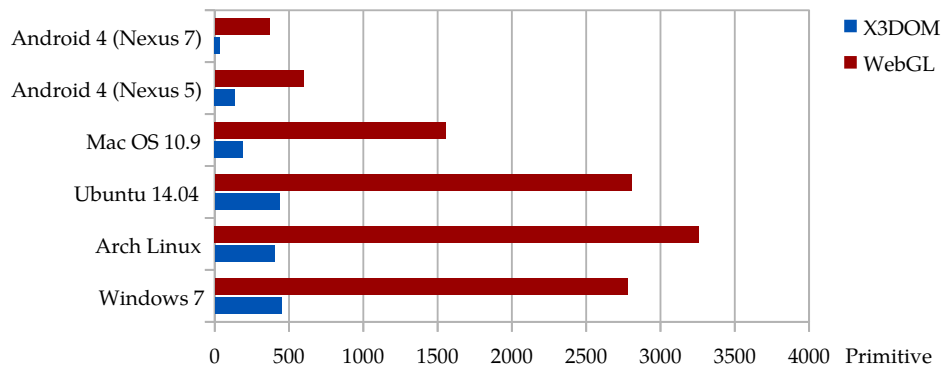


Abb. 6.3: FPS-Stresstest für 25 FPS in Google Chrome.

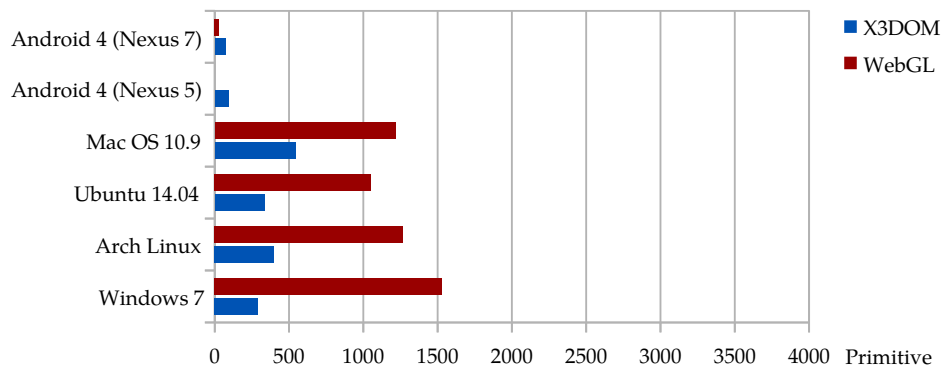


Abb. 6.4: FPS-Stresstest für 25 FPS in Firefox.

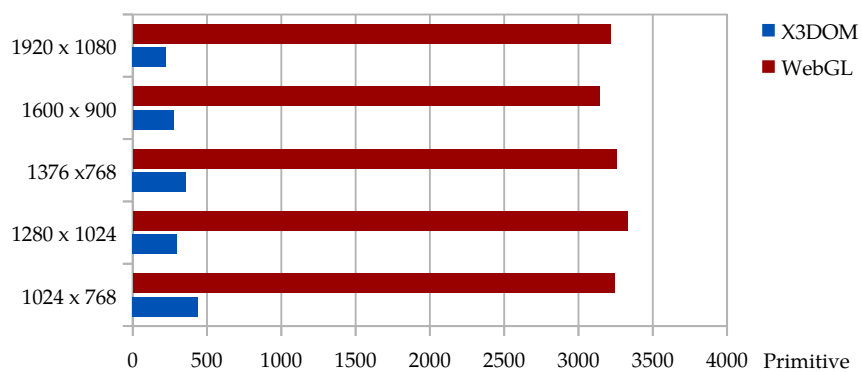


Abb. 6.5: FPS-Stresstest für verschiedene Auflösungen.

Implementierungen in den verschiedenen Browsern. Auch die Performance der jeweiligen JavaScript-Engine ist hierbei relevant, da sie beispielsweise die Geschwindigkeit von DOM-Manipulationen direkt beeinflusst.

Der Stress-Test für 25 Frames wurde zudem bei verschiedenen Bildschirmauflösungen durchgeführt, um deren Einfluss auf die Bildwiederholfrequenz zu untersuchen. Die Auflösung wird durch dynamisches Setzen der *iFrame*-Größe simuliert. Um das Ergebnis nicht durch Clipping zu verfälschen, wird der Test im Vollbild-Modus durchgeführt. Der Bildschirm des Testsystems muss dabei die maximale Auflösung von 1920 x 1080 Pixeln unterstützen. Abbildung 6.5 auf der vorhergehenden Seite zeigt die gemessenen Ergebnisse.

Die Zahl der darstellbaren Geometrien bleibt für jede der untersuchten Auflösungen wider Erwarten sowohl für X3DOM als auch für WebGL auf fast gleichem Niveau. Eine mögliche Erklärung für dieses Ergebniss könnte in JavaScript liegen. Zwar hat die Performance heutiger JavaScript-Engines innerhalb der letzten Jahre stark zugenommen [11], dennoch liegt diese aufgrund der dynamischen Natur von JavaScript weit hinter klassisch kompilierten Sprachen wie C++. Des Weiteren führen die insbesondere bei X3DOM häufigen DOM-Manipulationen zu einem möglichen Flaschenhals.

6.2.3 Import von 3D-Modellen

Der nächste Test behandelt das Laden eines extern erstellten 3D-Modells. Beide Technologien unterstützen diese Anforderung. Das Vorgehen bei X3DOM und WebGL unterscheidet sich dabei jedoch in der Komplexität, wie nachfolgend ausgeführt werden soll.

