FluidDiagrams: A Cross-Platform, Web-Based Information Visualisation Framework using JavaScript and WebGL

Master's Thesis

Benedict Wright

FluidDiagrams: A Cross-Platform, Web-Based Information Visualisation Framework using JavaScript and WebGL

Master's Thesis

at

Graz University of Technology

submitted by

Benedict Wright

Institute for Information Systems and Computer Media (IICM),
Graz University of Technology
A-8010 Graz, Austria

24 January 2014

Advisor: Ao.Univ.-Prof. Dr. Keith Andrews

FluidDiagrams: A Cross-Platform, Web-Based Information Visualisation Framework using JavaScript and WebGL

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Benedict Wright

Institut für Informationssysteme und Computer Medien (IICM),
Technische Universität Graz
A-8010 Graz

24. Januar 2014

Diese Arbeit ist in englischer Sprache verfasst.

Begutachter: Ao.Univ.-Prof. Dr. Keith Andrews

Abstract

Information visualisation is the process of transforming data and information into a graphical representation. Visualisation helps the human mind understanding and interacting with large data sets. The aim of this thesis was to create an information visualisation framework using WebGL as its rendering engine. The benefit of using WebGL is to take advantage of the computing power of any installed graphics hardware increase the performance of visualisations.

Creating web-based interactive information visualisations can be very cumbersome when not using specialised libraries and toolkits. This thesis first analyses current best practice when creating general web-based applications using JavaScript. The second part looks at current technologies for creating web-based graphics, and using short examples, shows the usage and benefits of current JavaScript libraries. Four existing information visualisation toolkits, JIT, D3, Aperture, and Highcharts are discussed, before introducing FluidDiagrams.

FluidDiagrams is a web-based information visualisation framework which uses WebGL for rendering the graphics to the browser. FluidDiagrams was created during this thesis and is based on Three.JS which provides WebGL, Canvas, and SVG rendering engines, depending on the technologies supported by the browser and the underlying operating system and hardware. This enables faster and richer visualisations, since the rendering process is shifted to the graphics card wherever possible. FluidDiagrams was used during the course Information Visualisation [706.057] in SS2013 at Graz University of Technology, where multiple visualisations for FluidDiagrams were created. Finally, this thesis gives a brief outlook into possible adaptations and refinements to the FluidDiagrams framework.

Kurzfassung

Informationsvisualisierung ist der Prozess, Daten und Informationen in eine graphische Darstellung umzuwandeln. Visualisierung hilft dem Menschen große Datensätze zu verstehen, und mit ihnen zu interagieren. Das Ziel dieser Arbeit war es, ein Informationsvisualisierungsframework zu erstellen, welches WebGL für den render Prozess verwendet. Der Vorteil von WebGL ist es, die Rechenleistung von jeglicher installierter Graphik-Hardware zu nutzen, um die Performance von Visualisierungen zu verbessern.

Web-basierte Informationsvisualisierungen zu erstellen, kann sehr mühsam sein, wenn nicht spezialisierte Bibliotheken zum Einsatz kommen. Diese Arbeit analysiert zuerst aktuelle bewährte Methoden, zur Erstellung von web-basierten Applikationen mittels JavaScript. Der zweite Teil untersucht aktuelle Technologien zur Erstellung von web-basierten Graphiken. Danach werden, anhand von kurzen Beispielen, die Verwendung und der Vorteil von modernen JavaScript Graphik Bibliotheken gezeigt. Vier existierende Informationsvisualisierungstoolkits, JIT, D3, Aperture und Highcharts werden diskutiert.

FluidDiagrams ist ein web-basiertes Informationsvisualisierungsframework, welches WebGL für den render Prozess verwendet. FluidDiagrams wurde während dieser Arbeit erstellt, und basiert auf Three.JS. Three.JS stellt WebGL, Canvas und SVG rendering Engines zur Verfügung. Abhängig von den unterstützten Technologien des Webbrowsers, dem Betriebssystem und der zur Verfügung stehenden Graphik-Hardware, wird aus diesen drei rendering Engines gewählt. Dies ermöglicht es, schnellere und aufwendigere Visualisierungen zu erstellen, da der render Prozess, wenn möglich, auf die Graphikkarte ausgelagert wird. FluidDiagrams kam während der Lehrveranstaltung Information Visualisation [706.057] im SS2013 an der Technischen Universität Graz zum Einsatz. Während dieser Lehrveranstaltung wurden einige Visualisierungen für FluidDiagrams erstellt. Abschließend, gibt diese Arbeit einen kurzen Ausblick auf die zukünftige Entwicklung von FluidDiagrams.

Eidesstattliche Erklärung

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 entnommene Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

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 marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

I would like to thank the following people for their help and support during the creation of this thesis.

First of all, I would like to express my gratitude to my supervisor Keith Andrews, for introducing me to the topic of information visualisation, and supporting me in every possible way during the development and writing phases of this thesis.

I want to thank all the participants of the course: Information Visualisation [706.057] in SS2013 at Graz University of Technology, for their feedback and their constructive work during the practical part of this thesis.

Furthermore, I want to thank my colleagues at the Institute for Software Technology for the feedback and the discussions which provided valuable input during the writing of the thesis and the development of FluidDiagrams.

I also want to thank all of my friends, and Helena Aspernig in particular, for being there throughout the ups and downs of my study.

Finally, a very special thanks goes to my parents who supported me mentally and financially during the years of my study.

Benedict Wright
Graz, Austria, January 2014

Credits

I would like to thank the following individuals and organisations for permission to use their material:

- Figure 2.1[a] is taken from Wikipedia [2006]. used under the Creative Commons Attribution-Share Alike 3.0 Unported license.
- Figure 2.1[b] is used with kind permission of Keith Andrews.
- Figure 2.1[c] is taken from Patokallio [2012] and is used with kind permission of Jani Patokallio.
- Figure 2.2 is taken from Wikipedia [2010], where it is published under the Creative Commons Attribution-Share Alike 3.0 Unported license.
- Figure 2.3 was extracted from [Fertig, Freeman, and Gelernter, 1996] and is used under the terms of the ACM copyright Notice found on Page xii.
- Figure 2.4 was extracted from Lamping, Rao, and Pirolli [1995] and is used under the ACM copyright Notice found on Page xii.
- Figure 2.5 is a screenshot taken by the author from the demo software on the original web site [Heer, 2010].
- Figure 2.7 was extracted from the InfoVis lecture notes [Andrews, 2013] and is used with kind permission of Keith Andrews .
- Figure 4.4 was redrawn from "Professional WebGL Programming: Developing 3D Graphics for the Web" by Andreas Anyuru.
- Figure 8.1 was adapted by the author from the original in "Interacting with Huge Hierarchies: Beyond Cone Trees" by Carrière and Kazman [1995].
- Figures C.1 and C.2 were adapted from "Computergraphic: Lecture Slides" by Henning Wenk.
- Figures 8.6, 8.5, and 8.4 were created during the course: Information Visualisation [706.057] at Graz University of Technology. Three groups provided these visualisations.
  - Group 1 (Scatterplot): Mahmoud Diab, Mark Dokter, Lena Hatbauer,and Jan Rocnik
  - Group 2 (Bar Chart): David Kikelj, Daniel Krenn, and Jakob Strauss.
  - Group 3 (Line chart): Maja Savanovic, Ari Rauhala, and Christine Pichler.

ACM Copyright Notice

Chapter 1

Introduction

The next chapter (Chapter 2) of this thesis gives an overview of visualisation, and explains the differences between information, scientific, and geographic visualisation. The focus is on information visualisation, its uses and its sub fields, depending on the type on input data. The chapter demonstrates the necessity of graphical representation of data, using the classic example of Anscombe's Quartet. Then an overview of the five subcategories of information visualisation is given, together with a short example of each of them. These categories are: linear data, hierarchies, networks and graphs, multidimensional meta data, and feature spaces.

Chapter 3 gives an introduction to web development using JavaScript, with special focus on web-based graphics and information visualisation. First, a general survey on modern software development with JavaScript is presented. The general development principles for creating web content using JavaScript are presented. These principles include the document object model (DOM), reacting to events such as mouse clicks. Next, the usage of standard libraries is discussed. For this purpose, the industry standard jQuery library is introduced, showing how it helps to develop readable, maintainable code. Then, JavaScript supersets are discussed, which provide a means of further increasing the readability and maintainability of the code, by providing additional functionality, such as classes, modules, and interfaces. Ways of testing JavaScript applications, using the Jasmine unit test framework, are discussed. Finally an overview of development best practice is given, discussing coding standards, design patterns, and writing testable code.

The third part of the thesis (Chapters 4–5) discusses web-based graphics, using modern technologies such as HTML5, SVG, WebGL, and Flash. These chapters show the necessity of using specialised libraries to create graphics in the web, which are described in more detail in the following section. In Chapter 5, open source libraries for creating 2d and 3d graphics are presented. The first two libraries discussed are for creating 2d content. EaselJS, uses the HTML5 canvas element for drawing content, Raphaël uses SVG to create graphics embedded in the document object model of the web page. The third library discussed, Three.JS, is used to create web-based 3d graphics using WebGL.

Chapter 6 then discusses specialised open source information visualisation toolkits, by giving an introduction to two existing frameworks. The JavaScript infoVis Toolkit (JIT) provides methods of creating interactive Canvas visualisations. D3 provides, means of creating data driven documents of any XML form, making it possible to create interactive SVG visualisations, HTML pages, or any other XML document.

Chapters 7–10 then present the practical development work done as a part of this thesis. In Chapter 7, the FluidDiagrams framework is introduced. FluidDiagrams is a web-based information visualisation framework, and distinguishes itself from existing frameworks, through its use of WebGL, which enables the toolkit to use the computing power of the graphics card. The use of WebGL was made possible by using the existing Three.JS rendering engine. Its main features, the modularity and interchangeability of its components, are described. Then the architecture of the framework is presented, and its advan-

tages and disadvantages discussed. The suite of visualisations, currently implemented are presented in Chapter 8. These include:

- Cone Tree: A hierarchy visualisation (3d).
- Parallel Coordinates: A multi-dimensional meta data visualisation (2d).
- Bar Chart: A visualisation for discrete categorised data (2d).
- Line Chart: A visualisation for sample data of a continuous process (2d).
- Scatter Plot: A multi-dimensional meta data visualisation (2d).
- Hyperbolic Browser: A hierarchy visualisation, using hyperbolic geometry (2d).

Chapter 9 then discusses some selected details of implementation. These are parts of special interest, or functions that proved to be harder to implement. The most prominent of these is the click event, which requires the detection of geometry at the position of the mouse at the time of the event.

Finally, Chapter 10 outlines some ideas for future work regarding FluidDiagrams.

Chapter 2

Information Visualisation

Visualisation provides a graphical means for transforming data and information so that the human mind can more easily interact with large sets of data. It enhances the human ability to discover characteristics, patterns and trends in data [Gershon, Eick, and Card, 1998]. The field of visualisation can be split in to three subfields. in Figure 2.1:

- *SciVis*: Scientific visualisation (SciVis) focuses on the graphical representation of biological and physical data. This includes medical data such as CAT scans and simulations such as flows over surfaces. The visual representation is usually suggested by the data itself. A series of CAT scans can only be represented in a very limited way, either as a series of images or as a 3d model.

- *GeoVis*: Geographic visualisation (GeoVis) aims at displaying information relative to a geographic location. This is usually done by overlaying information over a 2d or 3d map. Again the visual representation of the data is suggested by the data itself, since it makes most sense to provide the information positioned on a map.

- *InfoVis*: Information visualisation (InfoVis) is the graphical representation of *abstract* data, such as lists, hierarchies, networks, and multi-dimensional data. A visual representation must be carefully chosen or constructed.

The three subfields are illustrated in Figure 2.1. All three subfields can of course be combined to gain further insight into certain data. For instance, one could provide bar charts on a map representing election results. The combination of InfoVis and GeoVis is also called Data visualisation or DataVis [Andrews, 2013].

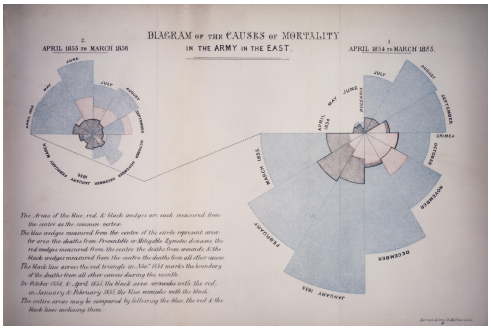
A major role in any visualisation in addition to the graphical representation is the user interaction. Some method of data filtering and/or highlighting points of interest is usually required. Additionally, navigation through large data sets, is a necessity. However, the interaction model must correspond to the actual visualisation and has to be developed in conjunction with the graphical representation.

Information visualisation is a combination of interaction design and graphics, with a foundation in visual perception. In contrast to GeoVis and SciVis, the graphical representation is not inherent in the data. This means there is no clear and obvious physical representation. Visual representations of the data have to be carefully designed, together with any required interaction methods [Gershon, Eick, and Card, 1998].

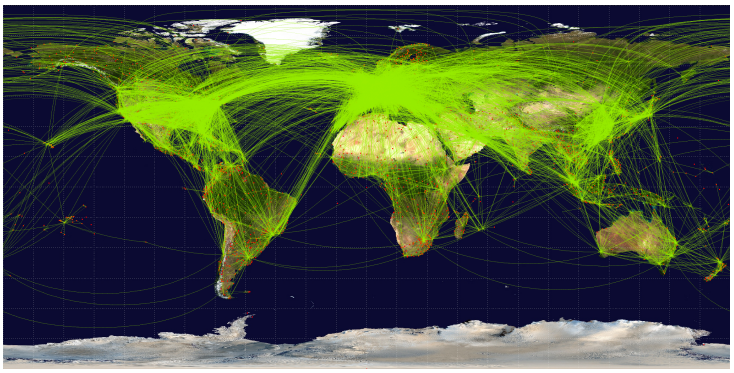
The goal of InfoVis is to support the users in making sense of abstract data. According to Andrews [2013] graphical representations of data make it easier to :

- See trends and patterns.

(a) SciVis: 3d representation of a CAT scan. Image extracted from Wikipedia [2006] and is used under the Creative Commons Attribution-Share Alike 3.0 Unported license.



(b) InfoVis: Polar Area Diagram by Nightingale [1858], showing causes of mortality in the army. The area of a wedge represents the number of deaths. Image used with kind permission of Keith Andrews.



(c) GeoVis: Visual representation of airline routes. Used with kind permission of Patokallio [2012].

Figure 2.1: Examples for the three visualisation subfields: SciVis, GeoVis, and InfoVis.

- Compare data.
- Navigate through large data sets.

Anscombe's Quartet [Anscombe, 1973] is a classic example that shows the importance of graphical representation of data. Anscombe's Quartet consists of 4 data sets, each of eleven data points [x,y]. Each one of the sets has the same mean values $mean(x) = 9.0, mean(y) = 7.5$ and the same standard deviation of $sd(x) = 3.3166, sd(y) = 2.0316$, as can be seen in Table 2.1. However when plotted as 2d graphs, the difference between the data sets becomes clear, as can be seen in Figure 2.2.

Information visualisations can be divided into five sub-categories depending on the type of input data [Andrews, 2013]:

- *Linear*: Linear visualisations are used to display data such as alphabetic lists, chronological lists, and program source code. Lifestreams is an example of a chronological visualisation of documents, shown in Figure 2.3. It displays old documents at the tail of the stream, while the beginning of the stream represents current or future documents such as to-do lists and reminders. One major feature is that streams can be filtered to sub-streams enabling faster navigation through the documents.

	v_1		v_2		v_3		v_4	
	x_1	y_1	x_2	y_2	x_3	y_3	x_4	y_4
	10.00	8.04	10.00	9.14	10.00	7.46	8.00	6.58
	8.00	6.95	8.00	8.14	8.00	6.77	8.00	5.76
	13.00	7.58	13.00	8.74	13.00	12.74	8.00	7.71
	9.00	8.81	9.00	8.77	9.00	7.11	8.00	8.84
	11.00	8.33	11.00	9.26	11.00	7.81	8.00	8.47
	14.00	9.96	14.00	8.10	14.00	8.84	8.00	7.04
	6.00	7.24	6.00	6.13	6.00	6.08	8.00	5.25
	4.00	4.26	4.00	3.10	4.00	5.39	19.00	12.50
	12.00	10.84	12.00	9.13	12.00	8.15	8.00	5.56
	7.00	4.82	7.00	7.26	7.00	6.42	8.00	7.91
	5.00	5.68	5.00	4.74	5.00	5.73	8.00	6.89
mean	9.00	7.50	9.00	7.50	9.00	7.50	9.00	7.50
sd	3.3166	2.0316	3.3166	2.0316	3.3166	2.0316	3.3166	2.0316

Table 2.1: The four variables v_1 to v_4 of Anscombe's Quartet Anscombe [1973] look statistically almost identical.

- *Hierarchies*: Hierarchical visualisations are used to represent data such as file systems, family trees, and classification systems. The major challenge here is that hierarchies can become very large, making it impossible to see the whole structure on one screen. Therefore clever interaction is required. One visualisation which achieves this in an elegant way is the Hyperbolic Browser, shown in Figure 2.4. It creates a circular tree in hyperbolic space and projects it to a 2d unit disc. This enables the user to see the whole hierarchy on one screen, while focusing on a certain subset of the data [Lamping, Rao, and Pirolli, 1995]. A more detailed description of the hyperbolic browser can be found in Section 8.6.

- *Networks and Graphs*: These are used to visualise data that describe social networks, document relations, and interconnected elements. Mathematically, they are represented as a set of nodes and links. The greatest difficulty is the large amount of data, which can lead to a very high density of information to be visualised. It is important to create a way of interacting, filtering, and focusing on a subset of the data. The Flare Dependency Graph in Figure 2.5 is a ring-based layout which visualises the dependencies between the classes in the Flare library [Heer, 2010]. It enables the user to select a class and highlight its dependencies.

- *Multi-Dimensional Metadata*: Visualising multi-dimensional metadata can be very challenging, since every data point can have large amounts of metadata associated with it. Parallel coordinates [Inselberg, 1985] (Figure 2.6) solves this problem elegantly by providing a way of seeing trends and filtering data in real-time. A more detailed description of parallel coordinates can be found in Section 8.2.

- *Feature Spaces*: Feature Spaces are used to visualise object collections according to similarities between objects. This can be accomplished by many different methods. VisIslands, shown in Figure 2.7, uses a clustering algorithm together with Force-Directed Placement (FDP) to position objects relative to their similarity [Andrews et al., 2001].

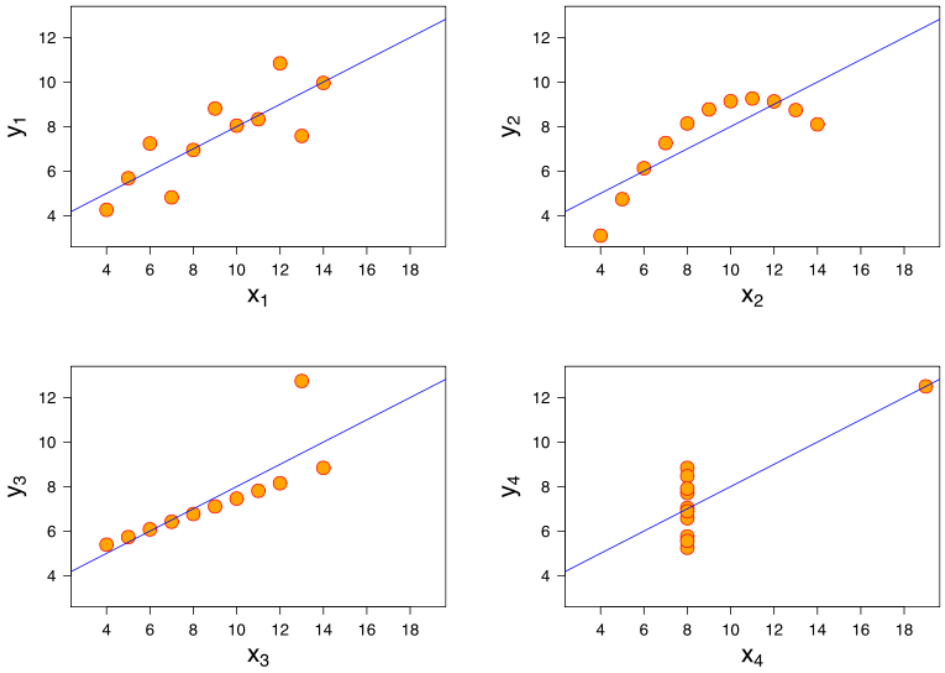


Figure 2.2: The four data sets of Anscombe's Quartet [Anscombe, 1973] look statistically identical (See Table 2.1). When plotted, it is immediately apparent that they are in fact very different. Image extracted from Wikipedia [2010] and used under the Creative Commons Attribution-Share Alike 3.0 Unported license.

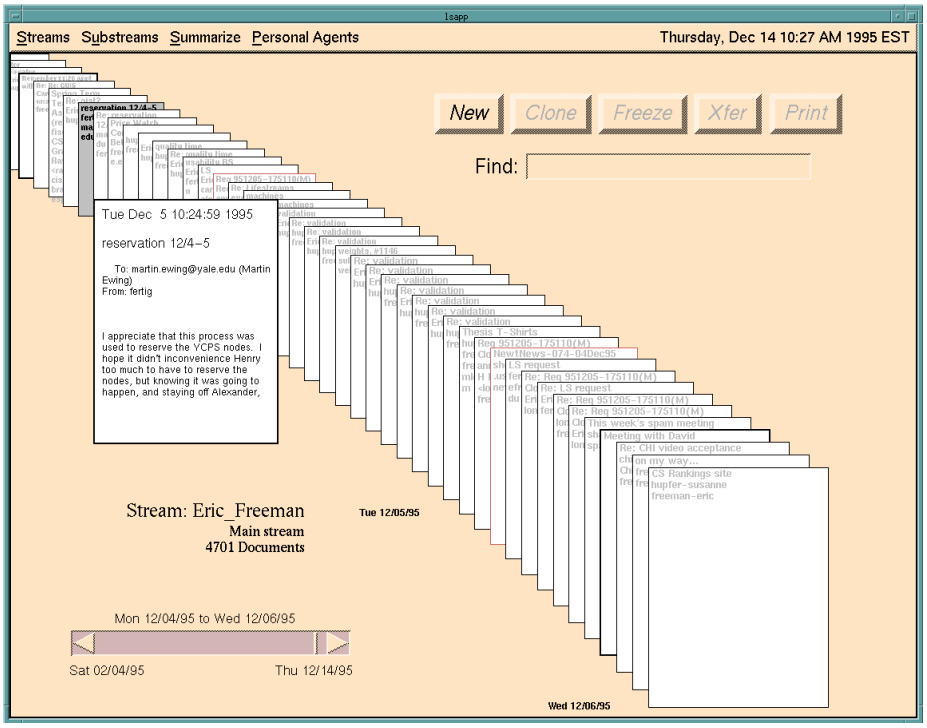


Figure 2.3: Lifestreams [Freeman and Fertig, 1995] displays documents in chronological order. The tail of the stream contains documents from the past while documents at front of the stream represent future documents such as to-do lists and reminders. [Copyright © by the Association for Computing Machinery, Inc.] Used under the terms of the ACM copyright Notice found on Page xii.

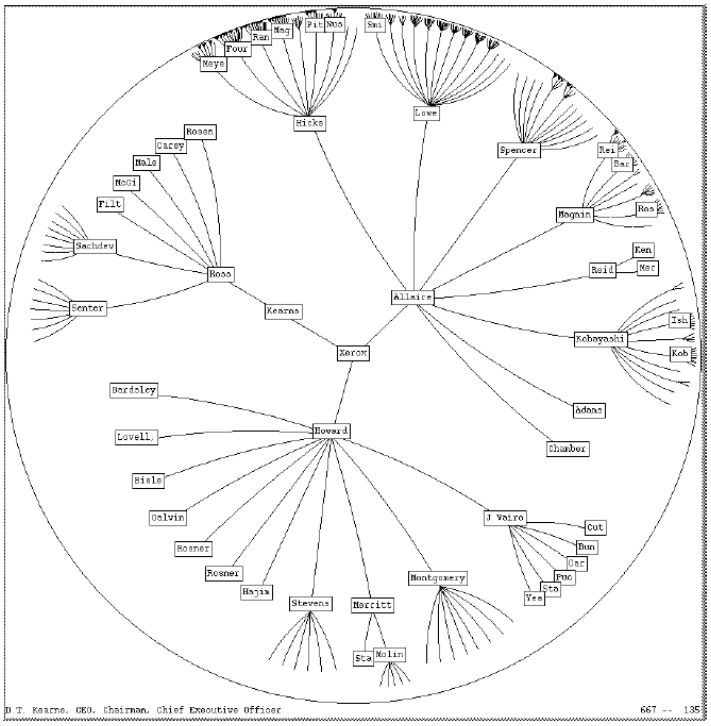


Figure 2.4: The Hyperbolic Browser creates a circular tree in hyperbolic space and projects it to a 2d unit disc. This enables the user to see the whole hierarchy in context while focusing on a subset of the data. Image extracted from Lamping, Rao, and Pirolli [1995]. [Copyright © by the Xerox Palo Alto Research Center] Used under the terms of the ACM copyright Notice found on Page xii.

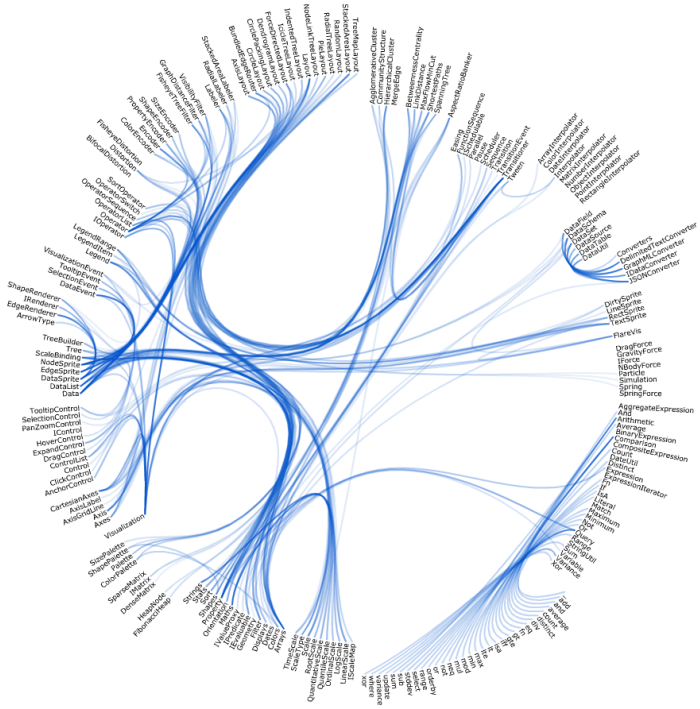


Figure 2.5: The Flare Dependency Graph [Heer, 2010] is a ring-based layout, showing the dependencies between classes from the Flare library. Edge bundling is used to collect links together, which have similar paths. Screenshot taken from [Heer, 2010] by the author.

