Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelorarbeit Medieninformatik

Analyse und Vergleich von WebGL-Frameworks

Moritz Kornher

26. August 2013

Gutachter

Prof. Dr. rer. nat. Andreas Schilling
Lehrstuhl für Medieninformatik (Computer Vision)
Universität Tübingen

Kornher, Moritz:
Analyse und Vergleich von WebGL-Frameworks
Bachelorarbeit Medieninformatik
Eberhard Karls Universität Tübingen
Bearbeitungszeitraum: 25. April 2013 - 26. August 2013

Danksagung

Ich danke meinen Eltern, die mir dieses Studium ermöglicht haben und allen Korrekturlesern für ihre Unterstützung.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum Unterschrift

Kurzfassung

Als Zukunftstechnologie wird WebGL immer häufiger zur Entwicklung von 3D-Computergrafik-Anwendungen im Web eingesetzt. Allerdings ist WebGL ein komplexer Standard und seine Programmierschnittstelle ist nicht einfach zu lernen. Frameworks können daher bei der Entwicklung von WebGL-Anwendungen helfen. In meiner Arbeit stelle ich mit three.js, Processing.js und SceneJS exemplarisch drei Frameworks vor. Anhand einer Demoanwendung, die ich mit allen drei Frameworks umgesetzt habe, analysiere und vergleiche ich die Frameworks untern bestimmten Kriterien. Ich lege dar welche Konzepte der Softwareentwicklung sie befolgen, wie groß ihr Funktionsumfang ist und ob die Entwicklung weitergeführt wird. Ebenfalls von Interesse sind die Qualität und Quantität von Dokumentationen und anderen Hilfestellungen zu den Frameworks. Auch auf spezielle Eigenschaften der Frameworks gehe ich ein. Die Ergebnisse zeigen, dass diese Frameworks dabei helfen können, WebGL-Anwendungen zu entwickeln. Abschließend gebe ich einen Überblick über die Vor- und Nachteile der Frameworks und schlage Verbesserungen vor.

Abstract

As a future technology, WebGL will be used more and to develop 3d computer graphics web applications. Since WebGL is a complex standard and its application programming interface is difficult to learn. Frameworks may be the answer for future development of WebGL applications. As an example for such frameworks this paper presents three.js, Processing.js and SceneJS. I analyse and compare these frameworks on the basis of a demo application, which I implemented in all three frameworks. I explain their adherence of software developing concepts and present the scale of their functionality. It is also shown whether they are still actively developed. Documentation and other assistances are considered in terms of quality and quantity, as well. Additionally, I talk about their specific characteristics. The results show that these frameworks help to develop WebGL applications. Finally, I give a review on their advantages and disadvantages plus possible improvements.

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Inhaltsverzeichnis

1 Einleitung

Die moderne 3D-Computergrafik hat sich in den letzten Jahren erheblich weiterentwickelt. Getrieben von neuen und aufstrebenden Branchen, wie der Computerspiel-Industrie und der Werbebranche, wurden die Grenzen der Computergrafik immer mehr erweitert. Insbesondere seit Mitte der 1990er Jahre auch im Privatanwender-Bereich gesonderte Hardware zur Berechnung von 3D-Computergrafik eingesetzt wurde, hat sich der Fortschritt beschleunigt. (1) Die Rechenleistung stieg und komplett neue Entwicklungen, wie programmierbare *Shader*, ermöglichten eine Vielzahl neuer Effekte.

Zeitgleich begann sich das World Wide Web zu entwickeln und spätestens Anfang der 2000er Jahre zogen immer mehr klassische Anwendungen ins Web um. Ob Bürosoftware oder Computerspiel – viele Programme sind inzwischen eng mit dem Internet verzahnt oder werden sogar vollständig darin realisiert. (2) Entsprechend entwickelten sich auch die Techniken dahinter weiter und heute bieten HTML5, CSS3 und JavaScript zusammen eine mächtige und flexible Entwicklungsplattform. Der Browser wurde so zum wichtigsten Programm für den Privatanwender.

Es war daher nur eine Frage der Zeit, bis diese beiden Zukunftstechnologien einen gemeinsamen Weg einschlagen würden. Mit der Veröffentlichung von WebGL wurde ein erster großer Schritt getan. Wo wird uns dieser Weg hinführen, nachdem die Grundlagen nun gelegt sind? Eine mögliche Antwort auf diese Frage geben uns die Entwickler mit ihren Vorstellungen von Frameworks für WebGL-Anwendungen. Die Arbeit dieser Pioniere wird mit entscheidend dafür sein, wie moderne 3D-Computergrafik-Anwendungen für das Web in den nächsten Jahren aufgebaut sein werden; welche Möglichkeiten sie bieten und welche Entwicklungen uns noch erwarten können.

In dieser Bachelorarbeit werde ich drei WebGL-Frameworks mit unterschiedlichen Ansätzen analysieren und einen Statusbericht darüber geben, was diese neue Technologie heute schon leisten kann und in Zukunft vielleicht leisten wird.

2 Grundlagen

Der WebGL-Standard basiert auf verschiedenen Konzepten und Standards aus der 3D-Computergrafik. Darüber hinaus ist er darauf ausgelegt mit anderen Web-Technologien zu interagieren. Im Folgenden gehe ich näher auf diese Bereiche ein und beschreibe ihre Verbindung zu WebGL.

2.1 3D-Computergrafik in der Webprogrammierung

Schon früh nach der Etablierung des World Wide Webs gab es Bestrebungen das WWW in die dritte Dimension zu erweitern und bereits zur erstmals stattfindenden „International World Wide Web Conference“ (3) im Mai 1994 wurde ein entsprechender Beitrag veröffentlicht. Unter dem Titel „Extending WWW to support Platform Independent Virtual Reality“ beschrieb David Raggett den Entwurf für eine „Virtual Reality markup language“ (4); VRML war geboren. Noch im November desselben Jahres wurde die erste Spezifikation veröffentlicht (5) auf die 1997 eine überarbeitete Version 2.0 (später umbenannt in VRML97) folgte (6), welche später auch als ISO-Standard anerkannt wurde. Obwohl bereits 1995 erste Plug-Ins für populäre Browser veröffentlicht wurden, konnte sich VRML vor allem im Privatanwenderbereich nicht weiter verbreiten. Die Gründe dafür lagen in Unzulänglichkeiten der Spezifikation und den Grenzen der damaligen Computer- und Verbindungstechnik. VRML galt schon bald als „gestorben und seiner Zeit voraus“. (7) Dennoch entwickelte das Web3D Consortium den XML-basierten X3D-Standard als Nachfolger, der auch bis heute aktualisiert wird. (8)

Während es im Bereich der Auszeichnungssprachen für 3D-Szenen also eine stetige Entwicklung gab, konnte lange Zeit kein Programm zur Darstellung dieser 3D-Inhalte eine größere Verbreitung erreichen. Zwar gab (und gibt) es spezielle Browser und Plug-Ins für VRML und X3D (9), aber erst Macromedia Shockwave (sowie später und weitaus erfolgreicher, das 2D-Pendant Macromedia Flash) (10) und die Java-Technologie erreichten entsprechend viele Anwender, um als allgemeine Plattform für 3D-Anwendungen im Web in Frage zu kommen. Gleichzeitig versuchten viele Firmen mit proprietärer und spezialisierter Software ihren Erfolg. Insbesondere Ende der 2000er

Jahre kamen, getrieben vom kommerziellen Erfolg von 3D-Computerspielen und Browsergames, mehrere Grafik-Engines als Browser-Plug-Ins auf den Markt. Beispielhaft genannt sei hier der Unity Web Player[1] der als Browser-Client für die Unity3D-Engine, einer weitverbreiteten Cross-Platform-Engine, dient (11) und einmal installiert eine Vielzahl von Spielen im Browser unterstützt.

2.2 OpenGL

OpenGL ist eine plattformunabhängige Programmierschnittstelle für interaktive 2D- und 3D-Computergrafik. Die erste Version der Spezifikation wurde 1992 als offene Variante von IRIS GL, einer proprietären Graphik-API von SGI, veröffentlicht. Noch heute ist die Nutzung des Standards ohne Kosten oder Lizenz möglich. (12) Nach der Veröffentlichung wurde die Verantwortung für die zukünftige Entwicklung an das Firmen-Konsortium OpenGL Architecture Review Board (ARB) übertragen. In den folgenden Jahren wurde die Entwicklung weiter vorangetrieben und verschiedene Versionen veröffentlicht. Mit Herausgabe der Version 2.1 ging 2006 die Verantwortung für den OpenGL Standard auf die Khronos Group über. Das ARB agiert seitdem als Arbeitsgruppe innerhalb der Khronos Group. (13)

Die Spezifikation definiert eine Anzahl Funktionen die von einem Programm aufgerufen werden können. Das Design von OpenGL zielt darauf ab, dass möglichst viele Funktionen auf der Hardware implementiert werden; die Berechnungen können jedoch auch softwareseitig erfolgen. (14) OpenGL ist dadurch prinzipiell unabhängig von bestimmten Plattformen oder Programmiersprachen. In der Praxis bietet ein Grafiktreiber dem Betriebssystem diese Schnittstellen an und kümmert sich intern um die Ausführung auf einer Hardware. Hersteller von Grafikhardware können auch eigene Erweiterungen bereitstellen. Durch ein stufenweises Verfahren können diese Eingang in den Hauptstandard finden, sofern sie von anderen Herstellern und dem ARB unterstützt werden. (15)

[1] http://unity3d.com/webplayer

Als wichtige Neuerungen späterer OpenGL-Spezifikationen seien das Hinzufügen der OpenGL Shading Language (GLSL) in Version 2.0, sowie die Einführung eines neuen Konzepts zur (Rückwärts-) Kompatibilität mit Version 3.0 und den folgenden Veröffentlichungen, genannt. (16) Derzeit aktuell ist Version 4.4 des Standards. (17)

Im Jahr 2008 wurde Kritik an OpenGL bzw. der Khronos Group laut, da man es nicht geschafft hat einen dem Hauptkonkurrenten DirectX ebenbürtigen Standard zu veröffentlichen. Erst durch die neuen Features der Version 4.0 konnte OpenGL wieder mit DirectX gleichziehen. (16) (18)

2.2.1 OpenGL ES

Um den immer größer werdenden Markt eingebetteter Systeme zu bedienen, wurde 2002 eine funktionsreduzierte Variante von OpenGL 1.3 als OpenGL for Embedded Systems 1.0 veröffentlicht. (19) Im März 2007 folgte auf Basis von OpenGL 2.0 die Veröffentlichung von OpenGL ES 2.0, das nun für die Verwendung auf Hardware mit programmierbaren Shadern vorgesehen ist. Da gleichzeitig die meisten Elemente aus der *Fixed Function Pipeline* entfernt wurden, besteht keine Abwärtskompatibilität zu OpenGL ES 1.x. (20) Dieser Versionszweig ist weiterhin für Systeme mit Fixed Function Pipeline vorgesehen und soll entsprechend gewartet werden. Im August 2012 folgte die Veröffentlichung von OpenGL ES 3.0 als abwärtskompatible Weiterentwicklung der Version 2.0. Die neue Spezifikation basiert auf OpenGL 3.0 und bringt neben einer neuen Version der GLSL ES vor allem verbesserte Funktionen im Bereich des *Texture-Mappings* mit sich. (21)

Besonders im Smartphone-Bereich hat OpenGL ES einen extrem hohen Verbreitungsgrad erreicht. (18) Nicht zuletzt deshalb ist in den letzten Jahren der Anteil an OpenGL basierter Software stark angestiegen. (22)

2.3 WebGL

Auf Basis von OpenGL ES 2.0 wurde im April 2009 damit begonnen eine auf JavaScript basierende Version der API für Webbrowser zu entwickeln. Gleichzeitig begannen Browserhersteller mit der Implementierung in ihren Produkten. (23) Im März

2011 erfolgte dann die Veröffentlichung der ersten Version der Spezifikation. (24) Bereits zum offiziellen Start wurde die API von den Browsern Firefox, Chrome und testweise von Opera und Safari unterstützt. (25) Da OpenGL ES unter Windows nicht standardmäßig unterstützt wird, existiert seit November 2011 mit der von Google entwickelten Angle-Bibliothek[2] eine Zwischenschicht die OpenGL ES Befehle in DirectX übersetzt. Somit sind Open GL ES und WebGL auf allen populären Plattformen verfügbar. (26)

Wie bereits erwähnt basiert WebGL auf der Spezifikation von OpenGL ES 2.0. Dennoch wurden diverse Änderungen vorgenommen, um die Portabilität zwischen den verschiedenen Software- und Hardware-Plattformen zu gewährleisten. Hauptsächlich konzentriert sich der Standard jedoch auf seine Umsetzung mit JavaScript und im Zusammenspiel mit HTML5. Die Integration in bisherige Web-Standards besteht darin, dass ein *HTML5 Canvas* als Container für den WebGL-Kontext dient. Dadurch fügt sich die neue Technik nahtlos in das *Document Object Model* von HTML ein.

