

Fakultät für Bauingenieur- und Vermessungswesen  
Lehrstuhl für Computergestützte Modellierung und Simulation  
Prof. Dr.-Ing. André Borrmann

# Webbasierte 3D-Postvisualisierung von Fußgängersimulationsdaten

**Daniel Büchele**

Bachelorarbeit

Autor: Daniel Büchele  
Matrikelnummer: 10022975 (LMU)  
BetreuerIn: Dipl.-Inf. Angelika Kneidl  
Mustafa K. Isik  
Anmeldedatum: 30. April 2012  
Abgabedatum: 17. September 2012



## Beteiligte Organisationen

Fakultät für Bauingenieur- und Vermessungswesen  
Lehrstuhl für Computergestützte Modellierung und Simulation  
Technische Universität München  
Arcisstraße 21  
D-80333 München

Ludwig-Maximilians-Universität München  
Institut für Informatik  
LFE Medieninformatik  
Amalienstraße 17  
Vordergebäude, 5. Stock, Zimmer 507  
D-80333 München

## Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe. Diese Arbeit wurde zuvor nicht bei einer anderen akademischen Institution eingereicht oder anderweitig veröffentlicht.

München, 17. September 2012

---

Daniel Büchele

Daniel Büchele  
Clermontstr. 10  
D-82131 Gauting  
e-Mail: daniel@buechele.cc

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
<b>2 Moderne Webtechnologien</b>	<b>3</b>
2.1 Der HTML5-Standard . . . . .	3
2.1.1 DOM und JavaScript APIs . . . . .	4
2.1.2 Das canvas-Element . . . . .	5
<b>3 Dreidimensionale Computergrafik im Browser</b>	<b>6</b>
3.1 Historische Entwicklung der 3D-Technologie im Web . . . . .	6
3.2 Deklarative und imperative 3D-Darstellung in Browsern . . . . .	7
<b>4 WebGL</b>	<b>8</b>
4.1 Technik und Funktionsumfang . . . . .	9
4.1.1 Die WebGL-Rendering-Pipeline . . . . .	10
4.2 Verbreitung und Implementierung in den verschiedenen Browsern . . . . .	10
4.2.1 Vergleich der WebGL-Fähigkeiten ausgewählter Browser . . . . .	11
4.2.2 Marktanteile der Webbrowser . . . . .	11
4.2.3 Mobile Verfügbarkeit von WebGL . . . . .	12
4.3 Sicherheit in WebGL . . . . .	12
4.3.1 Denial-of-Service-Attacken . . . . .	13
4.3.2 Zugriff auf den Frame-Buffer . . . . .	13
4.3.3 Cross-Domain Texturen . . . . .	13
4.4 WebGL-Bibliotheken . . . . .	14
4.4.1 die THREE.js-Bibliothek . . . . .	14
<b>5 Fußgängersimulation und Visualisierung</b>	<b>16</b>
5.1 SumoViz . . . . .	17
5.2 SumoViz3D . . . . .	17
<b>6 Serverkomponenten: CouchDB und node.js</b>	<b>19</b>
6.1 Importieren der Simulationsdaten . . . . .	20
6.1.1 Import der Fußgängerdaten . . . . .	20
6.1.2 Import der Geometrie . . . . .	21
6.1.3 Import der Gruppendaten . . . . .	21
6.2 Ausgabe der Simulationsdaten . . . . .	21
6.2.1 Ausgabe der Fußgängerdaten . . . . .	22
6.2.2 Ausgabe der Geometrie- und Gruppendaten . . . . .	23

<b>7 Clientseitige Entwicklung von SumoViz3D</b>	<b>24</b>
7.1 Das Interface von SumoViz3D . . . . .	24
7.2 Darstellung der Geometrie . . . . .	26
7.2.1 Skalierung der Geometrie . . . . .	26
7.2.2 Darstellung der Grundebene und des Gitternetzes . . . . .	27
7.2.3 Darstellung von Objekten des Typ “obstacle” . . . . .	27
7.2.4 Darstellung von Objekten des Typ “wall” . . . . .	28
7.2.5 Darstellung anderer Objekte . . . . .	28
7.2.6 Veränderung während der Laufzeit . . . . .	28
7.3 Animationen in JavaScript . . . . .	31
7.4 Steuerung der Kamera . . . . .	33
7.5 Darstellung der Fußgängerdaten . . . . .	35
7.5.1 Fußgänger als 3D-Objekte . . . . .	35
7.5.2 Fußgänger als Partikelsystem . . . . .	36
7.5.3 Einfärbung der Fußgängerobjekte . . . . .	36
<b>8 Analyse und Zusammenfassung</b>	<b>38</b>
8.1 Leistungsanalyse von SumoViz3D . . . . .	38
8.1.1 Analyse mit bis zu 10000 Fußgängern . . . . .	38
8.2 Mögliche Erweiterungen für SumoViz3D . . . . .	40
8.3 Zusammenfassung . . . . .	41
<b>A Beigefügte CD</b>	<b>43</b>

# Kapitel 1

## Einleitung

Das Nutzungsverhalten des Internets hat sich in den letzten Jahren sehr gewandelt. Noch vor fünf Jahren bestand die Nutzung großteils aus dem Betrachten einzelner Webseiten, die als Dokumente wahrgenommen wurden. Besucht man heute eine der modernen Webseiten, so kann man schon lange nicht mehr von einem Dokument sprechen. Vielmehr sind Webseiten zu Webapplikationen geworden. Anwendungen deren Betriebssystem der Browser ist. Den größten Schritt in diese Richtung geht Google mit *Chrome OS*, einem Betriebssystem das nur noch aus einem Browser besteht. Alle Applikationen laufen ausschließlich im als Webanwendungen im Browser. Die Verfügbarkeit verschiedener Webapplikationen in allen Bereichen der Computernutzung hat enorm zugenommen und es gibt kaum mehr Aufgaben, für die ausschließlich native Anwendungen zur Verfügung stehen. So scheint Googles Ziel, den Browser als neues Betriebssystem zu etablieren, nicht in all zu weiter Ferne.

Eine Reihe an Voraussetzungen muss erfüllt werden, damit der Benutzer von Webanwendungen im Vergleich zur Nutzung nativer Applikationen keine Einbußen hinnehmen muss: Die Geschwindigkeit der Internetverbindung muss ausreichend hoch sein, um die Applikation und deren Inhalte ohne lange Wartezeiten zu übertragen. Anders als bei nativen Applikationen muss die Applikation selbst bei jeder Nutzung mitübertragen werden. Obwohl der Browser eine zusätzliche Schicht zwischen Applikation und Prozessor darstellt, darf die Ausführung nicht wesentlich langsamer sein als die nativen Applikationen. Sowohl für die Geschwindigkeit der Internetverbindungen, als auch für die Geschwindigkeit der Ausführung ist bereits heute ein Niveau erreicht, das die Nutzung von Webapplikationen ermöglicht und für die nächsten Jahre ist noch ein deutliches Wachstum in beiden Bereichen zu erwarten.

Damit aus dem Browser eine Umgebung zur Ausführung von Applikationen wird, müssen die Hardwarekomponenten des Computers aus dem Browser heraus verfügbar sein. Diese Schnittstellen (*APIs*) finden unter dem Schlagwort *HTML5* Einzug in moderne Browser und ermöglichen so beispielsweise den Zugriff auf das Adressbuch des Nutzers, seinen aktuellen Standort, Kamera und Mikrofon oder auch Systemkomponenten wie die Grafikkarte oder den Speicher [spec]. All das sind Funktionen, die native Applikationen schon immer nutzen konnten, die für Webanwendungen aber erst seit kurzer Zeit zur Verfügung stehen. Das stellt die Browserhersteller mitunter auch vor Sicherheitsprobleme, da diese Zugriffe keinesfalls unauthorisiert erfolgen dürfen.

Vorteilhaft an Webapplikationen ist unter anderem die schnelle und einfache Verfügbarkeit.

Ohne Einlegen eines Datenträgers und ohne Installationsprozess steht die Anwendung sofort zur Verfügung. Da die Daten bereits online gespeichert sind, ist Kollaboration und Synchronisation zwischen mehreren Rechnern und Benutzern einfacher möglich.

Für den Applikations-Entwickler stellt der Browser eine Möglichkeit zur plattformübergreifenden Entwicklung dar. Durch Allianzen der Browserhersteller und die Festsetzung gemeinsamer Standards ist es, abgesehen von kleinen Anpassungen, meist möglich eine Webapplikation unter beliebigen Betriebssystemen und Browsern auszuführen. In den Standards definierte Technologien erfordern üblicherweise auch keine Installation eines Plug-Ins. Lediglich die unterschiedliche Verfügbarkeit verschiedener Technologien ist bisher noch ein Problem, das der Entwickler bedenken muss.

Die Entwicklung und der Einsatz von Webapplikationen wird in den kommenden Jahren weiter zunehmen und viele native Anwendungen vom Markt verdrängen. Trotzdem wird es sicherlich weiterhin Bereiche geben, in denen der Einsatz einer nativen Anwendung sinnvoller oder notwendig ist.

## Kapitel 2

# Moderne Webtechnologien

Das *World Wide Web Consortium (W3C)* standardisiert Technologien die das Internet betreffen. Das Konsortium besteht aus über 300 Mitgliedern<sup>1</sup> [W3m], die über die Entstehung neuer Standards beraten und entscheiden. Da die Entstehung und Veröffentlichung neuer Standards beim W3C auf Grund der hohen Mitgliederzahl und demokratischen Strukturen oft sehr lange dauert, gründeten Vertreter von Opera, Apple und Mozilla die *Web Hypertext Application Technology Working Group (WHATWG)*. Eine Arbeitsgruppe, deren Mitgliedschaft nur auf Einladung erfolgt und deren endgültige Entscheidungen einem Vorsitzenden unterliegen. Erarbeitete Entwürfe reicht die WHATWG beim W3C als Vorschlag zur Standardisierung ein. Einen Vorschlag der WHATWG nahm das W3C auch als Basis für die Spezifizierung von *HTML5*. [W3a] [Kei10]

### 2.1 Der HTML5-Standard

In der Spezifikation des W3C wird mit “HTML5” die Weiterentwicklung der Auszeichnungssprache HTML beschrieben. Die WHATWG dehnt den Begriff etwas weiter aus und versammelt unter dieser Bezeichnung auch einige JavaScript-Schnittstellen, wie etwa für das canvas-Element (siehe Kapitel 2.1.2), das *WebSockets*-Netzwerkprotokoll oder die Nutzung von lokalem Speicher (*WebStorage*). Im Sprachgebrauch wird der Begriff “HTML5” oft noch weiter gefasst und beinhaltet diverse weitere Technologien, die im Zuge moderner Web-Applikationen entwickelt wurden. Darunter fallen dann die Grafikbibliothek *WebGL*, die *Geo-Location*-API zur Abfrage des Standorts oder auch der *CSS3*-Standard zur Gestaltung von Webseiten (siehe Abbildung 2.1).

Eine der Design-Entscheidungen bei der Entwicklung von HTML5 ist, den bestehenden Standard HTML 4.01 nicht zu ersetzen, sondern weiterzuentwickeln. Dadurch entsprechen die meisten Webseiten im HTML-4.01-Standard nach Anpassung der Dokumententypdeklaration direkt dem HTML5-Standard. [Pil10]

Die Spezifikation hat den Status “Last Call”<sup>2</sup>. Damit fordert das W3C auf letzte Änderungsvorschläge einzureichen, bevor der Standard im Jahr 2014 den Status “Recommendation”

---

<sup>1</sup>Stand: August 2012

<sup>2</sup>Stand: 22. August 2012

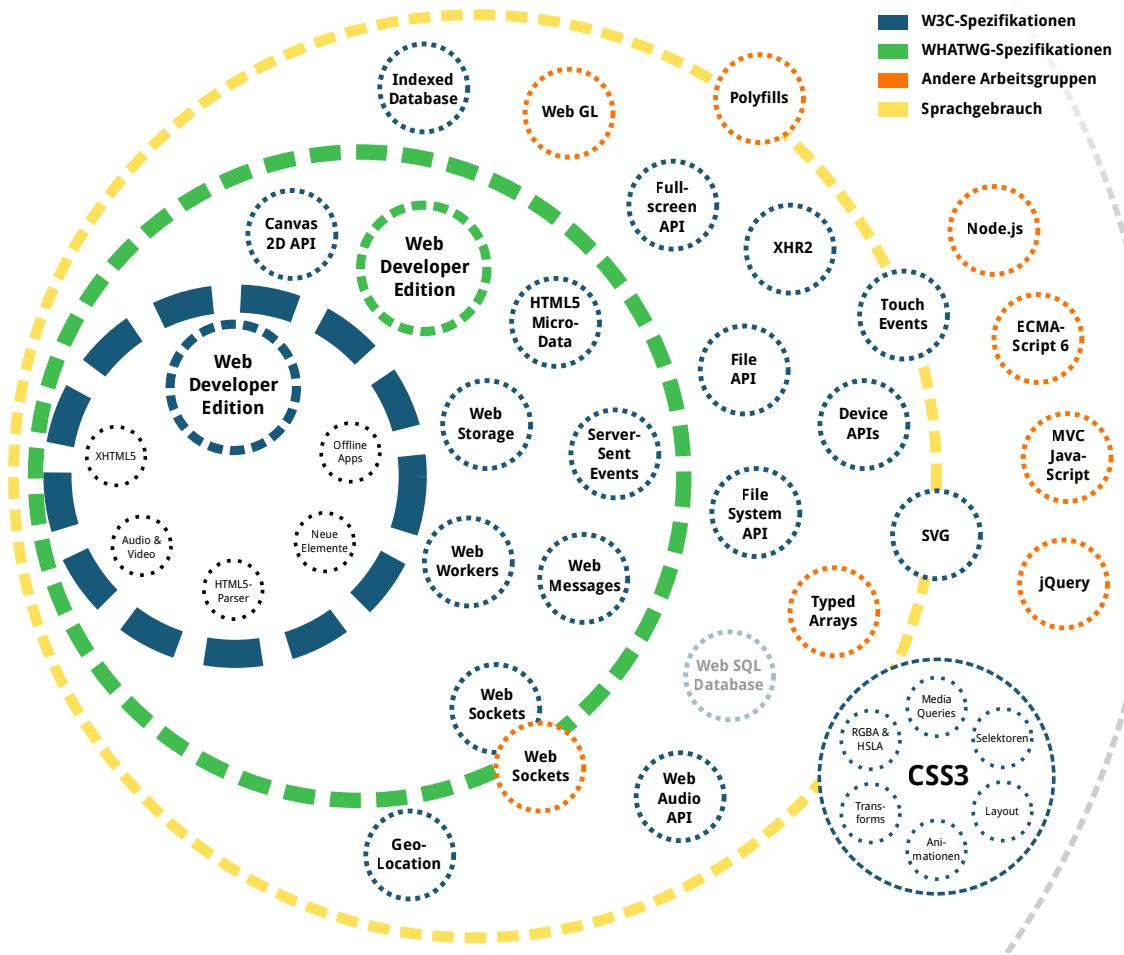


Abbildung 2.1: HTML5 Spezifikations-Übersicht (CC-BY-3.0 Peter Kröner) [spec]

erhalten soll und damit final ist. Da jedoch die Implementierung in den einzelnen Browsern weit fortgeschritten ist und kaum noch Änderungen am Standard erwartet werden, kann er bereits jetzt eingesetzt werden. [W3l]

### 2.1.1 DOM und JavaScript APIs

Nach dem parsen bildet der Browser intern die hierarchische Struktur eines HTML-Dokuments in Form eines Baumes ab. Diese Abbildung wird als *Document Object Model (DOM)* bezeichnet. Der Zugriff auf den DOM-Baum ist mittels einer Schnittstelle (*Application Programmable Interface (API)*) für JavaScript-Anwendungen möglich. Elemente, Attribute und Texte sind als Knoten-Objekte angelegt und können ausgelesen oder manipuliert werden. [MDN1]

Mit der zunehmenden Nutzung von Web-Applikationen sind auch die Anforderungen an die Browser gestiegen. Zur besseren Interaktion des Nutzers mit dem Browser wurden im Umfeld des HTML5-Standards viele weitere JavaScript-APIs definiert und vom W3C standardisiert (vgl. Abbildung 2.1). [Kei10] [Pil10]

### 2.1.2 Das canvas-Element

Das *canvas*-Element, wie in HTML5 definiert, ist ein rechteckiges Blockelement, das ohne weitere Anweisungen komplett unsichtbar ist: Es hat keinen sichtbaren Rahmen und der Inhalt des Elements ist transparent. In einem Dokument können sich beliebig viele canvas-Elemente befinden, die komplett unabhängig von einander sind. Jedes canvas ist im DOM-Baum verankert und kann mit einer ID versehen werden. Innerhalb des canvas-Elements kann alternativer Inhalt angegeben werden, der angezeigt wird wenn der Browser die Darstellung von canvas-Elementen nicht unterstützt. Eine beispielhafte Nutzung des Elements kann wie folgt aussehen: [Kei10] [Pil10]

```
<canvas id="c" width="300" height="200">Fallback</canvas>
```

Das Element definiert ein auflösungsabhängiges Bitmap, das in Echtzeit mit Grafikinhalten gefüllt werden kann. Laut Spezifikation definiert canvas einen JavaScript-Funktionsaufruf `canvas.getContext(contextId [, ... ])` bei dem ein Schlüsselwort in Form eines Strings als `contextId` übergeben wird. [W3c] Die WHATWG spezifiziert zwei mögliche Kontexte: `2d` und `webgl`. [WHc] Der Funktionsaufruf liefert ein Objekt zurück das die jeweilige API bereitstellt oder `null` falls ein entsprechender Kontext nicht unterstützt wird.

