HASSO-PLATTNER-INSTITUT
Fachgebiet Computergrafische Systeme

Analyse und Vergleich von WebGL-Frameworks zum webbasierten Rendering massiver Graphstrukturen

Analysis and Comparison of WebGL Frameworks for Web-Based Rendering of Massive Graph Data Structures

Bachelorarbeit
zur Erlangung des akademischen Grades
"Bachelor of Science"
(B.Sc.)
im Studiengang IT-Systems Engineering
des Hasso-Plattner-Instituts an der
Universität Potsdam

vorgelegt von

Stefan Lehmann

betreut durch
Prof. Dr. Jürgen Döllner
Dr. Johannes Bohnet, Jonas Trümper

Potsdam, Deutschland
29. Juni 2012

Inhaltsverzeichnis

Kapitel 1

Exploration von Software-Analysedaten

Die steigende Komplexität moderner Softwaresysteme und die Notwendigkeit diese Systeme in Teams zu entwickeln, stellt hohe Anforderungen an die von Softwareentwicklern genutzten Werkzeuge. Der Einsatz von Softwarevisualisierung ermöglicht es Entwicklerteams, einen Einblick in komplexe Systeme zu gewinnen und die Qualität der entwickelten Software kontinuierlich zu überprüfen. Besonders in der Wartungsphase des Softwarelebenszyklus ist es wichtig, verschiedene Softwaremetriken zu überblicken und die Entwicklungsaktivität zu steuern. Bei Softwarekomponenten mit hoher Komplexität und Änderungshäufigkeit ist es beispielsweise besonders wahrscheinlich, dass Fehler auftreten und zusätzliche Kosten entstehen. Diese Elemente sollten daher im Fokus der Softwarewartung stehen.

Um das Auffinden risikobehafteter Softwarekomponenten zu erleichtern, wurde im Rahmen eines Bachelorprojekts am Hasso-Plattner-Institut, in Kooperation mit der Potsdamer Software Diagnostics GmbH[1], ein Werkzeug zur Softwarevisualisierung entwickelt. Die fortgeschrittene Verbreitung von Geräten, die *HTML5* und *WebGL* unterstützen, ermöglichte es, die bestehenden Lösungen um webbasierte 3D-Visualisierungsanwendungen zu erweitern. Basierend auf der bestehenden Metrikextraktion von Software Diagnostics wurde ein Graphvisualisierungs-Framework zur Exploration von Software-Analysedaten entwickelt. Diese Visualisierungen können dabei sowohl nativ auf Desktop-Rechnern als auch webbasiert durch eine Client-Server-Architektur genutzt werden.

Das Graphvisualisierungs-Framework wurde in einem Team von sieben Studenten entwickelt. Die Teilnehmer des Bachelorprojekts waren: Maria Graber, Jens Hildebrandt, Katrin Honauer, Stefan Lehmann, Cathleen Ramson, Mandy Roick und Jakob Zwiener.

1.1 Verwandte Arbeiten

Das Graphvisualisierungs-Werkzeug *CodeCrawler* von Demeyer et al. stellt Softwaremetriken durch Position, Größe und Farbe dar [Demeyer et al., 1999]. Die Visualisierung ist zweidimensional.

Perscheid et al. entwickeln das Visualisierungs-Werkzeug *PathMap* zur Darstellung von Softwaremetriken, zum Beispiel der Testabdeckung [Perscheid et al., 2012]. Die Darstellung basiert auf zweidimensionalen Treemaps und bildet Metrikwerte durch Variation der Farbe und Größe der Elemente ab. Der Nutzer kann die dargestellten Metriken auswählen und die Farbskala verändern.

Das Visualisierungs-Framework *VERSO* von Langelier et al. bildet Softwaremetriken wie Kopplung und Kohäsion auf die Farbe, Höhe und Drehung der visualisierten Elemente

[1] http://www.softwarediagnostics.com/ [Abgerufen am 22.06.2012]

ab [Langelier et al., 2005]. Mit Hilfe von VERSO können auch Treemaps zur Visualisierung von Software erstellt werden.

1.2 Anforderungen an das Graphvisualisierungs-Framework

Verschiedene Anforderungen haben die Architektur des Graphvisualisierungs-Frameworks beeinflusst. So soll es möglich sein, aus den entwickelten Komponenten neue Visualisierungsprodukte zu erstellen. Die Komponenten des entwickelten Frameworks müssen daher unabhängig voneinander wiederverwendbar sein. Ferner sollen die Visualisierungen sowohl durch die lokale Installation des Visualisierungswerkzeugs als auch webbasiert im Rahmen einer Client-Server-Architektur nutzbar sein. Die browserbasierte Client-Anwendung soll ohne Plugin umgesetzt werden und damit den flexiblen Einsatz des Visualisierungswerkzeugs auf verschiedenen Betriebssystemen und Gerätekategorien ermöglichen. Darüber hinaus sollte die Anwendung skalierbar in Bezug auf die Menge der zu explorierenden Software-Analysedaten sein und große Softwareprojekte verarbeiten und visualisieren können.

Die spezifischeren Anforderungen an das Graphvisualisierungs-Framework lassen sich in Anforderungen an die Client- und Server-Anwendung sowie an den Entwicklungsprozess kategorisieren. Sie werden im Folgenden vorgestellt.

1.2.1 Anforderungen an die Client-Anwendung

Die Client-Anwendung soll in der Lage sein, interaktive dreidimensionale Graphvisualisierungen im Browser anzuzeigen. Dabei sollen auch große und komplexe Szenen innerhalb weniger Sekunden geladen und angezeigt werden können. Darüber hinaus soll der Nutzer der Anwendung mit der Visualisierung in Echtzeit interagieren und ihre Eigenschaften konfigurieren können.

Die Client-Anwendung soll ohne den Einsatz von Plugins auf möglichst vielen Geräten nutzbar sein. Das Rendering der Graphvisualisierung soll deshalb mit Hilfe von WebGL umgesetzt werden. Dabei soll auch die Einsatzfähigkeit aktueller WebGL-Frameworks evaluiert werden. Der Berechnungsaufwand der Client-Anwendung soll gering gehalten werden, um den Einsatz auf mobilen Endgeräten zu ermöglichen. Ein möglichst großer Teil der Datenaufbereitung und Rendering-Vorverarbeitung soll daher auf dem Server stattfinden.

1.2.2 Anforderungen an die Server-Anwendung

Aus den Anforderungen an den Client ergeben sich Anforderungen an den Server. So kann eine Reduktion der Rechenlast für den Client durch umfangreiche Vorverarbeitung der Ausgabedaten auf dem Server erreicht werden. Außerdem soll das Visualisierungs-Framework unabhängig von Anzahl und Beschaffenheit der Eingabedaten eine einheitliche Datenstruktur erstellen. Als Eingabedaten dienen Geometrie- und Metrikdaten. Die Metrikdaten sollen dynamisch auf die Visualisierungsdimensionen, wie Farbe oder Höhe, abgebildet werden. Die Darstellung soll sowohl webbasiert als auch auf Desktop-Rechnern möglich sein. Für die webbasierte Visualisierung soll der Server eine geeignete Datenstruktur zur Verfügung stellen.

1.2.3 Anforderungen an den Entwicklungsprozess

Das Graphvisualisierungs-Framework wurde den Praktiken des *Extreme Programming (XP)* folgend entwickelt. Um ein stabiles und flexibles System zu garantieren, wurden die Softwarekomponenten testgetrieben entwickelt. Durch eine kontinuierliche Integration der Komponenten wurde ein permanent lauffähiges System gesichert. In der Bachelorarbeit von Jakob Zwiener [Zwiener, 2012] wird näher auf die testgetriebene Entwicklung des Graphvisualisierungs-Frameworks und den Entwicklungsprozess eingegangen.

1.3 Komponenten des Graphvisualisierungs-Frameworks

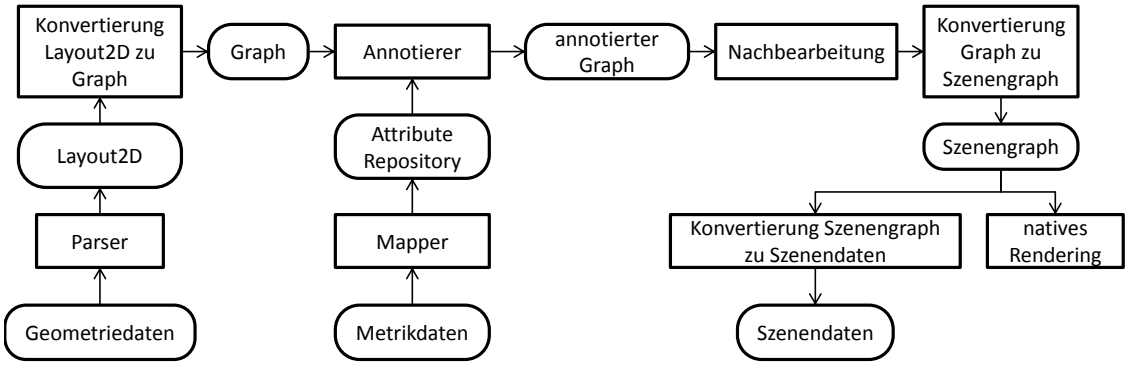

Abbildung 1.1: *Die Komponenten des Graphvisualisierungs-Frameworks bilden eine Pipeline.*

Die Struktur des Graphvisualisierungs-Frameworks ist eine Client-Server-Architektur. Die Server-Anwendung übernimmt den Großteil des Berechnungsaufwands, wodurch die Client-Anwendung entlastet wird. Die Architektur des Servers entspricht einer Pipeline-Struktur. Das heißt, dass die eingehenden Software-Analysedaten sequentiell verarbeitet werden. Hierbei dienen die Ausgaben einer Komponente als Eingaben der jeweils nachfolgenden Komponente. In Abbildung 1.1 wird das Zusammenwirken der Komponenten der Visualisierungs-Pipeline dargestellt.

1.3.1 Komponenten zur Verarbeitung der Eingabedaten

Die Pipeline beginnt mit der Erstellung eines Graphen. Dafür werden die Software-Analysedaten durch Parser eingelesen und als zweidimensionale Geometrieformen gespeichert. Aus diesen Formen wird die Graphdatenstruktur erstellt. In der Bachelorarbeit von Cathleen Ramson [Ramson, 2012] werden die Komponenten zum Aufbau der Graphdatenstruktur detailliert beschrieben.

1.3.2 Komponenten zur Annotation durch Metrikwerte

Im folgenden Schritt wird der erstellte Graph mit Metrikdaten annotiert. Die Metrikdaten werden zuerst auf Attributwerte wie Farbe und Höhe abgebildet. Diese Aufgabe übernimmt ein *Mapper*. Das *AttributeRepository* verwaltet als Depot von Attributen die Abbildung der Metrikdaten und gibt die Attributwerte an den Annotierer. In der Bachelorarbeit von Maria Graber [Graber, 2012] wird die Annotation des Graphen erläutert.

1.3.3 Komponenten zur Erstellung der Ausgabedaten

Im letzten Teil der Pipeline wird der Graph nativ dargestellt oder an den Client zum Rendering im Browser übertragen. Für beide Varianten wird aus dem nachbearbeiteten Graphen ein Szenengraph erstellt. Vor der Übertragung des Szenengraphen an den Client wird dieser in Szenendaten konvertiert. Diese enthalten Listen von Vektoren, Normalen und Farben. In der Bachelorarbeit von Jens Hildebrandt [Hildebrandt, 2012] befinden sich nähere Informationen zur Darstellung des Graphen.

1.3.4 Komponenten der Client-Anwendung

Die Client-Anwendung visualisiert die Szene im Browser. Sie unterteilt sich in eine Graphvisualisierungs- und eine Menükomponente (siehe Abbildung 1.2). Die Komponente zur Graphvisualisierung nutzt WebGL für die Darstellung der vom Server erhaltenen Graphdaten. Sie kann bei Bedarf durch eine Implementierung auf Basis eines WebGL-Frameworks ersetzt werden. In der vorliegenden Arbeit werden verschiedene Frameworks vorgestellt und evaluiert. Die Menükomponente ermöglicht eine Konfiguration der visualisierten Szene. Darüber hinaus zeigt diese Komponente zusätzliche Informationen über die Elemente des Graphen an. In der Bachelorarbeit von Katrin Honauer [Honauer, 2012] wird die Funktionsweise der Client-Anwendung beschrieben.

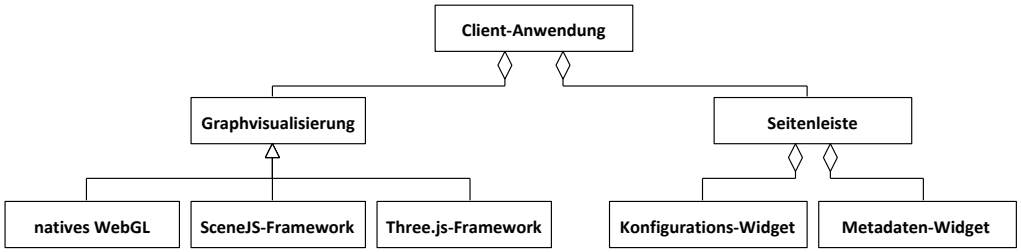

Abbildung 1.2: *Die Client-Anwendung besteht aus einer austauschbaren Visualisierungskomponente und einer aus Widgets bestehenden Seitenleiste.*

1.4 Definitionen

In diesem Abschnitt werden zentrale Begriffe des Graphvisualisierungs-Frameworks definiert.

1.4.1 Definitionen zu Software-Analysedaten

Software-Analysedaten Software-Analysedaten resultieren aus der Analyse von Softwaresystemen und geben Auskunft über Struktur und Verhalten des Systems.

Annotation Bei der Annotation eines Graphen werden Eigenschaften der Knoten oder Kanten verändert und hinzugefügt. Beispielsweise kann ein Knoten mit einer Farbe und einer Höhe annotiert werden.

Softwaremetrik Eine Metrik ist eine Abbildung eines gemessenen Objekts auf einen numerischen Wert anhand einer spezifischen Charakteristik [Lanza & Marinescu, 2006]. Lanza definiert Softwaremetriken als Metriken, die verschiedene Aspekte der Softwareentwicklung messen. Er unterteilt Softwaremetriken in Metriken, die den Entwicklungsprozess betreffen, und Metriken, die Aufschluss über die Beschaffenheit des Softwaresystems geben.

Metadaten/Metainformationen Metadaten sind Zusatzinformationen zu den Dateien des Softwaresystems. Sie enthalten zum Beispiel Metrikwerte oder die Namen der Dateien. Zu einem ausgewählten Knoten können diese Daten auf der Benutzeroberfläche als Metainformationen angezeigt werden.

