Andreas Gratzer BSc

{v}Plot.js
A Web Based Information Visualization Framework

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Software Development and Business Management

submitted to

Graz University of Technology

Supervisor

Assoc.Prof. Dipl-Ing. Dr.techn. Denis Helic

Knowledge Technologies Institute (KTI)

Second Supervisor
Dipl-Ing. Florian Geigl BSc

Graz, May 2015

Andreas Gratzer BSc

{v}Plot.js
A Web Based Information Visualization Framework

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

Masterstudium Softwareentwicklung - Wirtschaft

eingereicht an der

Technischen Universität Graz

Betreuer
Assoc.Prof. Dipl-Ing. Dr.techn. Denis Helic

Knowledge Technologies Institute (KTI)

Zweitbetreuer
Dipl-Ing. Florian Geigl BSc

Graz, Mai 2015

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all ma- terial which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Danksagung

Am Ende dieser Masterarbeit angelangt, möchte ich mich bei allen bedanken, die mich unterstützt haben. Als allererstes gilt mein Dank Assoc.Prof. Dipl-Ing. Dr.techn. Denis Helic, der mir die Umsetzung dieser Arbeit ermöglicht hat und darüber hinaus auch die Freiheit gelassen hat, dieses Projekt nach meinen eigenen Vorstellungen zu gestalten. Besonderer Dank gilt auch Dipl-Ing. Florian Geigl für seine unermüdliche Unterstützung über die gesamte Projektlaufzeit hinweg. Ohne seine Ratschläge, Hinweise und Anregungen wäre diese Masterarbeit nicht möglich gewesen. Ebenfalls bedanken möchte ich mich bei Ao.Univ.-Prof. Dipl.-Ing. Dr.techn Keith Andrews für sein Feedback. Seine übermittelten Unterlagen waren sehr hilfreich und bildeten die Grundlage für einen wichtigen Teil dieser Arbeit. Großer Dank gebührt auch meiner Mutter, die zahlreiche Stunden Korrektur gelesen hat und durch kritisches Hinterfragen wesentlich zur Verbesserung dieser Arbeit beigetragen hat. Außerdem möchte ich mich bei Brigitta Pusterhofer und bei Maria Kresser bedanken, deren Korrekturvorschläge sehr hilfreich waren.

Abstract

"It makes the invisible visible" is one possible description of the term Computer-Based-Data-Visualisation. Information Visualisation, which is the process of visualising abstract data in order to gain insight, is one of its main subfields. The visualisation of abstract data is only one part of the story. Dynamic and interactive facilities are important as well. Furthermore, non data elements like legends or axes can improve the understanding of important information, too.

{v}Plot has been created as a part of this thesis. The aim of this project was to establish a Data Visualisation Framework, which supports the process of knowledge acquisition. Primarily it was designed for the field of 3D data visualisation. The dimensions can be reduced to two or even extended to four. Additional dimensions are automatically generated if these are missing within a dataset (e.g.: 1D data). Interactive facilities are provided. Click-Activities are starting animations, loading internal or external content and are providing additional information. Programming skills are not required. Components of a plot can be assembled via drag&drop. Each element can be configured through Graphical User Interfaces. If a Plot-Template already exists, new plots can be generated automatically. Datasets within a folder structure can be mapped to a Plot-Group.
{v}Plot has been implemented completely in Javascript. For the graphical representation THREE.js, which is a freely accessible WebGL Framework, has been used. In order to reach an audience as big as possible and simplify the installation process, {v}Plot has been embedded within a Wordpress Plugin.

Kurzfassung

Die computerunterstützte Datenvisualisierung macht das Nichtsichtbare sichtbar. Einen Teilbereich bildet die Informationsvisualisierung. Hierbei handelt es sich um einen Prozess, in welchem abstrakte Daten visuell dargestellt werden. Neben der eigentlichen Visualisierung der Daten spielen unterstützende Elemente, wie Legenden und Achsen, eine wichtige Rolle und tragen beträchtlich zum Verständnis bei.
Das in dieser Arbeit beschriebene Projekt {v}Plot hat zum Ziel, den Prozess der Wissensgenerierung zu unterstützen. Es ist für die Visualisierung von Daten im Bereich 3D ausgelegt. Die Darstellung kann aber auch auf zwei Dimensionen reduziert oder auf vier Dimensionen erweitert werden. Liegen einem Benutzer nur Daten im eindimensionalen Bereich vor, werden fehlende Dimensionen automatisch erzeugt. Des Weiteren kann ein Plot interaktiv gestaltet werden. Über sogenannte Click-Activities können Informationen eingeblendet, Animationen gestartet und interne sowie externe Inhalte geladen werden. Programmierkenntnisse sind nicht zwingend erforderlich. Anhand von grafischen Oberflächen, welche von Modulen verwaltet werden, kann ein Plot via drag&drop zusammengesetzt und anschließend konfiguriert werden.
Plots können auch anhand gegebener Datensätze automatisch generiert werden. Befinden sich diese Datensätze in einer verschachtelten Ordnerstruktur, so kann auch diese Struktur erfasst und als Plot-Gruppe dargestellt werden. Umgesetzt wurde das Projekt {v}Plot in Javascript. Für die grafische Darstellung von Daten wurde auf das WebGL Framework THREE.js zurückgegriffen. Um den Installationsaufwand auf ein Minimum zu reduzieren und ein möglichst breites Publikum zu erreichen, wurde das Framework in ein Wordpress Plugin eingebettet.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

> " Un bon croquis vaut mieux qu'un long discours (A good sketch is better than a long speech) " (Napoléon Bonaparte)

Im deutschsprachigen Raum ist wohl die Redewendung „Ein Bild sagt mehr als tausend Worte“ weitaus bekannter. Sinngemäß läuft es aber auf das Gleiche hinaus. Eine gut aufbereitete grafische Repräsentation eines Sachverhalts ist aussagekräftiger als Worte es je sein können.

Dies gilt heutzutage durch die zur Verfügung stehenden Computer-Systeme mehr denn je. Mit der Unterstützung von leistungsstarken Rechnern können riesige Datensätze immer schneller analysiert werden. Ausgereifte Grafik-Technologien ermöglichen die Entwicklung von Systemen, welche informationsintensive Probleme grafisch aufbereiten und die Ableitung von Wissen erleichtern. Umfangreiche Programme können direkt in einem Browser ausgeführt werden, wodurch ein sehr breites Publikum erreicht wird. Man ist nicht mehr an ein spezielles Gerät (Home-Computer, Tablet, Smartphone) beziehungsweise an ein gewisses Betriebssystem gebunden. Neue Webgrafik-Formate (wie WebGL) ermöglichen es, animierte und interaktive Grafiken im Browser zu rendern, ohne dass zuvor spezielle Plugins eingebunden werden müssen.

Um noch einmal auf die erwähnte Redewendung zurückzukommen, es ist alles andere als einfach, ein „Bild“ zu schaffen, welches die zu Grunde liegende Aussage erfüllt. Dies ist aber, wie bereits angedeutet, kein rein technologisches Problem. Vor der Erstellung einer Visualisierung steht eine Vision, eine Vorstellung, wie eine Erkenntnis am besten dargestellt werden könnte. Im Bereich der 2D-Visualisierung von Daten steht hierfür eine Fülle

von Systemen zur Verfügung (InfoVis Toolkit[1], nvd3[2], d3[3]). Sollen aber Daten im drei- oder höherdimensionalen Bereich visualisiert werden, findet man nicht so schnell ein passendes Tool. Vor allem dann, wenn man nicht auf Programmierkenntnisse zurückgreifen kann, ist die Suche nach einem passenden System äußerst schwierig.
Aus diesen Gründen ist es durchaus sinnvoll, sich mit der Umsetzung eines Frameworks zu beschäftigen, welches einerseits einfach zu bedienen und andererseits auch umfangreich genug ist, um eine abstrakte Vision Wirklichkeit werden zu lassen.

[1] http://philogb.github.io/jit/,
[2] http://nvd3.org/
[3] http://d3js.org/

2. Aufbau der Arbeit

Kapitel 3 grenzt den allgemeinen Begriff „Visualisierung“ von der computerunterstützten Daten-Visualisierung ab. Es werden unterschiedliche Definitionen zusammengefasst und die Vorteile einer externen Repräsentation von Daten aufgelistet. Anschließend werden die drei Hauptfelder (SciVis, GeoVis und InfoVis) der computerunterstützten Daten-Visualisierung beschrieben. Auf den Bereich der Informations-Visualisierung (InfoVis) wird näher eingegangen und der vierstufige Visualisierungsprozess von Colin Ware erläutert. Abschließend wird anhand eines Beispieles verdeutlicht, dass die beschriebenen Felder auch beliebig kombiniert werden können.
In Kapitel 4 wird auf unterstützende Elemente einer Visualisierung eingegangen. Es wird beschrieben, was Legenden und Achsen sind. Der Fokus dieses Kapitels liegt auf der Positionierung von Labels auf Achsen. Positionen, welche als „schön“ empfunden werden, sind algorithmisch nicht einfach zu finden. Unterschiedliche Lösungsansätze für dieses Problem werden gegenübergestellt.
Kapitel 5 behandelt das Thema „Grafik im Browser“ und hebt hervor, dass eine Visualisierung auch als Kommunikationsmittel gesehen werden kann und dass das Web ein ideales Medium ist, um ein möglichst breites Publikum zu erreichen. Der erste Abschnitt dieses Kapitels stellt unterschiedliche Web-Grafik-Technologien vor und geht sowohl auf Vorteile als auch auf Nachteile ein. Im zweiten Abschnitt werden die 3D-WebGL-Frameworks THREE, Scene, PhiloGL und GLGE vorgestellt. Anhand von Code-Beispielen, welche alle das gleiche Thema behandeln, wird ein direkter Vergleich geboten.
Da das in dieser Arbeit beschriebene Projekt in Javascript umgesetzt wurde, wird in Kapitel 6 auf diese Sprache näher eingegangen. Der erste Teil dieses Kapitels widmet sich dem historischen Hintergrund von Javascript und den Gründen, warum diese Sprache heutzutage so beliebt ist. Des Weiteren wird auf die Relation zu ECMAScript beziehungsweise die ECMA-262 Norm

eingegangen.
Der zweite Teil behandelt das Konzept der Sprache Javascipt und umfasst die Typisierung von Variablen, Objekte und Vererbung. Abschließend wird auf das Paradigma der modularen Programmierung näher eingegangen und darauf, wie dieses in Javascript umgesetzt werden kann, obwohl keine Module unterstützt werden. Hierfür werden die beiden Standards AMD und CommonJS vorgestellt. Zudem wird ein Ausblick auf ECMAScript 6 (Harmony) geboten und aufgezeigt wie Module in dieser kommenden Version von Javascript eingesetzt werden könnten.
Kapitel 7 bis einschließlich Kapitel 10 befassen sich mit dem Projekt {v}Plot, welches im Zuge dieser Arbeit umgesetzt wurde. Das Kapitel 7 wurde in drei größere Abschnitte gegliedert. Abschnitt 7.1 umfasst die Architektur des Projektes und beschreibt die tragenden Elemente *Templates*, *Module* und *Plugins*. Im darauffolgenden Abschnitt 7.2 wird das Konzept der Datenaufbereitung und der eigentlichen Visualisierung näher erörtert. Der Datenfluss wird außerdem anhand eines konkreten Beispieles beschrieben. Der dritte Abschnitt (7.3) gewährt einen genaueren Einblick in das Projekt und umfasst Implementationsdetails von wichtigen Komponenten des Frameworks.
In Kapitel 8 wird zunächst erläutert, warum Wordpress gewählt wurde, um das Projekt {v}Plot in ein bereits bestehendes CMS zu integrieren. Eine kurze Zusammenfassung zeigt die Erstellung und konkrete Umsetzung von Wordpress-Plugins auf. Kapitel 9 illustriert Visualisierungsbeispiele, welche mit {v}Plot erstellt wurden.
Abschließend bietet Kapitel 10 einen Ausblick auf mögliche Erweiterungen des Projektes.

3. Visualisierung

> " Almost any interesting task is too difficult to be done purely mentaly. " (Ware, 2004)

Visualisierung ist ein sehr allgemeiner Begriff, welcher in vielen Bereichen und Lebenslagen verwendet werden kann. Rein literarisch gesehen hat Visualisierung nichts mit Computern zu tun. Es beschreibt eine Technik zur Fokussierung auf ein mentales Bild, um ein gewisses Ziel zu erreichen (Collins, 2005).
Auch wenn man den Kontext einschränkt und Visualisierung als rein computerunterstützte Daten-Visualisierung betrachtet, gibt es keine einheitliche Definition. McCormick spricht von einer Transformation vom Symbolischen ins Geometrische. Es werden Methoden geboten, welche das nicht Sichtbare sichtbar machen (McCormick, 1987). Owen sieht Visualisierung als Abbildung von einer Computer-Repräsentation hin zu einer für Menschen wahrnehmbaren Repräsentation (Owen, 2005). Hullman, Adar und P. Shah sagen, dass bei schwer zu verstehenden Problemen der Einsatz von Visualisierungen das Verständnis von wichtigen Informationen erleichtern kann (Hullman, Adar und P. Shah, 2011). Murray beschreibt Visualisierung als Prozess, in welchem Informationen visuell abgebildet werden. Anhand von Regeln werden Daten interpretiert, visuelle Eigenschaften abgeleitet und diese schlussendlich dargestellt (Murray, 2013). In *What is Visualization Really for?* fasst der Autor eine Vielzahl an Definitionen zusammen und definiert Visualisierung als:

> " Study of transformation from data to visual representations in order to facilitate effective and efficient cognitive processes in performing tasks involving data. " (Chen, 2013)

Wie auch immer man Visualisierung definieren will, es geht um das Erlangen von Wissen.

Wissenserwerb ist das Endresultat eines iterativen und interaktiven Prozesses, welcher als „Knowledge Discovery“ bezeichnet wird. Im heutigen Informationszeitalter werden mehr Daten generiert als gespeichert werden können. Daten müssen gefiltert, aufbereitet, transformiert und analysiert werden. Es müssen Muster innerhalb der Daten gefunden werden, welche **gültig**, **nützlich**, **unerwartet** und **verständlich** sind (Helic und Kern, 2014). Visualisierungen sind hierbei sehr hilfreich und leisten einen wichtigen Beitrag bei der Evaluation und Interpretation von gefunden Mustern.

Nicht nur bei sehr großen Datensätzen ist die computerunterstützte Daten-Visualisierung ein wichtiges Werkzeug, auch bei kleinen kann eine Visualisierung zum Verständnis beitragen. Ein klassisches und oft genanntes Beispiel, um die Relevanz einer Visualisierung zu verdeutlichen, ist das Anscombe's Quartett. In einer Tabelle werden vier Datensätze gegenübergestellt. Die Wertebereiche sowie statistische Eigenschaften weichen nicht signifikant voneinander ab. Erst mit einer visuellen Darstellung wird die Diskrepanz zwischen den einzelnen Mengen ersichtlich (siehe Abbildung 3.1).

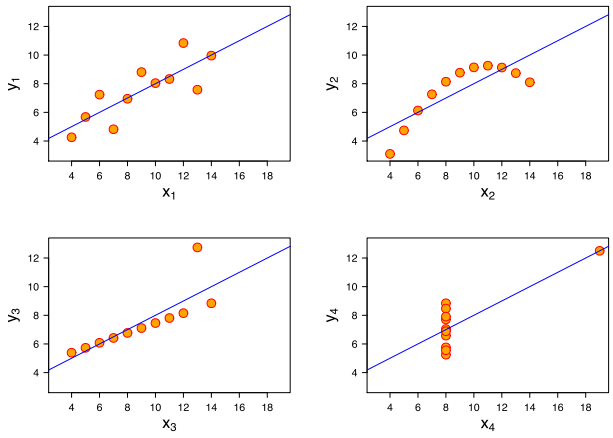


Abbildung 3.1.: Anscombe's Quartett
Entnommen aus (Wikipedia, 2015a). Licensed under CC BY-SA 3.0 via Wikimedia Commons

Ware fasst die Vorteile einer Visualisierung folgendermaßen zusammen (Ware, 2004):

- Visualisierungen bieten die Möglichkeit große Datenmengen zu erfassen. Die wichtigsten Informationen aus diesen Daten sind in kürzester Zeit verfügbar.
- Visualisierungen ermöglichen es Muster wahrzunehmen, welche ohne eine Visualisierung nicht direkt ersichtlich wären.
- Auffälligkeiten beziehungsweise Abweichungen innerhalb der Daten werden schnell erfassbar. Fehler bei der Datenerhebung werden sichtbar gemacht, wodurch Visualisierungen auch ein wichtiges Werkzeug in Bezug auf die Qualitätssicherung sind.
- Visualisierungen erleichtern die Hypothesenbildung. Muster werden sichtbar, wodurch Eigenschaften abgeleitet werden können.

Der umfassende Bereich der computerunterstützten Daten-Visualisierung kann in weitere Unterkategorien aufgeteilt werden (Andrews, 2015):

1. **SciVis**,
2. **GeoVis** und
3. **InfoVis**.

3.1. SciVis

„*Scientific Visualisation*“ beschäftigt sich mit der realistischen Darstellung von wissenschaftlichen Phänomenen, wie die Strömungen in Flüssigkeiten oder die Oberflächenbeschaffenheit von Objekten. Die Art, wie visualisiert werden kann, hängt für gewöhnlich von den Daten, beziehungsweise von dem untersuchten Phänomen selbst ab (Andrews, 2015).

3.2. GeoVis

„*Geographic Visualisation*“ umfasst die Darstellung von geografischen Informationen, für gewöhnlich in Form einer Karte (Andrews, 2015). Um ein Beispiel zu nennen, sind GPS-Koordinaten einer Reiseroute nur schwer zu deuten. Eingetragen in einer Karte sind diese Informationen aber leicht interpretierbar. Auch in diesem Bereich ist die Art, wie eine Visualisierung gestaltet werden kann, stark von den zugrunde liegenden Daten abhängig.

3.3. InfoVis

„*Information Visualisation*“ umfasst den Prozess des Verstehens von abstrakten Daten mittels einer externen Visualisierung. Im Gegensatz zu den Bereichen SciVis (Abschnitt 3.1) und GeoVis (Abschnitt 3.2) wird hier die mögliche Repräsentation nicht anhand des Kontextes, aus welchem die Daten stammen, vorgegeben. Eine Visualisierung muss vielmehr designt werden, um nützliche Informationen ableiten zu können (Andrews, 2015).

Abbildung 3.2 illustriert den vierstufigen Visualisierungsprozess von Ware. Die erste Stufe befasst sich mit dem Sammeln und der Speicherung von Rohdaten. Diese werden im zweiten Schritt in eine verwendbare Form transformiert. Mit Hilfe eines Grafiksystems werden die aufbereiteten Daten anschließend visualisiert. Abschließend wird die erstellte Grafik von einem Menschen analysiert. Ware's Visualisierungsprozess beinhaltet mehrere Feedback-Loops. Eine Analyse kann aus mehreren Gründen nicht

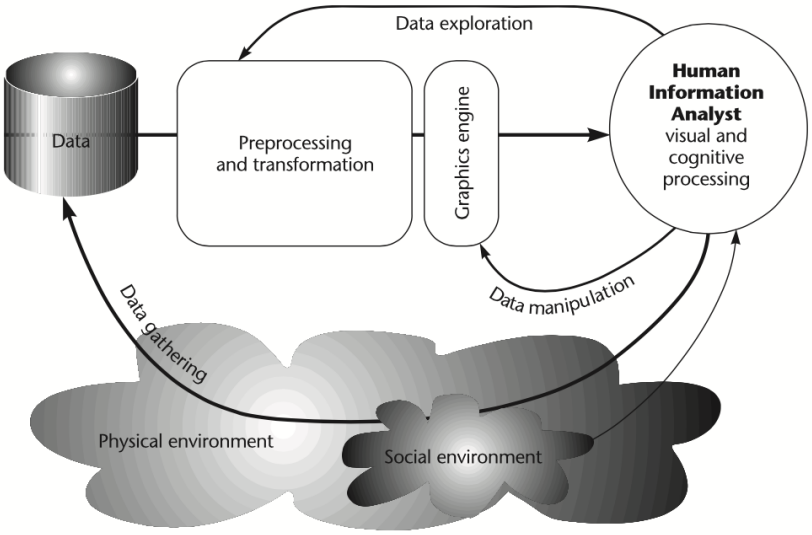


Abbildung 3.2.: 4 Visualisierungs-Stufen
Entnommen aus (Ware, 2004)

zielführend gewesen sein. Anhand der Visualisierung könnte erkannt worden sein, dass Fehler beim Sammeln der Daten aufgetreten sind. Ein Analyst könnte auch zu der Einsicht gelangt sein, dass zu wenige Daten vorliegen, oder dass es interessant wäre, mehr Daten aus einem anderen Bereich zu betrachten. Diese Änderungen an den Rohdaten werden durch den längsten Feedback-Loop zwischen Stufe eins und vier des Visualisierungsprozesses angedeutet. Durch Transformationen der Rohdaten in Stufe zwei könnte ein Sachverhalt verfälscht dargestellt worden sein, wodurch Adaptionen durchgeführt werden müssten. Diese Anpassungen illustriert der Feedback-Loop zwischen Stufe zwei und vier. Die dritte Schleife steht für eine direkte Manipulation der aufbereiteten Daten (*Data manipulation*). Um einen Sachverhalt besser verstehen zu können, ist es sinnvoll, eine Visualisierung anpassen zu können. Hierbei handelt es sich um **interaktive Elemente** eines Visualisierungssystems. Verlinkungen innerhalb der Daten, um gewisse Strukturen besser verstehen zu können oder ein „Zoom“, um einen gewissen Ausschnitt näher betrachten zu können, wären Beispiele für diesen Feedback-Loop.

3.4. DataVis

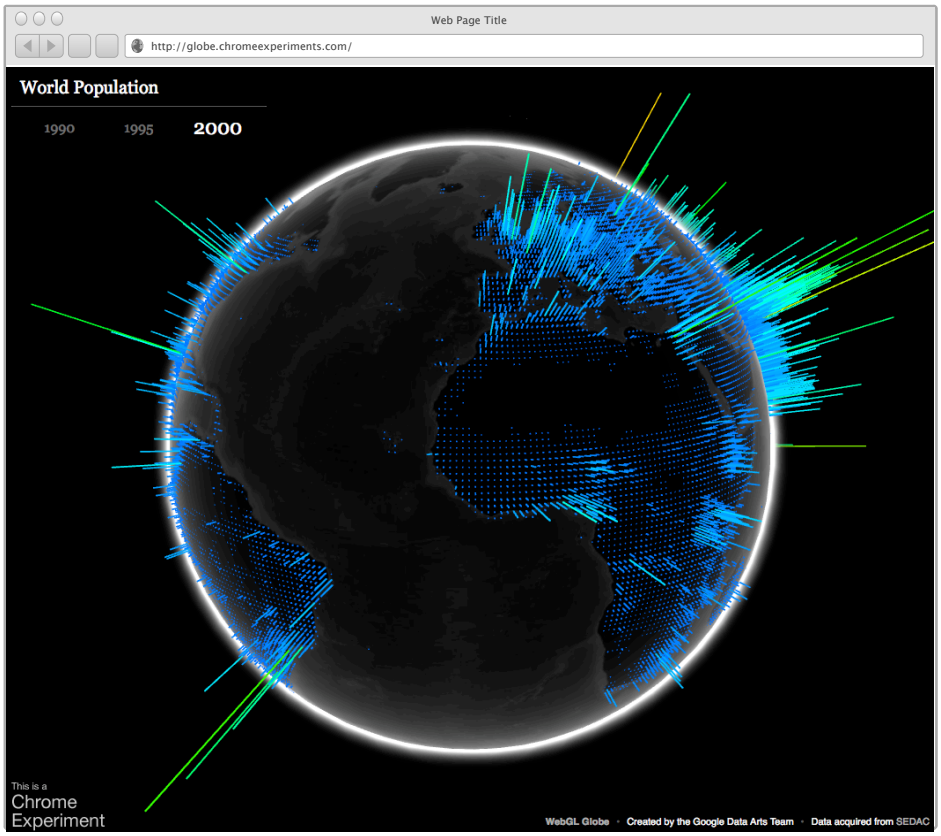


Abbildung 3.3.: DataVis - Bevölkerungspopulation
(Google-Data-Aris-Team, 2015)

Die beschriebenen Unterkategorien können auch beliebig miteinander kombiniert werden. Abbildung 3.3 illustriert eine solche Kombination. Die Weltbevölkerung wird anhand geografischer Informationen in Verbindung mit einem 3D-Barchart abgebildet. Eine solche Kombination aus den Bereichen **GeoVis** und **InfoVis** wird als **DataVis** bezeichnet (Andrews, 2015).

4. Unterstützende Elemente

Eine Visualisierung erleichtert die Interpretation von Daten. Muster oder Abweichungen von der Norm können sehr schnell ausgemacht werden. Aber was ist eine Visualisierung wert, wenn keine kontextuellen Informationen geboten werden?

4.1. Legenden

Legenden bieten kontextuelle Informationen. Elemente einer Legende referenzieren einzelne Komponenten einer Visualisierung und können Daten über die zugrunde liegenden Daten (Metadaten) beinhalten.

4.2. Achsen

Ein weiteres unterstützendes Element einer Visualisierung sind ***Achsen***, welche aus Beschriftungen, Tick-Marks und Tick-Labels bestehen.
Die *Beschriftung* von Achsen bietet Informationen über die vorliegenden Dimensionen einer Visualisierung und wie diese zu interpretieren sind. *Tick-Marks* unterteilen eine Achse in unterschiedliche Segmente. *Tick-Labels* sind Textfelder, welche Informationen zu einer gewissen Tick-Mark-Position beinhalten und auch dort dargestellt werden. Eine sinnvolle Anzahl an Ticks, sowie deren Positionierung, hängen stark von der Visualisierung ab. Zu viele Ticks „überladen“ eine Visualisierung, zu wenige erlauben keine Rückschlüsse auf dargestellte Daten, welche sich zwischen zwei Tick-Marks befinden. Auch die Frage, wo Tick-Marks positioniert werden sollten, ist nicht einfach zu klären.

Sollen zum Beispiel Tick-Marks für einen Wertebereich von 0.8 bis 2.8 gesetzt werden, würde ein Mensch wohl dazu neigen, einen Wertebereich von 0 bis 3 mit einer Schrittweite von 1 abzubilden ($[0, 1, 2, 3]$). Dies ist eine Frage der Ästhetik. Menschen sind gut darin, schöne Zahlen zu finden, Computer sind das nicht.
Die Relevanz von Achsen wird oft unterschätzt, sie sind aber ein wesentlicher Bestandteil eines jeden Plots.

Dieser Abschnitt wird sich im Weiteren mit unterschiedlichen Ansätzen zur Positionierung von Tick-Labels auf Achsen beschäftigen. Eine direkte Gegenüberstellung der folgenden Algorithmen wird abschließend im Abschnitt 4.2.4 geboten.

4.2.1. Naiver Ansatz

Ein naheliegender Ansatz ist es, ein Intervall anhand des Wertebereiches und einer gewünschten Anzahl an Schritten zu bestimmen. Ausgehend vom Minimum eines Datensatzes kann dieses Intervall so lange addiert werden, bis das Maximum des Datensatzes erreicht wird. Dies ist eine sehr triviale Lösung, welche nicht immer eine schöne Skalierung bietet. Abschließend wird eine mögliche Implementierung in Form von Pseudocode geboten.

Algorithm 1 $naiv(min, max, steps)$

Require: $max >$ min && steps > 1
  $interval \leftarrow \frac{max-min}{steps-1}$
  $result \leftarrow []$
  $tick \leftarrow min$
  while $tick \leq max$ **do**
    $result \xleftarrow{push} tick$
    $tick+ = interval$
  end while
  return $result$

4.2.2. Heckbert

Heckbert beschäftigte sich mit der Frage, welche Zahlen von einem Menschen als ästhetisch empfunden werden. Laut seinen Beobachtungen sind dies die Zahlen 1, 2 und 5 sowie deren Vielfache mit einer Zehner-Potenz (Glassner, 2013).
Für die Berechnung einer Intervallbreite, anhand welcher die Position eines Tick-Labels bestimmt wird, greift Heckbert genau auf diese Zahlen zu.

$$\mathrm{mf} * 10^{e} \tag{4.1}$$

Gegeben sind das Maximum und das Minimum eines Datensatzes. Wenn vom Maximum das Minimum subtrahiert wird, erhält man den Wertebereich (range). Dieser dient als Input und wird zur Bestimmung des Multiplikationsfaktors (*mf*) sowie des Exponenten (*e*) verwendet, sodass das Ergebnis der Formel (4.1) ein „schönes“ Intervall liefert.

Die Werte für den Exponenten *e*, sowie für den Multiplikationsfaktor *mf* werden wie folgt bestimmt:

1. Finde die am nächsten liegende ganzzahlige Zehner-Potenz zu range

```
var e = Math.floor( Math.log10( range ) );
```

2. Normalisiere

```
var factor = range / Math.pow( 10, e);
```

3. Bestimme anhand der Zahlen 1, 2, 5 und 10 den Multiplikationsfaktor.

```
var mf;
if(factor <= 1.5) mf = 1;
else if(factor <= 3) mf = 2;
else if(factor <= 7) mf = 5;
else mf = 10;
```

Werden die gefundenen Werte für e und mf in die Formel (4.1) eingesetzt, wird das gesuchte Intervall geliefert. Ausgehend vom abgerundeten Wert des Minimums wird so lange das gefundene Intervall addiert (und ein Label gesetzt), bis der Wert des aufgerundeten Maximums erreicht wird.

4.2.3. Wilkinson

Im Unterschied zu Heckbert wird bei Wilkinsons Algorithmus ein Tick-Intervall nicht berechnet, sondern gesucht (Wilkinson, 2005). Er sagt, dass „schöne" Zahlen eine unendliche Teilmenge der reellen Zahlen $\mathbb{R}$ sind und definiert eine „schöne" Skalierung R_{N} (4.2b) als geordnete Liste bestehend aus diesen Zahlen. Wilkinson erachtet eine solche Skalierung als besonders ästhetisch, wenn sie zusätzlich den Wert **Null** enthält.
Von diesem Algorithmus werden die Parameter Minimum (*min*), Maximum (*max*), sowie eine vom Benutzer bevorzugte Anzahl an Segmenten (m) entgegengenommen. Die Suche nach einer geeigneten Intervallbreite, um Tick-Labels zu positionieren, erfolgt in zwei Schritten.

Schritt eins

Im ersten Schritt (siehe Algorithmus 2) wird anhand von m eine Liste (*mrange*) erstellt. Jeder Wert in *mrange* steht für eine potentiell mögliche Anzahl an Labels. Jedes $k \in mrange$ wird mit Wilkinsons Suchalgorithmus (Algorithmus 3) getestet und schlussendlich wird das beste Ergebnis retourniert. Dieser Schritt ist notwendig, da eine vom Benutzer geforderte Anzahl an Ticks nicht unbedingt ein optimales Ergebnis liefert.

Schritt zwei

Der zweite Schritt umfasst die eigentliche Suche nach einer geeigneten Tick-Skalierung. Das Datenfeld Q (4.2a) ist eine Liste, welche bevorzugte Schrittweiten beinhaltet. Mit den gegebenen Werten (min, max, k) werden zu jedem $q \in Q$ „schöne“ Skalierungen R_{N} (4.2b) bestimmt.

$$Q = \{1, 5, 2, 2.5, 3\} \tag{4.2a}$$

$$R_{\mathrm{N}} = \{qx10^{z} : q \in Q, z \in \mathbb{Z}\} \tag{4.2b}$$

Eine mögliche Tick-Skalierung ergibt sich, indem anhand von R_{N}, neue Minima und Maxima approximiert werden. Zu jeder gefundenen Tick-Skalierung wird die Güte w (4.3) mit Hilfe der Gewichtungsfunktionen *simplicity* (4.4a), *granularity* (4.4b) und *coverage* (4.4c) bestimmt. Abschließend wird das beste Ergebnis retourniert.

$$\mathbf{w} = \frac{(s + g + c)}{3} \qquad (0 \leq w \leq 1) \tag{4.3}$$

$$simplicity : s = 1 - \frac{i}{|Q|} + \frac{v}{|Q|} \tag{4.4a}$$

$$granularity : g = \begin{cases} 1 - \frac{|k-m|}{m} & 1 < k < 6m \\ 0 & \text{otherwise} \end{cases} \tag{4.4b}$$

$$coverage : c = \frac{\Delta data}{\Delta scale} = \frac{data_{\max} - data_{\min}}{scale_{\max} - scale_{\min}} \tag{4.4c}$$

Simplicity (4.4a) bevorzugt jene Schrittweiten, die früher in der Liste Q vorkommen, zudem wird das Vorkommen des Wertes *NULL* belohnt. Der Parameter i ist ein Index für die aus Q gewählte Schrittweite, v ist ein Indikator dafür, ob eine konkrete Skalierung den Wert *NULL* enthält ($v = 1$), oder nicht ($v = 0$).

Granularity (4.4b) spiegelt die Abweichung zwischen der gewünschten Anzahl an Labels (m) und der Anzahl der vorhandenen Labels (k) wieder.

Coverage (4.4c) bewertet, wie gut eine Skalierung die gegebenen Daten abdeckt. Diese Funktion gibt das Verhältnis zwischen Datenbreite und Skallierungsbreite wieder. Eine Skalierung sollte nicht weit von den tatsächlichen Werten abweichen. Wird eine zu breite Skalierung gewählt, ergeben sich daraus große Leerräume zwischen den Labels. Für eine konkrete Implementierung empfiehlt Wilkinson einen Schwellwert einzuführen (mincoverage). Als geeignet wird 0.75 von ihm angesehen.