Mittels der 2D-API können einfache Zeichenoperationen ausgeführt werden. Dazu gehören Linien, Pfade, Rechtecke, Ellipsen, aber auch Texte oder Bilder können eingefügt werden. All diese Operationen werden direkt in der CPU berechnet und sind somit nicht hardwarebeschleunigt. Animationen können durch ständiges Neuzeichnen mittels JavaScript-Timer-Funktionen realisiert werden. [Whc2] [Pil10]

## Kapitel 3

# Dreidimensionale Computergrafik im Browser

### 3.1 Historische Entwicklung der 3D-Technologie im Web

Erste Bemühungen dreidimensionale Inhalte in den Webbrowser zu bringen begannen im Jahr 1994. Angelehnt an die Nutzung von HTML für die Darstellung von Webseiten sollte eine Auszeichnungssprache für die Darstellung dreidimensionaler Inhalte geschaffen werden. Anforderungen an die neue Sprache waren Plattformunabhängigkeit, Erweiterbarkeit und Nutzbarkeit auch bei geringer Internet-Bandbreite. Das zu diesem Zweck gegründete *Web3D*-Konsortium entwickelte die *Virtual Reality Modeling Language (VRML)*. Am 26. Mai 1995 wurde die finale Spezifikation von *VRML 1.0* veröffentlicht. Dort wird ein einfaches textbasiertes Format definiert, in dem Objekte geschachtelt abgelegt werden können. Diese Objekte können Kameras, Lichter, Materialien, aber auch dargestellte 3D-Objekte oder Transformationen sein. Zur Speicherung der Szenen wird die Dateiendung *.wrl* (für engl. *world*) vorgegeben. [VRML1a] 1997 wurde *VRML 2.0 (VRML97)* fertiggestellt und als ISO-Standard 14772-1:1997 definiert. Unter anderem wurde der VRML-1.0-Standard um Animationen und Nutzerinteraktionsmöglichkeiten erweitert. [VRMLa] [CB97]

Für die Darstellung von VRML-Inhalten gibt es einige Browser-Plugins, jedoch integrierte nahezu kein Browser-Hersteller den Standard direkt in sein Produkt. In den folgenden Jahren entstanden viele weitere Dateiformate zur Speicherung von 3D-Szenen mit Augenmerk auf die Darstellung im Browser. Unter anderem trat das XML-basierte Format *X3D* die Nachfolge von VRML an. Das Web3D-Konsortium schlägt vor den X3D-Standard komplett in HTML5 zu integrieren, jedoch hat das W3C das bisher abgelehnt: “*Embedding 3D imagery into XHTML documents is the domain of X3D, or technologies based on X3D that are namespace-aware.*” [W3no] Ähnlich wie bei *SVG* oder *MathML* soll so eine native Darstellung ohne Plugin ermöglicht werden. [Web3]

## 3.2 Deklarative und imperative 3D-Darstellung in Browsern

Die Darstellung von zwei- und dreidimensionalen Grafiken im Browser kann grundlegend auf zwei verschiedene Arten erfolgen: Zum einen ist es möglich den Szenengraph als Teil des HTML-Dokuments zu deklarieren. Somit sind die einzelnen Objekte der Grafik gleichberechtigt mit HTML-Elementen auf der Seite. Jedes Objekt der Grafik ist im DOM-Baum integriert und kann per CSS oder JavaScript selektiert und modifiziert werden. Zur Darstellung zweidimensionaler Grafiken ist das *SVG*-Format im HTML5-Standard aufgenommen. Das Äquivalent für dreidimensionale Grafiken, X3D, ist wie in Kapitel 3.1 beschrieben nicht Teil des Standards. [X3D]

Zum anderen können Grafiken imperativ auf eine Zeichenfläche gezeichnet werden. Als Zeichenfläche wird das *canvas*-Element genutzt und ist dabei das einzige Objekt der Grafik, das im DOM-Baum verankert ist. Inhalte die auf die Zeichenfläche gezeichnet wurden können nicht mehr verändert werden. Die Zeichenfläche kann lediglich komplett oder in Ausschnitten geleert und/oder überzeichnet werden. HTML5 definiert die 2D-Fähigkeiten des *canvas*-Elements und Verwendung von *WebGL* für 3D-Inhalte. [W32d]

	2D	3D
Deklarativ	SVG	X3D <sup>1</sup>
Imperativ	canvas	WebGL

Tabelle 3.1: Deklarative und imperative Grafikdarstellung im Web [X3D]

---

<sup>1</sup>nicht vom W3C standardisiert

## Kapitel 4

# WebGL

Die *Khronos Group* ist ein Konsortium aus über 100 Firmen, die es sich seit Januar 2000 zum Ziel gesetzt hat, offene Standards für Mediendarstellung auf Computern, mobilen Geräten und in integrierten Systemen zu definieren. Als gemeinnützige Organisation arbeitet die Gruppe nicht gewinnorientiert und finanziert sich lediglich aus den jährlichen Beiträgen ihrer Mitglieder. In verschiedenen Arbeitsgruppen sollen Industrie-Standards entwickelt und lizenzzfrei zur Verfügung gestellt werden. Dazu gibt es das *Khronos IP Framework* – eine Vereinbarung zwischen den Khronos-Mitgliedern, keine Patentstreitigkeiten bezüglich der Khronos-Standards gerichtlich auszutragen. [Tre11]

Der älteste Standard der Khronos-Gruppe ist *OpenGL (Open Graphics Library)*, der 1992 von *Silicon Graphics Inc. (SGI)* entwickelt wurde. OpenGL spezifiziert eine hardwareunabhängige Software-Schnittstelle (API) zum Grafikprozessor mit mehreren hundert Befehlen, einer kleinen Menge geometrischer Primitive und einer eigenen Shader-Programmiersprache *OpenGL Shading Language (GLSL)*. Die meisten großen Betriebssysteme, darunter Windows, Mac OS und Linux, unterstützen OpenGL. [Shr09] [SGI] [Khr]

Als proprietäre Alternative zu OpenGL bietet Microsoft *Direct3D* für die Windows- und XBox-Plattform an. Zudem gibt es eine angepassten Version für Microsofts mobiles Betriebssystem *Windows Phone Series*.

Ausgehend vom OpenGL-Standard entwickelte die Khronos Group einen weiteren Standard: *OpenGL for Embedded Systems (OpenGL ES)* ist eine Teilmenge von OpenGL, die speziell für den Einsatz auf mobilen Geräten wie etwa Smartphones oder Spielekonsolen entwickelt wurde und eine hohe Verbreitung unter anderem in Googles Android [Goo] oder Apples iOS [App] findet. Applikationen die gegen OpenGL ES geschrieben wurden laufen auch unter der entsprechenden Version von OpenGL. Andersherum ist das nicht der Fall.

WebGL bringt nun die Möglichkeit OpenGL-ES-2.0-Inhalte im Browser darzustellen und dabei die Hardware-Beschleunigung der Grafikkarte zu nutzen. Bisher konnten Webinhalte lediglich den Hauptprozessor (CPU) für ihre Berechnungen nutzen. Grafikkarten sind hochgradig für die typischen Berechnungen von 3D-Darstellungen optimiert und können die Prozessorleistung bei weitem übertreffen. Mittels der WebGL-Technologie können Applikationen die zum Beispiel für mobile Geräte konzipiert wurden ohne größere Anpassungen im Browser ausgeführt werden. Die Anwendungsgebiete sind vielfältig: Von Visualisierungen, Simulationen, Spielen, bis zu Kunst ist alles denkbar. [Cab] [Ope]

## 4.1 Technik und Funktionsumfang

Die WebGL-Programmierung erfolgt in JavaScript und die Programmierung der Shader in GLSL, deren Syntax ähnlich zu JavaScript ist. Shader sind das grundlegende Konzept von WebGL und OpenGL. Dabei handelt es sich um Unterprogramme, die in der Grafikkarte ausgeführt werden und über die Darstellung der 3D-Szene entscheiden. In WebGL gibt es mit Vertex- und Fragment-Shader zwei unterschiedliche Arten, deren Funktionsweise in Kapitel 4.1.1 erläutert wird. Die Shader können frei programmiert werden, aber viele Grafikbibliotheken bringen bereits vordefinierte Shader mit (vgl. Kapitel 4.4).

Das canvas-Element wird als Zeichenfläche für die WebGL-Inhalte verwendet. Wie jedes andere HTML-Element kann das canvas-Element über oder unter anderen Elementen des Dokuments dargestellt werden. So ist auch die Überlagerung von HTML- und WebGL-Inhalten möglich.

Mittels des in Kapitel 2.1.2 beschriebenen Funktionsaufrufs kann die Initialisierung eines WebGL-Kontexts wie Listing 4.1 aussehen. [HTML] Zunächst wird der Funktionsaufruf der Initialisierung auf den `onload`-Event-Listener gelegt (Zeile 11) und somit die Funktion `init()` beim Laden der Seite aufgerufen. Mittels der Element-ID wird das canvas-Element selektiert (Zeile 3) und dessen WebGL-Kontext aufgerufen (Zeile 4). Ist der Kontext nicht vorhanden, wird WebGL nicht unterstützt (Zeilen 5-7), andernfalls steht dann der WebGL-Kontext über die Variable `gl` zur Verfügung.

```

1 <script type="text/javascript">
2     function init() {
3         canvas = document.getElementById("c");
4         gl = canvas.getContext("webgl");
5         if (!gl) {
6             return; /*WebGL wird nicht unterstützt*/
7         }
8         ...
9     }
10
11     window.onload = init;
12 </script>
13 <canvas id="c"></canvas>
```

Listing 4.1: Initialisierung eines WebGL-Kontexts

Zusätzlich können Vertex- und Fragment-Shader mit folgendem Mark-Up ausgezeichnet werden und dann in GLSL programmiert werden:

```

1 <script id="vshader" type="x-shader/x-vertex">
2     ... /* Vertex-Shader */
3 </script>
4 <script id="fshader" type="x-shader/x-fragment">
5     ... /* Fragment-Shader */
6 </script>
```

Listing 4.2: Deklaration der WebGL-Shader

WebGL verwendet ein eigenes kartesisches Koordinatensystem das unabhängig von der Größe und Auflösung des canvas-Elements ist. Der Wertebereich der x-, y- und z-Achse liegt im

Intervall  $[-1; 1]$ . Die x-Achse beginnt auf der linken Seite mit  $-1$ , die y-Achse hat den Wert  $-1$  an der unteren Bildkante. Die z-Achse zur Tiefenbestimmung hat für die nächsten Objekte den Wert  $-1$  und für die entferntesten den Wert  $1$ . Für die Darstellung auf dem Bildschirm werden die WebGL-Koordinaten dann auf die canvas-Koordinaten umgerechnet.

#### 4.1.1 Die WebGL-Rendering-Pipeline

Folgende Schritte werden bei der Berechnung eines Bilds bei der OpenGL-ES-/WebGL-Technologie durchlaufen. Der Ablauf ist vereinfacht dargestellt und weitaus identisch mit der Arbeitsweise anderer Grafik-Bibliotheken. [CJ12] [Cab] [Shr09]

1. Wie für 3D-Renderingprozesse üblich, verwendet WebGL polygonale Modellierung. Das bedeutet, dass Objekte aus Polygonen zusammengesetzt werden. Üblicherweise und auch in WebGL werden dafür Dreiecke verwendet. In JavaScript-Code wird nun eine Menge von Punkten (Vertices) definiert und in einem Array gespeichert. Zu jedem Vertex können neben der Position im dreidimensionalen Raum auch die Farbe, Transparenz und Texturkoordinaten gespeichert werden. Ein zweites Array ordnet die Indizes des Vertex-Arrays zu Dreiergruppen zusammen und gibt die Reihenfolge an, in der die Punkte zu Dreiecken zusammengesetzt werden.
2. Das Vertex- und Index-Array werden dann an die Grafikkarte übergeben. Für jedes Vertex wird ein Unterprogramm, der so genannte **Vertex-Shader**, aufgerufen, der Operationen auf jedem einzelnen Vertex ausführen kann. Hier können Manipulationen an den Formen des Objekts vorgenommen werden. Außerdem wird mit Hilfe einer Matrixmultiplikation die Projektion des dreidimensionalen Punkts auf die zweidimensionale Fläche zur Anzeige auf dem Bildschirm berechnet.
3. Anschließend wird in der Grafikkarte das Bild **rasterisiert**. Dabei wird die zweidimensionale Projektion mit einem Pixelraster der Größe des canvas-Elements überlagert und jeder Punkt einem Pixel zugeordnet. Hier können auch Kantenglättungs-Mechanismen (Antialiasing) angewendet werden. Die Farben zwischen zwei Vertices können mit Hilfe der zugehörigen Farbwerte interpoliert werden.
4. Für jedes berechnete Pixel wird nun ein weiteres Unterprogramm aufgerufen: Der **Fragment-Shader** kann die Farbewerte des jeweiligen Pixels verändern und sorgt somit für die Darstellung von Texturen, Oberflächenmaterialien, Lichtern und Schatten.
5. Das fertige Pixelbild wird im **Frame-Buffer** der Grafikkarte abgelegt und kann von dort aus dann im canvas-Element der Webseite dargestellt werden.

## 4.2 Verbreitung und Implementierung in den verschiedenen Browsern

Mitglieder der Khronos Group sind mit Google (Chrome), Apple (Safari), Mozilla (Firefox) und Opera unter anderem einige namenhafte Browser-Hersteller, die gemeinsam an der Entwicklung von WebGL arbeiten. Deshalb geschieht die Adaption neuer Funktionen in den genannten Browsern meist schnell und flächendeckend.

### 4.2.1 Vergleich der WebGL-Fähigkeiten ausgewählter Browser

Für den Vergleich wurden die fünf Browser mit den größten Marktanteilen herangezogen. Gemeinsam decken diese fünf Browser 98,41%<sup>1</sup> des Desktop-Marktes ab. [Sta1] Mobile Browser sind in diesem Vergleich ausgenommen und werden gesondert in Kapitel 4.2.3 behandelt.

Der WHATWG-Standard [Whc2] gibt für den Aufruf der Schnittstelle des WebGL-Kontexts das Schlüsselwort `webgl` vor, jedoch implementieren die verglichenen Browser alle ausschließlich das Schlüsselwort `experimental-webgl` und liefern für den Kontext `webgl` den Wert `null` zurück.<sup>2</sup> Solange die Spezifikation nicht final ist und der Standard nicht vollständig implementiert wurde wird dieses Vorgehen vom W3C und der WHATWG empfohlen. [W3c] [Whc2]

#### Microsoft Internet Explorer

Microsoft stuft die Implementierung von WebGL als zu gefährlich ein und verzichtet deshalb auf eine Integration in den *Internet Explorer*. Problematisch sieht Microsoft dabei die direkte Bereitstellung der Grafikkarte für beliebige Webseiten-Betreiber. Sicherheitslücken in Grafikkarten-Treibern die bisher nur lokal ausgenutzt werden konnten, stehen somit für das Internet offen (vgl. Kapitel 4.3). [MSFT] Als alternative Technologie schlägt Microsoft die Eigenentwicklung *Silverlight* vor, die seit der Version 5 ebenfalls hardwarebeschleunigte 3D-Darstellung unterstützt. [Sil]

Dennoch lässt sich mittels Plugin eines Fremdanbieters die Unterstützung für WebGL ab Internet Explorer Version 8 nachrüsten. [IEWeb]

### 4.2.2 Marktanteile der Webbrowser

Browser	Marktanteil gesamt	WebGL ab Version <sup>3</sup>	Release	Marktanteil WebGL-fähig
Google Chrome	33,29%	9.0	03.02.2011 [Chr]	32,89%
Internet Explorer	32,17%	keine WebGL Unterstützung		0,00%
Mozilla Firefox	24,14%	4.0	22.03.2011 [FF4]	22,13%
Apple Safari	7,06%	5.1	20.07.2011 [Saf]	3,41%
Opera	1,75%	12.00	14.06.2012 [Ope]	0,78%
<b>Summe</b>	<b>98,41%</b>			<b>59,21%</b>

Tabelle 4.1: Marktanteile der führenden Webbrowser [Sta1] [Sta3]

Basierend auf den Zahlen von StatCounter.com für Juni bis Juli 2012 haben WebGL-fähige Browerversionen einen Marktanteil von 59,21% (vgl. Tabelle 4.1). Jedoch muss der Anteil der Browser, die WebGL-Inhalte korrekt darstellen geringer angegeben werden, da durch Inkompatibilitäten mit speziellen Grafikkarten einige Browerversionen WebGL deaktivieren.

<sup>1</sup>Stand: Juli 2012

<sup>2</sup>Stand: August 2012 (Chrome 21, Firefox 14, Safari 6.0, Opera 12.00)

<sup>3</sup>Für den Vergleich wurden nur finale Versionen der jeweiligen Browser herangezogen. Alpha-, Beta- und Preview-Versionen sind ausgenommen und waren teilweise schon wesentlich früher verfügbar.

Hinzu kommt, dass Safari mit deaktiviertem WebGL ausgeliefert wird, welches erst vom Benutzer manuell aktiviert werden muss. Daher lassen sich ohne eigener Erhebung dieser Daten keine verlässlichen Aussagen über die genaue Verbreitung von Browsern mit aktivierter WebGL-Fähigkeit treffen.

#### 4.2.3 Mobile Verfügbarkeit von WebGL

Es wurde die Verfügbarkeit des WebGL-Standards auf den beiden führenden mobilen Betriebssystemen verglichen. [Sta2] Dabei wird die grundlegende Verfügbarkeit der OpenGL-Technologie sowie die Implementierung der WebGL-Technologie im mitgelieferten Browser betrachtet.