1.4.2 Definitionen zur Software-Architektur

Framework Ein Framework ist eine Gruppe von kooperierenden Klassen, die ein wiederverwendbares Design für eine bestimmte Softwarefunktionalität anbieten [Gamma et al., 1995].

Pipeline Als Pipeline wird eine Architektur-Form bezeichnet, in der die Komponenten sequentiell angeordnet sind. Das heißt, die Ausgabedaten einer Komponente fungieren als Eingabedaten der nachfolgenden Komponente.

Double Dispatch Mit Hilfe von Double Dispatch können abhängig von der zur Laufzeit ermittelten Typinformation verschiedene Operationen ausgeführt werden [Gamma et al., 1995]. Dazu wird ein Methodenaufruf vom aufgerufenen Objekt zurück zum aufrufenden Objekt reflektiert. Dort können durch Methodenüberladung typabhängige Operationen durchgeführt werden.

Die typspezifische Variante der Operation wird durch Überladung und nicht durch eine Verzweigung bestimmt. Dies erlaubt das Hinzufügen neuer Operationen in dem betreffenden Objekt, ohne die aufrufende Methode anzupassen.

1.4.3 Definitionen zur 3D-Visualisierung

Rendering Als Rendering wird das Berechnen eines zweidimensionalen Bildes aus einer dreidimensionalen Szene bezeichnet [Wright et al., 2007]. Dabei werden die Eigenschaften der Objekte in der 3D-Szene berücksichtigt. So können sich Objekte gegenseitig verdecken. Auch Reflexion, Beleuchtung und Farbe werden bei der Berechnung berücksichtigt.

WebGL Die *Web Graphics Library* (WebGL) ist eine auf OpenGL ES 2.0 basierende Programmierschnittstelle für JavaScript, mit deren Hilfe 3D-Grafiken hardwarebeschleunigt und ohne Plugin in Webbrowsern gerendert werden können [Marrin, 2011]. WebGL ist plattformunabhängig und wird von den aktuellen Versionen der Browser Firefox, Chrome, Opera und Safari sowie von Firefox Mobile und Opera Mobile unterstützt [Deveria, 2012].

Graph im Kontext des Visualisierungs-Frameworks „Ein Graph ist eine Menge von Knoten und eine Menge von Kanten, wobei jede Kante jeweils zwei Knoten verbindet.“ [Sedgewick & Wayne, 2011] Die Graphstruktur des Visualisierungs-Frameworks ist gerichtet und verwaltet Knoten und Kanten in getrennten Datenstrukturen.

Szenengraph Ein Szenengraph ist eine Datenstruktur, die die räumliche und logische Struktur einer graphischen Szene zur effizienten Verwaltung und graphischen Darstellung bereitstellt [Wang & Qian, 2010, Reiners, 2002]. Es handelt sich um eine Baumstruktur, die die Szenenhierarchie abbildet. Die Blätter repräsentieren die Objekte der Szene. Die grafische Darstellung der Objekte wird durch alle Elternknoten beeinflusst. Ein Knoten des Szenengraphen kann eine Gruppe von ähnlichen Objekten zusammenfassen.

OSG OpenSceneGraph ist ein objektorientiertes C++-Framework, welches das Konzept der Szenengraphen nutzt und auf OpenGL aufbaut [Wang & Qian, 2010, Foping et al., 2007]. Es abstrahiert von der systemnahen OpenGL-Schnittstelle und ist dadurch plattformunabhängig. Das Framework kann für Grafik-Anwendungen genutzt werden, die echtzeitfähig arbeiten sollen.

1.5 Vorteile der Verwendung eines WebGL-Frameworks

WebGL basiert auf der OpenGL ES 2.0 Schnittstelle und besitzt ein ebenso geringes Abstraktionslevel [Simpson, 2009]. Dies bringt einerseits eine große Menge an Flexibilität mit sich und erlaubt dem Entwickler fast völlige Freiheit im Umgang mit 3D-Grafiken. Andererseits benötigen WebGL-Anwendungen aber eine aufwendige Initialisierung. Zusätzlich sind viele kleinschrittige Methodenaufrufe notwendig, um das gewünschte Ergebnis zu erhalten. So sind schon für einfachste Beispiele wie das Rendering eines Dreiecks weit mehr als 100 Zeilen Code nötig, wie von Honauer dargestellt [Honauer, 2012]. Die Entwicklung einer WebGL-Anwendung erfolgt dabei auf den unterschiedlichen Abstraktionsebenen von HTML, JavaScript und GLSL, einer C-ähnlichen Sprache, in der die Shader-Programme geschrieben sind.

Inzwischen existiert eine Vielzahl verschiedener WebGL-Frameworks, die von der WebGL-Schnittstelle abstrahieren und somit die Entwicklung von 3D-Webanwendungen deutlich vereinfachen können. Zugleich gelten jedoch viele Frameworks als unausgereift und fehlerhaft. Diese Arbeit untersucht daher die Tauglichkeit ausgewählter, aktueller WebGL-Frameworks zur Implementierung einer webbasierten Anwendung für die interaktive Visualisierung massiver Graphdaten.

In Kapitel 2 sollen Anforderungen an die webbasierte Rendering-Lösung vorgestellt werden. Im Verlauf des Projektes wurden mehrere WebGL-Frameworks untersucht. Kapitel 3 nimmt eine Vorselektierung der betrachteten Frameworks vor und zeigt Schwächen der nicht näher betrachteten Frameworks auf. Neben einer Implementierung der Graphvisualisierung ohne Zuhilfenahme einer Frameworks entstanden im Rahmen des Projektes zwei weitere Implementierungen, die auf die WebGL-Frameworks SceneJS und Three.js basieren. Diese Frameworks werden in den Kapiteln 4 und 5 vorgestellt. Kapitel 6 vergleicht die Implementierungsvarianten unter verschiedenen Gesichtspunkten wie Performanz und Funktionalität. Abschließend bietet Kapitel 7 eine Prognose über die zu erwartende Entwicklung der WebGL-Frameworks.

Kapitel 2

Anforderungen

In den folgenden Kapiteln werden verschiedene WebGL-Frameworks näher betrachtet. Zuvor soll dieses Kapitel einen Überblick über die Anforderungen, die an die Frameworks gestellt werden, aufzeigen.

Die Anforderungen an ein WebGL-Framework zur Visualisierung von Graphdaten lassen sich in fünf Kategorien einteilen. Dazu zählen eine hohe Performanz, Funktionalitätsumfang, grafische Effekte sowie die effiziente Arbeit mit dem Framework. Da die Visualisierung einen Teil einer größeren Webanwendung darstellt, ist eine gute Integration in den bestehenden Code ebenfalls wichtig. Diese Kategorien werden im Folgenden einzeln vorgestellt.

2.1 Hohe Performanz

Das gewählte Framework sollte in der Lage sein, auch große Datenmengen anzeigen zu können. So sind Dreiecksnetze mit 100.000 Dreiecken kein Extrem- sondern der Normalfall.

Auch entsprechende Interaktionen wie das Drehen der Ansicht sollen in Echtzeit möglich sein. Für eine Echtzeitinteraktion mit der Szene ist eine Bildwiederholrate von mindestens 15 Bildern pro Sekunde nötig [Rusdorf, 2008]. Somit sollte ein Rendering-Schritt auch bei großen Datenmengen nicht länger als rund 65 Millisekunden dauern. Angestrebt sind jedoch mindestens 60 Bilder pro Sekunde.

Ein weiteres wichtiges Kriterium ist die initiale Ladezeit, also die Zeit vom Aufrufen der Webseite bis zum ersten Rendern der Visualisierung. Das Framework sollte in der Lage sein, die vom Server generierten Szenendaten mit möglichst wenigen vorhergehenden Konvertierungsschritten durch interne Funktionen in die Buffer auf der Grafikkarte zu schreiben und zu rendern. Die Struktur der Szenendaten wird von Hildebrandt beschrieben [Hildebrandt, 2012]. Eine komplexe eigenhändige Konvertierung in die Framework-spezifische Datenstruktur sollte nach Möglichkeit vermieden werden.

Während der Ausführung der Webanwendung sollte das Framework möglichst wenig Speicher verbrauchen, auch wenn große Datenmengen zur Visualisierung übertragen wurden. Ein geringer Speicherverbrauch ist in Hinblick auf mobile Geräte wichtig, da diese über einen deutlich kleineren Arbeitsspeicher verfügen als ein Desktop-PC. Generell sollte der Browser nicht durch das Erstellen komplexer Strukturen innerhalb des Frameworks zusätzlich belastet werden.

2.2 Integration in den bestehenden Code

Das Framework sollte sich mit möglichst wenigen Änderungen in die bestehende Implementierung integrieren lassen. Dies geschieht an zwei Punkten. Zum einen sollte das Framework möglichst gut an die serverseitige Implementierung des Visualisierungs-Frameworks anschließen, zum anderen sollte es sich ohne größeren Aufwand in die bestehende Client-Anwendung

einfügen.

Zur Anbindung an den Server muss das Framework in der Lage sein, die von Hildebrandt beschriebenen Szenendaten im Json-Format schnell zu verarbeiten und zu rendern [Hildebrandt, 2012]. Die existierende Client-Anwendung ist bereits auf einen Austausch der für das Rendering zuständigen Komponente ausgelegt. Hierzu ist nur ein Austausch der von Honauer beschriebenen *GraphVis*-Komponente notwendig [Honauer, 2012]. Eine Klasse, welche die Schnittstelle dieser Komponente realisiert, sollte mit Hilfe des gewählten Frameworks mit geringem Aufwand erstellt werden können. Zu den wichtigsten Methoden dieser Schnittstelle zählt die Methode *refresh*, die neue Daten für das Rendering entgegennimmt, sowie die Methode *render*, welche einen neuen Rendering-Durchlauf einleitet, beispielsweise nach einer Interaktion mit der Szene. Weiterhin sollten die im folgenden Abschnitt beschriebenen Rückrufmethoden integriert werden können.

2.3 Funktionalität

Neben dem reinen Rendern von Geometrien sollte das Framework noch eine Reihe weiterer Funktionen bereitstellen. So sollte es das Framework ermöglichen, in Echtzeit mit der Szene zu interagieren. Verschiedene Interaktionen wie Drehen, Translieren und Zoomen müssen unterstützt werden. Die Webanwendung soll sowohl auf Desktop-Rechnern als auch auf mobilen Geräten und Tablet-PCs laufen. Daher ist es wichtig, dass die Komponenten, die diese Interaktion realisieren, möglichst austauschbar beziehungsweise konfigurierbar sind.

Zusätzlich zu den Geometrieinformationen erhält der Client weitere Metainformationen, die zusammen mit der ID des Graphknotens auf der Clientseite außerhalb der *GraphVis*-Komponente gespeichert werden. Hierfür stellt die von Honauer beschriebene *Sidebar*-Komponente entsprechende Funktionen bereit [Honauer, 2012]. Mit Hilfe von Picking soll die vom Nutzer ausgewählte Geometrie erkannt und eine entsprechende ID ausgelesen werden. Danach werden bereits integrierte Funktionen der *Sidebar* zum Anzeigen der Metainformationen in Form von *Tooltips* aufgerufen. Das Framework sollte also einerseits eine möglichst performante Implementierung von Picking unterstützen und andererseits entsprechende Rückruffunktionen zum Abrufen von zusätzlichen Informationen bereitstellen.

Weiterhin sollte das Framework in der Lage sein, beliebige Primitive zu rendern, um die Client-Anwendung in die Lage zu versetzen, eine Erweiterung auf neue und noch nicht betrachtete Visualisierungen generell zu ermöglichen. Lediglich vordefinierte Objekte wie Quader und Kugeln bereitzustellen, ist hierfür nicht ausreichend.

2.4 Grafische Effekte

Grundsätzliche grafische Effekte wie Beleuchtung mit Hilfe verschiedener Lichtquellentypen sollten bereits vom Framework unterstützt werden. Für eine realistischere Beleuchtung ist das Unterstützen mehrerer Renderingschritte sowie frei austauschbarer oder konfigurierbarer Shader sinnvoll. Somit können Effekte wie Screen Space Ambient Occlusion, eine effiziente Variante, Objekte anhand ihrer Umgebung zu schattieren, umgesetzt werden [Engel, 2009]. Die zu untersuchenden Frameworks sollten auch im Zuge der gewünschten Erweiterbarkeit des Clients einfaches Anzeigen von Texturen ermöglichen.

2.5 Effiziente Arbeit

Zusätzlich zu den bisher angesprochenen Anforderungen sollte die Arbeit mit dem Framework möglichst einfach von der Hand gehen. So muss das Framework dem Entwickler

eine klare, logische Schnittstelle bieten, um sowohl das Lesen als auch das Schreiben des Quelltextes zu erleichtern. Darüber hinaus erleichtert eine gut strukturierte, vollständige Dokumentation das Arbeiten mit einem Framework. Beispielimplementierungen ermöglichen einen schnellen Einstieg in die Funktionsweise eines Frameworks. Eine zusätzliche, wichtige Anforderung ist, dass der Quellcode des Frameworks offen vorliegen sollte und nicht nur minimiert als JavaScript-Datei verfügbar ist. Dies erleichtert das Beseitigen von auftretenden Fehlern. Weiterhin zu berücksichtigen ist, ob das Framework noch weiterentwickelt wird. Eine aktive Entwicklergemeinschaft ist wünschenswert, falls Fehler im Framework gefunden werden.

Kapitel 3

WebGL-Frameworks im Überblick

Rund um den neuen WebGL-Standard hat sich eine Vielzahl verschiedenster Frameworks gebildet. Dieses Kapitel stellt daher eine Übersicht einiger betrachteter WebGL-Frameworks dar.

Da WebGL selbst noch eine junge Technologie darstellt, sind auch viele zugehörige Frameworks unausgereift. Viele Frameworks sind weit entfernt davon, echte Anwendung zu unterstützen. Die meisten sind lediglich in der Lage kleinere Spielereien darzustellen. Weiterhin haben viele Frameworks ein sehr spezielles Einsatzgebiet und eignen sich somit nicht zur Visualisierung großer Datenmengen. Außerdem setzen verschiedene WebGL-Frameworks auf ganz unterschiedlichen Abstraktionsstufen an. Einige abstrahieren nur gering von der WebGL-Spezifikation und arbeiten auf dem Abstraktionsniveau von OpenGL und GLU, andere bieten jedoch Schnittstellen mit einem höheren Abstraktionsniveau an, zum Beispiel in Form eines Szenengraphen. Ein Szenengraph ist eine Datenstruktur, mit deren Hilfe Objekte logisch innerhalb einer Szenenhierarchie organisiert werden können [Reiners, 2002].

In den folgenden Abschnitten soll daher eine Vorauswahl getroffen werden, welche Frameworks für die Implementierung der *GraphVis*-Komponente geeignet sind, beziehungsweise welche von vornherein nicht zu den in Kapitel 2 erläuterten Anforderungen passen.