Algorithm 2 $Wilkinson(min, max, m)$

Require: $max >$ min && m > 1
  $mrange \leftarrow max(floor(m), 2) : ceil(6 * m)$
  for all $k \in mrange$ **do**
    $result \leftarrow Wilkinson.nice(min, max, k, m)$
    if $result$ && ($!best$ || $result.score > best.score$) **then**
      $best \leftarrow result$
    end if
  end for
  return $best$

Algorithm 3 *Wilkinson.nice*(min, max, k, m)

$Q = \{1, 5, 2, 2.5, 3\}$
$intervals \leftarrow k - 1$
$delta \leftarrow \frac{max - min}{intervals}$
$e \leftarrow floor(log_{10} delta)$
for all $q \in Q$ **do**
  $stepsize \leftarrow q * 10^e$
  $newMin \leftarrow floor(\frac{min}{stepsize}) * stepsize$
  $newMax \leftarrow newMin + intervals * stepsize$
  if $newMin \leq min$ && $newMax \geq max$ **then**
    $w \leftarrow \frac{simplicity + granularity + coverage}{3}$
    if $!best \mid\mid (w > best.w)$ **then**
      $best \leftarrow$ w, newMin, newMax, stepsize
    end if
  end if
end for
return *best*

4.2.4. Vergleich

Der vorgestellte naive Ansatz aus Abschnitt 4.2.1 ist einfach zu implementieren, hat aber den Nachteil, dass nicht immer gute Ergebnisse geliefert werden. Heckbert bindet das ästhetische Empfinden des Menschen mit ein und berechnet Skalierungen basierend auf den Zahlen 1, 2 und 5. Es wird aber nicht garantiert, dass die erstellte Skalierung den Wertebereich eines Datensatzes gut abdeckt. Der Algorithmus von Wilkinson sucht die besten Skalierungen anhand unterschiedlicher Mengen an schönen Zahlen. Außerdem werden verschiedene Anzahlen an Labels getestet. Anhand einer Gewichtungsfunktion wird das beste Ergebnis ausgewählt.
Der Multiparameter-Suchalgorithmus von Wilkinson liefert schöne Skalierungen. Im Artikel (Talbot, Lin und Hanrahan, 2010) wird gezeigt, dass dieser Algorithmus noch optimiert werden kann (xWilkinson). Die Funktionen zur Bestimmung von *simplicity*, *coverage* und *granularity* werden verfeinert. Die Funktion *legibility* wird eingeführt, welche Schriftgröße, Orientierung, Formatierung und Überschneidung einbezieht. *simplicity*, *coverage*,

granularity und *legibility* werden in der Optimierungsfunktion (w) auch unterschiedlich gewichtet.
In Tabelle 4.1 wird ein direkter Vergleich geboten. Skalierungen unterschiedlicher Inputwerte können auf `http://yettie.at/showcase/ticks` erzeugt und verglichen werden.

(min,max,steps)	Naiv	Heckbert	Wilkinson	XWilkinson
$0, 10, 3$	$0, 5, 10$	$0, 5, 10$	$0, 5, 10$	$0, 5, 10$
$0.3, 9.3, 3$	$0.3, 4.8, 9.3$	$0, 5, 10$	$0, 5, 10$	$0, 5, 10$
$-1, 12, 3$	$-1, 5.5, 12$	$-10, 0, 10, 20$	$-3, 0, 3,$ $6, 9, 12$	$-2.5, 2.5,$ $7.5, 12.5$
$8.1, 14.1, 4$	$8.1, 10.1,$ $12.1, 14.1$	$5, 10, 15$	$7.5, 10,$ $12.5, 15$	$8, 10,$ $12, 14$

Tabelle 4.1.: Naiv vs Heckbert vs Wilkinson vs XWilkison

5. Grafik im Browser

"Visualizations aren't truly visual unless they are seen" (Murray)

Visualisierungen können hilfreich dabei sein, gewonnene Erkenntnisse mit anderen Personen zu teilen. Kirk beschreibt eine Visualisierung als **Kommunikationsmittel**. Eine Nachricht wird in Form einer Visualisierung von einem Sender an einen oder mehrere Empfänger übertragen. Übermittelte Erkenntnisse führen auf der Empfängerseite zu Einsicht, Ideen zu Inspiration (Kirk, 2012).
Wenn das Ziel die Erreichung eines möglichst großen Publikums ist, ist das Web das ideale Medium.

5.1. Technologien

In diesem Abschnitt werden unterschiedliche Technologien vorgestellt, welche die Erstellung von Visualisierungen im Bereich Web ermöglichen.

5.1.1. Java

Applets sind Java-Programme, welche von einem Remote-Server heruntergeladen und im Browser des Benutzers dargestellt werden. Ausgeführt wird der Code eines Applets in einem separaten Prozess über das JRE[1] (Oracle, o.D.).
Der Vorteil von Java-Applets liegt in der hohen Performance. Ein Nachteil ist darin zu sehen, dass ein Browser-Plugin notwendig ist.

[1] JRE : Java-Runtime-Environment

JavaFX ist eine Bibliothek, welche zur Entwicklung von sogenannten Rich-Internet-Applications (RIAs) verwendet werden kann. Unter RIAs sind Webanwendungen zu verstehen, die mehr bieten als rein statische HTML-Seiten. Diese umfassen reichhaltige Interaktionsmöglichkeiten und auch Möglichkeiten zur Visualisierung im 2D- und 3D-Bereich.
JavaFX wurde als Java-API konzipiert, was auch den Zugriff auf andere Java-Bibliotheken ermöglicht. JavaFX-Code wird über ein Java-Runtime-Environment kompiliert, wodurch ein solches Programm in Desktop-Anwendungen, Browsern und Smartphones dargestellt werden kann (Oracle, 2014).

5.1.2. CSS

Cascading Style Sheets eignen sich nicht nur zur Gestaltung von Webseiten. Mit gängigen HTML-Elementen und CSS lassen sich auch einfache Grafiken erstellen. Mit Hilfe von Javascript können solche Grafiken manipuliert, Animationen erstellt, beziehungsweise eine Interaktivität mit einem Benutzer ermöglicht werden.
Ein Nachteil liegt darin, dass der Gestaltungsaufwand für komplexere Grafiken sehr groß werden kann.
CSS Level 3 ist der aktuelle Standard von CSS und umfasst unter anderem 2D- und 3D-Transformationen, Translationen, Animationen und Farbverläufe (W3C, 2015).

5.1.3. VRML

Die eigentliche Idee zu VRML ist auf Tim Berness-Lee und Dave Ragget zurückzuführen.
Bei der ersten jährlichen WWW–Konferenz (1994) in Geneva organisierten sie eine Diskussionsrunde, welche zum Ziel hatte, eine Virtual-Reality-Schnittstelle für das World Wide Web zu spezifizieren. Die Teilnehmer kamen zu dem Schluss, dass man auf bereits existierende Lösungen zurückgreifen und diese adaptieren kann.
Als Grundlage für VRML 1.0 wurde das Open-Inventor-Format von SGI[2]

[2]SGI : Silicon Graphics

gewählt.
Die eigentliche Idee, VRML als Erweiterung von HTML zu entwickeln und Hyperlinks für das interaktive Verhalten zu verwenden, wurde schlussendlich nicht weiter verfolgt und die Bedeutung des Akronyms VRML wurde von „Virtual Reality **Markup** Language“ in „Virtual Reality **Modeling** Language“ abgeändert (Graves, 1995).
1997 wurde VRML 2.0 schließlich standardisiert (VRML-Consortium, 1997).
VRML ist eine textbasierte Sprache, welche mit Hilfe eines Browser-Plugins[3] [4]gerendert wird.
Jede VRML-Datei (.wrl) beginnt mit einem Header *#VRML v2.0 utf8*, gefolgt von einem Szenengraphen. Der Szenengraph ist ein Konglomerat aus Objekten, welche eine 3D-Welt beschreiben.

```
1 #VRML V2.0 utf8
2 #A floor, i.e. a thin and large box
3 Shape {
4         geometry Box {
5                 size 20. 0.1 30.
6         }
7 }
```

Listing 5.1: Ein einfaches VRML Beispiel.
Entnommen aus (Schneider und Martin-Michiellot, 1998), Kapitel 1.2.2.

Für weitere Beispiele, sowie Tutorials wird hier auf die *VRML-Box* (Gratzer, 2007) verwiesen.

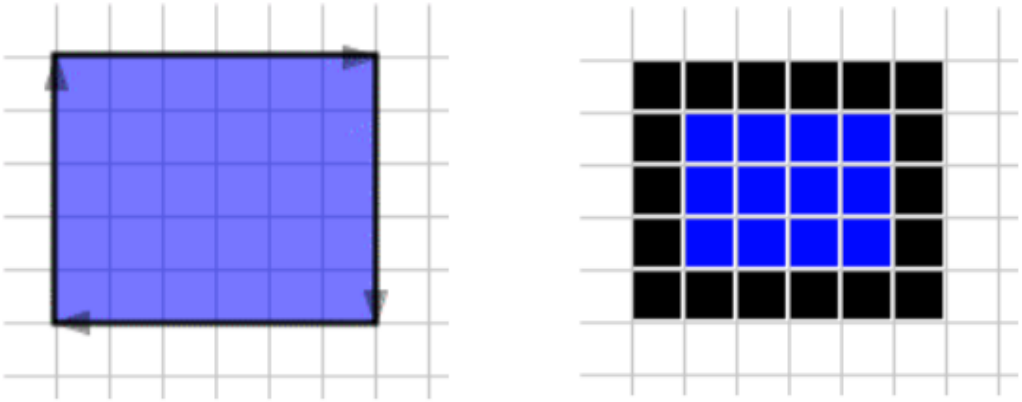
5.1.4. Flash

Adobe Flash (Adobe, 2014), früher auch als Macromedia Flash bekannt, ist eine Entwicklungsumgebung zur Erstellung multimedialer, plattformunabhängiger Inhalte. Über eine grafische Benutzeroberfläche können Elemente erzeugt werden. Vorgefertigte Effekte, Pfad-Definitionen für Objekte und eine integrierte Zeitleiste vereinfachen die Erstellung von Animationen.

[3] Cortona Player
[4] Cosmo Player

Die für Flash entwickelte Sprache ActionScript (ein ECMAScript-Dialekt) ermöglicht einen programmierten Zugriff beziehungsweise eine Manipulation von Elementen der Flash-Szene.
Erzeugt werden *.swf* Dateien, welche von einem Flash-Player geparst und dargestellt werden können.
Eingesetzt wird Flash hauptsächlich in Computer-Spielen und Werbebannern, welche in Webseiten eingebettet werden. Des Weiteren findet Flash Anwendung in Audio- und Video-Streaming-Plattformen, wie zum Beispiel Youtube, Twitch.tv und Spotify.

5.1.5. SVG



(a)Vektor-Grafik (b)Raster-Grafik

Abbildung 5.1.: Vektor-Grafik vs. Raster-Grafik
Entnommen aus (Eisenberg, 2002)

Das W3C (World Wide Web Consortium) vereinigte die beiden 1998 eingereichten Sprachen VML und GPML und entwickelte diese unter der Bezeichnung SVG (Scaleable Vector Graphics) weiter.
SVG ist ein auf XML basierendes Format zur Beschreibung von 2D-Grafiken. Unterstützt werden drei grundlegende Objekttypen:

1. Text,
2. Bild (Raster-Grafik) und
3. Form (Vektor-Grafik).

Vor allem die Möglichkeit Formen zu definieren unterscheidet SVG von vielen anderen Grafik-Formaten.
Eine Form wird durch Pfadangaben für Linien und Kurven beschrieben (siehe 5.1a). Skalierungen haben keinen negativen Effekt auf die Qualität des resultierenden Bildes, da eine optimale Darstellung anhand der zur Verfügung stehenden Beschreibung errechnet werden kann.
Im Gegensatz dazu besteht eine Rastergrafik (z.B.: jpg, png) aus einer endlichen Menge an Bildpunkten, sogenannten Pixel (siehe 5.1b). Jeder Pixel hat eine vorbestimmte Position auf einem Raster und repräsentiert einen zugeordneten Farbwert. Die Breite und die Höhe des Rasters bestimmen die Anzahl der Pixel. Rastergrafiken können zwar einfach dargestellt werden, Skalierungen können aber die Wiedergabequalität vermindern (Eisenberg, 2002).

SVG ist mehr als nur ein Grafik-Format.
Es bietet auch ein Interface (SVGDocument), über welches jedes definierte Element in der XML-Struktur manipuliert werden kann. SVGDocument ähnelt stark dem HTMLDocument Interface, welches zur Manipulation des Document Object Models (DOMs) verwendet wird. SVG kann somit in ein XHTML-Dokument eingebettet werden und direkt über das HTMLDocument Interface manipuliert werden (W3C, 2011, Kapitel 5.11).
Animationen und Interaktivitäten mit einem Benutzer lassen sich mit Hilfe einer ECMAScript-Sprache realisieren. Die Darstellung einer SVG-Grafik kann anhand von CSS (W3C, 2011, Kapitel 6.5) oder XSL (W3C, 2011, Kapitel 6.6) beeinflusst werden.

5.1.6. Canvas

Ursprünglich von Apple für WebKit entwickelt, ist Canvas heute ein fester Bestandteil von HTML.
Im Gegensatz zu SVG ist Canvas nicht textbasiert. Es werden auch keine Vektorgrafiken erstellt.
Canvas ist eine 2D Grafik-API. Unter Zuhilfenahme von Javascript können Bitmap-Grafiken dynamisch erstellt und innerhalb des Canvas-Elementes dargestellt werden (bucephalus.org, 2013).
Der Inhalt eines Canvas-Elementes erscheint wie ein gewöhnliches Bild. Änderungen der Skalierung müssen programmatisch gehandhabt werden. Außerdem wird keine besondere API zur User-Interaktion geboten. Dies ist auch nicht unbedingt notwendig, da man auf browserspezifische Event-Handler zurückgreifen kann (Moot, 2012).
Um ein Beispiel zu nennen, kann man die relativen Mauskoordinaten innerhalb eines Canvas-Elementes bestimmen, indem man von den absoluten Mauskoordinaten die Position des Canvas-Elementes subtrahiert. Die relative Position des Canvas-Elementes ist wiederum von seinen Style-Eigenschaften (top, left, margin, padding ...) beziehungsweise auch von allen Elementen, in welches es eingebettet wurde, abhängig (Kantor, 2011).
Im Vergleich zu SVG erscheint dieses Prozedere für eine User-Interaktion sehr aufwendig, aber gängige Canvas-Grafik-Frameworks übernehmen viele dieser notwendigen Schritte, wodurch sich der Aufwand auf ein Minimum reduziert (siehe Beispiel 5.2 und erzeugte Abbildung 5.2).

```
1  var stage = new Kinetic.Stage({
2   container: 'container',
3   width: 400,
4   height: 400
5  });
6
7  var layer = new Kinetic.Layer();
8  var radius = stage.getWidth()  / 4;
9
10 var circle = new Kinetic.Circle({
11  x: stage.getWidth() / 2,
12  y: stage.getHeight()/ 2,
13  radius: radius,
14  fill: 'gray',
15  stroke: 'black'
16 });
17 /* CLICK HANDLER */
18 circle.on('click', function() {
19  console.log('Type=%s Pos( %s, %s ) radius=%s',
20  this.className,this.attrs.x,this.attrs.y,
21  this.attrs.radius);
22 });
23
24 layer.add(circle);
25 stage.add(layer);
26 /*  OUTPUT - on click */
27 // Type=Circle Pos( 200, 200 ) radius=100
```

Listing 5.2: Einfacher Click-Event mit dem KineticJS Framework

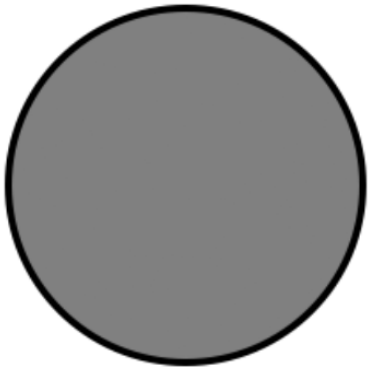


Abbildung 5.2.: Generierte Grafik aus Beispiel 5.2

5.1.7. WebGL

WebGL gilt als der neue Standard zur Erstellung von 3D-Grafiken im Bereich Web. Obwohl es nicht Teil der offiziellen HTML5 Spezifikationen ist, wird es trotzdem von den meisten aktuellen Browsern unterstützt (Deveria, 2015). Aufgrund von Sicherheitsbedenken stand Microsoft dieser neuen Technologie zunächst zwar sehr kritisch gegenüber (Microsoft, 2011), entschied sich schlussendlich aber doch, sie im Internet Explorer 11 zur Verfügung zu stellen (Welch, 2013).
Die Khronos Group entwickelte diese auf „OpenGL ES“ basierende Javascript-Grafik-API und stellte die erste Version am dritten März 2011 auf der Game Developers Conference in San Francisco vor (Khronos-Group, 2011).
Um eine Grafik mit WebGL zu rendern, sind mindestens acht Schritte notwendig (Parisi, 2012):

1. **Erzeugung eines Canvas-Elementes**:
   Wie die im Abschnitt 5.1.6 beschriebene 2D Canvas-Grafik-API wird auch WebGL über ein Canvas-Element zur Verfügung gestellt.

2. **Abfragen des WebGL Kontextes**:
   var canvas = document.getElementById(CANVAS_ID);

var gl = canvas.getContext('experimental-webgl');

3. **Initialisierung des Viewports**:
   Das Canvas-Element stellt nicht nur die API zur Verfügung, sondern ist gleichzeitig auch ein Container, in welchem die resultierende Grafik dargestellt werden kann.
   Mit dem Viewport definiert man einen Bereich innerhalb des Canvas-Elementes, in welchem gezeichnet werden soll.

4. **Erstellung von Buffer für die Vektoren**:
   Buffer sind Datenfelder, welche die Positionen von Vektoren halten.
   var buffer = gl.createBuffer();
   gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
   gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(DATA), gl.STATIC_DRAW);

5. **Erstellung von Matrizen für Transformationen**:

   - ModelView-Matrix:
     Definiert, wo sich das gezeichnete Objekt, relativ zur Kamera, in der 3D-Welt befindet.
   - Projektions-Matrix:
     Diese Matrix wird dazu verwendet, den von der Kamera abgebildeten 3D-Raum in den 2D-Raum des zuvor definierten Viewports abzubilden.

6. **Erstellung von Shader**:
   Jedes in WebGL erstellte Objekt wird über Shader gezeichnet.
   Shader sind Programme, welche in der OpenGL Shader Sprache GLSL geschrieben und direkt auf dem Grafikprozessor ausgeführt werden.
   Man unterscheidet grundsätzlich zwischen Vertex- und Fragment-Shader, welche zur Erstellung einer Grafik benötigt werden.
   Ein *Vertex-Shader* wird für jeden gegebenen Vektor ausgeführt und transformiert diesen unter Zuhilfenahme der ModelView- und Projektions-Matrix aus der 3D-Welt in den 2D-Raum des Viewports.
   Fragment-Shader (auch Pixel-Shader genannt) bestimmen die Farben der Pixel.

7. **Initialisierung der Shader mit Parameter**:
   Es muss nicht für jedes Objekt einer Szene ein eigener Shader geschrieben werden. Meist reicht es aus, einen Vertex- und einen Fragment-Shader zu erstellen und diese mit unterschiedlichen Parametern aufzurufen.

8. **Zeichnen**
   Schlussendlich wird die 3D-Szene als 2D-Grafik im Viewport dargestellt.

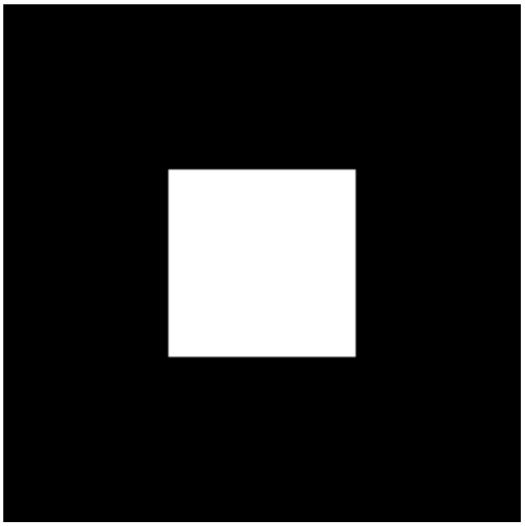


Abbildung 5.3.: Grafik erzeugt mit WebGL Programm aus Anhang A: WebGL Programm

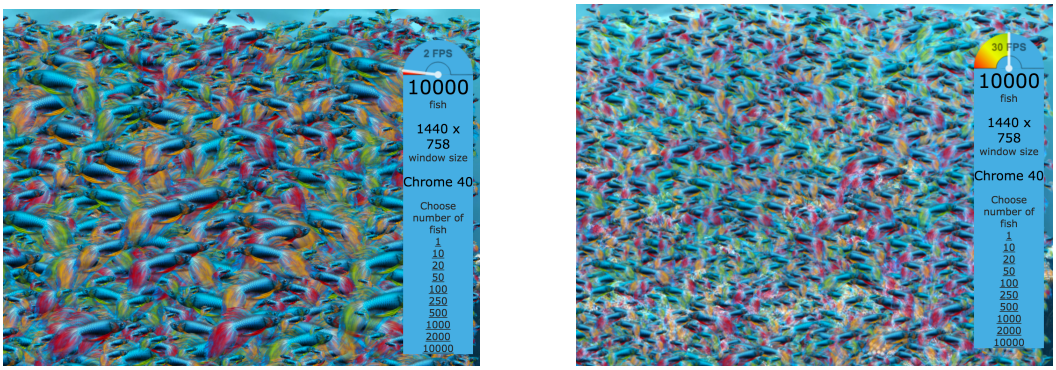
In Anhang A.1 - WebGL Programm ist der Code zur Erstellung eines simplen weißen Quadrates auf schwarzem Hintergrund beigelegt. Betrachtet man die generierte Grafik aus Abbildung 5.3 und die Länge des dazu gehörigen Programms, kommt man wohl zu dem Schluss, dass der Aufwand in keinem sinnvollen Verhältnis zum Endresultat steht. Mehr als 100 Zeilen Code zur Erstellung eines Quadrates erscheinen nicht sonderlich angemessen. Dies ist aber eine bewusste Design-Entscheidung der Entwickler.

- WebGL ist eine Low-Level API,

daher ist der Aufwand für einen Programmierer sehr hoch.
- Es wird nur ein 2D-Bild gerendert.
  Animationen werden von WebGL nicht unterstützt und müssen explizit über einen Render-Loop gehandhabt werden, über welchen die 2D-Grafik aktualisiert wird.
- Events werden nicht unterstützt,
  daher müssen Benutzereingaben, Kollisions-Detektionen und dergleichen auch explizit gehandhabt werden.

Diesen negativen Aspekten steht ein ganz wesentlicher Punkt gegenüber:

- WebGL ist sehr leistungsstark!



(a)Canvas (b)WebGL

Abbildung 5.4.: Canvas vs. WebGL

Um dies zu verdeutlichen wird hier auf die beiden Benchmarks [5] [6]von Jeff Muizelaar verwiesen.
Beide wurden vom Autor dieser Arbeit mit dem Test-System aus Tabelle 5.1 durchgeführt. Abbildung 5.4 stellt die Ergebnisse von Canvas und WebGL bei *10000* Objekten gegenüber. Eine flüssige Animation des Canvas-Benchmarks (5.4a) war nicht möglich (*2 FPS*), wohingegen der Benchmark mit WebGL(5.4b) einwandfrei mit *30 FPS* gerendert werden konnte.

[5]Benchmark-Canvas
[6]Benchmark-WebGL

	MacBook Pro
Prozessor	2,6 GHz Intel Core i7
Speicher	8 GB 1600 MHz DDR3
Grafik	NVIDIA GeForce GT 650M 1024 MB
Browser	Chrome v40

Tabelle 5.1.: Test-System

5.1.8. Schlussfolgerungen

In „*A Scalability Study of Web-Native Information Visualization*“ werden die Einsatzmöglichkeiten von Java-Applets, HTML, SVG und Canvas zur Visualisierung anhand von Parallel-Koordinaten (Wikipedia, 2015b) und dem Squarified-Treemap-Algorithmus (Bruls, Huizing und Wijk, o.D.) verglichen (Johnson, o.D.).
Getestet wurde mit unterschiedlich großen Datensätzen:

- klein: 32kB und 592kB
- mittel: 604KB und 4.3MB
- groß: 39.6MB und 80.3MB

Mit allen Technologien konnte eine Visualisierung erzeugt werden. Unterschiede ergaben sich hauptsächlich in Bezug auf die Performance.
Java schnitt bei allen Experimenten am besten ab, was darauf zurückzuführen ist, dass es in einer eigenen Laufzeitumgebung (JRM[7]) läuft, in welcher Byte-Code ausgeführt wird. Bei kleineren Datensätzen kann man SVG knapp hinter JAVA platzieren. Bei großen Datensätzen steigt auch die Komplexität des SVG Dokumentes, wodurch Canvas besser abschneidet.
Allgemein lässt sich bei allen Technologien sagen, dass Interaktivität nur für kleine bis mittelgroße Datensätze zu empfehlen ist.

Technologien wie Canvas oder SVG haben ein großes Potential, wo Applets (Java) und Plugins (VRML, Flash) nicht zur Verfügung stehen. Visualisierungen, basierend auf HTML und CSS, sind im größeren Ausmaß nicht zu empfehlen, da sie für einen solchen Aufgabenbereich auch nicht ausgelegt und optimiert sind.

[7]Java Runtime Environment

Bei großen Datensätzen eignet sich Canvas besser als SVG. Wenn es darum geht, große Datensätze interaktiv zu gestalten, sind weder Canvas noch SVG ideal.

WebGL bietet einen beinahe direkten Zugriff auf die OpenGL Hardware-Treiber, wodurch die Performance nahe an native Grafik-Applikationen reicht. Im Vergleich zu Java ist für WebGL kein Browser-Plugin notwendig. Die Entwicklung einer WebGL-Applikation ist aber alles andere als trivial. Wenn es zusätzlich darum geht, Animationen mit schönen Effekten zu erstellen oder auf Benutzereingaben zu reagieren, steigt die Komplexität eines solchen Projektes sehr rasch an.
Mittlerweile existieren bereits einige unterschiedliche Frameworks, welche die eigentliche Komplexität von WebGL verbergen und den Programmieraufwand reduzieren.
Im Kapitel 5.2 werden die verbreitetsten WebGL-Frameworks vorgestellt.

5.2. WebGL Frameworks

WebGL ist sehr leistungsstark, aber ohne die Unterstützung eines Frameworks ist die Entwicklung einer solchen Applikation sehr aufwendig. Glücklicherweise kann man mittlerweile auf unterschiedlichste Frameworks zugreifen. In den beiden Blog-Einträgen (techslides, 2013), (Kulaga und Johnston, 2015) werden umfangreiche Auflistungen an 2D- und 3D- WebGL-Frameworks geboten.
In den folgenden Abschnitten dieses Kapitels werden einige dieser Frameworks vorgestellt. Die beigefügten Codebeispiele sollen den Programmierstil der korrespondierenden Frameworks veranschaulichen. Des Weiteren erzeugt jedes Beispiel eine ähnliche Grafik, wie das im Anhang A.1 zu findende WebGL Programm, wodurch ein direkter Vergleich ermöglicht wird.

5.2.1. Three.js

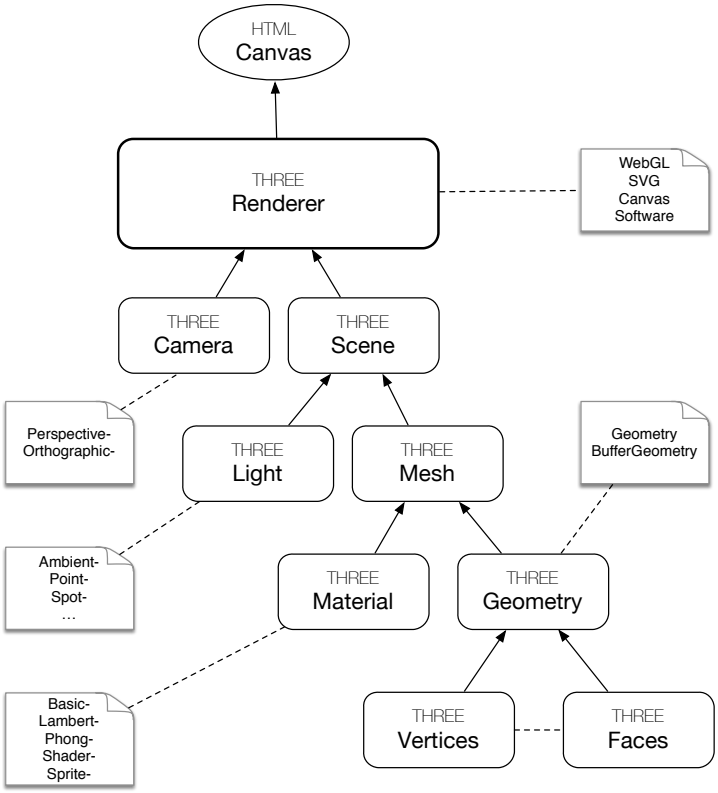


Abbildung 5.5.: Schematische Struktur eines Three.js Programmes

THREE ist eine von Ricardo Cabello Miguel entwickelte Open Source Javascript Bibliothek zur Erstellung von interaktiven 2D- und 3D- Grafiken (Miguel, 2015) .
Die Komplexität eines WebGL Programms und der daraus resultierende Aufwand werden hinter leichter verständlichen Objekten, wie Scene, Camera, Material und Geometry versteckt. Des Weiteren beinhaltet THREE eine Fülle an Werkzeugen, welche die Erstellung von 3D-Modellen, Animationen und Effekten unterstützt (Parisi, 2012). 3D-Grafiken werden mit THREE.js anhand eines Scene-Graphen bestimmt (siehe Abbildung 5.5).

Das *Scene* Objekt repräsentiert eine 3D-Welt, welche einerseits aus unterschiedlichen Lichtquellen (Ambient-, Point-, Spot-Light) und geometrischen Objekten besteht.
Meshes sind drei dimensionale Objekte einer Scene und bestehen aus Materialien und Geometrie-Objekten.
Material (Basic-, Normal-, Lambert-, Phong-Material) definiert anhand von Eigenschaften (color, opacity, blending, shading), wie die Oberflächen eines Scene-Objektes dargestellt werden.
Geometrie-Objekte bestimmen die Form eines Meshes anhand einer Liste von *Vektoren* (Vertices). *Faces* sind wiederum Flächen, welche zwischen drei (Face3) oder 4 Vektoren (Face4) aufgespannt werden.
Geometrie-Objekte sind leicht zu verstehen und zu manipulieren, haben aber den Nachteil, dass komplexe Konstrukte aus vielen Vektor- und Face-Objekten bestehen, worunter die Performance leiden kann.
Eine effizientere Alternative wird mit der *BufferGeometry* geboten, welche alle notwendigen Daten in Buffer speichert. Buffer sind Container für Typed-Arrays (MDN, 2014). Der Zugriff auf solche Datenfelder ist schneller als der Zugriff auf Listen welche Vektor-Objekte beinhalten. Des Weiteren kann ein solcher Datentyp schneller an eine GPU weitergeleitet werden.
Eine Scene kann gerendert werden, indem sie an einen *Renderer* übergeben wird. Primär kommt hier der THREE.WebGLRenderer zum Einsatz. Alternativ können auch andere Renderer, wie SVG-, Canvas-, Software-, CSS3D-Renderer, eingesetzt werden.
Schlussendlich wird anhand des *Kamera*-Objektes bestimmt, welcher Ausschnitt der 3D-Welt in einem Canvas-Element abgebildet wird.

```
1  var scene = new THREE.Scene();
2  var aspect = window.innerWidth / window.innerHeight;
3  var camera = new THREE.PerspectiveCamera( 45, aspect,
       0.1, 1000 );
4  var renderer = new THREE.WebGLRenderer();
5  renderer.setSize( window.innerWidth, window.innerHeight )
       ;
6  document.body.appendChild( renderer.domElement );
7
8  camera.position.z = 3;
9
10 var geometry = new THREE.PlaneGeometry(1, 1);
11 var material = new THREE.MeshBasicMaterial( );
12 var mesh = new THREE.Mesh( geometry, material );
13
14 scene.add( mesh );
15 renderer.render( scene, camera );
```

Listing 5.3: THREE.js Code zur Erstellung einer ähnlichen Grafik wie in Abbildung 5.3.

Im Abschnitt 5.1.7 - WebGL wurde anhand eines Beispieles (Anhang A:WebGL Programm) verdeutlicht, wie aufwendig die Erstellung von Grafiken mittels WebGL ist. Mehr als 100 Zeilen Code waren hier notwendig, um die Grafik in Abbildung 5.3 zu erstellen. Der Aufwand, eine ähnliche Grafik mit Hilfe von THREE.js zu erstellen, ist um einiges kleiner (siehe Code-Ausschnitt 5.3). Des Weiteren ist der Code im Vergleich zum reinen WebGL-Programm auch wesentlich leichter zu interpretieren.

Zu bemängeln ist die lückenhafte Dokumentation. Es werden auch wenig Informationen angeboten, wie und warum diverse Komponenten eingesetzt werden sollen. Dies ist aber kein großes Problem, da sehr viele Beispiel-Programme zu finden sind, anhand welcher man herausfinden kann, wie etwas umgesetzt werden kann. Des Weiteren existiert eine sehr große und aktive Community, welche bei Problemen sehr hilfreich ist (stackoverflow, 2015).