##### Apple iOS und Mobile Safari

Apple implementiert in iOS<sup>4</sup> OpenGL ES 1.1 und 2.0 für die Darstellung von 2D- und 3D-Inhalten. Applikationsentwickler können die Standards für die Darstellung ihrer Inhalte nutzen. [AoGL] Jedoch ist die Nutzung von OpenGL in Form von WebGL im systemeigenen Browser *Mobile Safari* nicht möglich. Für in Applikationen eingebettete Web-Darstellungen lässt sich WebGL aktivieren, jedoch widerspricht die Nutzung Apples Richtlinien für Entwickler. Lediglich innerhalb Apples webbasiertem Werbenetzwerk *iAd* ist die Nutzung von WebGL möglich. Außerdem kann die Darstellung von *CSS3-Animationen* im Browser auf Hardwarebeschleunigung zurückgreifen. [App]

##### Android

Für die Entwicklung von Applikationen unterstützt die Android-Plattform ebenso den OpenGL ES 1.1 und ab Android 2.2 (API Level 8) auch den OpenGL ES 2.0 Standard. [Goo] WebGL ist im Android Browser nicht verfügbar. Sony entwickelte eine Modifikation für den Android Browser die WebGL implementiert. [SNE] Diese wird bisher nur auf dem *Sony Xperia* Smartphone eingesetzt.

## 4.3 Sicherheit in WebGL

Im Mai 2011 beschrieb *Context Information Security* mehrere mögliche Angriffszenarien mit Hilfe von manipulierten WebGL-Inhalten. [Con1] Generell stellt WebGL ein hohes Bedrohungspotential dar, da es den direkten Zugriff von Webseiten auf die Hardware des Rechners zulässt. Natürlich versuchen Browserhersteller, die Khronos Group und die Grafikkartenhersteller diese Angriffe zu verhindern, trotzdem sind diverse Szenarien denkbar, in denen WebGL dazu genutzt werden kann den Rechner anzugreifen.

---

<sup>4</sup>Stand: iOS 5.1.1

### 4.3.1 Denial-of-Service-Attacken

Mittels sehr komplexer 3D-Geometrien oder extrem aufwändiger Shaderprogramme wird die Grafikkarte so lange beansprucht, dass weder andere Programme noch das Betriebssystem die Grafikkarte nutzen können und der Rechner so unter Umständen abstürzt oder unbenutzbar wird. [Con2] Gegen diese Art von Angriff wurden bisher kaum Maßnahmen ergriffen. Seit Windows Vista implementiert Microsoft einen Sicherheitsmechanismus, der die Grafikkarte zurücksetzt, wenn diese für über zwei Sekunden nicht mehr reagiert. [Win] Dieses Vorgehen schlägt auch die Khronos Group für OpenGL-kompatible Grafikkarten in der Erweiterung *GL\_ARB\_robustness* vor, jedoch haben noch nicht alle Grafikkartenhersteller diese Erweiterung implementiert. Somit sind aktuell viele Systeme für diese Art von Angriff gefährdet. [KhrSec]

### 4.3.2 Zugriff auf den Frame-Buffer

Die WebGL-Technologie ermöglicht Webseiten den Zugriff auf den Inhalt des Frame-Buffers der Grafikkarte. Der Frame-Buffer ist ein gemeinsam genutzter Speicherbereich in dem alle auf dem Bildschirm dargestellten Inhalte liegen. Deshalb ist laut Spezifikation der Zugriff auf den Teil des Frame-Buffers beschränkt, der den Inhalt des canvas-Elements speichert. Jedoch kam es durch fehlerhafte Implementierungen des WebGL-Standards und Bugs in Grafikkarten-Treibern dazu, dass auch andere Inhalte des Frame-Buffers in das canvas-Element gezeichnet werden konnten. So konnten Informationen ausgelesen werden, die der Nutzer beispielsweise in anderen Applikationen oder auf seinem Desktop dargestellt hat. Seitens Khronos Group und der Grafikkartenhersteller wurden diese Fehler behoben. Es ist jedoch nicht ausgeschlossen, dass durch neue Sicherheitslücken dieser Angriff wieder ermöglicht wird. [Con2]

### 4.3.3 Cross-Domain Texturen

Das Nachladen externer Modelle oder Texturen kann den Zugriff auf fremde Ressourcen erfordern. Ein Sicherheitskonzept, das mit *Netscape Navigator 2.0* für clientseitige Skriptsprachen, wie beispielsweise JavaScript, eingeführt wurde ist die *Same-Origin-Policy (SOP)*. Diese verbietet clientseitige lesende Zugriffe, zum Beispiel per *XMLHttpRequest*, auf fremde Ressourcen. Sie verlangt, dass Zugriffe innerhalb einer Webseite nur auf Ressourcen der selben Domain, über den selben Port mittels des selben Protokolls erfolgen dürfen. Das betrifft nicht das Einbinden fremder Inhalte, wie das Einbetten eines Bildes, sondern den lesenden Zugriff eines Skripts auf solche Inhalte. Durch einen clientseitigen lesenden Zugriff könnten Informationen von fremden Seiten ausgelesen werden. Hätte ein Nutzer beispielsweise eine Banking-Seite geöffnet, könnte eine andere Seite deren Inhalte einlesen und zu einem fremden Server übertragen. Daher wird der clientseitige lesende Zugriff im Normalfall auf die jeweilige Domain beschränkt. [MDN]

Um dennoch entfernte Ressourcen nutzen zu können wurde *Cross-Origin-Resource-Sharing (CORS)* entwickelt. Dabei gibt der ausliefernde Server in seinem HTTP-Header an, welche fremden Seiten seine Ressourcen lesen dürfen. Dabei ist auch die Verwendung einer Wildcard möglich um den lesenden Zugriff für alle fremden Seiten zu ermöglichen. [W3cors] [Moz]

## 4.4 WebGL-Bibliotheken

Um den Umgang mit WebGL zu vereinfachen gibt es einige Bibliotheken. Diese stellen beispielsweise geometrische Grundformen wie Würfel, Kugeln, Zylinder, etc. zur Verfügung. Außerdem lassen sich somit WebGL-Aufrufe mittels JavaScript ausführen, ohne eigenen Shader-Code in *GLSL* nutzen zu müssen. Komplexe mathematische Berechnungen die für die Realisierung verschiedener Kameras müssen nicht in einem eigenen Vertex-Shader realisiert werden, sondern können als fertige Programme geladen werden. Ebenso ist für die Realisierung von Oberflächenmaterialien und Beleuchtungsmodellen keine Programmierung eines eigenen Fragment-Shaders nötig, da gängige Materialien und Beleuchtungsmodelle bereits als vordefinierte Shader vorhanden sind. Für die meisten Anwendungsfälle reicht die Nutzung der durch Kamera-, Material- und Beleuchtungsobjekten vordefinierten Shader aus und es muss kein Shadercode geschrieben werden. Die Programmierung kann dann ausschließlich in JavaScript erfolgen. Zusätzlich können unterschiedliche Implementierungen verschiedener Browser angefangen und für den Programmierer unsichtbar gemacht werden.

### 4.4.1 die THREE.js-Bibliothek

Entwickelt von *Ricardo Cabello Miguel* ist *THREE.js* eine der am weitesten verbreiteten Bibliotheken für WebGL. Seit 2010 wird *THREE.js* entwickelt und befindet sich aktuell in der Version r50<sup>5</sup>. Einige elementare Bestandteile der Bibliothek sollen im Folgenden vorgestellt werden. [THREE]

- **Renderer:** Neben der Ausgabe in den `webgl`-Kontext, ist in einem begrenzen Maße auch die Ausgabe in eine *SVG*-Datei, oder in den *2d*-Kontext des `canvas`-Elements möglich. Jedoch ist nur die WebGL-Ausgabe hardwarebeschleunigt. Für nicht WebGL-fähige Browser kann in einigen Anwendungsfällen eine der alternativen Ausgaben genutzt werden.
- **Kameras:** *THREE.js* hat zwei verschiedene Kamera-Objekte zur Verfügung. Eine orthographische Kamera die eine parallele Projektion auf die Sichtebene vornimmt, sowie eine perspektivische Kamera, die dem Verhalten des menschlichen Auge entspricht. Hierbei wird das Sichtfeld mit einem Winkel angegeben.
- **Geometrische Primitive:** Einige geometrische Primitive sind in der Bibliothek bereits angelegt und können verwendet werden. Darunter befinden sich: Würfel, Zylinder, Ebene, Kugel und Torus.
- **Lichter:** In *THREE.js* sind verschiedene Lichter definiert, die sich unterschiedlich auf die Szene auswirken. Das `AmbientLight` hellt die gesamte Szene auf, hat keine Richtung und wirft keine Schatten. Das `DirectionalLight` ist gerichtet, hat aber keinen Ursprung. Das Licht verläuft parallel und wirft Schatten. `SpotLight` und `PointLight` haben eine punktförmige Position. Beide strahlen das Licht kugelförmig ab, wobei es linear abgeschwächt wird. Nur `SpotLights` können Schatten werfen.

---

<sup>5</sup>Stand: 19. August 2012

- **Oberflächenmaterialien:** Die unterschiedlichen Oberflächenmaterialien wirken sich auf das Verhalten von Licht auf den Objekten aus. Das `BasicMaterial` ignoriert einfallendes Licht und stellt das Objekt gleichmäßig in der angegebenen Farbe dar. Für die realistische Darstellung bieten sich `PhongMaterial` und `LambertMaterial` an, die einfallendes Licht nach den jeweiligen Beleuchtungsmodellen verarbeiten und so die Farbdarstellung berechnen. Zusätzlich ist es möglich Grafiktexturen auf Objekten abzubilden.

## Kapitel 5

# Fußgängersimulation und Visualisierung

Bei der Planung von Großveranstaltungen oder der Konzeption neuer Gebäude werden Fußgängersimulationen eingesetzt um Evakuierungsszenarien zu simulieren und ggf. Katastrophen vorzubeugen. Analysen und Simulationen, warum beispielsweise auf Autobahnen ein Stau entsteht, sind mittlerweile problemlos machbar. Die Vorhersage, welchen Weg ein Fußgänger einschlagen wird ist dagegen ein größeres Problem. Während sich Autos auf festen Fahrbahnen bewegen und sich wesentlich strikter an Verkehrsregeln halten, haben Fußgänger mehr Freiheitsgrade in ihrer Bewegung. Spontan können sich Fußgänger dazu entschließen die Richtung zu ändern, versuchen Gedränge aus dem Weg zu gehen und verlieren sogar manchmal ihr Ziel aus den Augen.

Um das Verhalten der Fußgänger zu simulieren werden verschiedene Verfahren angewandt. Ein populäres Modell ist aus der Strömungslehre entliehen. Es betrachtet einen einzelnen Fußgänger wie ein Molekül in der Luft, das sich durch eine Strömung bewegt. Wie Gase können Fußgänger nicht beliebig zusammengepresst werden. Dieses Modell funktioniert aber bei weniger dicht gedrängten Szenarien nicht mehr. Ein anderes Modell sind zelluläre Automaten. Dabei wird die zur Verfügung stehende Fläche schachbrettartig eingeteilt, wobei jedes Feld entweder von einer Person belegt oder frei sein kann. Anhand des aktuellen Zustands wird dann die Belegung im folgenden Schritt berechnet. [SPON1]

Außerdem spielen bei den Wegen von Fußgängern noch andere Faktoren mit. So gibt es beispielsweise einen kulturell geprägten Mindestabstand zu anderen Personen, der in Asien meist höher ist als in Europa, in Südamerika dagegen geringer. Soziale Konventionen und unterbewusste Entscheidungen haben ebenso einen Einfluss auf die Wegewahl. Fußgänger passen ihre Geschwindigkeit der Masse an um einen Aufprall zu vermeiden, in einem Gedränge können sich Bahnen bilden, die sich in die selbe Richtung bewegen und in Paniksituationen können sich Menschen komplett irrational verhalten. [SPON1]

Ein viel zitiertes Extremfall ist das Unglück auf der “Love Parade” am 24.07.2010 bei dem 21 Personen im Gedränge der panischen Menschenmassen zu Tode kamen. Dieses Ereignis hat sicherlich das Bewusstsein um die Notwendigkeit akurater Simulationen gestärkt, um so Engstellen und potentielle Gefahrenpunkte schon im Vorfeld zu erkennen und durch Sicherheitskräfte oder bauliche Maßnahmen zu entschärfen. [SPON2]

In dieser Arbeit wird die Erstellung einer dreidimensionalen webbasierten Visualisierung von Fußgängersimulationsdaten beschrieben. Die Ergebnisse der Simulationen sind als Webapplikation überall abrufbar und können einfach an Veranstalter, Sicherheits- und Rettungskräfte weitergegeben werden. So können diese sich bereits im Vorfeld ein Bild von der Situation vor Ort machen. Dabei hilft auch die dreidimensionale Darstellung, mit einer freien Navigation durch die Szene und realistischer Abbildung der Gegebenheiten vor Ort. So lässt sich erkennen ob Hindernisse ggf. entfernt werden könnten, die Höhe von Hindernissen einschätzen und eine schnellere Orientierung vor Ort gewinnen.

## 5.1 SumoViz

Mustafa K. Isik entwickelte unter dem Titel “SumoViz – HTML5-based Visualization of Pedestrian Simulation Data” [Isi12] eine Webanwendung zur Visualisierung von Simulationsergebnissen von Fußgängerströmen. Die Simulationsergebnisse liegen in teilweise sehr großen Textdateien vor, woraus sich der Name der Applikation ableitet. Die Anwendung kümmert sich um die Bereitstellung und zweidimensionale Darstellung der Simulationsergebnisse. Die Darstellung verwendet ausschließlich HTML5 JavaScript und die canvas-API und ist damit ohne die Installation von Plug-Ins in allen modernen Browsern abrufbar.

Die Anwendung stellt eine Draufsicht der Szenerie dar. Gebäude, Wände und andere Hindernisse werden mit schwarzen Linien symbolisiert, die Fußgänger werden als einzelne Punkt eingezeichnet (vgl. Abbildung 5.1). Ein Fortschrittsbalken zeigt den Zeitpunkt der Simulation an, der über Buttons veränderbar ist. Zusätzlich lässt sich ein *spawn graph* einblenden, der in einem Diagramm die Anzahl der Fußgänger pro Zeitpunkt anzeigt. Für das Importieren der Textdateien mit den Simulationsergebnissen ist ein Uploader integriert.

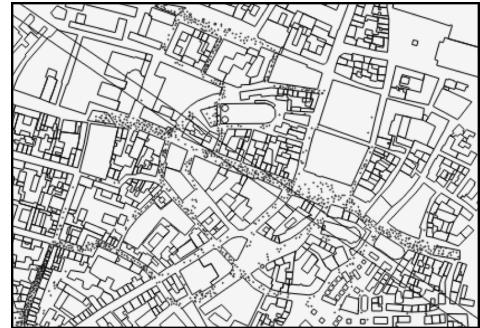


Abbildung 5.1: SumoViz [Isi12]

Die SumoViz-Anwendung läuft als Client-Server-Applikation (ähnlich wie in Abbildung 5.2), bei der die Daten serverseitig gespeichert und aufbereitet werden. Die Berechnung der Darstellung erfolgt mittels der 2D-API des canvas-Elements im Browser. Die Simulationsergebnisse werden nach dem Importieren serverseitig geparsst und dann in einer CouchDB-Datenbank abgelegt werden. Die Daten werden durch entsprechende Funktionen in der Datenbank gruppiert und über einen node.js-Webserver an den Client ausgeliefert.

## 5.2 SumoViz3D

In dieser Arbeit ist die Entwicklung der Software *SumoViz3D* als Weiterentwicklung von Mustafa K. Isiks SumoViz beschrieben. SumoViz3D soll den Ansatz von SumoViz aufgreifen, aber die browserbasierte Darstellung um eine dreidimensionale Ansicht der Fußgängersimulationsdaten erweitern. Zusätzlich sollen Werte wie die Geschwindigkeit oder Dichte der Fußgänger ablesbar sein. Die Software soll kleinere Anpassungen am Aussehen der Szene ermöglichen und diese dabei möglichst realistisch darstellen.

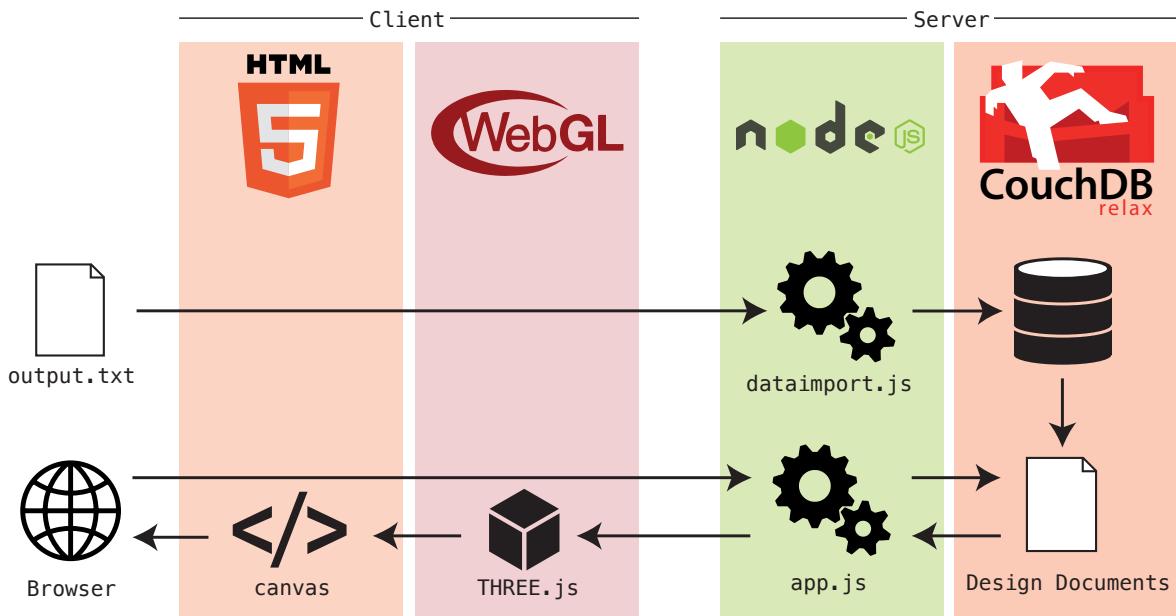


Abbildung 5.2: Struktureller Aufbau von SumoViz3D

Für SumoViz3D wird die bestehende serverseitige Architektur den Anforderungen entsprechend erweitert und die Client-Seite von Grund auf neu entwickelt. Die Veränderungen, die serverseitig durchgeführt werden müssen, passen lediglich die Ausgabe der Daten an, um die für die 3D-Darstellung benötigte Werte wie Höhe, Typ des Hindernis oder Dichte der Fußgänger auszuliefern. Der Dateiimport `dataimport.js` wird um eine Funktionalität für das Importieren von Gruppendaten erweitert.