X3D ermöglicht es, mittels des Inline-Knotens sehr einfach eine externe X3D-Datei einzubinden. Hierdurch wird dem aktuellen Szenengraph an dieser Stelle der Inhalt dieser Datei angefügt, sodass ein Subszenengraph entsteht. Das Framework lädt die Daten automatisch mittels asynchroner HTTP-Anfragen. Hierdurch wird das Laden der 3D-Darstellung nicht blockiert, sondern die Elemente der Szene dynamisch nach und nach hinzugefügt, sobald sie zur Verfügung stehen.

Die X3D-Dateien können beispielsweise mit der freien 3D-Modellierungssoftware Blender [55] erstellt werden. Blender ermöglicht den Import einer Vielzahl verschiedener

```

1 <Scene>
2   <!-- Weitere Elemente -->
3   <Transform>
4     <Inline url='model.x3d'></Inline>
5   </Transform>
6 </Scene>

```

Listing 18: Einbinden einer X3D-Datei in X3DOM.

3D-Formate und kann das Modell im Anschluss in eine X3D-Datei exportieren, welche wie gezeigt in X3DOM direkt eingebunden werden kann.

Das Laden eines 3D-Modells in WebGL wird durch Three.js vereinfacht, indem dieses einige zusätzliche Hilfsmodule für verschiedene 3D-Formate mitliefert. Ein einfaches, häufig verwendetes Format ist das *Wavefront OBJ Format*. Dieses spezifiziert die *Vertices*, Seiten und Normalen und optional die Texturkoordinaten (*Texture Mapping*) eines Polygons. Three.js stellt für dieses Format einen Mechanismus bereit, um den Inhalt der Datei asynchron zu laden und wandelt das enthaltene Modell direkt in eine Framework-eigene Datenstruktur um. Die so geladene Geometrie kann der Szene anschließend hinzugefügt werden.

6.2.4 Benutzerinteraktion

Navigation

Sowohl in X3DOM als auch in Three.js stehen einige vorgefertigte Navigations-Modi bereit, die es dem Benutzer erlauben, sich auf verschiedene Arten durch die 3D-Welt zu bewegen. Innerhalb der Testumgebung wird bei jedem Test eine in der Anforderungsanalyse beschriebene Orbit-Navigation verwendet. Hierdurch kann die Darstellung beliebig gedreht, verschoben und skaliert werden.

```

1 <Scene>
2   <NavigationInfo type='examine'></NavigationInfo>
3   <!-- Weitere Elemente -->
4 </Scene>

```

Listing 19: Setzen des Navigations-Modus in X3DOM.

Der Navigationsmodus wird in X3DOM mittels des *NavigationInfo*-Knoten ausgewählt und kann mit JavaScript jederzeit dynamisch verändert werden (vgl. Listing 19). Da viele der Modi lediglich geringfügige Abwandlungen einer anderen Navigationstechnik sind, werden hier nur die wichtigsten kurz umrissen: Die Option *Examine* entspricht

der Orbit-Navigation. Weiterhin stehen *Fly* und *Walk* zur Verfügung. Während *Fly* eine völlig freie Bewegung im Raum zulässt, ist der Benutzer bei *Walk* hinsichtlich der Y-Achse auf den Untergrund eingeschränkt. Schließlich unterstützt das Framework den Modus *Look at*. Hierbei bewegt sich die Kamera in Richtung eines durch Klick anvisierten Punktes im Raum, sodass dieser näher betrachtet werden kann.

Auch Three.js stellt durch die beigefügten Beispiele des Frameworks einige Navigations-techniken bereit, die den Modi von X3DOM sehr ähnlich sind. Die Hilfsmodule sind zwar kein offizieller Bestandteil des Frameworks, funktionieren jedoch zuverlässig. Zu den wichtigsten gehören die Orbit-Navigation, ein Flugmodus und eine Bewegung in der Art eines *First Person Shooters*.

Auswahl von Objekten (Picking)

Ein weiterer wichtiger Punkt der Benutzerinteraktion ist das Auswählen einzelner 3D-Objekte im Raum, um eine zugehörige Aktion wie beispielsweise die Anzeige eines Popups auszulösen. Hierbei wird ein sogenanntes *Picking*-Verfahren angewandt. Picking kehrt die Projektion der 3D-Darstellung auf den zweidimensionalen Bildschirm um, indem ein Lichtstrahl ausgehend von der Klickposition in Blickrichtung der Kamera ausgesendet wird [48]. Sobald der Strahl auf ein Objekt trifft, gilt dieses als angeklickt.

In X3DOM ist dieses Verfahren bereits integriert und äußerst einfach in der Anwendung. Einem Shape-Knoten kann mittels des Attributs `onClick` analog zu klassischem HTML eine aufzurufende Funktion (*Event-Handler*) zugewiesen oder ein JavaScript-Ausdruck direkt eingebettet werden:

```
1 <Shape>
2   <Appearance>
3     <Material diffuseColor='1 0 0'></Material>
4   </Appearance>
5   <Box onclick='clickHandler(event)'></Box>
6 </Shape>
```

Listing 20: Deklaration eines *onclick-Handlers* in X3DOM.

Bei WebGL ist eine Umsetzung von Picking komplizierter. Three.js vereinfacht die Implementierung dabei aber erheblich durch Bereitstellung einiger Hilfsmethoden. Mittels des *Projector*-Objekts des Frameworks kann der Strahlengang nach einem Mausklick im Raum verfolgt werden und ermöglicht so die Ermittlung der Objekte unterhalb des Mauszeigers. Diese können im Anschluss beliebig manipuliert werden. Im Test wer-

den die zur Demonstration eingefügten Würfel beim Überfahren mit der Maus schwarz eingefärbt und verschwinden, sobald auf sie geklickt wird.