Aufgrund der relativ leichten Handhabung, der umfangreichen unterstützenden Werkzeuge und der hohen Performance ist THREE.js eine sehr interessante

Option, um es für die Umsetzung eines Projektes, welches sich mit Daten-Visualisierung beschäftigt, zu verwenden.

5.2.2. Scene.js

SceneJS ist ein von Lindsay Kay entwickeltes Javascript Framework zur Erstellung von sehr detaillierten 3D-Visualisierungen (Kay, 2015).
Zentraler Bestandteil ist auch hier ein Scene-Graph, welcher eine 3D-Welt anhand von sogenannten Nodes abbildet.
Nodes sind im Gegensatz zu anderen 3D-Frameworks aber keine Javascript-Objekte, welche Methoden beinhalten, beziehungsweise Methoden von anderer Stelle erben. Sie beschreiben vielmehr den Aufbau einer Scene anhand einer JSON ähnlichen Struktur (siehe Code-Ausschnitt 5.4).
Der Vorteil dieser Vorgehensweise liegt darin, dass eine für einen User einfach interpretierbare Baumstruktur entsteht. Des Weiteren können Teile dieser Struktur auch wiederverwendet werden. Die Scene wird schlussendlich kompiliert und in eine sortierte Sequenz aus WebGL-Aufrufen umgewandelt. Durch Änderungen im Scene-Graphen muss nicht die komplette Struktur neu kompiliert werden, sondern nur jener Ast, welcher modifiziert wurde.

SceneJS bietet auch Methoden zur dynamischen Modifikation von Nodes, Erstellung von Animationen, sowie Handler zur Interaktion mit einem User. Die gute Dokumentation, sowie eine umfangreiche Sammlung an Beispielen und Tutorials erleichtert den Einstieg.

```
1  SceneJS.createScene( {
2   canvas      : "myCanvas",
3   transparent: true,
4   nodes: [
5    {
6     id    : "square",
7     type  : "material",
8     color: {r: 1, g: 1, b: 1},
9     nodes: [
10     {
11      type   : "geometry/plane",
12      width  : 5,
13      height: 5
14     }
15    ]
16   }
17  ]
18 } );
```

Listing 5.4: SceneJS Code zur Erstellung eines weißen Quadrates, wie in Abbildung 5.3.

5.2.3. PhiloGL

PhiloGL ist ein von Nicolas Garcia Belmonte entwickeltes 3D-Grafik Framework. Es soll die Verwendung von WebGL vereinfachen, wodurch die Visualisierung von Daten, sowie die Entwicklung von Spielen erleichtert wird (Belmonte, 2012).
Im Unterschied zu anderen Frameworks wird hier WebGL nicht komplett abstrahiert. PhiloGL ist eher als dünne Schicht, knapp über der WebGL-API zu verstehen, welche die Verwendung durch eine Fülle von nützlichen Klassen erleichtert.
Des Weiteren beinhaltet PhiloGL Module für Effekte (*PhiloGL.FX*), Event-Handling (*PhiloGL.Event*), Erstellung von Workern (*PhiloGL.Workers*) und das asynchrone Laden von Ressourcen (*PhiloGL.IO*).
Positiv hervorzuheben ist die übersichtlich gestaltete Dokumentation. Zur

Verfügung gestellte Beispiele und Tutorials erleichtern den Einstieg.
Ein Beispielprogramm ist im Anhang A.3 beigefügt.

5.2.4. GLGE

GLGE ist zwar schon eine etwas ältere, aber doch weit verbreitete Javascript 3D-Engine.
Ziel dieses Projektes ist es, den eigentlichen Einsatz von WebGL vor einem Entwickler zu verbergen, damit dieser sich voll auf die Inhaltsgenerierung konzentrieren kann (Brunt, 2010).

Um eine Scene mit GLGE zu visualisieren, sind für gewöhnlich drei Dinge notwendig:

1. Ein Canvas Element,
2. eine XML-Datei und
3. Javascript

Via Javascript wird das Canvas Element an den GLGE.Renderer gebunden. Die XML Datei enthält eine Scene, welche geladen und dem Renderer übergeben wird.

GLGE orientiert sich stark an Konzepten von Spiele-Engines. Neben umfangreichen Werkzeugen zur User-Interaktion und Animation von Modellen ist auch eine 3D-Physik-Engine (JigLib[8]) fester Bestandteil dieses Frameworks. Der Einstieg in GLGE ist relativ schwierig, da nur eine sehr oberflächliche Dokumentation geboten wird und wenige Tutorials existieren.

[8] https://github.com/supereggbert/JigLibJS

```
1  var canvasElement = document.getElementById("canvas");
2  var doc = new GLGE.Document();
3  doc.load("scene.xml");
4  doc.onLoad = function() {
5   var renderer = new GLGE.Renderer(canvasElement);
6   var scene = new GLGE.Scene();
7   scene = doc.getElement("scene");
8   renderer.setScene(scene);
9   scene.setBackgroundColor('black');
10  renderer.render();
11 }
```

Listing 5.5: GLGE Javascript Code zum rendern der Scene aus Anhang A.2

6. Javascript

> " And we were pushing it as a little brother to Java, as a complementary language like Visual Basic was to C++ in Microsoft's language families at the time. " (**Brendan Eich**)

Javascript wurde ursprünglich unter dem Namen Mocha von Brendan Eich entwickelt und am 18. September 1995 vorgestellt. Eingesetzt wurde es erstmals vom Unternehmen Netscape als „LiveScript" in deren Netscape Navigator 2 Browser, um Webseiten dynamischer gestalten zu können.
Im Zuge einer Kooperation mit Sun Microsystems ist LiveScript in Javascript umbenannt worden.
Auch wenn die Bezeichnung Javascript eine direkte Verbindung zu Java vermuten lässt, ist dies nicht der Fall. Der Grund für diese Umbenennung wird als Marketing-Trick angesehen, um den Bekanntheitsgrad zu steigern (Young, 2010)(Chapman, o.D.). Des Weiteren sah Brendan Eich Javascript als Produkt, um mit Microsoft zu konkurrieren.

Heutzutage kann Javascript von allen gängigen Browsern interpretiert, beziehungsweise kompiliert (JIT[1]) werden (Resing, 2007). Diese beinahe umfassende Verbreitung ist auf die Standardisierung von Javascript zurückzuführen (ECMA[2]-262 , ISO[3]/IEC 16262).

6.1. Zur Popularität von Javascript

Lange Zeit waren viele Entwickler der Auffassung, dass Javascript den Status einer „Spielsprache" hat und für die Umsetzung professioneller Ap-

[1]JIT - Just In Time
[2]ECMA - European Computer Manufacturers Association
[3]ISO - International Organization for Standardization

plikationen nicht würdig ist. Diese negative Haltung beruht zum Teil auf dem ursprünglichen Einsatzgebiet dieser Sprache. Verwendung fand sie zu Beginn im Bereich der User-Interaktion innerhalb eines Webbrowsers in Form von kurzen Skriptblöcken. Heutzutage zählt Javascript zu den beliebtesten und am weitest verbreiteten Programmiersprachen der Welt und gilt als de facto Standard im Bereich Webentwicklung (TIOBE, 2015). Der Autor des Artikels „On the Increasing Popularity of Javascript“ (acunetix, 2009) sieht einen wichtigen Grund für diesen Popularitätsaufschwung darin, dass Javascript eine neue Rolle in der Unternehmens-Informatik gefunden hat. Neben der Entwicklung schöner und interaktiver Webapplikationen ist Javascript zu einem leistungsfähigen Werkzeug für Unternehmen geworden, um deren Angestellte sowie Kunden zu unterstützen. Weitere Ursachen für die steigende Beliebtheit von Javascript fasst der Autor in den folgenden Punkten zusammen:

Browserfreundlichkeit
: Javascript kann in nahezu jedem Browser ausgeführt werden. Unabhängig davon, ob dieser in einem Desktop-Computer, Notebook, Smartphone oder Fernsehgerät eingebettet wurde. Des Weiteren vereinfacht Javascript die Bedienung.

Einfach und anpassbar
: In der sich heutzutage schnell ändernden Welt gilt die Fähigkeit sich anzupassen als überaus wichtiger Erfolgsfaktor. Javascript bietet die Flexibilität, um auf diese Änderungen zu reagieren.

Serverseitiger Einsatz von JS
: Mit dem Node.js Projekt (Dahl, 2009) wurde ein Framework geschaffen, welches den Einsatz von serverseitigem Javascript auf performante Weise ermöglicht. Eine komplette Client/Server Applikation kann somit rein in Javascript umgesetzt werden.

Der Einsatz von Javascript auf der Serverseite einer C/S Architektur ist grundsätzlich nichts Neues (siehe oracle, 1998, Helma, 2005). Das Problem lag vielmehr darin, dass es keine Standardbibliotheken, beziehungsweise Schnittstellen (APIs) dafür gab und daher der Aufwand bei jedem neuen Projekt sehr hoch war. Kevin Dangoor beschrieb dieses Problem im Blogeintrag *What Server Side JavaScript needs* (Dagoor, 2015). Daraus resultierte das Projekt ServerJS, welches zum Ziel hatte, standardisierte Schnittstellen für Javascript-Bibliotheken zu definieren und dadurch die Kompatibilität zu

erhöhen. Das Projekt wurde später in CommonJS umbenannt. Ryan Dahl verwendete diese Standards in seinem Node.js Projekt.

Die Einsatzbereiche von Javascript beschränken sich nicht mehr ausschließlich auf den Bereich Webentwicklung. Durch den Wandel von einer „Special Purpose“ Sprache hin zu einer „Common Purpose“ Sprache, erschloss Javascript viele weitere Felder (siehe Adobe, 2006, Heise, 2013, TBideas, 2013).

Ein weiterer Faktor für den Erfolg von Javascript liegt in dessen Flexibilität, wodurch Projekte in unterschiedlichen Programmierstilen umgesetzt werden können. Das Erlernen der Sprache wird dadurch immens erleichtert und man ist schnell in der Lage, produktiv zu arbeiten, auch wenn die Programmierkenntnisse nur einen kleinen Teil von dem erfassen, was Javascript bieten kann.
Im Buch *Javascript Objektorientierung und Entwurfsmuster* (Harmez und Diaz, 2008) beschreiben die Autoren unterschiedliche Strukturierungsmöglichkeiten anhand des folgenden Beispiels:

Mit Hilfe von Javascript soll eine Struktur geschaffen werden, um eine Animation zu starten und zu stoppen.
Jeder der in diesem Abschnitt vorgestellten Stile hat seine Vor- und Nachteile, auf welche hier nicht näher eingegangen wird. Die folgenden Beispiele dienen rein zur Veranschaulichung der Flexibilität, welche Javascript bei der Programmierung bietet.

```
1 function startAnimation(){
2   ...
3 }
4 function stopAnimation(){
5   ...
6 }
7
8 startAnimation();
9 stopAnimation();
```

Listing 6.1: Prozeduale Struktur

Codeausschnitt 6.1 illustriert eine sehr einfache Strukturierung, welche dem Aufbau eines C Programmes sehr ähnelt.

```
1  var Animation = function(){
2   this.start = function(){
3    ...
4   };
5   this.stop = function(){
6    ...
7   };
8  };
9
10 var animation = new Animation();
11 animation.start();
12 animation.stop();
```

Listing 6.2: Objektorientierter Ansatz

Codeausschnitt 6.2 orientiert sich eher am Konzept der Objektorientierung .

```
1  var Animation = function(){
2   /* PRIVATE METHODS */
3   return {
4    start : function(){
5     // call private methods
6    },
7    stop : function(){
8     // call private methods
9    }
10  }
11 };
12
13 var animation = new Animation();
14 animation.start();
15 animation.stop();
```

Listing 6.3: Objektorientierter Ansatz mit Closure

Codeausschnitt 6.3 veranschaulicht, wie ein erfahrenerer Javascript-Programmierer unter Zuhilfenahme von Closures öffentliche von nicht öffentlichen Bereichen trennen kann.

```
1 var Animation = function(){
2  ...
3 }
4 Animation.prototype = {
5  start : function(){ ... },
6  stop : function(){ ... }
7 }
```

Listing 6.4: Prototype Kette, Version 1

```
1  Function.prototype.method = function( name, fn ){
2   this.prototype[ name ] = fn;
3   return this;
4  };
5  var Animation = function(){
6   ...
7  };
8  Animation.method( 'start', function(){
9   ...
10 }).method( 'stop', function(){
11  ...
12 })
```

Listing 6.5: Prototype Kette, Version 2

Die Codeausschnitte 6.4 und 6.5 illustrieren den Einsatz des spracheigenen Konzeptes „Prototype“ (Crockford, o.D.) zur Strukturierung.

6.2. ECMAScript und ECMA-262

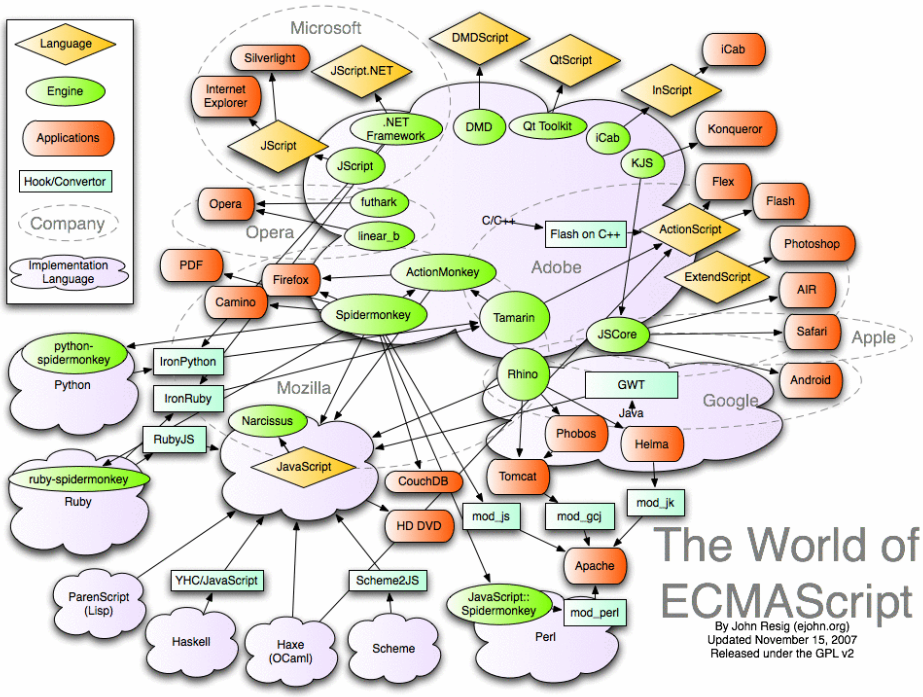


Abbildung 6.1.: The World of ECMAScript von Resing

ECMA-262 ist jener Standard, welcher die Sprache ECMAScript beschreibt. Die Entwicklung dieser Norm begann im November 1996 und beruht auf der von Brendan Eich entwickelten Sprache Mocha (LiveScipt, JavaScript). Im Juni 1997 erschien die erste Version, welche im Laufe der Zeit mehrfach überarbeitet wurde.
Die meisten ECMAScript-Interpreter, welche in den letzten 10 Jahren in Web-Browsern implementiert wurden, basieren auf der dritten Version des ECMA-262 Standards. Aktuell ist die Version 5, an Version 6 wird gearbeitet (Ecma, 2011).

Javascript ist ein eingetragener Markenname des Unternehmens Sun Microsystems (Oracle) und steht für einen Dialekt der Sprache ECMAScript. Demzufolge ist es streng genommen nicht korrekt, wenn man von der

Programmiersprache Javascript spricht, da dies nur auf eine konkrete Implementation von Netscape (Mozilla) verweisen würde.
Da sich aber der Terminus „Javascript“ durchgesetzt hat, wird diese Benennung auch in dieser Arbeit beibehalten.
Wenn nachfolgend von Javascript gesprochen wird, soll dies als Verweis auf „ECMAScript Version 5“ gesehen werden.
Eine Übersicht über die unterschiedlichen ECMAScript Dialekte, die dazugehörigen Engines, sowie von welchen Unternehmen diese umgesetzt wurden, wird in Abbildung 6.1 geboten.

Der Beginn dieses Kapitels widmete sich der Frage, was Javascript eigentlich ist. Im Weiteren wird auf die zugrunde liegenden Konzepte dieser Sprache eingegangen. Die folgenden Abschnitte beruhen zum Großteil auf dem Buch „Javascript The Definitive Guide“ von David Flanagan (Flanagan, 2011) sowie dem ECMA-262 Standard Version 5.1 (Ecma, 2011).

6.3. Das Grundkonzept

Javascript ist eine objektbasierte, lose typisierte Sprache. Unter objektbasiert ist zu verstehen, dass fast alle spracheigenen Elemente auf Objekte zurückzuführen sind. Lose typisiert bezieht sich darauf, wie bei der Definition einer Variable diese deklariert wird.

streng typisiert	lose typisiert
int $x = 1$;	**var** $x = 1$;
const double $x = 3.1415$;	**var** $x = 3.1415$;
boolean x = false;	**var** x = false;
string x = „Max Müller“;	**var** x = „Max Müller“;
Person x = new Person(„Max Müller“)	**var** x = new Person(„Max Müller“);

Tabelle 6.1.: Strenge vs lose Typisierung

Tabelle 6.2 veranschaulicht den Unterschied zwischen loser und strenger Typisierung. In Spalte 1 wird dem Interpreter/Compiler explizit mitgeteilt

(Deklaration), von welchem Typ der symbolische Name „x“(Variable) ist und welchen Wert dieser hat (Definition). Spalte 2 bildet die in Javascript typische Deklaration mittels „var“ab.

6.3.1. Variablen und ihre Typisierung

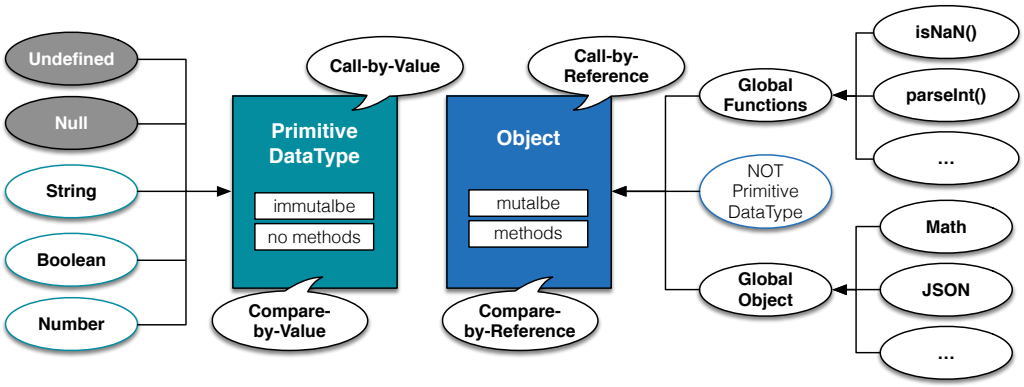


Abbildung 6.2.: Datentypen in Javascript

Die lose Typisierung mit Hilfe von *var* bedeutet aber nicht, dass Javascript Variablen nicht typisiert! Der Typ einer Variable hängt von den Daten ab, welche diese repräsentiert. In Javascript kann man Typen grob in 2 Kategorien unterteilen (Abbildung 6.2).

1. Primitive Datentypen und
2. Objekte

Zu den ***primitiven Datentypen*** zählen

- Zeichenfolgen,
- Boolesche Werte und
- Zahlen.

Zusätzlich gibt es noch die beiden speziellen Typen

- Null (mit dem Wert Null) und
- Undefined (mit dem Wert undefined).

Ein ***Objekt*** ist alles, was kein primitiver Datentyp oder spezieller Wert ist. Objekte können als Container gesehen werden, welche logisch zusammengehörige Eigenschaften (Properties) beinhalten. Attribute verweisen wiederum auf andere Objekte oder primitive Datentypen.

Neben den speziellen Typen Null und Undefined existieren auch spezielle Objekte, wie das globale Objekt. Es ist Träger von globalen Attributen (undefined, NaN, Infinity,...), globalen Funktionen (isNaN(), parseInt(), parseFloat(),...) und globalen Objekten (Math, JSON, Error,...). Auch Funktionen gehören zu den speziellen Objekten. Diese werden behandelt wie gewöhnliche Objekte, mit dem Unterschied, dass sie einen ausführbaren Code beinhalten. Wenn eine Funktion mit dem **new** Schlüsselwort initialisiert wird, bezeichnet man sie als Konstruktor.

```
1  function add10(value){
2    value +=10;
3    console.log('add10() = ',value);
4  }
5  var x = 1;
6  add10(x);
7  console.log('x = ',x);
8
9  /* OUTPUT */
10 // add10(x) = 11
11 // x = 1
```

Listing 6.6: Call-By-Value

```
1  function add10(vector2){
2    vector2.x +=10;
3    vector2.y +=10;
4    console.log('add10() = ',vector2);
5  }
6  var vec2 = { x : 0, y : 0 };
7  add10(vec2)
8  console.log('vec2 = ',vec2);
9  /* OUTPUT */
10 //add10() = Object { x=10, y=10}
11 //vec2 = Object { x=10, y=10}
```

Listing 6.7: Call-By-Reference

Wie es auch in anderen Sprachen üblich ist, kann man Funktionen Argumente übergeben. Hierbei gilt es zu beachten, dass primitive Datentypen kopiert (call-by-value 6.6) und Objekte per Referenz (call-by-reference 6.7) übergeben werden.
Ähnlich verhält es sich, wenn man primitive Typen, beziehungsweise Objekte miteinander vergleicht.
Primitive Typen werden als gleich erachtet, wenn sie gleich lang sind und wenn der Wert an jeder Stelle absolut gleich ist. Im Unterschied dazu werden Objekte anhand ihrer Referenz verglichen 6.8.

```
1  var obj1 = {x:1,y:1,z:1},
2  obj2 = {x:1,y:1,z:1},
3  obj3 = obj1;
4  console.log(obj1 === obj2);
5  console.log(obj2 === obj3);
6  console.log(obj1 === obj3);
7  /* Output */
8  // false
9  // false
10 // true
```

Listing 6.8: Vergleich von Objekten

Ein weiterer Unterschied zwischen den beiden Typen ist darin zu finden, wie sich deren Werte verändern lassen. Man spricht hierbei von „Mutable“ und „Immutable“. Der Wert eines primitiven Typs ist immer Immutable, kann also nicht verändert werden. Objekte sind veränderbar (Mutable).

```
1 var vector = Object.create({ x:1, y:2});
2 vector.z = 3;
3 vector.toString = function(){
4   return 'Vector('+this.x+', '+this.y+', '+this.z+')'
5 }
6 console.log(vector.toString());
7 //Output: Vector(1, 2, 3)
```

Listing 6.9: Mutable Object

Bei Objekten macht es durchaus Sinn, diese erweitern, beziehungsweise verändern zu können (siehe Abbildung 6.9).
Bei genauerer Betrachtung von Abbildung 6.2 bleibt noch ein letzter Unterschied zwischen den beiden grundlegenden Datentypen zu erwähnen. Im Gegensatz zu Objekten haben primitive Typen keine Methoden.

```
1 var name = 'max';
2 console.log( 'Hallo '+
3              name.charAt(0)
4              .toUpperCase()+
5              name.substring(1) );
6 /* Output */
7 // Hallo Max
```

Listing 6.10: Wrapper Objekte

Der Code–Ausschnitt in Abbildung 6.10 widerspricht gleich zwei Aussagen dieses Abschnittes. Die Variable *name* ist offensichtlich vom Typ *String* und ist demnach ein primitiver Datentyp. Sie hält den Wert max, welcher mit den Methoden *charAt()*, *toUpperCase()* und *substring()* manipuliert wird. Primitive Datentypen haben aber weder Methoden, noch können die Werte manipuliert werden. Javascript löst dieses Problem anhand von Wrapper-Objekten. Es wird ein temporäres Objekt erstellt, welches den primitiven Datentyp

umschließt und die Methoden zur Manipulation von Zeichenketten erbt. Der umschlossene Text wird nicht direkt manipuliert, sondern es wird eine neue, abgeänderte Zeichenkette zurückgegeben.

Javascript übernimmt nicht nur die Typisierung von Variablen, sondern handhabt auch die Konvertierung von einem Typ in einen anderen automatisch. Wenn die Verwendung eines arithmetischen Ausdrucks den Datentyp Zahl verlangt, wird Javascript versuchen, die gegebenen Variablen auch in Zahlen umzuwandeln. Das Ergebnis der Multiplikation der beiden Strings „2“ und „3“ vom Datentyp Zahl ergibt den Wert 6. Diese Flexibilität erscheint zwar auf den ersten Blick als Segen für jeden Programmierer, sie kann aber auch schnell zu einem Stolperstein werden.
Anhand des Codeausschnittes 6.6 soll dies verdeutlicht werden.
Die Funktion *add10* nimmt eine Variable entgegen und erhöht deren Wert um 10. In diesem Beispiel wird die Funktion mit der Variable $x = 1$ aufgerufen und das daraus resultierende Ergebnis ist korrekterweise 11. Angenommen der Wert von x wird nicht statisch definiert, sondern anhand einer Benutzereingabe über ein Input-Feld gesetzt, sieht die Rechnung innerhalb der Funktion *add10* nun folgendermaßen aus:

```
1 "1" + 10 // Result: "110"
```

In diesem Fall wird die Zahl 10 in eine Zeichenfolge umgewandelt und mit „1“ verbunden. Das Ergebnis von *add10* ist demnach nicht mehr 11 sondern „110“.
Es macht also auch in Javascript durchaus Sinn, sich mit dem Thema der Typisierung von Variablen auseinander zu setzen.

Wrapper	Typ-Konvertierung
var x = Number('1')	var x = +'1'
var x = Boolean(1)	var x = !!1

Tabelle 6.2.: Für eine explizite Typisierung kann auf die bereits erwähnten Wrapper-Objekte zurückgegriffen werden (Spalte 1). Es ist auch möglich, die internen Konvertierungsregeln von Javascript zu nutzen (Spalte 2).

6.3.2. Objekte

Javascript ist eine objektbasierte Sprache. Diese Aussage spiegelt sich in der Tatsache wieder, dass (fast) alles was kein primitiver Datentyp ist, als Objekt gesehen werden kann (siehe Bsp. 6.11). Warum ein Objekt eine Instanz von Object ist, wird im Abschnitt Vererbung näher erörtert.

```
1  var fun = function(){ // function
2  }
3  var obj = new fun; // constructor
4  console.log( fun instanceof Object );
5  console.log( obj instanceof fun );
6  console.log( obj instanceof Object );
7  // Output: 3x true
```

Listing 6.11: Nicht primitive Datentypen sind Objekte

Objekte sind Container, welche es ermöglichen, logisch zusammengehörige Elemente unter einem Kontext zu vereinen. Elemente eines Objektes bestehen aus *key : value* Paaren und werden als Eigenschaften (Properties) bezeichnet. Man kann Objekte mit Hashtables oder Maps vergleichen, da Eigenschaften unsortiert in ein Objekt eingefügt werden. Der Schlüssel (key) ist eine einzigartige Zeichenfolge und dient dazu, den Wert (value) einer Eigenschaft auszulesen oder zu setzen. Der Wert kann entweder ein primitiver Datentyp oder wiederum ein Objekt sein.

```
1 var obj = { x:0 }; // key = 'x', value = 0
2 obj.x = 1; // set value to 1 where key is 'x'
3 obj['y'] = 2; // add new <key,value> pair
4 console.log('x=%s, y=%s',obj['x'], obj.y);
5 // OUTPUT: x=1, y=2
6 delete obj.x;//delete property x
7 console.log(obj);
8 // OUTPUT: Object { y=2}
```

Listing 6.12: Setzen, lesen und löschen von Eigenschaften eines Objektes

Jede Eigenschaft (Property) eines Objektes hat zusätzlich weitere Eigenschaften (Property Attributes). Diese wurden mit der Version 5 von Ecmascript manipulierbar.

- writable:
  Dieser Bool'sche Wert ist ein Indikator dafür, ob eine Eigenschaft manipuliert werden kann.
  writable:false entspricht einem „lock".
- enumerable:
  Mit *enumerable:false* wird eine Eigenschaft nicht in einem *for-in loop* berücksichtigt.
- configurable:
  Dieser Bool'sche Wert bestimmt, ob eine Eigenschaft geändert oder gelöscht werden kann.

```
1  function log( o ){
2    for( key in o ){
3     console.log('object[%s] = %s',key, o[key])
4    }
5  }
6  // key: 'x', value: 1
7  var obj = { x:1};
8  // Object { configurable=true, enumerable=true,
9  // value=1, writable:true}
10 console.log(Object.getOwnPropertyDescriptor(obj,'x'))
11 // key: 'y', value: 2
12 Object.defineProperty(obj, 'y',{
13   value : 2,
14   writable : false,
15   enumerable : false,
16   cunfigurable : false
17 });
18 log(obj);  // OUTPUT: object[x] = 1
19 obj.y = 0;
20 console.log('y='+obj.y); // OUTPUT: y=2
21 delete obj.y; // false
```

Listing 6.13: Manipulation der Property-Attribute **writable**, **enumerable** und **configurable**

Neben den Eigenschaften haben Objekte zusätzliche Attribute:

- ***prototype*** ist eine Referenz zu einem anderen Objekt, von welchem Eigenschaften geerbt werden.
- ***class*** ist eine Zeichenfolge und dient zur weiteren Kategorisierung eines Objektes.
- ***extensible*** ist ein Indikator dafür, ob dem Objekt weitere Eigenschaften eingefügt werden können.

Alle Möglichkeiten, wie ein Objekt erzeugt werden kann, werden im Beispiel 6.14 erschöpfend aufgelistet.

```
1 var obj1 = {} // object literal
2 var obj2 = new Object; // new keyword
3 var obj3 = Object.create({}); // create function
```

Listing 6.14: Unterschiedliche Möglichkeiten zur Objekterzeugung

6.3.3. Vererbung

Vererbung ist ein grundlegendes Konzept von objektorientierten Programmiersprachen. Sie bietet eine Möglichkeit zur Strukturierung und Wiederverwendung von Code. Ausgehend von einer Basis-Klasse (Superclass) wird eine weitere (Sub-, Unter-Klasse) erzeugt, welche die Funktionalität der Basis-Klasse erweitert oder einschränkt.

Die folgenden Abschnitte 6.3.3 - Prototypbasierte Vererbung und 6.3.3 - Constructor Pattern beruhen auf (Ecma, 2011), (Flanagan, 2011), (Crockford, o.D.) und (A. M. Shah, 2013).

Prototypbasierte Vererbung

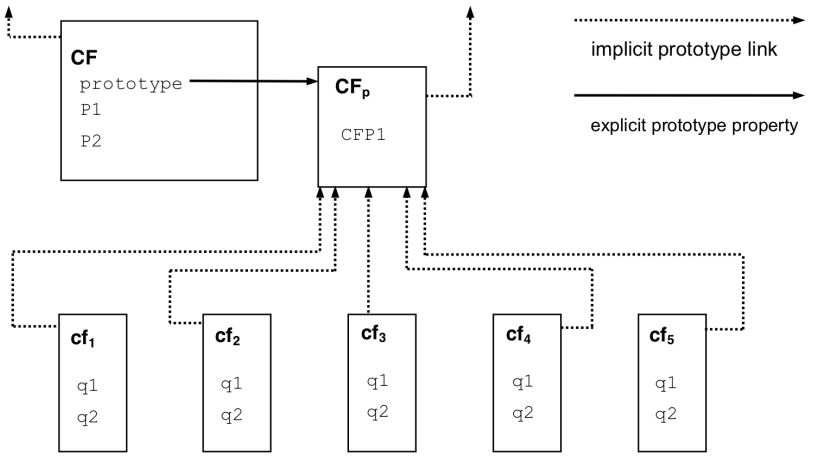


Abbildung 6.3.: Objekt - Prototype Relation aus *ECMA-262 Standard* (Ecma, 2011)
CF: Constructor Function
CF_P: Prototype-Objekt von CF
cf_1 - cf_5 Instanzen von CF
CFP1: Attribut, geteilt zwischen den Instanzen cf_1 - cf_5

Im Gegensatz zu anderen Sprachen, welche diesen klassischen Ansatz der Vererbung implementieren, verwendet Javascript „Prototype“ Objekte (siehe 6.3), um Vererbung von Informationen zu realisieren. Javascript kennt auch keine Klassen, welche als Baupläne zur Instanziierung von Objekten dienen. Es gibt (fast) nur Objekte, welche von bereits bestehenden Objekten geklont und anschließend erweitert werden. Dieser Klon-Prozess wird nicht nur explizit durch die Verwendung des new Schlüsselwortes, oder durch Aufruf der Factory-Methode Object.create() in Gang gesetzt, sondern auch implizit durch die Verwendung der Objekt-Literale { }. Auch ein vermeintlich leeres Objekt *var obj* = { } erbt Eigenschaften des Objektes „Object“.
Eine Ausnahme bilden hierbei Objekte, welche mit *Object.create(null)* erzeugt werden. Ein solches Objekt hat kein Prototype-Attribut und kann daher auch keine Eigenschaften erben.
Unter Vererbung ist nicht zu verstehen, dass Eigenschaften von einem Objekt in ein anderes kopiert werden. Vererbung „geschieht“ bei lesendem Zugriff auf ein Objekt. Der Code-Ausschnitt in Abbildung 6.15 soll dies

verdeutlichen. *obj1* und *obj2* haben jeweils ein eigenes Attribut („a“, „b“). Das Objekt *obj3* hat keine eigenen Attribute. *obj3* ist ein Klon von *obj2* und *obj2* ein Klon von *obj1*.

```
1  var obj1 = Object.create({});
2  obj1.a = 1
3  var obj2 = Object.create(obj1);
4  obj2.b = 1;
5  var obj3 = Object.create(obj2);
6  console.log(obj3.hasOwnProperty('a')); // false
7  console.log('obj3.a = '+obj3.a); //obj3.a = 1
8  obj3.a = 3;
9  console.log(obj3.hasOwnProperty('a')); // true
10 console.log('obj3.a = '+obj3.a); // obj3.a = 3
```

Listing 6.15: Prototypbasierte Vererbung bei lesendem Zugriff auf ein Objekt

Es wird zunächst geprüft, ob das gesuchte Attribut ein Attribut des aktuellen Objektes ist (own property). Falls dem nicht so ist, wird im Prototyp-Objekt des aktuellen Objektes weitergesucht (*obj3.prototype*). Wird das Attribut auch hier nicht gefunden und gibt es auch keinen impliziten Link auf ein weiteres Prototyp-Objekt, schlägt die Suche fehl. In diesem Beispiel existiert aber ein solcher Link, da *obj3*, *obj2* und *obj1* über ihre Prototyp-Objekte miteinander „verkettet“ sind und somit die Suche über *obj3.prototype.prototype* fortgeführt werden kann.
Aus diesem Grund spricht man auch von **Prototyp-Ketten** (Prototype chains).
In Zeile 8 erfolgt ein schreibender Zugriff auf das Attribut „a“. Die Prototyp-Kette wird hierbei nicht involviert, da es sich nicht um einen lesenden Zugriff handelt. Somit wird *obj3* direkt modifiziert (siehe Zeile 9 und 10).