In den folgenden Kapiteln werden die serverseitigen Veränderungen an der bestehenden SumoViz-Anwendung sowie die Neuentwicklung der Client-Seite beschrieben.

## Kapitel 6

# Serverkomponenten: CouchDB und node.js

Apache CouchDB ist ein Datenbanksystem der *Apache Software Foundation* das einen dokumentenorientierten Ansatz einsetzt. Im Gegensatz zu relationalen Datenbanken wird bei dokumentenorientierten Datenbanken kein festes Schema vorgegeben in dem die Datensätze abgelegt werden. Stattdessen wird jeder Datensatz als Dokument mit beliebiger eigener Struktur in der Datenbank abgelegt. Jedes Dokument ist als JSON-Objekt repräsentiert und hat mindestens ein Feld `_id` zur Identifikation. Der Zugriff auf die Daten erfolgt per HTTP-REST-Schnittstelle. Das bedeutet der Aufruf einer bestimmten URL liefert per HTTP die angeforderten Datensätze zurück. Über das HTTP-POST-Kommando können Daten in die Datenbank eingefügt werden. [ALS10]

Abfragen von einzelnen oder mehreren Datensätzen werden *Views* genannt. Views können in so genannten *Design documents* gespeichert werden und sind dann für die Abfrage per HTTP-Request verfügbar. CouchDB speichert die einzelnen Views zwischen, damit diese schneller verfügbar sind. Zur Realisierung von Filtern und gezielten Abfragen verwendet CouchDB das *MapReduce*-Verfahren, bei dem auf den Datenbestand nacheinander erst ein *map*- und dann ein *reduce*-Schritt ausgeführt wird. Diese Schritte werden durch JavaScript-Funktionen beschrieben und können komplexe Aufgaben auf den Daten ausführen. Solange der Datenbestand sich nicht verändert hat, muss das Ergebnis des MapReduce nicht neu berechnet werden. Der map-Schritt wird für jedes Dokument der Datenbank ausgeführt und transformiert es in eine neue Struktur. Als Rückgabe wird pro Datensatz kein, ein, oder mehrere Key-Value-Paare erwartet, die mittels `emit(key, value)` ausgegeben werden. Zusätzlich können hier mittels if-Abfragen Filter implementiert werden.

Der reduce-Schritt ist optional und wird ausgeführt, wenn das Zwischenergebnis des map-Schritts gruppiert werden soll. Als Parameter wird ein Array mit den Keys und ein Array mit den Values des map-Schritts übergeben. Auf diesen Arrays kann dann eine eigene Gruppierungsfunktion definiert werden. Bei großen Datensätzen wird die reduce-Funktion für einzelne Abschnitte des Zwischenergebnisses aufgerufen und anschließend nochmal für die bereits reduzierten Teilergebnisse. Deshalb wird noch der Parameter `rereduce` übergeben, der mitteilt, ob es sich um einen bereits reduzierten Datensatz handelt. Diese müssen dann meist nicht mehr reduziert sondern lediglich zusammengefügt werden. Hintergrund dieses Verfahrens ist es auch extrem große Datenmengen (in kleinen Schritten) verarbeiten zu können. [Hol11]

Die zweite serverseitige Komponente ist *node.js*, das aus Googles V8 JavaScript-Engine und einigen weiteren Komponenten besteht und damit die serverseitige Ausführung von JavaScript-Code ermöglicht. Node.js ist sehr leistungsfähig, da der JavaScript-Code vor der Ausführung in Maschinensprache übersetzt wird. Die Einbindung zusätzlicher Komponenten ist bei node.js über Module möglich, die über den Paketmanager *npm* (*Node Packaged Modules*) installiert werden können. [McL11] Auf Serverseite von SumoViz wird node.js für das Importieren (`dataimport.js`) und Ausliefern (`app.js`) der Daten eingesetzt.

## 6.1 Importieren der Simulationsdaten

Die Ergebnisse der Simulation werden in eine Textdatei geschrieben, die zur Verwendung mit SumoViz eingelesen und in der Datenbank abgelegt wird. Die Datensätze sind zeilenweise in der Ausgangsdatei abgelegt und in Blöcke unterteilt. Blöcke innerhalb der Datei werden mit **Begin** und **End** sowie dem Blocknamen gekennzeichnet. Beispielsweise sind die Daten der Baugeometrie im Block **Begin geometry** angelegt. Die Ausgangsdatei enthält immer einen Block mit Parametern zur Simulation, die in SumoViz nicht verwendet werden, da die Parameter hauptsächlich für die Erstellung der Simulation, nicht aber für die Visualisierung nötig sind.

Für das Importieren der Daten in die *CouchDB*-Datenbank exsistiert das node.js-Skript `dataimport.js`. Es verarbeitet die Ausgangsdatei block- und zeilenweise und legt die Daten dann in der Datenbank ab. Das Skript verarbeitet nur die Blöcke **Geometry**, **Data** und **Group Data**, alle anderen Inhalte der Ausgangsdatei werden ignoriert. Für die Verbindung zur CouchDB wird das Modul *cradle* verwendet. Zunächst legt das Skript ein JSON-Array mit allen einzufügenden Datensätzen an, dieses wird dann mittels *bulk insertion* in die Datenbank geschrieben.

### 6.1.1 Import der Fußgängerdaten

Durch Zeilen mit den Schlüsselwörtern **Begin Data** und **End Data** ist innerhalb der Ausgabedatei ein Abschnitt mit den Fußgängerpositionsdaten ausgezeichnet. Die Datensätze enthalten einen Zeitstempel, eine Identifikationsnummer des Fußgängers, Positionsdaten und Angaben zur Ebene auf der sich der Fußgänger befindet sowie zur Dichte. Pro Zeitpunkt und sichtbaren Fußgänger wird eine Zeile in der Ausgabedatei generiert. Die Werte des Datensatzes sind mit Leerzeichen getrennt und haben folgenden Aufbau:

	<b>timecode</b>	<b>pedid</b>	<b>x</b>	<b>y</b>	<b>z</b>	<b>level</b>	<b>density</b>
<b>Datentyp</b>	float	int	float	float	float	int	float
<b>Beispiel</b>	14.02	32	30.06	36.04	0.00	0	0.17

Tabelle 6.1: Struktur der Fußgängerdaten

### 6.1.2 Import der Geometrie

Der Block **Geometry** enthält eine Gliederung in unterschiedliche Stockwerke, gekennzeichnet mit **begin floor**, gefolgt von einer Höhenangabe zum jeweiligen Stockwerk. SumoViz und SumoViz3D unterstützen die Darstellung von Stockwerken nicht und ignorieren diese Daten deshalb. Dann folgen zeilenweise die einzelnen geometrischen Objekte. Beim Import werden nur Datensätze im Format **Polygon** importiert. Es gibt zum Beispiel noch den Typ **Image** mit dem Grafiken über die Szene gelegt werden können. Dieser wird aber nicht unterstützt und daher ebenfalls ignoriert.

Ein Polygon besteht aus einer beliebigen Anzahl Koordinatenpaare  $(x_i, y_i)$ , die jeweils einen Eckpunkt des Polygons darstellen. Daher sollte die Mindestzahl an Eckpunkten 3 sein. Eine Höhenangabe ist optional, wird aber von einigen Geometrietypen beim rendern erwartet (vgl. Kapitel 7.2). Die Struktur der Geometriedaten ist in Tabelle 6.2 dargestellt.

	<b>format</b>	<b>x<sub>0</sub></b>	<b>y<sub>0</sub></b>	<b>...</b>	<b>name</b>	<b>type</b>	<b>height<sup>1</sup></b>
<b>Datentyp</b>	String	float	float	...	String	String	float
<b>Beispiel</b>	Polygon	40.39	42.69	...	o2	obstacle	3.5

Tabelle 6.2: Struktur der Geometriedaten

### 6.1.3 Import der Gruppendaten

Für die Visualisierung der Gruppen in SumoViz3D wurde der Import für Gruppendaten im **dataimport.js**-Skript hinzugefügt. Gruppendaten sind nicht in allen Ausgabedateien angelegt, falls sie vorhanden sind ist der Block entsprechend mit **Group Data** gekennzeichnet und in der Visualisierung steht die Einfärbung nach Gruppen zur Verfügung (siehe Kapitel 7.5.3). Ein Datensatz beinhaltet die Zugehörigkeit eines Fußgängers zu einer Gruppe. Einzelne Gruppen werden dabei implizit durch die Verwendung der jeweiligen Gruppen-ID erzeugt. Zusätzlich ist noch die Gruppengröße mit angegeben (vgl. Tabelle 6.3). Dieser Wert ist aber redundant, da die Gruppengröße auch mittels Summenfunktion berechnet werden kann und wird deshalb nicht mit importiert.

	<b>pedid</b>	<b>groupid</b>	<b>size</b>
<b>Datentyp</b>	int	int	int
<b>Beispiel</b>	241	32	3

Tabelle 6.3: Struktur der Gruppendaten

## 6.2 Ausgabe der Simulationsdaten

Alle Daten der Visualisierung muss der Client vom Server nachladen. Dabei handelt es sich um den Namen der Animation, Positionsdaten der Fußgänger, die bauliche Geometrie und die Gruppendaten. Jede dieser Abfragen hat eine eigene URL als Endpunkt. Node.js agiert als Webserver, verarbeitet die Anfragen und liefert entsprechend die Daten aus CouchDB aus.

<sup>1</sup>optional

Die Aufbereitung der Daten findet ausschließlich in der Datenbank mittels der MapReduce-Funktionen statt. Nach dem Laden der Daten im JSON-Format werden diese im Client ge-parst und stehen dann als JavaScript-Objekt zur Verfügung.

### 6.2.1 Ausgabe der Fußgängerdaten

Für SumoViz wurden lediglich die x-y-Koordinaten der Fußgänger nach Zeitpunkt gruppiert und als JSON-Array ausgegeben. Die Visualisierung in SumoViz3D benötigt aber noch weitere Daten, weshalb das Datenformat umstrukturiert und erweitert werden musste. Die Fußgänger-ID (`pedid`) ist notwendig um die Gruppenzuordnung zu erhalten und die Positionsdaten eines Fußgängers zu früheren oder späteren Zeitpunkten zu bestimmen. Das ist für die Bestimmung der Geschwindigkeit und Bewegungsrichtung notwendig und kann bei zukünftigen Erweiterungen wie dem Tracing einzelner Fußgänger nützlich sein (vgl. Kapitel 8.2). Zusätzlich wird noch die Dichte (`density`) pro Fußgänger pro Zeitpunkt ausgeliefert. Diese Daten waren auch bisher schon gespeichert und wurden nur noch nicht mitausgeliefert. Das Format der Dokumente in CouchDB bleibt unverändert und bestehende Datensätze müssen nicht neu importiert werden. Lediglich die MapReduce-Funktionen der Design Documents wurden angepasst.

Die map-Funktion für die Fußgängerdaten (siehe Listing 6.1) von SumoViz wurde nur geringfügig erweitert, so dass im Value-Array nun neben der Position auch die ID und die Dichte ausgegeben werden.

```

1  function(doc) {
2      if (doc.pedid !== undefined) {
3          emit(doc.time, [doc.x, doc.y, doc.density, doc.pedid]);
4      }
}
```

Listing 6.1: CouchDB map-Funktion für Fußgängerdaten

Die ausgegebenen Key-Value-Paare werden mit einer komplett neu geschriebenen reduce-Funktion gruppiert (siehe Listing 6.2). Die Werte werden nicht mehr in einer Array gespeichert sondern in einem Objekt pro Zeitpunkt (Listing 6.2, Zeile 12). In diesem Objekt wird für jeden Fußgänger ein Key mit entsprechender Fußgänger-ID (`values[i][3]`) und einem Array aus Positions- (`values[i][0]` und `values[i][1]`) und Dichtedaten (`values[i][2]`) als Value angelegt (Listing 6.2, Zeile 13). Das ermöglicht den direkten Zugriff auf Daten einer bestimmten Fußgänger-ID mit dem Aufruf `pedestrianData[zeitpunkt][id]` ohne die Anwendung einer linearen Suche durch alle Werte des Zeitpunktes nach der gesuchten ID.

Wird die Bearbeitung der Daten auf Grund der Größe aufgeteilt, so bekommt die reduce-Funktion im rereduce-Fall (Listing 6.2, Zeile 2) ein Array aus Objekten, die je einen Teil der schon gruppierten Fußgängerdaten des jeweiligen Zeitpunkts enthalten. Dieses Array mit Objekten muss zu einem einzelnen Objekt (`values[0]`) zusammengefügt werden und ergibt dann das fertige Datenobjekt für den entsprechenden Zeitpunkt. Eine Kollision der Keys beim Zusammenfügen der Objekte ist ausgeschlossen, da die Keys den IDs der Fußgänger entsprechen, die zu jedem Zeitpunkt einmalig sind.

```

1 function(keys , values , rereduce) {
2     if(rereduce) {
3         for (i=1;i<values.length;i++) {
4             for (var key in values[i]) {
5                 values[0][key] = values[i][key];
6             }
7         }
8         return values[0];
9     } else {
10        var sortedValues = {};
11        for(var i = 0; i < values.length; i++) {
12            sortedValues[values[i][3]] =
13                [values[i][0],values[i][1],values[i][2]];
14        }
15        return sortedValues;
16    }
17 }

```

Listing 6.2: CouchDB reduce-Funktion für Fußgängerdaten

Ein beispielhaftes Datenobjekt der Fußgängerdaten für einen bestimmten Zeitpunkt der Simulation kann wie folgt aussehen:

```
{"0": [40.23,40.05,0],"1": [40.69,37.64,0],"2": [40.92,36.44,0],...}
```

### 6.2.2 Ausgabe der Geometrie- und Gruppendaten

Die Ausgabe der Geometriedaten benötigt keine Gruppierung und dementsprechend nur eine map-Funktion, die pro Geometrie-Objekt ein JSON-Objekt ausgibt, das die x-y-Koordinaten in Form eines Arrays und Typ, Name und Höhe als Properties darstellt. Ein entsprechendes JSON-Objekt kann zum Beispiel so aussehen:

```
{geometry: [[64.42, 33.3], [61.2, 35.7], ...],
 "height": 3.5, "name": "plant0", "type": "obstacle"}
```

Das Ursprungsformat der Gruppendaten wird nach dem Key, der Gruppen-ID, gruppiert. Pro Gruppen-ID wird dann ein Array mit den in der Gruppe enthaltenen Fußgänger-IDs ausgegeben. Die reduce-Funktion muss lediglich das schon gruppierte values-Array ausgeben, bzw. im rereduce-Fall werden die einzelnen value-Arrays zu einem einzigen Array (`values[0]`) verbunden (vgl. Listing 6.3 Zeile 4). Die Gruppen-ID wird für die Darstellung im Client nicht mehr benötigt.

```

1 function (keys,values,rereduce) {
2     if (!rereduce) return values;
3     else {
4         for (i=1;i<values.length;i++) values[0].concat(values[i]);
5         return values[0];
6     }
7 }

```

Listing 6.3: CouchDB reduce-Funktion für Gruppendaten

## Kapitel 7

# Clientseitige Entwicklung von SumoViz3D

Clientseitig verwendet SumoViz3D die THREE.js-Bibliothek um die Simulationsdaten in WebGL darzustellen. Mit der JavaScript-Bibliothek *jQuery UI*, HTML und CSS wird die Bedienoberfläche erstellt.

### 7.1 Das Interface von SumoViz3D

Das canvas-Element zur Darstellung der Simulation wird in der vollen Browsergröße dargestellt, alle anderen Elemente werden als Overlay aus HTML-Elementen gestaltet. Dabei werden viele Bestandteile mittels *jQuery UI* umgesetzt. Dieses JavaScript-Framework macht es möglich Interface-Elemente wie Buttons und Dialogboxen einfach und cross-browserkompatibel anzulegen. [jQu]

Die Toolbar (Abbildung 7.1 ①) ermöglicht die Steuerung des Ablaufs der Animation. Mit dem Play/Pause-Button kann die Animation abgespielt und angehalten werden. Der aktuelle Stand der Visualisierung kann mittels des Sliders verändert werden. Zusätzlich wird noch angezeigt, welcher Simulationsschritt aktuell dargestellt wird. Über die Pfeil-Buttons lässt sich schrittweise durch die Simulation navigieren. Über den Einstellungs-Button wird ein Dialog aufgerufen, über den globale Modifikationen an der Szene vorgenommen werden können (Abbildung 7.1 ②). Zusätzlich lassen sich durch einen Kontextklick auf die Geometrieobjekte die objektbezogenen Modifikationen aufrufen (Abbildung 7.1 ③).

