

Institut für Visualisierung und Interaktive Systeme

Universität Stuttgart
Universitätsstraße 38
D – 70569 Stuttgart

Studienarbeit Nr. 2189

Umsetzung einer Fokus-und-Kontext-Technik
zur Vergrößerung von graphischen
Benutzungsoberflächen als Firefox-Extension

Wadim Peters

Studiengang: Informatik

Prüfer: Prof. Dr. Thomas Ertl

Betreuer: Dipl. Inf. Christiane Taras

begonnen am: 01.09.2008

beendet am: 03.03.2009

CR-Klassifikation: H.1.2., H.4.3., H.5.2., H.5.4., K.4.2.

Inhaltsverzeichnis

Abbildungsverzeichnis.....	3
1 Einleitung.....	5
1.1 Motivation.....	5
1.2 Konzept.....	6
1.3 Aufgabenstellung.....	7
1.4 Struktur der Arbeit.....	8
2 Grundlagen und verwendete Technologien.....	9
2.1 Technologien.....	9
2.1.1 XUL.....	9
2.1.2 XPCOM.....	11
2.1.3 FUEL.....	12
2.1.4 CSS.....	12
2.1.5 JavaScript und DOM.....	14
2.1.5.1 JavaScript.....	14
2.1.5.2 DOM.....	14
2.1.5.3 Ereignisse.....	15
2.2 Firefox und Erweiterungen.....	15
2.2.1 Firefox.....	15
2.2.2 Aufbau einer Erweiterung.....	17
3 Vergrößerung der Elemente des Hauptfensters von Firefox.....	18
3.1 Konzept.....	18
3.2 Erweiterung des Konzepts.....	19
3.2.1 Fisheye-Methode.....	19
3.2.2 Vergrößerung der Leisten.....	20
3.3 Umsetzung des Konzepts.....	21
3.3.1 Vergrößerung der Leisten.....	21
3.3.2 Fisheye-Methode.....	25
3.3.3 Beeinflussung der Darstellung.....	26
3.3.4 Tooltips.....	27
3.4 Navigation mit der Tastatur.....	29
3.5 Weitere Konzeptlösungen für Toolbars.....	30
4 Vergrößerung von Webseitenelementen mit dem Lupenkonzept.....	31
4.1 Konzept.....	31
4.2 Umsetzung.....	32
4.2.1 Aufbau der Lupe.....	32
4.2.2 Abläufe.....	34

4.2.2.1 Öffnen einer neuen Lupe.....	34
4.2.2.2 Laden eines Webseitenelements in die Lupe.....	35
4.2.2.3 Synchronisation.....	36
4.2.3 Funktionen.....	36
4.2.3.1 Vergrößerung der Lupenleiste.....	36
4.2.3.2 Funktionen der Lupenleiste.....	38
4.2.3.3 Pop-up-Menü der Lupenleiste.....	40
4.2.3.4 Öffnen einer Lupe.....	40
4.3 Alternative Umsetzung - HTML statt XUL.....	41
5 Dialoge und Fenster.....	42
5.1 Konzept.....	42
5.2 Erweiterung des Konzepts.....	42
5.3 Umsetzung des Konzepts.....	44
5.4 Zusätzliche Funktionen.....	45
5.4.1 Speicherfunktion.....	45
5.4.2 Verändern der Größe von Elementen und Fenstern.....	46
5.4.3 Weitere Funktionen.....	46
6 Vergrößerung von Webseitenelementen ohne Lupen.....	48
6.1 Umsetzung.....	48
6.2 Navigation mit der Tastatur.....	49
7 Fazit.....	50
Anhang A Tastenkombinationen in FirefoxZoom (deutsche Lokalisierung).....	51
Anhang B Verzeichnisstruktur der Erweiterung.....	52
Literaturverzeichnis.....	53

Abbildungsverzeichnis

Abbildung 1: Das Fenster zum Codebeispiel.....	10
Abbildung 2: DOM-Baum des Beispielcodes.....	15
Abbildung 3: Gestreckte Schaltflächen.....	19
Abbildung 4: Schaltflächen mit Fisheye-Methode.....	19
Abbildung 5: Verhalten von Leisten.....	20
Abbildung 6: Leisten in Firefox in Originalansicht.....	21
Abbildung 7: Ablauf bei der Vergrößerung einer Leiste	24
Abbildung 8: Lupe im Einsatz.....	32
Abbildung 9: Lupenleiste unvergrößert und vergrößert.....	37
Abbildung 10: Ein vergrößertes Element über einem fokussierten Element.....	45

1 Einleitung

Der Personal Computer (PC) ist in Beruf und Alltag zu einem unverzichtbaren Werkzeug geworden. Bereits im Jahr 2007 arbeitete in der Europäischen Union (EU) die Hälfte aller Beschäftigten mindestens einmal die Woche mit einem PC [BIT09]. Drei Viertel aller Haushalte in Deutschland besaßen 2007 einen Computer [BIT209]. Auch das Internet ist inzwischen nicht mehr aus unserem Leben wegzudenken. Es ist heute eine unersetzliche Kommunikationsplattform und eine unerschöpfliche Informationsquelle. Im Jahr 2008 verfügten in Deutschland 75 Prozent der Haushalte über einen Online-Anschluss [BIT309]. Ohne den PC geht heute nichts mehr. Auch mobile Computer sind auf dem Vormarsch. Das Jahr 2008 war das Jahr der kleinen Notebooks, der Netbooks. Angesichts der Displaygrößen von etwa 9 Zoll (22,86 cm) kann die Darstellung auf diesen Geräten sehr klein werden und damit schlecht zu erkennen.