6.2.5 Realistische Grafikdarstellung

Für die realistische Darstellung von 3D-Modellen ist, wie in der Anforderungsanalyse ausgeführt, insbesondere die Beleuchtung, der Schattenwurf, das Shading und die Texturierung entscheidend. Der Realismus-Test demonstriert diese Aspekte durch eine entsprechende Darstellung der Utah-Teekanne¹⁴ durch X3DOM und Three.js.



Abb. 6.6: Vergleich der texturierten 3D-Modelle.

Abbildung 6.6 zeigt das jeweilige Resultat des mit einem Muster texturierten 3D-Modells. Beide Frameworks unterstützen Smooth-Shading mit einer diffusen und spekularen Reflexion. Hierdurch wird eine glatte Oberfläche der Teekanne mit Glanzeffekten erzielt. Auch Schatten kann in beiden Ansätzen realisiert werden. Dabei zeigt sich jedoch ein Unterschied hinsichtlich der Qualität der Darstellung. Bei genauer Betrachtung der Schattenkontur in Abbildung 6.6a zeigt sich, anders als bei WebGL, kein weicher Übergang der Schattierung, sondern eine harte, ausgefranste Kante. Generell wirkt das mit Three.js gerenderte 3D-Modell hierdurch visuell ansprechender.

¹⁴Die Utah-Teekanne wurde 1975 von dem Computergrafiker Martin Newell im Rahmen einer Forschungsarbeit an der Universität von Utah entworfen. Sie hat sich zu einer Art „Hello Word“ der Computergrafik entwickelt und dient sehr häufig als beispielhaftes 3D-Modell [47].

6.2.6 Entwicklungs-Aufwand

Aus den in Anforderung 4.3.5 ausgeführten Gründen ist auch der Entwicklungsaufwand ein wichtiger zu bedenkender Faktor für die Umsetzung einer Web3D-Anwendung. Die bisherigen Beispiele und Erläuterungen veranschaulichen gut die paradigmatisch grundverschiedene Vorgehensweise des deklarativen (X3DOM) und des imperativen (WebGL) Vertreters von Web3D.

Während X3D durch den XML-basierten Aufbau eine sehr hohe Konvergenz zu gut verstandenen Webtechnologien wie HTML aufweist, stellt WebGL einen typischen Webentwickler aufgrund des sehr niedrigen Abstraktionsniveaus der Bibliothek vor anspruchsvollere Konzepte [15]. Die große Nähe zu Open GL ES erfordert ein weitaus tiefgreifenderes Verständnis für die theoretischen Grundlagen der Computergrafik, als es X3D durch seinen deklarativen Stil verlangt. Populäre WebGL-Frameworks wie das behandelte Three.js können diese Hürde zwar durch Abstraktion der Funktionalität deutlich senken, dennoch ist der Einarbeitungsaufwand bei WebGL höher einzustufen.

6.3 Bewertung der Testergebnisse

Die Browser- und Plattformunterstützung ist bei beiden Technologien durch das WebGL-Backend von X3DOM sehr ähnlich. Wie Tabelle 6.1 zeigt, steht WebGL und damit auch X3D in nahezu allen aktuellen Browser-Versionen zur Verfügung. Wie ausgeführt besitzt X3DOM ein ausgereiftes Fallback-System und kann bei Bedarf dynamisch auf das weit verbreitete Flash-Plugin umschalten. Hierdurch können auch Benutzer des Internet Explorers 9 und 10 adressiert werden, woraus eine größere generelle Browserunterstützung resultiert. Mit der Veröffentlichung von iOS 8 im Herbst 2014 wird zudem auch eine der letzten verbleibenden Plattformen eine Unterstützung für WebGL erhalten.

Wie der Vergleich der Hardware-Anforderungen zeigt, ist WebGL zumindest innerhalb des betrachteten Testfalls, also bei vielen Geometrien in einer Szene, leistungstechnisch X3DOM überlegen. Bezogen auf die exemplarischen Anwendungsfälle, welche zu Beginn der Anforderungsanalyse in Abschnitt 4.2 beschrieben wurden, würde dieser Fall beispielsweise bei der Darstellung komplexer wissenschaftlicher Abbildungen mit sehr vielen Einzelementen auftreten. Bei Betrachtung eines einzelnen Produkts tritt dieser Aspekt hingegen in den Hintergrund und andere Kriterien wie der Entwicklungsauf-

wand und die generelle Einfachheit bei der Umsetzung gewinnen an Bedeutung. Hinsichtlich dieser Kriterien ist X3DOM gegenüber WebGL aufgrund seines deklarativen Ansatzes zu bevorzugen. Im Gegensatz zu WebGL, welches ein zumindest grundlegendes Verständnis computergrafischer Konzepte und der Grafik-Pipeline erfordert, sind hier bereits wenige HTML-Kenntnisse ausreichend, um eine 3D-Szene mittels X3DOM umzusetzen.

Beide Frameworks ermöglichen das Importieren einiger offener 3D-Grafikformate, so dass bestehende 3D-Modelle problemlos eingebunden werden können. Dabei ist gegebenenfalls eine Umwandlung des Ausgangsmodells in ein durch X3DOM beziehungsweise Three.js unterstütztes Format durch Modellierungssoftware wie Blender [55] notwendig. Durch die Unterstützung der gängigen Beleuchtungs-Modelle, Shading-Verfahren und Texturierung verhalten sich beide Ansätze hinsichtlich der realistischen Darstellung von 3D-Modellen ähnlich. X3DOM wies innerhalb des Tests hier jedoch einige Artefakte bei der Schattenkontur auf. Die spezifizierten Anforderungen bezüglich der Benutzerinteraktion können in beiden Technologien gleich gut realisiert werden.