Die THREE.js-Erweiterung *Stats* (Abbildung 7.1 ④) zeigt die aktuelle Framerate mit der die Szene gezeichnet wird. Dazu wird im Aufruf der Zeichenmethode (siehe Kapitel 7.3) zusätzlich noch ein Aufruf von `stats.update()` platziert, der dann die Zeit zwischen dem aktuellen Aufruf und dem letzten bestimmt und so die Framerate errechnet. [MrD] Durch einen Klick auf das Modul lässt sich auch die benötigte Ausführungszeit der Zeichenmethode in Millisekunden anzeigen. In Klammern sind Minimum und Maximum angegeben.

Die Legende (Abbildung 7.1 ⑤) passt sich den Ansichtsoptionen an und zeigt für die Einfärbung nach Geschwindigkeit und Dichte die entsprechende Farbkodierung der Fußgänger an. Wenn das Gitternetz sichtbar ist, wird der Abstand der Gitternetzlinien in Metern angezeigt.

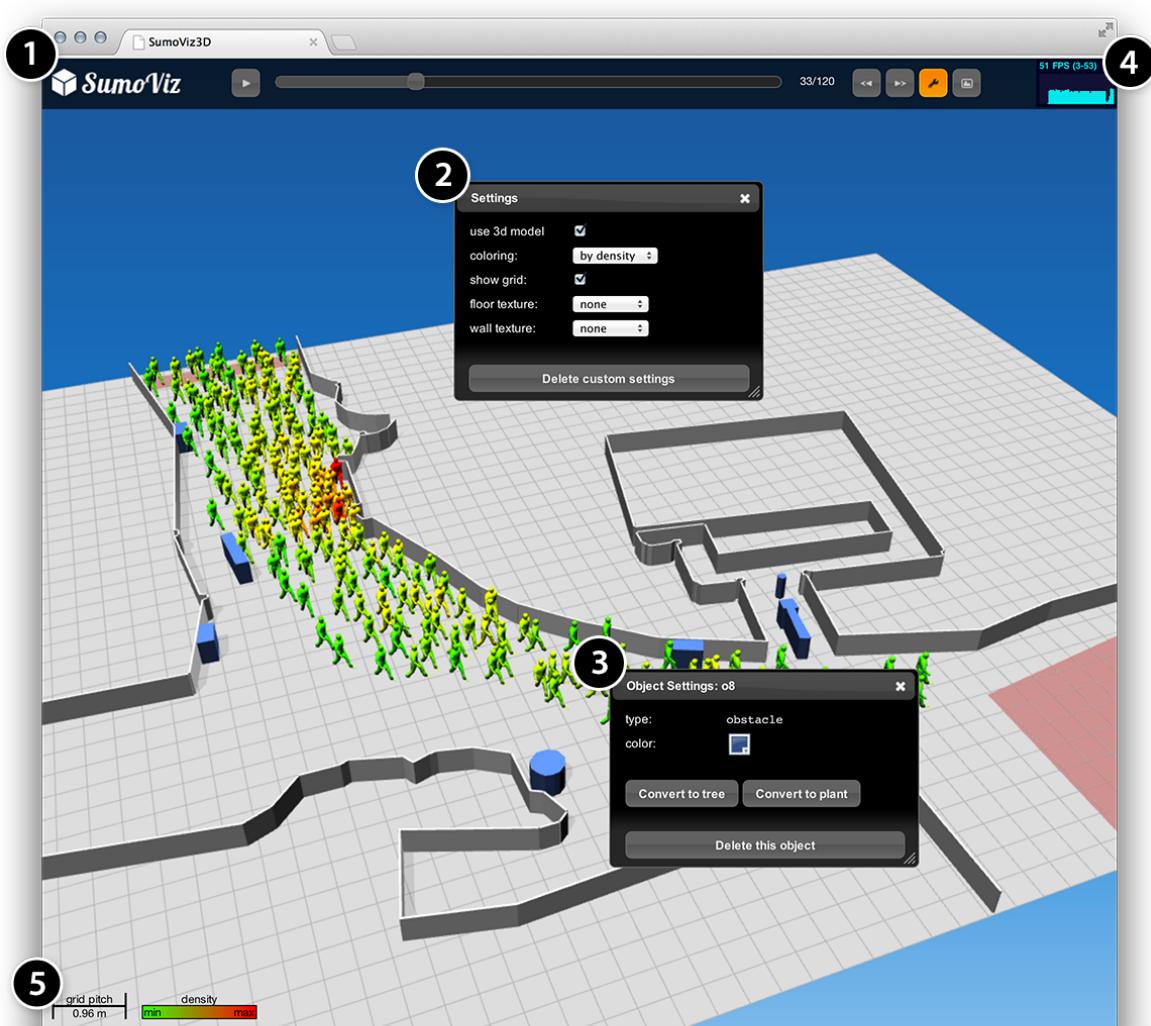


Abbildung 7.1: Screenshot des SumoViz3D-Interfaces

## 7.2 Darstellung der Geometrie

Die geometrischen Objekte zur Darstellung der Umgebung enthalten in ihrer Datenstruktur die Beschreibung ihres Grundrisses in Form von x-y-Koordinaten, eine Höhenangabe, den Namen und ein Typen-Feld. Für die Darstellung maßgeblich entscheidend ist der Inhalt des Typen-Felds. Es entscheidet, welche Renderingroutine verwendet wird. Folgende Typen werden von SumoViz3D unterschieden: `geometry`, `obstacle`, `wall`. Zudem gibt es noch eine vierte Renderingroutine für alle anderen Typen unter die vor allem `source`, `target` und `field` fallen.

Die Berechnung der Geometrie erfolgt beim Laden der Simulation und muss nur einmalig ausgeführt werden. Dazu wird die Methode `drawGeometry` definiert. Wird die Geometrie während der Laufzeit verändert, wird diese Methode erneut aufgerufen (siehe Kapitel 7.2.6).

### 7.2.1 Skalierung der Geometrie

Die darzustellende Geometrie kann extrem unterschiedliche Dimensionen haben. Trotzdem soll nach dem Laden der Szene die Kamera so positioniert sein, dass ein Überblick über die gesamte Szene möglich ist.

Erster Ansatz zur Lösung war es, die Kameradistanz entsprechend der Größe der Szene mitzuskalieren. Jedoch trat hierbei *Z-Fighting* auf. Ein bekanntes Problem aus dem 3D-Bereich, bei dem durch Rundungsfehler Fehler in der Darstellung auftreten. Bei einer hohen Distanz zwischen Kamera und Objekten ist der relative Abstand der Objekte zueinander sehr gering. Durch mangelnde Genauigkeit in der Speicherung der Tiefenwerte zweier Objekte kann nicht mehr entschieden werden, welches Objekt vor dem jeweils anderen liegt. Objekte die eigentlich verdeckt sein müssten, flimmern an manchen Stellen durch die Oberfläche hindurch (vgl. Abbildung 7.2).

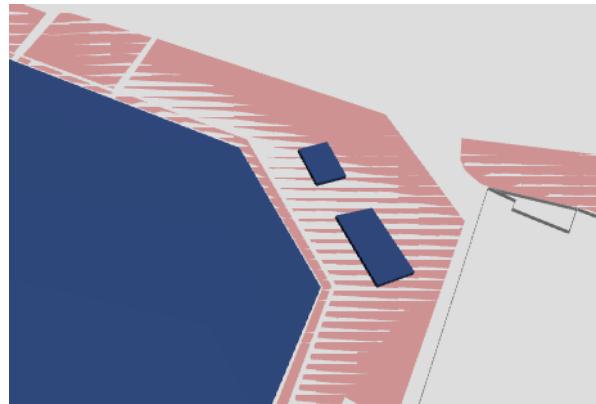


Abbildung 7.2: Z-Fighting zwischen Grundebene und Feldern bei großer Kameradistanz

Zur Lösung des Problems wird nun die Kameradistanz für jede Szene unverändert gelassen und die Geometrie- und Fußgängerobjekte entsprechend der Größe der Grundebene (`geometrySize`) skaliert. Dazu wird ein Skalierungsfaktor `globalScale` wie folgt berechnet:

```
globalScale = 50/((geometrySize.x+geometrySize.y)/2);
```

Der Faktor 50 wurde empirisch bestimmt und an die Einstellungen der Kamera angepasst. Jedes Objekt wird mit dem Skalierungsfaktor multipliziert und entsprechend vergrößert oder verkleinert. Einige weitere Vorteile ergeben sich aus diesem Vorgehen: Die Lichter müssen nicht an die Szenengröße angepasst werden um diese vollständig auszuleuchten, denn absolut gesehen hat jede Szene exakt die gleiche Größe. Selbiges gilt für die Steuerung der Kamera (vgl. Kapitel 7.4) die nicht auf die Szenengröße angepasst werden muss.

### 7.2.2 Darstellung der Grundebene und des Gitternetzes

In den Geometriedaten befindet sich immer ein Element vom Typ `geometry`, das eine rechteckige Grundfläche definiert die alle Objekte enthält. Diese Fläche wird orthogonal zur y-Achse des WebGL-Koordinatensystems eingezeichnet und beginnt im Ursprung. Auf dieser Grundebene wird ein Gitternetz eingezeichnet, das zwei Zwecke erfüllt: Einerseits helfen die Linien, die auf den Fluchtpunkt zulaufen bei der Tiefenwahrnehmung. Andererseits lassen sich mit dem in der Legende angegebenen “grid pitch” die Größenverhältnisse besser beurteilen. Dieser Wert gibt den Abstand zweier benachbarter Gitternetzlinien in Metern an. Eine Maßstabangabe wie in Landkarten ist auf Grund der perspektivischen Abbildung nicht möglich.

### 7.2.3 Darstellung von Objekten des Typ “obstacle”

Geometrie vom Typ `obstacle` sind bauliche Hindernisse wie Gebäude oder auch Bäume und Pflanzen. Diese müssen von den Fußgängern umgangen werden. Für die Darstellung dieser Hindernisse wird ein Polygon aus den x-y-Koordinaten erzeugt. Mit einer `ExtrudeGeometry` wird aus dem flachen Polygon dann ein dreidimensionales Objekt erzeugt (vgl. Abbildung 7.3). Hierfür wird die beim Objekt gespeicherte Höhenangabe benötigt.

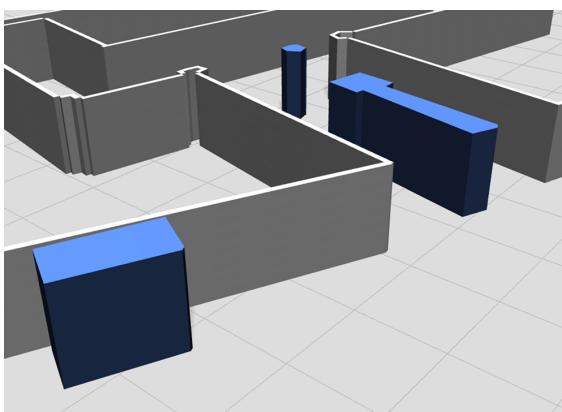


Abbildung 7.3: Darstellung von normalen Objekten (blau)

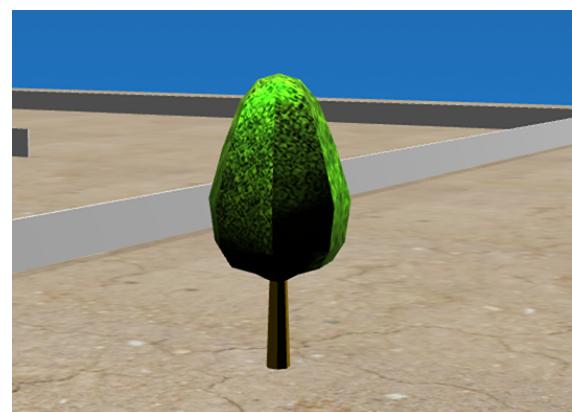


Abbildung 7.4: Darstellung von Bäumen mittels COLLADA-Modell

Bäume und Pflanzen werden gesondert dargestellt. Die Erkennung davon erfolgt über den Namen des Objekts, da diese ebenfalls den Typ “obstacle” haben. Ein entsprechendes Objekt kann beispielsweise `tree0` benannt sein. Enthält also ein Objektname `tree` oder `plant` wird die Darstellung angepasst. Dazu sind in SumoViz3D externe Modelle eines Baums und eines Buschs hinterlegt, wie in Abbildung 7.4 zu sehen. THREE.js bietet verschiedene Erweiterungen zum importieren von externen 3D-Modellen in unterschiedlichen Formaten. SumoViz3D verwendet den `ColladaLoader`, der Modelle im *COLLADA*-Format importieren kann. *COLLADA (COLLABorative Design Activity)* ist ein offenes XML-Format für die Speicherung von Modellen, Texturen und ganzen Szenen. Das Format wird wie WebGL auch von der Khronos Group verwaltet und kann in vielen 3D-Programmen importiert werden. [COL]

Das Baum- bzw. Busch-Modell wird mittels des Loaders aus der Datei geladen und geparsed. Danach steht es als normales 3D-Objekt zur Verfügung und kann skaliert und positioniert werden. Um die Ladezeit gering zu halten, wurden die Modelle mit sehr wenigen Vertices

angelegt und mit einfachen Texturen versehen. Die Dateigröße des Baum-Modells mit Grafiktextur beträgt etwa 50 Kilobytes. Komplexere Modelle könnten mehrere Megabytes groß sein und das Nachladen über die Internetverbindung würde zu lange dauern.

### 7.2.4 Darstellung von Objekten des Typ “wall”

Wände sind als Pfad definiert. Die einzelnen Koordinaten werden linear verbunden und ergeben so die Grundlinie der Wand. Um in der 3D-Darstellung realistischer zu wirken sollen die Wände auch noch eine Dicke erhalten. Dazu wird aus jeweils zwei benachbarten Pfadpunkten und den selben Punkten, die aber in der Höhe verschoben wurden, ein Rechteck gebildet. Die Höhe der Wand stammt dabei auch aus den importierten Ursprungsdaten. Dieses Rechteck wird wieder mit `ExtrudeGeometry` um die Dicke der Wand zu einem Quader ausgedehnt. Da eine Wand mehr als zwei Pfadpunkte haben kann, werden die entstandenen Quader dann wieder zu einem Objekt zusammen gefügt und angezeigt (vgl. Abbildung 7.5).

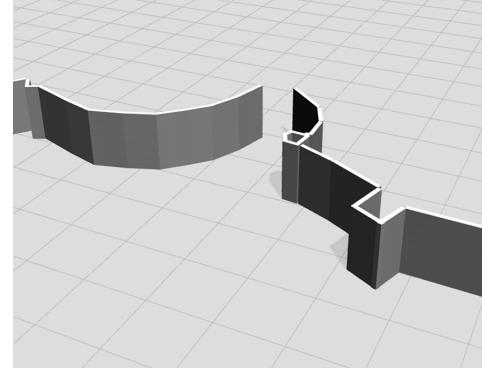


Abbildung 7.5: Darstellung von Wänden ohne Textur

### 7.2.5 Darstellung anderer Objekte

Hat die Geometrie einen anderen Typ, so wird sie als flaches, halbtransparentes Polygon auf der Grundebene dargestellt (vgl. Abbildung 7.6). Alle Szenen haben mindestens ein Objekt vom Typ `source` das der Startpunkt der Fußgänger ist. Entsprechend existiert `target`, welches das Ziel der Fußgänger darstellt. Pro Szene können auch mehrere Start- und Zielfelder vorhanden sein. Der Typ `field` wird ebenso dargestellt und markiert Bereiche, die beispielsweise besondere Auswirkungen wie Verlangsamung oder Beschleunigung auf die Fußgänger haben. Aber auch völlig unbekannte Geometrietypen können in diesem Fallback zumindest grundlegend angezeigt werden.

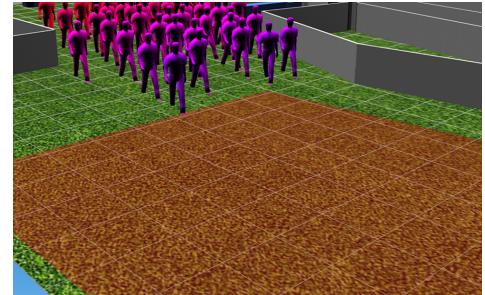


Abbildung 7.6: Darstellung von Quelle, Ziel und anderen Feldern (rot)

### 7.2.6 Veränderung während der Laufzeit

Eigentlich ist die Geometrie während der gesamten Zeit unverändert und wird deshalb nur initial berechnet. SumoViz3D ermöglicht dem Betrachter zusätzlich das Erscheinungsbild der Szene anzupassen und so ein realistischere Darstellung zu erzeugen. Folgende Modifikationen an der Geometrie sind dem Benutzer während der Laufzeit möglich:

**Globale Modifikationen:**

- Darstellung der Fußgänger als Partikel oder 3D-Modelle
- Das Gitternetz der Grundebene lässt sich ein- und ausblenden.
- Die Textur der Grundebene kann verändert werden. Es steht eine Auswahl verschiedener Grafiktexturen, wie beispielsweise Wiese oder Erde, zur Verfügung.
- Es kann eine Grafiktextur für alle Wände ausgewählt werden. Hier steht ebenfalls eine Vorauswahl zur Verfügung.

**Objektbezogene Modifikationen:**

- Einzelne Objekte der Szene lassen sich entfernen.
- Alle Objekte, außer Bäume und Büsche, lassen sich individuell einfärben.
- Objekte vom Typ `obstacle` lassen sich zu Bäumen oder Büschen konvertieren. Dazu wird vor den bestehenden Objektnamen der Prefix `tree_` oder `plant_` gesetzt.

Globale Modifikationen können über das Einstellungsmenü in der Toolbar vorgenommen werden. Für objektbezogene Modifikationen muss ein entsprechendes Objekt mit einem Kontextklick ausgewählt werden. Um zu erkennen welches Objekt durch den Kontextklick ausgewählt wurde ist eine Berechnung nötig, welche die 2D-Koordinaten des JavaScript-Events `contextmenu` auf die 3D-Szene umgerechnet. Die x- und y-Koordinaten des Klicks werden in das WebGL-Koordinatensystem (vgl. Kapitel 4.1) konvertiert und für die z-Achse wird ein fester Wert verwendet. Dann wird ein Strahl ausgehend von der Kameraposition in Richtung des Klickpunkts berechnet und die durchquerten Objekte ermittelt. Das erste dieser Objekte ist das vom Benutzer ausgewählte, da die später durchkreuzten Objekte verdeckt und damit für den Nutzer nicht sichtbar sind.