1.1 Motivation

Durch die zunehmende Alterung unserer Gesellschaft wächst die Zahl der leicht- bis mittelstark sehbehinderten Menschen. Für diese stellen die gewöhnlichen Darstellungsgrößen in Computern ein großes Hindernis dar. Die Lösung des Problems wäre eine Vergrößerung der grafischen Benutzeroberfläche und des dargestellten Anwendungsdokuments¹. Nach einer Untersuchung der vorhandenen Vergrößerungstechniken zeigte sich aber, dass keine der Techniken ein zufriedenstellendes Resultat bietet. Deshalb haben Dipl. Inf. Christiane Taras und Prof. Dr. Thomas Ertl ein neues Vergrößerungskonzept entwickelt [FKT08], das deutliche Vorteile gegenüber anderen Vergrößerungstechniken hat und sehbehinderten Menschen die Arbeit an einem PC erleichtern wird. Die vorliegende Arbeit soll das entwickelte Konzept anhand einer Erweiterung für den Webbrowser Mozilla Firefox umsetzen.

Dieser bietet sich für die Umsetzung an, weil er, aufgrund seines modularen Aufbaus, der leicht beeinflussbaren Benutzeroberfläche und der Unterstützung von zahlreichen Programmierschnittstellen, leicht erweiterbar ist. Der Webbrowser kann als Standardanwendung betrachtet werden und steht repräsentativ für andere Anwendungen, auf welche das Konzept angewendet werden kann.

Das neue Konzept beinhaltet wichtige Punkte, die derzeit verfügbare Vergrößerungstechniken nicht erfüllen. Die Erhöhung der Standard-Schriftgröße führt z.B. dazu, dass wegen der dann vergrößerten Benutzeroberfläche dem Anwendungsdokument weniger Platz zur Verfügung steht. Eine Alternative wäre die Vergrößerung eines rechteckigen Bildschirmausschnitts (Bildschirmlupe). Allerdings kann diese Methode nicht garantieren, dass alle Textinhalte auf eine für den Benutzer notwendige Mindestgröße der Schrift vergrößert werden. Dasselbe gilt für

¹Anwendungsdokument: Ein Dokument, das innerhalb einer Anwendung dargestellt wird, z.B. eine Webseite in einem Webbrowser

1.1 Motivation

die anwendungsbezogene Zoomfunktion wie sie in Webbrowsern und Büro-Anwendungen verfügbar ist. Außerdem kann mit dieser Funktion nur das Anwendungsdokument vergrößert werden. Die Vergrößerung des gesamten Bildschirminhaltes auf ein Vielfaches der Bildschirmgröße kann zwar Abhilfe schaffen, leider führt diese Vergrößerungstechnik zu einem Verlust der Übersicht über den Bildschirminhalt. Auch die Herabsetzung der Bildschirmauflösung ist keine ideale Lösung. Folglich erscheint zwar alles größer und man behält den Überblick, doch die Qualität der Darstellung leidet unter der geringeren Auflösung.

Der folgende Abschnitt beschreibt weitere Kriterien, die das neue Konzept erfüllt. Für die Betrachtung der Kriterien in den oben genannten Vergrößerungstechniken wird auf die wissenschaftliche Publikation von Dipl. Inf. Christiane Taras und Prof. Dr. Thomas Ertl verwiesen.

1.2 Konzept

Das neu entwickelte Konzept beschreibt eine Vergrößerungstechnik, die großen Wert auf die Vergrößerung derjenigen Bereiche legt, die für den Benutzer in einer bestimmten Situation relevant sind. Dabei wird auf semantische Zusammenhänge zwischen den Elementen geachtet. Beispielsweise soll bei Fokussierung einer Schaltfläche die gesamte Werkzeugleiste vergrößert werden. Die Erhaltung der Original-Ansicht ist ebenfalls wichtig, denn das erlaubt dem Benutzer den Überblick zu behalten und stellt sicher, dass Informationen vollständig und in gewohnter Qualität dargestellt werden. Das Ziel des Konzepts ist es, den Platz von Benutzeroberflächen möglichst effektiv zu nutzen.

Das Konzept erfüllt folgende Kriterien:

1. Textinhalte müssen in großer, gut erkennbarer Schrift dargestellt werden. Die Vergrößerung um einen Faktor ist dafür nicht geeignet, weil das Ergebnis dann von der Ausgangsschriftgröße abhängt, der Benutzer aber auf eine bestimmte Mindestschriftgröße angewiesen ist. Deshalb müssen alle Textinhalte auf eine bestimmte Schriftgröße erhöht werden. Dass dadurch das Unterscheiden von Elementen, wie z.B. Text und Überschrift, erschwert wird, ist nebensächlich, weil man diese auch anhand anderer Attribute, wie z.B. fette Schrift, unterscheiden kann. Zusätzlich muss die Schriftgröße veränderbar sein, um verschiedene Stufen der Sehschwäche zu unterstützen.
2. Vergrößerte Textinhalte müssen gut und klar erkennbar sein und dürfen nicht verschwommen oder verpixelt dargestellt werden.
3. Da der Mauszeiger das am häufigsten verwendete Zeigegerät ist mit dem die auf dem Bildschirm dargestellten Objekte beeinflusst werden, muss dieser in einer für den Benutzer ausreichenden Größe dargestellt werden.

4. Es muss möglich sein, sowohl die Benutzeroberfläche als auch das Anwendungsdokument zu vergrößern.
5. Alle auf dem Bildschirm dargestellten Elemente müssen auch bei Vergrößerung gut erkennbar sein. Sie dürfen z.B. nicht außerhalb des Sichtbereichs des Benutzers liegen.
6. Das fortlaufende Lesen von Texten sollte mit möglichst keinem horizontalen Scrollen verbunden sein, da dies ermüdend und verwirrend ist. Außerdem ist das vertikale Scrollen bequemer, da heutzutage alle Computermäuse ein Scrollrad besitzen.
7. Es muss möglichst viel Platz für die Darstellung der Anwendungsdokumente zur Verfügung stehen.
8. Da die Verschlechterung des Sehvermögens oft schleichend voranschreitet, sollte der Benutzer problemlos eine andere Schriftgröße einstellen können.
9. Ausreichend große Icons sind sehr wichtig um das Erkennen ihrer Bedeutung zu gewährleisten. Deshalb müssen diese in möglichst guter Qualität auf die notwendige Größe skaliert werden.
10. Logisch zusammenhängende Informationen (z.B. alle Einträge eines Menüs) müssen gemeinsam vergrößert werden.
11. Das Layout der Benutzeroberfläche sollte keine größeren Veränderungen erfahren. Zunächst einmal kann der Benutzer mit der Anwendung schon vertraut gewesen sein bevor die Augenschwäche einsetzte. Aber auch sonst ist eine Änderung des Layouts hinderlich, weil das eine zusätzliche Belastung bedeutet und weil sich dann die Darstellung der Anwendung von der der Allgemeinheit unterscheidet, was zu Missverständnissen führen kann.