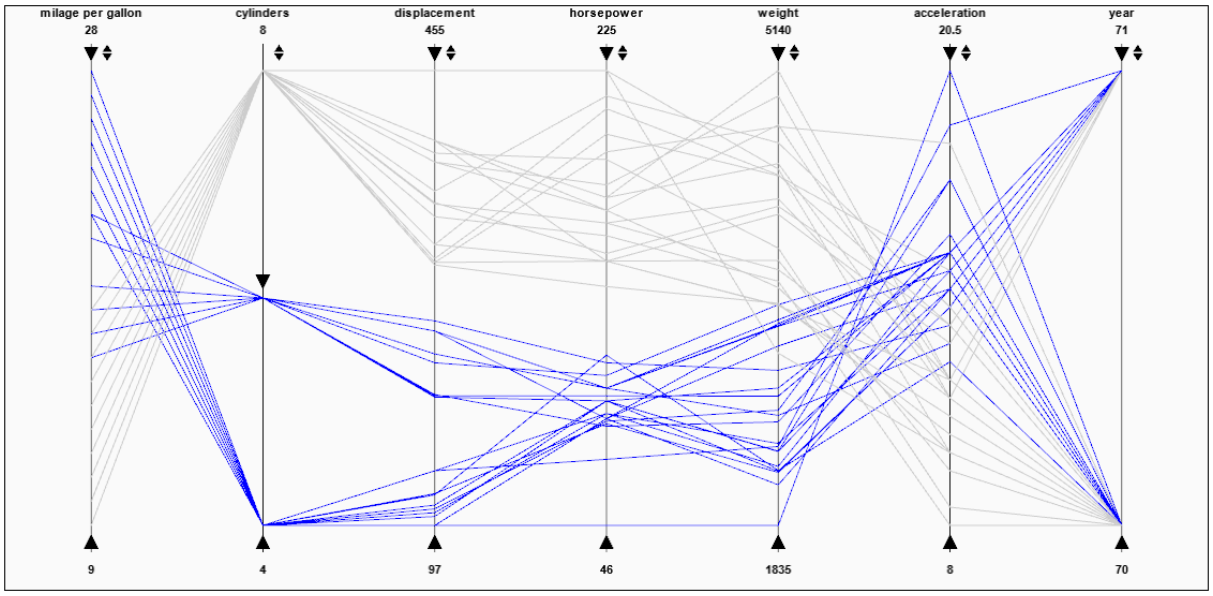


Figure 2.6: Parallel coordinates [Inselberg, 1985] create a poly line for each data point and a column for each metadata dimension. By using the handles, data can be filtered to detect trends or find data meeting certain criteria. This data set represents cars from the years 1970 and 1971 filtered by models with only 4 or 6 cylinders. Image created by the author using FluidDiagrams.

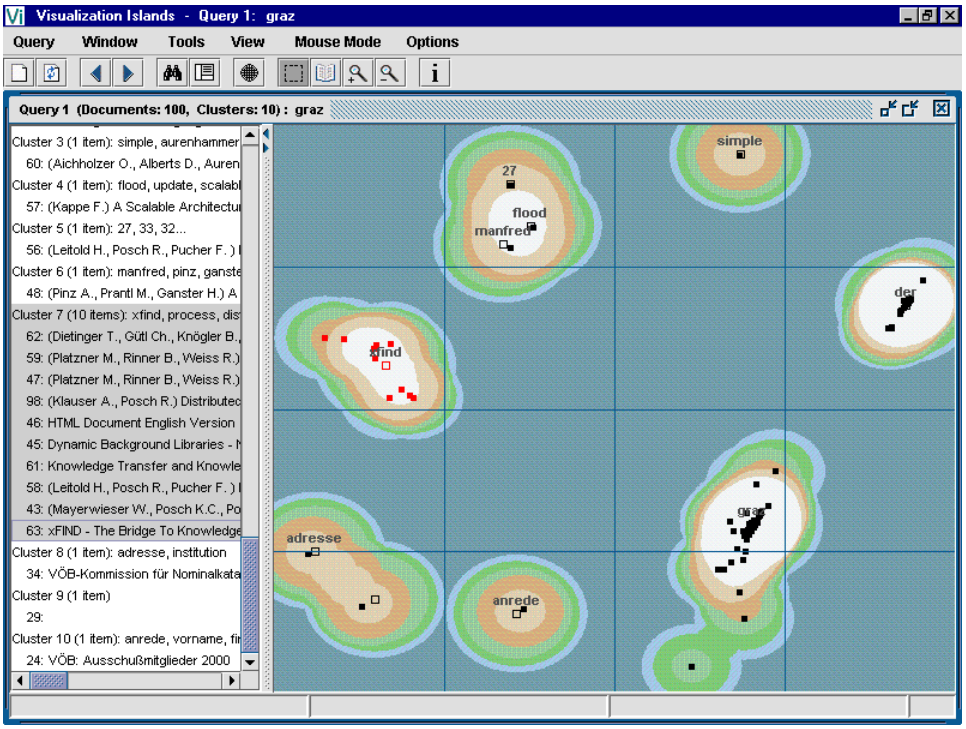


Figure 2.7: Visislands [Andrews et al., 2001] uses a clustering algorithm together with Force-Directed Placement (FDP) to position objects according to their similarity. Image extracted from Andrews [2013] and used with kind permission of Keith Andrews.

Chapter 3

Development Tools for JavaScript

In the beginning, JavaScript was used to add small interactive features to a web site. These features included form validation, mouse-over effects, and creating pop-ups. All of these features required very little code and were usually implemented in a single function. This lead to JavaScript files with hardly any structure and organisation, sometimes even a single file holding all the functions. This used to be no problem, because the functionality was easy to test and understand by a third party developer. Over time, the functionality of JavaScript was enhanced and developers started to create more sophisticated scripts that would add more user interaction and more functionality. Developers started taking control over the whole document object model with JavaScript, and even querying a server for resources asynchronously during user interaction. These new features require a new and more advanced way of developing in JavaScript. This paper gives a broad overview of these new technologies and some possible solutions to the arising problems.

3.1 JavaScript Concepts

Originally developed by Brendan Eich at Netscape under the names Mocha and later LiveScript, JavaScript was developed to enhance their browser by providing a simple lightweight programming language [Zakas, 2012]. JavaScript is an object-oriented dynamic programming language based on ECMAScript [ECMA, 2011]. Although it is object-oriented, it lacks the concept of classes, and instead uses object prototypes. Another aspect that separates JavaScript from most other languages, is that everything is treated as an object. So, a function is actually a container for executable code, which can be passed like ordinary objects and executed whenever needed. The following section is roughly based on the article "A Re-Introduction to JavaScript" by Willison [2006].

JavaScript has six built-in types:

- *Number*: In JavaScript, the type integer does not exist. Instead, it uses the type Number which are double-precision 64-bit format IEEE 754 values [IEEE, 2008]. Additionally, JavaScript provides a MATH object which handles advance mathematical functions and constants like $sin()$ and PI. Most languages cannot cope with $\frac{1}{0}$ and throw an exception. JavaScript deals with this problem by defining a special value ∞ so $\frac{1}{0} = \infty, \frac{-1}{0} = -\infty$.

- *String*: Strings are a sequence of Unicode characters, with each character represented as a 16-bit number. Strings come with a set of built-in methods, which can be directly called from a string, for example `"hello".length` will return 5.

- *Boolean*: A value of either true or false. The keyword "false", 0, an empty string, NaN(Not a Number), null and undefined evaluate to false. Every other value evaluates to true.

- *Object*: Objects are simple collections of name-value pairs, similar to dictionaries in Python or hash maps in Java. Object creation and usage is covered in Section 3.2.
- *Null*: A variable that has been assigned a null object.
- *undefined*: A variable that has not been assigned a value.

3.2 JavaScript Objects

Since there are no classes, JavaScript uses prototyping, which is the act of extending existing objects to add functionality and properties. To create a new object of a certain type, it is cloned from an existing object and all the constructors are invoked.

There are two basic ways of creating objects in JavaScript: `var obj = new Object();` and `var obj ={};` The second being identical to the coding of JSON described in Section 3.3.2

Accessing an object's elements can also be achieved in two different ways: `obj.name = "Bob"` and `obj["name"]="Bob"` The second way has the advantage that the access key to the element is a string and can be computed at runtime, enabling a more dynamic way of accessing object members.

Creating custom objects can be confusing for beginners, because objects are nothing else than functions, so defining a new object is achieved by defining a new function as follows:

```
function Person() {}
```

To create instances of this object, the *new* keyword is used:

```
var personA = new Person();
var personB = new Person();
```

The object function serves as the constructor of the object thus no constructor needs to be defined. The object function is called whenever a new instance is created. In the example shown in Listing 3.1, the constructor of the `Person` object requires a name. This is then stored in the object's property `name` during the construction phase of the object and can be accessed like any object property. Adding methods to an

```
1  function Person(name) {
2    this.name = name;
3  }
4
5  var person = new Person("Bob");
6
7  alert("the person's name is "+person.name) // the person's name is Bob
```

Listing 3.1: The constructor of `Person` requires a name as parameter. This is stored in the variable `name` and can then be accessed like any object property. Source code created by the author.

object works in the same way, as shown in Listing 3.2. Object inheritance allows the reuse of an object's properties and methods, with the possibility to add new ones. This is achieved using prototyping. To add a new method or property, one can simply add it using `.prototype` as shown in Listing 3.3. However, to use inheritance correctly without modifying the parent object, a slight modification is needed as shown in Listing 3.4.

```
1  function Person(name) {
2    this.name = name;
3    this.getName = function(){
4      return this.name;
5    }
6  }
7
8  var person = new Person("Bob");
9
10 alert("the persons name is "+person.getName()) // the persons name is
       Bob
```

Listing 3.2: Adding methods to an object is done in the same way as adding properties. The local variable `getName` is assigned a function which can then be accessed in the same fashion as object properties or method calls known from other programming languages. Source code created by the author.

```
1  Person.prototype.sayHello = function(){
2    alert("Hello World")
3  }
4
5  var person = new Person("Bob");
6  person.sayHello();
```

Listing 3.3: Adding additional methods to an existing object is achieved by using `.prototype`. Here, a method called `sayHello` is added to the object `Person`. A new object of type `Person` is then created which has the new method.Source code created by the author.

```
1
2    function Person(){
3      this.sayHello=function(){
4        alert("I am a person");
5      }
6    }
7
8    function Student(){
9      Person.call(this);
10     //call the Person's constructor
11   }
12
13   Student.prototype = new Person();
14   //Inherits all methods and properties from Person
15   Student.prototype.constructor = Student
16   // corrects the constructor to point to Student not to Person
17
18   //Now methods can be added or modified for Student without affecting
        Person
19   Student.prototype.sayHello = function(){
20     alert("I am a student");
21   }
22
23   var person = new Person();
24   var student = new Student();
25   person.sayHello(); // I am a person
26   student.sayHello(); // I am a student
```

Listing 3.4: To correctly use inheritance without altering the parent object, the parent is first copied in to the `.prototype` of `Student`. Then the `.constructor` of `Student` is set back to it's own constructor. This ensures that all functionality of `Person` is copied to `Student` and `Student` can be altered without changing `Person`. Source code created by the author.

3.3 JavaScript in the Browser

3.3.1 DOM

> "The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page" [W3C, 2005]

The Document Object Model (DOM) is the foundation of modern web sites, because it supports changing a web page dynamically. The DOM is a tree representation of all the elements of a web page, where the parent-child relation is determined by the nesting of the elements. For instance, a web page consisting of one div and an image within the div, would have the image as a child of the div, which is a child of the body which again is a child of the window. This DOM tree is provided by the web-browser and is accessible through a standardised API maintained by the W3C (World Wide Web Consortium).

There are two ways for JavaScript to interact with the DOM:

- Manipulating DOM elements.
- Reacting to DOM events.

Manipulating DOM elements

There are multiple ways of searching for and retrieving DOM elements, as described by Kantor [2013]:

1. `document.getElementById("id")`
   This obtains the DOM element with given id. It will find the element wherever it resides within the DOM tree. This method is very powerful if the id is unique and it is unknown where the element resides within the tree. In the event of the id not being unique, this method will return the first element with the id it can find, which may not be the desired element. This method is also not very efficient, because it has to search the whole DOM tree every time it is called. Hence, it is not advised to use `.getElementById('id')` within a loop or recursive functions where it is executed several times.

2. `document.getElementsByTagName("tag")`
   This will return an array with all the elements which match the tag. This method has the same downsides as `.getelementById()`, but it makes it possible to process multiple elements at once.

3. `document.getElementsByClassName()`
   Returns an array of elements with matching a CSS class name. It is supported by all browsers except IE 8.0 and lower.

4. `querySelector()`, `querySelectorAll()`
   This is a fairly new method which allows querying the DOM tree via CSS3 selectors. It is compatible with all modern browsers including IE 8+ (not in compatibility mode). The following code demonstrates the selection of the last `li` element that has `ul` as its direct parent:

   ```
   <script>
   var elements = document.querySelectorAll("ul > li:last-child")
   </script>
   ```

To optimise the selection of individual objects, the search tree can be reduced by selecting only a part of the whole DOM tree. This can be achieved by selecting an element and then restricting all resulting searches to that specific node, as shown in the following code example:

```
<script>
  var mydiv=document.createElement("div");
  var myspan=document.createElement("span");
  var mytext=document.createTextNode("My Text");
  var myimg = document.createElement("img");
  myimg.src="myImage.png";

  myspan.appendChild(mytext);
  mydiv.appendChild(myspan);
  mydiv.appendChild(myimg);

  document.getElementById("container").appendChild(mydiv);
</script>
```

Listing 3.5: Using `.createElement` and `.appendChild` to create and nest HTML elements is a more favourable way to add elements to a web page. It eliminates human error and ensures correct nesting. However, one has to still be cautious when using `.getElementById`. Source code created by the author.

```
<script>
  var element = document.getElementById("id")
  var list = element.getElementsByTagName("li")
</script>
```

When adding HTML elements to a web page, it is very important not to break the DOM. The code in listing below may break the DOM tree when the provided `id` is not unique, because `.getElementById("id")` returns the first element with a matching id. Additionally, creating and nesting HTML elements manually introduces a new source of errors due to human error.

```
<script>
  document.getElementById("container").innerHTML='<div id="outer"><span
  id="inner">inner HTML</span></div>';
</script>
```

It is recommended to create DOM elements and add them to a page using `.createElement` and `.appendChild`, as shown in Listing 3.5. This ensures that the DOM is always correct and that new elements are nested in the expected way.

Reacting to DOM events

There are many different events upon which can be listened for and reacted to. A detailed list of available events in Firefox can be found in Shepard et al. [2013]. For simplicity, only the most common event will be discussed here: `onClick`. Other events work in a similar way.

There are basically two ways of reacting to events, as described in Heilmann [2012]. The simplest way is by adding a method to the DOM element as follows:

```
<img src="img.png" onClick="function()" />;
```

The function is executed, in the event that the image is clicked. This is not the most elegant way of doing things, because it mixes functionality and presentation. A much cleaner way of dealing with events is by adding an event handler to the DOM element, as shown here:

```
<script>
  var element = document.getElementById("id")
  element.addEventListener('click', eventMethod, false);

  function eventMethod(ev){
        // do something
  }
</script>
```

Listing 3.6: Adding an event-listener to the DOM element with id `id`. When the element is clicked the method `eventMethod(ev)` is executed. Source code created by the author.

```
<script>
  function eventMethod(ev){
        var target = ev.target;
        if(target.tagName === 'A'){
                //a href fired the event
        }
        if(target.tagName === 'LI'){
                //a list element fired the event
        }
  }

document.body.addEventListener('click', eventMethod, true);
</script>
```

Listing 3.7: Using one method to handle all the events. This makes it easier to react to new events. By setting `useCapture=true` in the body's event listener, it is ensured that no other event handler will be triggered. The `ev` parameter is used to distinguish between the different events and targets. Source code created by the author.

```
element.addEventListener(event, handler,useCapture);
```

This will add an event listener to the element. The parameter `event` declares the type of event to listen to, `handler` is a callback method that is invoked when the event fires, and `useCapture` is a boolean defining if a capturing event is created or not. If `useCapture` is set to true, it prevents the event from being propagated to child elements. The code in the listing bellow shows a short example of this behaviour.

```
document.body.addEventListener('click', function(), true);
<body>
    <p onClick="foo()">click me</p>
```

Before the click event from `p` is executed, the click event from `body` is handled. In this case the `body`'s event handler will prevent `p`'s event-handler from being triggered. Usually, `useCapture` will be set to false, because this behaviour is is not often needed, and the parameter is optional in all browsers but Opera. This behaviour is described in more detail in Heilmann [2012]. The code in Listing 3.6 gives a short example of adding an event-listener to a DOM element: The parameter `ev` of `eventMethod` contains information about the event which called this method, for example the event target and the event type. The `ev` parameter makes it possible to do event delegation, meaning that one method can handle all events, as demonstrated in Listing 3.7.

Note that DOM elements do not always have to be HTML elements. In some cases, event handlers

```
1 var someJSON={
2   "firstName" : "John",
3   "lastName" : "Doe",
4   "destinations: ["Moon", "Mars"]
5 };
6
7 document.writeln(someJSON.lastName); //outputs Doe
8 document.writeln(someJSON.destination[1]); //outputs Mars
```

Listing 3.8: A new JSON object is stored in `someJSON`, and then accessed like a standard JavaScript object. Source code created by the author.

can also be assigned to SVG elements and other non-HTML DOM elements.

3.3.2 JSON

JSON stands for JavaScript Object Notation and is a lightweight data-interchange format, which is also easily read and written by humans [json.org, 2013] [Zakas, 2012] . One benefit of JSON compared to XML, is that it uses less bandwidth due to its slimmer structure.

JSON is composed of three simple structures:

1. Object: An object starts with { followed by a series of key-value pairs `string:value` separated by , and ends with }
2. Array: An array starts with { followed by a series of values separated by , and ends with }
3. Value: A Value can be one of following:
   - string
   - number
   - (JSON)object
   - array
   - boolean
   - null

With JSON it is easy to send data to and from a server, or store data for future use in an application. Since JSON is an object representation, it is accessible like an object in JavaScript, as illustrated in Listing 3.8 .

Due to the fact that functions in JavaScript are actually objects, JSON can be used to encode functions, although this is discouraged, because it enables script injection. However, to show the full picture, it is demonstrated in Listing 3.9.

3.3.3 AJAX

AJAX stands for Asynchronous JavaScript and XML and is a combination of techniques to create client-server communication for web development. With this new method of communication, data can be sent to and from the web server without having to reload the complete HTML page. This helps create richer web pages and enhances the user experience. AJAX also cleared the path for modern web applications, by enabling the kind of interfaces people are used to with standalone software inside the web browser.

```
1  var json={
2          "FirstName":"John",
3          "LastName":"Doe",
4          "greet":function(){alert("hello my name is "+this.FirstName+"
5  "+this.LastName)}
6          };
7
8          json.greet(); //Alert box with: "hello my name is John Doe"
```

Listing 3.9: Since functions in JavaScript are objects, they can be also stored in a JSON object. Such use is discouraged, because it enables script injection. The output of this code is an alert box containing `"hello my name is John Doe"`. Source code created by the author.

Even though AJAX implies the use of XML, it is not restricted to this. Using AJAX technologies, any kind of textual data such as JSON or HTML can be transmitted, and often is.

The technologies involved in AJAX are [Wikipedia, 2013a]:

- *HTML and CSS*: This is the context in which the retrieved data is put or taken to be sent to the server.
- *the DOM*: As described in Section 3.3.1 this is used to modify the existing web page.
- *XML, JSON and other text-based data types*: These data types are used to send data to and from the server. Usually, this data is processed both on the server side with Java, PHP, or other server-side languages, and on the client-side using JavaScript.
- *XMLHttpRequest*: This is the API implemented by the browser that enables asynchronous requests to the server. These requests are usually HTTP_GET or HTTP_POST, but can be any request implemented by the server. The API also allows a callback method to be specified, which is executed when the state of the transmission changes. These states can be:
  - *UNSET*: Object has been created.
  - *OPENED*: Connection has been opened.
  - *HEADERS_RECEIVED*: All the headers of the response have been received.
  - *LOADING*: The response body is being received.
  - *DONE*: All data has been transferred or an error has occurred. This is the state when modifications to the web page are usually made, informing the user about the state of the request, or by displaying new content and information.

  A detailed definition of XMLHttpRequests can be found in W3C [2012b].
- *JavaScript*: JavaScript is used to combine the above technologies to modify the DOM, invoke a `XMLHttpRequest`, and implement the callback method.

  Listing 3.10 shows how to create the relevant objects to make an AJAX call to a remote resource. It consists of creating the required object depending on the used browser, and a function which is executed upon the state change event of the request. An easier way to do this using jQuery is shown in Section 3.4.

  AJAX however brings new problems to web design which have to be taken in to consideration:
  - *Page History*: Pages requested via AJAX are not automatically registered with the browser, so clicking the back button may lead to unexpected behaviour.

```
var xmlhttp;
if (window.XMLHttpRequest)
  {// IE7+, Firefox, Chrome, Opera, Safari
  xmlhttp=new XMLHttpRequest();
  }
else
  {// IE6, IE5
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
xmlhttp.onreadystatechange=function()
  {
  if (xmlhttp.readyState==4 && xmlhttp.status==200)
    {
      //Do something widh xmlhttp.responseText;
    }
  }
xmlhttp.open("GET","URL",true);
xmlhttp.send();
```

Listing 3.10: This example shows how a XHTMLHttpRequest is created depending on the used browser, and how a function is defined, which is then executed when the state of the request changes. Source code created by the author.

- *Bookmarking*: Returning to a bookmarked page will lead to the page in its initial state without any user interaction. So the result may not be as expected.
- *Late Responses*: In the event of a slow connection, the callback method may be executed after the user has already moved on to a different task, resulting in a poor UI experience.
- *Indexing*: Current web crawlers cannot execute JavaScript making it impossible for them to index data only requested via AJAX methods.
- *Disabled JavaScript and/or Old Browser*: Older browsers or browsers with JavaScript disabled will not be able to execute AJAX requests.
- *Maintenance/Debug/Test*: Web pages with consecutive AJAX requests become very hard to test, debug, and maintain, due to the asynchronous nature of these pages.

This information was summarised from the web page: Ajax (Programming) [Wikipedia, 2013a].

```
<body>
  <div class="style1">Something</div>
  <div class="style1">Something else</div>

<script>
  $(".style1").css("border", "1em solid black");
</script>

</body>
```

Listing 3.11: Selecting elements by their CSS class, resulting in a list of elements, and setting their border properties to solid, black, and one em.

3.4 Using JavaScript Libraries

In modern software development one frequent objective is to create reusable code. This reduces the complexity of software and increases development speed. The reduction of complexity is a crucial part of ensuring functional and maintainable code, by making the software easier to understand. Reusing code also adds to the quality of the software by using already tested functionality without adding new bugs or unwanted behaviour. This urge to reuse existing code has lead to an increasing number of libraries for all sorts of applications. One benefit of JavaScript libraries, compared to traditional libraries, is the fact that they are not compiled making it easy to modify, debug, and extend the code. In recent years, there has been an explosion of JavaScript libraries, but probably most well-known is jQuery, which has become an industry standard in recent years.

jQuery

jQuery is currently the most used and best-known JavaScript library. It was designed to simplify HTML document traversing, event handling, animations, and AJAX interactions, as described by jQuery [2013a] and Flanagan [2011] . Its power lies in the fact that it runs in all browsers browsers and simplifies the most common tasks in JavaScript. jQuery also comes with dozens of plug-ins to further increase its functionality. In addition to its own functions, many new libraries and frameworks are built on top of jQuery. In this section, a brief overview of the most useful functions is given.

HTML Selector

There are multiple ways to select elements with jQuery. Three methods will be described here:

- *Selecting by Class*: The first method selects elements by class name and is the equivalent of getElementsByClassName(). The code in Listing 3.11 shows how to select all elements with a given class and set the CSS border property.
- *Selecting by ID*: The second method selects elements by id and is the equivalent of `document.getElementById()` as described in Section 3.3.1. The code in Listing 3.12 shows how to select the element with the given id and set the CSS border property.
- *Selecting by Element*: This method selects all the specified HTML elements. The code in Listing 3.13 shows how to select all `div` elements and set the CSS border property.

As can be seen, selecting elements by id, class, or type is almost identical. The only difference is the prefix which corresponds to the CSS way of differentiating between elements, ids, and classes.

```
<body>
  <div id="content1">Something</div>
  <div id="content2">Something else</div>

<script>
  $("#content1").css("border", "1em solid black");
</script>

</body>
```

Listing 3.12: Selecting elements by id, and then setting the border property to solid, black, and one em.

```
<body>
  <div">Something</div>
  <div>Something else</div>

<script>
  $("div").css("border", "1em solid black");
</script>

</body>
```

Listing 3.13: Selecting all divs, and setting the border property to solid, black, and one em.

AJAX Requests

Creating AJAX requests is, compared to standard JavaScript, extremely simple. The only code needed is shown in Listing 3.14. The main elements of the AJAX request are:

- *type*: can be any RESTFUL request supported by the server, usually `POST` or `GET`.
- *url*: the destination of the request.
- *data*: can be any textual data, but is usually JSON-encoded and is sent to the server. It is optional and is only needed in requests which send data to the server.
- *success*: the callback method executed when the request was successful. Additional functions can be defined, like "error".

Event Handling

The simplest way of implementing events in jQuery is to bind a function to an element's event. The implementation of a click event is demonstrated in the listing below. A list of events in the Firefox browser can be found in Shepard et al. [2013]. Even though this list is for Firefox, most of the events are implemented by all modern browsers.

```
$('elem').on('click', function(event){
  // do something
});
```

The full signature of the "on()" method defined by jQuery [2013b] is as follows:

```
1  $.ajax({
2              type: "POST",
3              url: "destination",
4              data: data,
5              success: function(data) {
6                 // do something with data
7              }
8          });
```

Listing 3.14: Making an AJAX `POST` request to `destination`, sending `data`, and executing a function upon success.

- *events*: A space-separated list of events.
- *selector*: A space-separated list of elements which trigger the event. If the list is null, the event is always triggered when the event reaches the selected element. The listing below shows an example which delegates the event from the element defined in `selector` to the assigned event handler.
- *data*: Optional data passed to the event handler.
- *handler(eventObject)*: A function executed if the event is triggered. The `eventObject` is a reference to the element which triggered the event.

```
1 $('body').on('click', 'tr', function(event){
2   // do something if tr is clicked but delegate it to body event.
3 });
```

There are multiple ways of adding, removing, delegating, and defining event handlers. To remove the event handler, a simple call to "off" is needed, as demonstrated in Listing 3.15. The first off(...) call removes only the specified event handler. The second off(...) call removes all the attached event handlers. This enables a very dynamic way of handling events. However, there are two major issues with this approach:

- Maintainability:
  It is very easy to lose track of the event handlers attached to different elements and the delegations to other element's event handlers. This makes it difficult to maintain and debug the code in later development stages.
- SVG:
  Event delegation does not work for elements within a block of SVG code.

Animation / UI

jQuery enables an easy way of animating CSS properties. This is achieved by defining a set of CSS properties which should change over time and a duration for the animation. Additionally, a callback method can be defined, which is called when the animation is finished and an easing parameter can be set, to define the function used for the transition. The only restriction is that the CSS property must be a numerical value. JQuery UI is a project which extends this animation system and allows animation of some non-numerical CSS values such as colours. Additionally jQuery UI enables animation between CSS classes and not only attributes [jQuery, 2013b]. The example in Listing 3.16 demonstrates a simple animation which changes the width of a div when it is clicked.

```
1
2  function myClick(){
3    alert("was clicked");
4  }
5  function anotherClick(){
6    alert("second handler");
7  }
8
9  $('#element').on('click',myClick);
10 $('#element').on('click',anotherClick);
11
12 $('#element').off('click', myClick);
13 // removes the myClick event handler from #element
14
15 $('#element').off();
16 // removes all event-handlers from #element
```

Listing 3.15: Removing event handlers from elements. The first `.off()` removes the `myClick` event handler from `element`. The second `.off()` removes all event handlers from `element`. Source code created by the author.

```
1
2  <html>
3    <head>
4      <style>
5        div {
6    background-color:#bca;
7    width:100em;
8    border:1em solid green;
9        }
10     </style>
11
12     <script src="http://code.jquery.com/jquery-latest.js"></script>
13   </head>
14   <body>
15   <div id="ele">Hello World!</div>
16
17   <script>
18     $("#ele").on('click', function(){
19       $("#ele").animate(
20   {
21     width: "300em"
22   },
23   1000
24       );
25     });
26   </script>
27
28   </body>
29 </html>
```

Listing 3.16: This example adds a click event handler to `ele`, which animates the width of the `ele` to become `300em`. The animation duration is 1000ms. Source code created by the author.