Derzeit wird WebGL von den wichtigsten Desktop-Browsern aller Plattformen unterstützt oder die Unterstützung wurde für eine kommende Version angekündigt. Auf mobilen Geräten ist WebGL aktuell unter Android und Blackberry 10 mit bestimmten Browsern verfügbar. Insgesamt ist die mobile Verbreitung allerdings noch nicht soweit fortgeschritten wie auf dem Desktop. (27)

2.4 Frameworks

Als Framework bezeichnet man ein abstraktes Design für eine bestimmte Art von Anwendung. Es stellt, unabhängig von einer konkreten Anwendung, robusten und leicht wiederverwendbaren Code bereit, der in seiner Gesamtheit eine einheitliche Struktur für eine Anwendung vorgibt. Frameworks werden dann verwendet, wenn eine einfache Sammlung von Klassen oder Funktionen nicht mehr ausreichend ist. (28) Sie werden

[2] *Almost Native Graphics Layer Engine*: http://code.google.com/p/angleproject/

daher eingesetzt um als Grundlage für eine Anwendung zu dienen. Auf einem Framework aufbauend, soll die Anwendung eine konkrete Aufgabe erfüllen.

Die Qualität eines Frameworks lässt sich durch seinen Umfang beschreiben. Je mehr Bereiche abgedeckt werden und je mehr Funktionen es bereitstellt, umso nützlicher wird es bei der Entwicklung sein. Natürlich muss sich der Umfang auf das gewählte Anwendungsgebiet beschränken. Aber auch der Grad der Kapselung muss bei einer Bewertung in Betracht gezogen werden. Als Ideal gilt hier das Modell einer Black-Box, aus der nur über definierte Schnittstellen Informationen nach außen dringen. (28)

3 Zielsetzung

In dieser Bachelorarbeit werden verschiedene WebGL-Frameworks daraufhin untersucht, wie geeignet sie sind bei der Entwicklung von WebGL-Anwendungen zu helfen. In Bezug auf die Funktionalität werde ich mich besonders auf die drei folgenden Bereiche konzentrieren, welche die Grundlagen der 3D-Computergrafik darstellen.

3D-Modellierung

Für jede Bildsynthese stellt eine 3D-Szene die Basis dar. Eine solche Szene wird aus verschiedenen 3D-Modellen zusammengesetzt, wobei aufwendigere Modelle üblicherweise mit einem speziellen Programm erstellt und dann über ein Austauschformat importiert werden. Jedes Objekt besteht aus einzelnen Vertices, welche wiederum Polygone bilden. Einfachere Objekte können auch aus geometrischen Primitiven gebildet werden.

Untersucht werden soll daher vor allem, wie Vertices, Polygone und geometrische Primitiven erzeugt und kombiniert werden können, aber auch welche Funktionen zum Laden von komplexen 3D-Modellen bereitgestellt werden.

Shading

Unter Shading können alle Vorgänge zusammengefasst werden, die Einfluss auf die Berechnung der Oberflächenfarben nehmen. In der Regel finden diese Berechnungen in Vertex- und Fragment-Shadern statt. Ausgehend von *Color Mapping* und einfachen Beleuchtungsmodellen als Grundlage, sollen auch fortgeschrittene Shading-Techniken betrachtet werden. Explizit genannt seien hier das *Bump-Mapping* zur Erhöhung des Detailgrades, *Shadow-Mapping* zur Darstellung von Schatten, sowie *Environment-Mapping* als Möglichkeit Reflexionen und Lichtbrechung zu simulieren. Von Interesse ist auch die Umsetzung von Transparenz und *Blending*. Darüber hinaus gilt es festzustellen, ob das Framework es ermöglicht eigene Shader zu verwenden.

Animation

Auf das Wesentliche reduziert entsteht eine Animation durch eine minimale Änderung der Szene zwischen zwei Frames. Dies kann sehr einfach durch eine Änderung der Transformations-Matrix eines Objekts erreicht werden. Es gibt natürlich auch fortgeschrittene Techniken. Beim sog. *Keyframing* werden die Eigenschaften eines Objekts zunächst für einzelne Schlüsselbilder gespeichert, die Zwischenschritte werden dann interpoliert. Für das *Rigging* wird zunächst ein Skelett aus mehreren Knochen erstellt, welches dann animiert werden kann. Das 3D-Modell wird wie eine Haut über das Skelett gelegt und übernimmt die Animationen entsprechend. *Morphing* ist eine Methode um die Position von Vertices in Relation zum Objekt selbst zu verändern, ohne gleich das 3D-Modell austauschen zu müssen. Auch hierzu werden Schlüsselbilder verwendet und die Positionen der eindeutigen Vertices jeweils gespeichert. Der Übergang wird wiederum interpoliert. Betrachtet werden soll, wie einfach sich eine Animation durch Änderungen der Transformationen umsetzen lässt und ob durch das Framework noch weitere Animationstechniken bereitgestellt werden.

Die Bewertung der Frameworks erfolgt in dieser Bachelorarbeit durch die Betrachtung der folgenden Kriterien. Anhand einer Demoanwendung habe ich diese Kriterien auch in der Praxis analysiert.

Konzepte der Softwareentwicklung

Ich betrachte, welche Konzepte der Softwareentwicklung das Framework nutzt und ob diese die Entwicklung einer 3D-Computergrafik-Anwendung für das Internet fördern. Geprüft wird insbesondere auch, ob und wie sich diese Prinzipien vom OpenGL-Zustandsautomaten unterscheiden. Ein gutes Framework sollte hier Abstraktionsebenen einführen. Wie sich das Framework in den bestehenden Kanon von JavaScript-Bibliotheken integrieren lässt, stellt sich als weitere Frage.

Funktionsumfang und Einschränkungen

Untersucht wird welche Funktionen von WebGL durch das Framework unterstützt werden. Der Schwerpunkt wird dabei auf die weiter oben beschriebenen

Bereiche gelegt. Ich werde darlegen auf welche Funktionen der Entwickler durch das Framework Zugriff erhält und welche in der Black-Box verschwinden. Einschränkungen, die durch die Verwendung des Frameworks entstehen, sollen ebenso erwähnt werden.

Demoanwendung

Hierfür werde ich alle Beobachtungen zusammenfassen, die sich mir bei der Implementierung der Demoanwendung mit dem jeweiligen Framework ergeben haben. Insbesondere stehen hier Fragen zu Zugänglichkeit, Erweiterbarkeit, Debugging und Programm-Struktur im Vordergrund.

Dokumentation, Beispiele und Community

Ich analysiere, ob eine Dokumentation zum Framework existiert und welchen Umfang diese im Vergleich zu den tatsächlich vorhanden Funktionen hat. Auch kommentierte Beispielanwendungen können helfen bei der Entwicklung auftretende Probleme zu lösen. Außerdem betrachte ich die Größe und Aktivität der Community und versuche einen Anhaltspunkt zu geben, wie hilfreich sie bei der Lösung von Problemen sein kann.

Weiterentwicklung und Erweiterungen

Wie bereits besprochen existiert der WebGL-Standard erst seit verhältnismäßig kurzer Zeit. Daher sind auch die entsprechenden Frameworks noch nicht lange in Entwicklung. Eine entscheidende Frage bei der Wahl eines WebGL-Frameworks, muss daher auch sein welche zukünftigen (Weiter-)Entwicklungen geplant sind.

Auch unabhängig von möglichen Weiterentwicklungen bieten (externe) Erweiterungspakte und Bibliotheken die Möglichkeit ein Framework zu verbessern und immer wiederkehrende Aufgaben zu abstrahieren. Analysiert werden die Qualität und Quantität vorhandener Erweiterungen, sowie die Rahmenbedingungen für das Entwickeln eigener Bibliotheken.

Weitere Besonderheiten des Frameworks

Falls ein Framework darüber hinaus Besonderheiten oder Alleinstellungsmerkmale besitzt, werde ich ebenfalls darüber berichten und sie wenn nötig genauer betrachten.

3.1 Demoanwendung

Die oben aufgeführten Kriterien werden für jedes Framework auch durch die Entwicklung einer Demoanwendung überprüft. Auf Grund der zunehmenden Komplexität moderner 3D-Computergrafik-Anwendungen beschränkt sich die Demoanwendung nur auf die Implementierung bestimmter Teilaspekte. Der Fokus soll dabei vor allem auf den Besonderheiten der Implementierung mit dem jeweiligen Framework liegen. Diese Beschreibung soll daher auch nicht im Sinne einer konkreten Vorgabe verstanden werden.

Konkret stellt die Demoanwendung eine Visualisierung der *Epipolargeometrie* dar: Eine Szene wird von zwei Kameras betrachtet. Für einen beliebigen Bildpunkt in einem der Bilder werden die korrespondierenden *Epipolarlinien*, sowie die *Epipole* der beiden Kameras berechnet und dargestellt. Die Epipolargeometrie dient in dieser Bachelorarbeit jedoch ausschließlich als Anwendungsbeispiel und wird daher nicht weiter erläutert. Für weiterführende Informationen zu diesem Thema siehe beispielsweise Hartley und Zisserman: *Multiple View Geometry in Computer Vision.* (29)

Szene

Die Szene enthält mehrere 3D-Objekte. Der aus einfachen geometrischen Primitiven zusammengesetzte sogenannte *Drinking Bird* (30) wird direkt mit den verfügbaren Vertex- und Geometriefunktionen erstellt. Das komplexere 3D-Modell des *Stanford Bunnys* (31) wurde ursprünglich durch einen Laserscan erstellt und hier mit einem externen 3D-Modellierungs-Programm in ein geeignetes Format konvertiert. Es wird mit Hilfe des Frameworks in die Szene importiert.

Eine *Skybox* bildet den Hintergrund der Szene. Dieses Verfahren ist eines der ältesten und einfachsten, um den Hintergrund einer Szene darzustellen. (32)

Shading

Die Szene wird durch mehrere unterschiedliche Lichtquellen beleuchtet. Es werden die verschiedenen verfügbaren Beleuchtungsmodelle des Frameworks verwendet. Die Objekte werfen einen Schatten auf sich selbst und ihre Umgebung. Der Boden ist texturiert und wird zusätzlich mittels Bump-Mapping detaillierter dargestellt. Wasser- und Glasoberflächen erscheinen durchsichtig und reflektieren ihre Umgebung.