3.1 Glow

Glow stellt einen minimalen WebGL-Wrapper dar und besitzt daher nur einen geringen Mehraufwand [Emtinger & Lassus, 2011]. Glow kapselt wichtige WebGL-Funktionalitäten leicht und bietet daher eine etwas schlankere API. Ein Großteil der WebGL-Befehle lässt sich direkt auf Befehle von Glow übertragen, weshalb die mit Glow geschriebene Anwendungen schnell recht umfangreich und unübersichtlich werden können. Der eindeutige Fokus von Glow liegt aber auf dem Gebiet der Shader, wodurch sich einige interessante Post-Rendering-Effekte erzielen lassen, die für unseren Anwendungsfall jedoch irrelevant sind.

3.2 O3D

Bereits vor dem Aufkommen vom WebGL stellte Google mit O3D ein Plugin vor, das hardwarebeschleunigte 3D-Anwendungen im Browser ermöglichte. Kurze Zeit später wurde der Entwurf zu WebGL von Mozilla und der Khronos Group vorgelegt [WebGL Working Group, 2011]. Aufgrund des aufkommenden WebGL-Standards, stellte Google die Weiterentwicklung des O3D-Plugins zugunsten der Unterstützung von WebGL ein [Google O3D Team, 2010b]. Die O3D-Schnittstelle steht jedoch weiterhin als JavaScript-Bibliothek auf Basis von WebGL zur Verfügung [Google O3D Team, 2010a].

Einige Komponenten der Bibliothek wurden nicht in die WebGL-Implementierung überführt, da diese laut Google nicht mit reinem JavaScript umzusetzen seien. Aber auch viele weitere Funktionen wie die eingebaute Matrixbibliothek wurden nicht übernommen, was einen halbfertigen Eindruck hinterlässt. Zudem ist die Dokumentation des mit einem Szenengraphen arbeitenden WebGL-Frameworks nur lückenhaft vorhanden, veraltet und verweist häufig auf die Plugin-Version des Frameworks.

3.3 OSGjs

OSGjs ist ein WebGL-Framework, das auf den Konzepten von Open Scene Graph [Foping et al., 2007] basiert und eine ähnliche Schnittstelle bietet [Pinson, 2011]. Dadurch soll für Entwickler, die bereits Erfahrung im Umgang mit OSG haben, trotz der umfangreichen Schnittstelle eine schnelle Einarbeitung in das Framework möglich sein.

Wie von Hildebrandt beschrieben, wird für die Berechnung der Geometrien auf Serverseite OSG verwendet, was die Möglichkeit bietet, berechnete Szenen im OSG-Dateiformat zu exportieren [Hildebrandt, 2012]. Zusammen mit dem OSGjs-Framework stellt der Entwickler einen Konverter zur Verfügung, der OSG-Datein in ein OSGjs-kompatibles Format überführt. Somit könnte die entsprechende Szene auf dem Client ohne weitere Konvertierungschritte geladen und gerendert werden.

OSGjs ist eines von zahlreichen WebGL-Frameworks, die noch sehr unausgereift und unfertig sind. Darüber hinaus wird es nur von einem einzigen Entwickler erstellt. Der angesprochene Konverter ist darüber hinaus fehlerhaft. Konvertierte OSG-Szenen konnten nicht richtig vom Framework eingelesen werden. Zusätzlich konnten entsprechende Beispiele zur Performanz des Frameworks nicht überzeugen. So konnten bereits etwa 4000 Quads nicht mehr in Echtzeit gerendert werden.

3.4 PhiloGL

PhiloGL wurde von Sencha entwickelt, einem Unternehmen, das bereits einige Frameworks und Entwicklungsumgebungen zur Erstellung mobiler Webanwendungen entwickelt hat [Kaneda, 2011]. Mit PhiloGL unterstützt Sencha die Visualisierung von Daten sowie die Entwicklung von Spielen als Webanwendungen.

PhiloGL kapselt viele WebGL-Funktionen auf einem hohen Abstraktionsniveau. Das Framework bietet dem Nutzer aber eine stets transparente Schnittstelle zu WebGL-Funktionen [Belmonte, 2011]. Besonders stark vereinfacht wurde die Initialisierung von WebGL sowie der Aufbau einer neuen Szene. Bereits bei der Initialisierung werden Interaktionen definiert, die zu einem späteren Zeitpunkt angepasst werden können, was aber gegen die Konventionen des Frameworks spricht.

Bei ersten Testimplementierungen zeigte das Framework jedoch einige Kompatibilitätsprobleme auf. Im Gegensatz zu Implementierungen basierend auf anderen Frameworks liefen die mit PhiloGL entwickelten Anwendungen auf vielen Firefox-Versionen nicht.

Kapitel 4

SceneJS

SceneJS ist ein quelloffenes WebGL-Framework für 3D-Anwendungen im Browser. Das Framework wurde für die Entwicklung von Anwendungen verschiedener interaktiver Visualisierungen aus den Bereichen Medizin, Architektur sowie Maschinenbau konzipiert [Kay, 2009]. Es fokussiert daher auf das effiziente Rendern komplexer Szenen mit vielen individuell auswählbaren Objekten.

Im Gegensatz zu vielen anderen WebGL-Frameworks, die auf aufwendige Effekte wie Schatten und Reflexionen mit Hilfe von mehreren Rendering-Durchläufen ausgelegt sind, beschränkt sich SceneJS auf einen einzigen optimierten Rendering-Durchlauf [Kay, 2011a]. Das Framework selbst basiert auf einer Szenengraphstruktur. Jegliche Methoden arbeiten dabei auf den Knoten des Szenengraphen. Dem Entwickler bietet SceneJS eine schlanke, auf Json basierende Schnittstelle.

Die folgenden Betrachtungen beziehen sich auf die Beta Version 2.0 des Frameworks, welche seit November 2011 frei verfügbar ist [Kay, 2010]. Im folgenden Abschnitt soll zunächst die Benutzung des SceneJS-Frameworks veranschaulicht werden. Es soll geklärt werden, ob das Framework im normalen Gebrauch die gestellten Anforderungen hinsichtlich Interaktivität und Performance erfüllen kann. Weiterhin sollen Optimierungsmöglichkeiten sowie daraus folgende Probleme aufgezeigt werden. Am Ende des Kapitels wird SceneJS hinsichtlich der Tauglichkeit in Bezug auf die in Kapitel 2 erläuterten Anforderungen kurz evaluiert.

4.1 Szenengraphbasierte Implementierung

In diesem Abschnitt soll veranschaulicht werden, wie das SceneJS-Framework verwendet wird. SceneJS arbeitet zwar intern mit einem objektorientierten Szenengraphsystem, stellt aber nach außen eine prozedurale Schnittstelle bereit. Diese ist äußerst schlank und nutzt hauptsächlich Objekte in Json-Format als Parameter. Das Framework bietet einen globalen *SceneJS*-Namensraum an, welcher sämtliche Funktionen des SceneJS-Frameworks kapselt. Dazu gehören Methoden zur Erstellung einer Szene und deren Manipulation sowie entsprechende Interaktionen. Wie diese genutzt werden, sollen die folgenden Abschnitte veranschaulichen.

4.1.1 Aufbau eines Szenengraphen in SceneJS

Um eine neue Szene zu erstellen, wird die Funktion *SceneJS.createScene* genutzt. Diese nimmt als einziges Argument ein Json entgegen, welches die gesamte, zu rendernde Szene beschreibt. Als Ergebnis dieser Methode entsteht ein Szenengraph wie beispielhaft in Abbildung 4.1 dargestellt. Das Json ist dabei ähnlich dem zu erstellenden Szenengraphen aufgebaut. Die einzelnen Elemente des Jsons werden durch SceneJS in verschiedene, verschachtelte Knotenobjekte des späteren Szenengraphen konvertiert.

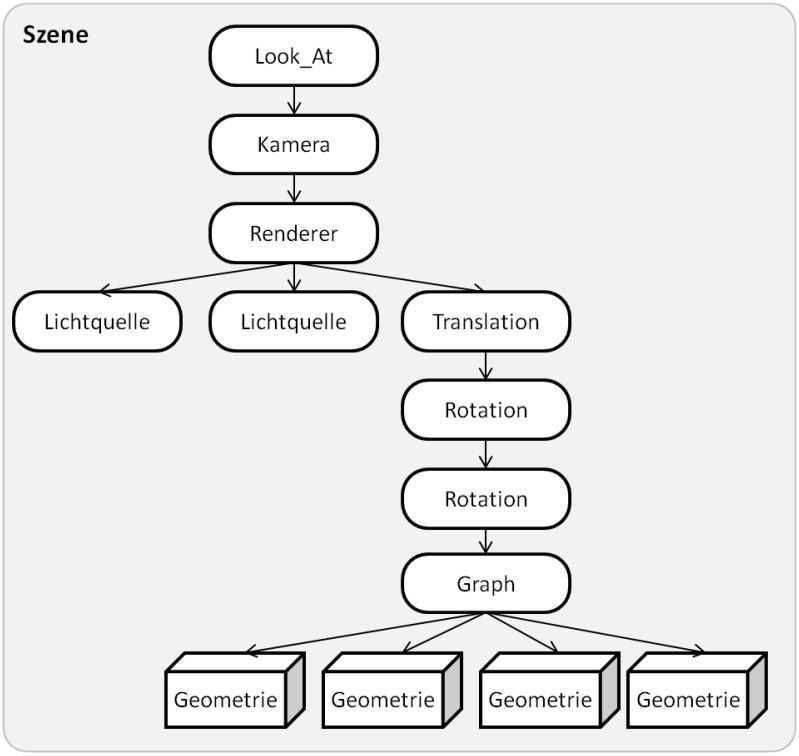

Abbildung 4.1: *Verschiedene Knotentypen mit spezialisierten Aufgaben werden zum Aufbau eines Szenengraphen in SceneJS benötigt.*

SceneJS verfügt über einen Basisknotentypen, welcher grundlegende Funktionalitäten für Szenengraphobjekte bereitstellt. Hierzu zählen unter anderem Knotenidentität sowie die Möglichkeit, Kindknoten zu definieren. Zusätzlich wird es ermöglicht, über den Subgraphen eines Knoten, sowie über alle Elternknoten bis zur Wurzel des Szenengraphen zu iterieren.

Jedes Element des Jsons, das in einen Szenengraphknoten konvertiert werden soll, muss bestimmte vordefinierte Attribute gesetzt haben. So zum Beispiel definiert das *type*-Attribut, in welchen konkreten Knotentyp umgewandelt wird. Zusätzlich verfügt jedes Element im Json über ein *nodes*-Attribut, einem Array, in dem die Kindknoten gespeichert werden. Darüber hinaus muss jedem Knoten, auf den später zugegriffen wird, eine ID hinzugefügt werden. Somit ergibt sich für ein Element des szenenbeschreibenden Jsons die in Quelltext 4.1 gezeigte Struktur.

```
1 {
2   type: "nodeType",
3   id: "nodeId",
4   nodes: [
5     // Kindknoten
6   ]
7 }
```

Quelltext 4.1: *Der Aufbau der Json-Elemente ist strikt vorgegeben. Knotentyp und Kindelemente müssen zwingend gesetzt werden, optional kann eine ID vergeben werden.*

Abhängig vom jeweiligen Knotentyp müssen weitere Felder im Json gesetzt werden. Die Eigenschaften eines Knotens beeinflussen dabei die Knoten des entsprechenden Subgraphen.

Wie in Abbildung 4.1 skizziert, ist bereits eine größere Menge verschiedener Knotentypen nötig, um eine grundlegende Szene in SceneJS zu beschreiben. So wird im obersten Element

des Jsons die Szene an sich beschrieben. Die Szene verfügt über eine eindeutige ID, für den Fall, dass mehrere Szenen gleichzeitig auf einer HTML-Seite verwaltet werden müssen. Über das Attribut *canvasId* wird das HTML5 Canvas-Element referenziert, in das die Szene gerendert werden soll.

Als einziger Wurzelknoten der Szene wird ein Knoten mit dem Typ *look_At* benutzt. Dieser stellt die Matrix für die Transformation der Szene vom Welt- in das Kamerakoordinatensystem bereit. Hierzu kann der Nutzer des Frameworks über die Attribute *eye* und *look* die Position der Kamera beziehungsweise des betrachteten Punktes spezifizieren. Matrizenrechnung wird dem Nutzer also vom Framework abgenommen.

Der darunterliegende Kameraknoten definiert Parameter für die perspektivische Projektion. Das Framework unterstützt sowohl perspektivische als auch orthographische Projektion. Im Falle der perspektivischen Projektion muss zusätzlich das Kamerafrustum angegeben werden.

Unterhalb des Kameraknotens befindet sich in der Szenenhierarchie ein Knoten vom Typ *renderer*. Über die Schnittstelle dieses Knotens lassen sich Konfigurationen am WebGL-Zustand vornehmen. So wird hier beispielsweise das Setzen der Hintergrundfarbe ermöglicht.

Ohne Licht werden Geometrien in SceneJS lediglich in Schwarz dargestellt. Deshalb ist es notwendig, ein oder mehrere Lichtquellen in die Szene zu integrieren. Dies geschieht über Knoten des Typs *light*. Momentan werden zwei unterschiedliche Typen von Lichtquellen von SceneJS unterstützt: Punktlichtquellen und gerichtete Lichtquellen.

Um eine Interaktion mit der Szene zu ermöglichen, ist es notwendig, Translations- beziehungsweise Rotationsknoten in den Szenengraphen einzuhängen. Die durch diese Knoten beschriebenen Transformationen werden in eine entsprechende Matrix konvertiert, die dann auf den zum Knoten gehörenden Subgraphen angewandt wird. In Abschnitt 4.1.3 wird geklärt, wie genau diese Rotations- und Translationsknoten für die Interaktion mit der Szene genutzt werden.

Als Elternknoten für jegliche Geometrien der Graphvisualisierung wird ein Knoten des Basisknotentyps *node* benutzt. Dieser dient dazu, einen Knoten im Graphen zu referenzieren, an den Geometrien als Kindobjekte angehangen werden können. Hierzu ist wie bei den Rotations- und Translationsknoten wieder die Angabe einer eindeutigen ID notwendig.

4.1.2 Integration der Geometrien in den Szenengraphen

Der Aufbau des Szenengrundgerüsts ist damit abgeschlossen. Die Client-Anwendung kann nun, wie von Honauer beschrieben, entsprechende Szenendaten vom Server anfordern [Honauer, 2012]. Zu Testzwecken variiert die Datenstruktur leicht von den in Hildebrandt beschriebenen Szenendaten [Hildebrandt, 2012], um sich optimal in die Json-Schnittstelle des Frameworks einzugliedern. So wurde die Graphdatenstruktur beispielsweise darauf ausgelegt, das Iterieren über Szenengraphobjekte zu vereinfachen.