3.5 JavaScript Supersets

JavaScript supersets are enhancements to the JavaScript language which add new features and paradigms. The main purpose of JavaScript supersets is to increase maintainability and readability. In some cases, supersets also add primitive runtime optimisations. One large benefit of supersets is the ability to compile the resulting code to native JavaScript, runnable on all browsers without plug-ins or extensions.

The largest and most used JavaScript superset is CoffeeScript. CoffeeScript was inspired by Ruby, Python and Haskel, and some syntactic elements of JacaScript (for example,`{ } ; )` have been removed. A recent superset is Microsoft's TypeScript. This superset adds classes, modules, interfaces, and typing to the JavaScript language. TypeScript is described in more detail in Section 3.5.2

3.5.1 CoffeeScript

CoffeeScript, as defined in CoffeeScript [2010] and described in detail by MacCaw [2012], aims at enhancing the readability and maintainability of JavaScript. Its syntax is close to Python or Ruby, meaning blocks like `if` and `while` do not use braces `{ }` but indentations. The code of CoffeeScript is compiled to JavaScript which means no additional interpreter or browser plug-in is required. Additionally, this means that existing libraries in JavaScript can be used within CoffeeScript. The listings below show a simple function written in CoffeeScript and the compiled result.

```
square = (x) -> x*x
```

```
var square;
square = function(x){
    return x*x;
}
```

CoffeeScript also introduces the concept of classes to JavaScript. Listings 3.17 and 3.18 show a simple example with one base class and two inheritances. These demonstrate the use of `super` which calls the implementation of its parent object.

As can be seen in the above examples, CoffeeScript reduces the amount of code written and thereby increases its readability and maintainability. A downside for some developers might be the loss of brackets and semicolons, which sometimes makes it harder to read.

3.5.2 TypeScript

TypeScript is a superset of JavaScript aimed at scalable application development [Fenton, 2013]. It is written in and compiled to standard JavaScript. This enables the use of existing JavaScript applications and libraries within a TypeScript project. TypeScript exists as a command-line compiler and as a Visual Studio plug-in. Typescript adds four major features to JavaScript:

- Classes: With the introduction of classes, it is easier to implement class inheritance.
- Modules: A module helps organise and structure code. This increases modularity which itself increases flexibility and interchangeability within and between applications.
- Interfaces: An interfaces provides a way of defining a class without implementing it, increasing abstraction and code reuse.
- Type checking: Typechecking at compile time makes it easier to debug the application.

In the next sections, the points above will be discussed in more detail. TypeScript is defined and documented in Microsoft [2013].

```
class Animal
  constructor: (@name) ->

  move: (meters) ->
    alert @name + " moved #{meters}m."

class Snake extends Animal
  move: ->
    alert "Slithering..."
    super 5

class Horse extends Animal
  move: ->
    alert "Galloping..."
    super 45

sam = new Snake "Sammy the Python"
tom = new Horse "Tommy the Palomino"

sam.move()
tom.move()
```

Listing 3.17: A basic class in CoffeeScript with a constructor which takes one parameter and automatically stores it in the `name` variable. Additionally, the method `move` is defined. Two classes `Snake` and `Horse` are then derived from the basic `Animal` class. Source code taken from the official documentation [CoffeeScript, 2010]

Classes

Listing 3.19 shows a simple class in TypeScript and is basically identical to the example given in Listing 3.1. As can be seen in the compiled result in Listing 3.20, the resulting JavaScript is well structured and human-readable. It also resembles something one might have written by hand. Extending classes in TypeScript is very easy, because it supports class inheritance. The example given in Listing 3.21 shows how to inherit from the `Person` class and overwrite the `sayHello()` method. It also demonstrates how to call the parent's constructor. Comparing the TypeScript code with the compiled result in Listing 3.20, one can see the more compact nature of TypeScript code.

Modules

As discussed in Section 3.7.2, modules are a good way of organising code into reusable blocks. TypeScript supports modules by using the keyword *module*. The example in Listing 3.23 shows how to create a module, add a class to it, and how to use it.

Interfaces

Interfaces are used to define an abstract type. Interfaces only define the way the object looks, but do not implement any functionality. Usually, interfaces only contain methods, but in TypeScript they can also contain member variables. In TypeScript, two types are compatible when their internal structure is compatible. A class *implements* an interface, when all methods and variables defined in the interface, are available in the class. No implements keyword is needed. This way it is easy for a single class to implement many interfaces. Interfaces are very useful when working with external libraries written in

```
var Animal, Horse, Snake, sam, tom, _ref, _ref1,
  __hasProp = {}.hasOwnProperty,
  __extends = function(child, parent) { for (var key in parent) { if
(__hasProp.call(parent, key)) child[key] = parent[key]; } function ctor()
    {
this.constructor = child; } ctor.prototype = parent.prototype; child.
   prototype =
new ctor(); child.__super__ = parent.prototype; return child; };

Animal = (function() {
  function Animal(name) {
    this.name = name;
  }
  Animal.prototype.move = function(meters) {
    return alert(this.name + (" moved " + meters + "m."));
  };
  return Animal;
})();

Snake = (function(_super) {
  __extends(Snake, _super);

  function Snake() {
    _ref = Snake.__super__.constructor.apply(this, arguments);
    return _ref;
  }
  Snake.prototype.move = function() {
    alert("Slithering...");
    return Snake.__super__.move.call(this, 5);
  };
  return Snake;
})(Animal);

Horse = (function(_super) {
  __extends(Horse, _super);

  function Horse() {
    _ref1 = Horse.__super__.constructor.apply(this, arguments);
    return _ref1;
  }
  Horse.prototype.move = function() {
    alert("Galloping...");
    return Horse.__super__.move.call(this, 45);
  };
  return Horse;
})(Animal);

sam = new Snake("Sammy the Python");
tom = new Horse("Tommy the Palomino");
sam.move();
tom.move();
}
```

Listing 3.18: The basic class and two inheritances in CoffeeScript from Listing 3.17, once they have been compiled to JavaScript [CoffeeScript, 2010].

```
class Person {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    sayHello() {
        return "Hello, I am " + this.name;
    }
}

var person = new Person("Bob");

alert(person.sayHello());
```

Listing 3.19: A new class called `Person` is defined in TypeScript. The constructor takes one argument, and stores it in the class variable `name`. The method `sayHello` returns a simple greeting string. Source code created by the author.

```
var Person = (function () {
    function Person(name) {
        this.name = name;
    }
    Person.prototype.sayHello = function () {
        return "Hello, I am " + this.name;
    };
    return Person;
})();
var person = new Person("Bob");
alert(person.sayHello());
```

Listing 3.20: The compiled result from Listing 3.19 with class Person. The clear structure of the resulting JavaScript code and its readability can be seen.

```
class Student extends Person{
  constructor(name:string){
    super(name);
  }
  sayHello(){
    return "Hi, My name is "+this.name+" and I am a Student";
  }

}
```

Listing 3.21: Simple class inheritance in TypeScript with `Student` extending `Person` from Listing 3.19. Source code created by the author.

```
var Student = (function (_super) {
    __extends(Student, _super);
    function Student(name) {
        _super.call(this, name);
    }
    Student.prototype.sayHello = function () {
        return "Hi, My name is " + this.name + " and i am a Student";
    };
    return Student;
})(Person);
```

Listing 3.22: The compiled result from Listing 3.21.

```
module Humans {
    export class Person {
      name: string;
      constructor(name: string) {
          this.name = name;
      }
      sayHello() {
          return "Hello, I am " + this.name;
      }
  }
}
var person = new Humans.Person("Bob")
alert(person.sayHello());
```

Listing 3.23: Create the module `Humans` in TypeScript, add a class `Person` to this module and demonstrate how the module is used. Source code created by the author.

```
var Humans;
(function (Humans) {
    var Person = (function () {
        function Person(name) {
            this.name = name;
        }
        Person.prototype.sayHello = function () {
            return "Hello, I am " + this.name;
        };
        return Person;
    })();
    Humans.Person = Person;
})(Humans || (Humans = {}));
var person = new Humans.Person("Bob");
alert(person.sayHello());
```

Listing 3.24: The compiled result of TypeScript module from Listing 3.23.

JavaScript. To make the method calls and classes available to TypeScript only the interface needs to be defined, the actual code lies within the library.

Type Checking

During the compilation of TypeScript to pure JavaScript the compiler can perform strict type checking, which raises errors upon incompatible types. This helps with debugging the code during development and reduces the overall risk of errors during execution of the JavaScript. After compilation this mechanism is lost, and type errors arising during the execution of the JavaScript code are not detected.

3.6 Testing JavaScript Code

Software testing is the process of verifying that an application meets the requirements defined by the project and works in the intended way. There are many different aspects when it comes to testing, including requirements engineering at the beginning of a project, functional testing during implementation, and usability testing which should be started as soon as a user interface is created. Testing JavaScript applications in the browser can be a challenge, due to the fact that most browsers are very fault-tolerant, meaning that an error or mistake in the JavaScript code does not necessarily result in misbehaviour of the application. This can be especially problematic when creating DOM elements dynamically, resulting in a broken DOM-Hierarchy, which might render correctly in the browser but behave unexpectedly.

3.6.1 Testing Strategies

- *Output Statements*: Up until recently, debugging JavaScript code was done by adding output to the program flow, such as alerts, and manually comparing the output with the expected program flow. This, even if it does work, requires a great deal of time and effort. Another problem with this method is that the code becomes full of "debugging statements", which have to be removed after the tests, eliminating the possibility to rerun the tests. Additionally, these tests do not cover many of the possible routes through the software and later bugs can be introduced without the developer noticing. However, this method can be used to trace and remove an already found bug.

- *Step Debugging*: A slightly better way of tracing and fixing bugs are so-called step debuggers which allow traversal of the code in the way the program would flow, inspecting each element at every step.

- *Automated Testing*: Ultimately, the only way to do testing in a way which can be repeated and does not break any piece of the code is by creating automated tests. On way of doing this is by creating so-called *unit tests*. Unit tests conduct a test of a very small piece, a unit, of the code in a way that can be repeated at any time of the development. This enables the tests to be re-run whenever changes are made, ensuring that no new errors are introduced. The next section gives an example using a unit test framework for JavaScript called Jasmine.

3.6.2 Jasmine

Unit Tests

Jasmine is a unit test framework for JavaScript [Pivotal, 2013]. Unit tests in Jasmine are built up in suites. Each suite consists of a describe function which takes a description and an in-line function containing the actual test, also called a spec. This spec then calls the actual method under test, and compares it to an expected value. The code in Listing 3.25 shows one suite with one test. A suite can have multiple specs as shown in Listing 3.26. Each spec contains a description and a function which calls the method under test and compares it to the expected value. The actual test is done by calling the method under test in `expect()` and comparing it to the value stated in `toBe()`. In the example given in Listing 3.25 `methodUnderTest()` is expected to return `true` [Pivotal, 2013].

Listing 3.26 gives an example of a boolean function under test. The function inverts the input and returns it, so true becomes false and vice versa. The test suite contains two tests, each testing for a different input. The examples in Listings 3.27 and 3.28 show how to test numeric functions and functions which modify strings.

Matchers are responsible for comparing the result of the method under test and the provided value. Jasmine comes with a rich set of matchers, as described in Pivotal, 2013:

```
describe("A suite", function() {
  it("contains spec with an expectation", function() {
    expect(methodUnderTest()).toBe(true);
  });
});
```

Listing 3.25: A basic unit test with Jasmine. The expected value of `methodUnderTest(). Source code extracted from \citet {jasminJS}` is `true` [Pivotal, 2013].

```
function invert(a){
  return !a;
}

describe("Testing invert(a)", function(){
  it('input was "true" expected "false"', function(){
    expect(invert(true)).toBe(false);
  });
  it('input was "false" expected "true"', function(){
    expect(invert(false)).toBe(true);
  });

});
```

Listing 3.26: Jasmine unit test with boolean values. Source code adapted from Pivotal [2013]

```
 function add(a,b){
                return a+b;
}

describe("Testing add(a,b)", function(){
  it('input was 1,2 expecting 3', function(){
    expect(add(1,2)).toBe(3);
  });
  it('input was 2,-1 expected 1', function(){
    expect(add(2,-1)).toBe(1);
  });
  it('input was -2,1 expected 1', function(){
    expect(add(-2,1)).toBe(-1);
  });


});
```

Listing 3.27: Jasmine unit test of method `add(a,b)` with different numerical values. Source code adapted from Pivotal [2013]

```
function UBBCodeToHTML($str){

  $format_search =  [/\[b\](.*?)\[\/b\]/ig,
                     /\[i\](.*?)\[\/i\]/ig,
                     /\[u\](.*?)\[\/u\]/ig
                    ];

  $format_replace = ['<strong>$1</strong>',
                     '<em>$1</em>',
                     '<span style="text-decoration:underline;">$1</span>'
                    ];

  for (var i =0;i<$format_search.length;i++) {
    $str = $str.replace($format_search[i], $format_replace[i]);
  }

  return $str;
}


describe("Testing UBB Code converter", function(){
  it('input was "[b]Bold Text[/b]"', function(){
    expect(UBBCodeToHTML("[b]BoldText[/b]")).toBe("<strong>Bold Text</
        strong>");
  });
  it('input was "[i]Italic Text[/i]"', function(){
    expect(UBBCodeToHTML("[i]Italic Text[/i]")).toBe("<em>Italic
Text</em>");lst:jasminAsync
  });
  it('input was "[u]Italic Text[/u]"', function(){
    expect(UBBCodeToHTML("[u]Underline
      Text[/u]")).toBe('<span style="text-decoration:
      underline;">Underline Text</span>');
  });
});
```

Listing 3.28: Jasmine unit test testing of a simple UBB to HTML converter with three different input values. Source code created by the author.

```
var foo = {
  a: 12,
  b: 34
};
var bar = {
  a: 12,
  b: 34
};
expect(foo).toEqual(bar);
```

Listing 3.29: Matching two objects in Jasmine, expecting them to be equal. Source code extracted from Pivotal [2013].

```
var pi = 3.1415926;
e1 = 3.14;
e2 = 3;
expect(pi).toBeCloseTo(e, 2);
expect(pi).toBeCloseTo(e2, 2); //WILL NOT PASS
```

Listing 3.30: Matching two numbers in Jasmine, expecting them to be almost equal. Source code extracted from Pivotal [2013].

- toBe compares with === operator.
- toEqual matches literals and `simple` objects. The code in Listing 3.29 gives an example of two objects being compared.
- toMatch matches regular expressions.
- toBeDefined returns true if object under test is defined.
- toBeUndefined inverse of toBeDefined.
- toBeNull compares against null.
- toBeTruthy checks if value is boolean.
- toContain matches if an element is contained in an array.
- toBeLessThan and toBeGreaterThan checks for greater or lesser.
- toBeCloseTo checks if value is equal up to given precision.
- toThrow tests if method throws an exception.

Additionally, custom matchers can be defined within a test. This is achieved by calling this.addMatchers within a test. The code in Listing 3.31 shows how to add a matcher which checks if two numbers are not equal. Jasmine also allows testing against types, using jasmine.any(TYPE) as shown in the listing below.

```
expect({}).toEqual(jasmine.any(Object));
expect(12).toEqual(jasmine.any(Number));
```

```
describe("A suite", function() {
  it("contains spec with an expectation", function() {
    this.addMAtcher({
      toBeNotEqual: function(expected){
    return this.actual != expected;
      }
    });

    expect(5).toBeNotEqual(1);
  });
});
```

Listing 3.31: Adding a custom matcher to a test, which checks if two values are not equal. Source code created by the author.

Asynchronous Unit Tests

Modern JavaScript applications make frequent use of asynchronous calls, thus a means of testing such calls is needed. Jasmine provides this necessary functionality with the `runs` and `waitsFor` methods. The example provided in Listing 3.32 demonstrates these methods. To test asynchronous method calls, three parts are required:

- `runs`: This calls the asynchronous method. In the example, this is a simple function that waits 500ms and then sets `flag` to true.
- `waitsFor`: This method expects a function that returns true or false. In the example, this is achieved by returning the value of `flag`. Additionally the method requires an error message and a timeout. The `waitsFor` method is polled until the return value is either true or the timeout has passed. If the timeout passes, the result is interpreted as false and the test fails returning the stated error message.
- `runs`: This second runs call specifies the actual test and usually contains an expect clause.

Spies

In addition to conducting unit tests with Jasmine, spies can be used to detect if certain functions were called, what parameters were passed, and how often a method was called [Pivotal, 2013]. Checking if a method was called is a very simple task with Jasmine. An example of how this is achieved is given in Listing 3.33, where a spy is added to an object's method. For example purposes, the method is then directly called, but could be called indirectly by another method under test. This is especially useful when testing complex control flows within methods with subroutines, branches, and object inheritance.

```
describe("Asynchronous specs", function() {
  var value, flag;

  it("should support async execution of test preparation
      and expectations", function() {

    runs(function() {
      flag = false;
      value = 0;
      setTimeout(function() {flag = true;}, 500);
    });

    waitsFor(function() {
      value++;
      return flag;
    }, "The Value should be incremented", 750);

    runs(function() {
      expect(value).toBeGreaterThan(0);
    });
  });
});
```

Listing 3.32: Testing an asynchronous method call. The method under test waits for 500ms and then sets the `flag` to true. The `waitsFor` method is polled until it either returns true or the timeout, in this case 750ms, is reached. the second `runs` method then compares `value` with its expected value. Source code extracted from Pivotal [2013]

```
describe("Spying on method", function(){
  it("Spying on ObjectUnderTest.MethodUnderTest ",
  function() {
    var obj = new ObjectUnderTest();
    spyOn(obj,'MethodUnderTest');
    obj.MethodUnderTest();
    expect(obj.MethodUnderTest).toHaveBeenCalled();
  });
});
```

Listing 3.33: Adding a spy to a method of an object `obj.MethodUnderTest`. The test expects the method to have been called. Source code created by the author.

```
1  var g = 0; //g is a global variable
2
3  function pow(x,y){
4    //x, y are local variables
5    var tmp = 1;  //tmp is a local variables
6    i = 0; // i is not declared => it is GLOBAL
7
8    while(i<y){
9      tmp = tmp*a
10     i++;
11   }
12   return tmp;
13 }
```

Listing 3.34: Undeclared variables are automatically global. The locally used variable `i` automatically becomes global. Source code created by the author.

3.7 JavaScript Best Practice

This chapter will discuss ways of creating high quality JavaScript code, which helps with finding and correcting bugs, understanding the code and better readability. Writing code with these points in mind is essential to modern development, because finding and fixing bugs is very costly and the cost and time required increases over time. Additionally, applications tend to grow and features often have to be added later in the development phase or even after publishing the application. To support this life cycle, it is necessary to be able to relearn the code and understand the underlying ideas a long time after it has been written. Most of the time, the person finding and fixing bugs is not the original author. This has to be taken in to account when writing code and even during the design phase.

These goals can be achieved by defining coding standards, implementing design patterns and writing testable code. These ideas will be demonstrated in the following sections.

3.7.1 Coding Standards

Stefanov [2010, pages 9–21] describes a coding standard (a set of coding conventions) which helps create more stable and maintainable JavaScript code.

Avoid Globals

JavaScript scopes are defined using functions. Any variable declared within a function is local. All variables defined outside of a function are global. Variables in JavaScript do not have to be declared at all. These are then automatically global, which if done without care, can lead to unwanted side effects.

Global variables in JavaScript are stored in the window object in the DOM, which means changing a global variable changes a value in the window element. The example in the listing bellow demonstrates one possible side effect, in which *window.top* is set by mistake, where it is in fact a read-only property. Changing the code to *var top = 42* would fix the problem in this example.

```
1  function foo(x){
2    top = 42 //is global
3    return x*42;
4  }
```

Other problems may occur when using libraries, embedding external resources such as adverts and banners, or simply reusing pieces of code in new applications. In these cases, global variables may interfere with existing code. Global variables also make debugging more complex, because it is not always clear where values come from and when they are set.

Single var Pattern

It is advisable to define all variables at the beginning of each function or block of code:

- There is one single place to look for the variables.
- The use of a variable before declaration is prevented.
- Unwanted use of globals is prevented.

One additional problem with not defining variables at the top of a function is that any declaration of a variable later in the function is treated as if it were declared at the beginning. In other words, a variable declaration anywhere within a function is implicitly moved to the beginning of the function. This behaviour is called hoisting. So, the code in Listing 3.35 will be interpreted as if it were as in Listing 3.36. This causes a problem with a global variable of the same name, since inside the function the local variable will always be referenced, regardless of where it is actually declared. The variable `message` in Listing 3.35 is defined globally. However, within the function `func` it is re-declared on line 4. Due to the effect of hoisting, the actual interpreted code would look more like the code in Listing 3.36, where the local definition of the variable `message` is moved to the top of the function. This results in variable `message` with undefined content in Listing 3.35 on line 3. Hence, it is good practice to always declare all variables at the top of each function.

```
1 message = "global";
2 function func(){
3   alert(message); // "undefined"
4   var message = "local";
5   alert(message); // "local"
6 }
```

Listing 3.35: Scattered vars will be interpreted as in Listing3.36. Source code extracted from Stefanov [2010].

```
1 message = "global";
2 function func(){
3   var message;
4   alert(message); // "undefined"
5   message = "local";
6   alert(message); // "local"
7 }
```

Listing 3.36: The interpreted version of the code in Listing 3.35. Source code extracted from Stefanov [2010].

eval() is Evil

Eval is used to execute a string as if it were JavaScript. This may seem useful, but brings major problems. Usually, if the string or code is known in advance, and it is not generated at runtime, there is no need to use eval(). If the code is generated dynamically there are usually better ways of dealing with it. The example shown in the listing below shows one way of avoiding `eval()`, by using the bracket notation instead of the "." notation. The most serious issue, however, regards security. By using eval(), code becomes vulnerable to the execution of tampered code.

```
1 var property = "name";
2 alert(eval("object."+property)); //bad
3 alert(object[property]); //better
```

A similar problem occurs when passing arguments to *.setTimeout* or *.setInterval* as a string. The listing below demonstrates how to prevent unwanted behaviour, by passing a function reference instead of a function name.

```
1 setTimeout("myCallback()", 1000); // Bad
2 setTimeout(myCallback, 1000); //
```

3.7.2 Design Patterns

A design pattern is a reusable solution for a commonly reoccurring problem within a given context. Design patterns help in finding well-tested solutions to design problems, and create a *vocabulary* to discuss design issues. Stefanov [2010] describes numerous JavaScript patterns, the most common ones are summarised below.

Namespace Pattern

One problem of JavaScript is the lack of namespaces. This often leads to many global objects. The three objects created in the listing below are all globally accessible.

```
1 var a;
2 var object ={};
3 function do{};
```

Fortunately, it is very easy to add namespace functionality to JavaScript. This is achieved by adding one global element and then adding all further objects to this one global element. The code from above listing is re-factored in the listing below, adding all objects to the global `MYAPP` object.

```
1 var MYAPP={};
2 MYAPP.a;
3 MYAPP.object ={};
4 MYAPP.do =function{};
```

Usually, the global object is written in all upper case to quickly identify the namespace. This approach however has two major drawbacks:

- It adds a prefix to every object, which increases the code size.
- One global object means that it can be modified at any point, and all of the application inherits the changes.

```
1  MYAPP.namespace('MYAPP.element1.child');
2
3  MYAPP.namespace = function(ns_string){
4    var parts = ns_string.split('.'),
5    parent = MYAPP,
6    i;
7
8    //trip redundant leading global
9    if(parts[0]==="MYAPP"){
10     parts=parts.slice(1);
11   }
12
13   //create the properties
14   for(i=0;i<parts.length;i+=1){
15     //create the property if it does not exist
16     if(typeof(parent[parts[i]])=="undefined"){
17       parent[parts[i]]={}
18     }
19     parent = parent[parts[i]];
20   }
21   return parent;
22 }
```

Listing 3.37: Using a namespace function prevents the declaration of the same property more than once. Source code adapted from Stefanov [2010].

When applications grow and are spread across multiple files it is necessary to prevent multiple declarations of the same global object or namespace. This can be achieved by using a namespace function. The code in Listing 3.37 shows the implementation of the namespace function suggested by Stefanov [2010]. Whenever a new property is added to the namespace using the namespace function, it is first checked for non-existence and only then created.

Module Pattern

Organising code into workable sections and encapsulating them in a way that makes the code reusable and maintainable can be very difficult when using a language such as JavaScript, due to the lack of language support for packages. This problem is nicely tackled by the module pattern. To create a module one first needs a namespace within which to operate. This is achieved using the namespace pattern described above. Then the module's methods and variables can be defined private and public declarations are also possible. The setup of a module is shown in Listing 3.38 consisting of one private variable *privateNumber*, one private method *privateMethod*, and two public methods *method1* and *method2*.

Model View Controller (MVC) Pattern

The Model-View-Controller (MVC) pattern is used to structure code by separating the interface from the logic and the data [Wikipedia, 2013b]. The MVC consists of three components:

- Controller
  The controller holds all the program logic. It makes calls to remote resources and updates the model.
- Model

```
1  MYMODULE.namespace('MYMODULE.utils'); //define the MYMODULE.utils
       namespace
2
3  //create the module
4  MYMODULE.utils =(function(){
5
6    //private properties
7    var privateNumber = 0;
8    var privateMethod = function(){...};
9
10   //public API
11   return{
12     method1: function(){...},
13     method2: function(){...};
14   }
15 });
```

Listing 3.38: Defining a module's namespace and adding private variables and public methods. Source code adapted from Stefanov [2010].

The model is responsible for holding the data and informing the view when the data or the state of the model changes.

- View
  The view is the element with which the user interacts. It displays data coming from the model and calls methods from the controller.

A class diagram of the MVC pattern can be seen in Figure 3.1

Web Workers

Modern web browsers offer a new way of doing heavy computation on the client by enabling multi-threading. These constructs are called web workers and are defined as a HTML5 JavaScript API [W3C, 2012a]. A web worker accepts a JavaScript script as input, which is then executed in a new thread. The thread can communicate with the calling script using *postMessage()* Web workers are supported in most of the widely used mobile and desktop browsers, see Deveria [2013] for a full list. The code in Listing 3.39 shows an example of how Web Workers can be used.

3.7.3 Writing Testable JavaScript

Cherry [2010] developed a number of principles for writing testable JavaScript code.:

- *Avoid Singletons:* A singleton object is a object for which there is always only one instance, meaning that `new` will always return the same instance of the object. Singleton objects can be difficult to test, since each test may modify the internal state, and make the result of further tests unpredictable. For instance, if one wants to test the adding of an element to an array, this will have a direct impact on any subsequent test that checks for the content or the length of the singleton array.