Kameras und Ansichten

Die Szene wird von drei Kameras aus unterschiedlichen Perspektiven betrachtet. Die Hauptkamera dient zur Betrachtung der Szene durch den Benutzer. Zwei weitere Kameras visualisieren das Epipolargeometrie-Modell. Jede der Ansichten wird gerendert und in einem eigenen Fenster ausgegeben. Die beiden Epipolargeometrie-Kameras werden in der Hauptansicht durch symbolische Kamera 3D-Modelle dargestellt; in den anderen beiden Ansichten sind diese aber nicht sichtbar.

Animation

In der Demoanwendung wird die Animation durch eine einfache Änderung des Rotationswinkels erzeugt. Bei jedem *Rendering*-Vorgang wird der Körper des Drinking Birds ein wenig weiter gedreht. Die Geschwindigkeit ist abhängig vom aktuellen Winkel der Drehung. So entsteht eine Animation, die grob den Kippvorgang simuliert.

Interaktion

Maus- und Tastatureingaben werden zur Positionierung der Kameras ausgewertet. Jede Kamera kann verschoben und gedreht werden. Um eine der Epipolargeometrie-Kameras zu positionieren, wird diese vorher mit der Maus ausgewählt. Des weiteren kann in den Epipolargeometrie-Ansichten per Mausklick ein Punkt in der Szene ausgewählt werden. Die Demoanwendung berechnet entsprechend die Epipole und Epipolarlinien und zeichnet diese in den beiden Ansichten ein.

3.2 Betrachtete WebGL-Frameworks

Die zur Betrachtung ausgewählten Frameworks sollten sich von ihrem Ansatz her möglichst stark unterscheiden. Da die genannten Kriterien vor allem Grundlagen der 3D-Computergrafik und von Webanwendungen beschreiben, sind die Ergebnisse dennoch vergleichbar. Mit *three.js* habe ich eines der derzeit populärsten WebGL-Frameworks gewählt (vgl. Tabelle 1). Es besticht neben seinem großen Funktionsumfang vor allem durch die große Anzahl konkreter Projekte. Als zweites Framework habe ich mich für *Processing.js* entschieden. Processing.js ist eine Portierung der *Processing Visualization Language* für JavaScript. Es teilt sich daher den Java-Syntax und Objektorientierten-Ansatz mit seinem Schwester-Projekt. Besonders interessant ist Processing.js, weil es die Möglichkeit bietet leicht portierbaren Code für mehrere Plattformen zu entwickeln. Neben der Unterstützung für Browser und Java-Applets, ist es auch möglich Apps für Android (33) und iOS (34) zu erstellen. Zuletzt fiel meine Wahl auf *SceneJS*. Dieses Framework überführt eine komplette Szene inklusive Kamera und Renderer in eine Baumstruktur, um eine effiziente Darstellung von sehr vielen Objekten zu erreichen. Darüber hinaus bietet es eine ausgereifte Plattform für Plug-Ins und integriert so beispielsweise ein dynamisches XHR-basiertes Ladesystem. SceneJS wird in kommerziellen Produkten aus dem Gesundheitsbereich[3] bereits produktiv eingesetzt. (35)

	GLGE	*J3D*	*jax*	*PhiloGL*	*Processing.js*	*SceneJS*	*Three.js*
Anzahl Sterne	321	470	738	501	1.083	253	12.478

Tabelle 1: Beliebtheit von WebGL-Frameworks auf GitHub (36)

[3] http://www.biodigital.com/biodigital-human.html

4 three.js

Im April 2010 wurde die erste Version von three.js durch Ricardo Cabello auf GitHub[4] veröffentlicht. Die Ursprünge des Frameworks liegen jedoch weiter zurück. Bereits 2006 hatte Cabello die Idee ein 3D-Framework zu entwickeln und begann später auch mit der Entwicklung von three.as für *ActionScript 2*. Geschwindigkeitssteigerungen bei der Interpretation von JavaScript, sowie die Möglichkeit unabhängig von bestimmten Programmen und Systemen arbeiten zu können, bewogen Cabello dann dazu den Code für JavaScript zu portieren. (37) Heute ist three.js eines der meistverwendeten Frameworks für WebGL und wird vielfach in kommerziellen Webanwendungen verwendet.

4.1 Untersuchung der Kriterien

Konzepte der Softwareentwicklung

Three.js folgt dem Prinzip der objektorientierten Programmierung und stellt dem Entwickler seine Klassen in einem eigenen Namespace bereit. Namenskonflikte mit anderen Bibliotheken werden dadurch vermieden. Das Framework weist starke Unterscheide auf im Vergleich zum klassischen zustandsbasierten OpenGL und verwaltet seine Daten in Objekten (vgl. Quellcode 1).

```
1    var scene = new THREE.Scene();
2    var geometry = new THREE.CubeGeometry(1,1,1);
3    var material = new THREE.MeshBasicMaterial({color: 0x00ff00});
4
5    var cube = new THREE.Mesh(geometry, material);
6    scene.add(cube);
```

Quellcode 1: Objektverwaltung in three.js

Besonders hervorzuheben ist dabei die klare Trennung zwischen Renderer, Szene und Kamera. Da erst beim Rendering alle relevanten Informationen zusammengeführt werden, ist es leicht Veränderung an einer dieser Komponenten vorzunehmen. Zum Nachladen von Texturen und Dateien stellt three.js verschiedene *Loader*-Klassen bereit. Da diese dem Event-System von JavaScript folgen,

[4] https://github.com/mrdoob/three.js/

lassen sich einfach *Listener* hinzufügen, um auf Events wie Fertigstellung und den Fehlerfall zu reagieren. Insgesamt integriert sich three.js sehr gut in die Welt von JavaScript, so ist es beispielsweise problemlos möglich dat.GUI [5] zu verwenden um Variablenwerte zu verändern.

Funktionsumfang und Einschränkungen

Im Bereich der 3D-Modellierung unterstützt three.js alle grundlegenden Funktionen: Das *Geometry*-Objekt verwaltet Stapel für Vertices, Farben, Normalen, Dreiecke und UV-Koordinaten. Die Indizes werden entsprechend referenziert, die Werte für Normale und Farbe können aber auch pro Dreieck festgelegt werden (vgl. Quellcode 2). Es stehen Klassen für viele, teils ausgefallenere, geometrische Primitive zur Verfügung. Mit den zahlreichen *Loader*-Klassen können außerdem Objekte in vielen verschiedenen 3D-Formaten importiert werden.

Zusammen mit einem oder mehreren Materialien bildet eine Geometrie ein *Mesh*. Three.js-Meshes können nun transformiert und mit anderen Meshes zu Hierarchien kombiniert werden. Alle Meshes zusammen ergeben eine Szene, welche dann wie bereits erwähnt vom Renderer dargestellt wird.

```
 1    var normal = new THREE.Vector3(0, 0, 1);
 2    var color = new THREE.Color(0xff0000);
 3
 4    var geometry = new THREE.Geometry();
 5
 6    geometry.vertices.push(new THREE.Vector3(-1, -1, 0));
 7    geometry.vertices.push(new THREE.Vector3( 1, -1, 0));
 8    geometry.vertices.push(new THREE.Vector3( 1,  1, 0));
 9    geometry.vertices.push(new THREE.Vector3(-1,  1, 0));
10    geometry.faces.push(new THREE.Face3(0, 1, 2, normal, color));
11    geometry.faces.push(new THREE.Face3(0, 2, 3, normal, color));
```

Quellcode 2: Eine einfache Geometrie in three.js

Shading ist in jedem Fall eine der vielen Stärken von three.js. Es bringt bereits vorgefertigte Shader für eine große Anzahl an Mapping-Techniken mit und bietet den Großteil davon auch komfortabel über Materialien und Materialeigenschaf-

[5] http://code.google.com/p/dat-gui/

ten an. Konkret unterstützt werden derzeit *Color-Mapping* mit vielen Filter-Einstellungen für Texturen, Bump- und Environment-Mapping, sowie ein gut konfigurierbares Shadow-Mapping. Neben komplett selbst geschriebenen Shadern bringt three.js eine Shader-Bibliothek mit, die es ermöglicht weitere Shading-Techniken einzusetzen: *Displacement-Mapping*, *Ambient Occlusion* oder *Refraction-Mapping* seien beispielhaft genannt (vgl. Quellcode 3). Transparente Materialien konnte ich ebenfalls problemlos verwenden. Es werden alle von mir beschriebenen Animationstechniken in three.js unterstützt.

```
1    var shader = THREE.ShaderLib["normalmap"];
2    var uniforms = THREE.UniformsUtils.clone(shader.uniforms);
3    uniforms["enableDisplacement"].value = true;
4    uniforms["tDisplacement"].value
5        = THREE.ImageUtils.loadTexture("displacement_map.jpg");
6
7    var material = new THREE.ShaderMaterial({
8        fragmentShader: shader.fragmentShader,
9        vertexShader: shader.vertexShader,
10       uniforms: uniforms,
11       lights: true
12   });
```

Quellcode 3: Displacement-Mapping mit Hilfe der Shader-Bibliothek in three.js

Dokumentation, Beispiele und Community

Es existiert zwar eine umfangreiche Dokumentation[6] zu three.js, zum Zeitpunkt der Veröffentlichung dieser Arbeit deckte diese allerdings nur etwa ein Drittel der Funktionen des Frameworks ab. Dieser Umstand kann teilweise durch ein großes und gut sortiertes Archiv von Beispiel-Implementierungen[7] ausgeglichen werden. Neben diesen beiden Quellen existiert noch ein Wiki[8], das ausführlichere Anleitungen zu einigen speziellen Komponenten enthält und vor allem eine umfangreiche Linkliste zu weiteren Hilfeseiten pflegt.

[6] http://threejs.org/docs/

[7] http://threejs.org/examples/

[8] https://github.com/mrdoob/three.js/wiki

Die Community von three.js ist im Vergleich zu den anderen in dieser Arbeit untersuchten Frameworks sehr groß. Auf GitHub erhielt das Projekt über 12.000 Sterne, es existieren ca. 2.700 *Forks* und im letzten Monat haben etwa 30 Entwickler Änderungen zum Quellcode beigetragen. Dies sind alles starke Hinweise auf eine aktive Entwickler-Community. Betrachtet man die gestellten Fragen auf der bekannten Hilfe-Plattform Stack Overflow[9] stellt man fest, dass sich User bei etwa zwei Drittel der Fragen an der Lösungsfindung beteiligt haben. Darüber hinaus bieten die Entwickler einen gut besuchten IRC-Channel[10] und eine Gruppe auf Google+[11] an, über die regelmäßig Neuigkeiten verbreitet werden.

Weiterentwicklung und Erweiterungen

Wie bereits erwähnt existiert eine große und aktive Entwickler-Community um three.js. Auch wenn die Hauptentwicklung derzeit nur von einer Person vorangetrieben wird, tragen regelmäßig weitere Programmierer zur Weiterentwicklung bei. Im letzten halben Jahr wurden fünf neue Versionen veröffentlicht. Durch mehrere gut gepflegte Erweiterungs-Sammlungen[12,13] werden auch viele Funktionen abseits des Kern-Frameworks abgedeckt. Die verwendete Architektur erleichtert es auch eigenen Code zu entwickeln, der sich gut in die three.js-Umgebung einpasst.