Constructor Pattern

Die prototypbasierte Vererbung ist im Grunde zwar nicht sonderlich schwer zu verstehen, erfordert aber trotzdem ein Umdenken. Vor allem wenn man schon Erfahrungen in anderen Programmiersprachen gesammelt hat, welche einen klassischen Ansatz der Vererbung verwenden. Um dies zu erleichtern, bietet Javascript Mechanismen, mit deren Hilfe diese Sprache so verwendet werden kann, dass sie wie eine Sprache mit klassischer Vererbung erscheint.

```
1  var Object3d = function( nam ){
2    this.name = name || 'none';
3    this.id = Math.ceil(Math.random() * 100);
4  };
5  Object3d.prototype.toString = function(){
6    return 'id: '+this.id+ 'name: '+this.name;
7  };
8
9  var o = new Object3d();
10 console.log( o.toString() ); // "id: 84 name: none"
```

Listing 6.16: Konstruktor Muster

Der Code aus Beispiel 6.16 kommt einer Instanziierung eines Objektes aus einer Klasse schon sehr nahe. Funktionen sind Objekte und haben daher auch ein Prototyp-Attribut, welches auf ein Prototyp-Objekt verweist. Das Schlüsselwort „new“ermöglicht es eine Funktion als einen quasi Konstruktor einer Klasse zu verwenden. Im Grunde genommen wird aber das referenzierte Prototyp-Objekt geklont und an den **this** Pointer des neuen Objektes gehängt. Beispiel 6.17 veranschaulicht diese Prozedur anhand prototypbasierter Vererbung.

```
1  var object3d = {
2    id : Math.ceil(Math.random() * 100),
3    construct : function( name ){
4     var self = Object.create( this );
5      self.name = name || 'none';
6      return self;
7    },
8    toString : function(){
9      return 'id: '+this.id+' name: '+this.name;
10   }
11 }
12 var o = object3d.construct();
13 o.toString(); // "id: 84 name: none"
```

Listing 6.17: Nachbau des Konstruktor-Musters

Um eine Vererbung von einer „Klasse“ auf eine andere zu realisieren sind zwei weitere Schritte notwendig:

1. Umhängen des Prototyp-Objektes:
   Subclass.prototype = new Superclass;
2. Aktualisierung des Konstruktor-Attributes:
   Subclass.prototype.constructor = Subclass;
   Dieses Attribut verweist auf die Konstruktorfunktion. Da das komplette Prototyp-Objekt ausgetauscht wurde, sollte dieses Attribut aktualisiert werden. Dieser Schritt ist aber nur notwendig, wenn das Konstruktor-Attribut auch verwendet werden soll.

Der Codeausschnitt in 6.18 veranschaulicht dieses Vorgehen. *Vector2* erbt von der „Klasse“ *Object3d* aus dem Beispiel 6.16.

```
1  var Vector2d = function( x, y ){
2    Object3d.call(this, 'Vector');
3    this.x = x;
4    this.y = y;
5  }
6  Vector2d.prototype = new Object3d();
7  Vector2d.prototype.constructor = Vector2d;
8
9  var v = new Vector2d(2,4);
10 console.log(v.toString()+' x:'+v.x+' y:'+v.y); // id: 19
     name: Vector x:2 y:4
```

Listing 6.18: Vererbung anhand des Konstruktor-Musters

Eine weitere Vereinfachung und Annäherung an die klassische Vererbung bildet die Implementation einer eigenen „extend“ Funktion, welche alle notwendigen Schritte übernimmt.

```
1  var extend = function(subclass, superclass){
2    var F = function(){};
3    F.prototype = superclass.prototype;
4    subclass.prototype = new F();
5    subclass.prototype.constructor = subclass;
6    subclass.superclass = superclass.prototype;
7  }
```

Listing 6.19: Klassische Vererbung. Extend Funktion von (Harmez und Diaz, 2008)

Codeausschnitt 6.19 bildet eine solche Funktion ab. Diese Funktion übernimmt alle notwendigen Schritte und erstellt zusätzlich ein neues Attribut (**superclass**) im erbenden Objekt. Dieses neue Attribut ist in vielerlei Hinsicht praktisch. **superclass** entspricht dem aus Java bekanntem Schlüsselwort **super**, welches den Zugriff auf Elemente einer Super-Klasse ermöglicht. Das erbende Objekt kann lose an das vererbende Objekt gebunden werden. Des Weiteren ermöglicht **superclass** die Erstellung von namensgleichen Methoden in der Super- und Unterklasse. Abbildung 6.20 veranschaulicht diese Vorteile, welche die Funktion extend bietet.

```
1  var Vector2d = function( x, y ){
2     //Object3d.call(this, 'Vector');
3    Vector2d.superclass.constructor.call( this, 'Vector' );
         //loose
4    this.x = x;
5    this.y = y;
6  }
7
8  extend(Vector2d, Object3d);
9
10 Vector2d.prototype.toString = function(){
11    // use toString() from superclass
12   return Vector2d.superclass.toString.call( this ) +
13     ', x: '+this.x +
14     ' y: '+this.y;
15 }
16 var v = new Vector2d(2,4);
17 console.log(v.toString());
18 // Out: id: 22 name: Vector, x: 2 y: 4
```

Listing 6.20: Vererbung mit extend Funktion

6.4. Module

Der Einsatzbereich von Javascript geht heutzutage über kleinere Skriptblöcke, welche Teile einer Webseite steuern, weit hinaus. Javascript ist zu einer Sprache geworden, die für riesige Applikationen mit hunderttausenden Zeilen Code eingesetzt wird.
Mit der Größe einer Applikation steigt auch dessen Komplexität. Um die Übersicht zu behalten, ist es essentiell, ein System in beherrschbare kleinere Blöcke aufzuteilen, welche eine einfache Schnittstelle nach außen bieten und die eigentliche Abarbeitung eines Problems verstecken. Dies beschreibt die Denkweise der modularen Programmierung (Osmani, 2012).
Das Dilemma ist nun folgendes:

Es gibt keine Module in Javascript!

Es existieren auch keine sprach-eigenen Elemente, welche das Verstecken von Attributen eines Objektes ermöglichen, beziehungsweise als privat zu deklarieren erlauben. Rein unter diesem Aspekt erscheint Javascript als relativ schlechte Wahl, wenn man zum Ziel hat, etwas Großes zu schaffen.
Eine Lösung dieses Problems bietet die Flexibilität von Javascript selbst.
Mit sprach-eigenen Mechanismen können Konstrukte geschaffen werden, welche das Verhalten von Modulen simulieren.

6.4.1. CommonJS

CommonJS ist mehr als eine Spezifikation dafür, wie Module in Javascript strukturiert werden sollten. Ziel dieses Projektes ist es, ein gesundes Ökosystem für Javascript zu schaffen, in welchem große Projekte umgesetzt werden können (CommonJS, 2015a) .
Im Blogeintrag *What Server Side JavaScript needs* (Dagoor, 2015) beschrieb Kevin Dagoor die aktuelle Situation bei der Entwicklung von serverseitigen Javascript-Applikationen und verwies darauf, dass es keine richtigen Standards für die Entwicklung von Javascript-Bibliotheken gibt.
Seiner Meinung nach war dies nicht auf technologische Probleme zurückzuführen, sondern es fehle vielmehr eine Plattform, welche es Entwicklern ermöglicht, Probleme zu diskutieren und gemeinsam eine allgemein gültige Lösung auszuarbeiten.
Eine solche Plattform schaffte Dangoor durch die Gründung der ServerJS Gruppe, welche später in CommonJS[4] umbenannt wurde.

Ein CJS-Modul wird anhand von drei Merkmalen definiert (CommonJS, 2015c):

1. In Modulen existiert die Funktion „require“.
   Diese Funktion nimmt Modulbezeichner entgegen und retourniert die entsprechenden Module.
2. In Modulen existiert ein Objekt namens „exports“.
   Ein Modul kann dieses Objekt dazu verwenden seine Schnittstellen zur Verfügung zu stellen.
3. Das „exports“ Objekt darf **nur** zum Exportieren verwendet werden.

[4] `https://groups.google.com/forum/#!forum/commonjs`

Der Geltungsbereich eines Moduls ist durch die Datei beschränkt, in welcher es definiert wurde. Abbildung 6.21 illustriert ein solches CJS [5] Modul und wie es verwendet wird.

```
/*********************************/
/* math.js */
exports.add = function() {
    var sum = 0, i = 0, args = arguments,
    l = args.length;
    while (i < l) {
        sum += args[i++];
    }
    return sum;
};
/*********************************/
/* increment.js */
var add = require('math').add;
exports.increment = function(val) {
    return add(val, 1);
};
/*********************************/
/* program.js */
var inc = require('increment').increment;
var a = 1;
inc(a); // 2
```

Listing 6.21: Definition und Verwendung von CJS Modulen
Dieses Beispiel wurde aus der Modulspezifikation von (CommonJS, 2015c) entnommen.

[5] CJS: CommonJS

6.4.2. AMD

Wie im letzten Abschnitt bereits erwähnt, handelt es sich bei CommonJS um einen eher allgemeinen Standard, welcher zum Ziel hat ein gesundes Ökosystem für Javascript-Applikationen zu schaffen. CJS geht nicht explizit auf Einschränkungen ein, welche ein Browser einer Javascript-Applikation auferlegt. Zum Beispiel:

- Geltungsbereich für Variable (File vs. global Scope),
- Synchrones / asynchrones Laden von Modulen,
- Zugriff auf externe Dateien (I/O),
- Zugriff auf Schnittstellen des Systems, ...

Aus diesem Grund wurde eine separate Gruppe gegründet, um einen Standard für Javascript Module zu schaffen, welche in einem Browser eingesetzt werden können. Der Ursprung von AMD (Asynchronus Modul Definition) ist in den CommonJS Transport/C Spezifikationen für Module zu finden.

Der AMD Standard spezifiziert eine einzige Funktion „define“ (Correia, 2014).

define(id?, dependencies?, factory **)**

id ist die Bezeichnung des Moduls. Mit Hilfe dieser Zeichenkette kann das definierte Modul an anderer Stelle geladen werden. Das Format entspricht den CJS Formatvorgaben für Modul IDs (CommonJS, 2015b).

dependencies ist ein Array und hält String Literale.
Jedes Element steht für ein Modul (Modul-ID), welches vor der Ausführung des definierten Moduls geladen werden muss. Diese zuvor geladenen Module werden der „factory“ Funktion als Argument übergeben und können in dieser verwendet werden.
Spezielle Elemente des dependency Arrays sind „require“, „exports“ und „module“, welche entsprechend der CJS Modulstandards gehandhabt werden.

factory ist entweder ein Objekt, welches als Modul exportiert wird, oder eine Funktion, die ein Modul initialisiert. Der Return-Wert der Funktion entspricht dem exportierten Modul.

Wenn man diesen Standard für ein Projekt verwenden will, muss man diesen nicht selbst implementieren, sondern kann auf Frameworks zurückgreifen. Solche Frameworks werden in der Regel als „Loader“ (Script-Loader, Modul-Loader) bezeichnet.
Zu den bekanntesten zählen unter anderen require.js (Burke, 2015), curl.js (Hann, 2015) und yabble (Brantly, 2015).

```
1  define(
2   /* ID handled by require.js */
3   [ 'core/VLibMediator', 'jquery' ],
4   function( Mediator, $ ) {
5   var VLib = function(  ) {
6    // ...
7    var mediator = new Mediator();
8    mediator.registerModule(this);
9    // ...
10  };
11  return VLib;
12 });
```

Listing 6.22: Definition eines require.js Moduls

6.4.3. Harmony

Harmonoy ist der Codename der sechsten Version von ECMASript, an welcher während der Entstehung dieser Masterarbeit noch gearbeitet wurde. Im Laufe des Jahres 2015 soll diese Version offiziell erscheinen (Mozilla, 2015).
Das Bedürfnis nach besseren Strukturierungsmöglichkeiten ist auch den Entwicklern des ECMAScript Standards nicht entgangen. Version 6 soll daher unter anderem auch Module unterstützen.
Dieser Abschnitt basiert auf dem Artikel *DRAFT Modules for JavaScript*

Simple, Compilable, and Dynamic Libraries on the Web (Herman und Tobin-hochstadt, o.D.), in welchem die Autoren ein Konzept für ein natives Modulsystem präsentieren.

David Herman und Sam Tobin-Hochstadt fassen die Nachteile der gängigen Modulsysteme in folgenden Punkten zusammen:

- Module sind Muster und kein Teil der Sprache.
  Für Programmierer ist daher der Aufwand sehr hoch, Module richtig zu implementieren, beziehungsweise diese Muster richtig zu interpretieren.
- Module verringern zwar die Anzahl an globalen Variablen, eliminieren dieses Problem aber nicht.
- Ein vorgegebenes Muster kann auch schnell falsch verwendet werden, wodurch der eigentliche Sinn dieser zusätzlichen Struktur verloren geht.
- Für Compiler ist es schwierig, Modulmuster zu optimieren.

Weiters beschreiben die Autoren ihr Konzept und welche Kriterien für die Umsetzung berücksichtigt wurden.

- Ein Modulsystem sollte es erlauben, ein Programm beliebig aufzuteilen.
  Eine rekursive Auflösung von Abhängigkeiten sollte daher unterstützt werden.
- Statische, sowie dynamische Auflösung von Variablen sollte unterstützt werden.
  Exports und Imports von Modulen sind eine statische Angelegenheit, welche während der Kompilierung des Programms gehandhabt werden können.
  Andererseits ist das Web eine sehr dynamische Umgebung, in welcher Code auch jederzeit nachgeladen werden kann.
- Globale Variablen müssen vermieden werden,
  da man keine Annahmen darüber treffen kann, in welcher Umgebung das System verwendet wird. Trotzdem müssen Module irgendwie auffindbar werden.

- Module sollten auch über externe Plattformen geladen werden können. Einerseits sollte dies über URLs möglich sein, andererseits sollten Module, welche in einer Serverumgebung eingesetzt werden, auch auf das lokale Dateisystem zugreifen können.

```
module Numbers {
 export module Even {
  import Odd.odd;
  export function even(x){
   return x == 0 || odd(x -1);
  }
 }
 export module Odd {
  import Even.even;
  export function odd(x){
   return x !== 0 && even(x-1);
  }
 }
}
```

Listing 6.23: Rekursives Modul
Code aus (Herman und Tobin-hochstadt, o.D.)

```
// loading module from external source
module JSON = require 'http://json.org/modules/json2.js';
// dynamic loading from external source
loader.load('http://json.org/json2.js',
    function(JSON) {
        alert(JSON.stringify([0,{ 1 : 1}]));
 });
// loading standard modules
module stdlib = require '@std';
module Lexer = require 'compiler/Lexer';
```

Listing 6.24: Laden von externen Modulen
Code-Beispiele aus (Herman und Tobin-hochstadt, o.D.)

6.4.4. Zusammenfassung

Module existieren in Javascript zwar nicht, können aber durch Verwendung von Mustern simuliert werden. Standardisierte Muster definieren die beiden Gruppen CommonJS und AMD.
Der Geltungsbereich eines CJS Moduls ist die Datei, in welcher es definiert wurde. Module kennen die Funktion **require**, welche zum Laden anderer Module verwendet wird und das Objekt **exports**, über welches APIs des Moduls zur Verfügung gestellt werden können.
AMD geht aus dem CJS Modul Standard hervor. Wie in CJS beschrieben, kennt ein Modul auch hier die Funktion require, sowie das Objekt exports. Des Weiteren wird die Funktion **define** definiert, welche ein Modul umschließt.
CommonJS ist eher für Server-Anwendungen geeignet, da dieser Standard nicht auf die Einschränkungen eines Browsers eingeht. Im Gegensatz dazu ist der AMD Standard für Javascript Module im Browser ausgelegt.
Mit ECMAScript 6 wird voraussichtlich ein natives Modulsystem zur Verfügung gestellt.

Für eine ausführlichere Beschreibung, sowie Gegenüberstellung der einzelnen Standards wird hier auf das Kapitel „Modern Modular Javascript Design Patterns“ in (Osmani, 2012) verwiesen.

7. {v}Plot.js

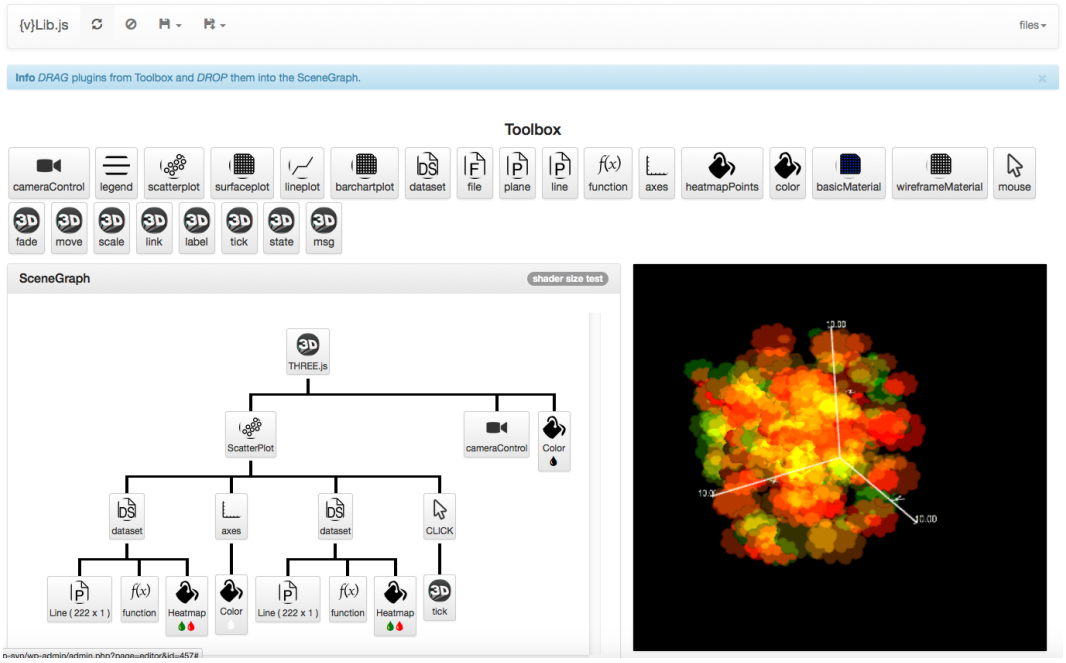


Abbildung 7.1.: Module, Plugins und Templates

{v}Plot ist ein web-basiertes Framework zur Datenvisualisierung. Es wurde komplett in der Sprache Javascript umgesetzt und kommt ohne weitere serverseitige Verarbeitung aus. Der Vorteil von Javascript liegt darin, dass das Framework in jede beliebige HTML-Seite eingebunden und plattformunabhängig in einem Browser ausgeführt werden kann. Die Server-Auslastung wird dadurch verringert, da die komplette Rechenlast auf die Client–Maschine übertragen wird.

Im Vergleich zu anderen Sprachen wie Flash oder VRML, bietet die Verwendung von Javascript zusätzlich den Vorteil, dass zur Darstellung kein weiteres Plugin notwendig ist.

Anhand von Abbildung 7.1 sollen die unterschiedlichen Komponenten veranschaulicht werden und wie diese miteinander interagieren. „Komponenten“ werden anhand ihres Aufgabenbereiches in Module, Plugins und Templates unterteilt. Eine ausführliche Beschreibung dieser Elemente wird im Abschnitt 7.1 geboten.

Die Bereiche Navigation, ToolBox, SceneGraph und Plot werden von Modulen erzeugt und verwaltet. Die Toolbox enthält eine Liste der zur Verfügung stehenden Plugins. Diese können via drag&drop von Toolbox nach SceneGraph gezogen werden. Anhand dieser Vorgehensweise werden sogenannte Templates erstellt, welche im SceneGraph in Form einer Baumstruktur dargestellt werden. Wird ein solcher Knoten mit der Maus angeklickt, öffnet sich ein Dialog-Menü, über welches das korrespondierende Plugin konfiguriert werden kann. Aktualisierte Templates werden vom SceneGraph-Modul an das Plot-Modul weitergereicht, wo diese rekursiv geparst und schlussendlich mit Hilfe von THREE.js grafisch dargestellt werden.

Neben der Anpassung eines Plots an die persönlichen Bedürfnisse ist es auch wichtig, dass die zugrundeliegenden Daten auf möglichst einfachem Weg in das System eingespeist werden können. Dies geschieht wiederum über Plugins. Daten können manuell eingegeben, automatisch generiert oder über externe Dateien eingebunden werden. Hierbei muss ein Datensatz nicht an das System angepasst werden. Das entsprechende Plugin kann so konfiguriert werden, dass Werte richtig interpretiert werden. Zusätzlich kann jeder Datensatz über Funktions-Plugins weiter verändert werden.

Der Prozess der Wissensgenerierung anhand gegebener Daten wird von Animationen und interaktiven Elementen unterstützt.

Die Verwendung des Frameworks setzt prinzipiell keinerlei Kenntnisse im Bereich Programmierung voraus. Folgende Zielgruppen werden angesprochen:

Aktive Benutzer
: Es handelt sich hierbei um Personen, welche Daten grafisch aufbereiten wollen. Sie verwenden den vollen Umfang, welchen das Framework an Modulen und Plugins zur Verfügung stellt.

Passive Benutzer
: Diese Gruppe von Usern ist rein daran interessiert, sich über einen Sachverhalt zu informieren. Sie konsumieren passiv das Werk eines aktiven Benutzers. In der Regel ist hierbei lediglich das Plot-Modul involviert, sowie Plugins, welche zur Darstellung einer Visualisierung notwendig sind.

Entwickler
: Entwickler verbessern das Framework und erweitern dessen Funktionalität. Außerdem verbessern sie die Einsatzmöglichkeiten, indem sie das Framework in bestehende Content Management Systeme integrieren.

7.1. Architektur

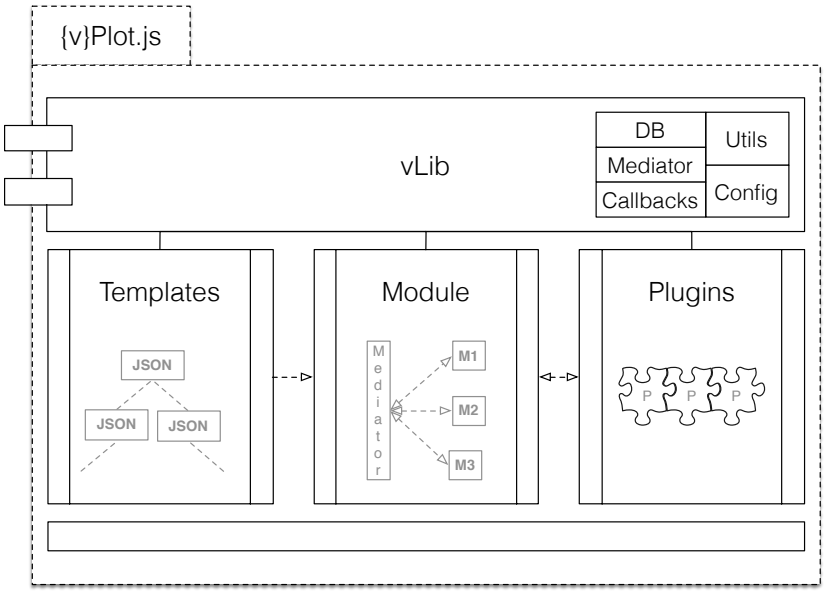


Abbildung 7.2.: Schematischer Aufbau des *{v}Plot.js* Frameworks

Große und komplexe Javascript-Applikationen sind in der Regel nicht leicht zu handhaben. Behält man bei der Planung Aspekte wie Erweiterbarkeit, Wartbarkeit oder Performance nicht stets im Auge, kann dies schnell zu Problemen führen.

Unter diesen Gesichtspunkten wurde das Framework als lose gekoppeltes Mehrkomponentensystem entwickelt. Jede einzelne Komponente des Systems ist für die Erledigung einer dieser Teilaufgaben verantwortlich.
Um die Austauschbarkeit einzelner Teile zu gewährleisten, ist eine strenge Kapselung unumgänglich. Daher gilt der Vorsatz, dass Komponenten nicht auf direktem Weg miteinander interagieren dürfen. Informationen können über einen Mediator ausgetauscht werden. Zur Verwaltung von Dateien, Frameworks und Bibliotheken kommt das AMD-Framework require.js zum Einsatz.

In diesem Kapitel wurden Teile des Frameworks vereinfacht als Komponenten bezeichnet. Betrachtet man deren Aufgabenbereiche genauer, kann man sie grob in zwei Klassen unterteilen. Umfangreiche Komponenten mit einem allgemeinen Aufgabenbereich, welche schon fast als eigenständige Programme innerhalb des Frameworks erscheinen, und spezielle Komponenten, welche für die Handhabung von Daten verantwortlich sind.
Um die Architektur des Frameworks übersichtlich zu gestalten, wurden die Komponenten auch dahingehend logisch voneinander getrennt und unterschiedlich bezeichnet.
Wenn im Kontext des entwickelten Frameworks von *Modulen* gesprochen wird sind jene Teile mit einem „allgemeinen“ Aufgabenbereich gemeint. Komponenten mit Bezug zur grafischen Darstellung und der Aufbereitung von Daten werden im Weiteren als *Plugins* bezeichnet.

Abbildung 7.2 illustriert die tragenden Elemente des Frameworks.
Daraus geht hervor, dass neben Modulen und Plugins, *Templates* einen Weiteren wichtigen Bestandteil des Frameworks bilden. Templates sind wiederverwendbare Vorlagen und bestimmen, welche Daten und vor allem auch welche Plugins für die Darstellung verwendet werden sollen.
Sie werden von Modulen erzeugt und verwaltet.
Die Struktur von Templates wird in 7.1.2 genauer erörtert. In den Abschnitten 7.1.3 und 7.1.4 werden implementierte Module und Plugins genauer beschrieben.

7.1.1. Zentrale Schnittstelle

Wie bereits erwähnt, ist vPlot.js ein sehr lose gekoppeltes System. Der Vorsatz, dass Module autark arbeiten und nichts von ihrer Umgebung wissen sollen, bietet zwar den Vorteil, dass durch diese Vorgehensweise das Framework sicher, einfach und beliebig erweitert werden kann, bringt aber auch gewisse Nachteile mit sich.
Das Problem ist offensichtlich.
Neben der strengen Kapselung nach außen, soll es Modulen dennoch möglich sein, mit anderen zu kommunizieren. Außerdem müssen Daten auch von irgendeiner Stelle zu einem Modul gelangen. Wie soll eine Interaktion vonstatten gehen, wenn ein Modul nichts kennt, außer sich selbst?
Eine Lösung wurde mit dem Publish-Subscribe Pattern gefunden, welches in Form eines Mediators implementiert wurde.

Mediator

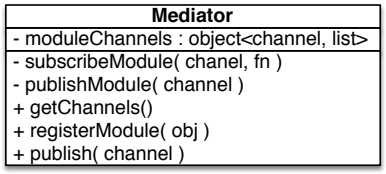


Abbildung 7.3.: core/VLibMediator

Ein Mediator gehört zur Gruppe der Verhaltensmuster und steuert den Informationsfluss zwischen Objekten. Über die öffentliche Methode *Mediator.registerModule(obj)* wird ein gegebenes Modul um die beiden Methoden *publish* und *subscribe* erweitert. Der Aufruf von *subscribe(CHANNEL, fn)* führt zu einem Eintrag in der *moduleChannels*-Liste des Mediators. Soll nun ein Modul durch Aufruf der Methode *Module.publish(CHANNEL, args)* eine Nachricht senden, wird die Mediator-Methode *publishModule* involviert. Anhand der *modulChannels*-Liste wird jede registrierte Callback-Funktion

mit den Argumenten *args* aufgerufen und somit die Nachricht an alle Interessenten des gegebenen Kanals zugestellt.

Kern Komponente - vLib

```
1  var VLib = (function( config ) {
2   // [ SNIPPET ]  Program logic, private variables and
       methods
3   var requestPluginsHandle = function( obj ){
4    // ...
5    this.publish( Config.CHANNEL_RESPOND_PLUGINS, data );
6   }
7
8   var VMediator = require('core/VLibMediator');
9   var mediator = new VMediator();
10  mediator.registerModule( this );
11
12  this.subscribe( Config.CHANNEL_REQUEST_PLUGINS,
       requestPluginsHandle );
13  // [ SNIPPET ] ... Subscriptions
14  bootstrapPlugins.apply(this,[ registerPlugin ]);
15  /* FACADE */
16  return {
17   getChannels     : mediator.getChannels(),
18   registerModule    : mediator.registerModule,
19   registerPlugin     : registerPlugin,
20   publish      :  mediator.publish,
21   setSaveTemplateCallback   :  setSaveTemplateCallback,
22   // [ SNIPPET ] public methods
23  };
24 });
```

Listing 7.1: Struktur der Klasse VLib

Der Codeausschnitt in Auflistung 7.1 illustriert die grundlegende Struktur der Klasse VLib und soll zum Verständnis der Architektur beitragen.
VLib wurde als zentrale Schnittstelle des Frameworks konzipiert und hält

eine Referenz auf den Mediator (Zeile 9).
Neben der Inter-Modul-Kommunikation ist eine Schnittstelle nach „außen“ wichtig, um Daten und Informationen in das System einzubinden beziehungsweise diese auch wieder abzufragen. Um diese Interaktion mit der „externen“ Welt zu ermöglichen, bietet diese Klasse ein öffentliches Interface, welches in Form einer Facade (Zeile 16) umgesetzt wurde.
Diese Informationen müssen auch an Module weitergeleitet, oder von diesen abgefragt werden können. Aus diesem Grund registriert sich VLib selbst beim Mediator (Zeile 10) und fungiert im Weiteren als Pseudo-Modul. Im engeren Sinn kann VLib nicht als Modul betrachtet werden, da es durch das öffentliche Interface den Vorsatz der strengen Kapselung verletzt. Durch die Registrierung beim Mediator wird aber die Funktionalität um die beiden Methoden publish und subscribe (Zeile 5 und 12) erweitert, wodurch die Kommunikation zu Modulen ermöglicht wird.

7.1.2. Templates

Templates sind JSON Objekte (Ecma, 2013), welche Informationen zu einer Visualisierung beinhalten. Sie halten einerseits beschreibende Elemente, wie *name*, *description*, *timestamp*, oder *thumbnail*. Andererseits auch Elemente, welche essentiell für die Erstellung einer Visualisierung sind. Das *SceneGraph*-Attribut ist ein solches Element. Es ist eine abstrakte, nicht sichtbare Repräsentation einer Grafik. *SceneGraph* ist ein Konglomerat aus Objekten, welche konkrete Plugins repräsentieren (FlyWeights). Anhand von Regeln werden diese in einer Baumstruktur zusammengefügt. Es ist nicht notwendig, dass sich ein Benutzer über die Existenz dieser Struktur bewusst ist, da der *SceneGraph* anhand von Aktionen eines Benutzers automatisch generiert wird.