7 Zusammenfassung

Seit dem Aufkommen der Web Graphics Library im Jahr 2009 ist eine neue Dynamik im Web3D-Umfeld entstanden. Mit der heutigen Verfügbarkeit von WebGL in nahezu jedem Webbrowser ist die Darstellung hardwarebeschleunigter 3D-Grafik heute ohne Plugins möglich. Mit X3DOM hat sich innerhalb der letzten Jahre ein umfangreiches JavaScript-Framework entwickelt, welches die Nutzung des freien X3D-Standards zur Deklaration von 3D-Szenen in HTML ermöglicht. Behr et al. erhoffen sich, langfristig eine native Unterstützung von X3D innerhalb der Browser ähnlich zu SVG zu erzielen. Auch auf Seite von WebGL sind seit dessen Erscheinung eine Vielzahl von Frameworks entstanden, welche die Programmierung anspruchsvoller 3D-Anwendungen mit großer Flexibilität erlauben. Durch Abstraktion der Funktionalität der Hardware-nahen Grafikbibliothek wird deren Benutzung stark vereinfacht.

Wie die Evaluation gezeigt hat, ähneln sich X3DOM und WebGL in vielen Aspekten wie der Browser- und Plattformunterstützung und den Möglichkeiten hinsichtlich der Benutzerinteraktion sehr. Auch der Import bestehender 3D-Daten ist bei beiden Ansätzen einfach zu realisieren. Der hauptsächliche Unterschied der zwei betrachteten Technologien liegt in ihrer paradigmatischen Grundlage. Zwar ermöglicht es der deklarative Stil von X3DOM sehr leicht, 3D-Szenen auch ohne Programmierkenntnisse umzusetzen, jedoch schränkt dies die Flexibilität des Entwicklers gleichzeitig ein, da nicht jeder Aspekt der Darstellung bis ins letzte Detail beeinflusst werden kann. Je nach konkreter Anwendung und Vorwissen des Software-Entwicklers muss die Wahl der 3D-Technologie somit von Fall zu Fall entschieden werden.

Die von Ortiz 2010 hauptsächlich gesehenen Probleme, die Web3D davon abhalten, sich im WWW stärker zu etablieren, – der Mangel an Standardisierung und die Notwendigkeit von Browser-Plugins – können im Jahr 2014 als überwunden betrachtet werden. Es bleibt abzuwarten, inwieweit sich WebGL in den nächsten Jahren weiterentwickeln wird und ob es X3D ebenso schaffen wird, eine native Unterstützung innerhalb der Webbrowser zu erreichen.

Quellenverzeichnis

Literatur

- [1] Matti Anttonen et al. „Transforming the Web into a Real Application Platform: New Technologies, Emerging Trends and Missing Pieces“. In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. SAC '11. TaiChung, Taiwan: ACM, 2011, S. 800–807. ISBN: 978-1-4503-0113-8. DOI: 10.1145/1982185.1982357. URL: <http://doi.acm.org/10.1145/1982185.1982357>.
- [2] Johannes Behr et al. „A Scalable Architecture for the HTML5/X3D Integration Model X3DOM“. In: *Proceedings of the 15th International Conference on Web 3D Technology*. Web3D '10. Los Angeles, California: ACM, 2010, S. 185–194. ISBN: 978-1-4503-0209-8. DOI: 10.1145/1836049.1836077. URL: <http://doi.acm.org/10.1145/1836049.1836077>.
- [3] Johannes Behr et al. „X3DOM: A DOM-based HTML5/X3D Integration Model“. In: *Proceedings of the 14th International Conference on 3D Web Technology*. Web3D '09. Darmstadt, Germany: ACM, 2009, S. 127–135. ISBN: 978-1-60558-432-4. DOI: 10.1145/1559764.1559784. URL: <http://doi.acm.org/10.1145/1559764.1559784>.
- [4] Gavin Bell, Anthony Parisi und Mark Pesce. *The Virtual Reality Modeling Language. Version 1.0 Specification*. Version 1.0. 1995. URL: <http://web.archive.org/web/20131031165054/http://www.web3d.org/x3d/specifications/vrml/VRML1.0> (besucht am 22. 07. 2014).
- [5] Alberto Benin, G. Riccardo Leone und Piero Cosi. „A 3D Talking Head for Mobile Devices Based on Unofficial iOS WebGL Support“. In: *Proceedings of the 17th International Conference on 3D Web Technology*. Web3D '12. Los Angeles, California: ACM, 2012, S. 117–120. ISBN: 978-1-4503-1432-9. DOI: 10.1145/2338714.2338734. URL: <http://doi.acm.org/10.1145/2338714.2338734>.
- [6] Robin Berjon et al. *HTML5. Candidate Recommendation*. W3C, Juni 2014. URL: <http://www.w3.org/TR/2014/WD-html5-20140617/> (besucht am 29. 06. 2014).
- [7] Don Brutzman und Leonard Daly. *X3D: Extensible 3D Graphics for Web Authors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 012088500X, 9780080489889.