Weitere Besonderheiten des Frameworks

Erwähnenswert ist vor allem der große Umfang, den three.js bietet. Es gibt kaum eine Funktion die nicht schon als Klasse bereitgestellt wird oder zumindest durch ein Beispiel beschrieben wird. Darüber hinaus sind die verschiedenen Mathematik-Klassen hervorzuheben. Für „Helfer-Klassen“ sind diese vergleichsweise vollständig und gut strukturiert.

---

[9] http://stackoverflow.com

[10] http://webchat.freenode.net/?channels=three.js

[11] https://plus.google.com/104300307601542851567/

[12] http://jeromeetienne.github.io/tquery/

[13] https://github.com/jeromeetienne/threex

Ein konkretes Feature, das die beiden anderen Frameworks bisher nicht bieten, ist die Unterstützung von *Viewports*. In three.js kann dem Renderer ein Bereich des HTML5 Canvas zugeteilt werden, auf dem die Ausgabe stattfinden soll. *Scissor Tests* werden entsprechend ebenfalls unterstützt.

4.2 Implementierung der Demoanwendung

Die Demoanwendung konnte ich mit three.js problemlos und schnell implementieren. Die zahlreichen Beispiele und Anleitungen machen nicht nur den Einstieg sehr leicht, sie geben auch eine gewisse Grundstruktur für den Code vor, die zumindest für Anwendungen auf niedrigerem Niveau gut geeignet ist. Mit Hilfe der vielen mitgelieferten Primitive, konnte die Szene schnell erstellt werden. Lediglich beim Import des Stanford Bunnys traten kleinere Schwierigkeiten auf, da die Klasse zum Laden von *Wavefront .obj*-Dateien derzeit offensichtlich noch nicht alle Definitionen dieses Formats unterstützt. Der Drinking Bird konnte zwar auch leicht animiert werden, allerdings erfolgte die Implementierung schlussendlich etwas unsauber, da three.js selbst keinerlei Vorgaben über die Datenhaltung macht. So erfolgt letztendlich bei jedem neuen Rendering-Vorgang ein direkter Zugriff auf die Rotationsmatrix des drehbaren Teils des Vogels. Im Nachhinein wäre die Kapselung über eine eigene Drinking Bird-Klasse besser gewesen. Für alle vorgesehenen Shading-Techniken existierte neben der Dokumentation auch ein Beispiel oder eine Anleitung. Eine ansprechende Beleuchtung konnte durch die zahlreichen unterschiedlichen Lichtarten erzielt werden. Bei Darstellung von Schatten mittels gerichtetem Licht galt es allerdings zu beachten, dass die Lichtquelle wie bei einem Scheinwerferlicht zu platzieren war, im Gegensatz zur sonst üblichen einfachen Angabe der Richtung. Die erzeugte Shadow-Map kann durch diverse Einstellungsmöglichkeiten, wie der Auflösung und des zu betrachtenden Bereichs, so angepasst werden, dass ein guter Kompromiss zwischen Qualität und Performance entsteht.

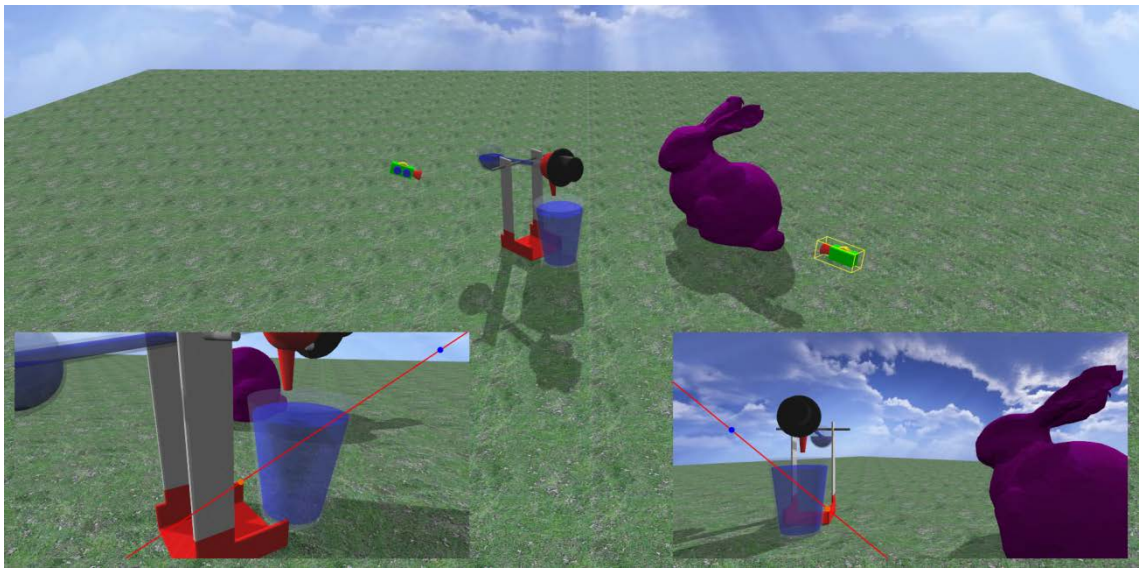

Abbildung 1:Screenshot aus der three.js-Demoanwendung

Zur Steuerung der Kameras konnte ich als Grundlage ebenfalls auf die Arbeit anderer Entwickler zurückgreifen.[14] Die Steuerung wurde von mir so angepasst, dass sie der Steuerung gängiger Programme in der sog. Ego-Perspektive entspricht. Die Koppelung mit einem Objekt zur Visualisierung der Kamera war ebenfalls einfach möglich, da in Objekten der Klasse *PerspectiveCamera* Translation und Rotation als öffentliche Eigenschaften verfügbar sind. Konkret erfolgte die Verbindung mittels eines Pointers auf den jeweils selben Speicherplatz, so dass auch bei Änderungen von Position und Ausrichtung keinerlei Aktualisierungen notwendig waren. Festgestellt habe ich hierbei, dass three.js Rotations-Daten standardmäßig als Quaternion (38) speichert und nicht wie oft üblich als Eulerwinkel. Three.js beinhaltet außerdem Klassen zum *Raycasting* und, darauf aufbauend, zum Auswählen von Objekten mittels 2D-Bildschirm-Koordinaten, wobei diese Koordinaten als *Normalized Device Coordinates* vorliegen müssen. Diese Funktion konnte ich nutzen, um eine jeweilige Kamera zur Steuerung auszuwählen. Eine Eigenart der *Raycaster*-Klasse in three.js ist jedoch, dass alle Objekte bekannt sein müssen, für die ein Schnitt mit dem ausgesendeten Strahl ermittelt werden soll. In der Praxis bedeutet dies, dass ein Stapel mit auswählbaren Objekten zu verwalten ist.

[14] Vgl. Anlage 2, Datei "./threejs/extensions/DemoControls.js"

Die Darstellung der drei verschiedenen Ansichten konnte in three.js vom Renderer mit Hilfe von Viewports und Scissor Tests vorgenommen werden. Dies hat insbesondere den Vorteil, dass für Renderer und Szene nur je ein Objekt existiert, welche dann nach Bedarf zusammen mit der entsprechenden Kamera eine Ansicht erzeugen. Gerade Änderungen innerhalb der Szene werden somit direkt an alle Ansichten weitergereicht. Die Behandlung von Mausklicks erfolgte unabhängig von three.js; jeder Klick muss abhängig von der Position auf dem Bildschirm einem Viewport zugeordnet werden, bevor er korrekt in Normalized Device Coordinates umgerechnet werden kann. Es war mir nicht ohne weiteres möglich mit three.js ein 2D-Interface zu erstellen, das über der 3D-Szene liegt. Daher erfolgte die Berechnung der Epipole und Epipolarlinen zwar mit Hilfe des Frameworks, das Einzeichnen der Punkte und Linien habe ich aber mit CSS3 realisiert.

4.3 Zusammenfassung und Bewertung

Meine Betrachtungen zeigen, dass sich three.js sehr gut dazu eignet WebGL-Anwendungen zu entwickeln. Es bietet viele Funktionen, eine ausgereifte Architektur und ist leicht erweiterbar. Das Framework arbeitet problemlos mit anderen JavaScript-Programmen zusammen und da keine besondere Struktur zur Datenhaltung gefordert wird, dürfte three.js auch in größeren Projekten flexibel nutzbar sein. Eine aktive Weiterentwickelung verspricht für die Zukunft weitere Neuerungen. Größere Lücken sind zwar noch in der Dokumentation auszumachen, allerdings können diese von der großen Community teilweise wieder ausgeglichen werden.

5 Proccessing.js

Processing.js ist eine Portierung von *Processing*, einer auf Grafik und Animation spezialisierten Programmiersprache, für JavaScript und wurde 2008 von John Resig veröffentlicht. (39) Es teilt sich daher die API und Java-ähnliche Syntax mit seinem Schwester-Projekt. Processing selbst möchte mit einer flachen Lernkurve vor allem Künstlern und Programmieranfängern den Einstieg in die Programmierung erleichtern. (40) Mit Processing.js sollen diese Ziele auch in das Web übertragen werden. Ein weiteres Prinzip ist die Kompatibilität zwischen verschiedenen Plattformen: Eine gut programmierte *Skizze*, so heißen Programme im Processing-Jargon, wird mit nur geringen Anpassungen auf allen unterstützen Plattformen ausführbar sein. Insbesondere in der Medienkunst sind Processing und Processing.js weiter verbreitet. (41)

5.1 Untersuchung der Kriterien

Da Anwendungen für Processing.js in der Programmiersprache Processing geschrieben werden und sich beide – theoretisch – die exakt gleiche API teilen, werde ich in den folgenden Abschnitten Processing als Oberbegriff verwenden. Bei expliziten Unterschieden werde ich direkt auf Processing.js verweisen.

Konzepte der Softwareentwicklung

Auch mit Processing kann objektorientiert entwickelt werden. Dies ergibt sich schon aus den Java-Ursprüngen. Die API orientiert sich allerdings sehr stark an der originalen zustandsbasierten OpenGL API, so dass das OOP-Konzept auf den ersten Blick in den Hintergrund tritt. Auch wird die eigene API innerhalb eines unsichtbaren Processing-Objektes *PApplet* global im Programm bereitgestellt und alle selbstgeschriebenen Klassen werden ebenfalls nur als Unterklassen erstellt. Die Verantwortung objektorientiert zu entwickeln, wird dem Programmierer überlassen. Die *setup*-Methode eines *PApplet* wird initial aufgerufen, anschließend bei jedem Rendering-Vorgang die Methode *draw*. Benutzereingaben werden durch weitere spezielle Methoden aufgefangen. Im Web interpretiert Processing.js den Code zu Beginn der Laufzeit als JavaScript.