7.1.3. Module

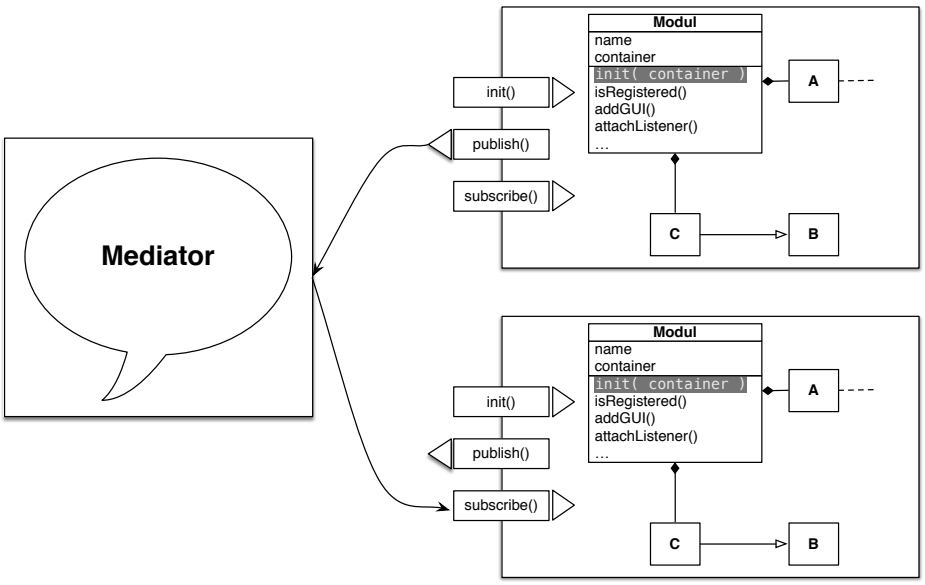


Abbildung 7.4.: Module - Kommunikationsfähige Programme im Programm

Module sind autark arbeitende, streng gekapselte Subsysteme innerhalb des {*v*}*Plot.js* Frameworks, welche für einen bestimmten Aufgabenbereich zuständig sind.
Die einzige vorgesehene Schnittstelle nach außen bildet die Methode *init(container)*, über welche dem Modul mitgeteilt werden kann, dass es mit seiner Arbeit beginnen soll.
Im Fall, dass ein Modul DOM-Elemente handhaben soll (z.B.: GUI), kann mittels des optionalen Parameters *„container“* der Methode *init* ein HTML-Element, beziehungsweise ein Identifikator eines HTML-Elementes, übergeben werden.
Falls es erforderlich ist, dass Ereignisse beziehungsweise Informationen von anderen Modulen verarbeitet werden sollen, kann die Modulschnittstelle durch Registrierung bei der zentralen Schnittstelle des Frameworks (siehe Abschnitt 7.1.5) erweitert werden. Dies bietet zwar keinen direkten Zugang zu anderen Modulen, es ermöglicht aber auf Nachrichten zu reagieren und

auch Informationen zu veröffentlichen.
Die Methode *isRegistered()* prüft, ob eine Registrierung durchgeführt wurde. Falls dem so ist, wird *attachListener()* aufgerufen, in welcher Mediator-Kanäle abonniert werden können (*subscribe*). Versendet werden Nachrichten über die Methode *publish*,

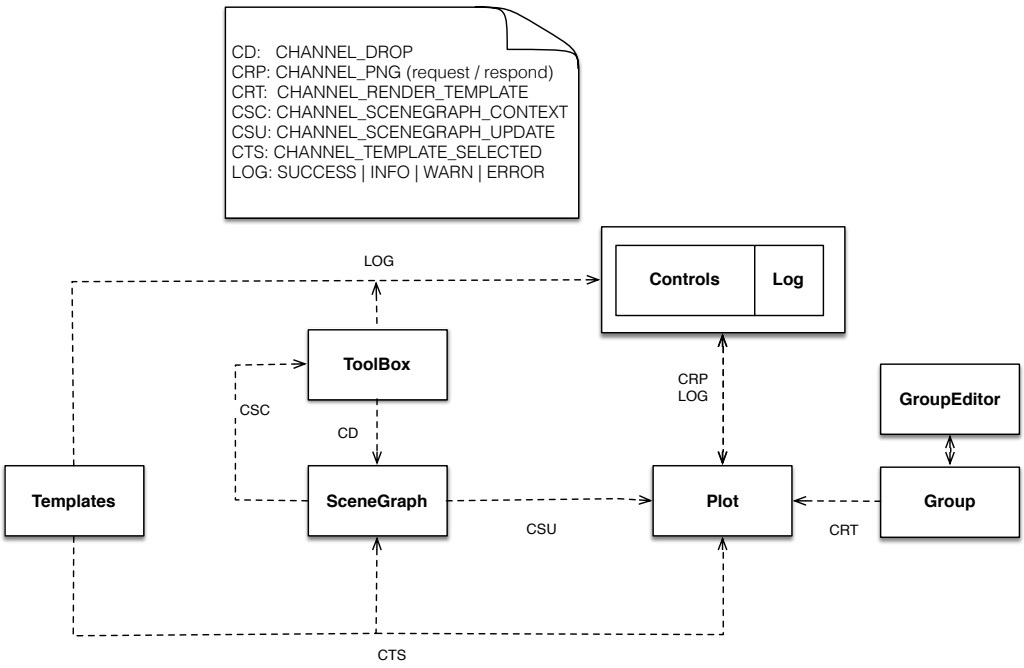


Abbildung 7.5.: Inter-Modul Kommunikation: Vereinfachte und unvollständige (7/41) Darstellung

Abbildung 7.5 veranschaulicht einen kleinen Teil der Modul-Kommunikation und soll bei der folgenden Vorstellung der implementierten Module zum Verständnis beitragen.
Insgesamt existieren *41* Kommunikations-Kanäle, welche in der Datei *config.vlib.js* definiert wurden.

Templates hält eine Liste von registrierten Template-Dateien, welche geladen, oder gelöscht werden können. Ein Template wird geladen, indem es von einem Objekt umschlossen und über den Mediator-Kanal *CHANNEL_TEMPLATE_SELECTED* veröffentlicht wird. SceneGraph (siehe Abschnitt 7.1.3 und Plot (Abschnitt 7.1.3) sind Abonnenten dieses Kanals.
Das SceneGraph Modul stellt das Template in Form einer Baumstruktur dar, Plot erzeugt anhand dieser Datei Plugins und stellt für die schlussendliche Visualisierung ein HTML-Container Element zur Verfügung.

SceneGraph verwaltet eine Baumstruktur. Jeder Knoten repräsentiert ein Plugin.
Durch einen Mausklick wird ein Dialogfeld geöffnet und ein HTML-Formular zur Konfiguration des entsprechenden Plugins eingebunden. Methoden zum Setzen von bereits bekannten, beziehungsweise zum Lesen gesetzter Werte werden über Callbacks der korrespondierenden Plugins realisiert.
Jede Aktualisierung der Struktur wird anhand eines dafür vorgesehen Mediator-Kanals veröffentlicht.

ToolBox hält eine Liste der registrierten Plugins und stellt diese über ein User-Interface grafisch dar. Plugin-Flyweights können via drag & drop auf Elemente der SceneGraph-Baumstruktur gezogen werden. Dies ist ein zentraler Bestandteil in der Prozesskette zur Generierung von Templates, anhand welcher Plugins instanziiert und aufgerufen werden.

Plot ist für die Darstellung von Daten anhand eines gegebenen Templates verantwortlich.
Plugins werden in einer Reihenfolge entsprechend der Baumstruktur des Templates instanziiert und exekutiert. Dabei ist das Teilergebnis eines Plugins gleichzeitig der Input eines anderen Plugins, welches sich in der Baumstruktur eine Ebene näher am Wurzel-Knoten befindet. Die resultierende Visualisierung wird vom Plot-Modul in ein DOM-Element eingebettet und wird somit für einen Benutzer sichtbar.

Abbildung 7.6.: Plot Modul - Navigations - Elemente

Dieses Modul ermöglicht Interaktionen mit einem Benutzer. Einerseits indirekt über Plugins, welche auf Maus-Events reagieren, andererseits über ein Navigations-Element (siehe Abbildung 7.6).
Die Navigation stellt folgende Aktionen zur Verfügung:

- Information:
  Blendet eine Informationsebene ein oder aus. Nachrichten von Plugins können hier dargestellt werden.
- Aktualisieren:
  Das aktuelle Template wird geparst und neu gerendert.
- Download:
  Anhand der Visualisierung wird eine Grafik im PNG Format erzeugt und zum Download über ein Dialog-Fenster zur Verfügung gestellt.
- Vollbild:
  Unter Zuhilfenahme der HTML5-Fullscreen-API wird eine Vollbilddarstellung der Visualisierung erzeugt. Durch die Änderungen der Canvas-Dimensionen muss der Render-Prozess aber nicht neu in Gang gesetzt werden.

Controls stellt eine grafische Steuereinheit für einen Benutzer zur Verfügung. Dieses Modul ermöglicht das Speichern von Templates, das Laden und Löschen von Datensätzen sowie die Erstellung von downloadbaren PNG-, und Template-Dateien.
Controls ist auch Empfänger von Log-Nachrichten und stellt diese grafisch dar.

Group erlaubt die strukturierte Auflistung von mehreren Templates.
Man kann diese Darstellung mit verschachtelten Ordnern, welche Dateien enthalten, vergleichen.

GroupEditor

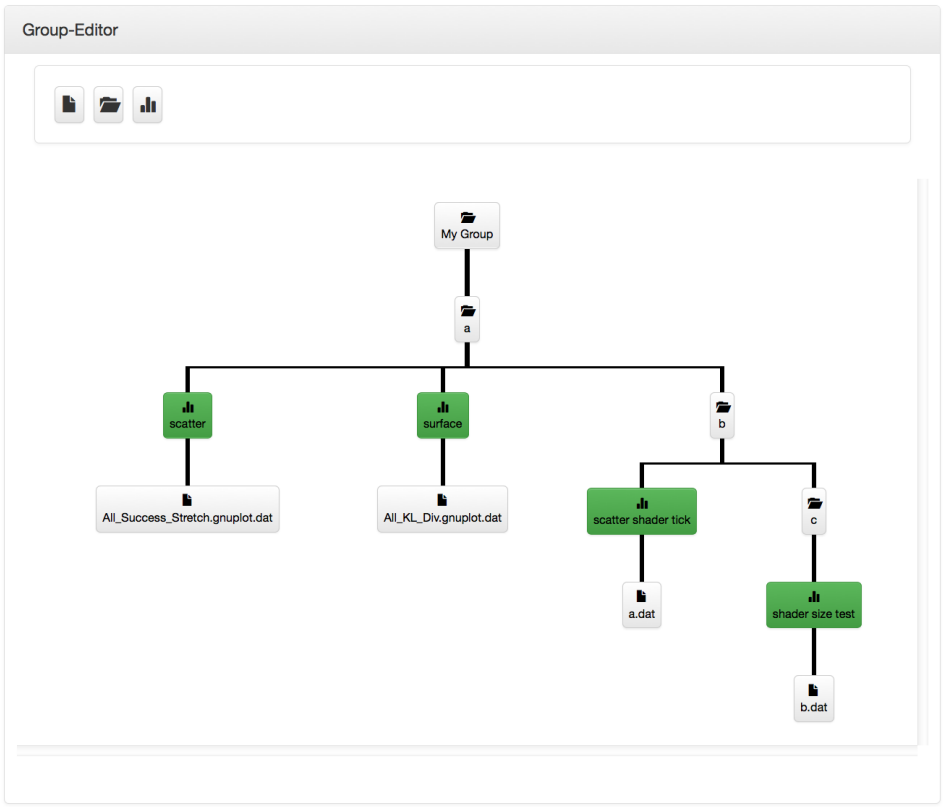


Abbildung 7.7.: GroupEditor - User Interface

Dieses Modul stellt eine Gruppe von Templates (siehe Abschnitt 7.1.3) anhand einer Baumstruktur grafisch dar.
Elemente zur Einbindung von Ordnern, Datensätzen und Templates werden zur Verfügung gestellt. Ein neuer Ordner kann als Nachfolge-Knoten eines bereits eingefügten „Ordners “, via drag&drop angehängt werden. Dies gilt auch für Templates. Diese können wiederum einen oder mehrere Datensätze als Nachfolger haben (siehe Abbildung 7.7).

7.1.4. Plugins

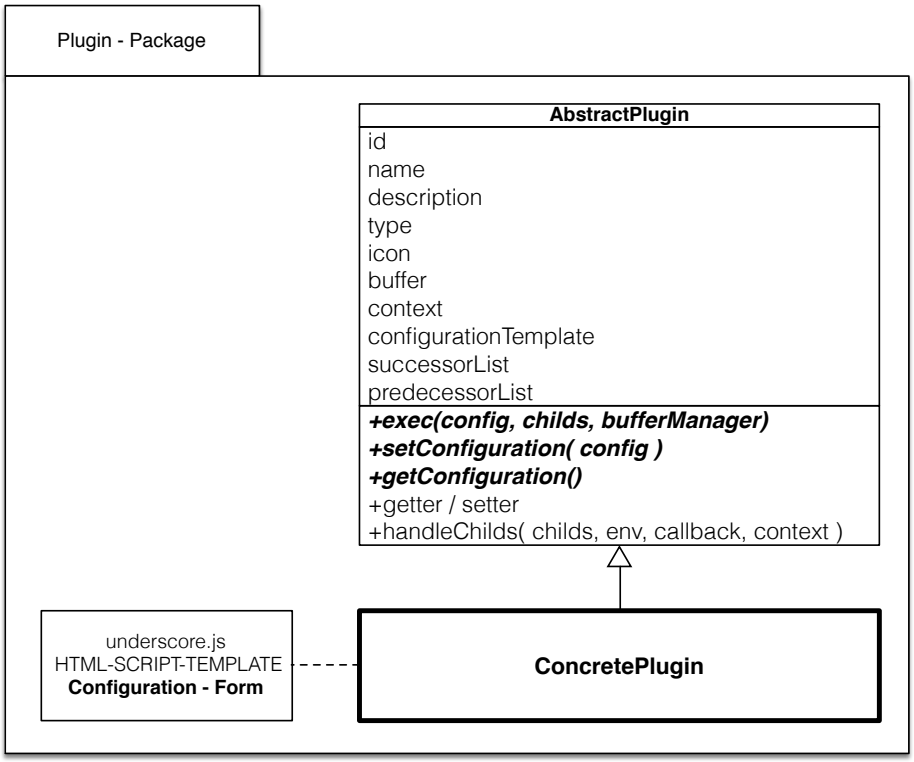


Abbildung 7.8.: Minimale Struktur eines Plugins

Plugins (siehe Abbildung 7.8) sind jene Teile des Systems, welche für die grafische Darstellung der gegebenen Daten verantwortlich sind. Jede einzelne Komponente ist dabei für einen kleinen Teil des resultierenden Gesamtbildes zuständig. Wie ein Datensatz dargestellt wird, ist vom verwendeten Plugin-Konglomerat und der Konfiguration abhängig. Unterschiedliche Kombinationen von Plugins sowie deren Konfiguration ergeben eine Fülle an unterschiedlichen Darstellungsmöglichkeiten.

Jedes Plugin muss von der abstrakten Basisklasse *AbstractPlugin* abgeleitet werden. Sie umfasst die grundlegende Funktionalität eines Plugins und stellt die Existenz wichtiger Methoden, welche bei einem konkreten Plugin

implementiert werden müssen, sicher.
Ein Plugin „weiß“, wie es konfiguriert werden kann und stellt dafür ein Javascript-HTML-Template zur Verfügung. Zum Setzen, beziehungsweise Lesen von Konfigurationsdaten, werden zusätzlich Callback-Methoden (*setConfiguration*, *getConfiguration*) geboten. Zum Einsatz kommt hierbei die Template-Engine von underscore.js (Ashkenas, 2015), welche vom SceneGraph-Modul (Abschnitt 7.1.3)verwendet wird.

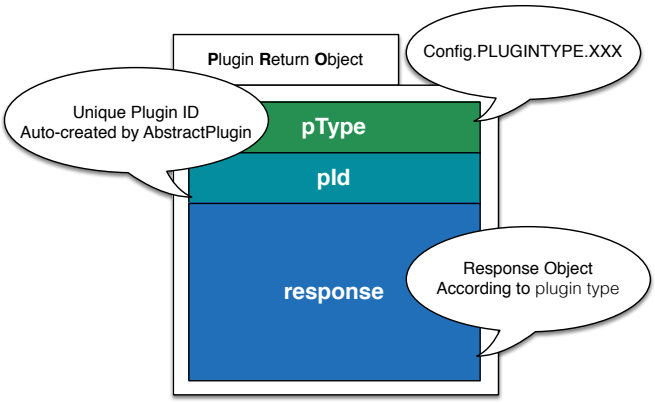


Abbildung 7.9.: PRO - Plugin Return Object

Zentraler Bestandteil eines jeden Plugins ist seine *exec* Methode, welche mit Hilfe ihrer Parameter **config**(object) und **childs**(array) alle nötigen Verarbeitungsschritte in Gang setzt und das Ergebnis über ein Return Objekt (siehe Abbildung 7.9) an eine höher liegende Instanz weiterreicht. Über den Parameter *childs* kann jedes Plugin auf die Ergebnisse seiner direkten Nachfolger zugreifen und für seine eigene Verarbeitung verwenden[1]. Dies setzt ein gewisses Wissen über andere Plugins voraus.
Um die Bindung und die Abhängigkeiten zwischen Plugins zu minimieren, wird jedem Plugin ein **Plugin-Typ** zugeordnet. Der Typ bestimmt die Struktur des Response-Objektes innerhalb des *PRO*.
Der Umgang mit Ergebnissen von Nachfolger-Plugins wird mit Hilfe der

[1] Anmerkung: Im Plot-Modul werden Templates rekursiv (deep-first) geparst, wodurch der „Nachfolger “eines Plugins zuvor ausgeführt wird.

handleChilds(childs, env, callback, context) Methode vereinfacht. Für gewöhnlich wird sie innerhalb von *exec* aufgerufen. Sie setzt Attribute und Werte im **ENV**-Objekt und ruft abschließend die übergebene Callback-Methode auf.

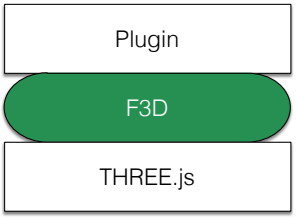


Abbildung 7.10.: F3D als Schicht zwischen einem Plugin und THREE.js

Visualisierungen eines Datensatzes werden mit Hilfe des WebGL Frameworks *THREE* (Abschnitt 5.2.1) realisiert (Abbildung 7.10). Es handelt sich hierbei um ein Projekt, an welchem aktiv gearbeitet wird. Um Adaptionen auf neuere Versionen zu erleichtern, greifen Plugins nicht direkt auf THREE.js zu, sondern über die Zwischenschicht Framework3D (kurz F3D).

Plugin Typen

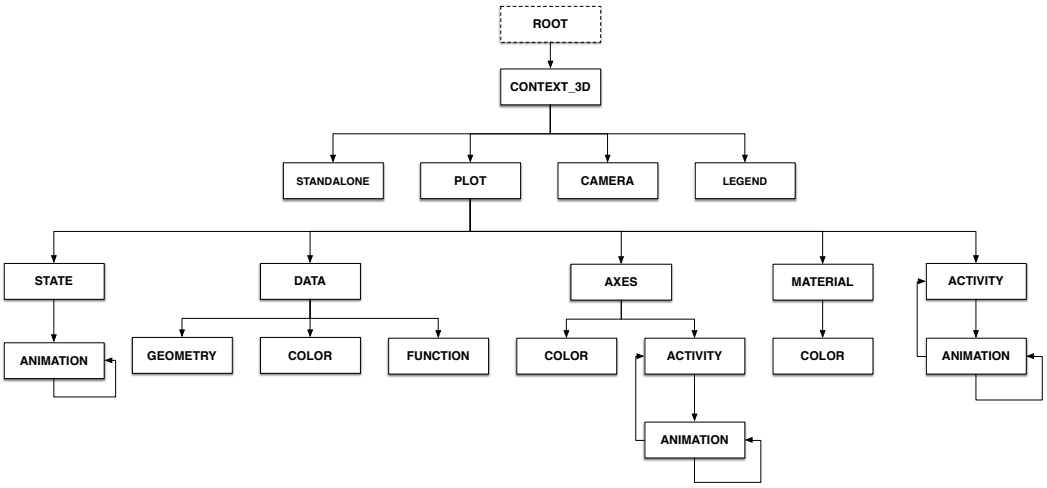


Abbildung 7.11.: Plugin Typen und ihre Relationen

Plugins arbeiten gemeinsam an der Lösung einer Aufgabe, wobei jedes Plugin einen kleinen Teil zur Lösung beiträgt. Der Output eines Plugins ist gleichzeitig der Input eines anderen und wird über den Parameter „childs“ der *exec* Methode weitergereicht.
Der Typ bestimmt die Struktur des Response-Objektes innerhalb des PRO (Abbildung 7.9) und definiert grob den Aufgabenbereich.
Konkrete Plugins wissen nur von diesem Zusammenhang, sie wissen aber nichts über die eigentliche Implementierung ihrer Nachfolger.
Anhand des Plugin-Typen werden auch Vorgänger und Nachfolger eines Plugins bestimmt. Dies beeinflusst die Abarbeitungsreihenfolge.
Eine Übersicht über Plugin-Typen und ihre Relationen zueinander wird in Abbildung 7.11 geboten.

Zu jedem Plugin-Typen wurde mindestens ein konkretes Plugin implementiert. Eine Auflistung aller Plugins ist in Tabelle 7.1 zu finden. Der Ablauf der Datenaufbereitung hin zur schlussendlichen Visualisierung wird im Abschnitt 7.3 anhand eines Beispieles näher erörtert.

3D	Camera	File	Dataset	Line
Plane	Color	Heatmap	Message	Link
Function	Basic - Material	Wireframe - Material	Axes	Legend
Fade	Move	Scale	Tick	Label
ScatterPlot	SurfacePlot	LinePlot	BarchartPlot	
Mouse	State			

Tabelle 7.1.: Implementierte Plugins

7.1.5. Initialisierung

Das {v}Plot Framework wird mit Hilfe des require.js Optimizer-Tools minimiert, optimiert und über die folgenden Wrapper-Objekte bereitgestellt:

- **vlib_base** und
- **vlib_full**.

vlib_base beinhaltet die Kern-Elemente des Frameworks, alle Plugins, sowie das Plot-Modul. *vlib_full* bietet alle Komponenten, welche erstellt wurden. Nicht jede Benutzergruppe verwendet den gleichen Funktionsumfang. In der Einleitung des Kapitels 7 wurden die Benutzer-Rollen, **Aktive-**, sowie **Passive**-Benutzer ausgemacht.
Ein aktiver Benutzer des Frameworks erstellt Inhalt und ist daher auf den vollen Funktionsumfang angewiesen (*vlib_full*), wohingegen ein passiver Benutzer nur indirekt das Werk eines Aktiven-Benutzers konsumiert. Hierfür ist lediglich das Plot-Modul notwendig (*vlib_base*). Man könnte auch eine Unterscheidung anhand des Einsatzgebietes bestimmen.

- **vlib_base** $\Rightarrow$ Frontend und
- **vlib_full** $\Rightarrow$ Backend.

```
require( ['vLib_base', './templates/t1.vlib'],
 function ( Wrapper, T1 ) {
 var v = new Wrapper.VLib({
  basePath : 'js/vplot',
  appPath : './'
 });
 var plot1 = new Wrapper.PlotModule();
 v.registerModule( plot1 );
 plot1.init('#vPlot');
 v.load( T1, plot1 );
});
```

Listing 7.2: Initialisierung

Code-Ausschnitt 7.2 veranschaulicht eine minimalistische Initialisierung des Frameworks. Hierbei ist ein Plot-Modul involviert, welches eine statisch geladene Template-Datei (*t1.vlib*) darstellt.
Die Schnittstelle *VLib* bietet eine Fülle an Möglichkeiten, um unterschiedlichste Callback-Funktionen zu setzen. Es können auch beliebig viele Plot-Module instanziiert und registriert werden, um gleichzeitig mehrere Datensätze zu visualisieren.

7.2. Visualisierungs-Prozess

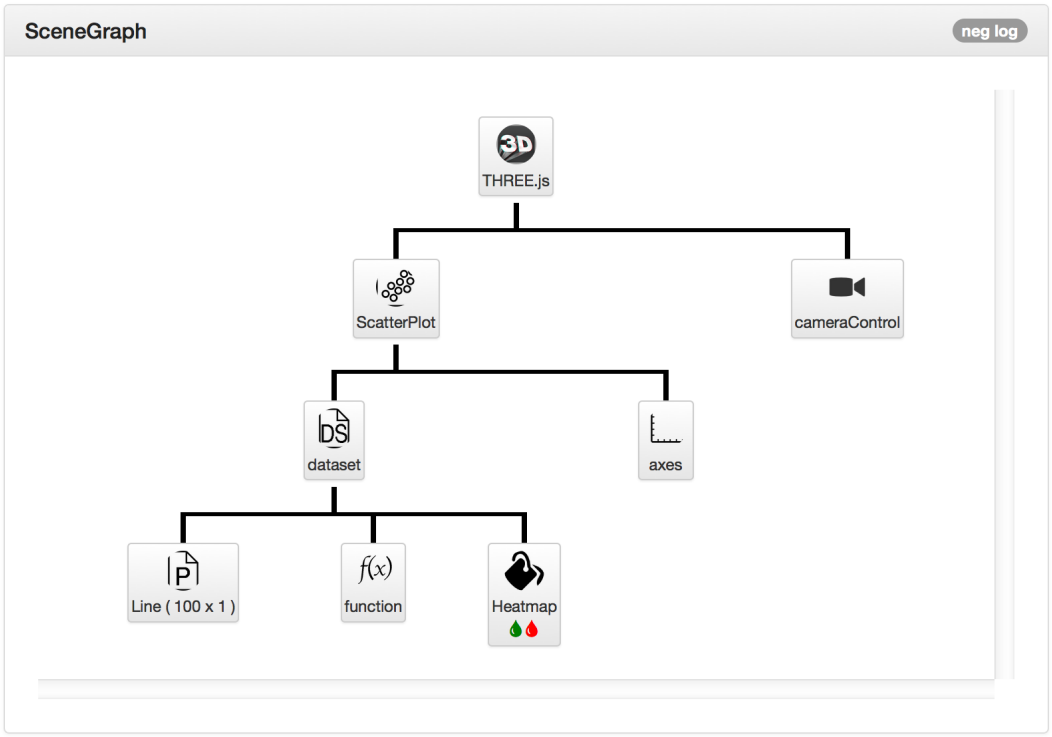


Abbildung 7.12.: 2D Visualisierung von log(x) mit 101 Datensätzen im Wertebereich zwischen -10 und +10

In den folgenden zwei Abschnitten wird das Konzept der Datenaufbereitung und der Visualisierung näher erörtert. Die Abschnitte bestehen aus zwei Teilen. Zuerst wird auf die Strukturen involvierter Plugins näher eingegangen und im Anschluss wird der Datenfluss anhand eines konkreten Beispieles beschrieben.

Die Struktur des verwendeten Beispiel-Templates wird in Abbildung 7.12 geboten. Die Konfiguration der Plugins, sofern diese von Standardwerten abweichen, ist in Tabelle 7.2 aufgelistet.

Plugin	Konfiguration
3D	Kamera-Position: manuell angepasst
CameraControl	Pivot Point: $200, 200, 0$
ScatterPlot	Material: Shader
	Particle size: 35
	Image: Star
Axes	Ticks: „Mirror tick lines“ ausgewählt
	Ticks: „Nice Ticks“ ausgewählt
	Steps: $x = 6$, $y = 6$
	Raster: (x,y) und (y,x) aktiviert, $opacity = 0.2$
Line	Value range: -10 bis 10
	Number of segments: 100
Function	$y = log(x)$
Heatmap	„Map y“ ausgewählt
	Number range: 0 bis 1

Tabelle 7.2.: Konfiguration der Plugins aus Beispiel 7.12

7.2.1. Datenaufbereitung

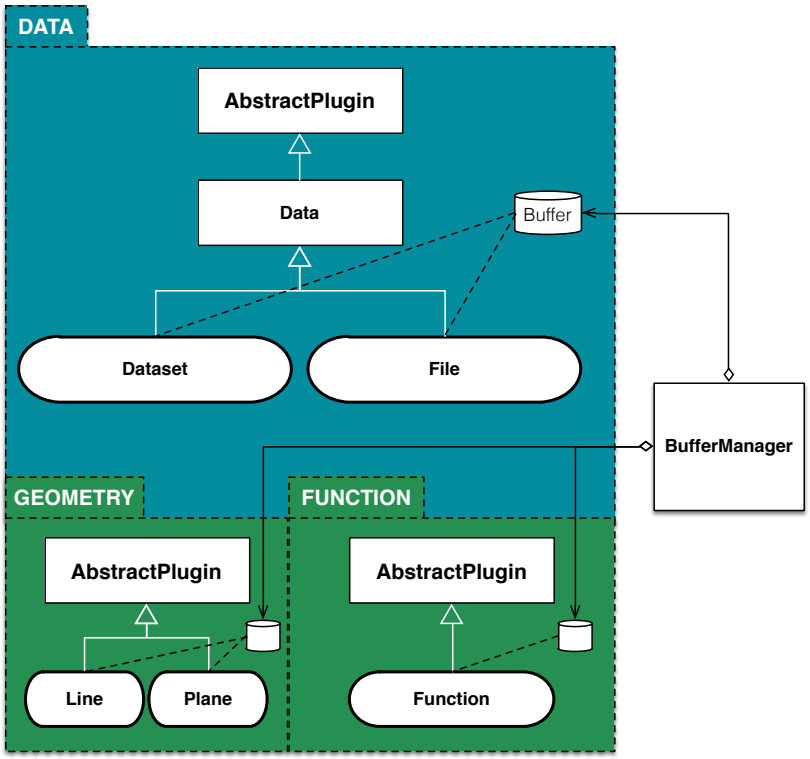


Abbildung 7.13.: Plugins vom Typ DATA, GEOMETRY und FUNCTION
Rechtecke: Klassen
Ovale Symbole: Konkrete Plugins

In Abbildung 7.13 wird eine Übersicht über jene Plugins geboten, welche in den Prozess der Datenaufbereitung involviert sein können. Diese Grafik umfasst Plugins von den Typen *DATA*, *GEOMETRY*, *FUNCTION* und *COLOR*. Ovale Symbole repräsentieren konkrete Plugins, Rechtecke entsprechen Klassen, von welchen Funktionalität geerbt oder verwendet wird.
Im Beispiel 7.12 sind die Plugins *Dataset*, *Line*, *Function* und *Heatmap* involviert.

Die Konfigurationsoberfläche von *Dataset* bietet zwei wesentliche Einstellungsmöglichkeiten:

1. **Mapping**:
   Ein Mapping erzeugt ein *<Key, Value>* Paar.
   Key entspricht einem vordefiniertem Label (z.B.: *x*, *y*, *z*, *color*, *size*, et cetera)
   Value steht für einen konkreten Wert eines Datensatzes.
2. **Data**:
   Anhand eines zuvor definierten Mappings können manuell Datensätze eingetragen werden.

Die direkten Nachfolger von *Dataset* bieten Callback-Methoden, mit deren Hilfe Datensätze modifiziert werden können. So erzeugt *Line* einen Array an Werten, welche vom *Dataset*-Plugin anhand des definierten Mappings eingefügt werden. Das Callback des *Function*-Plugins wird auf jeden Datensatz angewendet und das *Heatmap*-Callback erweitert jeden Datensatz um Farbeigenschaften.

Die eigentliche Datenaufbereitung ist ein nicht blockender Prozess (siehe Abschnitt 7.3.2 - SafeLoop), welcher innerhalb von *Dataset* in Gang gesetzt und über eine Instanz der Klasse Data gehandhabt wird.
Dieser Prozess umfasst zehn Schritte, welche anhand der Methode *Data.preprocess()* abgearbeitet werden:

1. **initBuffer()**:
   Diese Methode erzeugt Buffer-Attribute für Text-Labels und Datensätze. Mit Hilfe des Bezeichners *RAW_DATA* kann ein Datenobjekt abgerufen werden, welches einen Array hält. *RAW_LABELS* hält einen Array, in welchem Text-Labels gespeichert werden können.
   Eine ausführlichere Erklärung von Buffer-Objekten und deren Attribute wird im Abschnitt 7.3.1 geboten.

2. **handleChilds()**:
   Hier werden Callback-Funktionen der Plugins *Line*, *Function* und *Heatmap* im *ENV*-Objekt des *Dataset* Plugins gesetzt.

3. **fillDataBuffer()**:
   Die Methode *fillDataBuffer()* füllt *RAW_DATA* mit Datensätzen.

```
1  [{ x:-10}, { x:-9.8,}, ..., { x:9.8}, { x:10}]
```

Listing 7.3: RAW_DATA nach dem Aufruf von *fillDataBuffer()*

Codeausschnitt 7.3 illustriert den Zustand des *RAW_DATA* Arrays. Die ersten beiden und letzten beiden Datensätze werden angezeigt.
Das Attribut *x* jedes Datensatzes wird über die Callback-Funktion des *Line*-Plugins gesetzt.

4. **fillAxesLabels()**:
   Diese Methode durchsucht Datensätze nach Label-Mappings und verschiebt diese nach *RAW_LABELS*. Im Beispiel 7.12 wurden keine Text-Labels definiert, daher bleibt *RAW_LABELS* leer.

5. **autofillMissingValues()**:
   Die schlussendliche Visualisierung des Datensatzes wird mit Hilfe von THREE.js realisiert. Da es sich hierbei um ein 3D-Framework handelt, werden neben dem Attribut *x* noch mindestens Attribute für Koordinatenangaben der Dimensionen *y* und *z* benötigt. Da im Beispiel 7.12 keine Angaben zu diesen Attributen getroffen wurden, werden diese mit Default-Werten gefüllt.

```
1  [{ x:-10, y:0, z:0, size:1},
2  { x:-9.8, y:0, z:0, size:1},
3  ...,
4  { x:9.8, y:0, z:0, size:1},
5  { x:10, y:0, z:0, size:1}]
```

Listing 7.4: RAW_DATA nach dem Aufruf von *autofillMissingValues()*

6. **ensureNumeric()**:
   Datensätze können von verschiedenen Quellen kommen (autogeneriert, externe Dateien, manuelle Benutzereingabe). Die Methode *ensureNumeric* stellt sicher, dass im Weiteren nicht mit String-Literalen

gerechnet wird.

7. **processFunctionPlugins()**:
   Diese Methode prüft, ob Callback-Funktionen von *Function*-Plugins gesetzt wurden. Im Beispiel 7.12 ist dies der Fall.
   Die Funktion

```
y = log(x)
```

   wird daher auf jeden Datensatz des Arrays *RAW_DATA* angewandt.

```
[{ x:-10, y:NaN, z:0, size:1},
{ x:-9.8, y:NaN, z:0, size:1},
...,
{ x:0, y:-Infinity, z:0, size:1},
...,
{ x:9.8, y:0, z:0, size:1},
{ x:10, y:0, z:0, size:1}]
```

Listing 7.5: RAW_DATA nach dem Aufruf von *processFunctionPlugins()*.