1.3 Aufgabenstellung

Die vorliegende Arbeit hat das Ziel das in Abschnitt 1.2 beschriebene Konzept vollständig in einer stabil laufenden Firefox-Erweiterung² umzusetzen³. Die Verwendung der Vergrößerung sollte sowohl mit der Maus als auch mit der Tastatur möglich sein. Außerdem sollte der Benutzer die Schriftgröße zur Laufzeit ändern können.

Die Arbeit soll sich nacheinander mit folgenden Punkten befassen:

1. Vergrößerung der Elemente des Hauptfensters von Firefox
2. Vergrößerung von Webseitenelementen mit dem Lupenkonzept

²Der Name der Erweiterung ist FirefoxZoom.

³Quelle: Aufgabenstellung der Studienarbeit.

1.3 Aufgabenstellung

3. Vergrößerung kleiner Dialoge, wie z.B. Passworteingabedialoge

Optionale Punkte sind:

4. Vergrößerung komplexer Dialoge, wie z.B. Konfigurationsdialoge
5. Vergrößerung von Webseitenelementen ohne Lupen

Probleme bei der Umsetzung sind festzuhalten, wo möglich sollen Lösungsmöglichkeiten aufgezeigt werden. Außerdem soll erfasst werden, ob Maus- und Tastaturnavigation unterschiedliche Anforderungen an das Konzept oder die Umsetzung stellen. Für die optionalen Punkte muss das Konzept eventuell erweitert werden.

1.4 Struktur der Arbeit

Das Konzept, das in Abschnitt 1.2 vorgestellt wurde, ist die Grundlage dieser Arbeit. Das Kapitel 2 behandelt weitere grundlegende Themen wie den Webbrowser Firefox, den Aufbau von Erweiterungen sowie diverse Technologien, die bei der Erstellung einer Erweiterung verwendet werden. Betrachtet man die Aufgabenbereiche aus der Aufgabenstellung im vorigen Abschnitt, dann lassen sich die einzelnen Bereiche bestimmten Kapiteln zuweisen. Kapitel 3 widmet sich dem ersten Punkt, der Vergrößerung der Elemente des Hauptfensters von Firefox. Kapitel 4 behandelt die Vergrößerung von Webseitenelementen mit dem Lupenkonzept. Kapitel 5 beinhaltet die Punkte drei und vier, die Vergrößerung kleiner und komplexer Dialoge. Und schließlich geht es in Kapitel 6 um die Vergrößerung von Webseitenelementen ohne Lupen. Das letzte Kapitel fasst die Ergebnisse zusammen und gibt eine kritische Betrachtung bezüglich der Erweiterung und des Konzepts.

2 Grundlagen und verwendete Technologien

Im ersten Teil dieses Kapitels werden Technologien vorgestellt, die bei der Entwicklung einer Firefox-Erweiterung verwendet werden und auch bei der Entwicklung von FirefoxZoom von Bedeutung waren. Im zweiten Teil wird auf den Webbrowser eingegangen sowie auf die im weiteren Verlauf der Ausarbeitung häufig verwendeten Firefox-Begriffe. Außerdem wird die Funktionsweise und der Aufbau einer Erweiterung erläutert.

2.1 Technologien

2.1.1 XUL

XUL (XML User Interface Language) ist eine auf XML basierende Auszeichnungssprache zur Beschreibung von plattformunabhängigen grafischen Benutzeroberflächen [XUL09]. Mit XUL legt man die Struktur einer Benutzeroberfläche fest. Für die Darstellung wird CSS (siehe Abschnitt 2.1.4) verwendet, für das Verhalten JavaScript (siehe Abschnitt 2.1.5). Beispiele für XUL-basierte Anwendungen sind die Mozilla-Anwendungen Firefox, Thunderbird und Sunbird.

Da XUL eine XML-Sprache ist, unterliegt sie den XML-Regeln. Ein XML-Dokument darf genau ein Wurzelement enthalten. Die einzelnen Elemente werden mittels Tags beschrieben. Ein öffnender Tag `<element>` muss stets geschlossen werden, entweder durch `</element>` oder direkt mit `<element/>`. Öffnende und geschlossene Tags müssen sich auf derselben Ebene befinden. Elemente können Eigenschaften besitzen, die entweder als Attribute oder als Kindknoten dargestellt werden. In XUL werden alle Attribute klein geschrieben und dürfen nicht innerhalb eines Tags doppelt vorkommen. Ihre Werte werden in Anführungszeichen gesetzt: `<element attribut="wert"/>`.