Funktionsumfang und Einschränkungen

In Processing stehen alle nötigen Funktionen zur Erzeugung von Vertices und Polygonen zur Verfügung. Mehrere Vertices können zu einer Form zusammengefasst werden. Die Reihenfolge in der Vertices miteinander verbunden werden, wird durch den Modus festgelegt. Processing entfernt sich hier kaum von den entsprechenden OpenGL-Funktionen (vgl. Quellcode 4). Mittels eigener Funktionen können Farben und Normalen pro Vertex, aber auch pro Form festgelegt werden. Außerdem können in Processing die Verbindungen zwischen einzelnen Vertices als Line dargestellt werden. Es existieren mit der Box und der Kugel nur zwei vorgefertigte 3D-Primitve in Processing. Wenn auch in Processing vorgesehen, war die Funktion 3D-Modelle aus Dateien zu importieren in der aktuellen Version 1.4.1 von Processing.js noch nicht implementiert. Es gibt jedoch eine, wohl vorläufige, Implementierung, so dass in zukünftigen Versionen mit dieser Erweiterung gerechnet werden kann. (42) Transformationen werden auf die aktuelle Transformation-Matrix angewandt und können mittels des Matrizen-Stapels auf einen Wirkungsbereich beschränkt werden.

```
 1    beginShape(QUAD_STRIP);
 2    vertex(1, 1, 0,   0, 0); //x, y, z, U, V
 3    vertex(1, 6, 0,   0, 1);
 4    vertex(3, 1, 0, 0.5, 0);
 5    vertex(3, 6, 0, 0.5, 1);
 6    vertex(4, 1, 0, 0.6, 0);
 7    vertex(4, 6, 0, 0.6, 1);
 8    vertex(6, 1, 0,   1, 0);
 9    vertex(6, 6, 0,   1, 1);
10    endShape();
```

Quellcode 4: Eine einfache Form in Processing.js

Funktions-Einschränkungen gibt es im Bereich des Shadings. Zwar werden alle Standard-Lichttypen unterstützt, man kann aber praktisch keinen Einfluss auf die genauen Eigenschaften von Materialien nehmen. Auch beim Color-Mapping muss man die Vorgaben von Processing hinnehmen. Processing.js unterstützt aktuell auch noch kein *Texture Wrapping*. Weitere Mapping-Techniken stehen nicht zur Verfügung. Eigene Shader wurden ebenfalls noch nicht in Processing.js implementiert, obwohl sie durch das Schwester-Projekt unterstützt werden. Dar-

über hinaus standen mir im 3D-Kontext von Processing.js weder Blending noch Transparenz zuverlässig funktionierend zur Verfügung (vgl. Abbildung 2). Die Dokumentation trifft keine Aussagen dazu, ob diese Funktionen auch im 3D-Kontext unterstützt werden sollten. (43) Processing verfügt über keine besonderen Animationstechniken und Schatten gehören ebenfalls nicht zum Standard-Umfang.

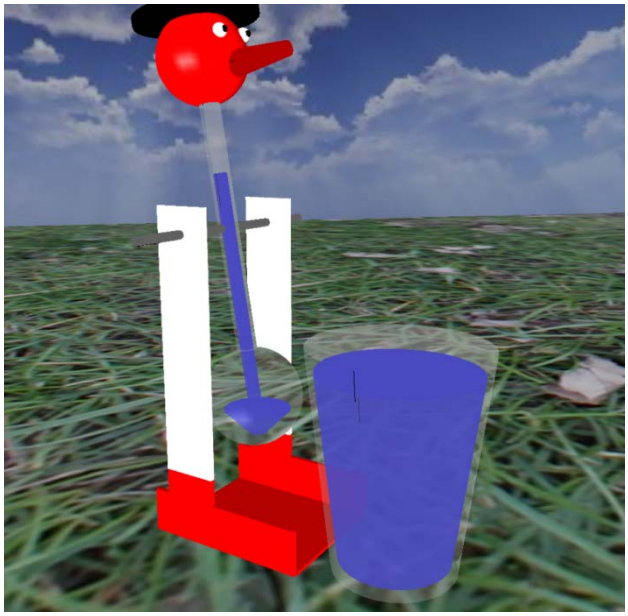

Abbildung 2: Probleme mit Transparenz in Processing.js

Die Matrizen für Modelview und Projektion werden durch spezielle Kamera-Funktionen erzeugt. Leider verhindert die Datenkapselung auch lesenden Zugriff darauf. Dies kann zwar für einfache Anwendungen Sinn machen, führt allerdings schon in meiner Demoanwendung zu Problemen.

Dokumentation, Beispiele und Community

Alle API-Funktionen von Processing.js sind vollständig dokumentiert.[15] Zusätzlich existiert eine Javadoc-Referenz[16] zu Processing, die weitere Funktionen zu den internen Klassen bereithält. Allerdings kann man sich nicht darauf verlassen,

[15] http://processingjs.org/reference/

[16] http://processing.org/reference/javadoc/core/

dass diese auch in Processing.js so verfügbar sind. Mit OpenProcessing[17] existiert eine Plattform, die sich ausschließlich dem Austausch von Processing-Skizzen widmet. Ein IRC-Channel[18] verspricht direkte Hilfe von anderen Entwicklern.

Auf GitHub wurden über 350 Forks von Processing.js erstellt und etwa 1.000 Sterne verteilt. Die Werte liegen damit sogar über denen des Schwester-Projekts. Im letzten halben Jahr, haben allerdings nur drei Entwickler Code zum Projekt beigetragen. Auf Stack Overflow finden sich nur 184 Fragen zu Processing.js, wobei zu etwas mehr als zwei Dritteln eine Antwort gegeben wurde. Zu Processing finden sich dort über 1.000 Fragen. Der Anteil unbeantworteter Fragen ist in etwa gleich hoch wie bei Processing.js.

Weiterentwicklung und Erweiterungen

Die aktuelle Version 1.4.1 von Processing.js ist vor einem Jahr erschienen. Von der Processing-Seite aus wurde im Mai 2013 festgestellt, dass die JavaScript-Portierung nicht mit der Entwicklung des Hauptprojekts Schritt halten könne. (44) Dennoch wird in den nächsten Wochen die Veröffentlichung einer neuen Version erwartet und das Entwickler-Team scheint gewillt den Rückstand aufzuholen. (45) Laut GitHub wird das Projekt allerdings derzeit nur von einem Hauptentwickler betreut. Auf Grund der Besonderheit, dass Erweiterungen für Processing in der Regel auf Java-Code aufbauen (46), können diese nicht mit Processing.js verwendet werden. Tatsächlich existiert derzeit kein Mechanismus um Plug-Ins für Processing.js in Processing-Code zu entwickeln. (47) Obwohl es auch möglich ist anderen JavaScript-Programmcode in Processing.js zu nutzen, sind mir dennoch keine speziellen Erweiterungen für Processing.js bekannt.

[17] http://www.openprocessing.org/

[18] irc://irc.mozilla.org/processing.js

Weitere Besonderheiten des Frameworks

Die wichtigste Besonderheit von Processing ist wie bereits erwähnt die Verfügbarkeit auf vielen verschiedenen Plattformen. Ansonsten ist noch die Nähe zu reinem OpenGL bemerkenswert.

5.2 Implementierung der Demoanwendung

Ein Vorteil von Processing ist, dass schon zu Beginn der Entwicklung schnell Ergebnisse zu sehen sind. Genauso schnell bin ich allerdings auch an die Grenzen von Processing gestoßen. Da Processing nur zwei geometrische Primitive mitbringt, habe ich eine Zylinder-Klasse selbst geschrieben. War die Geometrie des Drinking Birds noch problemlos zu erstellen, traten bei der Umsetzung der Transparenz Probleme auf. So konnte ich Alpha-Blending nicht mit den Processing.js-API-Funktionen aktivieren, sondern musste auf die nativen WebGL-Funktionen zurückgreifen. Dazu kommen Fehler in der Reihenfolge des *Transparency-Buffers* bzw. es ist nicht eindeutig, ob es einen solchen überhaupt gibt (vgl. Abbildung 2). Wie bereits erwähnt gibt es noch keine Funktion um 3D-Modelle zu importieren, so dass ich den Stanford Bunny weglassen musste. Auch im Shading-Bereich sind die Möglichkeiten eingeschränkt. Wie in Abbildung 3 zu sehen ist, konnte ich keine Textur-Wiederholung implementieren. Die wichtigsten Beleuchtungsmodelle sind zwar vorhanden, lassen sich aber kaum beeinflussen und sind teilweise nur schwer nachzuvollziehen. Beispielsweise war es mir nicht möglich die Zylinder so zu erzeugen, dass sie korrekt beleuchtet werden. Schatten und andere Mapping-Techniken sind nicht standardmäßig vorhanden und wurden daher nicht von mir in die Demoanwendung integriert.

Da in Processing Kamera und Szene direkt an einen Rendering-Kontext gebunden sind, musste ich insgesamt drei Skizzen erstellen, die in den jeweiligen HTML5 Canvas-Elementen dargestellt werden. Diese teilen sich zwar die meisten Daten, allerdings konnte ich diese globale Datenhaltung nur direkt in JavaScript umsetzen. Eine einfache Processing-Skizze kennt ihre Umgebung normalerweise nicht. Es ist in Processing.js, wie bereits erläutert möglich, direkt auf den JavaScript-Kontext zuzugreifen. Dies gilt es allerdings zu vermeiden, da ansonsten die Kompatibilität zu normalem Processing-

Code verloren geht. Stattdessen wird ein Interface für die benötigten JavaScript-Objekte mit allen Eigenschaften und Funktionen erstellt. Aus dem JavaScript-Teil der Anwendung heraus, können dann die entsprechenden Processing-Objekte mit realen Daten verbunden werden. Bei einer Portierung auf eine andere Plattform, muss nun lediglich ein Ersatz für diese Interfaces gefunden werden.

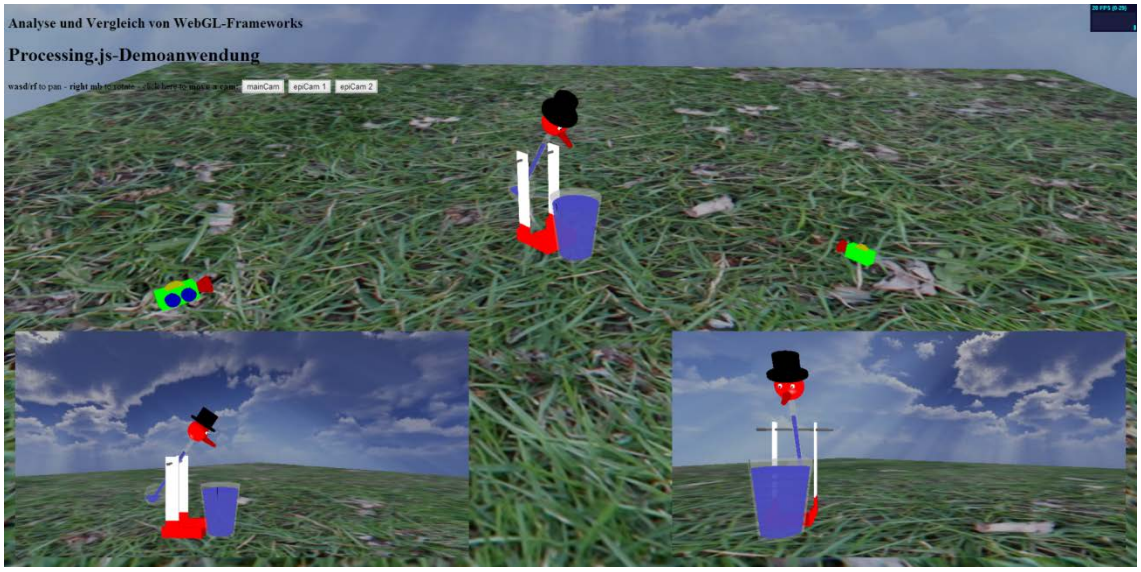

Abbildung 3: Screenshot aus der Processing.js-Demoanwendung

Die Interaktion mit Benutzereingaben stellt sich in Verbindung mit Webtechnologien als problematisch heraus. Processing.js überwacht alle Maus- und Tastaturereignisse die innerhalb des zugehörigen HTML5 Canvas-Elements stattfinden. In der Demoanwendung ist es aber nötig, die Ereignisse global für das *Document* abzufangen, da zumindest zwei weitere HTML5 Canvas-Elemente über der Haupt-Zeichenfläche liegen. Schlussendlich musste ich eine ähnliche Implementierung wie bei den anderen beiden Frameworks verwenden, welche die Ereignisse an die korrekte Skizze weiterleitet. Processing bringt keine Funktionen zum *Raytracing* oder *Raypicking* mit sich. Daher erfolgt die Auswahl der zu steuernden Kamera durch HTML Buttons. Die Maus- und Tastatur-Steuerung selbst konnte ich anhand der three.js-Steuerung problemlos implementieren. Etwas komplizierter wurde die Verwaltung der Kameras. Hier war es nötig Projektions- und Modelview-Matrizen selbst zu erstellen, da ein Zugriff über das Framework nicht möglich war. Insbesondere zur Berechnung von 3D-Weltkoordinaten aus 2D