- *Avoid Private Methods:* This is a fairly straightforward idea. Any code that is hidden from the test suite cannot be tested. A better approach would be to make the methods public but mark them

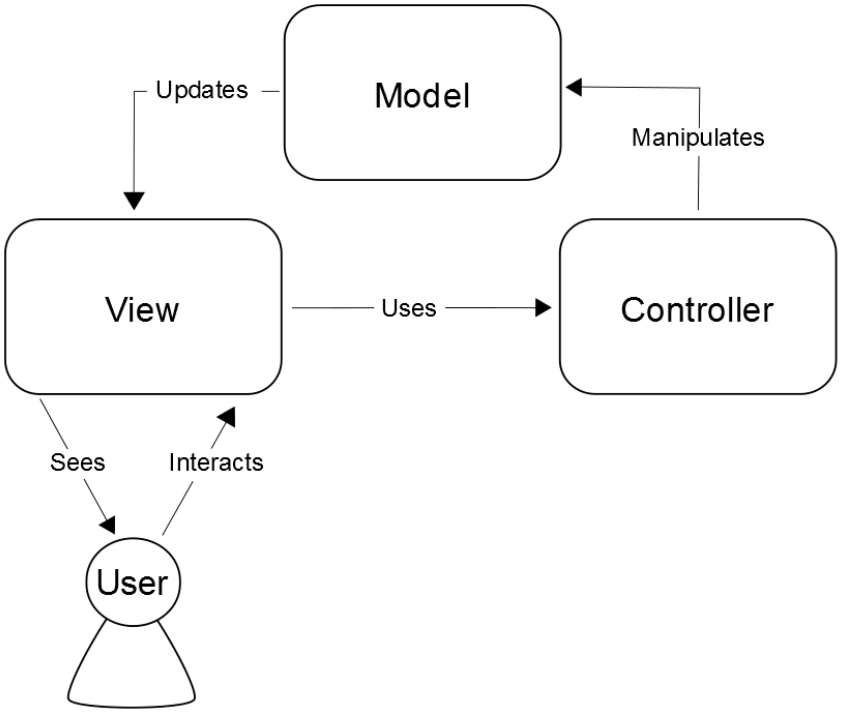


Figure 3.1: The MVC pattern consists of the model, view, and controller. The model holds all the data and informs the view when data or the state of the model has changed. The view is responsible for displaying the data and providing means of interaction. It calls methods from the controller which holds the business logic and manipulates the data in the model. Image created by the author.

```
1  var worker = new Worker('my_long_running_Script.js');
2  worker.onmessage = function(event){
3    document.body.innerHTML += event.data+"<br />";
4  }
5
6
7  //the content of  my_long_running_Script.js
8
9  var count =0;
10 while(count < Number.MAX_VALUE){
11   postMessage("count = "+count);
12   count++;
13 }
14 postMessage("done");
```

Listing 3.39: Creating a new web worker and passing `my_long_running_Script.js` to its constructor. The executed script can pass data to the calling script using the `postMessage()` method. Source code created by the author.

in a way that indicates their `private` state. One way of marking them would be by adding an underscore (_) prefix to the method name.

- *Use Light-Weight Functions:* Testing large methods which incorporate a great deal of functionality is difficult. Large methods also become very hard to read and thus hard to debug. It is advised to split large methods in to smaller sub-methods, each incorporating a subset of the logic. This way the code is easier to read and test.

Chapter 4

Web-Based Graphics

Up until 1996, images such as JPEGs and GIFs were the only way of adding graphics to a web site. Interaction with these images was not possible since they were static images or simple frame-based animations. With the introduction of flash in 1996 vector graphics and interactive animations were made possible [EoW, 2010]. However, a plug-in was required. Only with the introduction of HTML5 in 2008 were new and more flexible ways of creating and interacting with graphics made possible [W3C, 2013]. Additionally, new technologies have emerged and have been implemented in many modern browsers. These use non-HTML elements such as SVG, and some even support hardware-acceleration like WebGL. This chapter gives a brief overview of current technologies, and shows how these can be used in web pages.

Table 4.1 shows the availability and openness of the technologies discussed in the following sections.

4.1 Flash

Flash is an authoring tool which also comes with a flash player. A Flash player can be used to display Flash content as a standalone application or embedded in other software such as a web browser using plug-ins. For many years, Flash was the only way to add rich interaction and animation to web sites, but is now being replaced by technologies natively implemented in browsers. Creating content using Flash requires an authoring tool such as Adobe Flash Professional. Flash combines vector graphics with other multimedia resources like videos and audio to create a wide variety of content. Together with its own scripting language `ActionScript`, it is very powerful for creating interactive content for any platform. Some of the main features of flash [Veer, 2006] include:

- *Drawings and animations*: With Flash it is fairly simple to create vector-based graphics and animations using tween and key-frame based animation.

	Internet Explorer	Chrome	Firefox	Opera	iOS	Android	Plugin	Open Standard
Canvas	9+	✓	✓	✓	✓	✓	✗	✓
CSS3	~	~	~	~	✓	✓	✗	✓
SVG	9+	✓	✓	✓	✓	✓	✗	✓
WebGL	11+	18+	~	15+	✗	~	✗	✓
Flash	✓	✓	✓	✓	✗	~	✓	✗

Table 4.1: A comparison of web-based graphics technologies

```
1  <?xml version="1.0" standalone="no"?>
2  <!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
3  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
4  <svg viewBox="0 0 100 100" version="1.1"
5    xmlns="http://www.w3.org/2000/svg" >
6
7    <rect width="100" height="100" style="fill:#cccccc;stroke-width:1;
       stroke:#000000" />
8    <circle cx="50" cy="50" r="35" style="fill:#FFFF00;stroke-width:1;
       stroke:#000000" />
9    <circle cx="35" cy="40" r="5" style="fill:#000000;stroke-width:5;stroke
       :#000000" />
10   <circle cx="65" cy="40" r="5" style="fill:#000000;stroke-width:5;stroke
       :#000000" />
11   <path d="M 35 65 q 15 15 30 0" stroke="black" stroke-width="9" fill="
       none" />
12 </svg>
```

Listing 4.1: An SVG graphic. Each element of the graphic is represented by its own SVG element. One rectangle for the background, one circle for the yellow circle, two circles for the black dots, and one path for the arc. Source code created by the author.

- *Multimedia web sites*: Embedding animations, videos and audio in Flash makes it very easy to create multimedia web sites.

- *Banner ads*: Most banner ads currently used in the web are made using Flash. These ads are usually embedded via third party web sites.

- *Games and other applications*: Using ActionScript it is possible to create whole applications in Flash. Most common are small web-based games.

Since Apple has announced not to support Flash on any of their mobile devices, Flash has become less used, and is being replaced by more modern HTML5 technologies.

4.2 SVG

SVG is an open language, maintained by the W3C, for creation of 2d scalable vector graphics. Since it is not a technology aimed purely at browsers, the resulting SVG can be stores as a separate `.svg` file. It has its own Document Object Model making it compatible with other languages such as JavaScript, CSS and HTML. This makes it possible to embed SVN elements within the DOM of the browser. In particular, its compatibility with CSS and JavaScript makes it very flexible, enabling interaction and animations on web pages using SVG only. SVG is vector-based, meaning that instead of having colour values for each pixel (as in raster graphics), the graphics are composed of basic shapes. This enables unlimited scaling without resulting in jagged edges [Dailey, Frost, and Strazzullo, 2012, Chapter 1]. A full description and the specifications of SVG can be found in W3C [2011]. The code in Listing 4.1 shows how to create the same graphic as in Section 4.3.1. The result of Listing 4.1 can be seen in Figure 4.1.

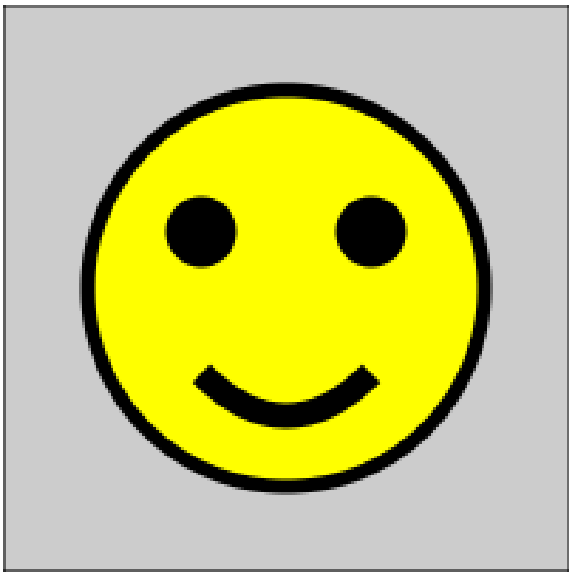


Figure 4.1: The resulting graphic created with SVG as described in Listing 4.1.

4.3 HTML5

HTML5 is the fifth major version of the Hypertext Markup Language (HTML). The term HTML5 is often also used to describe web applications that use the new HTML5, CSS3 standards, and JavaScript. HTML5 introduces new elements for multimedia such as `<audio>` and `<video>` which define sound and video contents. New form elements have been introduced to better define form fields. The majority of the new additions in HTML are in the category of semantic or structural elements, which are used to structure the document. The most powerful new element however is the canvas element, which allows drawing in the browser using JavaScript. Together with the new CSS3 specification and new JavaScript APIs, HTML5 is now capable of creating rich multimedia web-based graphical interfaces and web-based graphics.

4.3.1 Canvas

Canvas is a HTML5 element that enables the developer to create 2d graphics and animations using JavaScript. For 3d graphics with Canvas, WebGL is required, which is described in Section 4.3.3. The graphic generated by JavaScript is converted to a bitmap and drawn resolution-dependant by the browser. This implies, that it is not possible to create graphics that are in a size relative to the browser window, or that rescale when the browser window changes size. However, this can be overcome by using JavaScript to manually re-scale the images and redraw them once the scaling has finished. Rescaling is also possible by using CSS, this will result in a bad resolution, when scaled to much. Most JavaScript graphics libraries described in Chapter 5 provide a way of achieving this. Adding JavaScript event handlers enables the interaction and manipulation of these graphics, which allows highly sophisticated user interfaces to be created. This even enables the creation of graphically intense web-based games developed purely in JavaScript. Canvas does not require any browser plug-in and is supported by all modern browsers, as shown in Table 4.1 [Sheridan, 2013]. The example in Listing 4.2 shows how to create the image in Figure 4.2 using JavaScript and Canvas.

Most modern browsers will automatically render the canvas using the graphics card. This increases the performance of animations and decreases the time needed to display the graphics. Browsers use Direct2D and DirectWrite to perform the operations on the graphics card [Sheridan, 2013].

```
<html>
    <head>
        <title>Canvas example</title>
    </head>
<body>
    <canvas id="ca" width="200" height="200"></canvas>
    <script>
        var c=document.getElementById("ca");
        var cc=c.getContext("2d");

        //create rectangle to fill canvas
        cc.fillStyle="#cccccc";
        cc.fillRect(0,0,200,200);

        //create the large circle
        cc.arc(100,100,70,0,2*Math.PI);
        cc.fillStyle = '#FFFF00';
        cc.fill();
        cc.lineWidth = 5;
        cc.strokeStyle = 'black';
        cc.stroke();

        //create left eye
        cc.beginPath();
        cc.arc(70,80,10,0,2*Math.PI);
        cc.fillStyle = 'black';
        cc.fill();
        cc.lineWidth = 5;
        cc.strokeStyle = 'black';
        cc.stroke();

        //create right eye
        cc.beginPath();
        cc.arc(130,80,10,0,2*Math.PI);
        cc.fillStyle = 'black';
        cc.fill();
        cc.lineWidth = 5;
        cc.strokeStyle = 'black';
        cc.stroke();

        //create mouth
        cc.beginPath();
        cc.arc(100,120,30,0.1*Math.PI,0.9*Math.PI);
        cc.lineWidth = 9;
        cc.strokeStyle = 'black';
        cc.stroke();
    </script>
</body>
</html>
```

Listing 4.2: Creating a simple graphic using canvas. First, the canvas is defined and the size is set to 200x200 pixel. Since canvas uses bitmap graphics, absolute sizes have to be used. Then three circles are created using the arc method. Finally the arc method is used to create a semi-circle to create the mouth. Source code created by the author.

Figure 4.2: The graphic created in JavaScript using canvas shown in Listing 4.2.

4.3.2 CSS3

An alternative way of creating graphics in HTML5 is using standard HTML elements and styling them using the new CSS3 properties. These include rounded corners, box and text shadows, gradients and the possibility to embed custom fonts. Together with element positioning it is possible to create simple graphics with CSS3. This method uses very little bandwidth and can be used to generate graphics that are scaled relative to the window size. However this process requires a great deal of time to design good looking graphics. The example in Listings 4.3 and 4.4 shows how to create the image in Figure 4.3 using spans and CSS3 properties such as `border-radius` and positioning using margins, paddings and floats. Like SVG (described in Section 4.2), the great advantage of this method is that each element of the graphic is also an HTML element, which can be manipulated using JavaScript and CSS for interaction and animations.

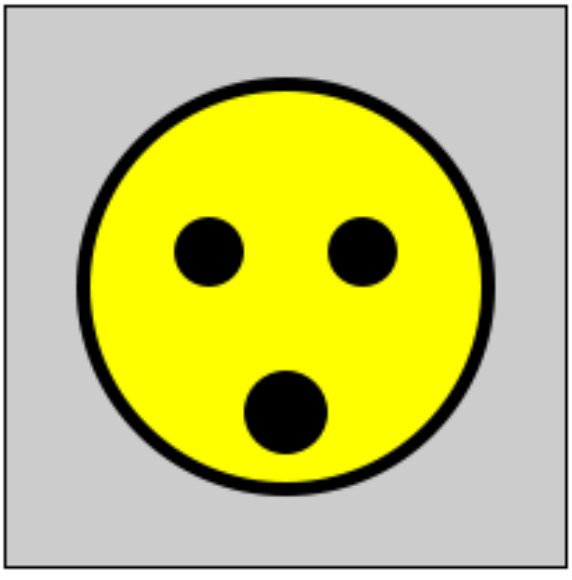


Figure 4.3: The graphic created using the CSS shown in Listing 4.3.

```
5               span.bg{
6                   width:100%;
7                   height:100%;
8                   background:#cccccc;
9                   border:0.1em solid black;
10                  display:block;
11                  margin:0;
12                  padding:0;
13              }
14              span.face {
15                  display: block;
16                  width: 70%;
17                  height: 70%;
18                  margin-top: 12%;
19                  margin-left: 12%;
20                  padding: 0;
21                  background: #FFFF00;
22                  border: 0.5em solid black;
23                  -moz-border-radius:55%;
24                  -webkit-border-radius: 55%;
25                  border-radius:55%;
26              }
27              span.eyeLeft, span.eyeRight{
28                  float:left;
29                  display:block;
30                  width:12%;
31                  height:12%;
32                  margin-top:25%;
33                  margin-left:25%;
34                  padding:0;
35                  background:#000000;
36                  border:0 solid black;
37                  -moz-border-radius:55%;
38                -webkit-border-radius:55%;
39                border-radius:55%;
40              }
41              span.mouth{
42                  float:left;
43                  display:block;
44                  width:25%;
45                  height:25%;
46                  margin-top:25%;
47                  margin-left:37%;
48                  padding:0;
49                  background:#000000;
50                  border:0 solid black;
51                  -moz-border-radius:55%;
52                -webkit-border-radius:55%;
53                border-radius:55%;
54              }
```

Listing 4.3: The five spans are nested and then layouted using CSS3. Using margins, rounded corners, `display:block`, and `float:left` it is possible to create the image as shown in Figure 4.3. The corresponding HTML can be seen in Listing 4.4. Source code created by the author.

```
57 <body>
58     <span class="bg"><span class="face">
59             <span class="eyeLeft"></span>
60             <span class="eyeRight"></span>
61             <span class="mouth"></span>
62     </span></span>
63 </body>
```

Listing 4.4: The HTML for the example in Listing 4.3.

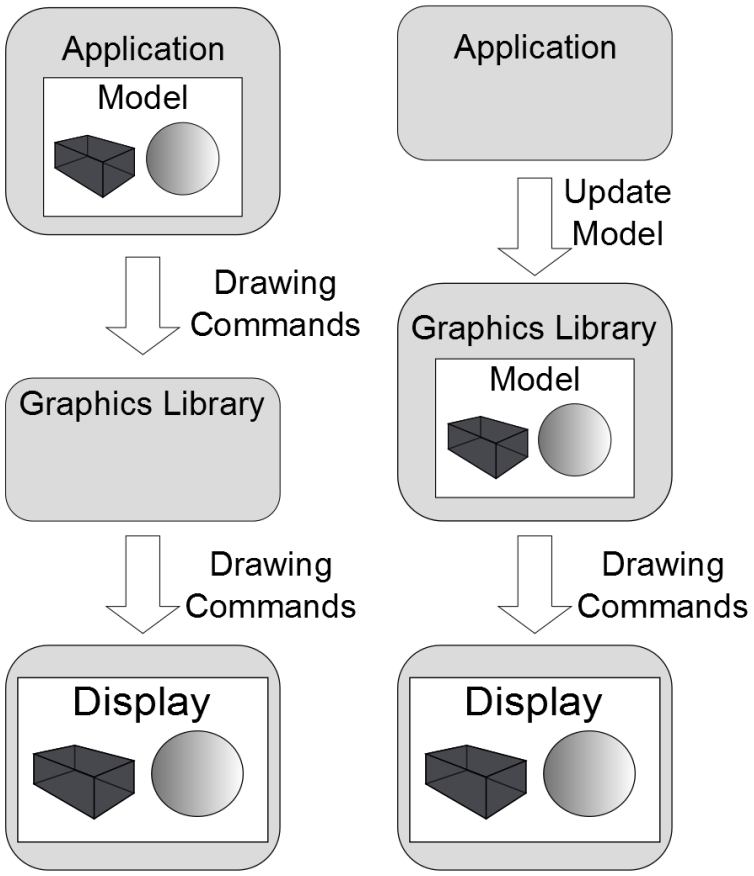
4.3.3 WebGL

WebGL is a JavaScript API that provides 3d graphics on the web [Leung and Salga, 2010]. It is based on the OpenGL standard for 3d computer graphics, and is being developed by the KHRONOS group which consists of most browser vendors including Apple, Google, Mozilla, and Opera. It uses the HTML5 Canvas object to render, which is described in Section 4.3.1. The main advantages of WebGL are:

- WebGL is fully hardware-accelerated which means it uses the full potential of the graphics card. This gives applications a great speed improvement, if graphics hardware is present.
- WebGL does not require any plug-ins. All current browsers except Internet Explorer support WebGL. WebGL support has been announced for IE11.

With it is possible to create highly complex 3d applications and interfaces. This enables a new way of creating content for the web [Anyuru, 2012, Chaper 1].

WebGL uses an immediate-mode API, which means that the application needs to keep the whole scene in memory, and has to redraw the whole image for every frame. This makes the API very flexible, but also requires significant hardware resources. In comparison, retained-mode APIs hold the scene in the library and can decide when to draw something. One example of a retained-mode API is SVG written to the DOM (seen in Section 4.2). A comparison between the two modes is briefly illustrated in Figure 4.4. The example in Listing 4.5 shows how to set up WebGL. It initialises the required shaders and buffers, creates the Canvas context on which to draw, and sets up a simple scene with a white triangle on a black background (shown in Figure 4.5). As can be seen, there is much to be done before one can actually start creating 3d scenes. Therefore, it is recommended to use some sort of library. Section 5.5 introduces Three.JS, a 3d graphics framework built on top of WebGL.



(a) In an immediate-mode API, the application hands the whole scene to the graphics API. This results in a redrawing of the whole scene each frame. [Anyuru, 2012]

(b) In an retained-mode API, the scene is kept in the application, and only changes of the scene are passed to the API. This enables the application to choose when the image is updated. [Anyuru, 2012]

Figure 4.4: Comparison between immediate-mode and retained-mode APIs. An immediate-mode API requires the application to have the whole scene in the memory, while a retained-mode API keeps the scene in the library. Immediate-mode has to redraw the whole scene every frame, whereas retained-mode can choose when to update the image itself as necessary. Images redrawn from [Anyuru, 2012]

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Listing 2-1, A First WebGL Example</title>
<meta charset="utf-8">
<script type="text/javascript">
var gl;
var canvas;
var shaderProgram;
var vertexBuffer;
function createGLContext(canvas) {
  var names = ["webgl", "experimental-webgl"];
  var context = null;
  for (var i=0; i < names.length; i++) {
```

```
15      try {
16         context = canvas.getContext(names[i]);
17      } catch(e) {}
18      if (context) {
19         break;
20      }
21    }
22    if (context) {
23      context.viewportWidth = canvas.width;
24      context.viewportHeight = canvas.height;
25    } else {
26      alert("Failed to create WebGL context!");
27    }
28    return context;
29  }
30  function loadShader(type, shaderSource) {
31    var shader = gl.createShader(type);
32    gl.shaderSource(shader, shaderSource);
33    gl.compileShader(shader);
34    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
35         alert("Error compiling shader" + gl.getShaderInfoLog(shader));
36         gl.deleteShader(shader);
37         return null;
38    }
39    return shader;
40  }
41  function setupShaders() {
42    var vertexShaderSource =
43       "attribute vec3 aVertexPosition;                \n" +
44       "void main() {                                  \n" +
45       " gl_Position = vec4(aVertexPosition, 1.0);      \n" +
46       "}                                              \n";
47      var fragmentShaderSource =
48        "precision mediump float;                    \n"+
49        "void main() {                               \n"+
50        " gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);   \n"+
51        "}                                           \n";
52    var vertexShader = loadShader(gl.VERTEX_SHADER, vertexShaderSource);
53    var fragmentShader = loadShader(gl.FRAGMENT_SHADER,
         fragmentShaderSource);
54    shaderProgram = gl.createProgram();
55    gl.attachShader(shaderProgram, vertexShader);
56    gl.attachShader(shaderProgram, fragmentShader);
57    gl.linkProgram(shaderProgram);
58    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
59       alert("Failed to setup shaders");
60    }
61    gl.useProgram(shaderProgram);
62    shaderProgram.vertexPositionAttribute =
63    gl.getAttribLocation(shaderProgram, "aVertexPosition");
64  }
65  function setupBuffers() {
66    vertexBuffer = gl.createBuffer();
67    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
68    var triangleVertices = [
69            0.0, 0.5, 0.0,
70           -0.5, -0.5, 0.0,
```

```
71              0.5, -0.5, 0.0
72      ];
73      gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
74      gl.STATIC_DRAW);
75      vertexBuffer.itemSize = 3;
76      vertexBuffer.numberOfItems = 3;
77    }
78    function draw() {
79      gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
80      gl.clear(gl.COLOR_BUFFER_BIT);
81      gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
82                             vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
83      gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
84      gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);
85    }
86    function startup() {
87      canvas = document.getElementById("myGLCanvas");
88      gl = createGLContext(canvas);
89      setupShaders();
90      setupBuffers();
91      gl.clearColor(0.0, 0.0, 0.0, 1.0);
92      draw();
93    }
94    </script>
95    </head>
96    <body onload="startup();">
97      <canvas id="myGLCanvas" width="500" height="500"></canvas>
98    </body>
99    </html>
```

Listing 4.5: Each WebGL application needs a vertex shader and a fragment shader. The shaders here are defined in-line and are kept as simple as possible. First, create the WebGL context on the specified Canvas element. For this, the `webgl` context is tried first, if that does not work the `experimental-webgl` context is used. The buffer holding the vertices is created. The `draw` method sets the parameters required for rendering the actual scene and draws the triangle defined in the vertex buffer. Finally, the `startup` method binds it all together and calls the methods in the appropriate order. This source code is extracted from [Anyuru, 2012, Chapter 1].

4.4 Summary

In summary, one can say that all graphics technologies have their advantages and disadvantages. CSS3 is not suitable for complex graphics, but can be used for simple decorations. Flash is no longer optimal due to its requirement of a plug-in, which may not be available on all platforms and browsers, and the fact that most developers are trying to remove plug-ins from their browsers. When it comes to rendering speed, SVG is not very suitable for animations, due to its slow render time. Canvas performs slightly better, but is still slower than WebGL when it comes to animations, due to the fact, that WebGL can use the graphics card, which is optimised for this purpose.

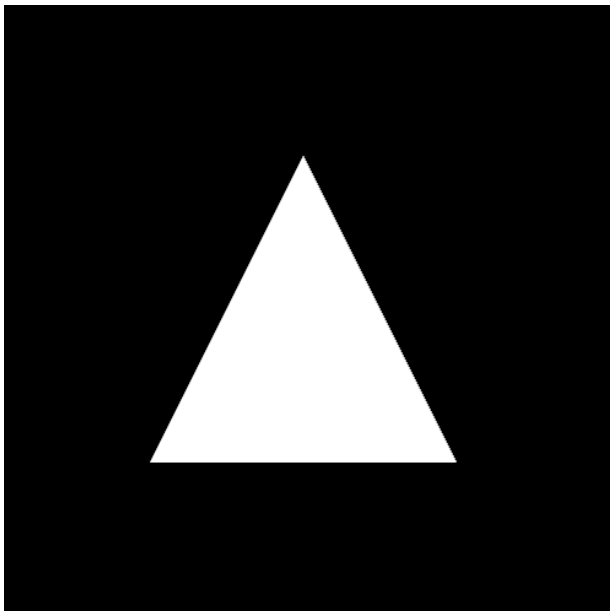


Figure 4.5: The graphic created using the WebGL code in Listing 4.5.

Chapter 5

JavaScript Graphics Libraries

As described in Chapter 4 there are many ways of creating web-based graphics. However, as was also shown, the process of creating these graphics can be rather complex. To tackle this problem, many specialised JavaScript libraries have been developed over the last few years. This chapter gives a short overview over some of the different libraries by showing small examples. The libraries can be split into two categories: 2d and 3d. The 2d can further be categorised into Canvas-based and SVG-based libraries.

5.1 EaselJS (2D)

CreateJS is a suite of multiple libraries, each independently usable but designed to work perfectly together [CreateJS, 2013]. For this section, only the EaselJS library responsible for graphics will be discussed by giving short examples.

EaselJS is a library developed to make the use of the HTML5 canvas element easier. It does so by providing a JavaScript API to create basic shapes, manipulate them, add animations, and respond to UI events. A very simple example of an image created with EaselJS is given in Listing 5.1. Nine rectangles are drawn, creating a simple bar chart shown in Figure 5.1. It can be seen, that the code is almost identical to the pure Canvas. However, new primitive shapes have been added, which can be inherited or changed using `.prototype` as shown in Listing 5.2.

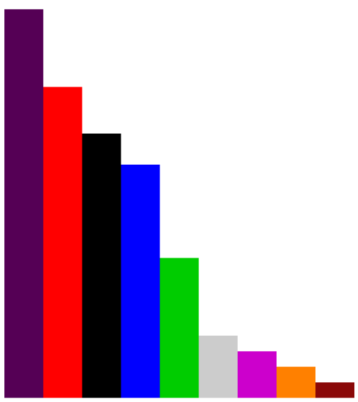


Figure 5.1: The resulting graphic created with EaselJS from the code in Listing 5.1.

```
1  <head>
2      <script src="http://code.createjs.com/easeljs-0.6.0.min.js"></script>
3      <script>
4          function init() {
5              var width = 25;
6              var heightscale = 10;
7              var stage = new createjs.Stage("can");
8
9              var bar1 = new createjs.Shape();
10             bar1.graphics.beginFill("#550055").drawRect(0, 25 *
                   heightscale, width, -25 * heightscale);
11             stage.addChild(bar1);
12
13             var bar2 = new createjs.Shape();
14             bar2.graphics.beginFill("#ff0000").drawRect(width, 25 *
                   heightscale, width, -20 * heightscale);
15             stage.addChild(bar2);
16
17             var bar3 = new createjs.Shape();
18             bar3.graphics.beginFill("#000000").drawRect(width * 2, 25 *
                   heightscale, width, -17 * heightscale);
19             stage.addChild(bar3);
20
21             var bar4 = new createjs.Shape();
22             bar4.graphics.beginFill("#0000ff").drawRect(width * 3, 25 *
                   heightscale, width, -15 * heightscale);
23             stage.addChild(bar4);
24             var bar5 = new createjs.Shape();
25             bar5.graphics.beginFill("#00cc00").drawRect(width * 4, 25 *
                   heightscale, width, -9 * heightscale);
26         .
27         .
28         .
29             stage.update();
30         }
31     </script>
32 </head>
33 <body onLoad="init();">
34     <canvas id="can" width="225" height="250"></canvas>
35 </body>
36 </html>
```

Listing 5.1: Creating a simple image using EaselJS. As can be seen, the syntax is similar to pure Canvas code. The only difference is that EaselJS provides primitives to create the shapes. Nine rectangles are drawn, resulting in a simple bar chart. The size of the resulting image is defined by the canvas element in line 42 or can be defined using CSS. Bars 5-9 have been omitted from the code. The result can be seen in Figure 5.1. Source code created by the author.