- [8] Robert L. Cook. „Shade Trees“. In: *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), S. 223–231. ISSN: 0097-8930. DOI: 10.1145/964965.808602. URL: <http://doi.acm.org/10.1145/964965.808602>.
- [9] L. Daly und D. Brutzman. „X3D: Extensible 3D Graphics Standard [Standards in a Nutshell]“. In: *Signal Processing Magazine, IEEE* 24.6 (Nov. 2007), S. 130–135. ISSN: 1053-5888. DOI: 10.1109/MSP.2007.905889.
- [10] Fraunhofer Institut für graphische Datenverarbeitung. *Official X3DOM Documentation. Class: NavigationInfo*. Version 3.3. 2014. URL: <http://doc.x3dom.org/developer/x3dom/nodeTypes/NavigationInfo.html> (besucht am 23.07.2014).
- [11] Alun Evans et al. „3D graphics on the web: A survey“. In: *Computers & Graphics* 41 (2014), S. 43–61. ISSN: 0097-8493. DOI: <http://dx.doi.org/10.1016/j.cag.2014.02.002>. URL: <http://www.sciencedirect.com/science/article/pii/S0097849314000260>.
- [12] James D. Foley et al., Hrsg. *Computer Graphics : Principles and Practice*. 2. ed. in C, reprint. with corr., 25. print. The systems programming series. Boston [u.a.]: Addison-Wesley, 2010. ISBN: 0-201-84840-6 - 978-0-201-84840-3.
- [13] Ian Hickson und David Hyatt. *HTML5. Working Draft*. W3C, Feb. 2009. URL: <http://www.w3.org/TR/2009/WD-html5-20090212/no.html#declarative-3d-scenes> (besucht am 16.07.2014).
- [14] ISO. *The Virtual Reality Modeling Language – Part 1: Functional specification and UTF-8 encoding*. ISO 14772-1:1997. Geneva, Switzerland: International Organization for Standardization, 1997.
- [15] Jacek Jankowski et al. „Declarative Integration of Interactive 3D Graphics into the World-wide Web: Principles, Current Approaches, and Research Agenda“. In: *Proceedings of the 18th International Conference on 3D Web Technology. Web3D '13*. San Sebastian, Spain: ACM, 2013, S. 39–45. ISBN: 978-1-4503-2133-4. DOI: 10.1145/2466533.2466547. URL: <http://doi.acm.org/10.1145/2466533.2466547>.
- [16] Arnaud Le Hors et al. *Document Object Model (DOM) Level 3 Core Specification*. Recommendation. W3C, Apr. 2004. URL: <http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/> (besucht am 22.07.2014).
- [17] Chris Marrin. *WebGL Specification*. Version 1.0. Feb. 2011. URL: <https://www.khronos.org/registry/webgl/specs/1.0.0/> (besucht am 17.05.2014).
- [18] Kouichi Matsuda und Rodger Lea. *WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL*. 1. Aufl. Harlow, England: Longman Group, 2013. ISBN: 0321902920, 9780321902924.
- [19] Aaftab Munshi und Jon Leech. *OpenGL ES. Common Profile Specification*. Version 2.0.25. 2010. URL: http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf (besucht am 22.07.2014).

- [20] T. Narten. *Considerations for Having a Successful Birds-of-a-Feather (BOF) Session*. RFC 5434 (Informational). Internet Engineering Task Force, Feb. 2009. URL: <http://www.ietf.org/rfc/rfc5434.txt>.
- [21] Sixto Ortiz. „Is 3D Finally Ready for the Web?“ In: *Computer* 43.1 (Jan. 2010), S. 14–16. ISSN: 0018-9162. DOI: 10.1109/MC.2010.15.
- [22] Tony Parisi. *WebGL: Up and Running. Building 3D Graphics for the Web*. 1. ed. Beijing [u.a.]: O'Reilly & Associates, 2012. ISBN: 978-1-449-32357-8 - 1-449-32357-X.
- [23] L.D. Paulson. „Developers shift to dynamic programming languages“. In: *Computer* 40.2 (Feb. 2007), S. 12–15. ISSN: 0018-9162. DOI: 10.1109/MC.2007.53.
- [24] A Taivalsaari und T. Mikkonen. „The Web as an Application Platform: The Saga Continues“. In: *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*. Aug. 2011, S. 170–174. DOI: 10.1109/SEAA.2011.35.
- [25] A Taivalsaari et al. „The Death of Binary Software: End User Software Moves to the Web“. In: *Creating, Connecting and Collaborating through Computing (C5), 2011 Ninth International Conference on*. Jan. 2011, S. 17–23. DOI: 10.1109/C5.2011.9.
- [26] Klaus Zeppenfeld und Regine Wolters. *Lehrbuch der Grafikprogrammierung : Grundlagen, Programmierung, Anwendung*. 1. Aufl. Lehrbücher der Informatik. Heidelberg [u.a.]: Spektrum Akad. Verl., 2004. ISBN: 3-8274-1028-2.

Online-Quellen

- [27] Luz Caballero. *Better Performance With requestAnimationFrame*. Juni 2013. URL: <http://dev.opera.com/articles/better-performance-with-requestanimationframe/> (besucht am 11.07.2014).
- [28] Web3D Consortium. *About Web3D Consortium*. Juli 2014. URL: <http://www.web3d.org/about> (besucht am 22.07.2014).
- [29] World Wide Web Consortium. *About W3C*. Juli 2014. URL: <http://www.w3.org/Consortium/> (besucht am 27.05.2014).
- [30] World Wide Web Consortium. *People of the W3C*. URL: <http://www.w3.org/People/#dsr> (besucht am 05.06.2014).
- [31] World Wide Web Consortium. *W3C Confirms May 2011 for HTML5, Last Call Targets 2014 for HTML5 Standard*. URL: <http://www.w3.org/2011/02/htmlwg-pr.html.en> (besucht am 27.06.2014).
- [32] Valve Cooperation. *Steam-Hard- & Software-Umfrage*. Apr. 2014. URL: <http://store.steampowered.com/hwsurvey> (besucht am 09.05.2014).