Canvas-Koordinaten war dies notwendig. Processing bietet im Gegensatz zu den beiden anderen Frameworks lediglich Funktionen zur Projektion auf einen Bildschirm und nicht von einem Bildschirm aus an. Diese Berechnungen wurden auch dadurch erschwert, dass die Größe der Rendering-Fläche einmalig bei der Initialisierung der Skizze festgelegt wird und nicht dynamisch an das HTML5 Canvas-Element gekoppelt ist (vgl. Quellcode 5). Wie aus Quellcode 5 ersichtlich wird, musste mehrmals zwischen beiden Bezugssystemen konvertiert werden. Auch bei diesem Framework erfolgte das Einzeichnen der Epipole und Epipolarlinen mit Hilfe von CSS3.

```
 1    PVector projectPoint2Canvas(PVector point, Camera cam,
 2                                PVector rendererSize,
 3                                PVector canvasSize) {
 4        PVector projected;
 5
 6        pushMatrix();
 7        cam.activate();
 8        projected = new PVector(screenX(point.x, point.y, point.z),
 9                                screenY(point.x, point.y, point.z),
10                                screenZ(point.x, point.y, point.z));
11        resetMatrix();
12        popMatrix();
13
14        projected.x = projected.x/rendererSize.x*canvasSize.x;
15        projected.y = projected.y/rendererSize.y*canvasSize.y;
16
17        return projected;
18    }
```

Quellcode 5: Funktion zur Projektion von 3D-Weltkoordinaten auf ein HTML5 Canvas-Element mit Processing.js

5.3 Zusammenfassung und Bewertung

Es ist schwierig, komplexere WebGL-Anwendungen mit Processing.js zu entwickeln. Der bisherige Funktionsumfang reicht nicht dafür aus und die gewählte Architektur verhindert es derzeit leider noch, gut integrierte Erweiterungen zu entwickeln. Wer jedoch das Ziel hat, eine einfache 2D- oder 3D-Anwendung auf möglichst vielen Plattformen zu veröffentlichen, wird mit dem Processing-Ökosystem seine Freude habe. Getragen von einer großen und aktiven Community, erhält man schnell Hilfe und Feedback zu den eigenen Skizzen. Es bleibt insbesondere abzuwarten, inwieweit die Entwicklung von Processing.js mit der des Schwester-Projektes mithalten kann. Ein unterschiedlich großer Funktionsumfang würde die Attraktivität des Projektes erheblich einschränken.

6 SceneJS

Im Herbst 2009 begann Lindsay Kay damit SceneJS zu entwickeln und veröffentlichte seither drei größere Versionen des Frameworks. Erklärtes Ziel des Entwicklers ist es, ein Framework bereitzustellen, das für das effiziente Rendering einer Vielzahl von Objekten optimiert ist.

6.1 Untersuchung der Kriterien

Konzepte der Softwareentwicklung

SceneJS unterscheidet sich konzeptionell stark von den beiden anderen betrachteten Frameworks. Es ist zwar ebenfalls objektorientiert, allerdings wird dieses Konzept vor Programmieren praktisch versteckt. In SceneJS wird die komplette Szene in einer JSON-basierten Baumstruktur aufgebaut. Ausgehendend von einem Szenen-Knoten werden alle anderen Operationen ebenfalls als Knoten repräsentiert (vgl. Quellcode 6). Einstellungen können mit einem speziellen *Flag*-Knoten gesetzt werden. Damit für den Zugriff auf bestimmte Knoten nicht manuell der ganze Baum durchlaufen werden muss, können IDs vergeben werden. SceneJS erlaubt damit einen direkten, allerdings asynchronen Zugriff.

```
1    var scene = SceneJS.createScene({
2      nodes:[
3        {
4          type:"material",
5          color: { r: 1.0, g: 0.0, b: 0.0 },
6          nodes:[
7            {
8              type: "translate",
9              x: 0.0,
10             y: 20.0,
11             z: 0.0,
12             nodes: [
13               {
14                 type:"prims/torus"
15               }
16             ]
17           }
18         ]
19       }
20     ]
21   });
```

Quellcode 6: Einfache Szene in SceneJS

Die Architektur des Frameworks ist auf Plug-Ins ausgelegt. SceneJS besteht nur aus einem verhältnismäßig kleinen Kern und lädt einzelne Klassen bei Bedarf asynchron nach. Da dieses Prinzip auch auf Texturen und ähnliches angewandt wird, fällt diese Aufgabe nicht mehr dem Entwickler zu.

Funktionsumfang und Einschränkungen

3D-Modelle werden in SceneJS durch *Geometry*-Knoten repräsentiert, die Stapel für Vertices, Farben, Normalen, UV-Koordinaten und Dreiecke oder Linien verwalten. Erzeugt wird das Modell in Abhängigkeit der Werte und Indizes auf den Stapeln. Um ein Dreieck zu erstellen, müssen beispielsweise die Indizes von drei Vertices auf den Stapel gelegt werden. Die Werte der Normalen werden entsprechend aus ihrem Stapel gelesen. Im Stapel der UV-Koordinaten referenzieren je zwei Werte (U und V), im Farb-Stapel vier Werte (RGBA) auf ein Vertex (vgl. Quellcode 7). Normalen können von SceneJS aber auch automatisch berechnet werden. Eine Möglichkeit zum Importieren von 3D-Modelle ist in der aktuellen Version 3.1 nicht vorhanden, aber bereits in der Entwicklung (48). Es stehen einige geometrische Primitive zur Verwendung bereit. Neben separaten Knoten für die drei Transformationen, kann auch ein Matrix-Knoten definiert werden, der alle Transformationen vereint. SceneJS unterscheidet außerdem zwischen Knoten für Projektion und *LookAt*-Matrix. Viewports hingegen werden nicht unterstützt.

SceneJS unterstützt verschiedene Mapping-Techniken und kann unterschiedliche Maps in Schichten miteinander kombinieren. Konkret unterstützt werden derzeit Color-, *Alpha-*, Bump-, *Specular-* und *Glow-Mapping*, sowie *Texturatlanten*. In allen Textur-Schichten ist es auch möglich, Videos als Quelle zu verwenden. Für Shadow-Mapping gibt es derzeit keine vorgefertigte Lösung in SceneJS. Als Implementierung eines Shader-Knotens ist darüber hinaus auch Displacement-Mapping möglich. Mit Shader-Knoten können natürlich beliebige Shader erstellt werden. Als Lichtquellen sind Umgebungslichter, gerichtete Lichter und Punktlichter möglich. Animationen können in SceneJS auch durch Morphing erzeugt werden. Es gibt eine Raypicking-Funktion zur Auswahl von Objekten. Häufig verwendete Kno-

ten bestimmter Typen lassen sich in einem Bibliotheksknoten ablegen und daraus instanziieren. Außerdem kündigte der Entwickler an, dass die nächste Version ein Physik-System enthalten soll. (49)

```
var scene = SceneJS.createScene({
  nodes:[
    {
      type:"geometry",
      primitive:"triangles",
      positions:[
        -1, -1, 0,
         1, -1, 0,
         1,  1, 0,
        -1,  1, 0
      ],
      normals:[
        0, 0, 1,
        0, 0, 1,
        0, 0, 1,
        0, 0, 1
      ],
      uv:[
        0, 0,
        1, 0,
        1, 1,
        0, 1
      ],
      indices:[
        0, 1, 2,
        0, 2, 3
      ]
    }
  ]
});
```

Quellcode 7: Geometrie aus Vertices in SceneJS

Dokumentation, Beispiele und Community

Eine Dokumentation zu SceneJS gibt es aktuell nicht. Die Basis zur Entwicklung bildet eine relative große Sammlung gut kommentierter Beispiel-Implementierungen.[19] Zusätzlich gibt es auf GitHub ein Wiki[20], in dem vor allem höhere Konzepte des Frameworks beschrieben werden sollen. Eine veraltete Klassen-Referenz[21] kann in einigen Fällen dennoch nützlich sein. Die Communi-

[19] http://scenejs.org/examples.html

[20] https://github.com/xeolabs/scenejs/wiki

[21] http://xeolabs.github.io/scenejs/docs/index.html

ty zu SceneJS ist eher klein. Auf Stack Overflow wurden bisher nur 15 Fragen zu SceneJS gestellt und 11 davon beantwortet. Das Projekt hat auf GitHub 254 Sterne erhalten und es wurden 38 Forks erstellt. Außer dem Hauptentwickler scheint niemand sonst Änderungen zum Quellcode beizutragen. Neuigkeiten werden über die eigene Webseite[22] und in einer Facebook-Gruppe[23] bekannt gegeben. Über letztere erhält man auch direkten Support vom Entwickler.

Weiterentwicklung und Erweiterungen

SceneJS wird derzeit nur von Lindsay Kay als Hauptentwickler betreut. Er scheint dieser Aufgabe im Zuge seiner normalen Arbeit nachzugehen. Tatsächlich steht also wohl eine Firma hinter dem Projekt. In den letzten drei Monaten wurden eine neue Hauptversion und ein Feature-Update veröffentlicht. Für die kommenden Wochen wird außerdem ein weiteres Update erwartet, das neue Funktionen enthalten wird. (50) Da die Entwickler-Community von SceneJS bisher relativ klein ist, gibt es keine Plug-Ins, die nicht schon Bestandteil des Kern-Frameworks sind. Mit steigender Verbreitung könnte sich dies allerdings schnell ändern, die Architektur ist wegen der bereits beschriebenen Eigenschaft von SceneJS, beliebige Plug-Ins asynchron nachzuladen, gerade prädestiniert dafür.

Weitere Besonderheiten des Frameworks

Wichtigstes Alleinstellungsmerkmal ist sicherlich die JSON-basierte Baumdarstellung der kompletten Szene. Aufteilungen und Referenzen auf einzelne Knoten werden im Prinzip nur aus Gründen der Übersicht und Bequemlichkeit benötigt. Die komfortable XHR-Loading-Mechanik wird zwar wegen der auf Plug-Ins basierenden Architektur benötigt, erleichtert aber darüber hinaus auch die Verwaltung von verwendeten Dateien, wie beispielsweise Texturen.

[22] http://scenejs.org/

[23] https://www.facebook.com/groups/350488973712/

6.2 Implementierung der Demoanwendung

Auch mit SceneJS konnte ich schnell sichtbaren Erfolg erzielen. Der Boden der Szene war inklusive Color- und Bump-Mapping schnell erstellt und dank eines Plug-Ins musste ich die Skybox nicht selbst erzeugen. Beim Erstellen *des Drinking Brids* musste ich jedoch feststellen, dass bei aufwändigeren 3D-Modellen schnell die Übersicht verloren geht. Das Problem liegt vor allem darin, dass die einzelnen Anweisungen nicht prozedural hintereinander angegeben, sondern immer in einen Baum eingefügt werden müssen. Dazu kommt, dass es bisher keine Plug-Ins zum Import von 3D-Modellen gibt. Auch für SceneJS musste ich den Code der Zylinder-Geometrie selbst schreiben; auf Grund der Vorleistung für Processing.js und einer guten Vorlage für Geometrie-Plug-Ins war dies schnell realisiert.