```
1     <script>
2         function init() {
3             var stage = new createjs.Stage("can");
4
5             var RCircle = function(radius){
6                 this.initialize(radius);
7             }
8
9             RCircle.prototype = new createjs.Shape();
10            RCircle.prototype.Shape_initialize = RCircle.prototype.
                  initialize;
11            RCircle.prototype.initialize = function(radius) {
12                this.Shape_initialize();
13                this.graphics.beginFill("red").drawCircle(0,0,radius);
14            }
15
16            var circle = new RCircle(70);
17            circle.x=100;
18            circle.y=100;
19            stage.addChild(circle);
20
21            stage.update();
22        }
23    </script>
```

Listing 5.2: Creating a custom shape which implements a circle, but has its colour coded in the constructor. First, `Shape` of of the basic `Shape` class is copied to the `.prototype` of `RCircle` and then the `initialize` method is extended by defining a circle and setting its colour. The constructor of this new `RCircle` class takes one parameter which defines the radius of the circle. Source code created by the author.

```
1      <script>
2          function init() {
3              var width = "11%";
4              var stage = Raphael("container","90%","90%");
5
6              var bar = stage.rect(0, 0, width, "100%");
7              bar.attr("fill", "#550055");
8              var bar2 = stage.rect("11%", "20%", width, "80%");
9              bar2.attr("fill", "#ff0000");
10             var bar3 = stage.rect("22%", "32%", width, "68%");
11             bar3.attr("fill", "#000000");
12             var bar4 = stage.rect("33%", "40%", width,"60%");
13             bar4.attr("fill", "#0000ff");
14             var bar5 = stage.rect("44%", "64%", width, "36%");
15             bar5.attr("fill", "#00cc00");
16             var bar6 = stage.rect("55%", "84%", width, "16%");
17             bar6.attr("fill", "#cccccc");
18             var bar7 = stage.rect("66%", "88%", width, "12%");
19             bar7.attr("fill", "#cc00cc");
20             var bar8 = stage.rect("77%", "92%", width, "8%");
21             bar8.attr("fill", "#ff8000");
22             var bar9 = stage.rect("88%", "96%", width, "4%");
23             bar9.attr("fill", "#8A0808");
24         }
25     </script>
26 </head>
27 <body onLoad="init();">
28     <div id="container">
29     </div>
30 </body>
31 </html>
```

Listing 5.3: Creating a simple bar chart using Raphaël. First the stage is created inside the `container` with 100% width and 100% height. This creates an empty SVG image. Then nine squares are drawn, resulting in a simple bar chart. Source code created by the author.

5.2 Raphaël (2D)

Raphaël is a JavaScript library with the aim of simplifying work with vector graphics in 2d [Raphaël, 2013]. It uses the SVG element to create graphics. Using SVG has the benefit of each graphical element being a DOM element, which behaves like any other regular DOM element. This makes it possible to add event handlers to the elements and manipulate them easily using JavaScript and CSS. The example in Listing 5.3 shows how to set up a stage, and connect it to a HTML element. Then, a simple bar chart is drawn using relative sizes shown in Figure 5.2.

Additionally, Raphaël provides a simple way of creating animations with `.animate({"attr":value }, time, easing, callback)` which changes a given `attr` to the given `value` over time defined in `time`. The attributes to be changed can be defined as a JSON object enabling multiple attributes to be animated at the same time with only one method call. Additionally, an easing function can be defined, which defines the exact behaviour of the animation. A callback method can be defined which is executed at the end of the animation.

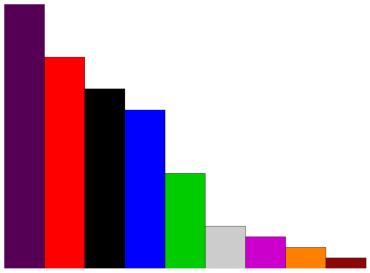


Figure 5.2: The resulting graphic created with Raphaël from the code in Listing 5.3.

5.3 Paper.js (2D)

Paper.js is a JavaScript framework for creating vector based graphics on top of the HTML5 canvas element [Lehni and Puckey, 2014]. Using vector based graphics on the canvas element, enables creating nice crisp images. However, since canvas is a bitmap based, rescaling of the image has to be done manually to maintain a nice image. Paper.js provides a way of scaling objects by calling the `onResize(event)` method. Paper.js graphics are ordered in layers and groups, making it very flexible to interact and change elements. The difference between layers and groups, is that new elements are placed on the current active layer, and can then be grouped in to arbitrary groups. Each Paper.js project has at least one layer, but additional ones can be added and selected as being active. This paradigm is also well known from other applications such as GIMP, Photoshop and Flash. It has the advantage of enabling transformation of a whole set of objects at once. Listing 5.4 shows how to create a simple bar chart, and how to implement scaling. The resulting image is shown in Figure 5.3.

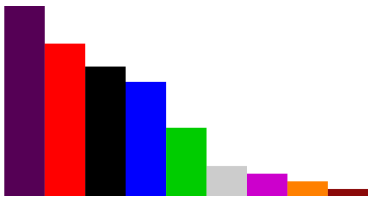


Figure 5.3: The resulting graphic created with Paper.js from the code in Listing 5.4.

5.4 Pixi.js (2D)

Pixi.js is a JavaScript library for 2d graphics that uses the WebGL API for rendering [Groves, 2014]. Additionally, it has a canvas fallback in case WebGL is not supported. The use of WebGL in 2d graphics allows for high performance by taking advantage of the graphics cards computing power. The main features of Pixi.js are:

- *WebGL Renderer*: The WebGL renderer uses the graphics card for rendering, if hardware is present.
- *Canvas Renderer*: The canvas renderer is used in case WebGL is not supported.
- *Scene Graph*: The scene graph is used to organise the scene object in a hierarchy. This makes it easier to modify parts of the scene.
- *Interaction*: Full mouse and multi-touch interaction is implemented. This makes it possible to create interaction for devices using a mouse and a touch-screen.

The source code in Listing 5.5 demonstrates how to create a simple bar chart.

```
<script type="text/paperscript" canvas="canvas">
    var size = view.size;
    var width=size.width / 9;
    var heightscale = (size.height / 25);

    var scaleXOld = size.width;
    var scaleYOld = size.height;

    var bar1 =  new Rectangle(0, 25 * heightscale, width, -25 *
       heightscale);
    var path1 = new Path.Rectangle(bar1);
    path1.fillColor = '#550055';

    var bar2 =  new Rectangle(width, 25 * heightscale, width, -20 *
       heightscale);
    var path2 = new Path.Rectangle(bar2);
    path2.fillColor = '#ff0000';

    var bar3 =  new Rectangle(width * 2, 25 * heightscale, width, -17 *
       heightscale);
    var path3 = new Path.Rectangle(bar3);
    path3.fillColor = '#000000';

    var bar4 =  new Rectangle(width * 3, 25 * heightscale, width, -15 *
       heightscale);
    var path4 = new Path.Rectangle(bar4);
    path4.fillColor = '#0000ff';
  .
  .
  .
    function onResize(event) {
            var sizeNew = view.size;
            var scaleX= sizeNew.width / scaleXOld;
            var scaleY = sizeNew.height / scaleYOld;

          project.activeLayer.scale(scaleX, scaleY, project.activeLayer.
             bounds.topLeft);

            scaleXOld = sizeNew.width;
            scaleYOld = sizeNew.height;
    }
</script>
</head>
<body>
  <canvas id="canvas" resize></canvas>
</body>
</html>
```

Listing 5.4: Creating a simple bar chart using Paper.js. First the size of the new image is initialised, using the `view.size` property. Then the nine rectangles are created and drawn as paths. For rescaling the scene, the `onResize` method is implemented. The `onResize` method scales the whole active layer (the only layer available), and scales it relative to the upper left corner. The bars 5 to 9 where omitted from the code. The resulting image is shown in Figure 5.3. Source code created by the author.

```
1     <script>
2
3      var defaultWidth = window.innerWidth;
4      var defaultHeight = window.innerHeight;
5
6      var width = defaultWidth/9;
7      var heightscale = defaultHeight /25
8
9      var stage = new PIXI.Stage(0xffffff);
10     var renderer = PIXI.autoDetectRenderer(defaultWidth, defaultHeight);
11
12     document.body.appendChild(renderer.view);
13     var graphics = new PIXI.Graphics();
14
15     stage.addChild(graphics);
16
17     graphics.beginFill(0x550055);
18     graphics.drawRect(0, 25 * heightscale, width, -25 * heightscale);
19
20     graphics.beginFill(0xff0000);
21     graphics.drawRect(width, 25 * heightscale, width, -20 * heightscale);
22
23     graphics.beginFill(0x000000);
24     graphics.drawRect(width * 2, 25 * heightscale, width, -17 *
          heightscale);
25
26     graphics.beginFill(0x0000ff);
27     graphics.drawRect(width * 3, 25 * heightscale, width, -15 *
          heightscale);
28
29     graphics.beginFill(0x00cc00);
30     graphics.drawRect(width * 4, 25 * heightscale, width, -9 *
          heightscale);
31    .
32    .
33    .
34     renderer.render(stage);
35     </script>
36  </body>
37  </html>
```

Listing 5.5: Creating a simple bar chart using Pixi.js. First the size of the image is calculated (in this case 100% of the window size). Then the stage is set up, and the renderer created. Finally, nine rectangles are created using the `drawRect` method. The resulting image is shown in Figure 5.4. The bars 5 to 9 where omitted from the code. Source code created by the author.

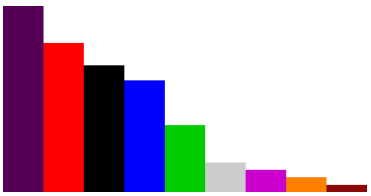


Figure 5.4: The resulting graphic created with Pixi.js from the code in Listing 5.5.

5.5 Three.JS (3D)

As was shown in Section 4.3.3, creating WebGL content can be somewhat cumbersome. Three.JS, originally developed by Ricardo Cabello Miguel, is a library providing a large set of functionality in an easy and intuitive way. The following list shows the major features and benefits of Three.JS [Parisi, 2012]:

- *Abstracted WebGL API*: Three.JS abstracts the details of WebGL and provides a scene-based API with meshes, objects , materials, shaders, cameras and lights.
- *Extensible*: Due to its setup and language, it is fairly easy to extend the functionality to meet one's own requirements.
- *Fall-back to Alternative Renderers*: Three.JS provides a fall-back to Canvas and SVG renderers, in case WebGL is not supported by the web browser.
- *Fast*: Uses 3d best practises to maintain a high performance.
- *Feature-Rich*: Implements high-level objects for gaming, animations, and special effects.
- *File Format Support*: Provides methods to import and export multiple common 3d files such as JSON, 3D Max, Blender, and OBJ.
- *Interaction*: Provides relatively easy ways of adding interaction to WebGL applications.
- *Math*: Provides a solid math library for all 3d related operations, such as matrix- and vector-operations.
- *Object-oriented*: Provides JavaScript objects instead of only method calls.

These features make Three.JS one of the most powerful 3d libraries available for JavaScript. The example in Listing 5.6 shows how to create a simple scene consisting of nine cubes, resulting in a simple bar chart. To achieve the 2d look, an orthographic camera was used. The resulting image is shown in Figure 5.5.

```
1     window.onload = function() {
2         var renderer = new THREE.WebGLRenderer();
3         renderer.setSize( 800, 600 );
4         document.body.appendChild( renderer.domElement );
5
6         var scene = new THREE.Scene();
7         var camera = new THREE.OrthographicCamera(0, 20, 15,0, 0.1, 100);
8         camera.position.set( 0, 0, 10 );
9         camera.lookAt( scene.position );
10
11        var bar1 = new THREE.CubeGeometry( 2, 25, 1 );
12        var material = new THREE.MeshBasicMaterial({ color: 0x550055 });
13        var mesh = new THREE.Mesh(bar1, material);
14        mesh.position.x = 1;
15        scene.add(mesh);
16
17        var bar2 = new THREE.CubeGeometry(2, 20, 1);
18        var material2 = new THREE.MeshBasicMaterial({ color: 0xff0000 });
19        var mesh2 = new THREE.Mesh(bar2, material2);
20        mesh2.position.x = 3;
21        scene.add(mesh2);
22
23        var bar3 = new THREE.CubeGeometry(2, 17, 1);
24        var material3 = new THREE.MeshBasicMaterial({ color: 0x000000 });
25        var mesh3 = new THREE.Mesh(bar3, material3);
26        mesh3.position.x = 5;
27        scene.add(mesh3);
28
29        var bar4 = new THREE.CubeGeometry(2, 15, 1);
30        var material4 = new THREE.MeshBasicMaterial({ color: 0x0000ff });
31        var mesh4 = new THREE.Mesh(bar4, material4);
32        mesh4.position.x = 7;
33        scene.add(mesh4);
34    .
35    .
36    .
37        renderer.render( scene, camera );
38    };
```

Listing 5.6: Creating a simple scene in Three.JS consisting of a nine cubes. First, the WebGL renderer is initialised and added to the DOM. Then, the scene and the camera are created and configured. To create the 2d look, an orthographic camera was used. Finally, each cube is initialised, and together with a material, a mesh is created and added to the scene. The creation of the last five cubes was omitted in this listing. Source code created by the author.

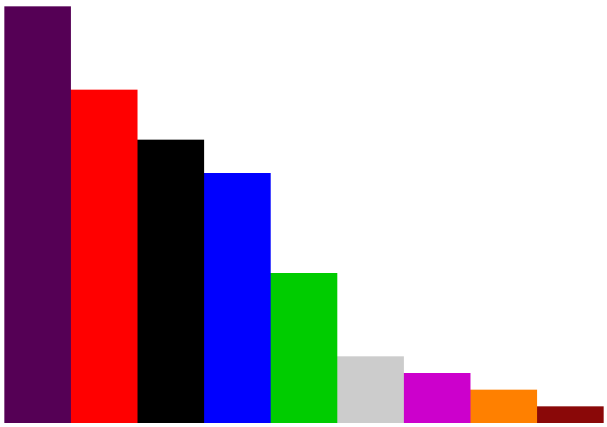


Figure 5.5: The image created with Three.JS from the code in Listing 5.6.

Chapter 6

JavaScript InfoVis Toolkits

Information visualisation can be a tough task to master using web technologies only. Creating meaningful interactive graphics using the techniques described in Chapters 4 and 5 is still very complex. This is particularly true when trying to create a complex visualisation such as Parallel Coordinates (Figure 2.6) or the Flare Dependency Graphs (Figure 2.5). Several InfoVis toolkits have been developed using JavaScript and possibly one of the graphics libraries described in Chapter 5.

This chapter will show four toolkits that approach this problem in two significantly different ways. All provide a very effective API for creating interactive visualisations using web technologies only. The libraries that will be analysed are:

- *JIT*: The JavaScript InfoVis Toolkit (JIT), based on the Canvas element [Belmonte, 2013a].
- *D3*: D3 uses SVG, HTML, and CSS [Bostock, Ogievetsky, and Heer, 2011].
- *Aperture*: Aperture uses SVG, HTML, and CSS [Jonker et al., 2013].
- *Highcharts*: Highcharts uses SVG, HTML, and CSS [Kuan, 2012].

JIT is a very specialised toolkit for creating visualisations, providing much of the logic needed for layouts and interactions. D3 is a more general toolkit, providing ways of interacting with data bound to the DOM. Aperture is a framework for creating multiple visualisations, that can be stacked upon each other, to create rich visualisations. Highcharts provides a very simple way of creating predefined visualisations.

6.1 The JavaScript InfoVis Toolkit (JIT)

The JavaScript InfoVis Toolkit (JIT) is a framework that provides the functionality to create information visualisations [Belmonte, 2013a]. It provides the necessary functionality to draw elements to a canvas and interact with each created element. In addition to the already implemented visualisations, it is possible to create one's own visualisations by extending the core libraries. This, however, requires in-depth knowledge of the JIT libraries and the visualisation algorithm. By sticking to the predefined layouts, it is very easy to create web-based visualisations. The visualisations implemented in JIT are:

- *Tabular Data*: Area, Stacked Bar, and Stacked Pie Charts (with slight modifications standard bar and pie charts can be created).
- *Hierarchical Data*: Tree Map, Radial Graph, Space Tree, Hyper Tree, Icicle, and Sunburst.
- *Graph Data*: Force-Directed Graph.

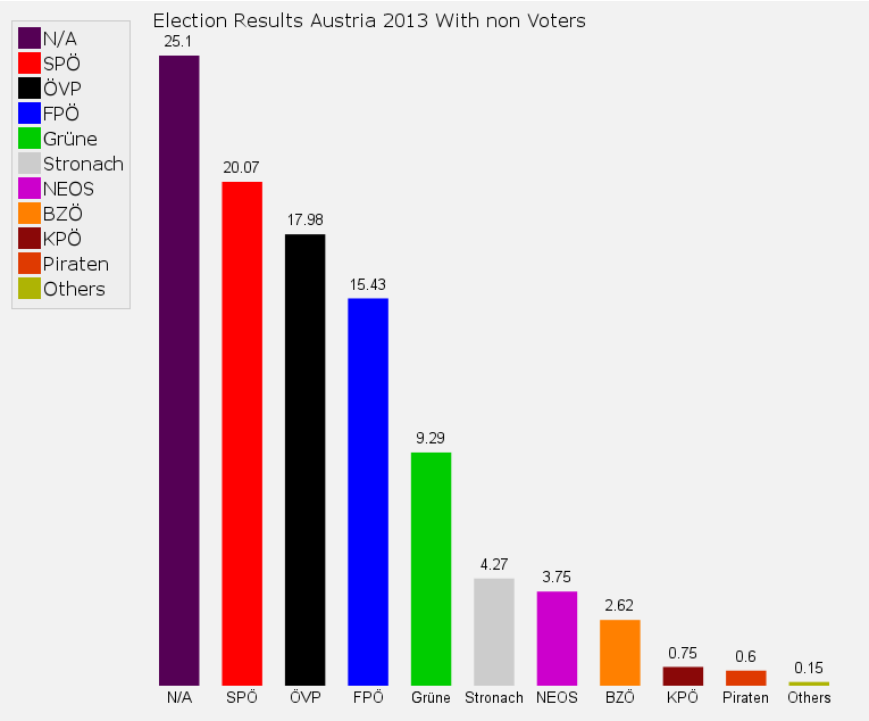


Figure 6.1: The JIT visualisation from the code in Listing 6.1. Image created with JIT by the author.

The example in Listing 6.1 shows how to create a simple bar chart. Since the JIT has only implemented multi-value bar charts, slight modifications to the library had to be made to create single-valued columns. These alterations only concern 4 lines of code inside the `loadJSON` method of the bar chart visualisation on line 33 of Listing 6.1. The result can be seen in Figure 6.1. Listing 6.2 shows how to use the integrated method `getLegend` to create a legend. As can be seen, JIT makes it easy to create out-of-the-box visualisations.

Adding custom visualisations can be difficult however, since the functionality has to be added to the core library to function properly. However, JIT provides the necessary functionality needed to create any kind of visualisation, such as converting JSON to nodes and edges, traversing the nodes of the graph, drawing and positioning the elements and providing callback methods for event handling [Belmonte, 2013b].

6.2 D3

D3 stands for Data-Driven Documents and is a toolkit for creating rich web pages and visualisations focusing on data using SVG, HTML, and CSS [Bostock, Ogievetsky, and Heer, 2011]. D3 focuses on general visualisation, providing a method to create one's own visualisations. It implements some ready to use visualisations, but also provides methods to create layouts and interactions [Bostock, 2013]. The easiest way of demonstrating D3's behaviour is through a simple example shown in Listing 6.3. The example shows how data is bound to paragraph elements. If there are no paragraphs already bound to the data, new paragraphs will be added to the DOM. The example shows the `.enter()` method, which is always called when new data is added. There are three ways of interacting with the data:

- *Update:* This is the default and updates existing data binds: `.selectAll("p").data(...).text(...)`.
- *Enter:* This is called for every new data point, as shown in Listing 6.3.

```
1   var barChart = new $jit.BarChart({
2       mono:1, // Added for single value bars.
3       injectInto: 'infovis',
4       animate: true,
5       orientation: 'vertical',
6       barsOffset: 20,
7       Margin: {
8           top:5,
9           left: 5,
10          right: 5,
11          bottom:5
12      },
13      labelOffset: 5,
14      type: 'stacked',
15      showAggregates:true,
16      showLabels:true,
17      Label: {
18          type: "Native",
19          size: 13,
20          family: 'Arial',
21          color: 'Black'
22      },
23      Events: {
24          enable: true,
25          onClick: function(node){
26              if (node) {
27                      // do something with the clicked node
28              }
29          }
30      },
31  });
32
33  barChart.loadJSON(json);
```

Listing 6.1: Creating a bar chart using JIT. First, the bar chart is initialised, and some parameters are passed to the constructor. The most important parameter is `injectInto`, which defines the DOM object the visualisation will be appended to. `Events:` defines events to be handled by the visualisation. In this example, the `onClick` event was implemented to react to a click on a node (in this case a bar). The remaining parameters, which are defined in JSON style, define the look of the bar chart. Source code created by the author.

```
var list = $jit.id('id-list');
legend = barChart.getLegend(),
    listItems = [];
for(var name in legend) {
  listItems.push('<div style=\'background-color:'
      + legend[name] +'; width:2em; float:left;\'> </div>' +
          name);
}
list.innerHTML = '<li>' + listItems.join('</li><li>') + '</li>';
```

Listing 6.2: Creating the legend for the bar chart described in Listing 6.1. First, the data for the legend is fetched from the library, then injected into a div, which is appended to a list. Source code created by the author.

- *Exit:* This is called every time a data point is removed.

This approach to data manipulation has the great advantage of being open to new technologies, such as new HTML tags or completely new XML-based languages. Basically D3 can create and manipulate any XML-based document, making it extremely flexible.

```
<body>
      <script type="text/javascript">
          d3.select("body").selectAll("p")
              .data([4, 8, 15, 16, 23, 42])
              .enter().append("p")
              .text(function(d) { return "I'm number " + d + "!"; });
      </script>
</body>
```

Listing 6.3: Selecting the body and all paragraph elements `p`. The data is then bound to the selection (the selected `p`'s). If there are too few `p`s in the selection new paragraphs will be appended with `.enter()`.`append("p")`. The `.enter()` function is always called when new data is added. Finally, the text of the paragraph is set in the `.text` (..) function. Source code extracted from Bostock [2013].

To demonstrate the flexibility of D3, the example in Listing 6.4 shows how to create an SVG bar chart by binding data to SVG rectangles. The text and the legend are created in a similar fashion. The data in this example is JSON encoded with each data point in the format {'`label`': '`Others`', '`value`': 0.15, '`color`':'`#AEB404`']. No further manipulation of the data is required, since D3 can handle multiple input data types. The resulting image is shown in Figure 6.2.

6.3 Aperture

Aperture is a JavaScript framework for creating visualisations using SVG or VML (IE7, IE8) [Jonker et al., 2013] . Aperture is also capable of creating geographic visualisations, by providing ways of interacting with geographic information systems (GIS). Aperture also supports layering, thus enabling combinations of visualisations, such as pie charts on maps, or line charts as nodes of a hierarchy. The source code in Listing 6.5 shows how to create a simple bar chart. First, a generic graph is created and the axis labels and title are defined. Then, a new layer with the actual bar chart is added. The resulting image is shown in Figure 6.3.

```
var svg = d3.select("body")
      .append("svg")
      .attr("viewBox",'0,0,100,100')
      .attr("height","100%")
      .attr("width","100%")

svg.selectAll("rect")
   .data(dataset)
   .enter()
   .append("rect")
   .attr("x", function(d, i) {
      return i * (w / dataset.length);
   })
   .attr("y", function(d) {
      return h-d.value*3-2;
   })
   .attr("width", w / dataset.length - barPadding)
   .attr("height", function(d) {
      return d.value*3 //*3 scaling factor
   })
   .attr("fill", function(d) {
    return d.color;
   });
```

Listing 6.4: Creating a bar chart with D3. First, a new SVG element is added to the body element of the page. Then the data is bound to SVG rect elements and appended to the SVG. The `.attr()` calls define SVG styles for the position, size, and colour of the rectangles. All sizes are given in screen space. The resulting image is shown in Figure 6.2. Source code created by the author.

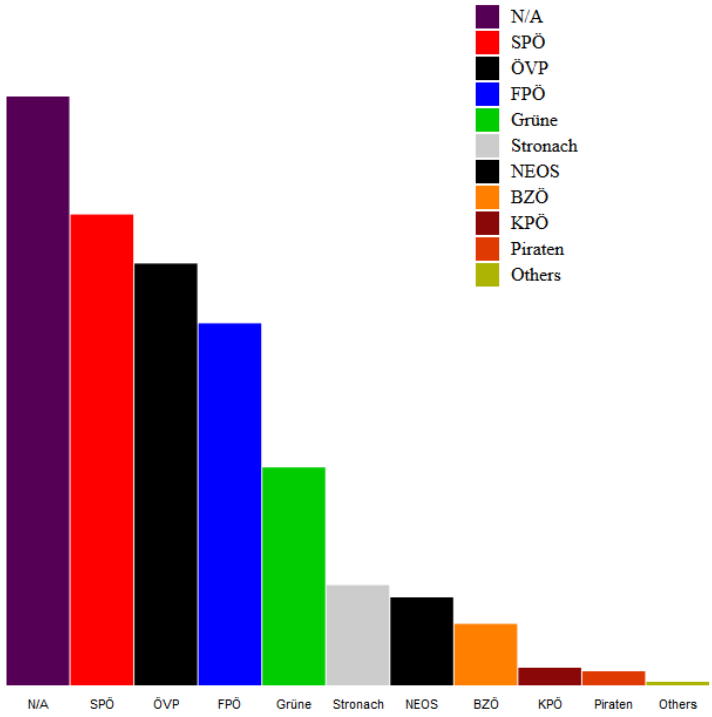


Figure 6.2: The D3 visualisation from the code in Listing 6.4. Image created with D3 by the author.