Zum Erstellen und Hinzufügen neuer Knotenobjekte stellt der Basisknotentyp *SceneJS.Node* die Methode *add* bereit, mit der sich neue Objekte in den Subgraphen des Knotens einfügen lassen. Als Parameter wird unter anderem ein Json erwartet, welches das neu zu erstellende Objekt beschreibt. Mittels der Methode *SceneJS.scene* lässt sich über die entsprechende Szenen-ID die Szene finden, in die neue Knoten eingefügt werden sollen. Der Elternknoten der einzufügenden Geometrie wird durch die *findNode*-Methode der Szene unter Angabe der ID des Knoten zurückgegeben. Durch Iteration über die vom Server generierten Graphdaten werden die übermittelten Geometrieobjekte dem Graphen hinzugefügt. Hierzu werden Knoten des Typs *geometry* verwendet. Zur späteren Identifikation kann die ID aus den Graphdaten in das gleichnamige Attribut übernommen werden. Im Falle der Dreiecksgeometrien müssen zusätzlich die Koordinaten der Punkte, die Normalen

```
1 function translate(x, y, z)
2 {
3   this._translationNode.inc({x: x, y: y, z: z});
4   this._graphvis.render();
5 };
```

Quelltext 4.2: *Mit Hilfe der translate-Methode wird der Subgraph des Translationsknotens um die angegebenen Parameter verschoben. Nach einer Änderung des Szenengraphen muss dieser neu gerendert werden.*

sowie Farben angegeben werden. Wurden alle Attribute des Knotens definiert, kann dieser dem Graphen hinzugefügt werden. Die bisher beschriebenen Knotentypen sind ausreichend, um einen kompletten Szenengraphen zu erstellen. Der Aufbau des Szenengraphen ist in Abbildung 4.1 skizziert. Zum Rendern der Szene wird die Methode *renderFrame* verwendet. Weiterhin verfügt SceneJS über eine eingebaute Rendering-Schleife, die unter anderem Rückruffunktionalitäten für Interaktionen bereitstellt. Auf diese Variante des Renderings wird jedoch verzichtet, da so nach einem browserspezifischen Zeitintervall ein neuer Rendering-Schritt angestoßen wird. Um die Client-Anwendung möglichst zu entlasten, soll jedoch nur dann ein neuer Rendering-Schritt durchgeführt werden, wenn eine entsprechende Interaktion durch den Nutzer getätigt wurde.

4.1.3 Interaktion mit der Szene

SceneJS bietet dem Nutzer keine vorgefertigte Klasse zur Szenenmanipulation. Mit wenigen Zeilen Code kann diese aber selbst definiert werden. Um möglichst viel Funktionalität der in Honauer beschriebenen Client-Anwendung zu verwenden, werden die *EventLister*-Klassen des Clients wiederverwendet [Honauer, 2012]. Diese fangen die konkreten, geräteabhängigen Nutzerinteraktionen ab und leiten diese standardisiert weiter. Somit kann die Klasse zur Szenenmanipulation von den konkreten Events, die spezifisch für das jeweilige Gerät sind, abstrahieren. Die Klasse muss dazu lediglich die Schnittstelle des Szenenmanipulators implementieren. Eine Nutzerinteraktion wird in SceneJS durch einfaches Ändern der Attribute der Translations- und Rotationsknoten erreicht. Hierzu bieten die Knoten zur Modelltransformation spezifische Methoden an, wie in Quelltext 4.2 gezeigt. Vor dem erneuten Rendering der Szene wird der Szenengraph neu berechnet und so Änderungen an den Transformationsmatrizen vorgenommen.

4.1.4 Picking zum Auslesen von Metadaten

Das Anzeigen und Speichern der Metadaten wird von der in Honauer beschriebenen Komponente *Sidebar* übernommen [Honauer, 2012]. Die einzige notwendige Information, die diese Komponente dazu benötigt, ist die ID der ausgewählten Geometrie. Diese lässt sich durch Picking ermitteln.

Picking wird in SceneJS bereits per Farbkodierungs-Verfahren unterstützt [Kay, 2011b]. Somit ist die Performanz dieser Implementierungsform von Picking unabhängig von der Anzahl der Objekte, die sich in der Szene befinden. Hierzu werden Knoten vom Typ *name* verwendet. Dieser Knotentyp macht alle Knoten seines Subgraphen mit Hilfe von Picking auswählbar. Auf jedem Namensknoten muss das Attribut *name* gesetzt haben. Jedem Namen wird dann eine Farbe zugeordnet und diese für die entsprechenden Geometrien in einen speziellen Pick-Buffer geschrieben, wie in Abbildung 4.2 veranschaulicht. Dieser Pick-Buffer wird in einem zusätzlichen Rendering-Schritt als Farb-Buffer verwendet. Die

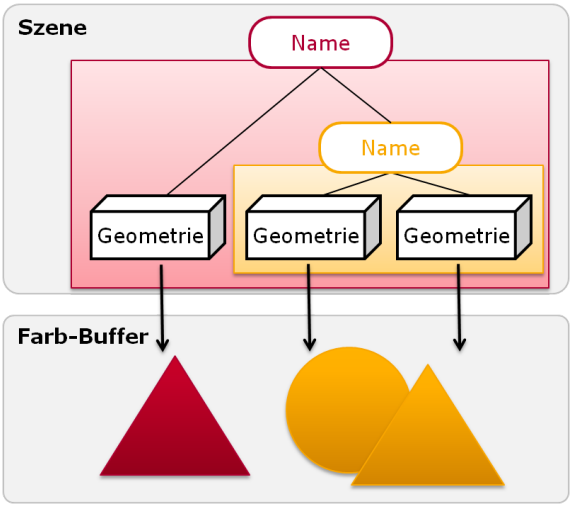

Abbildung 4.2: *Namen werden Farben zugeordnet und in einen zusätzlichen Farb-Buffer geschrieben. Innere Namesknoten überschreiben dabei äußere.*

Szene wird mit diesen Farben in eine Textur gerendert und bleibt dort gespeichert, bis ein neuer Rendering-Vorgang durchgeführt wird.

Um Picking im Szenengraphen zu ermöglichen und Metadaten auslesen zu können, muss oberhalb jedes Geometrieknotens ein Namensknoten mit der entsprechenden ID der Geometrie als Name in den Szenengraphen eingefügt werden. Mit Hilfe der Methode *pick* der Szene kann nun der Name der Geometrie ermittelt werden, die sich mit den angegebenen Koordinaten überschneidet. Der Name beziehungsweise die ID der ausgewählten Geometrie wird nun an weitere Komponenten zur Anzeige der Zusatzinformationen weitergeleitet.

4.2 Optimierung durch Verwendung eines einzelnen Geometrieobjektes

Erste Tests zeigen, dass die in Abschnitt 4.1 vorgestellte Implementierung schlecht auf größere Datenmengen skaliert. So konnte bereits mit einer Szene mit 10.000 Szenenobjekten nicht mehr in Echtzeit interagiert werden. Daher soll im folgenden Abschnitt eine mögliche Optimierung vorgestellt werden, welche eine höhere Performance verspricht.

Um das Traversieren des Szenengraphen zu beschleunigen, könnte statt mehreren Geometrieknoten ein einziger, komponierter Knoten verwendet werden, der sämtliche Geometrien eines Primitivs enthält. Dieser Ansatz lässt sich darüber hinaus gut in das bestehende System integrieren, da die vom Server generierten Graphdaten somit direkt vom SceneJS-Framework entgegen genommen werden können und keiner weiteren Konvertierung bedürfen. Somit lässt sich die Json-Schnittstelle optimal nutzen.

Für den grundsätzlichen Aufbau der Szenenhierarchie sowie der Interaktion mit der entstandenen Szene kann die Implementierung aus Abschnitt 4.1.1 übernommen werden. Beim Einhängen der Szenengeometrie wird nur noch ein einziger Geometrieknoten an den dafür vorgesehenen Knoten mit der ID *parentNode* verwendet.

Dieser Ansatz führt jedoch zu zwei Problemen, die im Folgenden erläutert sowie mögliche Lösungen vorgeschlagen werden sollen. Zum einen können größere Szenenstrukturen nicht vollständig angezeigt werden. Dies liegt daran, dass das verwendete Index-Array von SceneJS intern in ein UInt16Array konvertiert wird. Dessen Elemente besitzen nur einen Wertebereich von 0 bis 65.535. Dadurch lassen sich maximal 65.536 verschiedene

Vertices adressieren. Das Ergebnis einer Wertebereichsüberschreitung zeigt Abbildung 4.3. Um dieses Problem zu lösen, müssen die vom Server generierten Graphdaten in mehrere

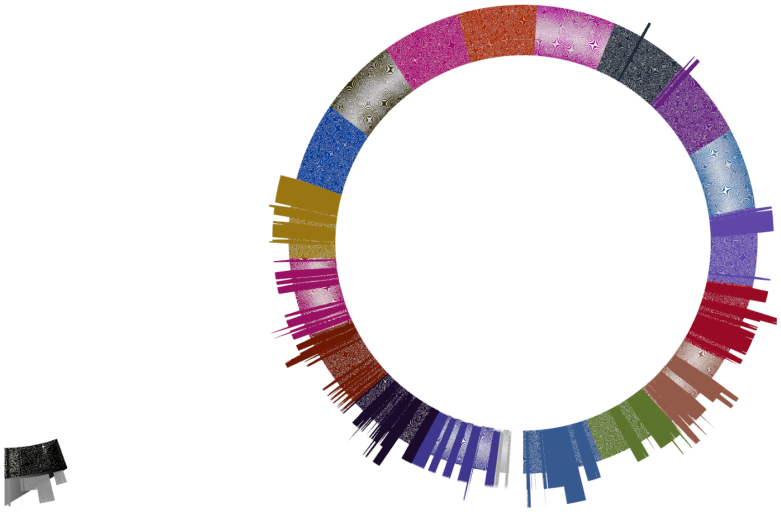

Abbildung 4.3: *Zwei Darstellungen einer BundleView in SceneJS mit mehr als 400.000 Dreiecken. Mehr als 65.536 verschiedene Vertices können nicht referenziert werden (links). Durch die Verwendung mehrerer Geometrieknoten wird die gesamte BundleView angezeigt. Diese einzelnen Teile sind hier farbkodiert dargestellt (rechts).*

Teile zerlegt werden, die jeweils maximal 65.536 Vertices umfassen. Danach kann ein entsprechendes Index-Array erstellt und die Teile der Szene jeweils als Geometrieobjekte mit diesem Index-Array aufgebaut werden. Abbildung 4.3 zeigt die Aufteilung einer BundleView in mehrere Szenenobjekte anhand von Farben. Zum anderen ist Picking, wie es im vorangegangenen Kapitel beschrieben wurde, nicht mehr möglich. Da ein Namensknoten jeweils seinem gesamten Subgraphen einen Namen zuweist, kann nicht zwischen einzelnen Objekten innerhalb eines Geometrieknotens unterschieden werden. Zwar bietet SceneJS einen *Framebuffer*-Knoten, der es ermöglicht, selbst eine Textur für ein Farbkodierungs-Verfahren mit eigenen Farben zu rendern. Diese wird jedoch vom Framework gekapselt und kann nur mit Hilfe eines speziellen Texturknotens als Textur innerhalb einer Szene verwendet werden. Von außen kann jedoch nicht auf diese Textur zugegriffen werden. Diese Funktionalität soll jedoch in zukünftigen Versionen des Frameworks bereitstehen [Kay, 2011c].

4.3 Evaluierung von SceneJS

Dieses Kapitel soll den Eindruck, den das SceneJS-Framework vermittelt, zusammenfassen. Der Vergleich zu anderen Implementierungen findet sich in Kapitel 6.

Die Implementierung einer Visualisierung massiver Graphdaten wird durch SceneJS vereinfacht. Der Aufbau einer neuen Szene verläuft in SceneJS strikt nach einem vorgegebenen Muster und lässt dem Entwickler kaum Freiheiten. So ist es beispielsweise nicht möglich, neue Knoten zu erstellen, ohne sie gleich in einen Szenengraphen einhängen zu müssen. Aufgrund der Verwendung des Json-Formates zur Beschreibung der Szene wird die Initialisierung einer SceneJS-Anwendung schnell groß und unübersichtlich. Da der Szenenaufbau immer nach dem gleichen strengen Muster abläuft, vereinfacht dies die Einarbeitung in das Framework.

	SceneJS
aktuelle Version	Beta 2.0 (15. November 2011)
erste Version	Alpha 0.7.0 (25. März 2010)
weitere wichtige Versionen	Beta 0.8.0 (13. Mai 2011)
Anzahl aktiver Entwickler	5
Entwicklung seit	2009
Größe der Datei (minimiert)	337 KB
Dokumentation	stark lückenhafte Dokumentation, lediglich Kernkomponenten dokumentiert, wenige Beispiele
Unterstützte grafische Primitive	Punkte, Dreiecke, Dreieckslinien, Dreiecksfächer, Linien, Linienzüge
Picking-Verfahren	Farbkodierung
Szenengraph	ja
Beleuchtung/ Schattierung	Punktlichtquellen, gerichtete Lichtquellen, Phong-Schattierung
Typische Anwendungen	Interaktive Visualisierungen in den Bereichen Medizin, Architektur und Maschinenbau
Besonderheiten	ein einziger optimierter Rendering-Durchlauf, Json-Schnittstelle

Stand Juni 2012

Tabelle 4.1: *Ein Überblick über die wichtigsten Daten und Fakten von SceneJS [Kay, 2009].*

Die einzelnen Knotentypen besitzen in SceneJS klar verteilte Aufgaben und lassen sich häufig direkt auf einen bestimmten Schritt in einer Rendering-Pipeline übertragen. So fügen Rotationsknoten beispielsweise eine neue Rotationsmatrix zur Model-View-Transformation hinzu. Dies führt zu einem klaren Verständnis darüber, was im SceneJS-Framework passiert, wenn eine bestimmte Funktion genutzt wird. Da SceneJS komplett auf einer Json-Schnittstelle basiert, fügt es sich sehr gut in den bestehenden Arbeitsfluss der Client-Anwendung ein.

Was die Arbeit mit SceneJS deutlich erschwert, ist die unfertige und stark lückenhafte Dokumentation. So sind viele wichtige Knotentypen gar nicht oder nur unzureichend beschrieben. Nur wenige, aber ausreichende Beispiele wurden für den schnellen Einstieg ins Framework bereitgestellt. Weiterhin lassen sich grafische Effekte wie Screen Space Ambient Occlusion nicht in SceneJS umsetzen, da das Framework darauf ausgelegt ist, pro Szenengraphobjekt einen einzigen, optimierten Rendering-Durchlauf zu durchlaufen.