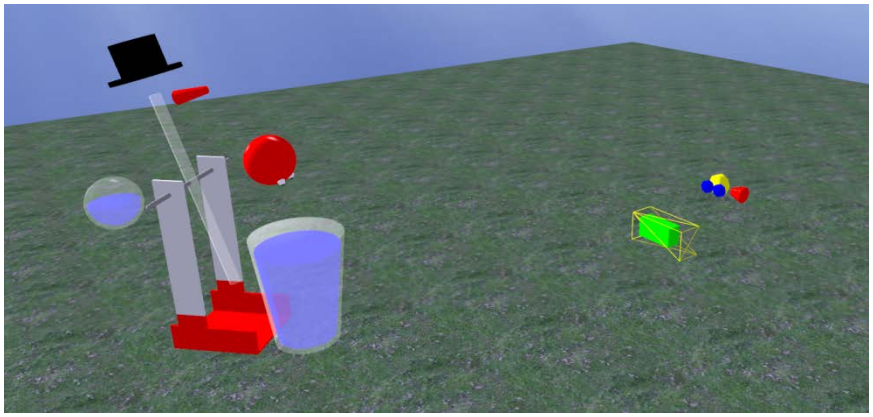

Abbildung 4: Nicht ausgeführte Transformation von Objekten in SceneJS

Als ich zur Animation des Drinking Birds kam, musste ich jedoch feststellen, dass sich ein Teil des Körpers nicht mitdrehte (vgl. Abbildung 4). Da ich keinen Fehler in meiner Implementierung feststellen konnte, wandte ich mich an den Entwickler von SceneJS. Auch wenn zum Zeitpunkt der Veröffentlichung dieser Arbeit der Fehler noch nicht korrigiert wurde, geht der Entwickler von einem Bug im Framework aus. Das gleiche Problem trat später noch beim Verschieben der kleinen Kamera-Modelle auf. Beleuchtung stellt hingegen in SceneJS keine besondere Herausforderung dar; die Auswahl an verfügbaren Lichtquellen ist jedoch vergleichsweise gering. Mit Transparenz hat auch SceneJS Schwierigkeiten (vgl. Abbildung 5). Schatten werden aktuell nicht unterstützt.

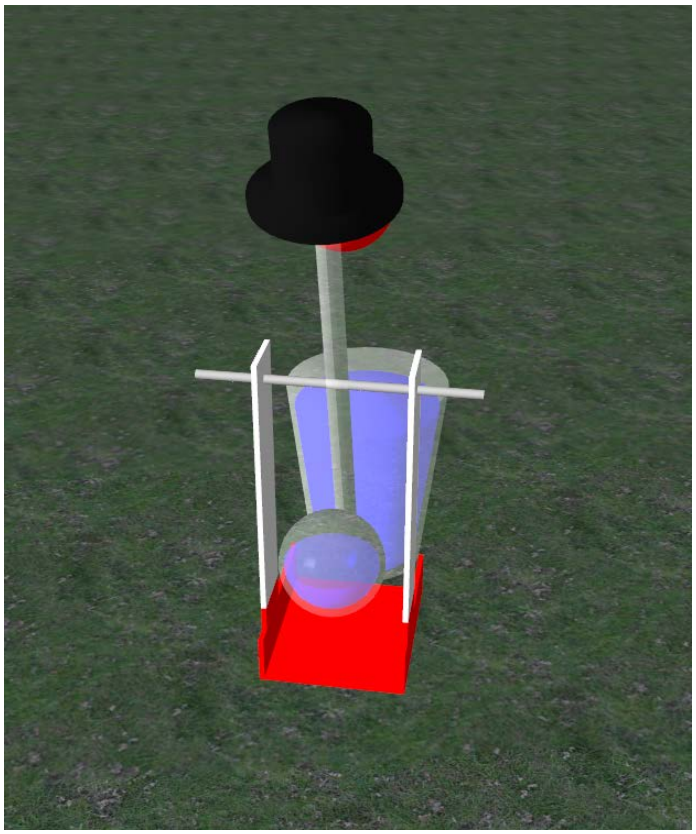

Abbildung 5: Probleme mit Transparenz in SceneJS

Die Kamerasteuerung zu implementieren war problemlos möglich. Ich konnte wiederum auf die Erfahrungen mit den anderen Frameworks zurückgreifen. Für die Positionierung und Ausrichtung der Kamera-Modelle konnte ich einfach auf die Matrix des *LookAt*-Knotens zurückgreifen. Allerdings ist der Zugriff darauf nicht vorgesehen, da die Daten nur in einem, gemäß Namenskonvention als privat gekennzeichneten Objekt, zur Verfügung stehen. Es wäre sinnvoll, wenn hierfür eine öffentliche Methode existieren würde. SceneJS verfügt über einen Raypicking Mechanismus, so dass die Auswahl der Kameras einfach zu implementieren war. Auswählbar sind in SceneJS allerdings nur spezielle *Name*-Knoten, die sonst keine weitere Funktion haben. Da in SceneJS die Unterstützung für Viewports fehlt, musste ich auch in dieser Demoanwendung wieder drei HTML5 Canvas-Elemente mit jeweils eigener Kopie der Szene einsetzen. Das 2D-Interface zur Darstellung der Epipolargeometrie habe ich aufgrund der Erfahrungen mit den anderen beiden Frameworks wieder mit CSS3 umgesetzt.

6.3 Zusammenfassung und Bewertung

SceneJS eignet sich in jedem Fall dafür WebGL-Anwendungen zu entwickeln. Der Funktionsumfang ist ausreichend und durch die Architektur des Frameworks ist es sehr einfach eine Anwendung darauf aufzubauen. Allerdings könnte sich die Baumstruktur im Zusammenspiel mit anderen JavaScript-Programmen als unflexibel erweisen. Das Framework wird aktiv betreut und weitere Neuerungen dürfen erwartet werden. Vielleicht würde eine größere Beteiligung anderer Entwickler neue Ansätze liefern. Unabdingbar ist jedoch, dass die Dokumentation erheblich verbessert wird.

7 Ergebnisse

Mit allen drei Frameworks konnte ich die Demoanwendung in den wichtigsten Punkten umsetzen. Leicht fiel es mir sowohl mit three.js, als auch mit SceneJS, da beide Frameworks eine durchdachte Architektur besitzen. Bei der Implementierung hat sich gezeigt, dass die Menge der Features die unterstützt werden, stark variiert. Eine Übersicht welche Features in welchem Framework verfügbar sind, ist in Tabelle 2 zu sehen. Three.js liegt hier deutlich vorne. Mit der Zeit könnte SceneJS allerdings weiter aufholen. Processing.js leidet in diesem Punkt zum einen unter seiner langsamen Weiterentwicklung, zum anderen unter den Vorgaben der großen Schwester Processing.

	three.js	*Processing.js*	*SceneJS*
Grundlegende Vertexfunktionen	✓	✓	✓
Import von 3D-Modellen	✓	–	–
Color-Mapping	✓	–	✓
Bump-Mapping	✓	–	✓
Shadow-Mapping	✓	–	–
Environment-Mapping	✓	–	–
Transparenz und Blending	✓	(✓)	(✓)
Eigene Shader	✓	–	✓
Grundlegende Beleuchtungsmodelle	✓	✓	✓
Keyframe-Animation	✓	–	–
Rigging	✓	–	–
Morphing	✓	–	✓

✓ Feature ist vorhanden, (✓) Feature vorhanden aber teilweise fehlerhaft

Tabelle 2: Feature-Matrix

Jedes der Frameworks verfolgt konzeptionell einen etwas anderen Ansatz. Besonders konsequent und deshalb lobenswert setzt SceneJS seinen Ansatz der baumbasierten Struktur um. Bei three.js vermisse ich den Mut neue Wege zu gehen und dadurch die

Entwicklung zu vereinfachen. Einen äußerst interessanten Ansatz dazu bietet beispielsweise die three.js Erweiterung tQuery[24], welche versucht die Funktionen von three.js über eine jQuery-ähnliche API bereitzustellen. Die beste Dokumentation gibt es zu Processing.js in Kombination mit der Austausch-Plattform OpenProcessing.org wird man zu jeder Fragestellung fündig – solange man nicht die vorgesehenen Wege verlässt. Auch wenn es derzeit offensichtlich modern ist seine Dokumentation auf Basis von Beispielen zu gestalten, sollten three.js und SceneJS unbedingt ihre klassische Dokumentation verbessern. Alle drei Frameworks haben eine gewisse Community. Sowohl three.js als auch SceneJS werden derzeit mit großer Aktivität weiterentwickelt. Erweiterungen konnte ich nur zu three.js finden.

	three.js	*Processing.js*	*SceneJS*
Konzepte der Softwareentwicklung	+	+	++
Funktionsumfang und Einschränkungen	++	–	+
Demoanwendung	++	–	++
Dokumentation, Beispiele, Community	+	++	–
Weiterentwicklung und Erweiterungen	++	–	+
Weitere Besonderheiten des Frameworks	+	+	++

++ sehr gut, + akzeptabel, – mangelhaft

Tabelle 3: Vergleich der Frameworks

Demnach sind alle drei Frameworks dazu geeignet bei der Entwicklung einer WebGL-Anwendung zu helfen (vgl. Tabelle 3). Es ist jedoch unerlässlich die eigenen Anforderungen zu analysieren und basierend darauf das passende Framework auszuwählen.

[24] http://jeromeetienne.github.io/tquery/

8 Fazit

Diese Arbeit zeigt, dass die Entwicklung von professionellen WebGL-Anwendungen durch den Einsatz moderner Frameworks erleichtert werden kann. Ich konnte darlegen, wo ihre jeweiligen Stärken und Schwächen liegen und welche Schritte nötig wären um eine Verbesserung zu erzielen. Meine Betrachtungen können dabei helfen, das aus heutiger Sicht am besten geeignete Framework für eine 3D-Computergrafik-Anwendnung im Web auszuwählen. Dennoch befinden sich alle drei besprochenen Frameworks noch am Anfang ihrer Entwicklung. Auf Desktopsystemen brauchte die 3D-Computergrafik Jahrzehnte, um ihre Möglichkeiten annähernd auszunutzen. Es ist daher davon auszugehen, dass auch WebGL-Anwendungen erst in einigen Jahren ihr volles Potential entfalten werden.

Die zugehörige Technik wird sich weiter entwickeln; Browser werden JavaScript noch schneller interpretieren können; die Geschwindigkeit von Internetverbindungen wird weiterhin steigen. Smartphones und Tablets sind schon heute die prägende Plattform im privaten Bereich und WebGL kann auch auf ihnen Erfolg haben. Mit neuen Formen der Eingabe, wie Touchbedienung, wird eine neue Interaktion mit 3D-Anwendungen möglich und innovative Geräte wie das *Leap Motion*[25] zeigen auf, in welche Richtung die Entwicklung in Zukunft gehen kann. WebGL ist auf dem besten Weg, ein entscheidender Teil davon zu sein.

[25] https://www.leapmotion.com/

Literaturverzeichnis

1. **Sevo, Daniel.** *HISTORY OF COMPUTER GRAPHICS.* [Online] 1 2005. [Zitat vom: 25. 8 2013.] http://hem.passagen.se/des/hocg/hocg_1960.htm.