**Speichern der Einstellungen**

Die Modifikationen, die durch den Benutzer vorgenommen wurden, sollen persistent gespeichert werden. Zu diesem Zweck wird die *WebStorage-API*, die mit HTML5 spezifiziert wurde, verwendet. Es handelt sich dabei um lokalen Speicher in Form von Key-/Value-Paaren. Einschränkend ist hierbei, dass nur Strings gespeichert werden können. Um andere Datentypen zu speichern, können diese beispielsweise im JSON-Format hinterlegt werden. Für die Größe des Speichers schlägt das W3C 5 Megabytes vor. [W3s] Zugriff auf den Speicher erfolgt über das `localStorage`-Objekt mittels der Methoden `setValue(key, value)` und `getValue(key)` oder alternativ über die Punktschreibweise `localStorage.key` oder assoziative Arrays `localStorage["key"]`. Änderungen im Speicher müssen nicht mehr explizit gespeichert werden und sind über die Session hinaus beliebig lange verfügbar. Mit `localStorage.clear()` lässt sich der Inhalt des Speichers löschen. [Pil10]

Alle Veränderungen an den in Kapitel 7.2.6 beschriebenen Einstellungen werden pro Simulations-Datensatz im lokalen Speicher abgelegt. Die vorgenommenen Änderungen sind sofort, ohne

explizites speichern, persistent und lokal gesichert und damit nicht für andere Nutzer sichtbar. Nach dem Laden der Szene wird geprüft ob für die jeweiligen Simulationsdaten bereits Einstellungen hinterlegt wurden und diese gegebenenfalls geladen und angewendet.

Zum Ablegen der globalen Modifikationen wird das Schlüsselformat `animationName_setting` verwendet. Zudem werden unter dem Schlüssel `animationName_objectSettings` die objektbezogenen Anpassungen in einem Array gespeichert. Die Reihenfolge der Geometrieobjekte, wie sie aus der Datenbank ausgegeben werden, wird zugleich zur Identifikation der Objekte verwendet. An der jeweiligen Array-Position werden die Anpassungen für das entsprechende Objekt gespeichert. Bleibt das Objekt unverändert, wird der Wert `null` eingetragen.

In Listing 7.1 wird ein Beispielhaftes JSON-Objekt zur Speicherung der Veränderungen an der Animation `out_aStern` abgebildet.

```

1 {
2     "out_aStern_showGrid": "no",
3     "out_aStern_floorTexture": "textures/gray.jpg",
4     "out_aStern_pedestrianColoring": "density",
5     "out_aStern_objectSettings": [
6         {"color": {"r": 1.00, "g": 0.00, "b": 0.00}},
7         null,
8         {"convert": "tree_obstacle0"}
9     ]
10 }
```

Listing 7.1: Beispielhaftes JSON-Objekt zur Speicherung der Änderungen

In diesem Beispiel wurde das Gitternetz ausgeblendet (Zeile 2) und auf der Grundebene eine Grafiktextur aus der Datei `textures/gray.jpg` geladen (Zeile 3). Die Einfärbung der Fußgänger erfolgt nach der Dichte (Zeile 4). Außerdem wurden zwei objektbezogene Modifikationen durchgeführt. Das Objekt mit der ID 0 wurde rot eingefärbt (Zeile 6) und das Objekt mit der ID 2 wurde in einen Baum konvertiert (Zeile 8).

## 7.3 Animationen in JavaScript

Um Animationen in JavaScript auszuführen wurden lange Timer-Funktionen verwendet. Meist wurde `setTimeout(function, delay)` oder `setInterval(function, interval)` genutzt, die in einem festgelegten Rhythmus eine Funktion aufrufen, die inkrementell die Animation ausführt. Während `setInterval` nur einmalig aufgerufen wird und dann von alleine die Funktion im angegebenen Intervall aufruft, muss `setTimeout` nach der Ausführung des Animationsschritts erneut rekursiv aufgerufen werden. [Res12] Jedoch gibt es einige Nachteile bei der Verwendung dieser Methoden: Die Wiederholrate wird fest vorgegeben, unabhängig von der Leistung des Rechners. Ist die Ausführung zum angegebenen Zeitpunkt nicht möglich, da gerade eine andere Berechnung ausgeführt wird, wird die Funktion in eine Warteschlange gestellt. In der Warteschlange wird nur ein Aufruf jeder Funktion gespeichert, alle weiteren werden ignoriert und nicht ausgeführt. Üblicherweise liegt die Wiederholrate eines Bildschirms bei 60 Hz. Daher lohnt es sich nicht die Animations-Schleife öfter auszuführen und es ergibt sich eine Intervallzeit von ca. 17 ms. [Fla06]

Ein grundlegendes Problem bei der beschriebenen Vorgehensweise ist, dass für den Browser nicht erkennbar ist, dass es sich bei der aufgerufenen Funktion um die Ausführung einer Animation handelt und deshalb keine spezifischen Optimierungen möglich sind. Mozilla entwickelte den Funktionsaufruf `requestAnimationFrame`. Da der Browser weiß, dass es sich dabei um die Ausführung von Animationen handelt, kann er beispielsweise parallel ablaufende Aufrufe in einem repaint-Zyklus zusammenfassen. Wie bei der `setTimeout`-Methode muss die Funktion nach dem Ablauen des Animationsschritts rekursiv aufgerufen werden, wodurch die Animation mit maximaler Wiederholrate abläuft. Der Browser kann die Ausführung komplett stoppen, wenn das Fenster im Hintergrund und nicht für den User sichtbar ist. Das spart CPU-Ressourcen und senkt so den Stromverbrauch. [Iri11]

Die Standardisierung vom W3C ist noch nicht final, weshalb einige Browser-Hersteller die Funktion mit einem so genannten Vendor-Prefix implementieren. Für den Einsatz in Firefox wird beispielsweise `mozRequestAnimationFrame` verwendet. Deshalb setzt SumoViz3D ein *Polyfill* ein, das beim Aufruf von `requestAnimationFrame` die im jeweiligen Browser verfügbare Methode verwendet. [Iri11] Für Browser ohne `requestAnimationFrame`-Unterstützung gibt ein Fallback mit der Verwendung von `setTimeout`. Dieses Fallback versucht die Intervallzeit zu optimieren, um eine maximale Framerate zu erreichen. Dazu wird die Ausführungszeit der Methode berechnet und die Intervallzeit entsprechend gestzt.

Die Wiederholrate der Animation ist abhängig von der Geschwindigkeit des Rechners. Der Ablauf der Visualisierung muss aber unabhängig von der Rechenleistung sein, damit auf schnellen Rechnern nicht alle Simulationsschritte innerhalb weniger Sekunden ablaufen. Deshalb wird das Aktualisieren der Fußgängerkoordinaten und deren Einfärbung von der Animationsschleife entkoppelt und mit einer `setInterval`-Methode regelmäßig (fünf mal pro Sekunde) ausgeführt. Innerhalb der Animationsschleife findet dann das Neuzeichnen des canvas-Elements und das Aktualisieren der Kameraposition (vgl. Kapitel 7.4) statt. Zusätzlich wird noch das THREE.js-Stats-Objekt zum Auslesen der Framerate aktualisiert (Listing 7.2, Zeile 8).

```
1 setInterval(updatePedestrians, 1000/5); //Fußgängerposition aktualisieren
2 animate(); //initialer Aufruf
3
4 function animate() {
5     requestAnimationFrame(animate); //rekursiver Aufruf
6     controls.update(); //Aktualisieren der Kameraposition
7     renderer.render( scene, camera ); //Neuzeichnen des canvas-Elements
8     stats.update(); //Messen der Geschwindigkeit
9 }
```

Listing 7.2: Animationen mittels requestAnimationFrame

## 7.4 Steuerung der Kamera

SumoViz3D verwendet eine perspektivische Kamera mit einem festeingestellten Sichtfeld von 45°. Die Positionierung und Ausrichtung der Kamera ist, mit einigen Einschränkungen, dem Nutzer überlassen. Während der Visualisierung können Kameraposition und -ausrichtung verändert werden. Dazu wurde die Klasse `SphereControls` angelegt. Im Konstruktor der Klasse wird das Kamera-Objekt übergeben, dessen Position manipuliert werden soll und das DOM-Element auf dem die Benutzereingaben erfolgen, in diesem Fall also das canvas-Element. Damit werden nur Eingaben die direkt auf dem canvas-Element erfolgen verarbeitet. Eingaben auf den HTML-Overlay-Elementen wie der Toolbar oder Dialogen werden für die Kamerasteuerung ignoriert. Event-Listener die auf Tastatur- und Mauseingaben reagieren werden mittels `addEventListener` an das canvas-Element angefügt. [Fla06]

Zur Ausrichtung der Kamera wird der Punkt `lookAt` definiert, auf den die Kamera stets ausgerichtet ist. Dieser Punkt liegt immer in der Grundebene der dargestellten Geometrie und kann dort bewegt werden. Die Kamera kann sich halbkugelförmig um diesen Punkt herum bewegen, aber dabei nicht unterhalb der Grundebene bewegt werden. Zur Berechnung der Position der Kamera sind neben der Position des `lookAt`-Punktes noch der Radius  $r$  der Kugel, der Neigungswinkel  $\theta$  und der Drehwinkel  $\phi$  notwendig. Mit folgenden Formeln lassen sich die Positionskoordinaten wie folgt berechnen: [Gub09]

$$\begin{aligned} x &= \text{lookAt}.x + r \cdot \sin \theta \cdot \cos \phi \\ y &= \text{lookAt}.z + r \cdot \cos \theta \\ z &= \text{lookAt}.y + r \cdot \sin \theta \cdot \sin \phi \quad (0 \leq \theta \leq \pi \wedge 0 \leq \phi < 2\pi) \end{aligned}$$

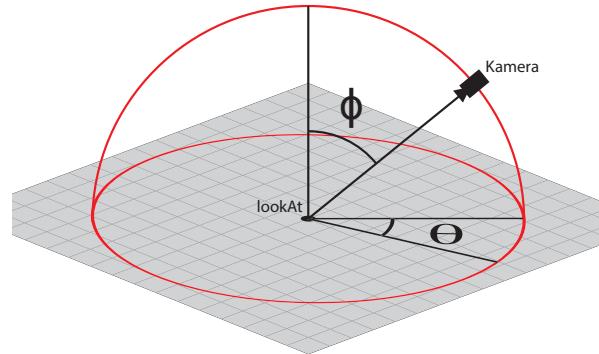


Abbildung 7.7: Kameraposition

In der Animations-Schleife wird die `update`-Methode der Steuerung aufgerufen. Ist ein `keydown`-Event aber noch kein `keyup`-Event auf einer der Cursor-Tasten ausgelöst, so ist diese Cursor-Taste gerade gedrückt und die Position des `lookAt`-Punkts wird in entsprechender Richtung verändert. Die Richtung der Bewegung muss aber relativ zu aktuellen Drehung der Kamera erfolgen. Dazu wird der Vektor zwischen Kameraposition und `lookAt`-Punkt berechnet. Von diesem Vektor wird die y-Komponente auf 0 gesetzt, damit liegt er auf der Grundebene. Dann wird der Vektor (bzw. ein orthogonaler Verktor) normiert und beim drücken der Cursor-Tasten auf den `lookAt`-Punkt addiert oder von diesem subtrahiert.

Folgende Event-Listener sind zur Steuerung der Visualisierung auf das canvas-Element gelegt. Solange der Event-Listener ausgelöst ist wird mit jedem Aufruf der `update`-Methode entsprechende Variable(n) modifiziert:

EventListener	modifizierte Variable	Funktionalität
keydown	lookAt	Bewegen
mousedown + mousemove	$\theta$ und $\phi$	Kamera drehen/neigen
mousewheel <sup>1</sup>	$r$	Zoomen
contextmenu	—	Objekte anpassen

Tabelle 7.1: Event-Listener und zur Steuerung modifizierte Variablen

Für das Drehen und Neigen der Kamera wird die Distanz zwischen dem Punkt, an dem das `mousedown`-Event ausgelöst wurde und der aktuellen Mauscursorposition berechnet. Die horizontale Distanz wird auf den Drehwinkel  $\phi$  addiert und bewegt die Kamera im mathematischen Drehsinn auf der Kugeloberfläche. Wird die Maus nach links gezogen, so hat die Distanz einen negativen Wert und wird vom Drehwinkel subtrahiert, was eine Bewegung gegen den mathematischen Drehsinn zur Folge hat. Die vertikale Distanz wird vom Neigungswinkel  $\theta$  subtrahiert, da dieser der y-Achse her aufgespannt wird (vgl. Abbildung 7.7). Außerdem wird verhindert das  $\theta > \frac{\pi}{2}$  wird, um stets über der Grundebene zu bleiben.

---

<sup>1</sup>In Firefox wird das Event DOMMouseScroll verwendet

## 7.5 Darstellung der Fußgängerdaten

Die Visualisierung der Fußgänger kann auf zwei Arten erfolgen. Entweder wird jeder Fußgänger als eigenes Modell dargestellt (vgl. Abbildung 7.8), oder es wird ein Partikelsystem verwendet (vgl. Abbildung 7.9), das für große Mengen an Fußgängerobjekten geeignet ist. Hier wird eine vereinfachte Darstellung als Kugel verwendet. Standardmäßig stellt SumoViz3D die Fußgänger immer als Partikel dar, jedoch kann über die Einstellungen die Darstellung als 3D-Modell aktiviert werden (siehe Kapitel 7.2.6). Je nach verfügbarer Rechenleistung sollte ab 2000 Fußgängern die Darstellung als Partikelsystem verwendet werden, um die Animation noch flüssig abspielen zu können.



Abbildung 7.8: Darstellung der Fußgänger mittels 3D-Modell

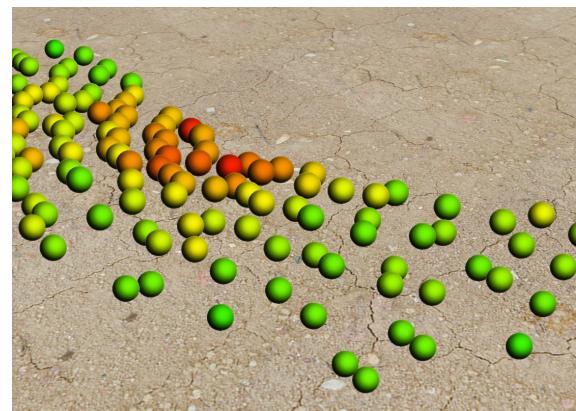


Abbildung 7.9: Darstellung der Fußgänger mittels Billboard-Grafik

### 7.5.1 Fußgänger als 3D-Objekte

Wie bei der Darstellung von Bäumen und Büschen gibt es für die Fußgängerfiguren ein Modell im COLLADA-Format, das geladen wird. Zu Beginn der Szene wird die benötigte Anzahl an Instanzen dieses Modells erzeugt, jedoch ist die Sichtbarkeit der Modelle noch deaktiviert. Somit ist es während des Ablaufs der Animation schneller möglich, die Modelle an der entsprechenden Position einzublenden. Da das Ändern der Sichtbarkeit wesentlich schneller geschieht als das Erzeugen neuer Objekte, kann so ein flüssiger Ablauf der Animation gewährleistet werden.

Damit das Fußgängermodell in Bewegungsrichtung blickt, muss das Modell an der y-Achse gedreht werden. Die Blickrichtung wird als Vektor zwischen der aktuellen Position und der position im folgenden Schritt beschrieben. Der Differenzwinkel zwischen der aktuellen Drehung und der Blickrichtung ist die anzuwendende Drehung. Falls kein folgender Simulationsschritt verfügbar ist, da es sich beispielsweise um den letzten Animationsschritt handelt, wird die Drehung aus dem vorherigen Schritt beibehalten. Das Fußgängermodell wird wie auch der Rest der Geometrie mit dem globalen Skalierungsfaktor skaliert. Im Maßstab gesehen hat das Modell eine reale Höhe von 1,75m.

Das Fußgängermodell besteht aus einer geringen Anzahl Vertices, um die Berechnungsdauer möglichst kurz zu halten. Trotz der geringen Auflösung sollte auf gewöhnlichen Rechnern ab 2000 Fußgängern die Darstellung als Partikelsystem verwendet werden (vgl. Kapitel 8.1).

### 7.5.2 Fußgänger als Partikelsystem

Für die Darstellung von Effekten oder vieler kleiner Elemente gibt es in der Computergrafik sogenannte Partikelsysteme. Haupteinsatzzweck ist die Berechnung von Feuer, Rauch, Nebel oder Wasser. Ein Partikelsystem besteht aus einer großen Menge einzelner Partikel, von denen jeder eigene Eigenschaften haben kann. Der Vorteil der Verwendung eines Partikelsystems gegenüber einzelnen Objekten ist die Minimierung der Objektanzahl. Ein Partikelsystem ist technisch nur ein einziges Objekt. Jeder Partikel wird als Vertex dieses Objekts im Vertex-Array an die Grafikkarte übergeben (vgl. Kapitel 4.1.1). Zur Darstellung der Partikel gibt es verschiedene Möglichkeiten. Die Partikel können als einfacher Punkt angezeigt werden, es kann eine Grafikdatei an Stelle des Vertex angezeigt werden und in manchen 3D-Bibliotheken können auch 3D-Modelle an jedes Vertex gezeichnet werden, jedoch unterstützt THREE.js das nicht. [vdB00]

Deshalb wird in SumoViz3D die Anzeige von Grafikdateien an jedem Vertex genutzt. Nach der Berechnung der Position des Partikels wird die Grafik entsprechend der Tiefe skaliert und dann frontal ohne Drehung an dieser Stelle angezeigt. Dieses Verfahren wird als *Billboard* (engl. für Plakatwand) bezeichnet, da die Grafik eigentlich nur zweidimensional ist, sich aber nicht mitdreht und deshalb immer nur frontal auf den Betrachter zeigt. [MSDN] Damit in der Darstellung nicht auffällt, dass es sich um eine 2D-Grafik handelt wird die Grafik einer Kugel verwendet. Eine Kugel hat aus jedem Betrachtungswinkel den gleichen Umriss und so fällt nicht auf, dass die Grafik nur stets frontal zur sehen ist.