Die einfache Implementierung weist Performanzprobleme bei größeren Szenenstrukturen auf. Aber auch die optimierte Implementierungsvariante weist einige Nachteile auf. Der wohl größte Nachteil der optimierten Variante ist das nicht unterstützte Picking. Laut den Entwicklern soll dieses Problem jedoch in den nächsten Versionen des Frameworks behoben werden. Aber auch viele andere Funktionalitäten konnten aufgrund der speziellen Anforderungen nicht verwendet werden. So würde die eingebaute Rendering-Schleife von SceneJS bereits ein passendes Rahmenwerk für die Interaktion mit der Szene bereitstellen. Aufgrund der gewünschten Entlastung des Clients sowie der Unabhängigkeit der Szenenmanipulation vom konkreten Eingabegerät, wie in Abschnitt 4.1.2 beschrieben, konnte diese Funktion jedoch nicht genutzt werden.

Darüber hinaus sollte SceneJS als Framework gerade WebGL-Implementierungsdetails, wie beispielsweise den maximalen Wertebereich der Elemente des Index-Buffers vor dem Nutzer verstecken. Daher wäre es zu erwarten, dass SceneJS entsprechend große Eingabedaten automatisch in mehrere Szenenobjekte aufteilt, somit in verschiedene Buffer aufteilt, um diese in mehreren Rendering-Durchläufen zu zeichnen.

Kapitel 5

Three.js

Three.js stellt das momentan am häufigsten genutzte WebGL-Framework zur Erstellung von 3D-Anwendungen im Browser dar [Cabello, 2011]. Ursprünglich als Hobby-Projekt des Webentwicklers Ricardo Cabello erstellt, wird Three.js mittlerweile von einer spezialisierten Entwicklergemeinschaft gepflegt und weiterentwickelt. Von dem etwa 30 Entwickler umfassenden Team wird jeden Monat eine aktualisierte Version des Frameworks veröffentlicht. Ziel von Three.js ist es, eine leichtgewichtige Bibliothek zum Entwickeln von 3D-Anwendungen im Browser bereitzustellen. Dabei soll das Komplexitätslevel für den Entwickler möglichst gering gehalten werden.

Die folgenden Betrachtungen beziehen sich auf die Version 49 des Frameworks, die seit April 2012 verfügbar ist. Im folgenden Abschnitt soll die Arbeitsweise mit dem Framework veranschaulicht werden. Darüber hinaus soll eine mögliche Optimierung der vorgestellten Implementierung aufgezeigt werden. Zum Abschluss des Kapitels soll Three.js in Hinblick auf die in Kapitel 2 beschriebenen Anforderungen evaluiert werden.

5.1 Szenengraphbasierte Implementierung

Im Gegensatz zu SceneJS, bei dem lediglich eine Funktion zur Erstellung einer renderbaren Szene notwendig ist, erfordert Three.js das Zusammenspiel einer Vielzahl verschiedener Objekte zum Rendern einer Szene. Alle Klassen des Frameworks befinden sich im gemeinsamen Namesraum *THREE*. Die einzelnen, im Framework enthaltenen Komponenten folgen dem *Convention over Configuration*-Paradigma. Somit sind für viele Komponenten bereits Voreinstellungen getroffen worden, was den grundsätzlichen Aufbau einer Szene stark vereinfacht, sowie eine schnelle Einarbeitung in das Framework ermöglicht.

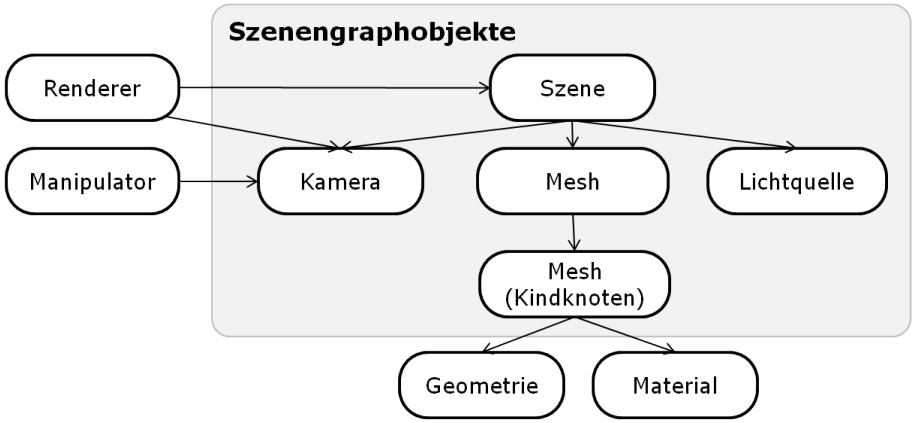

Abbildung 5.1: *Zur Erstellung einer Three.js-Anwendung ist das Zusammenspiel verschiedener Objekte notwendig.*

Zu den Grundbausteinen einer Three.js-Anwendung gehört die Szene, die gerendert werden soll, zusammen mit entsprechenden Szenenobjekten wie Meshes und Lichtquellen. Weiterhin wird ein Renderer sowie eine Kamera benötigt. Das Zusammenwirken dieser Komponenten ist in Abbildung 5.1 illustriert und soll im Folgenden näher erläutert werden.

5.1.1 Aufbau eines Szenengraphen in Three.js

Zum Anzeigen einer Szene wird in Three.js ein Renderer-Objekt genutzt. Three.js stellt hierfür mehrere Renderer zur Verfügung. So ist neben dem Rendering mit Hilfe von WebGL auch das Rendern durch den 2D-Kontext des HTML5 Canvas-Elements oder über SVG möglich. Somit können auch auf nicht WebGL-fähigen Geräten Szenen gerendert werden, indem einer der alternativen Renderer eingebunden wird. Der Renderer kann über verschiedene Parameter konfiguriert werden, die in einem Parameterobjekt, wie in JavaScript üblich, im Konstruktor des Renderers übergeben werden können. So kann zum Beispiel die Canvas, in die die Szene gerendert wird, angegeben oder bei Bedarf automatisch vom Framework ein neues Canvas-Element erstellt werden.

Three.js implementiert wie in SceneJS ein Szenengraphen, dieser ist allerdings weniger strikt ausgelegt als in SceneJS. Grundbaustein eines Szenengraphen bildet in Three.js die Klasse *Object3D*, von der sowohl die Kamera, alle Szenenobjekte als auch die Szeneklasse selbst erben. Neben Funktionen zur Modelltransformation bietet *Object3D* Methoden zum Hinzufügen sowie zum Entfernen von Kindelementen und erlaubt somit den Aufbau eines Szenengraphen. Die Szene ist in Three.js ein einfacher Datenhalter, der intern aber zwischen Licht-, Kamera- und anderen Szenenobjekten unterscheidet.

Three.js stellt dem Nutzer mehrere Kameratypen zur Verfügung. Hierbei werden sowohl orthografische als auch perspektivische Projektion unterstützt. Abhängig vom konkreten Kameratyp sind bei der Initialisierung weitere Parameter nötig. Im Falle der perspektivischen Kamera muss beispielsweise das Kamerafrustum angegeben werden. Zusätzlich muss die Kamera der Szene als Kindknoten hinzugefügt werden, die sie betrachten soll. Die Kamera stellt die Matrix für die Transformation der Szene vom Welt- in das Kamerakoordinatensystem bereit. Diese kann vom Nutzer über die Position der Kamera sowie des betrachteten Punktes konfiguriert werden.

Für die Beleuchtung der Szene bietet Three.js eine große Auswahl verschiedener Lichtquellen an. Unter anderem werden Punkt-, Umgebungs- und gerichtete Lichtquellen vom Framework bereitgestellt. Neben der normalen Lichtberechnung stellt Three.js ebenfalls eingebaute Funktionen für *Shadow Mapping*, eine Methode Schattenwürfe zu simulieren, bereit [Engel, 2009]. Da diese Funktionalität jedoch wenig performant ist, wird sie in dieser Arbeit nicht weiter beschrieben. Wurden keine Lichtquellen vom Nutzer angegeben, so nutzt Three.js automatisch ein Umgebungslicht zur Beleuchtung.

5.1.2 Integration der Geometrien in den Szenengraphen

Im Folgenden soll das Integrieren von Szenenobjekten in eine bestehende Szene gezeigt werden. Dabei wird für eine Beispielimplementierung von der in Quelltext 4.1.2 beschriebenen Szenendatenstruktur ausgegangen. Durch Iteration über diese Struktur kann auf Geometrieinformationen sowie Farben der einzelnen Graphknoten zugegriffen werden.

Szenenobjekte, welche in Three.js *Mesh* genannt werden, bestehen aus Geometrien und Materialen. Materialien legen die Eigenschaften des Meshes in Hinblick auf die Schattierung fest und interagieren eng mit den bereits beschriebenen Lichtquellen. In Three.js stehen eine Reihe von Materialien zur Auswahl. Hierzu zählen unter anderem ein Basismaterial, durch das das entsprechende Mesh unbeleuchtet dargestellt wird sowie das *MeshPhongMa-*

terial, welches die Phong-Schattierung nutzt [Phong, 1975]. Neben der Schattierung bieten Materialien in Three.js eine Reihe weiterer Funktionalitäten, die das Erscheinungsbild des Meshes beeinflussen. So ist das Initialisieren von Materialien mit Farben oder verschiedenen Texturen möglich. Somit kann das Material eines Knotens, wie in Quelltext 5.1 gezeigt, festgelegt werden.

```
1  var geometry = new THREE.Geometry();
2
3  for(var i = 0; i < node.vertices.length; i += 3) {
4      geometry.vertices.push(new THREE.Vector3(
5          node.vertices[i  ], // x-Koordinate
6          node.vertices[i+1], // y-Koordinate
7          node.vertices[i+2]  // z-Koordinate
8      ));
9      geometry.faces.push(new THREE.Face3( i, i+1, i+2));
10 }
11 geometry.computeFaceNormals();
12
13 var material = new THREE.MeshBasicMaterial({
14     color: new THREE.Color().setRGB(
15         node.colors[0], // rot
16         node.colors[1], // grün
17         node.colors[2]  // blau
18     )
19 });
20
21 var mesh = new THREE.Mesh(geometry, material);
22 scene.add(mesh);
```

Quelltext 5.1: *Beispielhafte Erstellung eines Meshes aus einem übertragenen Knoten sowie das Hinzufügen zu einer bestehenden Szene.*

Die Geometrieinformationen müssen ebenfalls in ein Three.js-eigenes Format übertragen werden. Das Framework bietet eine große Auswahl vordefinierter Geometrien wie Quader oder Kugeln. Da vom Server jedoch beliebige Meshes generiert werden können, muss ein allgemeingültiger Ansatz gewählt werden. Mit Hilfe der Klasse *Geometry* lassen sich beliebige Szenengeometrien erstellen, wie in Quelltext 5.1 illustriert. Three.js bietet verschiedene Möglichkeiten, die Normalen einer Geometrie automatisch zu errechnen. Neben diesem Ansatz können die Normalen auch manuell für jeden Vertex beziehungsweise jeder Oberfläche angegeben werden.

Sind Geometrie und Material eines Meshes angegeben, kann dieses erstellt und einer Szene hinzugefügt werden. Sobald auf diese Weise alle Szenenobjekte der Szene hinzugefügt wurden, kann die Szene mittels der *render*-Methode des Renderers angezeigt werden. Als Parameter dieser Methode muss die Szene, die gerendert werden soll, sowie die zu verwendende Kamera angegeben werden.

5.1.3 Interaktion mit der Szene

Die Interaktion mit der Szene ist in Three.js mit Hilfe einer Reihe von Szenenmanipulatoren möglich. Im Gegensatz zu SceneJS, bei dem die Szenenobjekte verschoben wurden, um eine Interaktion durchzuführen, passen die unterschiedlichen Manipulatoren in Three.js jeweils die Kamera an. Dies geschieht über die Position der Kamera sowie über den betrachteten Punkt.

Viele der verfügbaren Manipulatoren wurden durch Entwickler, die Three.js nutzen, erstellt und offiziell ins Framework integriert. So beispielsweise auch die *TrackballControls* von Eberhard Gräther [Gräther, 2011]. Diese bieten eine große Auswahl an verschiedenen

Konfigurationsmöglichkeiten, um die durchgeführte Interaktion zu beeinflussen. Neben der Interaktion mit der Szene muss ein Manipulator weitere Anforderungen erfüllen, welche in Kapitel 2 beschriebenen sind. Dazu gehört das Unterstützen verschiedener Rückruffunktionen, um beispielsweise ein Neuzeichnen der Szene nach einer getätigten Interaktion zu ermöglichen. Diese Funktionalität wird allerdings nicht von den verwendeten *Trackball-Controls* bereitgestellt, wodurch diese leicht angepasst werden müssen. So kann von der entsprechenden Klasse geerbt und eine Möglichkeit zur Registrierung von Rückruffunktionen sowie deren Aufruf integriert werden. Dadurch kann nach einer entsprechenden Interaktion die Szene neu gerendert werden.

5.1.4 Picking zum Auslesen von Metadaten

Three.js stellt keine Komponente für Picking bereit, bietet jedoch alle Funktionalität, um ohne viel Aufwand Picking mit Hilfe von Raycasting [Glassner, 1989] zu realisieren. Hierzu zählt die Klasse *Projector*, die es erlaubt, perspektivische Verzerrungen, die durch Verwendung einer entsprechenden Kamera entstehen, zu errechnen. Mit dessen Hilfe lässt sich unter Angabe der Kamera ein Vektor in ein Objekt der Klasse *Ray* umwandeln. Dieses Objekt ermöglicht es, eine sortierte Liste aller vom Strahl getroffenen Szenenobjekte zu ermitteln. Die in Quelltext 5.2 gezeigte Implementierung liefert beispielsweise das vorderste getroffene Objekt, falls vorhanden, zurück. Mit Hilfe der ID dieses Objekts kann durch die Komponente *Sidebar* das Anzeigen von Metainformationen realisiert werden.

```
1 var projector = new THREE.Projector();
2 var vector = new THREE.Vector3(mouse.getX(), mouse.getY(), 0.5);
3 var pickingRay = projector.pickingRay(vector, camera);
4 var intersections = pickingRay.intersectObjects(scene);
5 if (intersections.length > 0 && intersections[0].object)
6   return intersections[0].object;
```

Quelltext 5.2: *Eine Beispielimplementierung von Raycasting in Three.js.*

5.2 Optimierung durch Verwendung eines einzelnen Meshes

Ähnlich wie bei SceneJS zeigten frühe Performanztests, dass die im vorherigen Kapitel beschriebene Implementierung auf größeren Datenmengen schlecht skaliert. Daher soll im Folgenden eine Optimierung vorgestellt werden, die den Zusatzaufwand durch das Framework nach Möglichkeit minimiert.