2. **Cormode, Graham und Krishnamurthy, Balachander.** First Monday. *Key differences between Web 1.0 and Web 2.0.* [Online] 2. 6 2008. [Zitat vom: 26. 8 2013.] http://journals.uic.edu/ojs/index.php/fm/article/view/2125/1972.

3. **CERN.** *First International Conference on the World-Wide Web.* [Online] 1994. [Zitat vom: 19. 8 2013.] http://www94.web.cern.ch/WWW94/Welcome0522.html.

4. **Raggett, David.** *Extending WWW to support Platform Independent Virtual Reality.* [Online] 1994. [Zitat vom: 19. 8 2013.] http://www.w3.org/People/Raggett/vrml/vrml.html.

5. **Bell, Gavin, Parisi, Anthony und Pesce, Mark.** *The Virtual Reality Modeling Language Version 1.0 Specification.* [Online] [Zitat vom: 19. 8 2013.] http://www.web3d.org/x3d/specifications/vrml/VRML1.0/index.html.

6. **The VRML Consortium Incorporated.** *The Virtual Reality Modeling Language, International Standard ISO/IEC 14772-1:1997.* [Online] 1997. [Zitat vom: 19. 8 2013.] http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-VRML97/.

7. **Mirapaul, Matthew.** The New York Times. *Three-Dimensional Space Is the Next Frontier for the Internet.* [Online] 5. 10 2000. [Zitat vom: 19. 8 2013.] http://www.nytimes.com/2000/10/05/technology/three-dimensional-space-is-the-next-frontier-for-the-internet.html?pagewanted=all&src=pm.

8. **Web3D Consoritum.** *X3D Specifications, Encodings and Language Bindings.* [Online] [Zitat vom: 19. 8 2013.] http://www.web3d.org/x3d/specifications/#x3d-spec.

9. **National Institute of Standards and Technology.** VRML Plugin and Browser Detector. [Online] [Zitat vom: 19. 8 2013.] http://cic.nist.gov/vrml/vbdetect.html.

10. **Adobe Systems Incorporated.** *Shockwave content reaches 41% of Internet viewers.* [Online] 7 2011. [Zitat vom: 19. 8 2013.] http://www.adobe.com/products/shockwaveplayer/shockwaveplayerstatistics.html.

11. **DeLoura, Mark.** *Game Engines and Middleware.* [Online] 6. 9 2011. [Zitat vom: 19. 8 2013.] http://de.slideshare.net/markdeloura/game-engines-and-middleware-2011.

12. **Khronos Group.** OpenGL Wiki. *History of OpenGL.* [Online] [Zitat vom: 26. 8 2013.] http://www.opengl.org/wiki/History_of_OpenGL.

13. —. Khronos press release. *OpenGL ARB to Pass Control of OpenGL Specification to Khronos Group.* [Online] 31. 7 2006. [Zitat vom: 26. 8 2013.] https://www.khronos.org/news/press/opengl_arb_to_pass_control_of_opengl_specification_to_khronos_group.

14. **Segal, Marc und Akeley, Kurt.** *The Design of the OpenGL Graphics Interface.* [Online] 1994. [Zitat vom: 26. 8 2013.] http://www.graphics.stanford.edu/courses/cs448a-01-fall/design_opengl.pdf.

15. **Leech, Jon.** *How to Create Khronos API Extensions.* [Online] 13. 8 2006. [Zitat vom: 26. 8 2013.] http://www.opengl.org/registry/doc/rules.html.

16. **Abi-Chahla, Fedy.** Tom's Hardware. *OpenGL 3 & DirectX 11: The War Is Over.* [Online] 16. 9 2008. [Zitat vom: 26. 8 2013.] http://www.tomshardware.com/reviews/opengl-directx,2019-2.html.

17. **Khronos Group.** Khronos press release. *Khronos Releases OpenGL 4.4 Specification.* [Online] 22. 7 2013. [Zitat vom: 19. 8 2013.] https://www.khronos.org/news/press/khronos-releases-opengl-4.4-specification.

18. **Klaß, Christian.** golem.de. *OpenGL 3.3 und 4.0 - tschüss DirectX 11?* [Online] 11. 3 2010. [Zitat vom: 19. 8 2013.] http://www.golem.de/1003/73785.html.

19. **Khronos Group.** *OpenGL ES Common/Common-Lite Profile Specification.* [Online] 2002. [Zitat vom: 26. 8 2013.] http://www.khronos.org/registry/gles/specs/1.0/opengles_spec_1_0.pdf.

20. —. *OpenGL ES Common Profile Specification Version 2.0.25.* [Online] 2010. [Zitat vom: 26. 8 2013.] http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf.

21. —. *OpenGL ES Version 3.0.2.* [Online] 8. 4 2013. [Zitat vom: 26. 8 2013.] http://www.khronos.org/registry/gles/specs/3.0/es_spec_3.0.2.pdf.

22. **Boyles, Joshua.** OSNews.com. *The State of Linux Gaming 2011.* [Online] 14. 11 2011. [Zitat vom: 26. 8 2013.] http://www.osnews.com/story/25328/The_State_of_Linux_Gaming_2011.

23. **Beier, Andreas.** heise online. *Firefox mit anfänglicher WebGL-Anbindung.* [Online] 20. 9 2009. [Zitat vom: 26. 8 2013.] http://www.heise.de/newsticker/meldung/Firefox-mit-anfaenglicher-WebGL-Anbindung-789376.html.

24. **Khronos Group.** Khronos press release. *Khronos Releases Final WebGL 1.0 Specification.* [Online] 3. 3 2011. [Zitat vom: 26. 8 2013.] https://www.khronos.org/news/press/2011/03.

25. **Ihlenfeld, Jens.** golem.de. *WebGL 1.0 ist fertig.* [Online] 4. 3 2001. [Zitat vom: 26. 8 2013.] http://www.golem.de/1103/81890.html.

26. —. golem.de. *Vollständige Implementierung von OpenGL ES 2.0 für Windows.* [Online] 29. 11 2011. [Zitat vom: 26. 8 2013.] http://www.golem.de/1111/88074.html.

27. **Deveria, Alexis.** Can I use WebGL? [Online] 2013. [Zitat vom: 20. 8 2013.] http://caniuse.com/webgl.

28. **Johnson, Ralph E. und Foote, Brian.** Designing Reusable Classes. *Journal of Object-Oriented Programming.* June/July 1988, 1988, Bd. 1, 2, S. 22-35. http://www.laputan.org/drc/drc.html.

29. **Hartley, Richard und Zisserman, Andrew.** *Multiple View Geometry in Computer Vision.* 2. Auflage. Cambridge : Cambridge University Press, 2004. Kapitel 9. ISBN 0521540518.

30. **Haines, Eric.** *Interactive 3D Graphics - Lesson 2: Drinking Bird Blueprints.* [Online] 22. 03 2013. [Zitat vom: 13. 8 2013.] https://www.udacity.com/wiki/cs291/notes#lesson-2-drinking-bird-blueprints.

31. **Stanford Computer Graphics Laboratory.** The Stanford 3D Scanning Repository. [Online] 1994. [Zitat vom: 15. 8 2013.] http://graphics.stanford.edu/data/3Dscanrep/.

32. **Zerbst, Stefan.** *3D Spieleprogrammierung mit DirectX in C/C++.* Braunschweig : s.n., 2002. S. 398 ff. Bd. II. ISBN: 3-8311-3878-8.

33. **Thorp, Jer.** *Processing & Android: Mobile App Development Made (Very) Easy.* [Online] 25. 9 2010. [Zitat vom: 23. 8 2013.] http://blog.blprnt.com/blog/blprnt/processing-android-mobile-app-development-made-very-easy.

34. **Luckybite LLP.** iProcessing. [Online] [Zitat vom: 23. 8 2013.] http://luckybite.com/iprocessing/.

35. **Kay, Lindsay.** Google Groups. *SceneJS › New BioDigital Human.* [Online] 12. 8 2011. [Zitat vom: 25. 8 2013.] https://groups.google.com/forum/#!topic/scenejs/kmHCvaO_uCM.

36. **Khronos Group.** WebGL Public Wiki. *User Contributions - Frameworks.* [Online] [Zitat vom: 23. 8 2013.] http://www.khronos.org/webgl/wiki/User_Contributions#Frameworks.

37. **Cabello, Ricardo.** GitHub.com. *White Paper for Three.js?* [Online] [Zitat vom: 24. 8 2013.] https://github.com/mrdoob/three.js/issues/1960.

38. **Lam, T. Y.** *Hamilton's Quaternions.* [Online] [Zitat vom: 24. 8 2013.] http://math.berkeley.edu/~lam/quat.ps.

39. **Resig, John.** *Processing.js.* [Online] 8. 5 2008. [Zitat vom: 24. 8 2013.] http://ejohn.org/blog/processingjs/.

40. **Processing Team.** *Overview. A short introduction to the Processing software and projects from the community.* [Online] [Zitat vom: 14. 8 2013.] http://processing.org/overview/.

41. **ProcessingTeam.** *Exhibition. A curated collection of projects created with Processing. New software added each month.* [Online] [Zitat vom: 25. 8 2013.] http://processing.org/exhibition/.

42. **Salga, Andor.** Andor Salga's Processing.JS Web IDE. *Obj 2.* [Online] [Zitat vom: 14. 8 2013.] http://matrix.senecac.on.ca/~asalga/pjswebide/index.php?sketchID=58.

43. **Processing.js Team.** processingjs.org. *Transparency.* [Online] [Zitat vom: 25. 8 2013.] http://processingjs.org/learning/basic/transparency/.

44. **Processing Team.** *PROCESSING 2.0b9 (REV 0217) - 18 May 2013.* [Online] [Zitat vom: 24. 8 2013.] https://raw.github.com/processing/processing/master/build/shared/revisions.txt.

45. **@ProcessingOrg.** Twitter. [Online] 20. 8 2013. [Zitat vom: 24. 8 2013.] https://twitter.com/processingOrg/status/369673883084804096.

46. **Processing Team.** Processing Wiki on GitHub. *Library Basics.* [Online] 6 2013. [Zitat vom: 24. 8 2013.] https://github.com/processing/processing/wiki/Library-Basics.

47. **David, Humphrey et al.** Processing.js Issue Tracker. *Improve plugin/addon/library support and mimic p5 style libraries.* [Online] [Zitat vom: 24. 8 2013.] https://processing-js.lighthouseapp.com/projects/41284/tickets/990-improve-pluginaddonlibrary-support-and-mimic-p5-style-libraries.

48. **Kay, Lindsay.** SceneJS Examples. *Importing geometry from Wavefront .OBJ (unfinished).* [Online] [Zitat vom: 25. 8 2013.] http://scenejs.org/examples.html?page=importObj.

49. **xeolabs.** SceneJS Issue Tracker. *Rigid-body physics.* [Online] 22. 8 2013. [Zitat vom: 25. 8 2013.] https://github.com/xeolabs/scenejs/issues/256.

50. —. SceneJS Issue Tracker. *Milestone V3.2*. [Online] [Zitat vom: 25. 8 2013.] https://github.com/xeolabs/scenejs/issues?milestone=5&state=open.

Anlagen

Auf der beigelegten CD befinden sich:

Anlage 1: Eine digitale Kopie dieser Arbeit
Anlage 2: Der Quellcode zu den Implementierungen der Demoanwendung mit Hilfe der genannten Frameworks
Anlage 3: Ein Dokument mit einer Anleitung zur Verwendung des Quellcodes, sowie Hinweisen auf die jeweiligen Urheber von Teilen des Quellcodes