8. **removeInvalidDatasets()**:
   Die Modifikation von Datensätzen anhand von Funktionen kann dazu führen, dass ungültige Werte gesetzt werden. Im Beispiel 7.12 wurde ein Wertebereich von *-10* bis *+10* für *x* definiert und die Funktion *y = log(x)* eingebunden. Der Logarithmus von negativen Zahlen ist aber nicht definiert. Des Weiteren ist *log(0) = minus unendlich*.
   Die Methode *removeInvalidDatasets()* prüft Koordinatenattribute mit Hilfe der Javascript-Funktion *isFinite* und filtert ungültige Datensätze heraus.

```
[{ x:0.193, y:-1.609, z:0, size:1},
{ x:0.4, y:-0.9162, z:0, size:1},
...,
{ x:9.8, y:2.282, z:0, size:1},
{ x:10, y:2.302, z:0, size:1}]
```

Listing 7.6: RAW_DATA nach dem Aufruf von *removeInvalidDatasets()*

9. **processColorPlugins()**:
   Diese Methode vollzieht die letzte Modifikation des *RAW_DATA* Arrays. Jeder Datensatz wird um ein Farbattribut erweitert. Falls eine Callback-Funktion, welche von einem Plugin des Typs *COLOR* stammt, gefunden wird, wird diese zur Bestimmung eines konkreten Farbwertes herangezogen, ansonsten wird ein Standardwert gesetzt.

10. **updateDirtyFlags()**:
    Abschließend wird geprüft, ob eine ältere Version des *RAW_DATA* Buffer-Attributes vorliegt und ob sich die aktuellen Werte, im Vergleich zu diesem, unterscheiden. Falls dem so ist, wird die *dirty* Flag des Buffer-Attributes auf *TRUE* gesetzt. Dies ist ein Indikator für andere Plugins, welche dieses Buffer-Attribut verwenden, um ihre eigenen Buffer zu aktualisieren.

7.2.2. Visualisierung

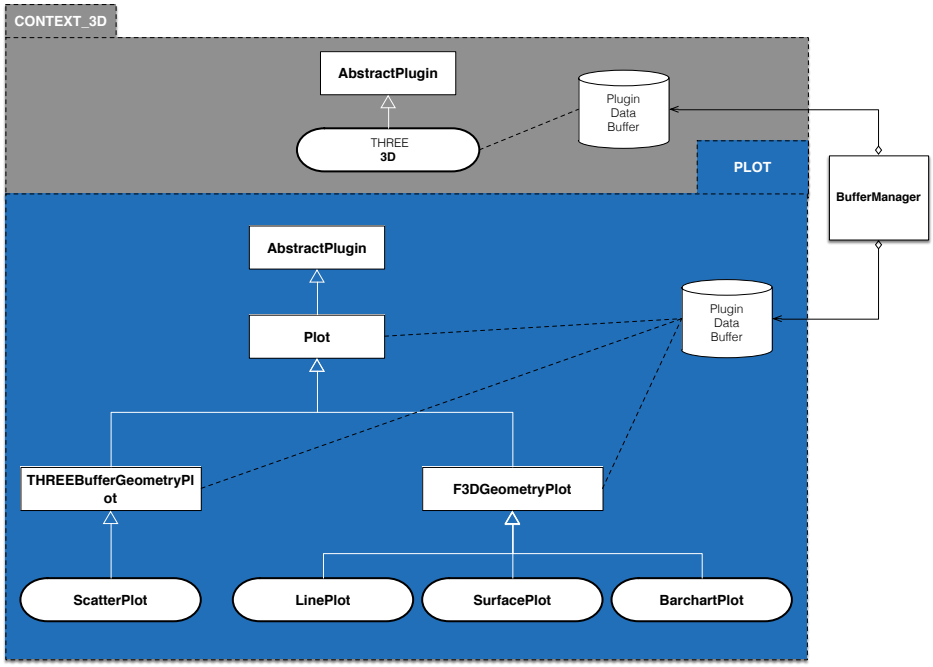


Abbildung 7.14.: Plugins vom Typ CONTEXT_3D und PLOT

Abbildung 7.14 bietet einen Einblick in Plugins mit direktem Bezug zur Visualisierung von aufbereiteten Daten. Ovale Elemente symbolisieren konkrete Plugins. Rechtecke symbolisieren Elemente, von welchen Funktionalität geerbt oder verwendet wird.

(THREE) 3D ist ein Wurzel-Knoten jeder Template-Datei und steht an letzter Stelle in der Ausführungsreihenfolge.
Dieses Plugin instanziiert THREE.js spezifische Elemente (wie *WebGLRenderer*, *Scene*, *Camera*, et cetera), initialisiert den Render-Loop und stellt Handler zur Verfügung (z.B.: für Größenänderungen des Browser-Fensters, *Mouse-* oder *Renderer-Activities*).

Direkte Nachfolger von *(THREE) 3D* sind unter anderem Plugins vom Typ-Plot. Diese stellen über ihr Return-Objekt ein Callback (*run*) zur Verfügung. *(THREE) 3D* setzt den Prozess der eigentlichen Datenvisualisierung durch den Aufruf dieser Callabacks in Gang.

```
1 for ( var i = 0, len = plotList.length; i < len; ++i ) {
2  plot = plotList[ i ];
3  /* SET DONE CALLBACK */
4  plot.response.onDone.call(plot.response.context,
5   onDone, this);
6  /* RUN PLOT */
7  plot.response.run.apply( plot.response.context,
8   [ scene, camera ] );
9 }
```

Listing 7.7: Ausschnitt aus der Methode runPlots() des Plugins 3D

Plot Plugins (Scatter-, Surface-, Line- und Barchart-Plot) werden entweder von der Klasse *F3DGeometryPlot* oder *THREEBufferGeometryPlot* abgeleitet. Beide sind wiederum Spezialisierungen von der Klasse *Plot*.
F3DGeometryPlot und *THREEBufferGeometryPlot* bieten zum größten Teil die gleiche Funktionalität, greifen dabei aber auf unterschiedliche Datenstrukturen zurück.
THREEBufferGeometryPlot erzeugt anhand gegebener Daten (*RAW_DATA Buffer*) eigene Speicherobjekte, welche Datenfelder vom Typ *Float32Array* halten.
Die Verwendung eines solchen Datentyps bringt mehrere Vorteile mit sich. Der Zugriff auf Elemente eines solchen Datenfeldes ist sehr performant. Unabhängig von der Größe kann diese Datenstruktur sehr schnell in den Kontext eines WebWorkers übergeben werden (Bidelman, 2011). Außerdem kann es direkt in einem THREE.BufferGeometry-Attribut referenziert werden.
F3DGeometryPlot verwendet gewöhnliche Arrays, welche THREE.js spezifische Objekte beinhalten (THREE.Vector3D, THREE.Color).

Die Klasse *Plot* übernimmt allgemeine Aufgaben, welche alle konkreten Plot-Plugins gemeinsam haben, ohne THREE.js spezifische Konstrukte zu erstellen beziehungsweise zu verwenden. Ihre Methoden arbeiten mit *RAW_DATA* Buffern von *Data-Plugins.*

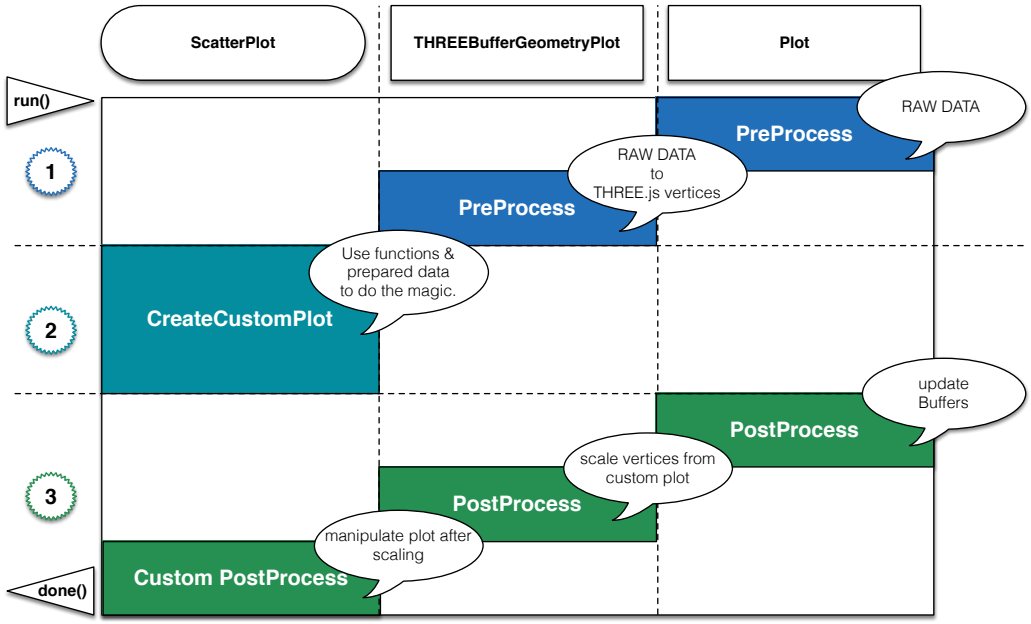


Abbildung 7.15.: Drei Schritte zur Visualisierung - PreProcess, Create, PostProcess

Anhand des Beispieles 7.12 wird der Prozess der Visualisierung im Folgenden näher erörtert.
Abbildung 7.15 illustriert diesen Prozess, welcher durch den Aufruf der Methode *run(scene,camera)* des konkreten Plot-Plugins „ScatterPlot“ in Gang gesetzt wird.

Der Ablauf kann in drei wesentliche Schritte unterteilt werden:

1. **PreProcess**,
2. **CreateCustomPlot** und
3. **PostProcess**.

PreProcess umfasst alle Schritte, welche vor der eigentlichen Visualisierung durchgeführt werden müssen. *Plot.preProcsess()* initialisiert Buffer und fasst Daten, welche von Data-Plugins stammen (*RAW_DATA Buffer*) zusammen. Maxima und Minima werden ermittelt, welche für die spätere Skalierung gebraucht werden. Anhand der zusammengefassten Daten aus *Plot.preProcsess()* erzeugt *THREEBufferGeometryPlot.preProcsess()* Speicherelemente für Koordinaten (Config.BUFFER.COORDS) und Farben (Config.BUFFER.COLORS) und füllt diese mit THREE.js typischen Elementen. Der Codeausschnitt 7.8 veranschaulicht den Zustand der Buffer-Attribute *COLORS*, *COORDS* und *BORDERS*.

```
1  // COLORS: [r1,g1,b1,r2,g2,b2, ...]
2  [0, 0.501960813999176, 0, ...]
3  // COORDS: [x1,y1,z1, x2, y2, z2, ...]
4  [0.20000000298023224, -1.6094379425048828, 0, ...]
5  // BORDERS:
6  {
7   nice : true,
8   xMin : 0, yMin : -2, zMin : 0,
9   xMax : 10, yMax : 2.5, zMax : 0
10 }
```

Listing 7.8: Buffer Attribute COLORS, COORDS und BORDERS nach dem Aufruf von preProcess().

CreateCustomPlot ist eine Methode eines Plot-Plugins, in welcher eine konkrete Visualisierung anhand von THREE.js erzeugt wird. Mittels Helper-Methoden, welche von *THREEBufferGeometryPlot*, beziehungsweise von *F3DGeometryPlot* zur Verfügung gestellt werden, können Buffer-Attribute wie COORDS oder COLORS abgefragt und Geometrieobjekte erstellt werden.

```
1 var coords = Plugin.superClass.getCoords( bufferManager,
    env );
2 var colors = Plugin.superClass.getColors( bufferManager,
    env );
3 var geometry = Plugin.superClass.getGeometry( coords,
    colors );
4 var mesh = new F3D.Mesh( geometry, new F3D.BasicMaterial
    () );
5
6 Plugin.superClass.setSystem( mesh );
7 Plugin.superClass.registerCustomPostprocess( env,
    function ( system ) {
8  ...
9 });
```

Listing 7.9: Beispielhafter Aufbau der Methode createCustomPlot()

Im konkreten Beispiel 7.12 wird ein ScatterPlot anhand eines THREE.js Partikelsystems erstellt. Über die Methode *getShader()* wird ein *THREE.ShaderMaterial* retourniert, welches eigens hierfür erstellte Vertex- und Fragment-Shader involviert.

PostProcess umfasst alle Schritte, die durchgeführt werden müssen, nachdem Informationen über eine konkrete Visualisierung bekannt sind. *Plot.postProcess()* aktualisiert Buffer (Buffer.SCALING, BUFFER.BORDERS, BUFFER.METADATA, et cetera), ruft Callbacks von Nachfolger-Plugins auf und gibt Speicher von nicht mehr benötigten Buffern frei.
Die Aufgabe von *THREEBufferGeometryPlot.postProcess()* beziehungsweise *F3DGeometryPlot.postProcess()* umfasst hauptsächlich die Skalierung von Vektoren anhand gegebener Skalierungsfaktoren und den Maxima/Minima der Daten. Die Skalierung ist ein asynchroner Prozess, welcher über WebWorker (siehe Abschnitt 7.3.3) gehandhabt wird.
Innerhalb eines Plot-Plugins kann ein zusätzlicher PostProcess registriert werden, um bereits skalierte Vektoren im Nachhinein zu modifizieren (siehe Codeausschnitt 7.9 Zeile 7).

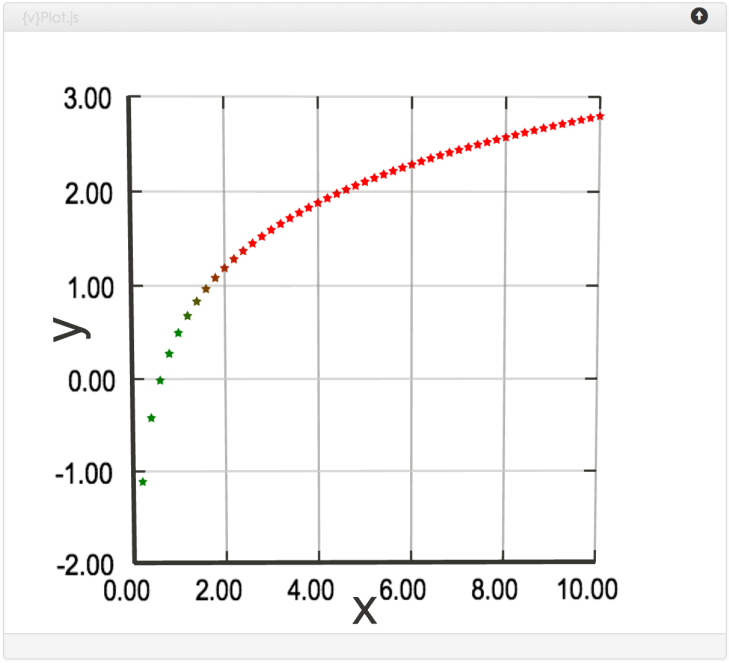


Abbildung 7.16.: Visualisierung des Beispiels 7.12

Nachdem Rohdaten die zehn Schritte der Datenaufbereitung durchlaufen haben, die Prozesskette der Datenvisualisierung (PreProcess, Create, Post-Process) abgeschlossen wurde und zusätzliche Plugins, wie Axes (siehe Abschnitt 7.3.5) ihre Arbeit abgeschlossen haben, wird das Template aus Beispiel 7.12 im Plot-Modul dargestellt (siehe Abbildung 7.16).

7.3. Details

Die folgenden sechs Abschnitte beschreiben nennenswerte Komponenten des Frameworks, welche in den Prozess der Datenaufbereitung oder der Visualisierung involviert sind.

7.3.1. BufferManager und Buffer

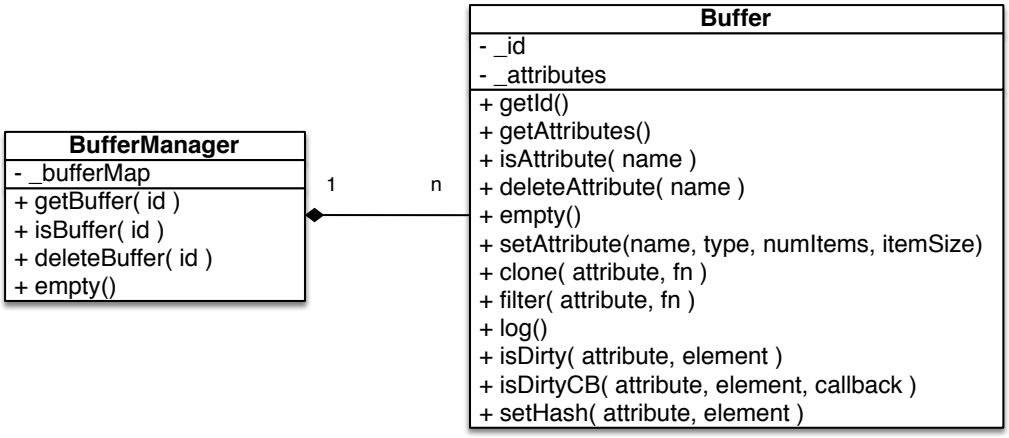


Abbildung 7.17.: Klassen Diagramm - BufferManager und Buffer

Jedes Plot-Modul hält eine Instanz eines BufferManagers und übergibt eine Referenz an jedes involvierte Plugin. Anhand des BufferManagers können Plugins Buffer erstellen beziehungsweise abrufen.
Ein BufferManager kann mit einer flüchtigen Datenbank verglichen werden und Buffer mit entsprechenden Tabellen, welche Attribute halten. Elemente eines Buffers bleiben auch bestehen, nachdem ein Render-Prozess neu in Gang gesetzt wurde.

Buffer dienen als zentraler Speicherort, über welchen plugin-übergreifend Informationen ausgetauscht werden können. Ein Buffer wird anhand einer Plugin-ID erzeugt. Kennt ein Plugin die ID eines anderen Plugins,

so kann es auf dessen Buffer zugreifen. Ist auch der Plugin-Typ bekannt, können auch direkt Buffer-Attribute abgefragt werden (vgl. Grafik 7.9 - PRO aus Abschnitt 7.1.4 - Plugins). Zum Beispiel erzeugen Plugins vom Typ *DATA* Buffer-Attribute mit Bezeichner *Config.BUFFER.RAW_DATA* und *Plot*-Plugins erzeugen *Config.BUFFER.COORDS* Attribute.

Ein Gedanke hinter diesem Buffer-System liegt in der Optimierung der Performance.
Die ursprüngliche Idee war, dass bei einem neuem Anlauf des Render-Prozesses nicht die gesamte Baumstruktur eines Templates geparst werden muss, sondern nun jene Teile, in welchen Veränderungen vorgenommen wurden. Im kleinen Rahmen konnte dies auch umgesetzt werden. Zum Beispiel bei der Rekonfiguration eines Plugins. Eine Änderung löst einen schreibenden Zugriff auf ein Buffer-Attribut aus, wodurch ein Indikator (*dirty flag*) gesetzt wird. Der Nutzen ist hier aber relativ gering.
Interessanter wäre es, den aufwendigen Prozess der Datenaufbereitung und der Skalierung zu umgehen und auf bereits aufbereitete Daten zurückzugreifen. Es lässt sich aber nicht im Vorhinein sagen, ob sich ein Datensatz geändert hat. Daten können aus einer externen Quelle stammen, welche sich zwischen zwei Render-Aufrufen geändert haben oder sie könnten über Funktionen, wie *Math.random()*, manipuliert worden sein. Ein Vergleich zu einer älteren Version eines Datensatzes ist dadurch unumgänglich.
Es ergeben sich hierbei zwei Probleme:

1. Der Aufwand steigt mit der Größe des Datensatzes.
2. Der Speicherbedarf für Rohdaten verdoppelt sich.

Um den Speicherbedarf zu reduzieren und den direkten Vergleich zwischen zwei Versionen eines Buffer-Attributes zu beschleunigen, bietet jedes Buffer-Objekt Methoden an, um HASH Werte zu erzeugen und somit Attribute schneller zu vergleichen. Im großen Maßstab bietet dies aber keine befriedigende Lösung, da die Erzeugung eines HASH-Wertes, anhand eines sehr großen Datensatzes, alles andere als schnell vonstatten geht. Da die Kosten-Nutzen-Relation hier in keinem sinnvollen Verhältnis steht, wurde das implementierte „BYPASS“ System zur Umgehung der Datenaufbereitung und Skalierung vorläufig deaktiviert.

Der Codeausschnitt 7.10 veranschaulicht die Erstellung des Buffer-Attributes RAW_DATA.

```
1 var buffer = bufferManager.getBuffer( env.id );
2 if (!buffer.isAttribute( Config.BUFFER.RAW_DATA)) {
3  buffer.setAttribute(Config.BUFFER.RAW_DATA, Array);
4 }
```

Listing 7.10: Erzeugung des Buffer-Attributes RAW_DATA für Datensätze

7.3.2. SafeLoop

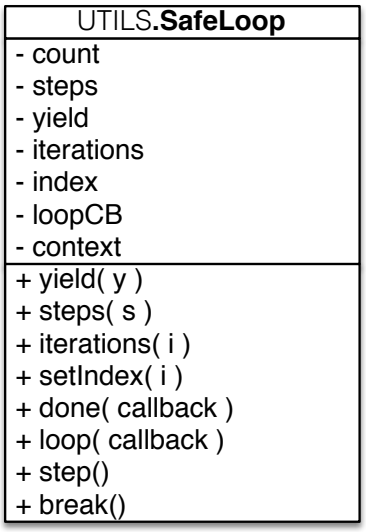


Abbildung 7.18.: Nicht blockender Loop

Die Aufbereitung von Daten geschieht, für gewöhnlich, innerhalb einer Schleife (while, do-while, for), wobei einzelne Werte eines Datensatzes manipuliert werden. Bei großen Datensätzen kann dies viel Zeit in Anspruch nehmen und zu Problemen mit den Sicherheitsrichtlinien von Browsern führen.

Wenn ein Skript lange genug läuft, hat dies eine Warnmeldung des Browsers zur Folge. Daher muss ein Weg gefunden werden, um dem Browser mitzuteilen, dass es sich hierbei nicht um eine Endlos-Schleife beziehungsweise eine Endlos-Rekursion handelt.
Eine Lösung bildet hierbei die Aufteilung eines lange laufenden Prozesses in eine Vielzahl kürzerer Prozesse. Realisiert werden kann dies mit Hilfe der Javascript Funktionen *setTimeout* oder *setInterval*. *SafeLoop* (Abbildung 7.18) beruht auf Timeouts und ist eine Erweiterung des von Tapia vorgestellten Algorithmus (Tapia, 2010).

```
1  var raw_data = buffer.getAttribute( Config.BUFFER.
       RAW_DATA );
2  var raw_data_len = raw_data.array.length;
3  var dataset;
4
5  var sl = UTILS.safeloop();
6  sl.iterations( raw_data_len )
7  .done( function () {
8  // onDone action
9  } )
10 .loop(function( index ){
11 dataset = raw_data.array[ index ];
12 // ...
13 })
```

Listing 7.11: SafeLoop im Einsatz

7.3.3. WorkerFactory

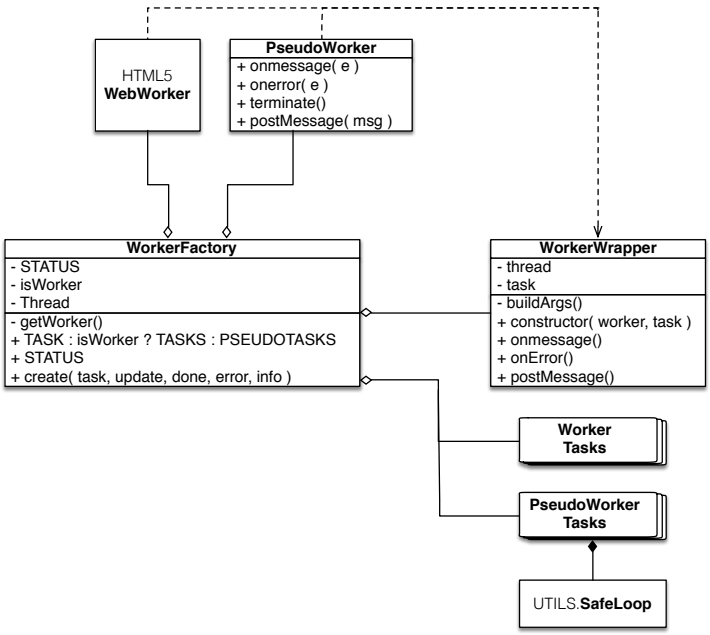


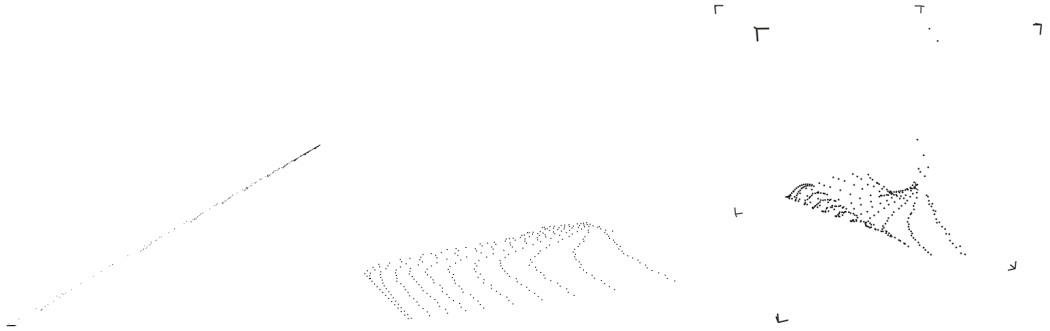
Abbildung 7.19.: Klassendiagramm - WorkerFactory

Ruft man die Methode *WorkerFactory.create(task, /* optional callbacks */)* auf, wird ein *WorkerWrapper* Objekt retourniert. Je nachdem, ob WebWorker zur Verfügung stehen, hält *WorkerWrapper* eine Referenz auf einen echten WebWorker oder einen PseudoWorker. Des Weiteren beinhaltet dieses Objekt einen konkreten Worker-Task, welcher anhand des ersten Parameters (*String*) der Methode *create()* bestimmt wird.
Ein konkreter Worker (WebWoker oder PseudoWorker) wird anhand eines *Blobs* erstellt. Dieser Blob beinhaltet den Programmcode aus einem Worker-Task.
Nachrichten, welche von einem Worker kommen, haben ein STATUS-Attribut. Unterschieden werden **STATUS.OK**, **STATUS.ERROR**, **STATUS.INFO**, **STATUS.DONE**. Je nach Status wird eine Nachricht an eine korrespondierende Callback-Funktion weitergereicht, welche über die Methode *WorkerFactory.create(task, /* optional callbacks */)* gesetzt werden kann.

```
1 var WorkerFactory = require( 'core/WorkerFactory.vlib' );
2 var wf = new WorkerFactory();
3 var worker = wf.create(
4  wf.TASK.TASK_SCALE_FLOAT32ARRAY_LIN,
5  onUpdateCB,onDoneCB,onErrorCB,onInfoCB);
6 var msg = { /* info + data : Float32Array */}
7 worker.postMessage(msg, [ msg.data.buffer ]);
```

Listing 7.12: WorkerFactory im Einsatz

7.3.4. Skalierung



(a) unskaliert (b) x-y gestaucht (c) z gestreckt

Abbildung 7.20.: Skalierung gewährt tieferen Einblick in die Struktur von Daten

Eine grafische *1:1* Darstellung von Roh-Daten ist meist nicht ausreichend, um nützliche Informationen ableiten zu können (vgl. Abbildung 7.20a). Oft ist es notwendig, Daten in einer „gestauchten“ (vgl. Abbildung 7.20b) oder „gestreckten“ (vgl. Abbildung 7.20c) Form abzubilden. Dies verzerrt zwar einen realen Sachverhalt, kann aber einen besseren Einblick gewähren, wenn die Relationen erhalten bleiben.
Es muss eine Funktion gefunden werden, welche die Ausgangsdaten optimal abbildet und die ursprünglichen Relationen erhält.

Eine lineare Skalierung ist eine Abbildung von Daten aus einer Definitionsmenge in eine Bildmenge. Eine optimale Skalierung erfordert, dass unterschiedliche Dimensionen (x, y, z) auch unterschiedlich skaliert werden können. Daher muss für jede Dimension eine Skalierungsfunktion gefunden werden.
Die Definitionsmenge der Ausgangsdaten wird im Folgenden als *Domain* (D) bezeichnet, die Bildmenge als *Range* (R).

Gesucht wird nun eine Funktion mit folgenden Eigenschaften:

$$\begin{aligned} F &: \mathbb{R}^1 \to \mathbb{R}^1 \\ f &: D \to R \end{aligned} \tag{7.1}$$

Reelle Zahlen aus dem Domain werden in die Bildmenge (Range) abgebildet, welche wiederum aus reellen Zahlen besteht.

$$\begin{aligned} &\forall y \in R \\ &\exists x \in D : y = f(x) \end{aligned} \tag{7.2}$$

Jeder Wert aus dem Domain kann in der Bildmenge abgebildet werden.

$$\begin{aligned} &\forall x, x' \in D \\ &f(x) = f(x') \Rightarrow x = x' \end{aligned} \tag{7.3}$$

Zu jedem Wert aus dem Domain existiert genau ein Wert in der Range-Menge.
Die gesuchte Funktion ist **surjektiv** (7.2) und **injektiv** (7.3). Daraus folgt, dass sie eindeutig und daher auch **bijektiv** ist.

$$y = \boldsymbol{k}x + \boldsymbol{d} \tag{7.4}$$

Eine solche Abbildung kann mit Hilfe einer Geradengleichung (7.4) realisiert werden, wobei die Steigung **k** und der Schnittpunkt der y-Achse (**d**) anhand gegebener Daten bestimmt werden müssen.