```
1             var createBarChart = function(width, height){
2                 rangeX = new aperture.Ordinal('label');
3                 rangeY = new aperture.Scalar('value');
4
5                 var data = dataset.results;
6                 for (var j=0; j < data.length; j++) {
7                     rangeX.expand(data[j].label);
8                     rangeY.expand(data[j].value);
9                 }
10                rangeY.expand(0);
11
12                chart = new aperture.chart.Chart('#container');
13                chart.all(dataset);
14                chart.map('width').asValue(width);
15                chart.map('height').asValue(height);
16                chart.map('x').using(rangeX.banded().mapKey([0,1]));
17                chart.map('y').using(rangeY.banded(30,0.1).mapKey([1,0]))
                      ;
18
19                chart.map('title-spec').asValue({text: 'Election Results
                     Austria 2013', 'font-size':15});
20                chart.map('title-margin').asValue(30);
21                chart.map('stroke').asValue('#000000');
22                chart.xAxis().mapAll({
23                    'title' : 'Party',
24                    'margin' : 40,
25                    'rule-width': 1
26                });
27                chart.yAxis().mapAll({
28                    'title' : 'Result',
29                    'margin' : 40,
30                    'tick-length' : 6,
31                    'label-offset-x' : 2
32                });
33
34                barSeries = chart.addLayer( aperture.chart.BarSeriesLayer
                      );
35                barSeries.all(dataset.series);
36                barSeries.map('x').from('results[].label');
37                barSeries.map('y').from('results[].value');
38                barSeries.map('spacer').asValue('10');
39                barSeries.map('point-count').from('results.length');
40
41                chart.all().redraw();
42            }
```

Listing 6.5: Creating a bar chart with Aperture. First, the range of the chart is defined, by expanding `rangeX` and `rangeY` to the values of `data.label` and `data.value` in the data set. Then, a generic chart is created using `rangeX` and `rangeY` to define the axis marks and labels. Finally, a new layer is added, containing the actual bar chart. The resulting image is shown in Figure 6.3. Source code adapted from [Oculus, 2014].

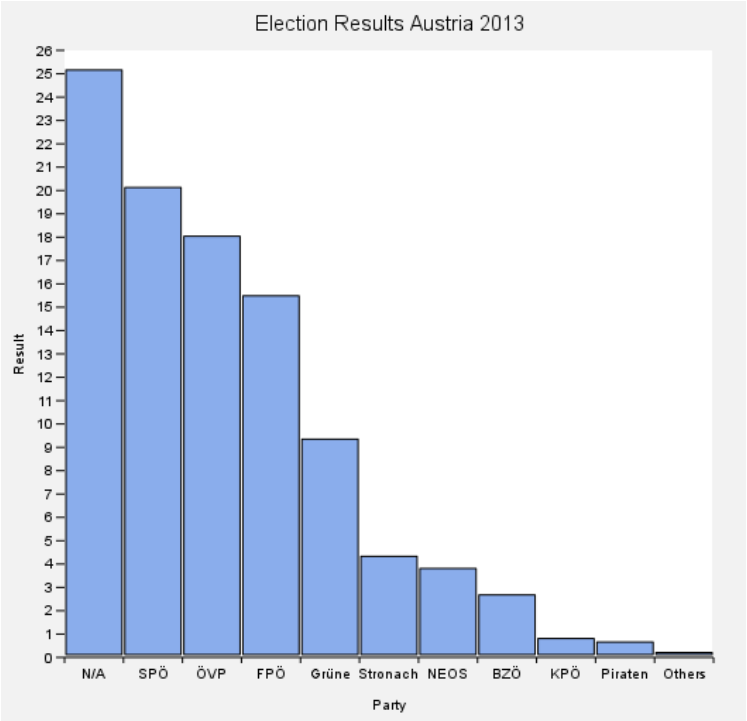


Figure 6.3: The Aperture visualisation from the code in Listing 6.5. Image created with Aperture by the author

6.4 Highcharts

Highcharts is a JavaScript visualisation framework [Highsoft, 2014]. It uses HTML5 and SVG to create interactive visualisations. Currently supported visualisations are: line chart, area chart, column chart, bar chart, pie chart, scatter plot, and many more. Some of the main features of Highcharts are:

- *Browser Compatibility*: Highcharts is compatible with all modern browsers, and uses SVG for rendering. For older IE versions, where SVG is not supported, VML is used.
- *Simple Configuration Syntax*: Configuration is done using simple JSON syntax. This is shown in the example in Listing 6.6
- *Multiple Axes*: Highcharts supports multiple data sets for each axis, making it possible to compare data.
- *Export and Print*: Using the export module, visualisations can be printed or exported as PNG, JPG, PDF, or SVG.

The source code in Listing 6.6 shows a simple example of creating a bar chart. Highcharts is available under the Creative Commons Attribution-NonCommercial 3.0 License for personal and non-profit use. For commercial use, license fees apply.

```
1             $(function () {
2                 $('#container').highcharts({
3                     chart: {
4                         type: 'column'
5                     },
6                     title: {
7                         text: 'Election Results Austria 2013'
8                     },
9                     yAxis: {
10                        title: {
11                            text: 'Results'
12                        }
13                    },
14
15                    series: [{
16                        'name': 'N/A',
17                        'data': [25.1],
18                        'color':'#550055',
19                    },
20                    {
21                        'name': 'SPO',
22                        'data': [20.07],
23                        'color': '#ff0000',
24                    },
25                    {
26                        'name': 'OVP',
27                        'data': [17.98],
28                        'color': '#000000'
29            .
30            .
31            .
32                    }]
33                });
34            });
```

Listing 6.6: A bar chart created using the simple configuration provided by Highcharts. First, the type of the chart is defined. Then the title and the axis labels are set. Finally, the data is passed to the `series` parameter. The last eight records are omitted from the listing. The resulting image is shown in Figure 6.4. Source code adapted from Highsoft [2014].

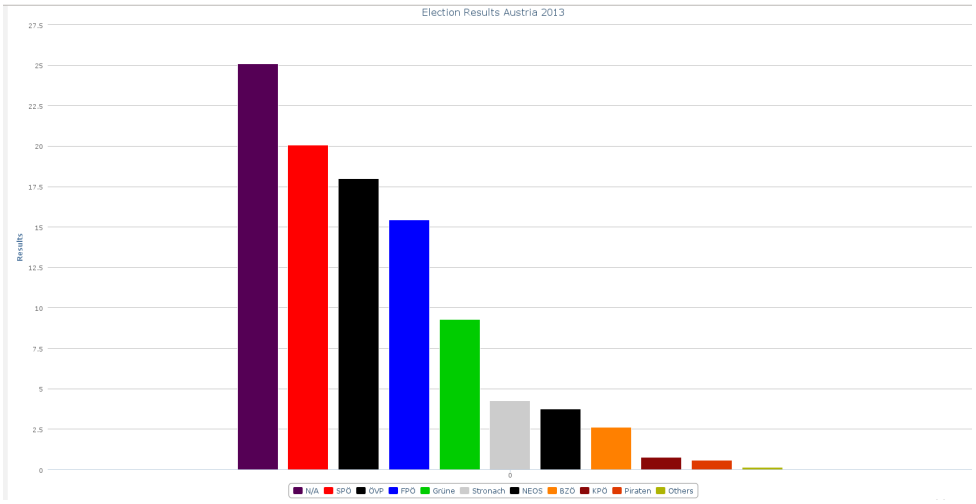


Figure 6.4: The Highcharts visualisation resulting from the code in Listing 6.6. Image created with Highcharts by the author.

Chapter 7

FluidDiagrams

FluidDiagrams is a web-based information visualisation framework using JavaScript and WebGL, and was developed during the creation of this thesis. A web-based approach was chosen, due to the fact that more and more different devices, such as PCs, smart phones, and tablets are in use. Using web technologies, it is possible to use the visualisations created with FluidDiagrams on any platform, without the need for a native application for each device. In addition, it is not predictable what other devices will become available for the customer in future, thus making web technologies such as HTML5 and JavaScript the best choice for FluidDiagrams. Moreover, using JavaScript over any plug-in based language, such as Adobes Action Script, one is not dependant on a third party product.

As was shown in the previous chapters, the creation of information visualisations for the web can be very cumbersome, when not using frameworks or toolkits. FluidDiagrams aims at solving these problems by providing a framework in which the developer can focus on the actual visualisation. Additionally, it takes advantage of WebGL which uses the graphics card for rendering, wherever possible, increasing performance significantly. There are currently many graphics libraries under development using both GPU (graphic processor unit) and CPU rendering, some of which have been discussed in Chapter 5. However these prove not to be optimal when creating interactive graphical visualisations. For this task InfoVis toolkits are available as described in Chapter 6. None of these toolkits however use the opportunities WebGL provides by exporting the rendering process to the graphics card. FluidDiagrams fills this gap and provides a framework with which to create interactive information visualisations, harnessing the newest of web browser technologies such as JavaScript supersets, HTML5 elements, and WebGL.

FluidDiagrams uses the Three.JS graphics library described in more detail in Section 5.5, which provides the WebGL rendering engine, together with a fallback to alternative rendering engines, if there is no support of WebGL by the web browser. These fallbacks are a CPU-rendered version on canvas and an SVG version. However the CPU-rendered canvas version has significantly lower frames per second, reducing the performance, or forcing the visualiser to reduce the amount of geometry. Reduction of geometry can be achieved either by using fewer 3d objects in the scene, reducing it to a very minimalistic visualisation, or by reducing the amount of data displayed each time. These limits will become obsolete as soon as all browsers and devices incorporate graphics hardware and support WebGL, which will be the case sooner or later. In the case of the SVG fallback, this is only really feasible for static images, because the performance is lowered so dramatically, that no proper animation or interaction is possible. This bad performance when using SVG is mainly a problem of the Three.JS library and could perhaps be solved using a different more specialised rendering engine. At the time of this thesis, a new rendering engine, FDGL (FluidDiagrams Graphics Library) was being planned to investigate these performance issues.

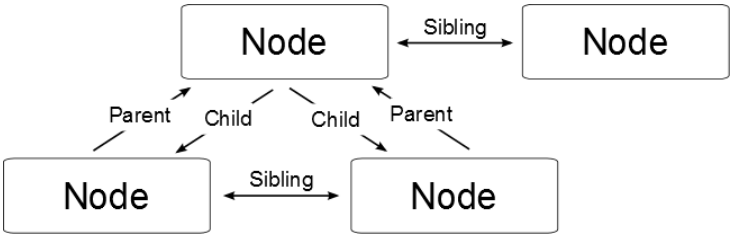


Figure 7.1: The internal data structure of FluidDiagrams. Image created by the author.

7.1 Design

FluidDiagrams addresses three kinds of audience:

- *Developers*: Developers are responsible for the development of the actual framework. They add functionality to the framework, such as new events or new data parsers.
- *Visualisers*: Visualisers create the actual visualisations, either developing their own visualisation or extending existing algorithms, already implemented in the framework.
- *Users*: The users are the consumers of the visualisation. They visit the website containing the finished visualisation and use the interactions and graphics to extract information.

The first two roles will often be mixed: so a visualiser might also create a new data parser, or add events to the framework.

Visualisations in FluidDiagrams are created in four steps, as illustrated in Figure 7.2. Each step is closely related to a module from the architecture and is easily replaced. These four steps are:

- *Initialise*: In this initial step, the visualisation basics are set up. The parser, the layout algorithm, and the event handler are defined, and the basic configuration is set, such as width and height of the visualisation. A list of parameters can be found in Appendix A.
- *Parse*: In this second step, the parser takes the raw data and creates the internal node structure. Each data record then correlates with a node in the scene graph and the internal data structure. A relation between data is represented by a link, which can define parent, child or sibling relations. Figure 7.1 shows the resulting internal data structure.
- *Layout*: This step takes the internal data structure and creates a visual representation according to the defined layout algorithm.
- *Interact*: Finally, the visualisation is finished, and the user can interact with it, if an event handler was defined.

7.2 Architecture

The main focus during the design phase was to enable multiple data sources, interchangeable layout algorithms, and event handlers. For this reason a modular architecture was created, encapsulating parser, layout, and event handler into sub-components. This enables data sources and layouts to be swapped out. However, the examples that were created during the development showed that the event handler could not be separated from the layout algorithm. This suggests that it might be a good idea to couple the layout and the event handler, so the layout algorithm provides the event handler class. This could be achieved by a simple function such as `.getEventHandler()`, which could be called by the main FluidDiagrams class.

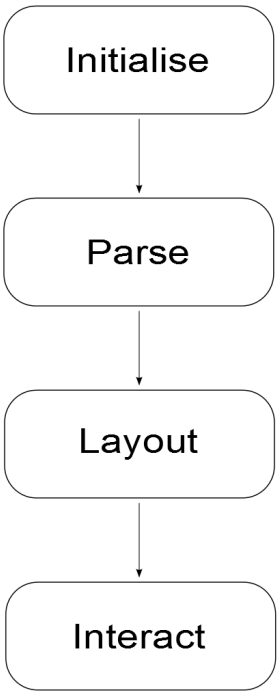


Figure 7.2: The four steps of the FluidDiagrams visualisation pipeline. Image created by the author.

This would remove the necessity of the visualiser to choose the correct implementation, and guarantee the use of the correct event handler. To support this modularity, the JavaScript superset TypeScript, as described in Section 3.5.2, was chosen. The full class diagram can be seen in Figure 7.3, and shows all the major elements required for creation of a FluidDiagrams visualisation.

7.2.1 Initialise (FluidDiagrams)

`FluidDiagrams` is the main class. It is responsible for setting up the scene and choosing the correct rendering engine. First, it executes the `.parse()` method of the set parser. Then the layout algorithm is called. After the layout has been calculated, the geometry attached to the FDNode is added to the scene object from Three.JS. In addition to the scene, FluidDiagrams keeps a reference list of meshIDs to identify the FDNode from the actual mesh. This is needed for the click event, to determine the correct node after clicking geometry within the scene. The main class is also responsible for calling the correct event handler method when an event is launched. This is achieved by attaching the event handler to the DOM element of the Three.JS renderer: `this.domElement.addEventListener('mouseup', function(e){that.onMouseUp(e); }, false);`. Additionally a method, described in more detail in Section 9.1, is provided which returns a node id or a mesh id at a specified position. The FluidDiagrams class also handles resizing of the browser window for relative sizing of the visualisation. Resizing is described in Section 9.3.

7.2.2 Parse (FDParser)

`FDParser` is responsible for converting any data type to FluidDiagrams internal data structure shown in Figure 7.1. It creates a hierarchical structure of FDNodes. Each data record is stored in an FDNode. The only required member of FDNode is the unique identifier for this particular data point. The result of the parser is a hierarchical node structure, with possible multiple root elements, and a list of all nodes. In the case of a non-hierarchical data set the resulting FDNode tree will be very shallow and consist only of

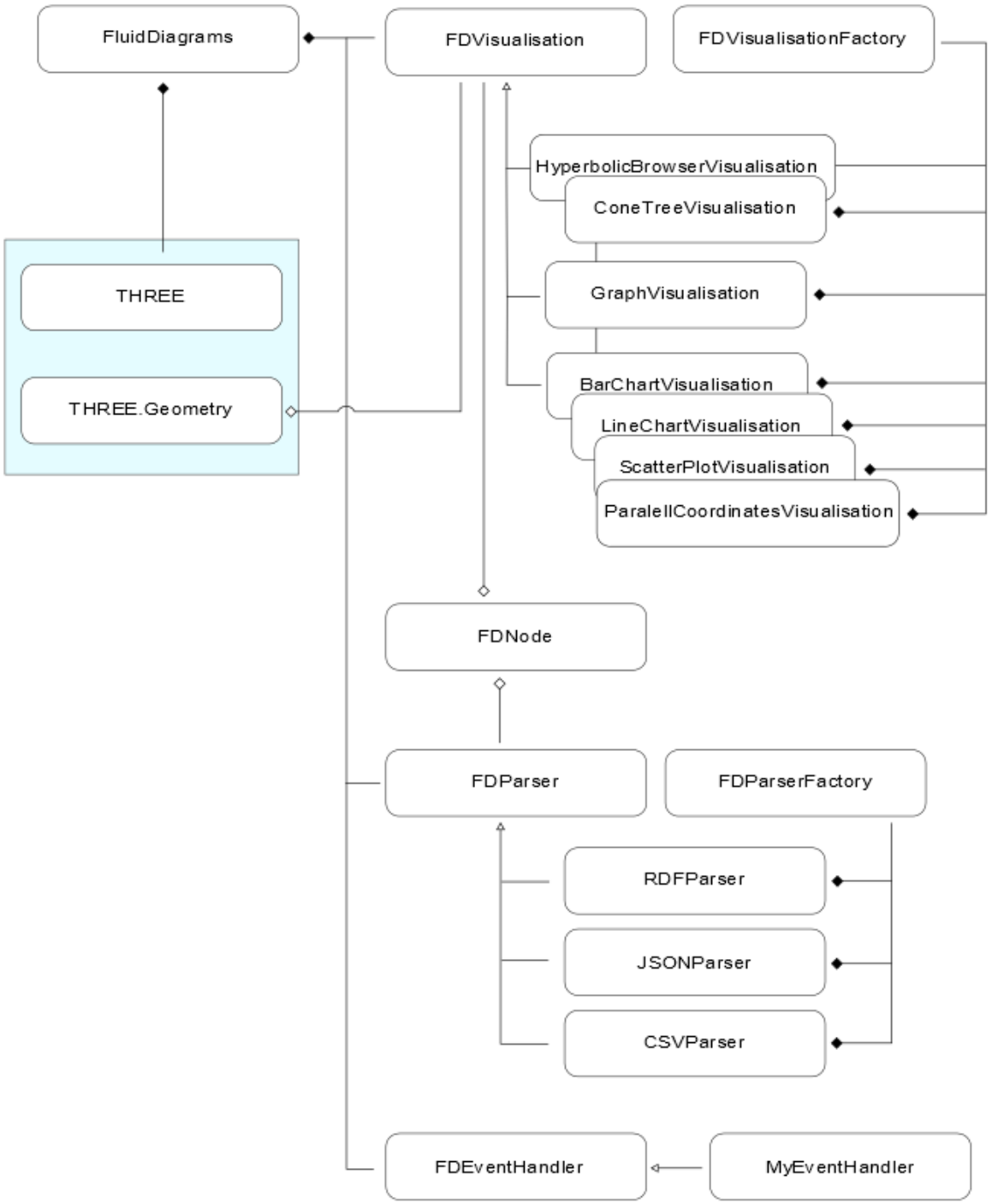


Figure 7.3: The FluidDiagrams class diagram showing the four major elements: FluidDiagrams main class, FDParser, FDLayout, and FDEventHandler. Image created by the author.

root elements. The relationships between two nodes are stored within the FDNode as one of three types: children, siblings, and parents. FluidDiagrams currently only supports graphs that consist only of three types of relations: Parent, Child, and Sibling. For arbitrary graph structures, additional links are required in the internal data structure.

7.2.3 Layout (FDLayout)

`FDLayout` is the base class for any visualisation layout. It implements the visual representation of the data. It maps FDNodes to THREE.geometry. THREE.geometry comes from the Three.JS library and is later passed to the renderer to display on canvas or as SVG, as described in Section 5.5. The layout algorithm is responsible for the positioning and texturing of the nodes in 3d. 2d visualisations are accomplished by setting the z axis to 0, and by setting the camera to `THREE.OrthographicCamera`, which means that there will be no perspective distortion in the image. For 3d visualisations, a `THREE.PerspectiveCamera` is chosen to give a proper sense of depth.

7.2.4 Interact (FDEventHandler)

The event handler implements the methods which are called when a certain event occurs, such as a click or mouseover. These events are caught by the FluidDiagrams main class, but are then delegated to the event handler. Currently, only a few events are implemented by the framework:

- `onMouseDown`: executed when the left mouse button is pressed.
- `onMouseUp`: executed when the left mouse button is released.
- `onMouseMove`: executed when the mouse is moved over the canvas element.
- `mouseWheel`: executed when the scroll wheel is turned.
- `onMouseClick`: executed when the left mouse button is pushed and released within a certain time frame.

Additionally, the event handler also implements an `update` function which is called once in every frame, thus enabling animations that are independent of any event. Additionally to the implemented events, any other browser event can be added to FluidDiagrams in future development.

Chapter 8

Visualisations

This chapter covers the visualisations created in FluidDiagrams during the development phase and in conjunction with the Information Visualisation course [706.057] at the Graz University of Technology in SS 2013. During the said course, four groups were tasked with creating visualisations using FluidDiagrams. In addition to the visualisations created, this provided valuable feedback on the toolkit and also showed the usability of the API.

8.1 Cone Tree

A cone tree is a method of visualising large hierarchies, using 3d to maximise the available screen space [Robertson et al., 1991]. This enables the hierarchy to be presented as a whole. The cone tree uses animated interaction to enable the user to focus on a certain subset of the hierarchy, while maintaining the context to the entire hierarchy. This interaction is implemented by rotating the element in focus to the foreground, and giving the user the ability to zoom in and out. Additionally, it is possible to traverse up and down the tree to view different depths of the hierarchy. Carrière and Kazman [1995] proposed an algorithm for positioning each node of the tree, that reduces overlap in 3d space. The idea is to start from the bottom of the tree and calculate the space each node requires as the hierarchy is traversed up to the root. Figure 8.1 shows how each node of each layer of the hierarchy is positioned. At the centre of each circle lies the parent node, and the children are positioned on the circle surrounding it. For each level, the circumference is estimated with:

$$C_{n-1} \cong 2 \sum_{i} r_{i,n}$$

where the radius $r_{i,n}$ of the child at level n is calculated as:

$$r_n = \frac{C_n}{2\pi}$$

The arc length of the parent's circumference that each cone requires is estimated by:

$$s_i \cong r_{i-1,n} + r_{i,n}$$

Then each sub tree is positioned around its parent with the angles defined by:

$$\Theta_i = \frac{s_i}{r_n}$$

This method provides each node with enough space in its parent's cone as necessary. An example of the implemented Cone Tree visualisation can be seen in Figure 8.2. The implemented visualisation has two configurable parameters:

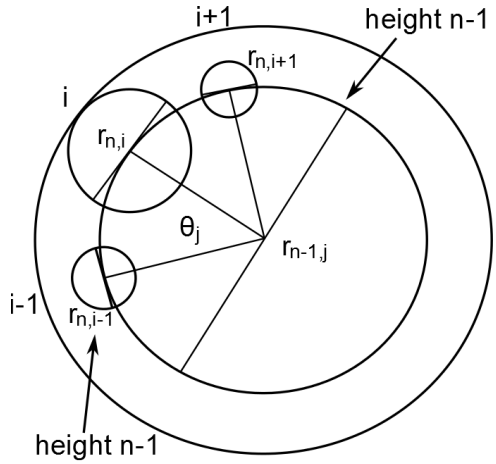


Figure 8.1: The basic principle of the Cone Tree layout, as suggested by Carrière and Kazman [1995]. Image redrawn by the author from the original in Carrière and Kazman [1995].

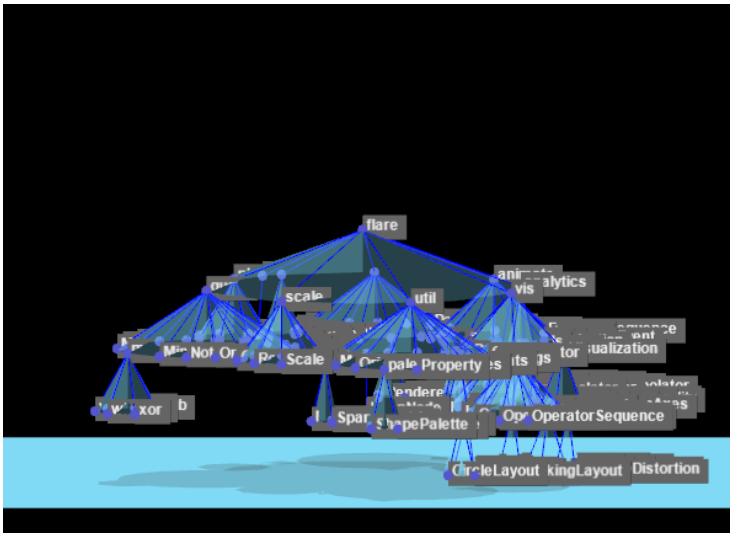


Figure 8.2: A Cone Tree as implemented by FluidDiagrams. Image created by the author using FluidDiagrams.

- *Activate/Deactivate Shadows*: Each cone of the tree casts a shadow, which enhances the viewers ability to keep the whole context in mind.

- *Activate/Deactivate Animations*: The transition of the nodes can either be animated, or happen instantaneously. This feature was added for demonstration purposes only. Animations should generally be activated when using the cone tree.

Interaction with a cone tree can happen in two ways:

- *Clicking a Node*: When a node is clicked, the tree and all sub-cones are rotated to move the selected node to the front and centre of the visualisation.

- *Zooming*: Using the mouse wheel it is possible to zoom in and out of the visualisation, for a more detailed or general view of the hierarchy.

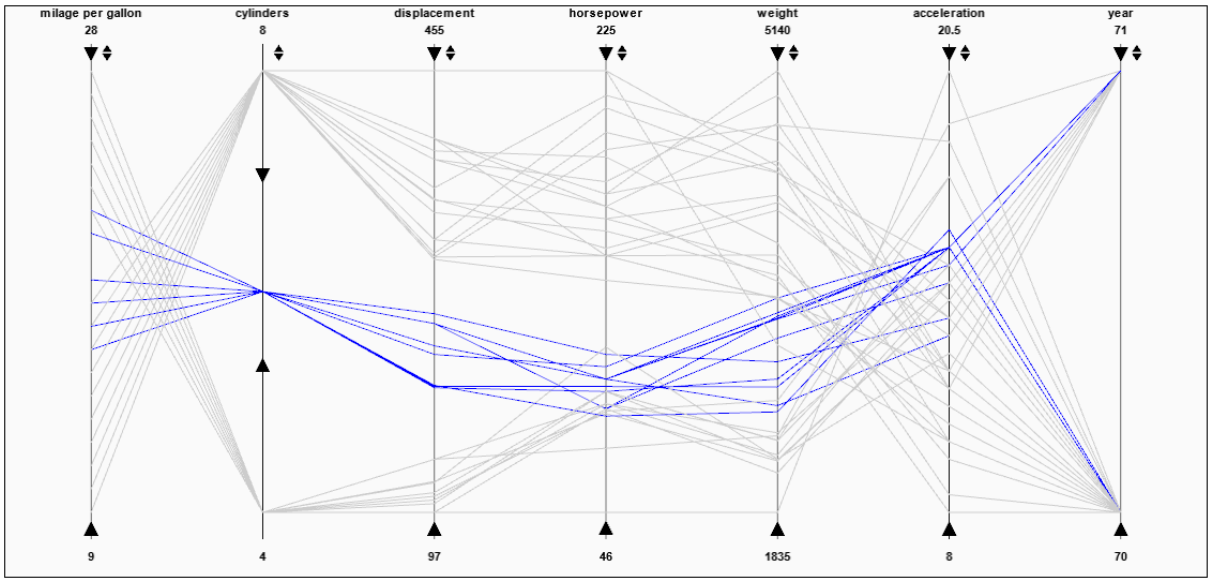


Figure 8.3: Parallel coordinates as implemented by FluidDiagrams. Shown here are cars manufactured between 1971 and 1972, filtered by vehicles with six cylinders only. Each car is represented by a polyline, showing the car's corresponding attribute value in each dimension. For this visualisation, a modified version of the classic `cars` data set was used [Ramos and Donoho, 1983]. The image was created by the author using fluidDiagrams.

8.2 Parallel Coordinates

A parallel coordinates visualisation aims at visualising multi-dimensional data in a 2d space [Inselberg, 1985]. For this, each dimension or axis is assigned to a vertical line in the visualisation. A record, say an individual car in a data set of cars and their attributes, is then represented by a polyline connecting its position on each axis. Before the lines can be drawn, the values in each dimension have to be normalised. This is required because each dimension can have a different value range. In the implementation in FluidDiagrams, this is achieved by normalising the values to a given range (the height of the visualisation).

$$Val_{norm} = \frac{Val - min}{max - min} * normVal$$

with max, min being the maximum and minimum values for the current dimension respectively, and $normVal$ the value range defined by the height of the visualisation.

As well as the layout, the functionality of filtering and inverting each dimension was implemented. This feature allows the visual detection of trends within the original data. The reordering of dimensions is important in order to carry out an analysis. This, however, is not implemented in the current version of FluidDiagrams parallel coordinates. The example in Figure 8.3 shows a data set of cars manufactured between 1971 and 1972, filtered by vehicles with six cylinders only.

Parallel coordinates provides a very simple interaction scheme. Using the sliders at the top and bottom of each column, it is possible to filter the displayed data sets. Inactive (filtered out) records are displayed using light grey lines, while active records are displayed using dark blue. A dimension can be inverted using the toggle button next to the top filter slider.

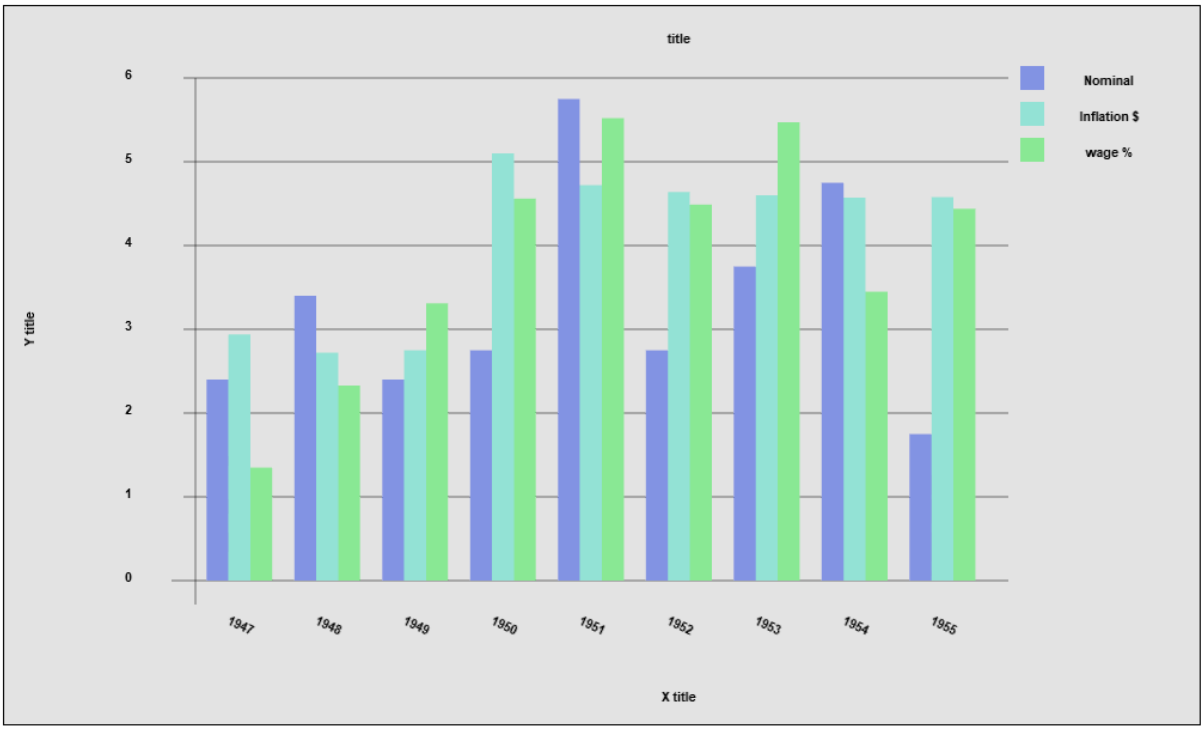


Figure 8.4: Bar chart as implemented by FluidDiagrams. Here, categories are the years from 1947 to 1955, and each category consists of three values. Image created by the author using FluidDiagrams.

8.3 Bar Chart

The FluidDiagrams bar chart visualisation was created during the Information Visualisation course [706.057] at Graz University of Technology in SS 2013 by Group 2. A bar chart is used to visualise discrete values according to categories [Kelley and Donnelly, 2009]. Each category can be constructed from a single value or multiple values grouped together. Typically, categories are years, countries, names, or other discrete units. The discrete value is represented by a rectangle, with the height representing the value. The width of the rectangle is arbitrary and does not correspond to any value in the original data. Bar charts are very useful, when trying to compare different categories. The bar chart in Figure 8.4 shows nine categories (years from 1947 to 1955), each consisting of three values (bars).

The bar chart visualisation provides five basic configurable parameters, which can be set either prior to visualisation or during interaction:

- *Set Colour*: It is possible to either set the colour for a single bar, or change the whole colour scheme of the visualisation.
- *Show/Hide Legend*: Toggle the display of the legend.
- *Set Number Delimiters*: Set the thousands separator and decimal delimiter to a chosen character for multi-language support.
- *Set Title and Axis Labels*: Set the labels of the x and y axis, as well as the title of the chart.

Hovering the mouse over a bar will highlight it and show the actual value of the selected bar.

8.4 Line Chart

The line chart visualisation was created during the Information Visualisation course [706.057] at the Graz University of Technology in SS 2013 by Group 3. A line chart is used for any data where the y axis is a function of the x axis, and the data is a sample of a continuous process [Kelley and Donnelly, 2009]. The data points are plotted on the graph, and then the points in between are linearly interpolated, approximating a continuous function. Usually, a line chart is used for time-dependent data, with the time on the x axis, and the corresponding value on the y axis. Multiple variables can be plotted, by overlaying multiple lines. This is often used to compare processes, or developments. In the example in Figure 8.5, the different causes of deaths in traffic are visualised. Line charts offer numerous configurations:

- *Set Colour*: Sets the colour of the currently selected line.
- *Change Shape*: Changes the shape of nodes to one of a predefined set of geometries. These geometries are: square, circle, triangle, diamond, and star.
- *Set Node Size*: Sets the size of the node shape.
- *Set Line Width*: Sets the width of the lines.
- *Manipulate Grid*: Toggles the display of x and y grid lines. Increase or decrease the number of grid lines.
- *Delimiters*: Sets the thousands separator and decimal delimiters.
- *Show/Hide Legend*: Toggles the display of the legend.
- *Date Values*: Defines x values as date values.
- *Set Titles and Labels*: Sets the title and the axis labels of the chart.

This visualisation also provides basic interaction methods. Hovering the mouse over a line displays the name of the record. Hovering over a node of a record, displays its name and value. Clicking on a line highlights the selected line by colouring non-selected lines grey.

8.5 Scatter Plot

the scatter plot visualisation was created during the Information Visualisation course [706.057] at the Graz University of Technology in SS 2013 by Group 1. A scatter plot is a 2d visualisation of multi-dimensional data [Kelley and Donnelly, 2009]. It can present multiple dimensions at once, by mapping 2 dimensions to the axis, and additional dimensions to colours, sizes, and shapes of the data points. The main aim of this visualisation is to detect patterns and clusters in the data. Also, correlations between dimensions can be observed. The example in Figure 8.6 shows the correlation between the weight and the miles per gallon of a data set of cars.

The implemented version of scatter plot supports selecting dimensions for the x and y axes, as well as to the size of the icon. The icons can be selected from a list of predefined objects, and coloured. No further interaction or customisation is currently implemented.

8.6 Hyperbolic Browser

The hyperbolic browser is a technique for representing large hierarchies using hyperbolic geometry [Lamping, Rao, and Pirolli, 1995]. In this technique, a radial tree is laid out in hyperbolic space, and

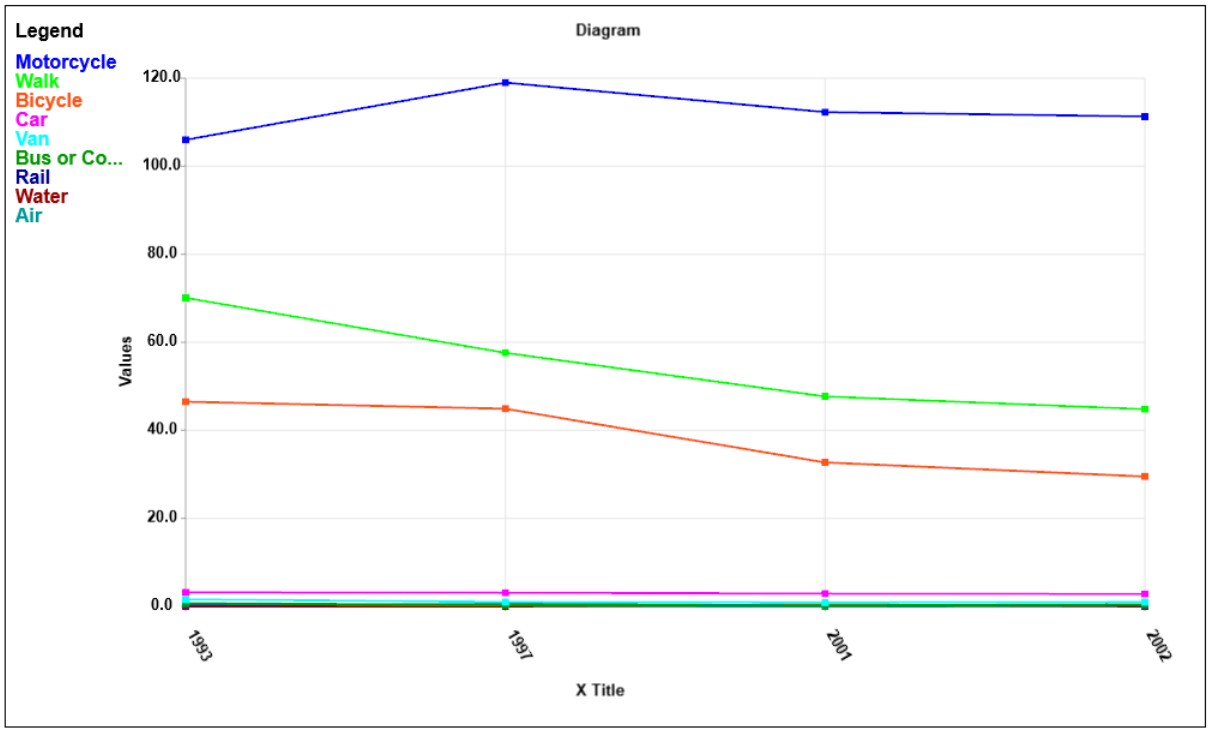


Figure 8.5: Line chart as implemented by FluidDiagrams. The data compares different kinds of deaths in traffic. Image created by the author using FluidDiagrams.

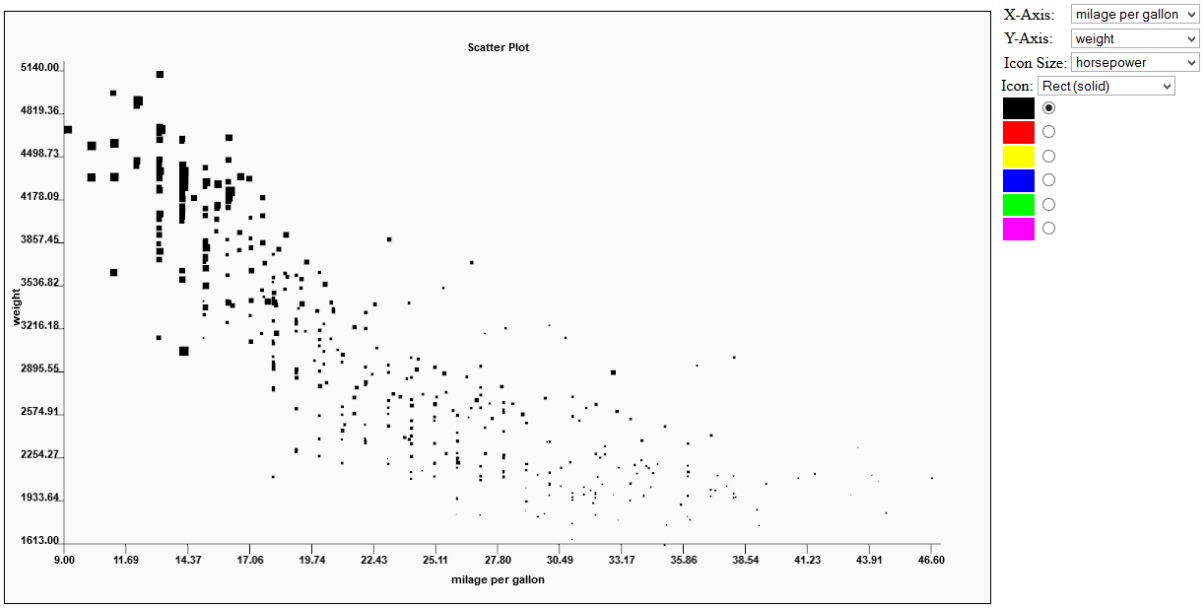


Figure 8.6: A scatter plot as implemented by FluidDiagrams, showing the correlation between weight and miles per gallon. Icon size is mapped to horsepower. For this visualisation, a modified version of the classic `cars` data set was used [Ramos and Donoho, 1983]. Image created by the author using FluidDiagrams.

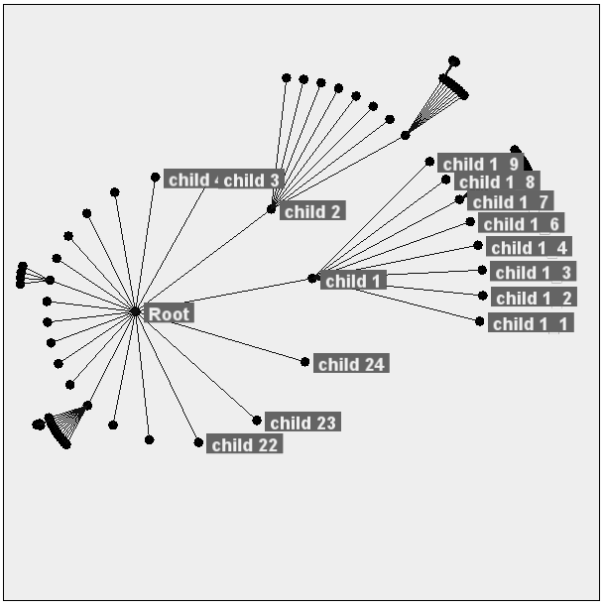


Figure 8.7: The hyperbolic browser as implemented in FluidDiagrams. The node `child_1` was clicked to bring it into focus at the centre of the window. As can be seen, the context is preserved while the focus is in the centre of the visualisation. Image created by the author using FluidDiagrams.

then projected to the unit disc, after which it is scaled to the screen space available. This method enables large hierarchies to be viewed as a whole, keeping the context of the data, while focusing on a subset of the tree at the centre. Figure 8.7 shows the implementation by FluidDiagrams. The code is based on the Java implementation of a hyperbolic browser for the Hierarchical Visualisation System (HVS) written by Alexander Nussbaumer. At the time of writing, there were still some minor issues with the current FluidDiagrams version of the hyperbolic browser. The only interaction methods are dragging the tree, and clicking on a node to centre it in the view. No customisations are implemented at this time.

Chapter 9

Selected Details of the Implementation

In this chapter some functions or features which proved to be especially interesting will be described in detail, to document the problems that arose and how these problems where tackled.

9.1 Determining the Clicked Element

The `onClick` event requires the element at the mouse position to be determined. This could be an FDNode or a mesh. For this purpose the method `objectFromMouse(x,y)` was implemented, which returns either an FDNode if one was present at the provided position, a mesh id if a mesh that is not associated with any node was present, or null if nothing was clicked. This method can easily be used outside of the click event to detect objects that are at a specific screen space position. To accomplish this, several steps had to be taken. This process differs slightly depending on the camera used. First, the method for perspective camera is shown:

1. Transform the screen space x and y coordinates to the local coordinates within the 3d scene. First the offset of the DOM element is subtracted from the pagex and pagey coordinates. Then the coordinates are translated to the origin of the 3d space, which is at the centre of the DOM element. The full code can be seen in Listing 9.1.

2. Calculate a directional vector from the camera to the clicked coordinates. This step requires three smaller steps. First, a vector is defined with the local x and y coordinates and a positive z component. Then the vector is projected using a THREE.Projector and the camera. From this resulting vector, a directional vector is calculated by subtracting the camera position and normalising it. The code can be seen in Listing 9.2.

3. Cast a ray from the camera in the direction calculated in previous step. Three.JS provides a method with which a ray can be sent into the scene and which returns all intersected geometry objects. The use of this method is demonstrated in the listing below.