Die Billboard-Grafik ist ein transparentes PNG-Bild einer weißen Kugel, auf der Schattierungen zu sehen sind. Neben dem Vertex-Array wird ein Color-Array angelegt, das zum jeweiligen Index die Farbe des entsprechenden Partikel speichert. Diese Farbe wird auf das Billboard multipliziert und so werden vor allem die hellen Flächen der Kugel entsprechend eingefärbt (vgl. Abbildung 7.9).

### 7.5.3 Einfärbung der Fußgängerobjekte

Unabhängig von den Darstellung der Fußgänger als 3D-Modell oder als Partikel gibt es verschiedene Möglichkeiten die Fußgänger einzufärben und damit weitere Daten zu visualisieren.

#### Einfärbung nach Dichte

Pro Zeitpunkt und Fußgänger ist die Dichte schon in der Ausgabedatei gespeichert und müssen nicht mehr berechnet werden. Der Wertebereich ist mit  $0.0 \leq density \leq 1.0$  definiert. Zur Farbgebung der Fußgänger ist die Verwendung des HSV-Farbraums (vgl. Abbildung 7.10) geeignet, zudem THREE.js die Methode `setHSV(h, s, v)` implementiert. Der Farbwert wird beim HSV-Farbraum über den Farbton *H* (engl. *hue*), die Farbsättigung *S* (engl. *saturation*) und den Hellwert *V* (engl. *value*) bestimmt. Während *S* und *V* als Prozentwerte angegeben werden, wird für *H* eine Farbwinkelangabe auf dem Farbkreis verwendet. [Wie]

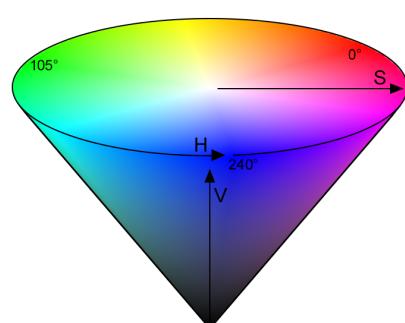


Abbildung 7.10: HSV-Farbraum

Für die Dichte soll der Farbton zwischen grün ( $H = 105^\circ$ ) für einen Wert von 0.0 und rot ( $H = 0^\circ$ ) für den Maximalwert von 1.0 variieren. Sättigung und Hellwert bleiben unverändert. Das Setzen des Farbwerts bei Verwendung eines Partikelsystems für den Partikel  $i$  sieht wie folgt aus:

```
particles.colors[i].setHSV(105-105*density,1,1);
```

Dieser Funktionsaufruf muss für jeden Fußgänger zu jedem Zeitschritt der Simulation erfolgen.

### Einfärbung nach Geschwindigkeit

Es ist möglich die Geschwindigkeit der Fußgänger ebenfalls farblich zu kodieren. Jedoch ist diese Information nicht in den Datensätzen enthalten und muss in Echtzeit berechnet werden. Dazu wird die Distanz zwischen den aktuellen Koordinaten und den Koordinaten des vorhergehenden Zeitpunkts berechnet. Je höher die zurückgelegte Distanz, um so höher auch die Geschwindigkeit. Empirisch wurde ermittelt, dass die zurückgelegte Distanz zwischen zwei aufeinander folgenden Zeitpunkten zwischen 0 und 30 liegt. Entsprechend wird mit einer ähnlichen Funktion wie bei der Einfärbung nach Dichte der Farbwert des Fußgängers bestimmt. Niedrige Geschwindigkeiten werden in rot ( $H = 0^\circ = 360^\circ$ ), hohe Geschwindigkeiten in blau ( $H = 240^\circ$ ) dargestellt (vgl. Abbildung 7.11). War der Fußgänger im vorherigen Schritt nicht sichtbar, ist es nicht möglich eine Geschwindigkeit zu berechnen und der Fußgänger wird in grau dargestellt.

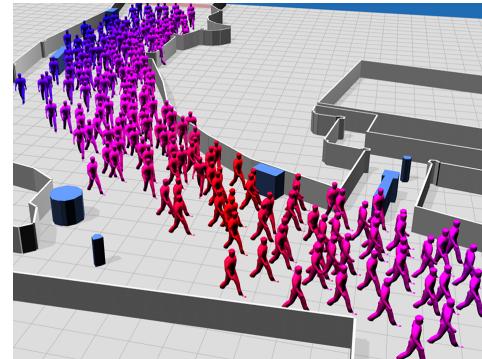


Abbildung 7.11: Einfärbung nach Geschwindigkeit

### Einfärbung nach Gruppen

In der Ausgabedatei kann ein Datenblock angelegt werden, in dem einzelne Fußgänger zu Gruppen geordnet werden können. Gruppen haben Auswirkungen auf die Simulationsergebnisse und sind deshalb auch bei der Auswertung interessant. Die Gruppenzugehörigkeit kann sich, sofern vorhanden, auch farblich visualisieren lassen. Bei der Farbverteilung der Gruppen wird bei  $H_0 = 0^\circ$  begonnen und nach der Funktion  $H_i = (i * 173) \bmod 360$  die weiteren Farbwerte vergeben. Durch die Verwendung der Primzahl 173 ist bereits bei zwei Gruppen eine gute Unterscheidung der Farben möglich, bei vielen Gruppen die Wahrscheinlichkeit einer doppelten Verwendung einer Farbe niedrig und die Verteilung über das Farbspektrum groß. Die Einfärbung der Fußgängerobjekte nach Gruppenzugehörigkeit muss nur einmal zu Beginn durchgeführt werden und ändert sich während der Visualisierung nicht mehr.

# Kapitel 8

## Analyse und Zusammenfassung

### 8.1 Leistungsanalyse von SumoViz3D

Um die Leistungsfähigkeit von SumoViz3D zu analysieren wurden zwei Tests durchgeführt. Einmal wurde ein relativ kleiner Datensatz verwendet, in einem weiteren Test eine sehr große Datenmenge. Die Tests wurden auf einem Testsystem<sup>1</sup> in Google Chrome<sup>2</sup> durchgeführt und können sich je nach System extrem unterscheiden. Auch auf dem selben System hängt die Leistungsfähigkeit noch von vielen Faktoren wie beispielsweise der Auslastung der Grafikkarte durch verschiedene Monitorgrößen, den verwendeten Browser oder den Treibern der Grafikkarte ab.

Grundsätzlich sind zwei Faktoren ausschlaggebend für die Geschwindigkeit von SumoViz3D. Einerseits müssen zu Beginn der Visualisierung alle benötigten Daten übertragen werden. Das beinhaltet hauptsächlich Simulationsdaten, die je nach Umfang der Simulation erhebliche Größen annehmen können. Hinzu kommen JavaScript-Bibliotheken und -Code (832 KB), 3D-Modelle und Texturen (710 KB), sowie HTML und CSS zur Darstellung (43 KB). In Summe betragen die benötigten Daten 1585 KB, können aber variieren, wenn beispielsweise aufwändiger Texturen verwendet werden.

Andererseits benötigt auch das Einlesen und Verarbeiten der Daten Zeit. Diese wird im Folgenden anhand zweier Beispieldatensätze analysiert.

#### 8.1.1 Analyse mit bis zu 10000 Fußgängern

Die Analyse der Leistung wurde mit einem Testdatensatz einer Simulation des “Fritz-Walter-Stadions” (ehemals Betzenbergstadion) in Kaiserslautern durchgeführt. Das Stadion hat eine schwierige Lage, da es in einem Wohngebiet liegt und die Fußgänger das Stadion nicht in alle Richtungen verlassen können.

Die Ausgabedatei der Simulation hat eine Größe von 63 MB und enthält 101 Geometrieobjekte sowie 301 Simulationsschritte der Fußgängerdaten. Die maximale Anzahl dargestellter

---

<sup>1</sup>Apple MacBook Pro unter Mac OS 10.8.1, 2,3 GHz Intel Core i7, 8GB Arbeitsspeicher, NVIDIA GeForce GT 650M mit 1GB Grafikspeicher

<sup>2</sup>Version 21.0.1180.89

Fußgängerobjekte wird im letzten Simulationsschritt mit 10716 Objekten erreicht. Durchschnittlich werden 5392 Objekte dargestellt. Neben der Übertragungszeit der Daten benötigt bei dieser Simulation das parsen der Geometrie- und Fußgängerdaten zu einem JavaScript-Objekt 12,25 Sekunden. Die Anzahl der Fußgänger steigt bei diesem Datensatz stetig an und die Leistung wurde von 0 bis 10000 angezeigten Fußgängern an 21 Punkten (in Schritten von 500) analysiert.

Die Werte in den Abbildungen 8.1, 8.2 und 8.3 wurden mittels *Webkit Inspector* erhobenen. JavaScripts *Garbage Collection* oder andere Auslastung des Testsystems lassen wie gemessenen Werte variieren. Deshalb wurde jeweils der Mittelwert mehrerer Messungen gebildetSchwankungen auszugleichen. Nichtsdestotrotz handelt es sich bei dieser Messung um die Werte eines einzelnen Testsystems, die in anderen Umgebungen abweichen können.

Der Arbeitsspeicherbedarf der Anwendung ist für die 3D-Darstellung etwa 150 MB, bei der Verwendung des Partikelsystems etwa 120 MB. Der Unterschied entsteht durch die im Speicher abgelegten 3D-Modelle, die bereits zu Beginn der Szene erstellt werden und somit unabhängig von der Anzahl der angezeigten Objekte sind. Beide Darstellungsoptionen stellen für aktuelle Systeme kein Problem im Bezug auf den verfügbaren Speicher dar.

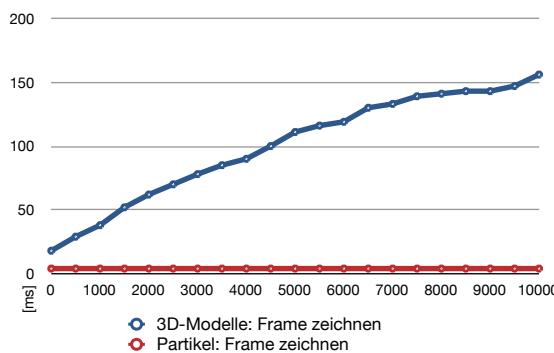


Abbildung 8.1: Dauer für das Zeichnen eines Frames in Millisekunden

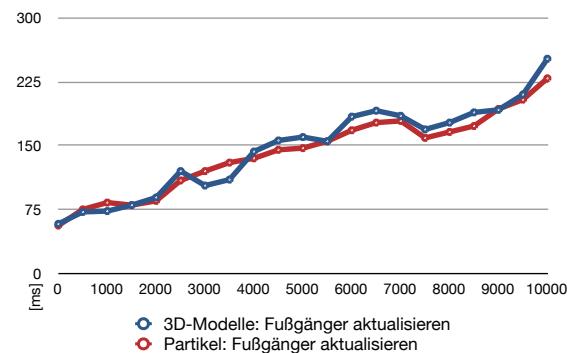


Abbildung 8.2: Dauer für das Aktualisieren der Fußgänger in Millisekunden

Bei der Messung der JavaScript- und WebGL-Aufrufe lassen sich zwei Engstellen erkennen. Zum einen ist das die Berechnung der einzelnen Frames in der Grafikkarte, wie in Abbildung 8.1 dargestellt. Ausgeführt wird der Aufruf durch die Funktion `requestAnimationFrame` (vgl. Kapitel 7.3). Bei Verwendung der 3D-Modelle steigt die Berechnungsdauer kontinuierlich mit der Zahl der Fußgänger an. Dagegen dauert das Zeichnen des Partikelsystems konstant etwa 4 ms, unabhängig von der Fußgängeranzahl. Bei der Berechnungszeit eines Frames ist es irrelevant ob die Animation abgespielt wird oder nicht, da die Grafikkarte die Frames auch bei angehaltener Simulation berechnen muss.

Während des Abspielens müssen zusätzlich die Fußgängerpositionen und -farben aktualisiert werden. Diese Berechnung stellt die zweite Engstelle dar. Unabhängig von der Berechnung der Frames wird alle 200 ms der nächste Animationsschritt geladen. Die Aktualisierung muss per JavaScript im Prozessor erfolgen. Hier verhalten sich Partikelsystem und 3D-Modelle nahezu identisch, wie in Abbildung 8.2 zu erkennen ist. Die Ausführungszeit bei der Darstellung der 3D-Modelle ist etwas höher, da die Drehung der Figuren zusätzlich berechnet werden muss. Ab etwa 9000 Objekten überschreitet die Berechnungszeit die 200 ms und damit kann nicht mehr jeder einzelne Animationsschritt dargestellt werden.

Abschließend wurde die Framerate (engl. *frames per second*, kurz *fps*) analysiert. Dieser Wert gibt die Anzahl der Bilder an, die pro Sekunde in das canvas-Element gezeichnet werden und ist hauptsächlich abhängig von den Ausführungszeiten der in Abbildung 8.1 und 8.2 betrachteten Funktionen. Welche Framerate für eine flüssige Animation notwendig ist unterscheidet sich im subjektiven Empfinden, spätestens bei einer Framerate unter 20 fps ist jedoch ein deutliches Ruckeln bemerkbar. Ist die Animation pausiert, so müssen keine neuen Positionen der Fußgänger berechnet werden und es steht mehr Leistung für die Berechnung der Frames zur Verfügung. Die Framerate bei der Verwendung des Partikelsystems ist im pausierten Zustand konstant bei 55 fps, während des Abspielens fällt sie aber mit zunehmender Fußgängerzahl. Für die Größe des Testdatensatzes von etwa 10000 Fußgängern ist eine flüssige Darstellung gerade noch möglich. Bei der Verwendung der 3D-Modelle fällt die Framerate deutlich schneller ab. Pausiert lassen sich etwa 2000 Fußgängerobjekten mit 20 fps darstellen, läuft die Animation sind ungefähr 1500 Objekte bei gleicher Framerate möglich. An dieser Stelle mangelt es an der Leistungsfähigkeit der Grafikkarte, mit schnelleren Grafikkarten wären mehr 3D-Modelle möglich.

Die Darstellung als Partikelsystem ist für große Datenmengen geeignet, da auch während dem Abspielen der Simulation noch eine akzeptable Framerate erreicht werden kann. Für Standbilder und zur Analyse einzelner Zeitpunkte können trotzdem die 3D-Modelle verwendet werden. Ab 2000 Fußgängern ist die Darstellung als Partikelsystem notwendig, bei Szenen mit mehr als 10000 Objekten war auf dem Testsystem keine flüssige Animation mehr möglich.

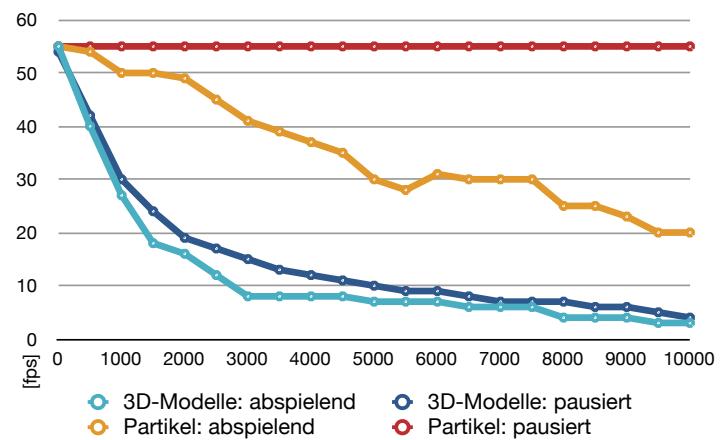


Abbildung 8.3: Vergleich der Framerate für Partikel und 3D-Modelle

## 8.2 Mögliche Erweiterungen für SumoViz3D

Mit der Entwicklung von SumoViz3D wurde der Grundstein für eine vollwertige Visualisierung und Auswertung der Fußgängersimulationsdaten gelegt. Eine große Menge an Erweiterungsmöglichkeiten steht offen. Im Folgenden werden einige davon kurz erläutert.

- Bisher ist die Darstellung verschiedener Stockwerke nicht möglich. Diese Daten sind aber in der Ausgabedatei mit gespeichert und stehen somit zur Verfügung. Gerade durch die dreidimensionale Darstellung in SumoViz3D würde es sich anbieten diese Funktionalität zu integrieren. Problematisch bei der Darstellung der Stockwerke ist die Verdeckung tieferliegenderer Stockwerke.
- Aktuell ist es zwar möglich, Standbilder der Visualisierung herunterzuladen, nicht aber den Ablauf der Animation in Form eines Videos. Die Standbildfunktion ist als Teil der canvas-API definiert und in der Funktion `toDataURL()` implementiert. Zwar ist es

nicht möglich Bewegtbilder direkt aus dem canvas-Element zu exportieren, aber über den Abruf aller Einzelbilder könnte ein Video generiert werden. Das Video müsste entweder clientseitig berechnet, oder alle Einzelbilder an der Server übertragen werden. Bei der clientseitigen Berechnung ist jedoch die Funktionalität mittels JavaScript sehr eingeschränkt.