- [33] Statistisches Bundesamt (Destatis). *Zahl der mobilen Internetnutzer im Jahr 2013 um 43 % gestiegen*. März 2013. URL: https://www.destatis.de/DE/PresseService/Presse/Pressemitteilungen/2014/03/PD14_089_63931.html (besucht am 21.05.2014).
- [34] Alexis Deveria. Juli 2014. URL: <http://caniuse.com/svg> (besucht am 03.07.2014).
- [35] Internet Engineering Task Force. *BOF Procedures*. März 1993. URL: <http://www.ietf.org/wg/bof-procedures.html> (besucht am 05.06.2014).
- [36] Khronos Group. *About The Khronos Group*. 2014. URL: <http://www.khronos.org/about/> (besucht am 13.07.2014).
- [37] Khronos Group. *OpenGL ES 2.0 for the Web*. 2014. URL: <http://www.khronos.org/webgl/> (besucht am 21.05.2014).
- [38] Khronos Group. *OpenGL Overview*. Juli 2014. URL: <http://www.opengl.org/about/> (besucht am 23.07.2014).
- [39] Ilmari Heikkinen. *Typed Arrays: Binary Data in the Browser*. Juli 2014. URL: http://www.html5rocks.com/en/tutorials/webgl/typed_arrays/ (besucht am 22.07.2014).
- [40] Google Inc. *Blocked plug-ins*. Juli 2014. URL: <https://support.google.com/chrome/answer/1247383> (besucht am 10.07.2014).
- [41] Benoit Jacob und Robert Nyman. *The Concepts of WebGL*. Apr. 2013. URL: <https://hacks.mozilla.org/2013/04/the-concepts-of-webgl/> (besucht am 07.07.2014).
- [42] Iker Jamardo. *WebGL on iOS 8 Safari and WebView!* Juni 2014. URL: <http://blog.ludei.com/webgl-ios-8-safari-webview/> (besucht am 23.06.2014).
- [43] Tim Johansson. *Taking the canvas to another dimension*. Nov. 2007. URL: <http://web.archive.org/web/20071117170113/http://my.opera.com/timjoh/blog/2007/11/13/taking-the-canvas-to-another-dimension> (besucht am 13.07.2014).
- [44] Bruce Lawson. *300 Million Users and Move to WebKit*. Feb. 2013. URL: <http://dev.opera.com/blog/300-million-users-and-move-to-webkit/> (besucht am 22.07.2014).
- [45] Rob Mauceri und Sandeep Singhal. *IE11 for Windows 7 Globally Available for Consumers and Businesses*. Nov. 2013. URL: <http://blogs.msdn.com/b/ie/archive/2013/11/07/ie11-for-windows-7-globally-available-for-consumers-and-businesses.aspx> (besucht am 14.07.2014).
- [46] Mozilla. *Add-ons that cause stability or security issues are put on a blocklist*. Juli 2014. URL: <https://support.mozilla.org/en-US/kb/add-ons-cause-issues-are-on-blocklist> (besucht am 10.07.2014).
- [47] Computer History Museum. *The Utah Teapot*. 2014. URL: <http://www.computerhistory.org/revolution/computer-graphics-music-and-art/15/206> (besucht am 24.07.2014).

- [48] Soledad Penadés. *Object picking*. 2014. URL: <http://soledadpenades.com/articles/three-js-tutorials/object-picking/> (besucht am 15.07.2014).
- [49] David Raggett. *Extending WWW to support Platform Independent Virtual Reality*. Nov. 1995. URL: <http://www.w3.org/People/Raggett/vrml/vrml.html> (besucht am 04.06.2014).
- [50] StatCounter. *Top 9 Desktop, Mobile & Tablet Browsers from Jan 2009 to Apr 2014*. Mai 2014. URL: <http://gs.statcounter.com/#desktop+mobile+tablet-browser-ww-monthly-200901-201404> (besucht am 09.05.2014).
- [51] Vladimir Vukićević. *Canvas 3D: GL power, web-style*. Nov. 2007. URL: <http://web.archive.org/web/20140222194911/http://blog.vlad1.com/2007/11/26/canvas-3d-gl-power-web-style/> (besucht am 13.07.2014).

Software

- [52] Autodesk, Inc. *3DS Max*. San Rafael, CA, USA, 2014. URL: <http://www.autodesk.de/products/3ds-max> (besucht am 26.06.2014).
- [53] Autodesk, Inc. *3ds Max*. San Rafael, CA, USA, 2014. URL: <http://www.autodesk.de/products/3ds-max> (besucht am 26.06.2014).
- [54] Nicolas Garcia Belmonte. *PhiloGL: A JavaScript WebGL Framework*. 2014. URL: <https://github.com/senchalabs/philogl> (besucht am 04.06.2014).
- [55] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation. Blender Institute, Amsterdam, 2014. URL: <http://www.blender.org> (besucht am 26.06.2014).
- [56] Brandon Jones. *glMatrix*. 2014. URL: <https://github.com/toji/gl-matrix> (besucht am 28.05.2014).
- [57] Ricardo Cabello (Mr.doob). *three.js. JavaScript 3D library*. 2014. URL: <https://github.com/mrdoob/three.js/> (besucht am 04.06.2014).

Abbildungsverzeichnis

3.1	Verschiedene Darstellungen eines 3D-Modells.	5
3.2	Ambiente, diffuse und spekulare Beleuchtung.	5
3.3	Exemplarischer Szenengraph (unvollständig).	6
3.4	Schematische Darstellung der Standard-Grafikpipeline.	7
3.5	Berechnung der Modelmatrix.	8
3.6	Berechnung der MV-Matrix.	9
3.7	Berechnung der MVP-Matrix.	9
4.1	Marktanteil von Webbrowsern weltweit [50].	14
4.2	Zehn meistgenutzte Bildschirmauflösungen auf Steam [32].	17
5.1	Architektonischer Aufbau von X3DOM. Geringfügig adaptiert nach Behr et al. [2].	24
5.2	Orthogonale und perspektivischen Darstellung der Pyramide.	25
5.3	Grundarchitektur einer WebGL-basierten Web3D-Anwendung.	30
5.4	Verschiedene Ansichten der Beispiel-Anwendung.	34
5.5	Topologische Struktur der Pyramide.	38
6.1	Verschiedene Tests innerhalb der Testumgebung.	43
6.2	Architektur der Testumgebung.	44
6.3	FPS-Stresstest für 25 FPS in Google Chrome.	49
6.4	FPS-Stresstest für 25 FPS in Firefox.	49
6.5	FPS-Stresstest für verschiedene Auflösungen.	49
6.6	Vergleich der texturierten 3D-Modelle.	53

Tabellenverzeichnis

6.1 Unterstützung von X3DOM und WebGL auf verschiedenen Testsystemen.	47
---	----

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst habe. Ich versichere, dass ich keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe, und dass die eingereichte Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen ist.