Der Quellcode einer XUL-Datei könnte z.B. folgendermaßen aussehen:

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
<window xmlns="http://www.mozilla.org/.../there.is.only.xul">
  <vbox>
    <label value="Warnung! Fenster wird geschlossen"/>
    <checkbox label="Warnung nicht mehr anzeigen"/>
    <hbox>
      <spacer flex="1"/>
      <button label="OK"/>
      <button label="Abbrechen"/>
      <spacer flex="1"/>
    </hbox>
  </vbox>
</window>
```

2.1.1 XUL

In der ersten Zeile wird die XML-Version festgelegt, die zweite Zeile gibt an, wo sich die Darstellung der XUL-Elemente verbirgt. Das `<window>`-Element in der folgenden Zeile ist das Wurzelement der XUL-Datei, das Attribut `xmlns` legt den Namensraum der Elemente fest. Die Elemente `<label>` und `<checkbox>` sind selbsterklärend, `<vbox>` und `<hbox>` ordnen ihre Kindknoten vertikal bzw. horizontal an. Erwähnenswert sind noch die beiden `<spacer>`-Elemente, womit Abstände gemeint sind. Das Verhältnis der beiden Abstände ist eins zu eins, was mit dem Attribut `flex` und dem Wert „1“ festgelegt wird. Lässt man den Code von der Gecko Rendering Engine interpretieren, sieht das Ergebnis folgendermaßen aus:

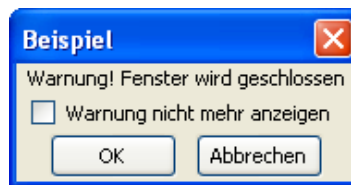


Abbildung 1: Das Fenster zum Codebeispiel

Ein besonderer Typ sind die anonymen Elemente. Das sind Elemente, die innerhalb des DOM-Baums (siehe Abschnitt 2.1.5.2) unsichtbar sind. Der Grund dafür ist, dass diese Elemente automatisch eingefügt werden und XUL sowohl die eingefügten Elemente als auch das Element, in das eingefügt wird, als ein einziges Element sieht [ANC09]. Die Technologie, die das ermöglicht, nennt sich XBL (XML Binding Language). Nimmt man das oben beschriebene Fenster als Beispiel, dann könnte man den Inhalt eines standardisierten `<hbox>`-Elements mit „OK“- und „Abbrechen“-Schaltflächen automatisch einfügen lassen. Der entsprechende Quellcode müsste dann geändert werden:

XUL:

```
...
<label value="Warnung! Fenster wird geschlossen"/>
<checkbox label="Warnung nicht mehr anzeigen"/>
<hbox class="OKCancel"/>
...
```

Das `<hbox>`-Element enthält hier keine Kindknoten. Stattdessen wird dem Element ein Attribut zugewiesen, infolgedessen die `<hbox>` mit einer XBL-Datei verbunden wird, welche die einzufügenden Elemente beschreibt:

CSS:

```
.OKCancel {
    -moz-binding: url('chrome://.../codebeispiel.xbl#OKCancel');
}
```

XBL:

```
<binding id="OKCancel">
  <content>
    <xul:spacer flex="1"/>
    <xul:button label="OK"/>
  </content>
</binding>
```

```

        <xul:button label="Abbrechen"/>
        <xul:spacer flex="1"/>
    </content>
</binding>

```

Mit allgemein bekannten Methoden ist es nicht möglich beim Traversieren eines DOM-Baums die anonymen Elemente zu erfassen. Allerdings lassen sich diese mit XPCOM erweitern.

2.1.2 XPCOM

XPCOM (Cross Platform Component Object Model) ist ein plattformunabhängiges Komponentenmodell von Mozilla, vergleichbar mit COM von Microsoft [XPC09]. Es ist eine Entwicklungsumgebung, die eine Vielzahl an Komponenten zur Verfügung stellt, auf die dank Sprachanbindungen mit verschiedenen Programmiersprachen zugegriffen werden kann. Auch das Erstellen neuer Komponenten ist mit mehreren Programmiersprachen möglich. Die Kommunikation zwischen den Komponenten und Anwendungen erfolgt über Schnittstellen, die von der XPCOM-Schicht XPConnect nach JavaScript oder eine andere Sprache reflektiert werden [MOZ09].

In FirefoxZoom werden an verschiedenen Stellen XPCOM-Schnittstellen genutzt. Die wichtigste Komponente, auf die hier zugegriffen wird, ist *Stylesheet Service*. Die damit zur Verfügung stehenden Funktionen ermöglichen es, ein Stylesheet auf die Firefox-Benutzeroberfläche, die darin geladenen Dokumente und alle sonstigen Firefox-Fenster gleichzeitig anzuwenden. Diese Komponente ist sehr wichtig für die Erweiterung, da jede Änderung der Schriftgröße durch den Benutzer, eine Änderung der Darstellung der Benutzeroberfläche und des Anwendungsdokuments mit sich bringt. Dank dieser Komponente wird es möglich die Darstellung zu ändern, indem erzeugte CSS-Dateien dynamisch eingebunden oder ausgetauscht werden.

Das Einbinden geschieht, indem die CSS-Datei geladen und „registriert“ wird:

```

var sss = Components.classes["@mozilla.org/content/
    style-sheet-service;1"].getService(Components
    .interfaces.nsIStyleSheetService);
var ios = Components.classes["@mozilla.org/network/io-service;1"]
    .getService(Components.interfaces.nsIIOService);
var uri = ios.newURI("chrome://firefoxzoom/content/always.css", null,
    null);
sss.loadAndRegisterSheet(uri, sss.USER_SHEET);

```

Die hier geladene Datei enthält CSS-Regeln, die unabhängig von der Schriftgröße sind und somit während der gesamten Laufzeit der Firefox-Erweiterung gültig sind. Eigentlich werden hier sogar zwei Komponenten genutzt. Zum einen *Stylesheet Service*, zum anderen ein I/O-Service, der die Datei vor dem Laden in ein URI-Objekt einbindet.

2.1.2 XPCOM

Bevor eine neue CSS-Datei mit einer neuen Standard-Schriftgröße „registriert“ werden kann, muss die alte Datei entfernt werden. Das bewirkt folgende Anweisung:

```
sss.unregisterSheet(uri, sss.USER_SHEET);
```

XPCOM bietet professionellen Entwicklern zahlreiche Verwendungsmöglichkeiten. Für Hobby-Entwickler von Erweiterungen ist es aber zu anspruchsvoll. Deshalb hat Mozilla FUEL entwickelt.

2.1.3 FUEL

FUEL (Firefox User Extension Library) ist eine JavaScript-Bibliothek, die in Firefox mit Version 3.0 integriert wurde. Sie richtet sich an Entwickler von Erweiterungen und bietet Lösungen für die gängigsten Programmierprobleme an. Vor allem aber wird eine Alternative für lange und komplizierte XPCOM-Anweisungen angeboten, indem diese durch moderne JavaScript-Befehle ersetzt werden [FUE09].

In FirefoxZoom wird FUEL überwiegend in Zusammenhang mit Variablen verwendet, die auch nach Neustart erhalten bleiben sollen. In dem folgenden Beispiel sieht man wie das Abfragen solcher Variablen mit FUEL erheblich vereinfacht wird:

XPCOM:

```
var prefs = Components.classes["@mozilla.org/preferences-  
service;1"].Service(Components.interfaces.nsIPrefService);  
prefs = prefs.getBranch("extensions.ffz.");  
var fSize = prefs.getIntPref("fffontsize");
```

FUEL:

```
var fSize = Application.prefs.get("extensions.ffz.fffontsize");
```

Auf Änderungen der Variablen kann automatisch reagiert werden. Die folgende Anweisung sorgt dafür, dass eine Funktion aufgerufen wird, wenn sich eine Variable ändert:

```
Application.prefs.get("extensions.ffz.fffontsize")  
    .events.addListener("change", OnFontSizeChange);
```

Langfristig möchte Mozilla mit FUEL die Portabilität von Erweiterungen verbessern und Entwickler anziehen. Nach und nach soll die Bibliothek erweitert werden.

2.1.4 CSS

CSS (Cascading Style Sheets) ist eine deklarative Stylesheet-Sprache, die dazu verwendet wird die Darstellung von HTML- und XML-Inhalten zu beschreiben. Der Gedanke hinter CSS ist die Trennung von Inhalt und Darstellung. Durch die Deklaration von Klassen können bestimmte Elemente oder sogar ganze Dokumente einheitlich, aber flexibel formatiert werden. Außerdem kann die Darstellung an das Ausgabemedium angepasst werden. In XUL-

basierten Anwendungen wird CSS dazu verwendet das Layout der Benutzeroberfläche zu gestalten. Im Folgenden wird die CSS-Syntax erläutert.

Die einzelnen Anweisungen werden als Regeln bezeichnet [W3C09]. Eine Regel besteht aus einem Selektor und einem Deklarationsblock. Der Selektor kann ein einfacher Selektor sein (z.B. „checkbox“ für alle Checkboxes) oder eine Verkettung von einfachen Selektoren, durch Kombinatorzeichen voneinander getrennt (z.B. „vbox > checkbox“ für alle Checkboxes, die Kindknoten einer vertikalen Box sind). Der Selektor bestimmt, auf welche Elemente die Deklarationen angewendet werden sollen. Diese Elemente werden als Subjekte des Selektors bezeichnet. Im Deklarationsblock befinden sich eine oder mehrere Deklarationen. Sie sind folgendermaßen aufgebaut: Eigenschaft (z.B. „font-size“ für die Schriftgröße), Doppelpunkt, Wert der Eigenschaft (z.B. „30px“), Semikolon. Der Wert der Eigenschaft bestimmt das Erscheinungsbild der entsprechenden Eigenschaft in den Subjekten.

```

selektor {
  eigenschaft1: wert1;
  eigenschaft2: wert2; } Deklarationsblock

```

Ein einfacher Selektor besteht aus einem Typselektor, der für einen Elementtyp steht, oder aus dem universellen Selektor „*“, der für alle Elemente steht. An diese kann eine oder mehrere Pseudo-Klassen angehängt werden. Pseudo-Klassen erlauben es Elemente anhand von veränderlichen Eigenschaften zu selektieren (z.B. „toolbar:hover“ für eine Toolbar, über welcher sich der Mauszeiger befindet). Die folgende Regel lässt an jedem Element, über welchem sich der Mauszeiger befindet, eine rote Umrandung erscheinen:

```

*:hover {
  outline: 3px solid red;
}

```

Anstatt von Pseudo-Klassen oder zusätzlich zu Pseudo-Klassen können Attribut-Selektoren verwendet werden. Damit werden Elemente anhand ihrer Attribute selektiert (z.B. „checkbox[checked='true']“ für alle Checkboxes, die aktiviert sind). Die folgende Regel sorgt für die Vergrößerung der Schrift in einer Textbox wenn sie fokussiert ist:

```

textbox[focused='true'] {
  font-size: 30px;
}

```

Für die Gestaltung von Elementen in FirefoxZoom werden verschiedene CSS-Eigenschaften verwendet, vor allem aber „font-size“ für die Schriftgröße, „height“, „width“ und Variationen davon für Abmessungen und „z-index“, das bei überlappenden Elementen die Reihenfolge der einzelnen Schichten festlegt.

Sehr wichtig für FirefoxZoom ist die Tatsache, dass Stylesheets zur Laufzeit erzeugt und mit Hilfe von XPCOM auf die Benutzeroberfläche angewendet werden, wodurch sich die Darstellung ändert. CSS kann allerdings nur für die Darstellung verwendet werden, für das Verhalten der Erweiterung ist der Einsatz von JavaScript nötig.

2.1.5 JavaScript und DOM

2.1.5 JavaScript und DOM

2.1.5.1 JavaScript

JavaScript ist eine im Dezember 1995 von Netscape veröffentlichte Skriptsprache [JSC09]. Sie wurde seitdem ständig weiterentwickelt und ist inzwischen nicht mehr aus der Webentwicklung wegzudenken. Das Einsatzgebiet von JavaScript ist vielfältig. Die Sprache kann z.B. zur Überprüfung von Formulareingaben verwendet werden, zum Setzen oder Auslesen von Cookies oder zum dynamischen Verändern einer Webseite. Im Allgemeinen wird mit JavaScript für die Interaktivität von Webseiten gesorgt. In Verbindung mit XUL ist JavaScript für das Verhalten der Benutzeroberfläche verantwortlich.

Mit JavaScript lässt sich sowohl prozedural als auch funktional bzw. objektorientiert programmieren. Der Code kann sowohl clientseitig als auch serverseitig ausgeführt werden. Hauptsächlich wird der Code clientseitig im Browser direkt interpretiert, er liegt also nicht in kompilierter Form vor. Daher ist die Ausführung von JavaScript-Code langsamer als bei kompilierten Anwendungen.

In Netscapes Bemühung die Grundfunktionalität von JavaScript zu standardisieren entstand 1997 die Skriptsprache ECMAScript [ECM09]. Sie wurde seitdem ebenfalls weiterentwickelt. JavaScript und JScript, das von Microsoft herausgebracht wurde, versuchen kompatibel zu ECMAScript zu sein. Aber sie bieten auch zusätzliche Features an, die in der ECMA-Spezifikation nicht enthalten sind.

2.1.5.2 DOM

DOM (Document Object Model) ist eine Spezifikation einer Programmierschnittstelle für den Zugriff auf HTML- und XML-Dokumente [DOM09]. Der Standard von W3C (World Wide Web Consortium) wird von allen gängigen Webbrowsern unterstützt. Damit kann der Inhalt und die Struktur von Webseiten und XUL-Benutzeroberflächen dynamisch verändert werden.

Der Zugriff auf ein Element ist mit der Methode *getElementById()* möglich, vorausgesetzt man kennt die ID des Elements [WDM09]. Kennt man nur den Namen oder die Klasse des Elements, dann kommt man mit *getElementsByName()* bzw. *getElementsByClass()* weiter. Mit *appendChild()* werden Kindknoten hinzugefügt und mit *removeChild()* entfernt. Das Auslesen von Attributen geschieht mit *getAttribute()*, das Setzen mit *setAttribute()* und das Löschen mit *removeAttribute()*.

Die Elemente eines HTML- oder XML-Dokuments befinden sich in verschiedenen Hierarchieebenen und in unterschiedlicher Beziehung zueinander. Bildet man alle Elemente auf die Baumstruktur ab, entsteht ein DOM-Baum (Abbildung 2). Der Knoten, der keine Elternknoten besitzt und sich damit auf der höchsten Hierarchieebene befindet, ist der Wurzelknoten. Bei XUL-Fenstern ist das ein `<window>`-, `<dialog>`- oder `<prefwindow>`-Element:

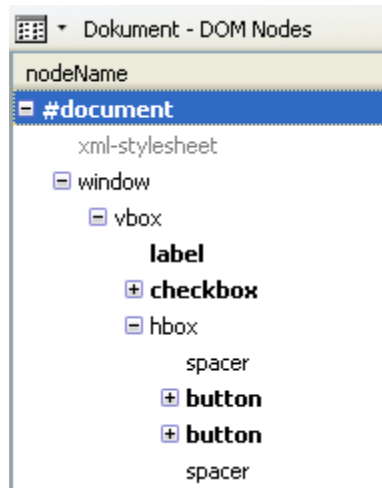


Abbildung 2: DOM-Baum des Beispielcodes

2.1.5.3 Ereignisse

Ein Ereignis tritt auf, wenn der Benutzer auf eine Taste drückt, den Mauszeiger über ein Element bewegt oder wenn der Browser eine Webseite vollständig geladen hat. Damit auf Ereignisse reagiert werden kann, besitzen alle Browser ein Ereignismodell. Diese können verwendet werden um Ereignisse abzufangen und als Reaktion darauf eine bestimmte Funktion auszuführen. Das nennt man einen Event-Handler.

Die von W3C definierte Methode *addEventListener()* [JAS03] kann jedem Element im DOM-Baum zugewiesen werden. Damit wird ein bestimmtes Ereignis abgefangen. Tritt das Ereignis auf, wird die zugewiesene Funktion ausgeführt.

In Firefox werden viele Event-Handler verwendet. Es wird u.a. auf Mausklicks („click“) „gehört“, auf Doppelklick („dblclick“), auf Bewegung des Mauszeigers über ein Element („mouseover“) oder Verlassen eines Elements mit dem Mauszeiger („mouseout“). Für Drag & Drop werden die Ereignisse „mousedown“, „mousemove“ und „mouseup“ verwendet.

2.2 Firefox und Erweiterungen

2.2.1 Firefox

Firefox ist ein Open-Source-Webbrowser des Mozilla-Projekts [FIR09], der aus dem freigegebenen Quellcode des Softwarepakets Netscape Communicator (Webbrowser, WYSIWYG-Editor, Mail- und News-Client) entstand [NSC09]. Im September 2002 kam die erste lauffähige Version des Firefox unter dem Namen Phoenix heraus. Inzwischen hat sich der

2.2.1 Firefox

Browser zum größten Konkurrenten des Internet Explorers von Microsoft entwickelt. Ende des Jahres 2008 hatte Firefox etwa 20 Prozent Marktanteil in Europa [FFO09].

Der Webbrowser kann vom Benutzer auf einfache Weise erweitert werden. Die Gestaltung der Benutzeroberfläche lässt sich optisch verändern, neue Funktionen können hinzugefügt werden. Für die optische Veränderung ist eine große Zahl von Themes verfügbar, für die Erweiterung der Funktionalität existieren Firefox-Erweiterungen (Add-ons), die von einer Mozilla-Webseite heruntergeladen oder direkt installiert werden können [ADO09]. Mit der Installation der kleinen Programme werden sie in den Browser integriert, worin sich der modulare Aufbau des Browsers zeigt.