Three.js nutzt zum Rendern jedes Szenenobjekts einen separaten Renderingschritt [Heikkinen, 2012]. Dies führt jedoch bei der Visualisierung eines Graphen dazu, dass sehr viele, aber kleine Geometrien einzeln gerendert und somit mehr Rendering-Durchläufe als nötig durchgeführt werden. Durch den Aufbau einer komponierten Geometrie lässt sich dieser Aufwand auf einen einzigen Renderingdurchlauf reduzieren.

Diese Geometrie steht bereits in Form der vom Server generierten Graphdaten zur Verfügung, welche in ein einziges Mesh-Objekt von Three.js konvertiert werden müssen. Die Vertices und Normalen der Geometrie können dabei wie im vorherigen Abschnitt beschrieben, in die entsprechende Form überführt werden. Jedoch besitzen Materialien normalerweise nur eine Farbe, die für das gesamte Mesh genutzt wird. Mit Hilfe des Attributes *vertexColors* ist es aber möglich, mehrere Farben für ein Mesh zu definieren. Dazu muss dieses Attribut auf eine von drei Konstanten gesetzt werden, um zu signalisieren, dass beispielsweise für jeden Vertex beziehungsweise für jede Oberfläche eine Farbe gesetzt wurde. Die Farben selbst werden als Attribute der Oberflächen definiert, wie in Quelltext 5.3

am Beispiel von *THREE.FaceColors* gezeigt. Somit lassen sich auch größere Datenmengen in Three.js in ein Mesh konvertieren und effizient rendern.

```
1  for (var i = 0; i < numberOfTriangles; i++) {
2    geometry.faces.push(new THREE.Face3(3*i, 3*i+1, 3*i+2));
3      geometry.faces[i].color = new THREE.Color().setRGB(
4          triangles.colors[12*i   ], // rot
5          triangles.colors[12*i+1], // grün
6          triangles.colors[12*i+2]  // blau
7      );
8  }
9  geometry.computeFaceNormals();
10
11 var material = new THREE.MeshBasicMaterial({ vertexColors: THREE.FaceColors });
12 var mesh = new THREE.Mesh(geometry, material);
```

Quelltext 5.3: *Konvertierung des vom Client empfangenen Meshes in ein Three.js-Szenenobjekt.*

5.3 Evaluierung von Three.js

Dieses Kapitel reflektiert im Folgenden den Eindruck, den das Three.js-Framework vermittelt. Der Vergleich zu anderen Implementierungen ist in Kapitel 6 zu finden.

	Three.js
aktuelle Version	Version 49 (22. April 2012)
erste Version	Version 1 (24. April 2010)
weitere wichtige Versionen	-
Anzahl aktiver Entwickler	30
Entwicklung seit	2009 (ActionScript-Version: 2002)
Größe der Datei (minimiert)	357 KB
Dokumentation	lückenhafte, teilweise automatisch generierte Dokumentation, umfangreiche Beispiele
Unterstützte grafische Primitive	Punkte, Dreiecke, Linien
Picking-Verfahren	einfaches Raycasting
Szenengraph	ja
Beleuchtung/ Schattierung	diverse Lichtquellen und Beleuchtungsmodelle, Shadow Mapping, Screen Space Ambient Occlusion
Typische Anwendungen	Interaktive Modellbetrachtungen, HTML5-Spiele, momentan experimentelles Stadium
Besonderheiten	Konzipiert auf Austauschbarkeit des Renderers

Stand Juni 2012

Tabelle 5.1: *Ein Überblick über die wichtigsten Daten und Fakten von Three.js [Cabello, 2011].*

Die vorgestellte Implementierung zur Visualisierung massiver Graphdaten konnte mit Three.js ohne größeren Aufwand erstellt werden. Das Framework ist speziell darauf ausgelegt, dem Entwickler eine Schnittstelle mit geringer Komplexität zu liefern. So nehmen die

vorgestellten Materialien dem Entwickler viel Arbeit ab und bieten eine komfortable Kapselung benötigter Schattierungsalgorithmen. Dank des *Convention over Configuration*-Prinzips wird der Aufbau einer Szene trotz umfangreicher Konfigurationsmöglichkeiten durch Standardwerte stark vereinfacht. Somit ist ein schnelles Einarbeiten in das Framework möglich.

Die zum Rendering einer Szene notwendigen Kernkomponenten einer Three.js-Anwendung lassen sich leicht verwenden und besitzen klar getrennte Aufgabenbereiche. Sie kapseln die darunterliegende WebGL-Schnittstelle stark, wodurch ein klares Übertragen der Three.js-Komponenten auf WebGL-Funktionalität oft nicht möglich ist.

Three.js arbeitet nahezu komplett mit eigenen Datentypen. So existiert beispielsweise eine eigene Matrixbibliothek für das Framework. Dies erschwert jedoch die Integration in das bestehende System, da die gesamten Szenendaten in Three.js-eigene Strukturen überführt werden müssen.

Eine Besonderheit des Three.js-Frameworks ist, dass es nicht nur das Rendern mit Hilfe von WebGL sondern auch über die Canvas und über SVG unterstützt. In vielen Fällen ist dies zwar deutlich langsamer als das entsprechende WebGL-Verfahren, ermöglicht aber auch in nicht WebGL-fähigen Browsern das Anzeigen der Graphdaten.

Three.js stellt ein Framework dar, welches sich noch in der Entwicklung befindet. Eine aktive Entwicklergemeinschaft von rund 30 Personen arbeitet an dem Framework. Dies bringt einerseits den Vorteil mit sich, dass auftretende Fehler im Framework schnell durch die jeweiligen Entwickler behoben werden können. Gleichzeitig ändert sich mit jeder neu herausgebrachten Version jedoch auch die bereitgestellte Schnittstelle. Neue Funktionen werden hinzugefügt, alte verschwinden. Die Schnittstellen vieler Funktionalitäten werden mit jeder neuen Version geändert, was es für den Nutzer des Frameworks schwierig macht, seine Implementierung aktuell zu halten.

Dieses Problem resultierte darüber hinaus in einer fast vollständig fehlenden Dokumentation des Frameworks [Cabello, 2012]. Viele Webseiten verweisen auf automatisch generierte Beschreibungen des Frameworks. Diese sind jedoch meist veraltet oder sehr lückenhaft.

Zusammenfassend ist Three.js ein Framework, das für Webentwickler sowie kleinere Anwendungen ausgelegt ist. Es besitzt eine sehr einfache Schnittstelle und ermöglicht dabei viele grafische Spielereien. Jedoch werden viele Funktionalitäten sehr stark gekapselt, wodurch es für Anwendungen mit speziellen Anforderungen ungeeignet sein könnte.

Kapitel 6

Vergleich

Honauer stellt die webbasierte Implementierung einer Graphvisualisierung ohne Zuhilfenahme eines WebGL-Frameworks vor, die im Folgenden als native Variante bezeichnet wird [Honauer, 2012]. Dieses Kapitel soll die vorgestellten Frameworks untereinander sowie mit der nativen Implementierung vergleichen. Zuvor soll die native Variante kurz in Hinblick auf die in Kapitel 2 gestellten Anforderungen evaluiert werden. Danach folgen Vergleiche in den Kategorien Performanz, Funktionalität und Erweiterbarkeit. Zum Abschluss des Kapitels soll die verwendete Lösung zur Visualisierung massiver Graphdatenstrukturen mit Hilfe von WebGL vorgestellt werden. Darüber hinaus sollen die eklatantesten Schwächen der jeweiligen Frameworks aufgezeigt werden.

Als Testsystem für die folgenden Performanzbetrachtungen stand ein 64-bit Windows 7 Betriebssystem zur Verfügung. Der Testrechner verfügte über 6 GB Arbeitsspeicher, einem 2.6 GHz Intel i5 QuadCore-Prozessor und einer NVIDIA GeForce 310 Grafikkarte mit Treiberversion 280.26. Alle Messungen wurden in Mozilla Firefox mit Hilfe von Firebug 1.9.2 durchgeführt.

6.1 Evaluierung natives WebGL

Dieser Abschnitt soll diskutieren, inwieweit eine Implementierung ohne Framework rein auf Basis von WebGL für den Client sinnvoll ist. Die native Implementierung ist stark auf Performanz ausgelegt und soll den entstehenden Zusatzaufwand möglichst gering halten. Die vom Client entgegengenommenen Graphdaten werden schnellstmöglich in getypte Arrays [Vukicevic & Russell, 2011] konvertiert und in die Buffer der Grafikkarte kopiert. Um den Client nicht unnötig mit größeren Datenmengen zu belasten, werden die Datenstrukturen danach dereferenziert.

Die Implementierung basiert nicht auf einem Szenengraphen, sondern nutzt lediglich eine flache Struktur, die auf die vom Server generierten Szenendaten zugeschnitten ist. Somit werden unnötige Konvertierungen in einen Szenengraphen und zurück in eine flache Array-Struktur vermieden. Darüber hinaus wurde auf die Verwendung eines Index-Arrays verzichtet, da WebGl maximal 16-Bit-Indizes unterstützt und somit mehrere Rendering-Durchläufe pro Bild nötig wären. Picking wurde mit Hilfe eines Farbkodierungs-Verfahrens [Hanrahan & Haeberli, 1990] umgesetzt. Die zum Anzeigen der Metainformationen notwendigen IDs werden dabei auf die Farben der Geometrien abgebildet. Vertex- und Normalen-Buffer des normalen Rendering-Schrittes können wiederverwendet werden, lediglich das Farb-Array wird ausgetauscht. Die Textur wird direkt nach dem Rendern der Szene erzeugt, nicht erst beim Abfragen der Information, da nach einer Veränderung der Szene häufig sofort weitere Informationen angefragt werden. Grafische Effekte wie Screen Space Ambient Occlusion lassen sich bei Bedarf deaktivieren, um größere Performanz zu ermöglichen.

Somit wurde eine schnelle Implementierung erreicht, welche direkt auf der WebGL-

Schnittstelle aufsetzt. Nachteil daran ist das geringe Abstraktionslevel bedingt durch die Schnittstelle zu WebGL.

6.2 Performanz

Im folgenden Abschnitt soll ein Vergleich der Implementierungen in Bezug auf ihre Performanz stattfinden. Untersucht wurden hierbei die initiale Zeit bis zum ersten vollständigen Rendering-Durchlauf, die Bildwiederholrate sowie die Dauer des Picking-Vorganges.

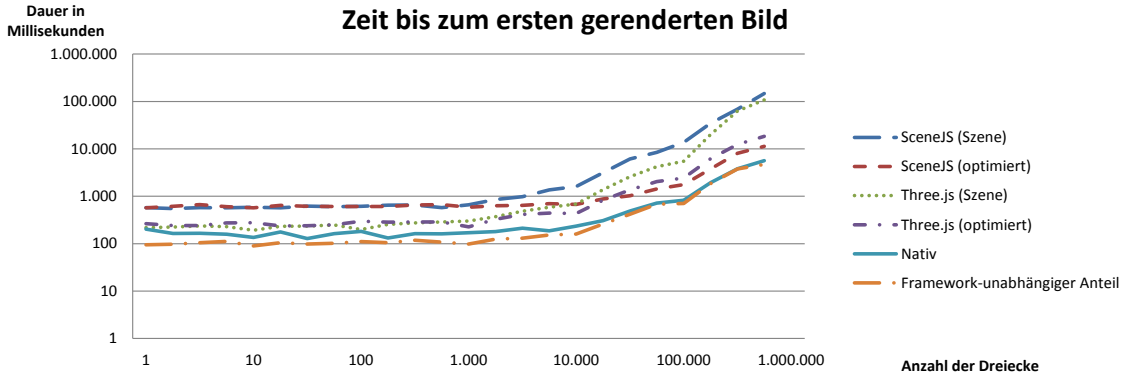

Abbildung 6.1: *Zeit bis zum ersten angezeigten Bild der betrachteten Frameworks im Vergleich.*

Initiale Wartezeit Abbildung 6.1 zeigt die Zeit bis zum ersten Rendering der verschiedenen Ansätze bei größer werdenden Szenengeometrien. Zu beachten ist, dass bei allen Untersuchungen, die in Abhängigkeit von der Anzahl der zu rendernden Dreiecke getätigt wurden, beide Achsen logarithmisch skaliert sind. Untersucht wurde lediglich die Zeit, die die Komponente *GraphVis* zum Rendering benötigt. So wurde beispielsweise das Erstellen der Menü-Elemente der *Sidebar*-Komponente ausgeblendet. Dabei wurde bei SceneJS die in Abschnitt 4.1 beschriebene Implementierung als szenenbasierte Variante bezeichnet, während die in Abschnitt 4.2 beschriebene Implementierung als optimiert gekennzeichnet ist. Analoge Bezeichnungen wurden für die Three.js-Implementierungen verwendet.

Die Dauer bis zum ersten Rendering setzt sich aus einem Framework-abhängigen Teil und einem Framework-unabhängigen Teil zusammen. Der Framework-unabhängige Teil der Gesamtzeit setzt sich hauptsächlich aus dem Einlesen und Parsen der Szenendaten zusammen, wodurch die Dauer abhängig von den entgegengenommenen Graphdaten ist. Abbildung 6.1 vergleicht den Framework-unabhängigen Teil mit der Gesamtzeit der beschriebenen Implementierungen. Im Folgenden soll die Framework-abhängige Dauer genauer aufgeschlüsselt werden. Hierzu wurde der Ablauf in die Abschnitte Initialisierung, Konvertierung und erstes Rendering unterteilt.

Initialisierung Die Initialisierung umfasst den Aufbau aller Komponenten, die für das Erstellen einer grundsätzlichen 3D-Anwendung in der jeweiligen Implementierung notwendig sind. Abstrahiert wurde dabei von der konkreten Geometrie, die dargestellt werden soll. Somit ist diese Betrachtung unabhängig von der Anzahl der zu rendernden Dreiecken. Während die native Implementierung nur den gl-Kontext laden muss und entsprechende Shader-Programme kompiliert, fällt bei beiden Frameworks ein Zusatzaufwand aufgrund des Aufbaus des Szenengraphen beziehungsweise der umfangreichen Objektstruktur an. Die jeweiligen Framework-Implementierungen basieren auf einem nahezu gleichen Aufbau.