```
1 var ray = new THREE.Raycaster(this.camera.position, direction);
2 var intersects = (<any>ray).intersectObjects(this.scene.children,
     true);
```

4. After the intersections have been calculated, the correct element to return from this function has to be found. For this, the first mesh in the array is selected, because all the following elements will be behind in the 3d scene, and thus not clickable. Then, the selected mesh is checked to determine if it is associated with an FDNode. If so then the node is returned, otherwise the mesh id is returned.

```
var offsetLeft = this.domElement.offsetLeft;
var offsetTop = this.domElement.offsetTop;
var eltx = pagex - offsetLeft;
var elty = pagey - offsetTop;
var vpx = ((eltx / this.domElement.width) * 2 - 1);
var vpy = (-(elty / this.domElement.height) * 2 + 1);
```

Listing 9.1: Transforming screen x and y coordinates (pagex, pagey) to local 3d coordinates. The resulting coordinates vpx and vpy are the coordinates in 3d space. The z coordinate is 0, since clicking happens in 2d space. Source code created by the author.

```
#--------FluidDiagrams------
var vector = new THREE.Vector3(vpx, vpy, 0.5);
var projector = new THREE.Projector();
projector.unprojectVector(vector, this.camera);
var direction = vector.sub(this.camera.position).normalize();

#--------Three.JS----------------
this.unprojectVector = function ( vector, camera ) {
  camera.projectionMatrixInverse.getInverse( camera.projectionMatrix );
  _viewProjectionMatrix.multiplyMatrices( camera.matrixWorld, camera.
     projectionMatrixInverse );
  return vector.applyProjection( _viewProjectionMatrix );
};

applyProjection: function ( m ) {
  // input: THREE.Matrix4 projection matrix
  var x = this.x, y = this.y, z = this.z;
  var e = m.elements;
  var d = 1 / ( e[3] * x + e[7] * y + e[11] * z + e[15] ); // perspective
      divide
  this.x = ( e[0] * x + e[4] * y + e[8]  * z + e[12] ) * d;
  this.y = ( e[1] * x + e[5] * y + e[9]  * z + e[13] ) * d;
  this.z = ( e[2] * x + e[6] * y + e[10] * z + e[14] ) * d;
  return this;
},
```

Listing 9.2: Unprojecting a vector with a camera. First the inverse projection matrix of the camera is calculated and multiplied with the camera's world matrix to give the view projection matrix. Finally, the vector is projected using the camera's view projection matrix. Homogeneous coordinates are used.

```
#--------FluidDiagrams------
var projector = new THREE.Projector();
var ray = projector.pickingRay(new THREE.Vector3(vpx, vpy, 0.5), this.
    camera);
var intersects = (<any>ray).intersectObjects(this.scene.children, true);

#--------THREE----------------
this.pickingRay = function ( vector, camera ) {
  // set two vectors with opposing z values
  vector.z = -1.0;
  var end = new THREE.Vector3( vector.x, vector.y, 1.0 );
  this.unprojectVector( vector, camera );
  this.unprojectVector( end, camera );
  // find direction from vector to end
  end.sub( vector ).normalize();
  return new THREE.Raycaster( vector, end );
};
```

Listing 9.3: When using an orthographic camera, a picking ray is created and cast into the scene to detect the intersecting objects.

If an orthographic camera is in use, the process of creating the ray is slightly different:

- After converting the window coordinates to local 3d coordinates, a picking ray is created. The `projector.pickingRay` takes 2d normalised device coordinates (NDC) and the camera to create a ray that can be cast in to the 3d scene. The cast ray returns all intersected objects, from which the correct mesh can be selected the same way as in perspective mode. This is shown in Listing 9.3.

9.2 Selecting a Render Engine

Selecting the correct render engine is crucial for the support of every web browser. FluidDiagrams provides a way of forcing either SVG or canvas renderer, but will first try to use WebGL if not otherwise stated. First, the support of the WebGL context is checked. If WebGL is not supported by the browser, the canvas renderer is initialised. If WebGL is supported the renderer is initialised, however if the WebGL context cannot be created successfully there may be something wrong with the browser or graphics card settings. Therefore a fallback to the canvas renderer is performed. Finally, in the case of WebGL raising an exception, which may occur if the graphics card is too old or does not have a 3d accelerator, again the canvas renderer is executed. The default set-up does not include the SVG renderer due to its lack of performance. The execution path in FluidDiagrams is shown in Listing 9.4.

9.3 Rescaling

Responding to the `onWindowresize` event makes the visualisation resolution-independent, by scaling it relative to available the screen space. This enables the use of the visualisations on small and large screens. There are two ways to update the size: keeping the aspect ratio or ignoring it. The aspect ratio is calculated by the `setDimensions()` method, during the initialisation phase of the framework. If `keepAspectRatio` is true, the height of the visualisation is calculated relative to the width of the DOM object, as shown in Listing 9.5.

```
if (this.forceRenderer != "SVG" && this.forceRenderer != "Canvas") {
            if (!(<any>window).WebGLRenderingContext) {
                //NO WEBGL
                this.renderer = new THREE.CanvasRenderer();
            } else {
                //WEBGL
                try {
                    var context = (<any>document.createElement('canvas'))
                        .getContext('experimental-webgl');
                    if (context) {
                        this.renderer = new THREE.WebGLRenderer({
                            precision: "highp", antialias: true });
                    } else {
                        this.renderer = new THREE.CanvasRenderer();
                    }
                } catch (e) {
                    this.renderer = new THREE.CanvasRenderer();
                    return;
                }
            }
}
```

Listing 9.4: Selection of a rendering engine in FluidDiagrams. First, the support of WebGL is checked. If there is no support for WebGL, the canvas renderer is initialised. If it is supported, the WebGL context is created and the renderer is initialised. If the creation of the context fails, the canvas renderer is used. If an exception is raised during the initialisation of the WebGL renderer, the library again falls back to using the canvas renderer. Source code created by the author.