Die Firefox-Fenster bestehen oftmals nicht aus einer einzigen XUL-Datei sondern aus einer Hauptdatei, die die Struktur des Fensters vorgibt und weiteren Dateien, die erst zur Laufzeit eingebunden werden [XUL07]. Beispielsweise werden beim Einstellungen-Fenster die Inhalte der einzelnen Tabs erst geladen, wenn der Benutzer sie das erste Mal anklickt. Alle diese Dateien befinden sich in Java-Archivdateien und werden zur Laufzeit entpackt und in den Speicher geladen.

Die Dateien können über eine Chrome-Adresse direkt im Webbrowser betrachtet werden. Das Einstellungen-Fenster hat z.B. die Adresse „chrome://browser/content/preferences/preferences.xul“. Der erste Tab des Einstellungen-Fensters ist unter der Chrome-Adresse „chrome://browser/content/preferences/main.xul“ erreichbar. Das Hauptfenster des Firefox hat die Adresse „chrome://browser/content/browser.xul“. Die ID des Wurzelements dieses Fensters ist „main-window“.

Gibt man eine Chrome-Adresse in das Adressfeld des Webbrowsers ein, dann wird der Inhalt im Browserbereich des Hauptfensters angezeigt. Das bedeutet, Firefox verwendet für das Rendern der eigenen Benutzeroberfläche dieselbe Rendering-Engine wie zum Anzeigen von Webseiten [PFF07].

Im Folgenden werden einige wichtige Begriffe erklärt, die im weiteren Verlauf der Arbeit verwendet werden:

- Hauptfenster – Damit ist das gesamte Browserfenster des Firefox gemeint, also alles von der Menü- bis zur Statusleiste. Die ID dieses Fensters ist „main-window“.
- Browserbereich – So wird der Bereich im Hauptfenster genannt, in welchem Webseiten angezeigt werden. In einem Standardfenster also der Bereich zwischen der Tab- und der Statusleiste.

2.2.2 Aufbau einer Erweiterung

Eine Firefox-Erweiterung kommt in Form einer XPI-Datei [XPI09]. Das auch als XPIInstall (Cross-Platform Install) bezeichnete Dateiformat wurde von Mozilla entwickelt und wird hauptsächlich von Mozilla-Anwendungen verwendet um den Funktionsumfang eines Programms zu erweitern. Im Grunde handelt es sich um ein ZIP-Archiv eines Verzeichnisses. Die Verzeichnisstruktur von FirefoxZoom ist im Anhang B (Seite 52) zu finden. Wichtige Dateien, die eine XPI-Datei enthalten kann, werden im Folgenden vorgestellt [EXT09].

- `install.rdf` – Diese Datei ist das Installationsscript der Erweiterung. Sie enthält Informationen, die für die Installation notwendig sind. Dazu gehören Angaben wie die ID, die Bezeichnung und die Versionsnummer der Erweiterung. Außerdem sind Angaben zur Anwendung erforderlich, auf welcher die Erweiterung installiert werden kann. Zu optionalen Angaben gehören u.a. die Beschreibung der Erweiterung, das Betriebssystem, für welches die Erweiterung entwickelt wurde, und die Chrome-Adresse des Optionen-Fensters.
- `chrome.manifest` – Diese Datei enthält Chrome-Adressen von wichtigen Dateien und Verzeichnissen. Möchte man das Aussehen eines Firefox-Fensters verändern, wird hier der Pfad der XUL-Datei angegeben, die die Änderungen enthält. Diese Datei, die man als Overlay bezeichnet, wird dann beim Start des Fensters geladen. Außerdem gibt man hier das Verzeichnis an, das Design (skin) der XUL-Elemente enthält. Zu weiteren Angaben gehören Verzeichnispfade zu Dateien, die Übersetzungsdaten der Erweiterung enthalten.
- `overlay.xul` – Ein Overlay enthält in der Regel Änderungen des Hauptfensters und wird beim Start der Erweiterung geladen. Hier können z.B. zusätzliche Menüeinträge oder Schaltflächen definiert werden. Overlays kann es auch für andere Fenster geben.
- `overlay.css` – Diese Datei wird mit dem Overlay geladen. Sie enthält Formatierungsregeln, die das Aussehen des Hauptfensters verändern.
- JavaScript-Dateien enthalten die Funktionalität der Erweiterung. Es ist üblich den Quellcode nach bestimmten Kriterien aufzuteilen. Beispielsweise können Objekte in einer eigenen Datei gespeichert werden.
- `*.dtd` und `*.properties` – Für jede unterstützte Sprache, existiert im Ordner „locale“ ein Ordner für diese Dateien. Sie enthalten Übersetzungen für die in der Erweiterung verwendeten Bezeichnungen, Namen, Überschriften und sonstige Textinhalte. Für Bezeichnungen in XUL-Dateien verwendet man DTD-Dateien, für übersetzte Texte in JavaScript-Dateien eignen sich Java-Properties-Dateien.

3 Vergrößerung der Elemente des Hauptfensters von Firefox

3.1 Konzept

Für die Vergrößerung der Benutzeroberfläche von Firefox wurde eine Vergrößerungstechnik gewählt, in der die Elemente direkt in der Originalansicht vergrößert werden. Ausgehend von der normalen, unvergrößerten Ansicht der Anwendung sollen bei Bedarf diejenigen Elemente in der Größe verändert werden, die für den Benutzer in der jeweiligen Situation relevant sind. Was den Benutzer interessiert kann durch die Position des Mauszeigers oder des Eingabecursors festgestellt werden. Es sollten aber keine Elemente dauerhaft vergrößert werden, da dadurch die Übersicht über die Anwendung gestört wird und dem Anwendungsdokument weniger Platz zur Verfügung steht. Für bestimmte Elemente, wie Werkzeugleisten, sollte es möglich sein, die Vergrößerung auf permanent einzustellen.