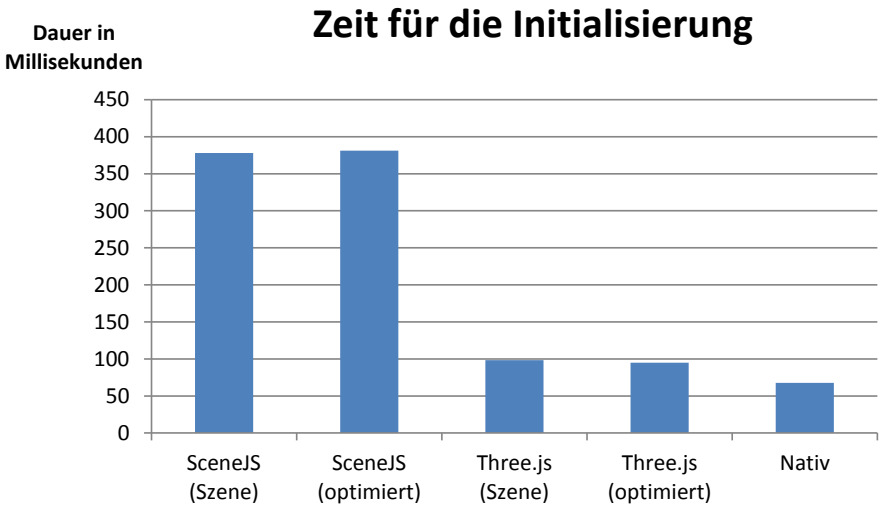

Abbildung 6.2: *Die Erstellung eines Szenengraphen in SceneJS ist relativ umfangreich, wodurch im Vergleich viel Zeit für die Initialisierung benötigt wird.*

Nur minimale Änderungen sind aufgrund der Optimierungen notwendig, daher variiert die Dauer dort nur leicht, wie in Abbildung 6.2 zu sehen.

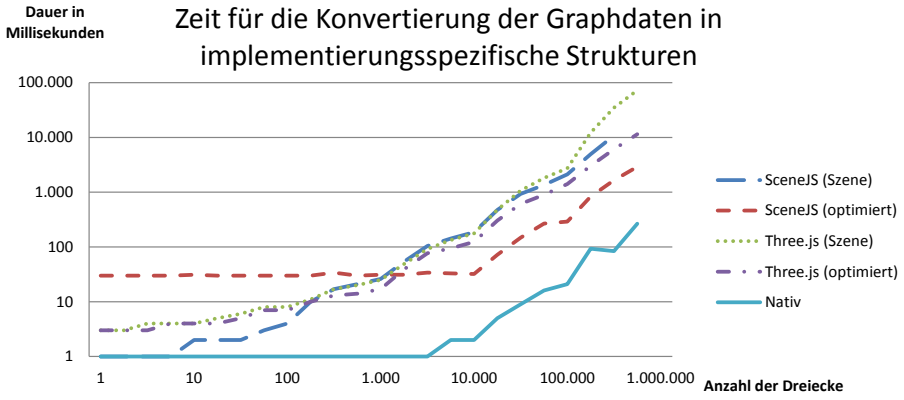

Abbildung 6.3: *Performanzanalyse der Konvertierung in Framework-spezifische Strukturen.*

Konvertierung der Szenendaten Wurden die Szenendaten eingelesen, so müssen sie in ein WebGL-kompatibles Format überführt werden. Abbildung 6.3 untersucht die Dauer der Konvertierung in Bezug auf große Datenmengen. Three.js basiert komplett auf eigenen Datentypen. Für beide Three.js-Varianten ist daher eine umfangreiche Konvertierung in diese Datenstrukturen notwendig. Daher wächst die Dauer des Vorganges mit größer werdenden Datenmengen linear an. Ebenso müssen bei der szenenorientierten SceneJS-Implementierung abhängig von der Datenmenge neue Knoten erstellt und zum Szenengraphen hinzugefügt werden. So erklärt sich das lineare Wachstum dieser drei Implementierungen. Die optimierte SceneJS-Implementierung ermöglicht die Konvertierung der Graphdaten in ein Szenenobjekt in konstanter Zeit. Dies ist bis zu einer Datenmenge von etwa 20.000 Dreiecken möglich. Danach müssen aufgrund der Beschränkung der Indizes auf 16 Bit die Daten in mehrere Szenenobjekte aufgespalten werden. Nähere Informationen dazu finden sich in Abschnitt 4.2. Die native Implementierung sieht vom Aufbau umfangreicher Objekte ab.

Lediglich die Überführung der übermittelten Szenendaten in getypte Arrays fällt hier ins Gewicht, wodurch eine sehr schnelle Konvertierung erreicht wird.

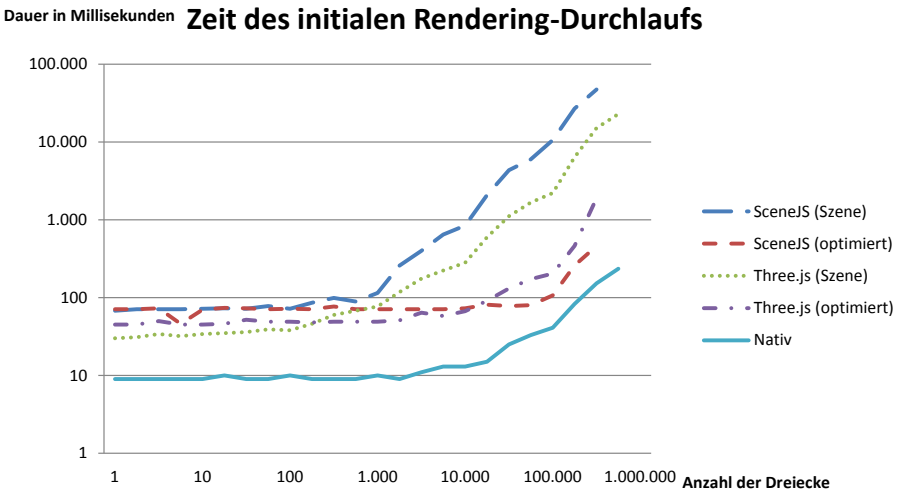

Abbildung 6.4: *Performanzvergleich des initialen Rendering-Durchlaufes.*

Initiales Rendering Abbildung 6.4 zeigt die Dauer des ersten Rendering-Durchlaufs in Abhängigkeit der Größe der darzustellenden Datenmengen. Zu beachten ist, dass sämtliche Framework-Implementierungen auch bei sehr kleinen Datenmengen langsamer als die native Variante sind. Dies liegt daran, dass die Frameworks erst kurz vor dem ersten Rendering-Durchlauf ihren Szenengraphen kompilieren, entsprechende Shader anlegen und Buffer befüllen. Die native Variante erzeugt beispielsweise die Shader-Programme bereits bei der Initialisierung. Beide szenenorientierten Varianten basieren auf einem Rendering-Durchlauf pro Szenenobjekt, wodurch die Rendering-Dauer stark mit der Anzahl der zu rendernden Objekte zunimmt.

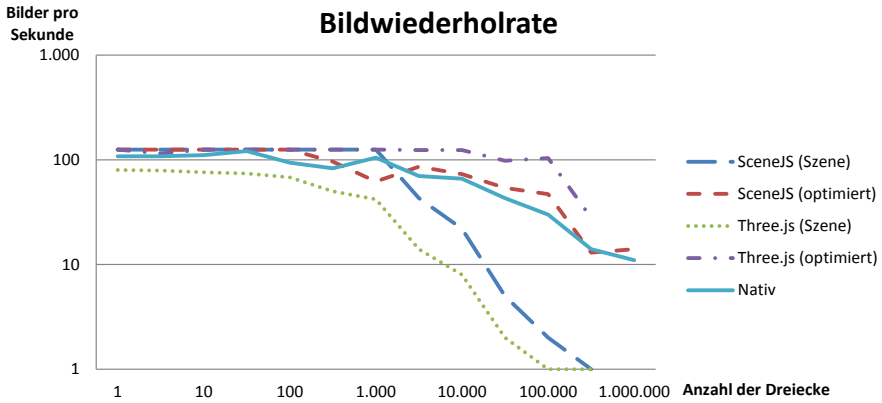

Abbildung 6.5: *Bildwiederholrate der vorgestellten Ansätze.*

Bildwiederholrate Abbildung 6.5 betrachtet die untersuchte Bildwiederholrate der verschiedenen Implementierungen. Hierbei sei zu beachten, dass der Client dazu konzipiert ist, die Szene nur nach einer vom Nutzer getätigten Interaktion neu zu rendern. Dies erklärt auch die anfangs konstante Bildwiederholrate von 125 Bilder pro Sekunde (fps), da Firefox alle acht Millisekunden neu Ereignisse abfängt. Im Chrome-Browser beträgt die maximal

erreichte Bildwiederholrate aus analogen Gründen nur rund 62 [Honauer, 2012]. Generell weisen alle Implementierungen bei kleineren Datenmengen bis zirka 1.000 Dreiecken eine konstante Bildwiederholrate auf. Die meisten Implementierungen erreichen hier sogar die maximale Anzahl von 125 Rendering-Durchläufen pro Sekunde. Lediglich die einfache Three.js-Variante erreicht diese Anzahl nicht. Die Wiederholrate der szenenbasierten Varianten bricht bei rund 3.000 Dreiecken rasant ein, bis schließlich bei 100.000 Dreiecken keine Interaktion mehr möglich ist. Bei allen anderen Varianten ist jedoch selbst bei einer 500.000 Dreiecke umfassenden Szene noch Echtzeitinteraktion [Rusdorf, 2008] möglich. Die optimierte Three.js-Variante schneidet hierbei am besten ab. So bleibt die Bildwiederholrate bei diesem Ansatz bis zu 100.000 Dreiecken nahezu konstant, sinkt danach ebenfalls stark ab.

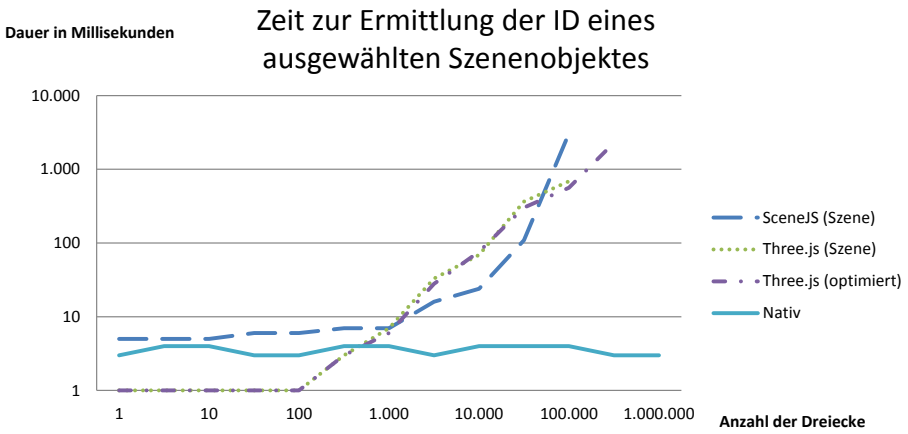

Abbildung 6.6: *Dank des Farbkodierungs-Verfahrens ist die Dauer des Pickings der nativen Implementierung unabhängig von der Anzahl der Szenenobjekte.*

Picking Abbildung 6.6 zeigt einen Vergleich der Picking-Implementierungen der verschiedenen Ansätze. Untersucht wurde die Dauer zur Ermittlung der ID. Das darauf folgende Anzeigen der Metainformationen verläuft in allen Fällen gleich und wurde außen vor gelassen. Beide Implementierungsvarianten des Three.js-Frameworks basieren auf einem ähnlichen, sehr einfachen Raycasting-Verfahren, somit steigt die Dauer des Picking-Vorgangs mit der Größe der dargestellten Geometrie an. Ab einer Szene mit etwa 10.000 Dreiecken ist die Interaktion nicht mehr echtzeitfähig. Die Dauer eines Picking-Vorgangs bleibt bei der nativen Variante dank des beschriebenen Farbkodierungs-Verfahrens konstant, dafür wird aber bei jedem Zeichnen der Szene zusätzlich ein zusätzlicher Rendering-Durchlauf benötigt. Somit wurde ein minimal langsameres Rendering der Szene in Kauf genommen [Honauer, 2012]. Die Szenenvariante von SceneJS basiert ebenfalls auf einem Farbkodierungs-Verfahren, wodurch in konstanter Zeit das Heraussuchen des Namensknotens möglich ist. Um jedoch die entsprechende ID zu ermitteln, ist ein zusätzliches Durchsuchen des Szenengraphen nach der darunterliegenden Geometrie notwendig. Dadurch steigt die Dauer des Picking-Vorganges mit komplexer werdender Szenengeometrie an. In der auf Performanz optimierten Variante von SceneJS wird kein Picking unterstützt, da mehrere IDs innerhalb eines Geometrieobjektes vom Framework nicht erkannt werden können.

6.3 Funktionalität und Erweiterbarkeit

Dieser Abschnitt soll diskutieren, inwieweit die vorgestellten Implementierungen die Anforderungen aus Kapitel 2 erfüllen. Hierzu bietet die Tabelle 6.1 einen Überblick über wichtige Daten zu den untersuchten Alternativen im Vergleich.

In allen vorgestellten Implementierungen ist eine Interaktion mit der Szene in Echtzeit möglich. Jedoch unterstützt kaum eine der gezeigten Variante Picking in gewünschter Form. Die optimierte SceneJS-Variante ermöglicht überhaupt kein Picking. In allen Varianten außer der nativen Implementierung wird Picking in größeren Szenen schnell inperformant.

Im Gegensatz zu den vorgestellten Frameworks arbeitet die native Variante ohne ein Szenengraphsystem. Jedoch haben die Untersuchungen im vorherigen Abschnitt gezeigt, dass die Nutzung eines Szenengraphen durch aktuelle Frameworks nur zu Lasten der Performanz möglich ist. Die optimierten Varianten versuchen den Zusatzaufwand durch einen Szenengraphen zu minimieren, indem möglichst wenig mit dem Szenengraphen gearbeitet wird. Somit ist diese Funktionalität für die Darstellung großer Datenmengen momentan noch nicht ausgereift.

SceneJS integriert sich dank der Json-Schnittstelle gut in die vorhandenen Komponenten. Die eigenen Datenstrukturen von Three.js erschweren die Integration in den bestehenden Client, da häufig umfangreiche Konvertierungen notwendig sind. Aufwendige grafische Effekte, wie die Schattierung der Szene durch *Screen Space Ambient Occlusion*, ist nur mit Three.js und der nativen Implementierung möglich. SceneJS ist nicht für solche Effekte konzipiert, da ein einzelner optimierter Rendering-Durchlauf angestrebt wird. Zusätzlich zu den eingeschränkten Funktionalitäten bringen Frameworks oft die Gefahr mit sich, in eine bestimmte angestrebte Richtung hin nicht erweiterbar zu sein. Bestimmte Funktionalitäten können nicht oder nur mit sehr großem Aufwand implementiert werden. Dennoch erhöhen die vorgestellten Frameworks die Wartbarkeit der 3D-Anwendung durch das Kapseln der WebGL-Funktionalitäten und verringern somit die Codekomplexität.