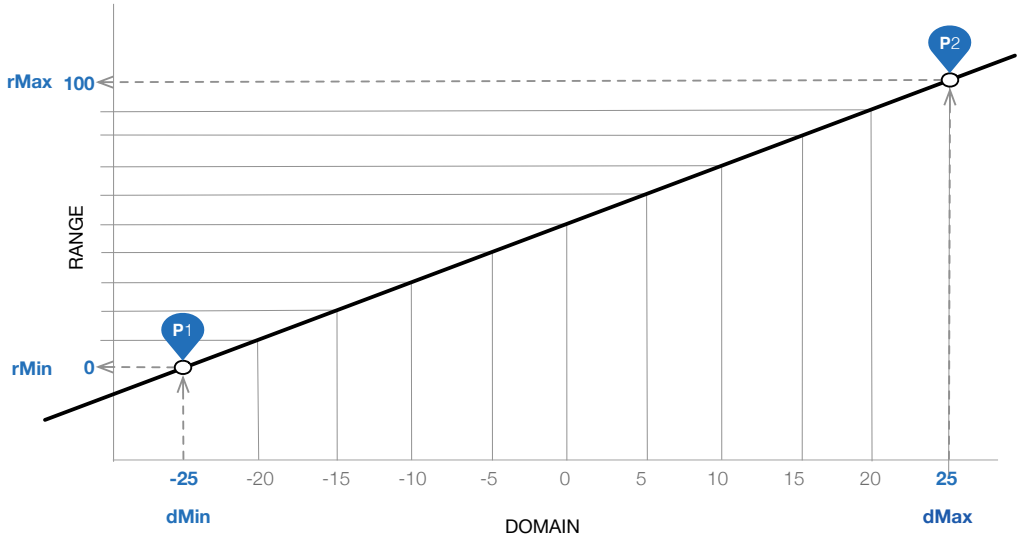


Abbildung 7.21.: Abbildungsfunktion - Domain nach Range

Im folgenden wird die Bestimmung der Abbildungsfunktion anhand von Abbildung 7.21 näher erörtert.
Entlang der x-Achse wird der Wertebereich einer Dimension eines Datensatzes (Domain) abgebildet, entlang der y-Achse die Bildmenge (Range). Bekannte Informationen werden blau hervorgehoben.
Dies sind im Domain *dMin* und *dMax*, welche entweder manuell definiert, oder während des Visualisierungsprozesses (Abschnitt 7.2) eruiert werden. Die Werte *rMin* und *rMax* werden manuell über ein *Plot-Plugin* definiert. Mit Hilfe dieser bekannten Werte können die Punkte $P_1(dMin,rMin)$ und $P_2(dMax,rMax)$ bestimmt werden.

$$k = \frac{\Delta R}{\Delta D} = \frac{R_{\text{Max}} - R_{\text{Min}}}{D_{\text{Max}} - D_{\text{Min}}} \tag{7.5}$$

Die Steigung $\boldsymbol{k}$ ergibt sich aus der Relation der Wertebereiche von Range zu Domain (7.5).

$$P_1 = (D_{\text{Min}}, R_{\text{Min}}) \tag{7.6a}$$
$$y = \boldsymbol{k}x + \boldsymbol{d} \tag{7.6b}$$
$$R_{\text{Min}} = k * D_{\text{Min}} + d \tag{7.6c}$$
$$\boldsymbol{d} = R_{\text{Min}} - k * D_{\text{Min}} \tag{7.6d}$$

Um den Schnittpunkt auf der y-Achse zu bestimmen, muss ein beliebiger Punkt der Geraden in die Funktion eingesetzt werden. Nach Umformung der Gleichung erhält man $\boldsymbol{d}$ (7.6d).

$$y = kx + R_{\text{Min}} - k * D_{\text{Min}} \tag{7.7a}$$
$$y - R_{\text{Min}} = k * (x - D_{\text{Min}}) \tag{7.7b}$$
$$y - R_{\text{Min}} = \frac{R_{\text{Max}} - R_{\text{Min}}}{D_{\text{Max}} - D_{\text{Min}}} * (x - D_{\text{Min}}) \tag{7.7c}$$
$$y = R_{\text{Min}} + \frac{R_{\text{Max}} - R_{\text{Min}}}{D_{\text{Max}} - D_{\text{Min}}} * (x - D_{\text{Min}}) \tag{7.7d}$$

Nachdem $\boldsymbol{k}$ und $\boldsymbol{d}$ in die Gleichung eingesetzt und diese vereinfacht wurde, erhält man schlussendlich die Abbildungsfunktion (7.7d).

```
1 var scale = new UTILS.Scale();
2 scale.range( [0, 100] ).domain( [-25, 25] );
3 console.log(scale.linear([-25,0,25]));
4 // OUTPUT: [0, 50, 100]
```

Listing 7.13: UTILS.Scale im Einsatz

7.3.5. Axes - Nice Labels

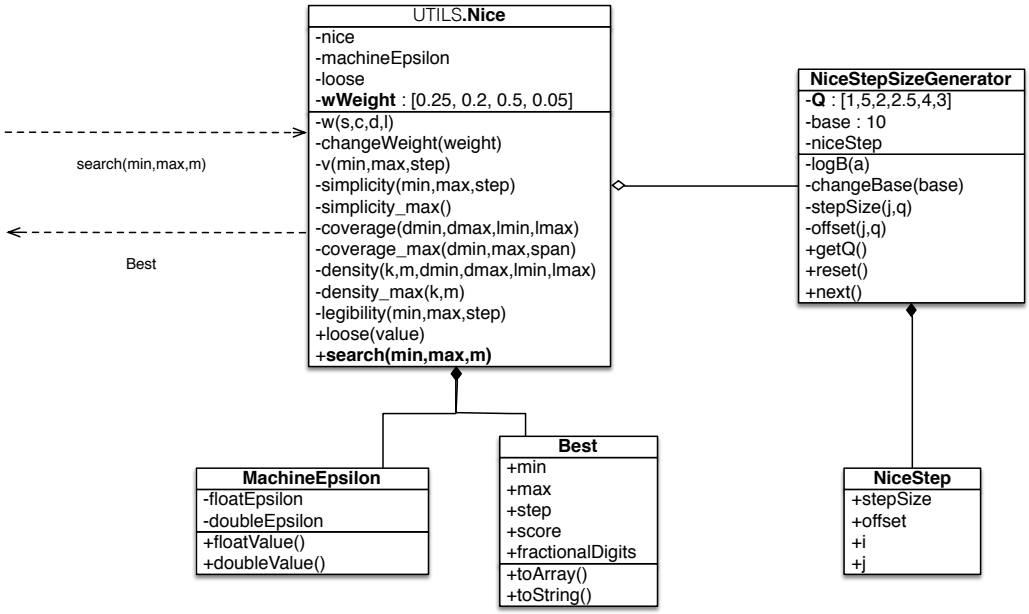


Abbildung 7.22.: UTILS.Nice - XWilkinson Portierung

UTILS.Nice beruht auf der Erweiterung von Wilkinson's Algorithmus zur Positionierung von Tick Labels auf Achsen (Talbot, Lin und Hanrahan, 2010). Genauer gesagt, wurde die von Ahmet Karahan umgesetzte Java Version (XWilkinson) nach Javascript portiert [2].
In der implementierten Version wird die Gewichtungsfunktion *legibility* nicht berücksichtigt. Durch *legibility* werden Faktoren wie Schriftgröße, Format, Ausrichtung und Überschneidung der Achen-Labels in die Optimierung mit einbezogen. Im dreidimensionalen Raum ist eine schlechte Positionierung von Labels aber schwer auszumachen. Außerdem erlaubt {v}Plot einem Benutzer die Kamera frei zu bewegen. Achsen-Labels richten sich nach der Kamera-Position aus. Aus diesem Grund müsste die Label-Positionierung bei jeder Bewegung der Kamera neu berechnet werden.

[2] `http://www.justintalbot.com/research/axis-labeling/`

7.3.6. Aktivitäten

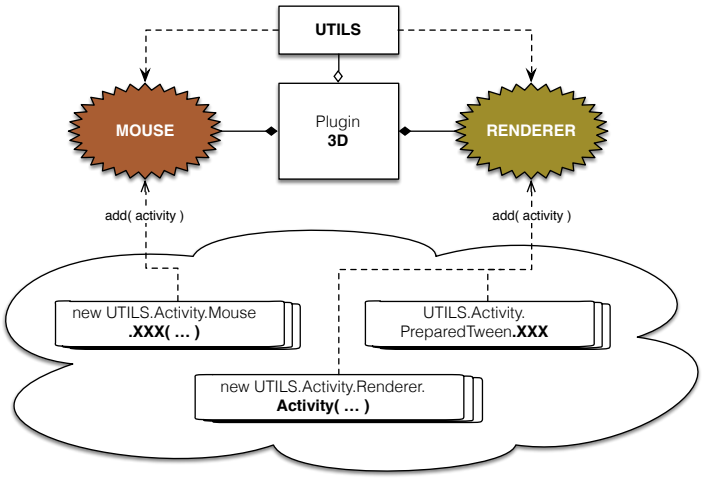


Abbildung 7.23.: Activities

Eine Aktion wird ausgelöst, wenn eine bestimmte Bedingung erfüllt wurde. Dies kann innerhalb eines Programmblockes geschehen oder durch ein bestimmtes Benutzerverhalten erfolgen.
Wenn sich die Position des Kamera-Objektes ändert, dann sollen sich auch die Koordinaten einer Lichtquelle ändern (siehe Beispiel 7.15). Oder, wenn ein Benutzer den Mauszeiger über ein bestimmtes dreidimensionales Objekt bewegt, soll dieses auf die xy-Ebene projiziert werden. Dies sind konkrete Beispiele für Bedingungen und Aktionen.
Aktivitäten werden über die Utility-Klasse *UTILS* zur Verfügung gestellt und vom Plugin *3D* gehandhabt. Dieses Plugin hält einerseits den *CANVAS*-Container, über welchen *Events* gesetzt werden können, andererseits wird hier auch der Render-Loop gesteuert. Somit stellt das Plugin *3D* alle Komponenten zur Verfügung, um auf Benutzereingaben reagieren und Animationen erstellen zu können. Gehandhabt werden Aktivitäten über die beiden *Singletons* MOUSE und RENDERER. Von jeder Stelle des $\{v\}$*Plot*-Projektes können konkrete *Activity*-Instanzen erzeugt werden, welche automatisch zu einem der genannten *Handler* hinzugefügt werden (vgl. Abbildung 7.23).

Mouse Activities

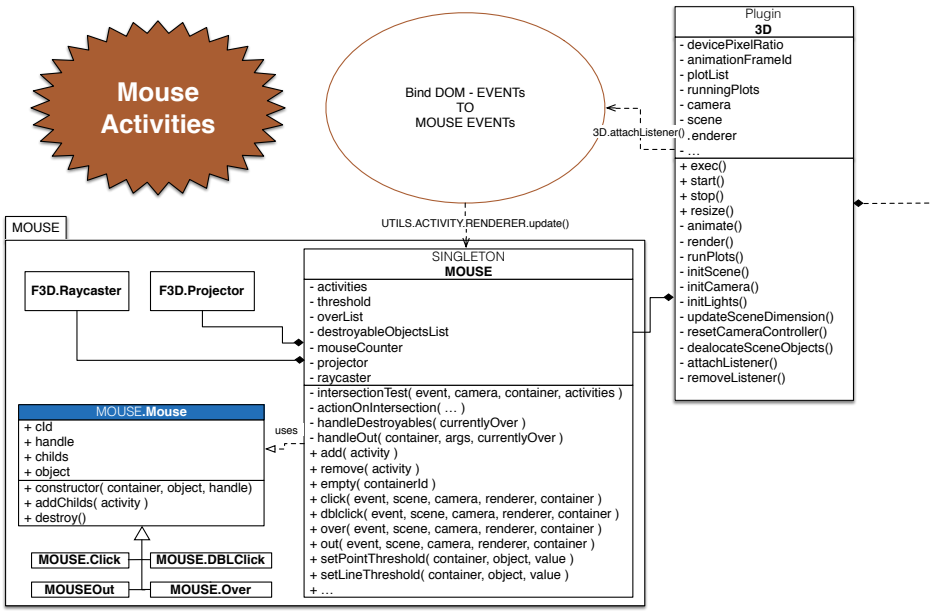


Abbildung 7.24.: Mouse Activities

Mouse-Activities (Abbildung 7.24) werden über das Singleton *MOUSE* gehandhabt. Das Plugin *3D* hält dieses Singleton und bindet *EventListner* eines DOM-Elementes an *MOUSE-Events*.
Wird ein Event ausgelöst, so wird zunächst geprüft, ob alle notwendigen Bedingungen erfüllt sind, um eine Aktivität zu starten. Dies geschieht innerhalb der Methode *intersectionTest(event, camera, container, activities)*. Über die Parameter *event* und *container* werden normalisierte Mauskoordinaten errechnet und mit Hilfe der Kamera-Koordinaten wird ein Strahl erzeugt. Mit diesem Strahl (*ray*) wird geprüft, ob Objekte aus der *activities* Liste geschnitten wurden. Wurde mindestens ein Objekt von diesem Strahl getroffen wird die korrespondierende *Activity* innerhalb der Methode *actionOnIntersection(...)* abgearbeitet.

Implementierte *Activities* sind

- MOUSE.Click,
- MOUSE.DBLClick,
- MOUSE.Over und
- MOUSE.Out.

Jeder *Activity* werden bei der Instanziierung drei Argumente übergeben (*container*, *object* und *handle*). Das Argument *container* bindet eine Aktivität an ein spezielles DOM-Element. Die Funktion *handle* wird aufgerufen, wenn *object* geklickt wurde.
Die Erzeugung einer konkreten *Click-Activity* wird in Code-Ausschnitt 7.14 veranschaulicht. Es handelt sich hierbei um eine sehr einfache Aktivität, welche dem Plugin *Legend* entnommen wurde.

```
1 new UTILS.ACTIVITY.MOUSE.Click(
2  $container,
3  textPlane,
4  function ( o ) {
5   toggleLegend();
6   o.intersection.object.material.color.setHex(...);
7 } );
```

Listing 7.14: Click-Activity aus dem Plugin Legend

Aktivitäten können aber durchaus auch komplexerer Natur sein. Dies betrifft Aktivitäten, welche von Plugins des Typs *Activity* erzeugt werden. An ein solches Plugin können *Animations-Plugins* gehängt werden, welche parallel ausgeführt werden. Nachfolger eines solchen *Animations-Plugins* kann wiederum ein Plugin vom Typ *Animation* sein. Durch dieses Vorgehen können Animations-Ketten erstellt werden, welche sequentiell abgearbeitet werden. Auf eine Animation kann eine andere Maus-Aktivität folgen, welche wiederum andere Animationen auslöst. Dieses komplexe Verhalten wird von speziellen *Handler*-Funktionen gesteuert (z.B.: *UTILS.ACTIVITY.MouseActivityHandler*).

Render Activities

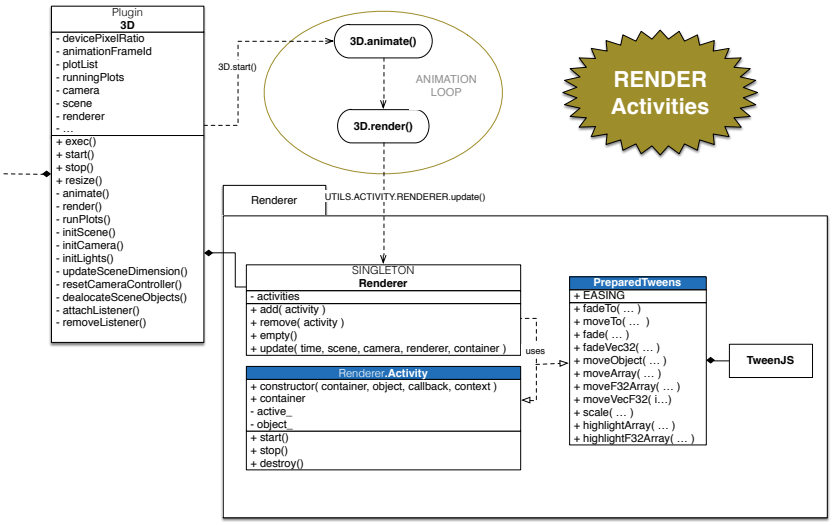


Abbildung 7.25.: Render Activities

Render-Activities unterscheiden sich nur geringfügig von *Mouse-Activities*. Auch hier gibt es ein vom *Plugin 3D* gehaltenes Singleton, welches Aktivitäten steuert. Die Erstellung von konkreten Render-Activities entspricht der von Mouse-Activities (vgl. Beispiele 7.14 und 7.15). Der wesentliche Unterschied liegt einerseits in der Vorbedingung, andererseits auch darin, wo eine Aktivität abgearbeitet wird.
Mouse-Activities werden durch unterschiedliche externe *DOM-Events* ausgelöst, *Render-Activities* werden gestartet oder gestoppt (siehe Beispiel 7.15, Zeile 13).
Die eigentliche Vorbedingung kann eine andere Aktivität sein, das Erreichen eines bestimmten Punktes im Programmfluss (z.B.: Pre-, Ready-, Post-Punkte des Plugins *STATE*) oder allein der Aufruf einer Methode.
Durch den Aufruf der Methode *start()* wird die *handle(e)* Methode einer Aktivität direkt in den Render-Loop involviert und *stop()* entfernt diese wieder.
Der *handle(e)* Methode wird ein Objekt übergeben, welches ein Attribut zur Messung der vergangenen Zeit hält. Dieses Attribut (*time*) wird nach jedem

Durchlauf des Render-Loops aktualisiert. Anhand von unterschiedlichen Zeitmessungen können Werte interpoliert und somit Animationen erstellt werden.
Vordefinierte Animationen werden über die Klasse *PreparedTweens* zur Verfügung gestellt. Sie umfasst Methoden, welche auf das Framework Tween.js (sole, 2015) zurückgreifen und grundlegende Animations-Möglichkeiten bieten.

```
1  var light3 = new F3D.PointLight( 0xffffff, 1.5 );
2
3  context.lightAnimation = new UTILS.ACTIVITY.RENDERER.
       Activity(
4   container.attr( 'id' ),
5   light3,
6   function ( e ) {
7    var offset = 500;
8    e.object.position.x = e.camera.position.x + offset;
9    e.object.position.y = e.camera.position.y + offset;
10   e.object.position.z = e.camera.position.z;
11  }, context );
12
13 context.lightAnimation.start();
```

Listing 7.15: Glanz-Effekt anhand eines PointLights und einer Render-Activity

7.25

8. Integration in ein Content Management System

{v}Plot.js muss in eine Web-Applikation integriert werden, um Visualisierungen innerhalb eines Browsers darstellen zu können. Dies erfordert zumindest rudimentäre Kenntnisse in HTML und Javascript. Aber nicht jeder, der Daten hat und diese grafisch darstellen will, ist auch in der Lage, dies umzusetzen.
Webbasierte **C**ontent **M**anagement **S**ysteme (Lackes, 2015) bieten eine grafische Oberfläche, um die Erstellung und Verwaltung von Inhalten (z.B.: Texte, Bilder, Dateien, et cetera) zu erleichtern. Eine spezielle, viel verwendete Form von CMS sind Blogs. „Blog" ist ein Akronym, welches aus den beiden Wörtern We**B** und **log** zusammen gesetzt wird. Es handelt sich um ein Online-Tagebuch, das von sogenannten Bloggern geführt wird (Kollmann, 2015).

Um den Installationsaufwand auf ein Minimum zu reduzieren und damit eine möglichst breite Zielgruppe anzusprechen, ist es durchaus sinnvoll, {v}Plot in ein bestehendes CMS zu integrieren.

Die Wahl eines geeigneten Systems wurde anhand des Verbreitungsgrades getroffen. Laut dem Unternehmen BuiltWith, welches Marktanalysen im Bereich Webtechnologien betreibt, ist das mit Abstand am meisten eingesetzte CMS **Wordpress** (Rogers und Brewer, 2015) (Abbildung 8.1).

8.1. Wordpress

Wordpress ist ein Open-Source CMS zur Erstellung von Websites, Blogs und Online-Shops. Die Wurzeln sind auf das im Jahr 2001, von Michel

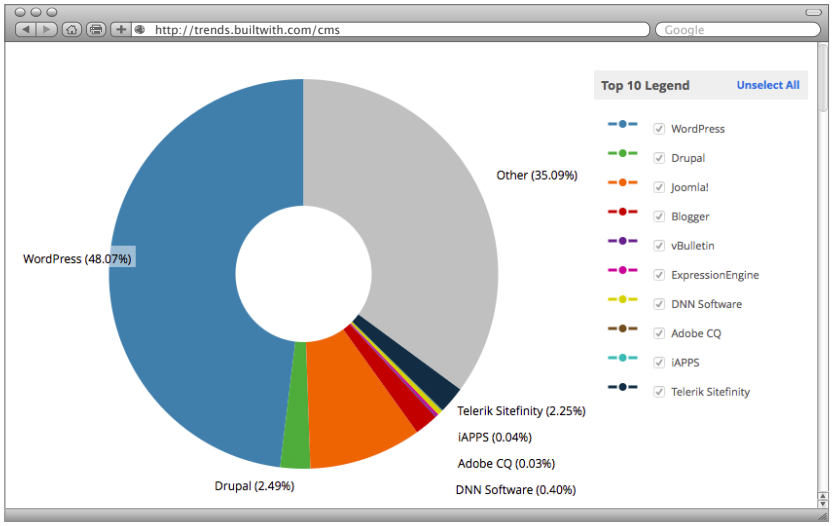


Abbildung 8.1.: Top 10 Content Management Systeme (Rogers und Brewer, 2015)

Valdrighi entwickelte Blog-System *b2/cafelog* zurückzuführen. Mittlerweile ist Wordpress ein vollwertiges Content-Management-System, das von mehr als 60 Millionen Menschen benutzt und von hunderten Entwicklern vorangetrieben wird (Wordpress, 2015).

Plugins erweitern die Kern-Funktionalität von Wordpress (Bondari und Griffiths, 2011).
Ein neues Plugin muss in den Unterordner */wp-content/plugins/* des Wordpress-Verzeichnisses gelegt und über das Backend-Menü „Plugins “ aktiviert werden. Ein Plugin besteht aus mindestens einer *PHP*-Datei, welche drei Elemente aufweist:

1. Einen Header,
2. benutzerdefinierte Funktionen und
3. Hooks.

Ein ***Header*** ist ein Kommentarblock, welcher Informationen in einem standardisierten Format beinhaltet. Dies sind zum Beispiel Plugin-Name, URI, der Name des Autors oder eine Beschreibung. Header sind ein wesentlicher

Bestandteil eines jeden Plugins. Anhand eines Headers wird ein Plugin erst als solches von Wordpress erkannt. Es definiert einen Einstiegspunkt.

Die eigentliche Funktionalität eines Plugins sollte in ***benutzerdefinierte Funktionen***, welche dem Wordpress-Codingstandard entsprechen, gepackt werden. Innerhalb von Wordpress existiert nur ein globaler Namensraum. Da dies zu Konflikten führen kann, wird empfohlen, Plugin-Elemente in folgendem Format zu bezeichnen: „*pluginbezeihnung_elenemtbezeichnung*“.

Hocks binden Funktionen eines Plugins an Events und bestimmen somit, wann eine Funktion ausgeführt wird. Es gibt zwei Möglichkeiten einen Hook zu setzen:

1. add_action(EVENT, FUNCTION) und
2. add_filter(EVENT, FUNCTION).

add_filter erwartet, dass eine gegebene Funktion Input entgegen nimmt und auch etwas retourniert. *add_action* stellt keine Bedingungen an eine gegebene Funktion. Grundsätzlich machen aber beide Hooks dasselbe.

8.2. {v}Plot Wordpress Plugin

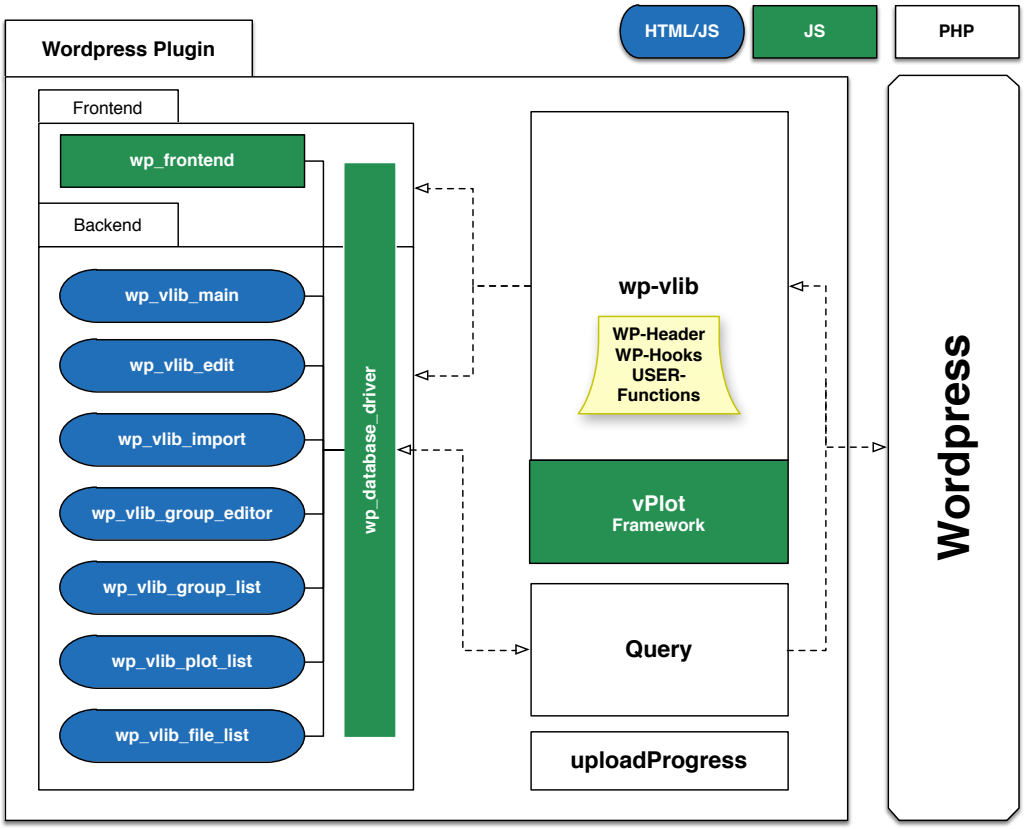


Abbildung 8.2.: Aufbau des WP-Plugins

Eine zentrale Rolle des {v}Plot Wordpress-Plugins bildet die Datei ***wp-vlib.php***. Sie enthält den vorgeschriebenen Header, bindet über *Hooks* Funktionen an Events und erstellt Datenbanktabellen, welche über Methoden der *Query* Klasse verwendet werden.
Das Plugin umfasst Komponenten für den Backend-, sowie den Frontend-Bereich von Wordpress.

8.2.1. Backend

Über ein Backend-Navigationselement können folgende Seiten aufgerufen werden:

1. **Main**
   Main bietet eine Übersicht über die Funktionalität des Plugins. Über Links wird man direkt zu den beschriebenen Komponenten weitergeleitet.

2. **Plot-Editor**
   Der Plot-Editor umfasst alle {v}Plot Module, welche zur Erstellung und Bearbeitung von Templates notwendig sind.

3. **Group-Editor**
   Der Group-Editor enthält Module zur Darstellung, Erstellung und Bearbeitung von Template-Gruppen.

4. **My Plots**
   Hierbei handelt es sich um eine Liste aller erstellten Templates. Zu jedem Template wird ein Vorschaubild, sowie der Name, eine Beschreibung und das Erstellungsdatum angezeigt. Des Weiteren werden drei Buttons zur Verfügung gestellt. „Delete“ löscht das korrespondierende Template. Über „Edit“ wird der *Plot-Editor* geöffnet und das Template geladen und „Post it“ erstellt einen neuen Wordpress-Post, wobei der Texteditor automatisch mit einem *ShortCode* befüllt wird.

5. **My Groups**
   My Groups entspricht *My Plots*, mit dem Unterschied, dass hier Gruppen von Templates aufgelistet werden.

6. **My Files**
   Dieses Menü bietet Informationen über importierte Datensätze. Bei jedem Datensatz kann ausgewählt werden, ob dieser auch im Plot-Editor über File-Plugins angezeigt werden soll.

7. **Import**
   Über ein Formular können hier Datensätze, Templates, sowie Gruppen hochgeladen werden. Unter „Gruppe“ ist hier eine Menge an Datensätzen zu verstehen, welche sich in einem Zip-Archiv befinden. Anhand einer solchen Gruppe kann automatisch eine Template-Gruppe erstellt werden. Falls sich Datensätze innerhalb einer verschachtelten Ordnerstruktur befinden, wird dies in der Template-Gruppe berücksichtigt.

8.2.2. Frontend

Über das Backend von Wordpress werden Seiten, sowie Posts mit Hilfe von Texteditoren erstellt. Diese werden dann im Frontend dargestellt. Ein Plot kann nun in Form von ShortCodes überall, wo auch ein Text-Editor involviert ist, eingebunden werden.

```
1 [vPlot id=460 w=100% h=400px]
```

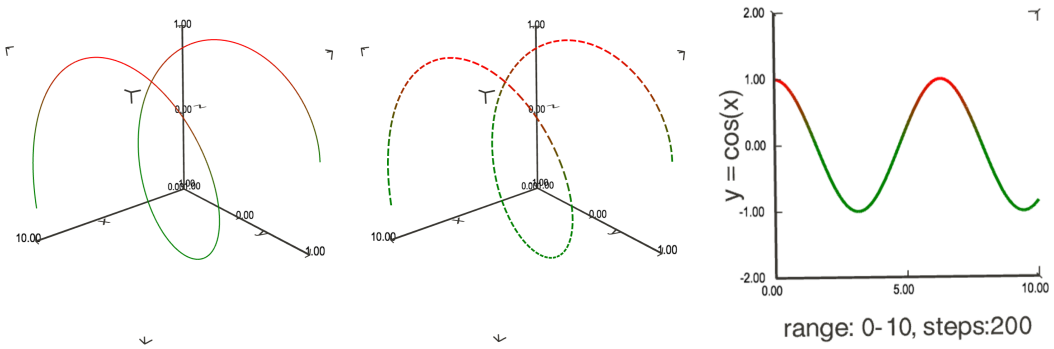
Listing 8.1: WP-ShortCode zur Einbindung eines Plots im Frontend

Code-Ausschnitt 8.1 illustriert einen vPlot-ShortCode. Durch die Aktivierung des {v}Plot Wordpress Plugins wird ein Skript im Frontend involviert, welches nach solchen ShortCodes sucht und diese durch konkrete Plots ersetzt. Anhand des Parameters **id** wird eine Template-Datei geladen. Die optionalen Parameter **w** und **h** bestimmen die Größe des Plot-Moduls, welches eingebunden wird.

9. Visualisierungsmöglichkeiten

Mit dem {v}Plot Framework können Daten in Form von Punkten (Abschnitt 9.2 - ScatterPlot), Linien (Abschnitt 9.1 - LinePlot) und Flächen (Abschnitt SurfacePlot - SurfacePlot) dargestellt werden. Des Weiteren wurde ein 2D/3D BarChart implementiert. In den folgenden Abschnitten werden Beispiele zu den genannten Visualisierungs-Typen geboten.

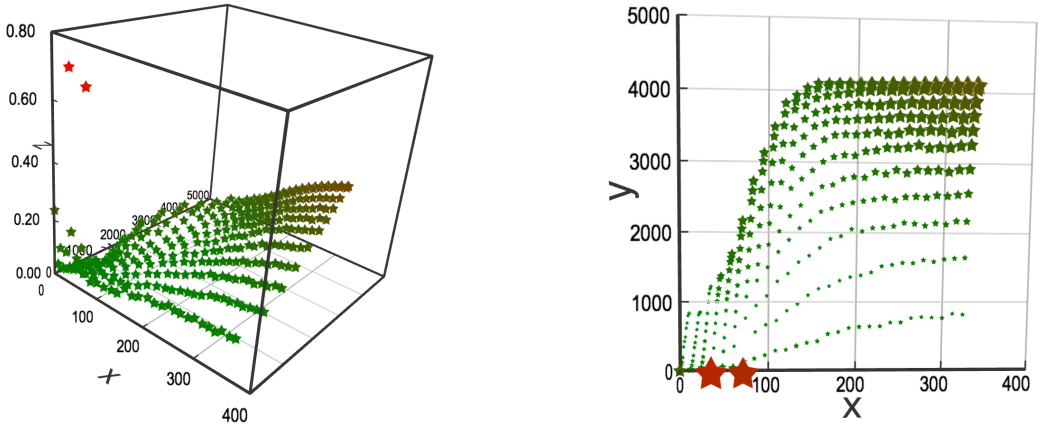
9.1. LinePlot



(a)3D solide (b)3D strichliert (c)2D solide

Abbildung 9.1.: Visualisierungen erstellt mit {v}Plot - LinePlot

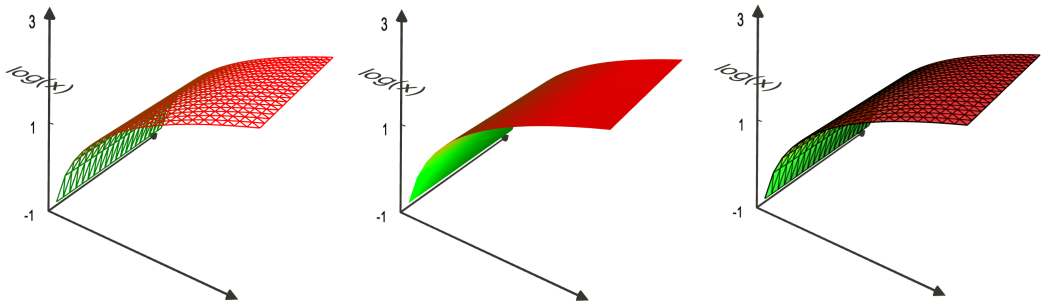
9.2. ScatterPlot



(a)3D (b)2D

Abbildung 9.2.: Visualisierungen erstellt mit {v}Plot - ScatterPlot

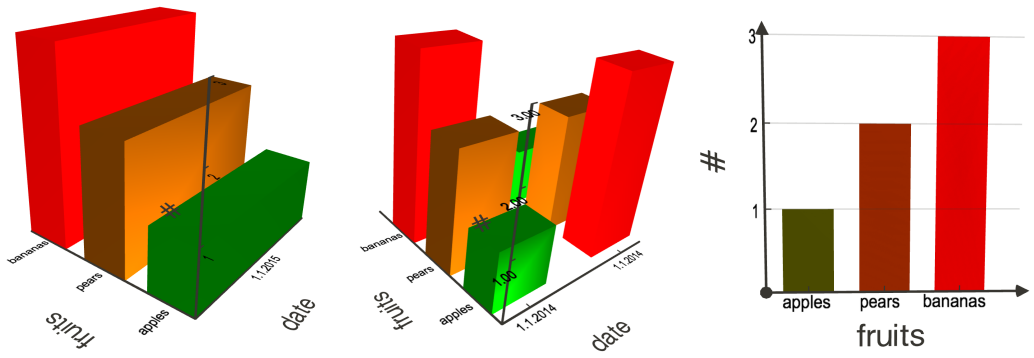
9.3. SurfacePlot



(a)Wireframe Material (b)Basic Material (c)Wireframe+Basic Material

Abbildung 9.3.: Visualisierungen erstellt mit {v}Plot - SurfacePlot

9.4. BarChart



(a) (b) (c)

Abbildung 9.4.: Visualisierungen erstellt mit {v}Plot - BarChartPlot

9.5. Kombination

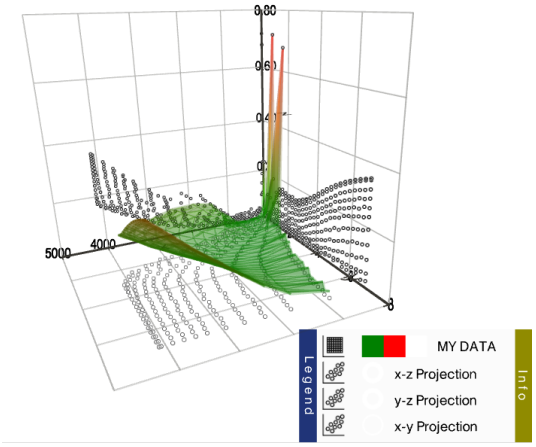


Abbildung 9.5.: Visualisierungen erstellt mit {v}Plot

10. Ausblick

Kann man eigentlich behaupten, dass ein Projekt wirklich fertig ist? Es finden sich nämlich immer Bereiche, welche ausgebaut, umgebaut oder verbessert werden können, um die Bedienbarkeit zu erleichtern beziehungsweise den Verbreitungsgrad zu steigern.
Dieses Kapitel bietet Anregungen, was in Zukunft am Projekt {v}Plot optimiert werden könnte.

10.1. Parser

Konfigurations-Templates (.vlib Dateien) können bereits exportiert und importiert werden. Diese Funktionalität ist in vielerlei Hinsicht vorteilhaft:

- Sicherheitskopien können erstellt werden.
- Eine Migration, bei einem Wechsel auf eine andere Plattform, wird erleichtert.
- Visualisierungen können zwischen Personen ausgetauscht werden, welche {v}Plot verwenden.