Erlangen, den 14. Februar 2017

Jonathan Gruber

A Quelltext der Beispiel-Anwendung

A.1 X3DOM

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset='utf-8'>
5      <title>X3D-Beispiel: Rotierende Pyramide</title>
6      <link rel='stylesheet' type='text/css' href='lib/x3dom/x3dom.css' />
7      <link rel='stylesheet' type='text/css' href='../examples-base/examples.css' />
8      <style type='text/css'>
9          X3D {
10             display: inline-block;
11             background: #fff;
12             border: solid 5px #ccc;
13         }
14     </style>
15 </head>
16 <body>
17     <div id='box'>
18         <h1>X3D-Beispiel</h1>
19         <X3D width='800px' height='600px'>
20             <Scene>
21                 <Viewpoint id='persp-vp' position='0 0 5'></Viewpoint>
22                 <OrthoViewpoint id='ortho-vp' position='0 0 10'
23                     fieldOfView='[-2, -2, 2, 2]'></OrthoViewpoint>
24                 <NavigationInfo type='none'></NavigationInfo>
25
26                 <Transform DEF='pyramid' translation='0 -0.75 0'>
27                     <Shape>
28                         <IndexedFaceSet coordIndex='
29                             4 0 1 -1
30                             4 1 2 -1
31                             4 2 3 -1
32                             4 3 0 -1
33                             3 2 1 0 -1
34                         '>
35                         <Coordinate point='
36                             1 0 -1
37                             -1 0 -1
38                             -1 0 1
39                             1 0 1
40                             0 2 0
41                         '></Coordinate>
42                     </IndexedFaceSet>
43                     <Appearance>
44                         <Material diffuseColor='0 0.66 0.66'></Material>
45                     </Appearance>
46                 </Transform>
47             </Scene>
48             <TimeSensor id='time' DEF='time' cycleInterval='4' loop='true'></TimeSensor>
```

```

50         <OrientationInterpolator DEF='move' key='0 1' keyValue='
51             0 1 0 0
52             0 1 0 3.14
53         '></OrientationInterpolator>
54
55         <Route fromNode='time' fromField='fraction_changed'
56             toNode='move' toField='set_fraction'></Route>
57
58         <Route fromNode='move' fromField='value_changed'
59             toNode='pyramid' toField='set_rotation'></Route>
60     </Scene>
61 </X3D>
62
63 <div id='controls'>
64     <div class='fieldwrap'>
65         <input type='checkbox' data-property='animation' checked='checked' />
66         <label>Animation</label>
67     </div>
68     <div class='fieldwrap'>
69         <label>Projektion:</label>
70         <select data-property='perspective'>
71             <option>Perspektivisch</option>
72             <option>Orthogonal</option>
73         </select>
74     </div>
75 </div>
76
77 <p class='notes'>Hinweis: X3DOM unterstützt derzeit keine
78 Drahtgitter-Darstellung.</p>
79 </div>
80
81 <script type='text/javascript' src='lib/x3dom/x3dom.js'></script>
82 <script src='../examples-base/base.js'></script>
83 <script>
84     document.onload = function () {
85         var timeSensor = document.getElementById('time');
86         var perspView = document.getElementById('persp-vp');
87         var orthoView = document.getElementById('ortho-vp');
88
89         document.addEventListener('configChanged', function(e) {
90             var enabled = (cfg.animation) ? 'true' : 'false';
91             timeSensor.setAttribute('enabled', enabled);
92
93             if (cfg.perspective) {
94                 perspView.setAttribute('set_bind', 'true');
95             } else {
96                 orthoView.setAttribute('set_bind', 'true');
97             }
98         });
99     };
100 </script>
101 </body>
102 </html>

```

A.2 WebGL

```

1 // Self invoking anonymous function to prevent pollution of global namespace
2 (function() {
3     'use strict'; // Strict mode
4
5     var gl; // WebGL context
6     var canvas; // The HTML canvas DOM element
7     var numOfTris; // Number of to be drawn triangles
8     var rotAngle = 0; // Current angle of rotation (animation)

```

```

9
10 var modelMatrix = mat4.create(); // Model matrix
11 var vpMatrix = mat4.create(); // View projection matrix
12 var mvpMatrix = mat4.create(); // Model view projection matrix
13 var mvpMatrixUniform; // Location of mvMatrix uniform
14
15 function main() {
16     var glOptions = {antialias: true};
17     try {
18         canvas = document.getElementById('gl-canvas');
19         if (canvas && Boolean(window.WebGLRenderingContext)) {
20             gl = canvas.getContext('webgl', glOptions) ||
21                 canvas.getContext('experimental-webgl', glOptions);
22         }
23     } catch (e) {
24         console.error('Your webbrowser does not seem to support WebGL.')
25     }
26
27     if (gl) {
28         initGl();
29         initShaders();
30         initBuffers();
31         tick();
32     } else {
33         console.error('WebGL could not be initialized.');

```