6.4 Verwendete Lösung

Beide szenenbasierten Implementierungen weisen große Nachteile in Hinblick auf die Performanz der Anwendung auf. Hochwertige Abstraktionen von der WebGL-Schnittstelle wie ein Szenengraphsystem wurden bisher nicht performant genug umgesetzt, um auch auf größere Datenmengen zu arbeiten. Daher kommt nur eine der optimierten Framework-Varianten oder die native Implementierung zur Umsetzung der *GraphVis*-Komponente infrage. Die optimierte SceneJS-Implementierung unterstützt kein Picking. In der optimierten Three.js-Variante wird Picking bei größeren Szenendaten nicht mehr echtzeitfähig. Darüber hinaus stellt Three.js ein Framework dar, welches sich in permanenter Entwicklung befindet. Die Schnittstelle, die das Framework bietet, ist keinesfalls stabil und ändert sich häufig mit den regelmäßigen Aktualisierungen. Somit wäre eine häufige Anpassung der Implementierung nicht zu vermeiden. Da sämtliche untersuchte Frameworks die erwarteten Anforderungen nicht erfüllen konnten, wird im Zuge des Projektes auf eine Implementierung ohne Zuhilfenahme eines WebGL-Frameworks gesetzt. Diese Implementierung ist speziell auf die konkrete Problemstellung zugeschnitten und bietet daher vor allem in Hinblick auf Performanz Vorteile gegenüber den untersuchten Frameworks.

Jedoch wurde der Client so konzipiert, dass zukünftige Frameworks, die den gezeigten Anforderungen besser entsprechen, schnell in den Client integriert werden können [Honauer, 2012]. Somit bleibt der Client erweiterbar und kann in Zukunft durch eine verbesserte Implementierung ergänzt werden.

	SceneJS		Three.js		Nativ
	Szene	Optimiert	Szene	Optimiert	
aktuelle Version	Beta 2.0 (15. November 2011)		Version 49 (22. April 2012)		-
Anzahl aktiver Entwickler	5		30		7
Entwicklung seit	2009		2009 (ActionScript-Version: 2002)		Oktober 2011
Größe der Datei (minimiert)	337 KB		357 KB		21 KB
Dokumentation	Stark lückenhaft, lediglich Kern-komponenten, wenige Beispiele		Lückenhaft, teilweise automatisch generiert, umfangreiche Beispiele		Umfangreiche Methodendokumentierung
Unterstützte grafische Primitive	Punkte, Dreiecke, Dreieckslinien, Dreiecksfächer, Linien, Linienzüge		Punkte, Dreiecke, Linien		Punkte, Dreiecke, Dreieckslinien, Dreiecksfächer, Linien, Linienzüge
Picking-Verfahren	Farbkodierung	Nicht möglich	Einfaches Raycasting		Farbkodierung
Szenengraph	Ja		Ja		Nein
Beleuchtung/ Schattierung	Punktlichtquellen, gerichtete Lichtquellen, Phong-Beleuchtung		diverse Lichtquellen und Beleuchtungs-modelle, Shadow Mapping, Screen Space Ambient Occlusion		Umgebungslicht, gerichtete Lichtquellen, Phong-Beleuchtung, Screen Space Ambient Occlusion
Performanz bei 100.000 Dreiecken:					
• Initialisierung	13.766 ms	1.767 ms	5.505 ms	2.432 ms	820 ms
• Bildwiederholrate	2 fps	47 fps	1 fps	104 fps	30 fps
• Picking-Dauer	3.065 ms	-	688 ms	561 ms	4 ms
Typische Anwendungen	Interaktive Visualisierung in den Bereichen Medizin, Architektur und Maschinenbau		Interaktive Modellbetrachtungen, HTML5-Spiele, experimentelle Beispiele		Anwendungen zur Visualisierung massiver Graphdaten, wie Software-Analysedaten
Besonderheiten	Json-Schnittstelle, ein einziger optimierter Rendering-Durchlauf		Konzipiert auf Austauschbarkeit des Renderers		Optimiert auf schnelles Rendering und Darstellung großer Datenmengen

Stand Juni 2012

Tabelle 6.1: *Wichtige Daten zu den untersuchten Implementierungen im Vergleich.*

Kapitel 7

Ausblick

Seit der Veröffentlichung der WebGL-Spezifikation im Februar 2011 hat sich innerhalb kürzester Zeit eine Vielzahl von Frameworks um den neuen Standard entwickelt. Die häufig von Hobbyentwicklern initiierten Projekte erleichtern das Erstellen hardwarebeschleunigter 3D-Anwendungen, indem sie beispielsweise Szenengraphen oder Beleuchtung anbieten. Somit eignen sich diverse Frameworks zum Einstieg in die neue Technologie, da sie die Komplexität der WebGL-Schnittstelle kapseln und so eine einfachere Verwendung ermöglichen. Jedoch sind die meisten Frameworks noch nicht so stark ausgereift, um auch größere Projekte unterstützen zu können. Viele sind schlichtweg nicht auf massive Datenmengen ausgelegt.

Momentan zeichnet sich der Trend ab, dass Three.js in naher Zukunft den Status des Standard-Frameworks erlangen könnte. Das sich noch stark in der Entwicklung befindende Framework bietet eine einfach strukturierte Schnittstelle und erfreut sich daher gerade bei vielen Webentwicklern großer Beliebtheit.

Die weitere Entwicklung von WebGL hängt stark von den aktuellen beziehungsweise zukünftigen Frameworks ab. Für eine breit gefächerte Nutzung der WebGL-Technologie muss das Erstellen der 3D-Anwendung einfach sein, gleichzeitig muss die Anwendung aber auch eine gute Skalierbarkeit bieten. Somit müssen zur Festigung von WebGL als 3D-Browsertechnologie leistungsstarke sowie leicht verständliche Frameworks entwickelt werden. Gleichzeitig steigt mit zunehmender Verwendung des WebGL-Standards und Akzeptanz durch die Browser-Entwickler auch die Nachfrage nach spezialisierteren und leistungsstärkeren Frameworks.

Die in den letzten Jahren immer effizienter gewordenen JavaScript-Implementierungen der Browser-Hersteller stellen einen Umstand dar, der die Entwicklung der neuen Technologie positiv beeinflussen könnte. Setzt sich dieser Trend fort, kann der Zusatzaufwand, der durch WebGL-Frameworks entsteht, weiter minimiert werden.

Eine Herausforderung, denen sich WebGL-Frameworks annehmen müssen, sind die unterschiedlichen Nutzergruppen der WebGL-Technologie. Die Bedürfnisse der einzelnen Gruppen unterscheiden sich stark. Während Webentwicklern eine einfache Schnittstelle wichtig ist, benötigen Entwickler konventioneller 3D-Anwendungen leistungsstarke Bibliotheken mit detaillierten Schnittstellen zur Optimierung ihrer Anwendungen. Die aktuellen Frameworks richten sich dabei meist an Webentwickler. Dabei bleibt abzuwarten, ob sich ein Framework zum sicheren Standard entwickelt oder ob auf lange Sicht mehrere Frameworks parallel existieren. Der neue Standard steht gerade erst am Anfang seiner Entwicklung. WebGL eröffnet neue Möglichkeiten für Webanwendungen und könnte den Browser als Applikationsplattform für 3D-Anwendungen revolutionieren.

Literaturverzeichnis

[Belmonte, 2011] Belmonte, N. G. (2011). PhiloGL: A WebGL Framework for Data Visualization, Creative Coding and Game Development. http://www.senchalabs.org/philogl/. [Abgerufen am 21.05.2012].

[Cabello, 2011] Cabello, R. (2011). mrdoob/three.js. https://github.com/mrdoob/three.js/. [Abgerufen am 13.06.2012].

[Cabello, 2012] Cabello, R. (2012). Documents and Specifications . mrdoob/three.js. https://github.com/mrdoob/three.js/wiki. [Abgerufen am 13.06.2012].

[Demeyer et al., 1999] Demeyer, S., Ducasse, S., & Lanza, M. (1999). A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualization. In *Proceedings of the 6th Working Conference on Reverse Engineering* (pp. 175–186).

[Deveria, 2012] Deveria, A. (2012). Compatibility tables for support of WebGL in desktop and mobile browsers. http://caniuse.com/#feat=webgl. [Abgerufen am 16.06.2012].

[Emtinger & Lassus, 2011] Emtinger, M. & Lassus, O. (2011). GLOW. http://i-am-glow.com/. [Abgerufen am 12.05.2012].

[Engel, 2009] Engel, W. F. (2009). *ShaderX7: Advanced rendering techniques*. Australia: Course Technology/Cengage Learning.

[Foping et al., 2007] Foping, S. F., Chapman, P., & Bale, K. (2007). A Realistic Rendering of a School of Fish in Openscenegraph and C++.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[Glassner, 1989] Glassner, A. (1989). *An introduction to ray tracing*. Morgan Kaufmann.

[Google O3D Team, 2010a] Google O3D Team (2010a). o3d - WebGL implementation of O3D. http://code.google.com/p/o3d/. [Abgerufen am 02.06.2012].

[Google O3D Team, 2010b] Google O3D Team (2010b). The O3D API Blog. http://o3d.blogspot.de/. [Abgerufen am 02.06.2012].

[Graber, 2012] Graber, M. (2012). Konfigurierbare Annotation von Graphen für 3D-Visualisierungen.

[Gräther, 2011] Gräther, E. (2011). TrackballControls for Three.js. http://egraether.com/. [Abgerufen am 10.03.2012].

[Hanrahan & Haeberli, 1990] Hanrahan, P. & Haeberli, P. (1990). Direct WYSIWYG painting and texturing on 3D shapes. In *SIGGRAPH Computer Graphics*, volume 24 (pp. 215–223).: ACM.

[Heikkinen, 2012] Heikkinen, I. (2012). Animating a Million Letters Using Three.js - HTML5 Rocks. http://www.html5rocks.com/en/tutorials/webgl/million_letters/. [Abgerufen am 13.06.2012].

[Hildebrandt, 2012] Hildebrandt, J. (2012). Architekturkonzepte und ihre Implementierung für native und webbasierte Graphvisualisierung.

[Honauer, 2012] Honauer, K. (2012). WebGL-Basiertes Rendering von interaktiven Graphvisualisierungen.

[Kaneda, 2011] Kaneda, D. (2011). HTML5 Framework for Desktop and Mobile Devices. Build HTML5 Apps for Any Browser. | Sencha. http://www.sencha.com/. [Abgerufen am 21.05.2012].

[Kay, 2009] Kay, L. (2009). SceneJS - WebGL Scene Graph Library. http://scenejs.org/. [Abgerufen am 23.05.2012].

[Kay, 2010] Kay, L. (2010). SceneJS 2.0 Release. http://scenejs.org/releases.html. [Abgerufen am 23.05.2012].

[Kay, 2011a] Kay, L. (2011a). A 3D Scene Graph Engine on WebGL. https://github.com/xeolabs/scenejs. [Abgerufen am 23.05.2012].

[Kay, 2011b] Kay, L. (2011b). Fast Ray Picking in SceneJS - Xeolabs. http://blog.xeolabs.com/ray-picking-in-scenejs. [Abgerufen am 23.05.2012].

[Kay, 2011c] Kay, L. (2011c). FrameBuffer - xeolabs/scenejs Wiki. https://github.com/xeolabs/scenejs/wiki/frameBuf. [Abgerufen am 23.05.2012].

[Langelier et al., 2005] Langelier, G., Sahraoui, H., & Poulin, P. (2005). Visualization-based Analysis of Quality for Large-scale Software Systems. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05 (pp. 214–223). New York, NY, USA: ACM.

[Lanza & Marinescu, 2006] Lanza, M. & Marinescu, R. (2006). *Object-Oriented Metrics in Practice. Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems.* Springer Verlag.

[Marrin, 2011] Marrin, C. (2011). *WebGL Specification 1.0.* Technical report, Khronos WebGL Working Group. https://www.khronos.org/registry/webgl/specs/1.0. [Abgerufen am 16.06.2012].

[Perscheid et al., 2012] Perscheid, M., Cassou, D., & Hirschfeld, R. (2012). Test Quality Feedback - Improving Effectivity and Efficiency of Unit Testing. In *Proceedings of the 10th Conference on Creating, Connecting and Collaborating through Computing (C5 2012).*

[Phong, 1975] Phong, B. (1975). Illumination for computer generated pictures. In *Communications of the ACM*, volume 18 (pp. 311–317).: ACM.

[Pinson, 2011] Pinson, C. (2011). osg.js. http://osgjs.org/. [Abgerufen am 02.06.2012].

[Ramson, 2012] Ramson, C. (2012). Effiziente Transformation von serialisierten Software-Analysedaten in Graphdatenstrukturen.

[Reiners, 2002] Reiners, D. (2002). Scene Graph Rendering.

[Rusdorf, 2008] Rusdorf, S. (2008). *Aspekte der Echtzeit-Interaktion mit virtuellen Umgebungen*. PhD thesis, Universitätsbibliothek der Technischen Universität.

[Sedgewick & Wayne, 2011] Sedgewick, R. & Wayne, K. (2011). *Algorithms*. Addison-Wesley.

[Simpson, 2009] Simpson, R. J. (2009). *GLSL - The OpenGL ES Shading Language 1.0*. Technical report, Khronos GLSL. http://www.opengl.org/documentation/glsl. [Abgerufen am 16.06.2012].

[Vukicevic & Russell, 2011] Vukicevic, V. & Russell, K. (2011). *Typed Array Specification 1.0*. Technical report, Khronos WebGL Working Group. https://www.khronos.org/registry/typedarray/specs/1.0. [Abgerufen am 16.06.2012].

[Wang & Qian, 2010] Wang, R. & Qian, X. (2010). *OpenSceneGraph 3.0: Beginner's guide*. Birmingham and U.K: Packt Pub.

[WebGL Working Group, 2011] WebGL Working Group, K. (2011). WebGL - OpenGL ES 2.0 for the Web. http://www.khronos.org/webgl. [Abgerufen am 16.06.2012].

[Wright et al., 2007] Wright, R. S., Lipchak, B., & Haemel, N. (2007). *OpenGL SuperBible: Comprehensive Tutorial and Reference*. Addison-Wesley Professional.

[Zwiener, 2012] Zwiener, J. (2012). Konzepte zur testgetriebenen Entwicklung einer webbasierten Anwendung zur Graphvisualisierung.

Selbstständigkeitserklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

Das einleitende Kapitel wurde bis einschließlich Abschnitt 1.4. gemeinsam von den Teilnehmern des Bachelorprojekts verfasst. Die Teilnehmer waren Maria Graber, Jens Hildebrandt, Katrin Honauer, Stefan Lehmann, Cathleen Ramson, Mandy Roick und Jakob Zwiener.

Potsdam, den 29. Juni 2012

_____

Stefan Lehmann