```
private onWindowResize(e: Event) {
  if (this.scale == true) {
        this.width = this.container.width();
              if (this.keepAspectRatio == true) {
              this.height = this.container.width() * this.aspectRatio;
    } else {
              this.height = this.container.height();
              }
                this.container.css("height", this.height);
                (<any>this.renderer).setSize(this.width, this.height);
  }
}
```

Listing 9.5: Enabling automatic rescaling of the visualisation on window resize. Depending on `keepAspectRatio`, either the height and the width is taken from the containing DOM element, or the height is calculated from the aspect ratio and the width. Then the size of the renderer is updated with the new width and height. Source code created by the author.

Chapter 10

Future Work

This chapter presents some ideas for further improvement to FluidDiagrams in the future.

10.1 Additional Parser and Layouts

The first most prominent development required is the addition of new parsers and new visualisations. Adding the possibility of using RDF data sources would greatly improve the utility of FluidDiagrams in the area of open data, and therefore make the framework usable in more diverse environments. For FluidDiagrams to be deployable in multiple environments, it is necessary to drastically increase the selection of visualisations, for example: pie chart, histogram, box plot, treemap, walker tree, and a selection of graph layouts.

10.2 Widgets

For better support of developers who create interactive visualisations, it would be very beneficial to provide a suite of standard widgets. These widgets should provide consistent interaction elements for the use within visualisations, for example:

- *Colour Selector*: A widget for selecting a colour and customising the visualisation.
- *Data Source Selector*: A widget for selecting the datasource of the visualisation.
- *Field Name Selector*: When multiple dimensions are visualised, the data for each axis should be selectable.
- *Label Editor*: For creating labels, used for setting and changing the title and other labels of the visualisation.
- *Shape Selector*: Where the visualisation uses shapes, these should be interchangeable.

This list is only a short selection of possible widgets that could be implemented.

10.3 Architecture Changes

As described in Section 7.2, it would be a good idea to combine the layout algorithm with the event handler. This would ensure the usage of the correct event handler for each visualisation. Also this would improve the usability of the toolkit, removing the task of selecting the correct event handler from

the developer. However, similar visualisations would have to re-implement the event handler instead of utilising parts of similar event handlers that otherwise could be reused.

To enable FluidDiagrams to visualise any arbitrary graph structure, additional link types are required. These link types should have the following characteristics:

- *Directed / Undirected*: Defining the direction of a link.
- *Link Types*: Arbitrary link types for maximum flexibility. For example, social network graphs might have objects (records) representing people, and links between them with types such as: sibling, friend, partner, wife, associate, colleague, etc.
- *Attributes*: Additional attributes such as distance, cost, or other data should be able to be stored in the link.

10.4 Switching to a new Render Engine

As described in Chapter 7, the current render engine lacks performance when using non-WebGL renderers. This could be overcome by implementing a custom render engine which is tailored to the needs of information visualisation, instead of providing functionality not required in this context such as: particles, bones, morph animations, and fog . By reducing the functionality of the render engine to the necessary features required by information visualisations, the performance could be increased, because the overhead of all the non-used features would be eliminated. At the time of this thesis, FluidDiagrams Graphics Library (FDGL) was being planned to investigate this possibility.

Chapter 11

Concluding Remarks

Chapter 2 of this thesis introduced the research area of information visualisation. The differences between scientific visualisation, geographic visualisation, and information visualisation where explained. Then the main goals of information visualisation , and the necessity of meaningful interaction for navigation and filtering were described. A categorisation of the field of information visualisation depending on the type of input data was outlined, consisting of the five categories:

- Linear Data
- Hierarchies
- Networks and Graphs
- Multi-Dimensional Metadata
- Feature Spaces

Each of these categories was described using a distinct example.

Chapter 3 analysed best practices for developing web-based applications using JavaScript. First, the basic concepts of the JavaScript language were introduced. Ways of interacting and manipulating web-pages using the DOM were then described, showing the benefits and disadvantages of the different methods. Also, new technologies such as JSON and AJAX as ways of improving the ways of creating web-applications, and increasing the flexibility of the resulting web pages, were shown. It was then shown how using industry standard libraries can further increase productivity, maintainability, and readability for JavaScript developers. Using jQuery, which implements the most common tasks used in web-development, it was shown that the amount of code required for a given task could be reduced drastically. It was also shown that JavaScript can become very cumbersome to work with on large scale projects, due to its lack of strict typing, modularity, and code reuse. JavaScript supersets are languages which add functionality and are then compiled to native JavaScript. TypeScript adds the concepts of classes, modules, interfaces, and strict type-checking at compile time, making it especially useful for large-scale software projects. Finally, this chapter discussed ways and techniques of automatic testing of JavaScript applications. The Jasmine framework provides all functionality needed for conducting unit tests. It even provides ways of testing asynchronous JavaScript calls, making it the framework of choice for modern JavaScript applications.

The creation of web-based graphics was discussed in Chapter 4. The five different approaches, canvas, CSS, SVG, WebGL, and Flash were illustrated using simple examples. It was shown that using the techniques as is, does not provide an effective way of creating graphics. Therefore, Chapter 5 introduces three graphics libraries, which simplify the task of creating web-based graphics significantly. The first two libraries discussed, EaselJS and Raphaël, use Canvas and SVG respectively, providing functionality

for 2d graphics. Three.JS was introduced for the creation of rich 3d graphics based on WebGL. This library provides a fallback to canvas rendering if WebGL is not supported by the web browser. Therefore this library was later chosen as the backend for FluidDiagrams.

For later comparison with FluidDiagrams, an overview of existing information visualisation toolkits was given in Chapter 6. Two frameworks, JIT and D3, using canvas and SVG respectively, were presented, and their functionality described with a short example of a bar chart.

Chapters 7 to 9 then cover FluidDiagrams, the framework created during the practical part of this thesis. It was described how the use of WebGL increased the performance of interactive visualisations by shifting the render process to the graphics card. By developing a modular architecture underlying the framework, it was possible to reach maximum flexibility for developing visualisations and interactions. The Information Visualisation course [706.057] at the University of Technology Graz in SS 2013, provided a perfect opportunity to test the framework. This test proved the flexibility and the usability of FluidDiagrams. It also provided additional visualisations which users can choose from. The parsers implemented by the author during the development phase were a JSON parser for hierarchies and a CSV parser for multi-dimensional data. The visualisations developed during both the course and the development phase were cone tree and hyperbolic browser for hierarchies, parallel coordinates and scatter plot for multi-dimensional data, and bar charts and line charts for linear data.

Finally, some ideas for future work were presented in Chapter 10, outlining a possible future progression for FluidDiagrams. The most prominent of these ideas is the introduction of FDGL (FluidDiagrams Graphics Library), a replacement rendering engine for Three.JS, and the combining of a layout algorithm with its event handler, for a cleaner structure of the framework.

Appendix A

User Guide

This appendix provides necessary information for people who want to embed FluidDiagrams interactive visualisations into web pages. The focus of this user guide is to demonstrate how FluidDiagrams is used, and not how to implement new visualisations and parsers. For implementation details, please refer to Appendix B.

A.1 Initialising and Setting Up FluidDiagrams

This section describes how initialise to set up FluidDiagrams. The three first steps of every visualisation are:

- `new FluidDiagrams( debug, forceRenderer)`: `FluidDiagrams` initialises the main class. Both parameters are optional. `debug`(the default is `false`) activates or deactivates debug output to the browser console. `forceRenderer` can take one of three values: `WebGL` (the default), `SVG`, and `Canvas`. This forces the toolkit to use the defined back-end for rendering. Note that the SVG renderer,in the current version of Three.JS has very poor performance and is not recommended with any animation or interaction.

- `setDimensions(width, height, scale, keepAspectRatio)`: sets the size of the visualisation within the container, and will usually be the same size as the containing DOM object. `scale`(default = false) is a flag to activate or deactivate automatic scaling on window resize. If `scale` is set to true, it is recommended to set the size of the container to a percentage value and define height and width with `width = $("#container").width();`. `keepAspectRatio` if set to true, the aspect ratio will be maintained when rescaling. The ratio is computed using the initial width and height. It is recommended to set `keepAspectRatio` to true to avoid distortions on window resize.

- `setContainer(container)`: sets the DOM elements name for embedding the visualisation.

A.2 Defining and Initialising the Parser, Event Handler, and Visualisation

This section describes how to select the parser, the Visualisation, and the event handler, and how to configure them. The required steps are described in the following list:

- `fluidDiagrams.setParserType(type)`: defines the type of parser in use. At the time of this documentation, there were two types available: JSON and CSV.

- `fluidDiagrams.getParser().setUIdentifierFieldName(id)`: defines the unique identifier for each data point. In the case of CSV this is the column name, and in the case of JSON, it can be any named data field, having only one value such as a string or a number.
- `fluidDiagrams.getParser().setchildIdentifier(id)`: defines the field name which represents child relations within the data.
- `fluidDiagrams.getParser().setSiblingIdentifier(id)`: defines the field name which represents sibling relations within the data.
- `fluidDiagrams.getParser().setRawData(data)`: passes the raw data to the parser. `data` is a variable containing the actual data such as the CSV string or the JSON object.
- `fd.setVisualisationType(visualisation)`: defines the visualisation that should be used. At the time of this documentation, there were six visualisations available: ConeTree, ParallelCoordinates, LineChart, BarChart, ScatterPlot, and HyperbolicBrowser.
- `new FDEventHandler(debug,fluidDiagram)`: Initialises the event handler that belongs to the selected layout. `debug` activates or deactivates debug output to the browsers console. The event handler requires an instance of FluidDiagrams.

A.3 Creating the Camera

There are three types of camera to choose from, each requiring slightly different setup. The differences between orthographic and perspective projection are demonstrated in Appendix C.1. It is recommended to look at the example visualisations in order to choose the correct camera setup.

- *Orthographic Camera*: Uses orthographic projection, and is used for 2d visualisations.
  - `new THREE.OrthographicCamera(left, right, top, bottom, near, far)`: `left`, `right`, `top`, `bottom` define the boundaries of the viewing plane. Usually, this is set relative to the size of the visualisation with `left` = $-$`width`/2, `right` = `width`/2, `top` = `height`/2, and `bottom` = $-$`height`/2. The parameters `near` and `far` define the clipping distances. Anything placed nearer or further away from the camera will not be displayed.
  - `camera.position.x = x`: positions the camera.
  - `camera.lookAt(new THREE.Vector3(x, y, z))`: points the camera in the desired direction.
- *Perspective Camera*: Uses perspective projection, and is used for 3d visualisations.
  - `new THREE.PerspectiveCamera(viewAngel, aspect, near, far);`: Creates a perspective camera and sets the view angle, the aspect ratio, and the near and far clipping plane. The view angle and the aspect ratio define the viewable area, while the clipping planes define the nearest and farthest point from the camera at which objects are displayed.
  - `camera.position.x = x`: positions the camera.
  - `camera.lookAt(new THREE.Vector3(x, y, z))`: points the camera in the desired direction.
- *Combined Camera*: This camera combines an orthographic and a perspective camera into one, making it possible to switch between the two modes in real time. It also provides additional functionality such as a zoom method.
  - `new THREE.CombinedCamera(width, height, `$-$`150, near, far, near,far);`
  - `camera.position.x = x`: positions the camera.

  - `camera.lookAt(new THREE.Vector3(x, y, z))`: points the camera in the wanted direction .

Putting everything together, create and position a camera as above and then:

- `fluidDiagrams.setCamera(THREE.camera)`: sets the camera.
- `fluidDiagrams.run()`: starts the visualisation

Appendix B

Developer Guide

This chapter aims at developers wanting to add new functionality to the framework, by implementing new visualisations, event handlers, and parsers. Although TypeScript is not mandatory, it is recommended for development of any FluidDiagrams extension. Using TypeScript makes it easier to include new components in the existing libraries. TypeScript is available with a VisualStudio plug-in and as a command-line compiler. It is recommended to use the VisualStudio plug-in, because it provides extra support, such as documentation and auto-complete, which comes in handy, especially for developers new to TypeScript and FluidDiagrams. Additionally, it is advised to read the Three.JS documentation at Three [2013] and study existing code before attempting to implement a new visualisation.

B.1 Implementing a Visualisation

Basically, the visualisation should match FDNodes created by the parser to corresponding geometry positioned in 2d or 3d. For this, the method `visualise()` needs to be implemented, which is called by FluidDiagrams during the setup of the visualisation. The first step when creating new visualisations is to extend the existing FDLayout class to create a custom visualisation class. This is achieved by using the `extends` keyword: `class MyVisualisation extends FDVisualisation{...}`. The two mandatory methods for this task are:

- `constructor(debug: Boolean, fluidDiagrams: FluidDiagrams)`: This is the constructor of the class and needs to call `super(debug,fluidDiagrams)`, which calls the parent constructor, for correct instantiation of this object.
- `public visualise()`: As mentioned above this is the function called by FluidDiagrams to create the visualisation.

The FDVisualisation base class provides member variables that are necessary for any visualisation:

- `nodes`: An array of FDNodes containing all root nodes from the data structure. In the case of a linear data structure, the array will hold all of the nodes. If the data hierarchy is composed of two or more separate trees, the array will hold each root node. In the case of a graph structure, the first record of the data will become the root. This array can be used as a starting point for traversing the FDNodes.
- `allNodes`: Unsorted list of all FDNodes.
- `fluidDiagrams`: A reference to the main class.

The FDNodes also provide functionality crucial to any visualisation:

- `setData(data)`: Sets the node's data as a JSON object, to be accessed as a normal member variable. This is set by the parser.
- `getData()`: Returns the data object.
- `addChild(child)`: Appends a child to this node. `child` is of type FDNode. Set by the parser.
- `addSibling(sibling)`: Appends a sibling of type FDNode. Set by the parser.
- `setUniqueId(uid)`: This sets the unique identifier for this node. Set by the parser.
- `getUniqueId()`: Returns the unique identifier.
- `getChildren()`: Returns the array of children.
- `setParent(parent)`: Defines the node's parent node. Set by the parser.
- `getParent()`: Returns the parent node.
- `transform(matrix)`: Transforms the nodes scene graph according to the given transformation matrix.
- `getNodeSceneGraph()`: Returns the nodes scene graph.
- `getMeshIds()`: Returns an array of mesh ids. These ids are unique to each THREE.mesh and are set when new THREE.meshes are created.
- `rotate(angle, axis)`: Rotates the node's scene graph. `angle` defines the rotation angle in radians. The `axis` defines the rotation axis as a vector. The vector [1,0,0] sets the rotation axis to be the x axis.
- `rotateLocal(angle, axis)`: Similar to `rotate`, but operates on each mesh individually.
- `getAllGeometry()`: Returns an array of THREE.Mesh, with all the geometry of this node.
- `addGeometry(geomObject)`: Adds geometry to this node.
- `setPosition(position)`: Sets the position of this node, relative to its position within the scene graph.
- `getPosition()`: Returns the position of this node, relative to its position within the scene graph.
- `getWorldCoordinates()`: Returns the node's position in world coordinates.

B.2 Implementing an Event Handler

The event handler is responsible for all interaction with the visualisation. Thus, it must implement the required browser events. Currently, FluidDiagrams supports the following events:

- `onMouseDown(e)`: executed when the left mouse button is pressed. `e` is a reference to the actual DOM event, and can be used to obtain additional information about the event.
- `onMouseUp(e)`: executed when the left mouse button is released. `e` is a reference to the actual DOM event, and can be used to obtain additional information about the event.
- `onMouseMove`: executed when the mouse is moved over the canvas element. `e` is a reference to the actual DOM event, and can be used to obtain additional information about the event.

```
1  public onMouseClick(e: Event, element) {
2    if (element != null) {
3    if(element instanceof FDNode){
4      // The element clicked was a FDNode
5    }else{
6      //a mesh that is not associated with a FDNode was clicked
7    }
8    }else{
9    //There was no object at the coordinates of the click event.
10   }
```

Listing B.1: Demonstrates a simple mouse click event handler, which detects the kind of element that was clicked.

- `mouseWheel(e)`: executed when the scroll wheel is turned. `e` is a reference to the actual DOM event, and can be used to obtain additional information about the event.
- `onMouseClick(e, element)`: executed when the left mouse button is pushed and released within a certain time frame. `e` is a reference to the actual DOM event, and can be used to obtain additional information about the event. `element` is the object, that was at the position of the click event.

Every event is optional, meaning that it does not have to be implemented. However, no basic interaction behaviour is provided by FluidDiagrams, resulting in a non-interactive visualisation when no event handlers are implemented. A short example of a click event is shown in Listing B.1.

B.3 Implementing a Parser

When implementing a new parser, it is only required to implement the `.parse()` method. The parser creates a data structure consisting of FDNodes for later use in any visualisation. Depending on the input data, this structure can be one or multiple trees, one or multiple directed or undirected graphs, or a list of root nodes.

The FDParser parent class defines a series of member variables, which can be used by the parser, or are set by the parser:

- `rootNodes`: `FDNode[]`: an array of all root nodes of the data. A hierarchy will have only one root node. Linear data however will have root nodes only.
- `allNodes`: `FDNode[]`: is an unsorted list of all FDNodes. When using linear data, this will be identical to `rootNodes`.

The above two members must to be assigned by the parser, as these are the entry points for every visualisation, and represent FluidDiagrams internal data structure.

- `rawData`: holds the unprocessed data.
- `uId`: defines the field name of each data point which acts as the unique identifier for the created FDNode.
- `childrenId` and `siblingId`: provide a means of identifying the fields which define child and sibling relations within the raw data.

Appendix C

Computer Graphics

This appendix provides some mathematical background about 3d computer graphics. For more details, see the classic 3d graphics text book by Hughes et al. [2013].

C.1 Projection

Projection is the process of creating a 2d image of a 3d scene. This can be done in multiple ways, of which the two used in FluidDiagrams, are perspective projection and orthographic projection.

C.1.1 Perspective Projection

This type of projection takes in to account that fact that objects which are further away from the viewer (or the viewing plane) appear to be smaller. This behaviour is demonstrated in Figure C.1. Calculating the position of any given point in 3d space on the 2d viewing plane is done by the simple matrix multiplication:

$$p' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix} * p$$

with

$$d = p_z$$

This projection is usually used for 3d graphics.

C.1.2 Orthographic Projection

The basic principle of orthographic projection is shown in Figure C.2. It is a projection, where objects in 3d space appear to be the same size regardless of the distance to the viewing plane. Orthographic projection is usually used for 2d graphics. Calculating a projected point on the viewing plane is done by the simple matrix multiplication:

$$p' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * p$$

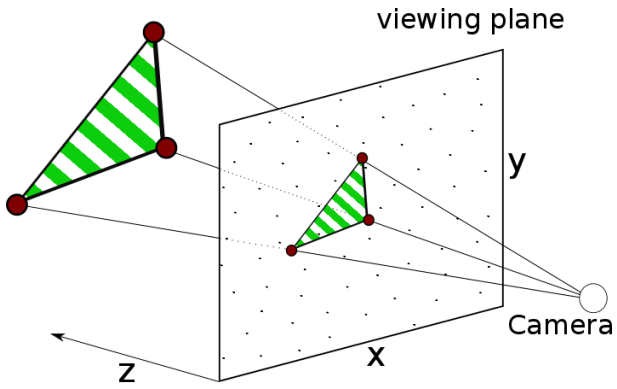


Figure C.1: The principle of perspective projection. Image adapted from [Wenke, 2012].

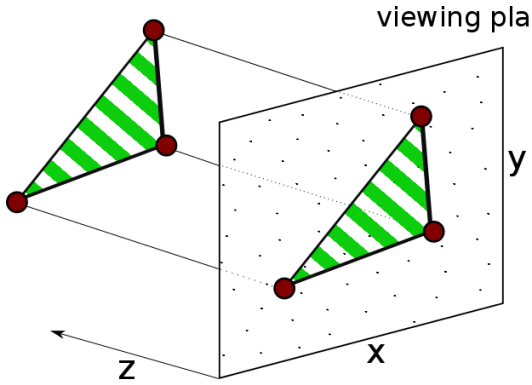


Figure C.2: The principle of orthographic projection. The scene is projected along parallel lines to the viewing plane. Image created by the author.

C.2 Scene Graph

A scene graph is a data structure for organising elements within a scene. It enables fast and efficient manipulation of the scene. The scene graph is composed of a hierarchy, which represents the relations of objects. Each node of the scene graph represents a object in the scene. Each node can again be composed of other nodes, creating a hierarchy of elements. The benefit of this hierarchical structure is that transformations applied to one node are also applied to its children. Each transformation is applied relative to its parent node. So, if the scene consists of a car on a road, and the car consists of multiple objects such as the body and the wheels, moving the car will result in moving all of its child elements as well. This allows for a simple way of creating complex transformations, such as turning wheels, while the car is moving, without having to calculate the rotation and translation of each wheel separately. An example scene graph representing a car and a street is demonstrated in Figure C.3.

C.3 Affine Transformation

Affine transformations are the three basic vector operations required in 3d graphics. They allow the manipulation of 3d objects in space. The three transformations are [Anyuru, 2012]:

- *Translation*: Translation is the process of moving an object in 3d space and can be represented by

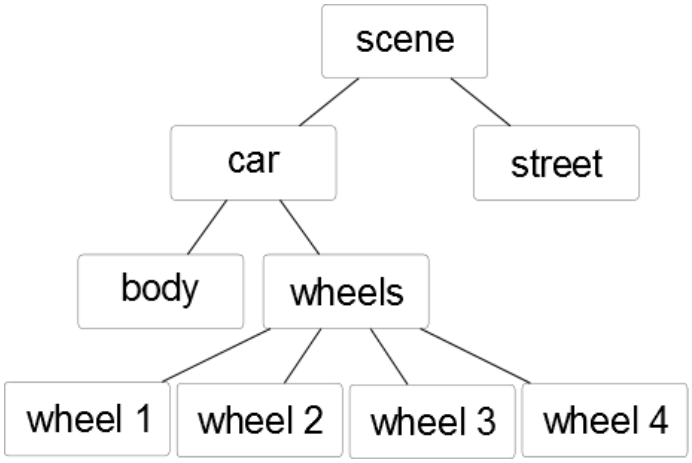


Figure C.3: Basic scene graph with a primitive car and a street. Image created by the author.

the following matrix multiplication:

$$p' = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 0 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} * p = \begin{bmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{bmatrix}$$

where t is the translation vector.

- *Rotation*: Rotation is the process of rotating an object by a given angle around a given axis passing through the origin of the object, and can be represented by the following transformation matrix:
  - Rotation around X-axis:
  $$R_x\theta = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
  - Rotation around Y-axis:
  $$R_y\theta = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
  - Rotation around Z-axis:
  $$R_z\theta = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
- *Scaling*: Scaling is the process of scaling along a given axis, represented by the following transformation matrix.
  $$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
  where s is the scaling vector.

Bibliography

Andrews, Keith [2013]. *Information Visualisation: Lecture Notes*. June 12, 2013. http://courses.iicm.tugraz.at/ivis/ivis.pdf (cited on pages xi, 3, 4, 8).

Andrews, Keith et al. [2001]. "Search Result Visualisation with xFIND". In: *Proc. User Interfaces to Data Intensive Systems (UIDIS 2001)*. June 1, 2001, pages 50–58. ISBN 0769508340. doi:10.1109/UIDIS.2001.929925. http://dl.acm.org/citation.cfm?id=884776 (cited on pages 5, 8).

Anscombe, Francis J. [1973]. "Graphs in Statistical Analysis". *The American Statistician* 27.1 (Feb. 1973), pages 17–21. ISSN 0003-1305. http://jstor.org/stable/2682899 (cited on pages 4–6).

Anyuru, Andreas [2012]. *Professional WebGL Programming: Developing 3D Graphics for the Web*. Wrox, May 1, 2012. ISBN 1119968860 (cited on pages 49, 50, 52, 106).

Belmonte, Nicolas Garcia [2013a]. *JavaScript InfoVis Toolkit*. Oct. 10, 2013. http://philogb.github.io/jit/ (cited on page 65).

Belmonte, Nicolas Garcia [2013b]. *JIT API Documentation*. Oct. 10, 2013. http://philogb.github.io/jit/static/v20/Docs/files/Core/Core-js.html (cited on page 66).

Bostock, Michael [2013]. *Data-Driven Documents*. Oct. 21, 2013. http://d3js.org/ (cited on pages 66, 68).

Bostock, Michael, Vadim Ogievetsky, and Jeffrey Heer [2011]. "D3: Data-Driven Documents". *IEEE Transactions on Visualization and Computer Graphics* 17.12 (Dec. 2011), pages 2301–2309. ISSN 1077-2626. doi:10.1109/TVCG.2011.185 (cited on pages 65, 66).

Carrière, Jeromy and Rick Kazman [1995]. "Interacting with Huge Hierarchies: Beyond Cone Trees". In: *Proc. IEEE Information Visualization (InfoVis '95)*. IEEE, Oct. 30, 1995, pages 74–81. doi:10.1109/INFVIS.1995.528689 (cited on pages xi, 81, 82).

Cherry, Ben [2010]. *Writing Testable JavaScript*. July 8, 2010. adequatelygood.com/Writing-Testable-JavaScript.html (cited on page 39).

CoffeeScript [2010]. CoffeeScript web site. Sept. 1, 2010. http://coffeescript.org/ (cited on pages 23–25).

CreateJS [2013]. *CreateJS*. Oct. 2, 2013. http://createjs.com/ (cited on page 55).

Dailey, David, Jon Frost, and Domenico Strazzullo [2012]. *Building Web Applications with SVG*. Microsoft Press, Aug. 8, 2012. ISBN 0735660123 (cited on page 44).

Deveria, Alexis [2013]. *Can I Use Web Workers*. July 1, 2013. http://caniuse.com/webworkers (cited on page 39).

ECMA [2011]. *ECMAScript Language Specification*. ECMA-262. ECMA International, June 1, 2011. http://ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf (cited on page 9).

EoW [2010]. *Evolution of the Web*. Sept. 1, 2010. http://evolutionoftheweb.com/ (cited on page 43).

Fenton, Steve [2013]. *TypeScript For JavaScript Programmers*. lulu.com, Apr. 22, 2013. ISBN 1291107371 (cited on page 23).

Fertig, Scott, Eric Freeman, and David Gelernter [1996]. "Lifestreams: An Alternative to the Desktop Metaphor". In: *CHI'96 Video Program*. (Vancouver, British Columbia, Canada). ACM. Apr. 13, 1996. doi:10.1145/257089.257404. http://sigchi.org/chi96/proceedings/videos/Fertig/etf.htm (cited on page xi).

Flanagan, David [2011]. *jQuery Pocket Reference*. O'Reilly, Jan. 4, 2011. ISBN 1449397220 (cited on page 19).

Freeman, Eric and Scott Fertig [1995]. "Lifestreams: Organizing your Electronic Life". In: *Proc. AAAI Fall Symposium: AI Applications in Knowledge Navigation and Retrieval*. Association for the Advancement of Artificial Intelligence, 1995, pages 38–44 (cited on page 6).

Gershon, Nahum, Stephen G. Eick, and Stuart Card [1998]. "Information Visualization". *interactions* 5.2 (Mar. 1, 1998), pages 9–15. ISSN 1072-5520. doi:10.1145/274430.274432 (cited on page 3).

Groves, Mat [2014]. *Pixi.js*. Jan. 7, 2014. https://github.com/GoodBoyDigital/pixi.js/ (cited on page 59).

Heer, Jeffrey [2010]. *Flare Dependency Graph*. Nov. 1, 2010. http://flare.prefuse.org/apps/dependency_graph (cited on pages xi, 5, 7).

Heilmann, Christian [2012]. *JavaScript Events And Responding To The User*. Aug. 17, 2012. http://coding.smashingmagazine.com/2012/08/17/javascript-events-responding-user (cited on pages 14, 15).

Highsoft [2014]. *Highcharts*. Jan. 7, 2014. http://highcharts.com/docs/getting-started/your-first-chart (cited on pages 71, 72).

Hughes, John F. et al. [2013]. *Computer Graphics: Principles and Practice*. 3rd edition. Addison-Wesley, July 22, 2013. ISBN 0321399528 (cited on page 105).

IEEE [2008]. *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2008. The Institute of Electrical and Electronics Engineers, Aug. 29, 2008. doi:10.1109/IEEESTD.2008.4610935. http://ac.usc.es/arith19/sites/default/files/3670a225-spec-session-DFP-paper2.pdf (cited on page 9).

Inselberg, Alfred [1985]. "The Plane with Parallel Coordinates". *The Visual Computer* 1.2 (1985), pages 69–91. ISSN 0178-2789. doi:10.1007/BF01898350 (cited on pages 5, 8, 83).

Jonker, D. et al. [2013]. "Aperture: An Open Web 2.0 Visualization Framework". In: *Proc. 46th Hawaii International Conference on System Sciences (HICSS 2013)*. Jan. 7, 2013, pages 1485–1494. doi:10.1109/HICSS.2013.96 (cited on pages 65, 68).

jQuery [2013a]. *jQuery*. Aug. 29, 2013. http://jquery.com/ (cited on page 19).

jQuery [2013b]. *jQuery API Documentation*. Aug. 29, 2013. http://api.jquery.com/ (cited on pages 20, 21).

json.org [2013]. *Introducing JSON*. Aug. 29, 2013. http://json.org/ (cited on page 16).

Kantor, Ilya [2013]. *Searching Elements in DOM*. July 31, 2013. http://javascript.info/tutorial/searching-elements-dom (cited on page 13).

Kelley, W. Michael and Robert A. Donnelly [2009]. *The Humongous Book of Statistics Problems*. ALPHA, Dec. 1, 2009. ISBN 1592578659 (cited on pages 84, 85).

Kuan, Joe [2012]. *Learning Highcharts*. Packt Publishing, Dec. 25, 2012. ISBN 1849519080. http://packtpub.com/learning-highcharts-for-javascript-data-visualization/book (cited on page 65).

Lamping, John, Ramana Rao, and Peter Pirolli [1995]. "A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies". In: *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI '95)*. (Denver, Colorado, USA). ACM, May 7, 1995, pages 401–408. ISBN 0201847051. doi:10.1145/223904.223956. http://www.sigchi.org/chi95/Electronic/documnts/papers/jl_bdy.htm (cited on pages xi, 5, 7, 85).

Lehni, Jürg and Jonathan Puckey [2014]. *Paper.js*. Jan. 7, 2014. http://paperjs.org/ (cited on page 59).

Leung, Catherine and Andor Salga [2010]. "Enabling WebGL". In: *Proc. of the 19th International Conference on World Wide Web (WWW '10)*. Raleigh, North Carolina, USA: ACM, Apr. 26, 2010, pages 1369–1370. ISBN 1605587990. doi:10.1145/1772690.1772933 (cited on page 49).

MacCaw, Alex [2012]. *The Little Book on CoffeeScript*. O'Reilly, Jan. 31, 2012. ISBN 1449321054 (cited on page 23).

Microsoft [2013]. *TypeScript*. Aug. 29, 2013. http://typescriptlang.org/ (cited on page 23).

Nightingale, Florence [1858]. *Notes on Matters Affecting the Health, Efficiency and Hospital Administration of the British Army*. 1858 (cited on page 4).

Oculus [2014]. *Aperture JS*. Jan. 7, 2014. http://aperturejs.com/ (cited on page 70).

Parisi, Tony [2012]. *WebGL Up and Running*. O'Reilly, Aug. 2, 2012. ISBN 144932357X (cited on page 62).

Patokallio, Jani [2012]. *Openflights Routedb*. Jan. 1, 2012. http://openflights.org/data.html (cited on pages xi, 4).

Pivotal [2013]. *Jasmine Testing Framework*. Aug. 29, 2013. http://pivotal.github.com/jasmine/ (cited on pages 29, 30, 32–34).

Ramos, Ernesto and David Donoho [1983]. *cars.csv*. 1983. http://lib.stat.cmu.edu/datasets/ (cited on pages 83, 86).

Raphaël [2013]. *Raphaël*. Oct. 2, 2013. http://raphaeljs.com/ (cited on page 58).

Robertson, George G. et al. [1991]. "Cone Trees: Animated 3D Visualizations of Hierarchical Information". In: *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI '91)*. New Orleans, Louisiana, USA: ACM, Apr. 28, 1991, pages 189–194. ISBN 0897913833. doi:10.1145/108844.108883 (cited on page 81).

Shepard, Eric et al. [2013]. *Mozilla Event Reference*. June 26, 2013. http://developer.mozilla.org/en-US/docs/Mozilla_event_reference (cited on pages 14, 20).

Sheridan, Malcom [2013]. *The Developer's Guide to HTML5 Canvas*. Sept. 13, 2013. http://msdn.microsoft.com/en-us/hh534406.aspx (cited on page 45).

Stefanov, Stoyan [2010]. *JavaScript Patterns*. O'Reilly, Sept. 28, 2010. ISBN 0596806752 (cited on pages 35–39).

Three [2013]. *Three.JS - Getting Started*. Oct. 2, 2013. https://github.com/mrdoob/three.js/wiki/Getting-Started (cited on page 101).

Veer, Emily A. Vander [2006]. *Flash 8: The Missing Manual*. O'Reilly, Mar. 29, 2006. ISBN 0596101376 (cited on page 43).

W3C [2005]. *Document Object Model (DOM)*. W3C. Jan. 19, 2005. http://w3.org/DOM (cited on page 13).

W3C [2011]. *Scalable Vector Graphics (SVG) 1.1 (Second Edition). W3C Recommendation*. Dean Jackson and Craig Northway. Aug. 16, 2011. http://w3.org/TR/SVG11/ (cited on page 44).

W3C [2012a]. *Web Workers. W3C Candidate Recommendation*. W3C. May 1, 2012. http://w3.org/TR/workers/ (cited on page 39).

W3C [2012b]. *XMLHttpRequest*. Julian Aubourg and Jungkee Song and Hallvord R. M. Steen. Dec. 6, 2012. http://w3.org/TR/XMLHttpRequest (cited on page 17).

W3C [2013]. *HTML 5.1. W3C Candidate Recommendation*. Robin Berjon and Steve Faulkner and Travis Leithead and Erika Doyle Navara and Edward O'Conner and Silvia Pfeiffer. Aug. 6, 2013. http://www.w3.org/TR/html5/ (cited on page 43).

Wenke, Henning [2012]. *Computergraphik: Lecture Slides*. May 22, 2012. http://www-lehre.inf.uos.de/~cg/2012/PDF/2012-05-22%20Viewing%20&%20Projection%%2020(web).pdf (cited on page 106).

Wikipedia [2006]. *Bone Reconstruction*. Jan. 29, 2006. http://en.wikipedia.org/wiki/File:Bonereconstruction.jpg (cited on pages xi, 4).

Wikipedia [2010]. *Anscombe's Quartet*. Mar. 26, 2010. http://en.wikipedia.org/wiki/File:Anscombe's_quartet_3.svg (cited on pages xi, 6).

Wikipedia [2013a]. *Ajax (Programming)*. Aug. 24, 2013. http://en.wikipedia.org/wiki/Ajax_(programming) (cited on pages 17, 18).

Wikipedia [2013b]. *Model-View-Controller*. Aug. 26, 2013. http://en.wikipedia.org/wiki/Model-view-controller (cited on page 38).

Willison, Simon [2006]. *A Re-Introduction to JavaScript*. Mar. 7, 2006. http://developer.mozilla.org/en-US/docs/JavaScript/A_re-introduction_to_JavaScript (cited on page 9).

Zakas, Nicholas C. [2012]. *Professional JavaScript for Web Developers*. 3rd edition. Wrox, Jan. 18, 2012. ISBN 1118026691 (cited on pages 9, 16).