- Die Auswertung und Aufbereitung der Simulationsdaten lässt sich noch wesentlich erweitern. Weitere Einfärbungsoptionen für die Fußgänger beispielsweise nach Quelle oder Ziel sind denkbar. Entsprechende Daten sind nicht in den Datensätzen enthalten und müssten berechnet werden. Denkbar sind auch statistische Auswertungswerzeuge, um zum Beispiel den Fußgängerdurchfluss an bestimmten Stellen zu messen.
- Da durch die Änderung des Datenformats jeder Fußgänger nun einzeln identifizierbar ist, können diese Daten für die Visualisierung genutzt werden. Beispielsweise können die Wege der einzelnen Fußgänger mittels Linien dargestellt werden, die Kamera kann den Weg eines Fußgängers verfolgen und zu jedem Fußgänger können Auswertungen der Geschwindigkeit und Dichte über die Zeit generiert werden.
- Die einzelnen Zeitpunkte der Simulation werden in dieser Version von SumoViz3D schrittweise gerendert, wodurch die Fußgängerobjekte von einer Position zur nächsten springen. In einer Erweiterung könnte zwischen zwei aufeinanderfolgenden Schritten interpoliert und so ein flüssigerer Animationablauf erzeugt werden. Dazu muss das Zeichnen der Fußgänger in der `requestAnimationFrame`-Methode stattfinden. In diesem Zuge kann auch eine Steuerung der Abspielgeschwindigkeit implementiert werden.
- Die Übertragung der Simulationsdaten über das Internet kann, gerade bei umfangreichen Szenarien, einige Zeit in Anspruch nehmen. Hier könnten die Daten partiell bei Bedarf nachgeladen werden. Mit der Verwendung von JavaScript *WebWorkers* wäre ein paralleles Herunterladen und parsen der kommenden Simulationsschritte möglich, ohne den Ablauf der Visualisierung zu unterbrechen. Größe der Daten und Frequenz der Anfragen sollten abhängig von der Verbindungsqualität ermittelt und angepasst werden.

## 8.3 Zusammenfassung

Zusammenfassend lässt sich sagen, dass die webbasierte Entwicklung von 3D-Applikationen noch in einer frühen Entwicklungsphase steckt. Sowohl bei der Implementierung der WebGL- und HTML5-Funktionen in den verschiedenen Browsern, als auch bei der verfügbaren Rechenleistung ist in den nächsten Jahren noch ein deutlicher Fortschritt zu erwarten. Auch die verwendete Grafikbibliothek THREE.js befindet sich in einer stetigen Weiterentwicklung und wird in neueren Versionen noch effizienter und schneller werden. Die SumoViz-Architektur, im speziellen die CouchDB-Datenbank, nutzt einen klassischen Trade-off der Informatik. Fehlende Rechenleistung wird durch einen Mehrverbrauch an Speicherplatz ausgeglichen. Die von CouchDB berechneten Daten werden gespeichert und müssen bei einem Aufruf nicht erneut berechnet werden. So kann die einmalige Berechnung der Daten längere Zeit dauern, da die Daten danach sofort zur Verfügung stehen. Zwar kostet dieses Verfahren viel Speicherplatz, aber nur so ist eine derartige Applikation mit den aktuellen Technologien überhaupt realisierbar. Trotzdem sind Webapplikationen eine gute Möglichkeit plattformübergreifend An-

wendungen zu entwickeln. Durch die Client-Server-Architektur stehen Daten überall zur Verfügung. Vor einigen Jahren wäre eine dreidimensionale Simulationsvisualisierung im Browser noch komplett undenkbar gewesen. Mit SumoViz3D steht jetzt ein umfangreiches Werkzeug für die Darstellung und Analyse von Fußgängerdaten zur Verfügung. Damit kann es sowohl für die Forschung an neuen Modellen der Simulation als auch die Sicherheit auf Großveranstaltungen einen Beitrag leisten.

## Anhang A

### Beigefügte CD

Auf der beigefügten CD befindet sich folgender Inhalt:

- SumoViz3D Quellcode
- verschiedene Testdatensätze (u.a. “Betzenberg”, vgl. Kapitel 8.1.1)
- MapReduce-Funktionen (vgl. Kapitel 6.2)
- diese Dokumentation, zusammen mit allen eingebundenen Grafiken

# Abbildungsverzeichnis

2.1	HTML5 Spezifikations-Übersicht (CC-BY-3.0 Peter Kröner) [spec]	4
5.1	SumoViz [Isi12]	17
5.2	Struktureller Aufbau von SumoViz3D	18
7.1	Screenshot des SumoViz3D-Interfaces	25
7.2	Z-Fighting zwischen Grundebene und Feldern bei großer Kameradistanz	26
7.3	Darstellung von normalen Objekten (blau)	27
7.4	Darstellung von Bäumen mittels COLLADA-Modell	27
7.5	Darstellung von Wänden ohne Textur	28
7.6	Darstellung von Quelle, Ziel und anderen Feldern (rot)	28
7.7	Kameraposition	33
7.8	Darstellung der Fußgänger mittels 3D-Modell	35
7.9	Darstellung der Fußgänger mittels Billboard-Grafik	35
7.10	HSV-Farbraum	36
7.11	Einfärbung nach Geschwindigkeit	37
8.1	Dauer für das Zeichnen eines Frames in Millisekunden	39
8.2	Dauer für das Aktualisieren der Fußgänger in Millisekunden	39
8.3	Vergleich der Framerate für Partikel und 3D-Modelle	40

# Tabellenverzeichnis

3.1	Deklarative und imperative Grafikdarstellung im Web [X3D]	7
4.1	Marktanteile der führenden Webbrowser[Sta1] [Sta3]	11
6.1	Struktur der Fußgängerdaten	20
6.2	Struktur der Geometriedaten	21
6.3	Struktur der Gruppendaten	21
7.1	Event-Listener und zur Steuerung modifizierte Variablen	34

# Literaturverzeichnis

[Kei10] **HTML5 for Web Designers**

*Jeremy Keith*

A Book Apart, 2010

[Pil10] **Dive into HTML5**

*Mark Pilgrim*, 2010

[CB97] **The Annotated VRML 97 Reference Manual**

*Rikk Carey, Gavin Bell*

Addison-Wesley Professional, 1997

[Shr09] **OpenGL Programming Guide** (Seventh Edition)

*Dave Shreiner*

Addison-Wesley Longman, 2009

[CJ12]  **WebGL Beginner's Guide**

*Diego Cantor, Brandon Jones*

Packt Publishing, 2012

[Isi12] **SumoViz, HTML5-based Visualization of Pedestrian Simulation Data**

*Mustafa K. Isik*

TU München, 10. Mai 2012

[ALS10] **CouchDB: The Definitive Guide**

*J. Chris Anderson, Jan Lehnardt, Noah Slater*

O'Reilly, 2010

[Hol11] **Writing and Querying MapReduce Views in CouchDB**

*Bradley Holt*

O'Reilly, 2011

[McL11] **What Is Node?**

*Brett McLaughlin*

O'Reilly, 2011

[Fla06] **JavaScript: The Definitive Guide**

*David Flanagan*

O'Reilly, 2006

## Web-Referenzen

- [Tre11] Neil Trevett: Building Markets for Advanced Devices through Open Standards  
[http://www.khronos.org/assets/uploads/developers/library/overview/khronos\\_overview.pdf](http://www.khronos.org/assets/uploads/developers/library/overview/khronos_overview.pdf), abgerufen am 18.08.2012.
- [SGI] SGI: OpenGL  
<http://www.sgi.com/products/software/opengl/?/overview.html>,  
abgerufen am 18.08.2012.
- [Khr] About The Khronos Group  
<http://www.khronos.org/about>, abgerufen am 18.08.2012.
- [Goo] Android Developers: Andriod 2.2 APIs  
<http://developer.android.com/about/versions/android-2.2.html>,  
abgerufen am 18.08.2012.
- [App] OpenGL ES for iOS  
<https://developer.apple.com/devcenter/ios/resources/opengl-es/>,  
abgerufen am 18.08.2012.
- [Cab] Luz Caballero (Opera): An introduction to WebGL  
<http://dev.opera.com/articles/view/an-introduction-to-webgl/>,  
abgerufen am 18.08.2012.
- [Ope] Khronos Group: WebGL - OpenGL ES 2.0 for the Web  
<https://www.khronos.org/webgl/>, abgerufen am 18.08.2012.
- [W3c] W3C: HTML5, The canvas element  
<http://www.w3.org/TR/html5/the-canvas-element.html#the-canvas-element>,  
abgerufen am 18.08.2012.
- [WHc] WHATWG: CanvasContexts  
<http://wiki.whatwg.org/wiki/CanvasContexts>, abgerufen am 18.08.2012.
- [Whc2] WHATWG: The canvas element  
<http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html>, abgerufen am 18.08.2012.
- [Sta1] StatCounter: Global Stats, Top 5 Browsers from Jun to Jul 2012  
<http://gs.statcounter.com/#browser-ww-monthly-201206-201207-bar>,  
abgerufen am 18.08.2012.
- [MSFT] Microsoft Security Research & Defense: WebGL Considered Harmful  
<http://blogs.technet.com/b/srd/archive/2011/06/16/webgl-considered-harmful.aspx>, abgerufen am 18.08.2012.
- [Sil] Silverlight Blog: Silverlight 5 Available for Download Today  
<http://blogs.msdn.com/b/silverlight/archive/2011/12/09/silverlight-5-available-for-download-today.aspx>, abgerufen am 18.08.2012.
- [IEWeb] IEWebGL: WebGL for Internet Explorer  
<http://iewebgl.com>, abgerufen am 18.08.2012.

- [MDN] Mozilla Developer Network: Same origin policy for JavaScript  
[https://developer.mozilla.org/en-US/docs/Same\\_origin\\_policy\\_for\\_JavaScript](https://developer.mozilla.org/en-US/docs/Same_origin_policy_for_JavaScript), abgerufen am 18.08.2012.
- [W3cors] W3C: Cross-Origin Resource Sharing  
<http://www.w3.org/TR/cors/>, abgerufen am 18.08.2012.
- [AoGL] OpenGL ES Programming Guide for iOS  
[http://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES\\_ProgrammingGuide/Introduction/Introduction.html](http://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/Introduction/Introduction.html), abgerufen am 19.08.2012.
- [SNE] Sony Ericsson Dev: WebGL  
<https://github.com/sonyericssondev/WebGL>, abgerufen am 19.08.2012.
- [Sta2] StatCounter Global Stats: Top 8 Mobile Operating Systems from Jun to Jul 2012  
[http://gs.statcounter.com/#mobile\\_os-ww-monthly-201206-201207-bar](http://gs.statcounter.com/#mobile_os-ww-monthly-201206-201207-bar), abgerufen am 19.08.2012.
- [Con1] James Forshaw: WebGL – A New Dimension for Browser Exploitation  
<http://www.contextis.com/research/blog/webgl/>, abgerufen am 19.08.2012.
- [Con2] James Forshaw: WebGL – More WebGL Security Flaws  
<http://www.contextis.com/research/blog/webgl/>, abgerufen am 19.08.2012.
- [Moz] Benoit Jacob: Cross-domain WebGL textures disabled in Firefox 5  
<http://tinyurl.com/hacks-mozilla>, abgerufen am 19.08.2012.
- [Chr] Google Chrome Blog: A dash of speed, 3D and apps  
<http://chrome.blogspot.de/2011/02/dash-of-speed-3d-and-apps.html>, abgerufen am 19.08.2012.
- [FF4] Firefox 4 Release Notes  
<http://www.mozilla.org/en-US/firefox/4.0/releasenotes/>, abgerufen am 19.08.2012.
- [Saf] WebGL Now Available in WebKit Nightlies  
<http://www.webkit.org/blog/603/webgl-now-available-in-webkit-nightlies/>, abgerufen am 19.08.2012.
- [Ope] Opera 12.00 for Windows Changelog  
<http://www.opera.com/docs/changelogs/windows/1200/>, abgerufen am 19.08.2012.
- [Sta3] StatCounter Global Stats: Top 12 Browser Versions from Jun to Jul 2012  
[http://gs.statcounter.com/#browser\\_version-ww-monthly-201206-201207-bar](http://gs.statcounter.com/#browser_version-ww-monthly-201206-201207-bar), abgerufen am 19.08.2012.
- [W3no] W3C HTML5: Things that you can't do with this specification because they are better handled using other technologies that are further described herein  
<http://www.w3.org/TR/html5/no.html>, abgerufen am 20.08.2012.
- [W3a] W3C: Über das W3C  
<http://www.w3.org/TR/html5/no.html>, abgerufen am 21.08.2012.

- [W3m] W3C: Current members  
[https://developer.mozilla.org/en-US/docs/Gecko\\_DOM\\_Reference/Introduction](http://www.w3.org/Consortium/Member>List</a>, abgerufen am 21.08.2012.</p><p>[MDN1] Mozilla Developer Network: What is the DOM?<br/><a href=), abgerufen am 21.08.2012.
- [HTML] HTML5 Rocks: WebGL Fundamentals  
[http://www.html5rocks.com/en/tutorials/webgl/webgl\\_fundamentals/](http://www.html5rocks.com/en/tutorials/webgl/webgl_fundamentals/),  
abgerufen am 20.8.2012
- [THREE] THREE.js r50 Dokumentation  
<http://mrdoob.github.com/three.js/docs/50/>, abgerufen am 21.08.2012
- [X3D] X3DOM: About  
[http://www.x3dom.org/?page\\_id=2](http://www.x3dom.org/?page_id=2), abgerufen am 21.08.2012
- [Web3] X3D and HTML5  
[http://www.web3d.org/x3d/wiki/index.php/X3D\\_and\\_HTML5](http://www.web3d.org/x3d/wiki/index.php/X3D_and_HTML5),  
abgerufen am 21.08.2012
- [VRML1a] The Virtual Reality Modeling Language: Version 1.0 Specification  
<http://www.web3d.org/x3d/specifications/vrml/VRML1.0/>,  
abgerufen am 21.08.2012
- [VRMLa] The Virtual Reality Modeling Language: International Standard  
<http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-VRML97/>,  
abgerufen am 21.08.2012
- [W32d] W3C: HTML Canvas 2D Context  
<http://www.w3.org/TR/2dcontext/>, abgerufen am 21.08.2012
- [spec] Peter Kröner: HTML5 Spezifikations-Übersicht  
<https://github.com/SirPepe/SpecGraph>, abgerufen am 22.08.2012
- [W3l] W3C: W3C Confirms May 2011 for HTML5 Last Call  
<http://www.w3.org/2011/02/htmlwg-pr.html>, abgerufen am 22.08.2012
- [VRML1a] The Virtual Reality Modeling Language: Version 1.0 Specification  
<http://www.web3d.org/x3d/specifications/vrml/VRML1.0/>,  
abgerufen am 22.08.2012
- [VRMLa] The Virtual Reality Modeling Language: International Standard  
<http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-VRML97/>,  
abgerufen am 22.08.2012
- [W32d] W3C: HTML Canvas 2D Context  
<http://www.w3.org/TR/2dcontext/>, abgerufen am 22.08.2012
- [Win] Microsoft: Timeout Detection and Recovery of GPUs  
<http://msdn.microsoft.com/en-us/windows/hardware/gg487368.aspx>,  
abgerufen am 28.08.2012

- [KhrSec] Khronos Group: WebGL Security  
<http://msdn.microsoft.com/en-us/windows/hardware/gg487368.aspx>,  
abgerufen am 28.08.2012
- [SPON1] Spiegel Online: Warum die Wege des Menschen unberechenbar sind  
<http://spon.de/ac51E>, abgerufen am 01.09.2012
- [SPON2] Spiegel Online: Der kürzeste Weg ist das Ziel  
<http://spon.de/adlhZ>, abgerufen am 01.09.2012
- [jQu] jQuery UI  
<http://jqueryui.com/home>, abgerufen am 03.09.2012
- [MrD] THREE.js Stats - JavaScript Performance Monitor  
<https://github.com/mrdoob/stats.js>, abgerufen am 01.09.2012
- [COL] COLLADA Digital Asset and FX Exchange Schema  
[https://collada.org/mediawiki/index.php/COLLADA\\_-\\_Digital\\_Asset\\_and\\_FX\\_Exchange\\_Schema](https://collada.org/mediawiki/index.php/COLLADA_-_Digital_Asset_and_FX_Exchange_Schema), abgerufen am 05.09.2012
- [W3s] W3C: WebStorage  
<http://www.w3.org/TR/webstorage/>, abgerufen am 06.09.2012
- [Res12] John Resig: How JavaScript Timers Work  
<http://ejohn.org/blog/how-javascript-timers-work/>, abgerufen am 06.09.2012
- [Iri11] Paul Irish: requestAnimationFrame for smart animating  
<http://paulirish.com/2011/requestanimationframe-for-smart-animating/>,  
abgerufen am 07.09.2012
- [Gub09] Martin Gubisch: Kugelkoordinaten  
<http://www.martingubisch.de/cms/upload/files/tutorien/09%20Kugelkoordinaten.pdf>, abgerufen am 07.09.2012
- [MSDN] Microsoft: Billboardng  
[http://msdn.microsoft.com/de-de/library/bb172358\(VS.85\).aspx](http://msdn.microsoft.com/de-de/library/bb172358(VS.85).aspx),  
abgerufen am 07.09.2012
- [vdB00] John van der Burg: Building an Advanced Particle System  
[http://www.gamasutra.com/view/feature/3157/building\\_an\\_advanced\\_particle\\_.php](http://www.gamasutra.com/view/feature/3157/building_an_advanced_particle_.php), abgerufen am 07.09.2012
- [Wie] Lyle Wiedeman: Information on the HSV color space  
<http://dcssrv1.oit.uci.edu/~wiedeman/cspace/me/infohsv.html>,  
abgerufen am 07.09.2012