```

80     function initShaders() {
81         var fragmentShader = getShader(gl, 'shader-fs');
82         var vertexShader = getShader(gl, 'shader-vs');
83
84         gl.program = gl.createProgram();
85         gl.attachShader(gl.program, vertexShader);
86         gl.attachShader(gl.program, fragmentShader);
87         gl.linkProgram(gl.program);
88
89         if (!gl.getProgramParameter(gl.program, gl.LINK_STATUS)) {
90             console.error('Unable to initialize the shader program.');
```

91 }

92 gl.useProgram(gl.program);

93

94 mvpMatrixUniform = gl.getUniformLocation(gl.program, 'uMVPMatrix');

95 }

96

97 /**

98 * Returns a compiled shader object from embedded shader source

99 * @param {WebGLRenderingContext} gl WebGL context

100 * @param {string} id ID of the shader script

101 * @return {WebGLShader|bool} Compiled shader object or false on error

102 */

103 function getShader(gl, id) {

104 var shader;

105 var shaderScript = document.getElementById(id);

106

107 if (!shaderScript)

108 return false;

109

110 switch (shaderScript.type) {

111 case 'x-shader/x-fragment':

112 shader = gl.createShader(gl.FRAGMENT_SHADER);

113 break;

114 case 'x-shader/x-vertex':

115 shader = gl.createShader(gl.VERTEX_SHADER);

116 break;

117 }

118

119 gl.shaderSource(shader, shaderScript.textContent);

120 gl.compileShader(shader);

121

122 if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {

123 console.error(gl.getShaderInfoLog(shader));

124 return false;

125 }

126

127 return shader;

128 }

129

130 /**

131 * Creates buffer objects, writes vertex data and passes them to WebGL

132 */

133 function initBuffers() {

134 var vertices = new Float32Array([

135 0.5, -0.5, -0.5, // P0

136 -0.5, -0.5, -0.5, // P1

137 -0.5, -0.5, 0.5, // P2

138 0.5, -0.5, 0.5, // P3

139 0.0, 0.4, 0.0 // P4

140]);

141

142 var indices = new Uint8Array([

143 4, 0, 1, // Back

144 4, 1, 2, // Left

145 4, 2, 3, // Back

146 4, 3, 0, // Front

147 0, 1, 2, // Bottom

148 0, 2, 3 // Bottom

149]);

150

```

151     var colors = new Float32Array([
152         0.0, 0.0, 1.0, // Blue
153         0.0, 1.0, 1.0, // Cyan
154         1.0, 0.0, 0.0, // Red
155         0.8, 0.0, 0.5, // Pink
156         0.0, 1.0, 0.0, // Green
157     ]);
158
159     numOfTris = indices.length;
160
161     /**
162     * Nested helper function: Creates, binds and writes the buffer data
163     * @param {Float32Array} data Buffer data
164     * @param {string} attributeName Name of shader attribute
165     */
166     var buffer, FSIZE, attribute;
167     function setupBuffer(data, attributeName) {
168         buffer = gl.createBuffer();
169         gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
170         gl.bufferData(gl.ARRAY_BUFFER, data, gl.STATIC_DRAW);
171
172         FSIZE = data.BYTES_PER_ELEMENT;
173         attribute = gl.getAttribLocation(gl.program, attributeName);
174         gl.vertexAttribPointer(attribute, 3, gl.FLOAT, false, FSIZE * 3, 0);
175         gl.enableVertexAttribArray(attribute);
176     }
177
178     setupBuffer(vertices, 'aPosition');
179     setupBuffer(colors, 'aColor');
180
181     var indexBuffer = gl.createBuffer();
182     gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
183     gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indices, gl.STATIC_DRAW);
184 }
185
186 /**
187 * Animaton loop - Recalled by requestAnimatinoFrame
188 */
189 function tick() {
190     if (cfg.animation) {
191         animate();
192     } else {
193         lastTick = Date.now(); // Save rotation state
194     }
195     render();
196     requestAnimationFrame(tick);
197 }
198
199 /**
200 * Animation - Takes time delta between ticks into account
201 */
202 var now, lastTick, delta;
203 function animate() {
204     if (typeof lastTick === 'undefined')
205         lastTick = Date.now();
206     now = Date.now();
207     delta = now - lastTick;
208     lastTick = now;
209
210     rotAngle += (0.03 * delta) % 360;
211     mat4.rotateY(modelMatrix, mat4.create(), degToRad(rotAngle));
212 }
213
214 /**
215 * Renders a single frame
216 */
217 function render() {
218     mat4.multiply(mvpMatrix, vpMatrix, modelMatrix);
219     gl.uniformMatrix4fv(mvpMatrixUniform, false, mvpMatrix);
220 }
221

```



```

222         gl.clear(gl.COLOR_BUFFER_BIT|gl.DEPTH_BUFFER_BIT);
223         var drawMethode = (cfg.wireframe) ? gl.LINE_STRIP : gl.TRIANGLES;
224         gl.drawElements(drawMethode, numOfTris, gl.UNSIGNED_BYTE, 0);
225     }
226
227     /**
228     * Helper function: Converts degrees to radians
229     * @param {number} deg Degree representation
230     * @return {number} Radian representation
231     */
232     function degToRad(deg) {
233         return deg * Math.PI / 180;
234     }
235
236     })();

```

B Quelltext der Testumgebung

Aufgrund des Umfangs der implementierten Testumgebung wird deren Quelltext hier nicht abgedruckt, sondern liegt lediglich digital auf der beigefügten CD vor. Der vollständige Quellcode wurde jedoch zusätzlich in ein GitHub-Repository hochgeladen. Dieses ist unter folgender Adresse zu erreichen:

<https://github.com/gruberjonathan/web3d>

Die Testumgebung ist zusätzlich unter folgender Adresse online verfügbar:

<http://web3d.webreaktor.net>