Der Export und Import der aktuellen Version ist aber rein auf {v}Plot spezifische Template-Dateien eingeschränkt. In diesem Bereich ergeben sich interessante Erweiterungsfelder.

1. **Import von Daten aus Fremdsystemen**
   Anhand der Dateiendung kann erkannt werden, von welchem Visualisierungssystem eine Datei stammt. Mit Hilfe dieser Information könnte ein Parser erzeugt werden, der den Dateiinhalt analysiert und in ein für {v}Plot verständliches Format übersetzt.

2. **Import von Programmcode**
   Visualisierungen werden für gewöhnlich in einer Programmiersprache mit Hilfe einer Bibliothek erstellt. GnuPlot stellt Interfaces für Perl, Python oder auch Java zur Verfügung. PyPlot ist ein weiteres Beispiel für eine solche Visualisierungs-Bibliothek.
   Um einen Programmcode in ein für {v}Plot verständliches Format zu transformieren, könnte ein Modul entwickelt werden, welches ein HTML-Formular verwaltet. In dieses Formular müssten der Programmcode, die Programmiersprache sowie die verwendete Bibliothek eingetragen werden können. Anhand dieser Informationen könnte dann ein {v}Plot-Template erstellt werden.

3. **Export in ein system-fremdes Format**
   Eine mit {v}Plot erstellte Visualisierung könnte in ein fremdes Format umgewandelt werden. Ein Export von Konfigurations-Dateien für ein fremdes System oder Programmcode für eine gewisse Bibliothek würden die Export-Funktionalität bereichern.

10.2. PMMPEvents: Plugin-Modul-Modul-Plugin Events

Module können miteinander über vordefinierte Kanäle (CHANNELS) kommunizieren. Plugins tauschen Informationen mit Nachfolgern aus und arbeiten gemeinsam an der Lösung einer großen Aufgabe. Eine direkte Kommunikation zwischen Plugins, welche innerhalb *unterschiedlicher* Plot-Module verwendet werden, ist aber noch nicht durchführbar.

Um den Vorsatz der strengen Kapselung nicht zu verletzen, müsste hierfür ein vierstufiger Informationsfluss realisiert werden:

- **P**lugin
  Ein Plugin verwaltet ein Element einer Visualisierung. Wird eine an dieses Element gebundene Aktivität ausgelöst, muss ein Event erzeugt werden können, der alle notwendigen Informationen enthält.

- **M**odul
  Dieser Event muss vom verwaltenden Plot-Modul gefangen und in Form einer Nachricht (message) über einen Mediator-Kanal versendet werden können.
- **M**odul
  Ein Ziel-Modul nimmt diese Nachricht entgegen, entnimmt den übermittelten Event und leitet diesen an ein Ziel-Plugin weiter.
- **P**lugin
  Anhand der übermittelten Informationen könnte somit eine Reaktion innerhalb einer zweiten Visualisierung erfolgen.

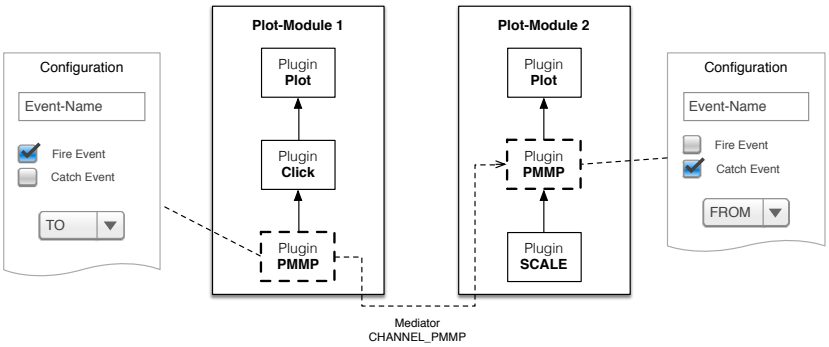


Abbildung 10.1.: Ausblick - Beispielhafter PMMPEvent

Beispiel eins: Ein Plot-Modul könnte eine einfache, zweidimensionale Übersicht eines Datensatzes mit verschiebbaren Reglern überlagern. Mit diesen Reglern könnte ein Auswahlbereich des Datensatzes bestimmt werden, welcher innerhalb eines anderen Plot-Modules in einer 3D-Darstellung angezeigt wird.

Beispiel zwei: Die Änderung der Koordinaten einer Kamera könnte automatisch die Kamera-Position einer andern Visualisierung zur Folge haben.

Beispiel drei: Nachdem eine Animation innerhalb einer Visualisierung abgeschlossen wurde, könnte eine Aktivität in einer andern Visualisierung ausgelöst werden.

Beispiel vier: Nach einem Zustandsübergang (STATE-PRE, STATE-READY, STATE-POST) einer Visualisierung könnte eine Aktivität in einer anderen Visualisierung erfolgen.

Dies sind konkrete Beispiele, welche mit solchen **PMMPEvents** realisiert werden könnten.

Für eine Umsetzung müsste das bereits vorhandene Activity-System (siehe Abschnitt 7.3.6 - Aktivitäten) erweitert werden, sodass eine konkrete Aktivität (zum Beispiel Mouse.Click) an einen *PMMPEvent* gebunden werden kann. Zudem müssten Plugins erstellt werden, welche diese **PMMPEvents** auch verwenden.

10.3. Konstruktionsgitter

Daten können über Plugins manuell eingegeben, aus externen Quellen gelesen und mit Hilfe von Geometrie-Plugins erzeugt werden.
Ein Plugin, welches einem Benutzer erlaubt, Daten anhand eines Mausklicks zu generieren, wäre eine sinnvolle Erweiterung. Das könnte mit einem Konstruktionsgitternetz realisiert werden, wobei dieses über der eigentlichen Visualisierung eingeblendet wird. Sobald mit der Maus ein Schnittpunkt angeklickt wird, kann an dieser Stelle ein neuer Datenpunkt eingefügt werden.

10.4. Weitere Visualisierungsmöglichkeiten

Das {v}Plot Framework wurde mit einem objektorientierten Ansatz umgesetzt. Dies gilt auch für alle implementierten Plot-Plugins. Daten können in Form von Punkten (ScatterPlot), Linien (LinePlot) und Flächen (SurfacePlot)

dargestellt werden. Zusätzlich wurde ein BarChart implementiert. Auf die bereits erstellten Objekte könnte zurückgegriffen werden, um die Liste der Visualisierungsmöglichkeiten zu erweitern.

11. Schlussbemerkung

Mit dem {v}Plot Projekt wird eine ideale Lösung geboten, um Informationen grafisch darzustellen. Animationen können erstellt werden und Interaktionen mit einem Benutzer werden ermöglicht. Programmierkenntnisse sind nicht zwingend erforderlich, da eine Visualisierung einfach via drag&drop zusammengesetzt und konfiguriert werden kann. Daten können auf einfachste Weise importiert beziehungsweise auch automatisch generiert werden. Es können Millionen von Daten in einem Browser dargestellt werden. Ein zusätzliches Browser-Plugin ist nicht notwendig. Da das Framework zusätzlich in ein Wordpress-Plugin eingebettet wurde, wird der Installationsaufwand auf ein Minimum reduziert.

In dieser Arbeit wurde ein hochgradig modular gestaltetes, interaktives Javascript- Framework zu Visualisierung von Daten vorgestellt. Um heutigen sowie zukünftigen Anforderungen gewachsen zu sein, wurde den Aspekten Flexibilität und Erweiterbarkeit große Bedeutung zugemessen. Auch auf hohe Performance wurde großer Wert gelegt. Das Web erwies sich als geeignete Plattform, um ein möglichst breites Publikum zu erreichen. Schnelle Inbetriebnahme sowie einfache Bedienung des Frameworks waren weitere Faktoren, welche bei der Umsetzung beachtet werden mussten.

Im ersten Teil dieser Arbeit (Kapitel 3) war es unumgänglich, sich der Frage, was Visualisierung eigentlich ist, zu widmen, zumal unterschiedliche Definitionen vorlagen und somit die Vorteile aufzulisten waren.
In Kapitel 4 wurde die Relevanz von Legenden und Achsen betont, um die unterschiedlichen Ansätze zur Positionierung von Labels auf Achsen zeigen und vergleichen zu können.
Ebenso wurde mit den verschiedenen Webgrafik-Technologien in Kapitel 5 verfahren, wobei sich zeigte, dass WebGL sehr leistungsstark ist und ohne weitere Plugins verwendet werden kann. Doch der Aufwand, ein Projekt

mit WebGL umzusetzen, ist enorm. Beim Vergleich unterschiedlicher Frameworks erwies sich THREE.js aufgrund seiner leichten und verständlichen Bedienbarkeit und seiner aktiven Community als interessant und wurde somit hervorgehoben.
Meines Erachtens war es auch notwendig, auf den historischen Hintergrund sowie auf das Konzept der Sprache Javascript einzugehen, um in Folge zu erklären, wie Module umgesetzt werden können, obwohl diese nicht direkt unterstützt werden. Der Ausblick auf ECMAScript 6 zeigte schließlich auf, wie in dieser kommenden Version Module verwendet werden könnten.
Dass gleich drei Abschnitte (Kapitel 7 bis 9) dem Framework {v}Plot gewidmet waren, ergab sich daraus, dass die grundlegende Architektur dargelegt sowie die tragenden Komponenten Module, Plugins und Templates erläutert werden mussten. Die Prozesskette der Datenaufbereitung und der Visualisierung ließ sich somit erklären und auch anhand eines konkreten Beispiels veranschaulichen. Die Gründe, warum {v}Plot in ein Wordpress-Plugin eingebettet wurde, wurden klar dargelegt. Somit blieb noch, einen Ausblick auf zukünftige Erweiterungen zu bieten.

Appendix

Anhang A.

Grundlagen - Codebeilagen

A.1. WebGL Programm

```
<html>
<head>
 <title>WebGL Up And Running  Example 1</title>
 <meta http-equiv="content-type" content="text/html; charset=ISO-8859-1">
 <script type="text/javascript">
  function initWebGL(canvas) {
   var gl;
   try
   {
    gl = canvas.getContext("experimental-webgl");
   }
   catch (e)
   {
    var msg = "Error creating WebGL Context!: " + e.toString();
    alert(msg);
    throw Error(msg);
   }
   return gl;
  }
  function initViewport(gl, canvas)
  {
   gl.viewport(0, 0, canvas.width, canvas.height);
  }
  var projectionMatrix, modelViewMatrix;
  function initMatrices()
  {
   // The transform matrix for the square - translate back in Z for the camera
   modelViewMatrix = new Float32Array(
     [1, 0, 0, 0,
      0, 1, 0, 0,
      0, 0, 1, 0,
      0, 0, -3.333, 1]);
   // The projection matrix (for a 45 degree field of view)
   projectionMatrix = new Float32Array(
     [2.41421, 0, 0, 0,
      0, 2.41421, 0, 0,
      0, 0, -1.002002, -1,
      0, 0, -0.2002002, 0]);
  }
  // Create the vertex data for a square to be drawn
  function createSquare(gl) {
   var vertexBuffer;
   vertexBuffer = gl.createBuffer();
   gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
   var verts = [
```

```
46        .5, .5, 0.0,
47        -.5, .5, 0.0,
48        .5, -.5, 0.0,
49        -.5, -.5, 0.0
50       ];
51       gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(verts), gl.STATIC_DRAW);
52       var square = {buffer:vertexBuffer, vertSize:3, nVerts:4, primtype:gl.TRIANGLE_STRIP};
53       return square;
54      }
55      function createShader(gl, str, type) {
56       var shader;
57       if (type == "fragment") {
58        shader = gl.createShader(gl.FRAGMENT_SHADER);
59       } else if (type == "vertex") {
60        shader = gl.createShader(gl.VERTEX_SHADER);
61       } else {
62        return null;
63       }
64       gl.shaderSource(shader, str);
65       gl.compileShader(shader);
66       if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
67        alert(gl.getShaderInfoLog(shader));
68        return null;
69       }
70       return shader;
71      }
72      var vertexShaderSource =
73        " attribute vec3 vertexPos;\n" +
74        " uniform mat4 modelViewMatrix;\n" +
75        " uniform mat4 projectionMatrix;\n" +
76        " void main(void) {\n" +
77        " // Return the transformed and projected vertex value\n" +
78        " gl_Position = projectionMatrix * modelViewMatrix * \n" +
79        " vec4(vertexPos, 1.0);\n" +
80        " }\n";
81      var fragmentShaderSource =
82        " void main(void) {\n" +
83        " // Return the pixel color: always output white\n" +
84        " gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);\n" +
85        "}\n";
86      var shaderProgram, shaderVertexPositionAttribute, shaderProjectionMatrixUniform,
            shaderModelViewMatrixUniform;
87      function initShader(gl) {
88       // load and compile the fragment and vertex shader
89       //var fragmentShader = getShader(gl, "fragmentShader");
90       //var vertexShader = getShader(gl, "vertexShader");
91       var fragmentShader = createShader(gl, fragmentShaderSource, "fragment");
92       var vertexShader = createShader(gl, vertexShaderSource, "vertex");
93       // link them together into a new program
94       shaderProgram = gl.createProgram();
95       gl.attachShader(shaderProgram, vertexShader);
96       gl.attachShader(shaderProgram, fragmentShader);
97       gl.linkProgram(shaderProgram);
98       // get pointers to the shader params
99       shaderVertexPositionAttribute = gl.getAttribLocation(shaderProgram, "vertexPos");
100      gl.enableVertexAttribArray(shaderVertexPositionAttribute);
101      shaderProjectionMatrixUniform = gl.getUniformLocation(shaderProgram, "projectionMatrix");
102      shaderModelViewMatrixUniform = gl.getUniformLocation(shaderProgram, "modelViewMatrix");
103      if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
104       alert("Could not initialise shaders");
105      }
106     }
107     function draw(gl, obj) {
108      // clear the background (with black)
109      gl.clearColor(0.0, 0.0, 0.0, 1.0);
110      gl.clear(gl.COLOR_BUFFER_BIT);
111      // set the vertex buffer to be drawn
112      gl.bindBuffer(gl.ARRAY_BUFFER, obj.buffer);
113      // set the shader to use
114      gl.useProgram(shaderProgram);
115      // connect up the shader parameters: vertex position and projection/model matrices
116      gl.vertexAttribPointer(shaderVertexPositionAttribute, obj.vertSize, gl.FLOAT, false, 0, 0);
117      gl.uniformMatrix4fv(shaderProjectionMatrixUniform, false, projectionMatrix);
118      gl.uniformMatrix4fv(shaderModelViewMatrixUniform, false, modelViewMatrix);
119      // draw the object
```

```
120     gl.drawArrays(obj.primtype, 0, obj.nVerts);
121    }
122    function onLoad() {
123     var canvas = document.getElementById("webglcanvas");
124     var gl = initWebGL(canvas);
125     initViewport(gl, canvas);
126     initMatrices();
127     var square = createSquare(gl);
128     initShader(gl);
129     draw(gl, square);
130    }
131   </script>
132  </head>
133  <body onload="onLoad();">
134  <canvas id="webglcanvas" style="border: none;" width="500" height="500"></canvas>
135  </body>
136  </html>
```

Listing A.1: WebGL Programm zur Erstellung eines weißen Quatrates auf schwarzem Hintergrund.
Der Code wurde von `https://github.com/tparisi/WebGLBook` (Chapter 1) entnommen und adaptiert.

A.2. GLGE XML-Scene

```
1  <?xml version="1.0"?>
2  <glge>
3      <mesh id="square">
4          <positions>
5              1, 1, 0,
6              1, -1, 0,
7              -1, 1, 0,
8              -1, -1, 0
9          </positions>
10         <normals>
11             0, 0, 1,
12             0, 0, 1,
13             0, 0, 1,
14             0, 0, 1
15         </normals>
16
17         <faces>
18             0, 1, 2, 1, 2, 3
19         </faces>
20     </mesh>
21     <material id="squareMat" specular="0" color="#ffffff" />
22     <camera id="camera" />
23     <scene id="scene" camera="#camera" ambient_color="#ffffff">
24         <light id="light" loc_x="0" loc_y="0" loc_z="50"  color="#fff" type="L_POINT" />
25         <object id="square" mesh="#square" material="#squareMat" loc_z="-4" />
26     </scene>
27 </glge>
```

Listing A.2: GLGE XML zur Erstellung eines weißen Quadrates auf schwarzem Hintergrund, wie in Abbildung 5.3.

A.3. PhiloGL

```
<script>
function webGLStart() {
 var square = new PhiloGL.O3D.Model({
  vertices: [
   1,  1, 0,
   -1,  1, 0,
   1, -1, 0,
   -1, -1, 0]
 });

 PhiloGL('canvas', {
  program: {
   from: 'ids',
   vs: 'shader-vs',
   fs: 'shader-fs'
  },
  onError: function() {
   alert("An error ocurred while loading the application");
  },
  onLoad: function(app) {
   var gl = app.gl,
     canvas = app.canvas,
     program = app.program,
     camera = app.camera,
     view = new PhiloGL.Mat4;

   gl.viewport(0, 0, canvas.width, canvas.height);
   gl.clearColor(0, 0, 0, 1);
   gl.clearDepth(1);

   camera.view.id();

   function setupElement(elem) {
    elem.update();
    view.mulMat42(camera.view, elem.matrix);
    program.setBuffers({
     'aVertexPosition': {
      value: elem.vertices,
      size: 3
     }
    });
    program.setUniform('uMVMatrix', view);
    program.setUniform('uPMatrix', camera.projection);
   }

   function drawScene() {
    gl.clear(gl.COLOR_BUFFER_BIT);
    square.position.set(0, 0, -4);

    setupElement(square);
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
   }

   drawScene();
  }
 });
}
</script>
<script id="shader-fs" type="x-shader/x-fragment">
  #ifdef GL_ES
  precision highp float;
  #endif

  void main(void) {
  gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
  }
</script>
<script id="shader-vs" type="x-shader/x-vertex">
    attribute vec3 aVertexPosition;
    uniform mat4 uMVMatrix;
    uniform mat4 uPMatrix;
```

```
72      void main(void) {
73        gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
74      }
75  </script>
```

Listing A.3: PhiloGL Code zum rendern eines weißen Quadrates.
Der Code wurde von `http://www.javascriptoo.com/philogl` übernommen und adaptiert.

Anhang B.

{v}Plot.js - Codebeilagen

B.1. Plugin Struktur

```
define( ['require', './getConfig.vlib', './setConfig.vlib', 'config', 'core/Utils.vlib'],
function ( require, getConfig, setConfig, Config, UTILS ) {

 var Plugin = function () {
  Plugin.superClass.constuctor.call( this, 'PLUGIN_NAME' );
  Plugin.superClass.setContext.call( this, Config.PLUGINTYPE.CONTEXT_3D );
  Plugin.superClass.setType.call( this, /* Config.PLUGINTYPE.XXX */ );
  /** path to plugin-template file * */
  Plugin.superClass.setTemplates.call( this, Config.getPluginPath() + '/templates.html' );
  Plugin.superClass.setIcon.call( this, Config.getPluginPath() + '/icon.png' );
  Plugin.superClass.setAccepts.call( this, {
   predecessors: [/*Config.PLUGINTYPE.XXX*/],
   successors  : [/*Config.PLUGINTYPE.XXX*/]
  } );
  Plugin.superClass.setDescription.call( this, 'DESCRIPTION');

  this.getConfigCallback = getConfig;
  this.setConfigCallback = setConfig;
  /** ********************************************************************* **/
  /** PRIVATE VARIABLES **/
  /** PUBLIC VARIABLES **/
  /** PRIVATE METHODS **/
  /** PUBLIC METHODS **/
  /** ********************************************************************* **/
  this.destroy = function () {
   //TODO: RELEASE
  };

  this.handleConfig = function ( config, env ) {
   //TODO: SET DEFAULTS
  };
  this.handleEnv = function ( env ) {
   //TODO: SET DEFAULTS
  };
  this.exec = function ( config, childs, bufferManager, communication ) {
   this.env = {};
   this.config = config;
   this.childs = childs;
   this.buffer = bufferManager.getBuffer( this.getId() );
   this.env.communication = communication;

   this.handleConfig( this.config, this.env );
   this.handleEnv( this.env );

   Plugin.superClass.handleChilds( this.childs, this.env, function(){
```

```
46      //TODO: ON CHILDS DONE...
47     },this );
48
49     /* Response file according to PTD */
50     var result = {
51      //TODO
52     };
53     var response = {
54      pId      : this.getId(),
55      pType    : this.type,
56      response: result
57     };
58
59     return response;
60    };
61   };
62   UTILS.CLASS.extend( Plugin, AbstractPlugin );
63   /** ********************************* */
64   /** PROTOTYPES      **/
65   /** ********************************* */
66
67   return Plugin;
68  } );
```

Listing B.1: Struktur eines Plugins

Literatur

acunetix (Okt. 2009). *Popularitätsaufschwung von Javascript*. URL: http://www.acunetix.com/blog/articles/increasing-popularity-javascript/ (siehe S. 42).

Adobe (2006). *Developing Acrobat Applications Using JavaScript*. URL: http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/js_developer_guide.pdf (siehe S. 43).

Adobe (2014). *Adobe Flash Professional*. URL: https://helpx.adobe.com/de/pdf/flash_reference.pdf (siehe S. 21).

Andrews, Keith (2015). *Information Visualisation*. http://courses.iicm.tugraz.at/ivis/ivis.pdf (siehe S. 7, 8, 10).

Ashkenas, Jeremy (Feb. 2015). *underscore.js*. URL: http://underscorejs.org/ (siehe S. 85).

Belmonte, Nicolas Garcia (2012). *PhiloGL*. URL: http://www.senchalabs.org/philogl/ (siehe S. 37).

Bidelman, Eric (2011). *Transferable Objects: Lightning Fast!* URL: http://updates.html5rocks.com/2011/12/Transferable-Objects-Lightning-Fast (siehe S. 98).

Bondari, Brian und Everett Griffiths (2011). *WordPress 3 Plugin Development Essentials*. PACKTpub. ISBN: 978-1-849513-52-4 (siehe S. 122).

Brantly, James (Feb. 2015). *jabble*. URL: https://github.com/jbrantly/yabble (siehe S. 66).

Bruls, Mark, Kees Huizing und Jarke J. van Wijk (o.D.). *Squarified Treemaps*. URL: https://graphics.ethz.ch/teaching/scivis_common/Literature/squarifiedTreeMaps.pdf (siehe S. 30).

Brunt, Paul (2010). *GLGE: WebGL for the lazy*. URL: http://www.glge.org/about/ (siehe S. 38).

bucephalus.org (2013). *The HTML5 Canvas Handbook*. URL: http://bucephalus.org/text/CanvasHandbook/CanvasHandbook.html (siehe S. 24).

Burke, James (Feb. 2015). *requireJS*. URL: http://requirejs.org/ (siehe S. 66).

Chapman, Stephen (o.D.). *A Brief History of Javascript*. URL: http://javascript.about.com/od/reference/a/history.htm (siehe S. 41).

Chen, Min (2013). *What is Visualization Really for?* http://arxiv.org/ftp/arxiv/papers/1305/1305.5670.pdf (siehe S. 5).

Collins (2005). *English Dictionary - Definition of Visualisation*. URL: http://www.collinsdictionary.com/dictionary/english/visualization (siehe S. 5).

CommonJS (2015a). *CommonJS*. URL: http://www.commonjs.org/ (siehe S. 63).

CommonJS (Feb. 2015b). *Module Identifieres*. URL: http://wiki.commonjs.org/wiki/Modules/1.1.1#Module_Identifiers (siehe S. 65).

CommonJS (Feb. 2015c). *Specs: Modules 1.0*. URL: http://www.commonjs.org/specs/modules/1.0/ (siehe S. 63, 64).

Correia, Pedro (Apr. 2014). *AMD API Specification*. URL: https://github.com/amdjs/amdjs-api/blob/master/AMD.md (siehe S. 65).

Crockford, Douglas (o.D.). *Prototypal Inheritance in JavaScript*. URL: http://javascript.crockford.com/prototypal.html (siehe S. 45, 55).

Dagoor, Kevin (Jan. 2015). *What Server Side JavaScript needs*. URL: http://www.blueskyonmars.com/2009/01/29/what-server-side-javascript-needs/ (siehe S. 42, 63).

Dahl, Ryan (Okt. 2009). *node.js*. URL: http://nodejs.org/ (siehe S. 42).

Deveria, Alexis (2015). *Can I Use WebGL*. URL: http://caniuse.com/#search=webgl (siehe S. 26).

Ecma (2011). *ECMA-262 Standard*. URL: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf (siehe S. 46, 47, 55, 56).

Ecma (2013). *ECMA-404 Standard*. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (siehe S. 78).

Eisenberg, J. David (2002). *SVG Essentials*. First. OReilly. ISBN: 0-596-00223-8 (siehe S. 22, 23).

Flanagan, David (2011). *Javascript The Definitive Guide*. Sixth. OReilly. ISBN: 9780596805524 (siehe S. 47, 55).

Glassner, Andrew S. (2013). »Graphics GEMS«. In: 1. Aufl. ACADEMIC PRESS LIMITED. Kap. 2.2, S. 61–63. ISBN: 978-0122861666 (siehe S. 13).

Google-Data-Aris-Team (2015). *Chrome Experiment - World Population*. URL: http://globe.chromeexperiments.com/ (siehe S. 10).

Gratzer, Christine (Feb. 2007). *VRML-Box*. URL: http://vrml-box.yettie.at (siehe S. 21).

Graves, Scott M. (Mai 1995). *VRML's Roots*. URL: http://home.southernct.edu/~gravess1/vrml-roots.html (siehe S. 21).

Hann, John (Feb. 2015). *curl.js*. URL: https://github.com/cujojs/curl (siehe S. 66).

Harmez, Ross und Dustin Diaz (2008). *Javascript Objektorientierung und Entwurfsmuster*. Franzis. ISBN: 9783772364884 (siehe S. 43, 60).

Heise (2013). *JavaScript für Mikrocontroller*. URL: http://www.heise.de/newsticker/meldung/JavaScript-fuer-Mikrocontroller-1934018.html (siehe S. 43).

Helic, Denis und Roman Kern (2014). *Knowledge Discovery and Data Mining 1 (VO) (707.003)*. http://kti.tugraz.at/staff/denis/courses/kddm1/intro.pdf (siehe S. 6).

Helma (2005). *Helma - A Server-Side Javascript Environment*. URL: http://dev.helma.org/ (siehe S. 42).

Herman, David und Sam Tobin-hochstadt (o.D.). *DRAFT Modules for JavaScript Simple, Compilable, and Dynamic Libraries on the Web* (siehe S. 66–68).

Hullman, Jessica, Eytan Adar und Priti Shah (2011). *Visualization in Scientific Computing*. http://www.sci.utah.edu/vrc2005/McCormick-1987-VSC.pdf (siehe S. 5).

Johnson, Donald W. (o.D.). *A Scalability Study of Web-Native Information Visualization* (siehe S. 30).

Kantor, Ilya (2011). *Coordinates*. URL: http://javascript.info/tutorial/coordinates (siehe S. 24).

Kay, Lindsay (2015). *SceneJS*. URL: http://scenejs.org (siehe S. 36).

Khronos-Group (März 2011). *Khronos Releases Final WebGL 1.0 Specification*. URL: https://www.khronos.org/news/press/khronos-releases-final-webgl-1.0-specification (siehe S. 26).

Kirk, Andy (2012). *Data Visualization: a successful design process*. 1. Aufl. PACKt. ISBN: 978-1-84969-346-2 (siehe S. 19).

Kollmann, Tobias (2015). *Definition - Blog*. URL: http://wirtschaftslexikon.gabler.de/Definition/blog.html (siehe S. 121).

Kulaga, Autumn und Semay Johnston (2015). *WebGL Frameworks * A Research Blog*. URL: http://webglframeworks.org/framework-documentation/framework-list/ (siehe S. 32).

Lackes, Richard (2015). *Definition - CMS*. URL: http://wirtschaftslexikon.gabler.de/Definition/content-management-system-cms.html (siehe S. 121).

McCormick, Bruce H. (1987). *Visualization in Scientific Computing*. http://www.sci.utah.edu/vrc2005/McCormick-1987-VSC.pdf (siehe S. 5).

MDN (Nov. 2014). *JavaScript typed arrays*. URL: https://developer.mozilla.org/de/docs/Web/JavaScript/Typed_arrays (siehe S. 34).

Microsoft (2011). *WebGL Considered Harmful*. URL: http://blogs.technet.com/b/srd/archive/2011/06/16/webgl-considered-harmful.aspx (siehe S. 26).

Miguel, Ricardo Cabello (2015). *Three.js*. URL: http://threejs.org/ (siehe S. 33).

Moot, Kevin (Juli 2012). *Handling user input in HTML5 Canvas-based games*. URL: http://www.ibm.com/developerworks/library/wa-games/ (siehe S. 24).

Mozilla (Jan. 2015). *ECMAScript 6 support in Mozilla*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_6_support_in_Mozilla (siehe S. 66).

Murray, Scott (2013). *Interactive Data Visualization for the Web*. 1. Aufl. O'Reilly. ISBN: 978-1-449-33973-9 (siehe S. 5, 19).

Oracle (o.D.). *Java Plug-in and Applet Architecture*. URL: http://docs.oracle.com/javase/7/docs/technotes/guides/jweb/applet/applet_execution.html (siehe S. 19).

Oracle (2014). *JavaFX*. URL: http://docs.oracle.com/javase/8/javase-clienttechnologies.htm (siehe S. 20).

oracle (1998). *Server-Side JavaScript Guide*. URL: http://docs.oracle.com/cd/E19957-01/816-6411-10/contents.htm (siehe S. 42).

Osmani, Addy (2012). *Learning JavaScript Design Patterns*. First. OReilly, S. 131–152. ISBN: 978-1-449-33181-8 (siehe S. 62, 69).

Owen, G. Scott (2005). *HyperVis - Definitions, History, and Goals of Visualization*. URL: http://www.siggraph.org/education/materials/HyperVis/visgoals/definiti.htm (siehe S. 5).

Parisi, Toni (2012). *WebGL Up And Running*. First. OReilly. ISBN: 978-1-449-32357-8 (siehe S. 26, 33).

Resing, John (Nov. 2007). *The World of ECMAScript*. URL: http://ejohn.org/blog/the-world-of-ecmascript/ (siehe S. 41, 46).

Rogers, Andrew und Gary Brewer (2015). *Statistics for websites using CMS technologies*. URL: http://trends.builtwith.com/cms (siehe S. 121, 122).

Schneider, Daniel K. und Sylvere Martin-Michiellot (1998). *VRML Primer and Tutorial*. URL: http://tecfa.unige.ch/guides/vrml/vrmlman/vrmlman.html (siehe S. 21).

Shah, Aadit M (Mai 2013). *Why Prototypal Inheritance Matters*. URL: http://aaditmshah.github.io/why-prototypal-inheritance-matters/#constructors_vs_prototypes (siehe S. 55).

sole, sole (Feb. 2015). *tween.js*. URL: https://github.com/tweenjs/tween.js/tree/master (siehe S. 119).

stackoverflow (2015). *StackOverflow - Three.js*. URL: http://stackoverflow.com/questions/tagged/three.js (siehe S. 35).

Talbot, J., S. Lin und P. Hanrahan (Nov. 2010). »An Extension of Wilkinson's Algorithm for Positioning Tick Labels on Axes«. In: *Visualization and Computer Graphics, IEEE Transactions on* 16.6, S. 1036–1043. ISSN: 1077-2626. DOI: 10.1109/TVCG.2010.130 (siehe S. 17, 114).

Tapia, Guido (2010). *How to prevent 'Stop running this script' message in browsers*. URL: http://www.picnet.com.au/blogs/guido/post/2010/03/04/how-to-prevent-stop-running-this-script-message-in-browsers/ (siehe S. 107).

TBideas (2013). *The easiest way to program your Raspberry Pi!* URL: http://pijs.io/ (siehe S. 43).

techslides (2013). *HTML5 Game Engines and Frameworks*. URL: http://techslides.com/html5-game-engines-and-frameworks (siehe S. 32).

TIOBE (Jan. 2015). *TIOBE Index*. URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (siehe S. 42).

VRML-Consortium (1997). *VRML97 ISO/IEC-14772-1*. URL: http://www.bitmanagement.com/developer/spec/vrml97specification.pdf (siehe S. 21).

W3C (2011). *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. URL: www.w3.org/TR/SVG/struct.html#DOMInterfaces (siehe S. 23).

W3C (2015). *Css3*. URL: http://www.w3schools.com/css/css3_intro.asp (siehe S. 20).

Ware, Colin (2004). *Information Visualization Perception of Design*. 2. Aufl. Elsevier - Morgan Kaufmann Publishers. ISBN: 1-55860-819-2 (siehe S. 5, 7–9).

Welch, Chris (2013). *Microsoft reportedly bringing WebGL support to Internet Explorer 11*. URL: http://www.theverge.com/2013/3/30/4165204/microsoft-bringing-webgl-support-internet-explorer-11-windows-blue (siehe S. 26).

Wikipedia (Jan. 2015a). *Anscombe-Quartett*. URL: http://en.wikipedia.org/wiki/Anscombe's_quartet (siehe S. 6).

Wikipedia (2015b). *Parallele Koordinaten*. URL: http://de.wikipedia.org/wiki/Parallele_Koordinaten (siehe S. 30).

Wilkinson, Leland (2005). *The Grammar of Graphics*. 2. Aufl. Springer. ISBN: 978-0387-24544-7 (siehe S. 14).

Wordpress (2015). *Wordpress.org*. URL: https://wordpress.org (siehe S. 122).

Young, Alex (2010). *History of JavaScript: Part 1*. URL: http://dailyjs.com/2010/05/24/history-of-javascript-1/ (siehe S. 41).