Außer dem gerade fokussierten Element, sollen auch Elemente vergrößert werden, die logisch mit dem fokussierten Element zusammenhängen. Wird z.B. eine Schaltfläche in einer Werkzeugleiste fokussiert, muss die gesamte Werkzeugleiste vergrößert werden. Bei Fokussierung eines Menüeintrags müssen alle Einträge auf gleicher Ebene und alle Einträge auf übergeordneten Ebenen vergrößert werden. Um dem Benutzer die Orientierung zu erleichtern, sollte in dem Fall die gesamte Menüleiste vergrößert werden. Dazugehörige Tooltips und Icons müssen natürlich ebenfalls vergrößert werden.

Elemente, die in der Benutzeroberfläche standardmäßig sichtbar sind, müssen für den Benutzer ohne Probleme anvisierbar sein und dürfen deshalb ihre Position bei Fokussierung bzw. Vergrößerung nicht verändern. Außerdem müssen die Elemente auch bei Vergrößerung auf ihre ursprüngliche Ausdehnung beschränkt werden (alternativer Ansatz siehe Abschnitt 3.2.1). Elemente, die nur in bestimmten Situationen sichtbar werden, können schon vor ihrem Erscheinen vergrößert werden, um Irritationen zu vermeiden, die entstehen wenn Elemente zunächst im normalen Zustand erscheinen aber sofort vergrößert werden. Zusätzlich muss gewährleistet sein, dass Elemente sowohl im vergrößerten als auch im unvergrößerten Zustand über dieselben Funktionalitäten verfügen. So soll das Verhalten von vergrößerten Eingabefeldern und Schaltflächen dem Verhalten im unvergrößerten Zustand entsprechen.

3.2 Erweiterung des Konzepts

3.2.1 Fisheye-Methode

Das Konzept fordert, dass Elemente auch im vergrößerten Zustand auf ihre ursprüngliche Breite beschränkt werden. Allerdings würden dann Schaltflächen in Werkzeugleisten bei sehr großen Schriftgrößen stark gestreckt aussehen wenn sie fokussiert sind (Abbildung 3).



Abbildung 3: Gestreckte Schaltflächen

Die Fisheye-Methode ist ein Kompromiss zwischen Position, Ausdehnung und Qualität der Elemente. Dabei wird das gerade fokussierte Element sowohl in der Höhe als auch in der Breite vollständig vergrößert, während alle anderen logisch zusammenhängenden Elemente nur soweit vergrößert werden, wie es der Platz für sie hergibt (Abbildung 4). Dadurch werden alle Elemente, die sich rechts von dem fokussierten Element befinden, um die Vergrößerungsdifferenz (Breite des fokussierten Elements in Abbildung 4 minus Breite des fokussierten Elements in Abbildung 3) nach rechts verschoben. Position und Ausdehnung der Elemente werden also nur zum Teil eingehalten. Allerdings hat diese Methode den Vorteil, dass die fokussierten Elemente besser zu erkennen sind. Gerade bei Elementen mit Beschriftung verspricht die Fisheye-Methode eine bessere Qualität, da ohne diese Methode solche Elemente bei hohen Schriftgrößen im vergrößerten Zustand nur noch aus wenigen Buchstaben bestehen würden und nicht mehr lesbar wären. Was das Anvisieren von Elementen angeht, bleiben diese in der Position unverändert, solange man ein noch unvergrößertes Element anvisiert.



Abbildung 4: Schaltflächen mit Fisheye-Methode

Der Nachteil dieser Methode ist das Anvisieren von teilweise vergrößerten Elementen, die sich rechts von dem fokussierten Element befinden. Ist das gerade fokussierte Element deutlich breiter als das rechts davon liegende Element, dann kann es vorkommen, dass bei Bewegen des Mauszeigers nach rechts, der Mauszeiger ein oder mehrere Elemente überspringt und sich plötzlich über einem weiter rechts liegenden Element befindet. Dieses Problem hätte man bei der Navigation mit der Tastatur nicht.

3.2.2 Vergrößerung der Leisten

3.2.2 Vergrößerung der Leisten

Ein ähnliches Problem wie im vorigen Abschnitt beschrieben, haben Leisten wenn mehrere von ihnen übereinander liegen. Man stelle sich vor, man möchte den Mauszeiger von einer fokussierten Leiste auf die darunter liegende Leiste bewegen. In dem Moment, in dem der Mauszeiger die fokussierte, also vergrößerte Leiste verlässt, wird diese auf die Originalgröße verkleinert, die darunter liegende Leiste würde also nach oben verschoben. Bei hohen Schriftgrößen kann es nun vorkommen, dass der Mauszeiger sich plötzlich eine Leiste tiefer befindet. Das ist natürlich nicht erwünscht und sollte verhindert werden. Die Lösung sieht so aus, dass die Leisten nicht verkleinert werden, wenn sich der Mauszeiger nach unten bewegt (Abbildung 5).



Abbildung 5: Verhalten von Leisten

Dieses Verhalten widerspricht dem Konzept, dass nur semantisch zusammenhängende Elemente gleichzeitig vergrößert werden. Das Problem hätte man bei der Navigation mit der Tastatur nicht, denn in dem Fall braucht man derartiges Verhalten nicht. Die Navigation mit der Tastatur, die hier nicht umgesetzt wurde, hat sogar einige Vorteile gegenüber der Navigation mit der Maus. Abgesehen von dem Platz, den man spart, wenn zu jedem Zeitpunkt maximal eine Leiste vergrößert ist, muss der Benutzer seine Fokussierung bei hohen Schriftgrößen kaum wechseln. Jede vergrößerte Leiste befindet sich dann nämlich beinahe an derselben Stelle wo sich auch die darüber oder darunter liegende Leiste im vergrößerten Zustand befinden würde. Weitere Konzeptlösungen zur Vergrößerung von übereinander liegenden Leisten sind in Abschnitt 3.5 aufgeführt.

3.3 Umsetzung des Konzepts

3.3.1 Vergrößerung der Leisten

In einem Firefox-Fenster gibt es verschiedene Arten von Leisten. Im oberen Bereich des Anwendungsfensters befindet sich die Menüleiste, darunter sind die Navigations-Symbolleiste, die Lesezeichen-Symbolleiste und die Tab-Leiste (Abbildung 6).

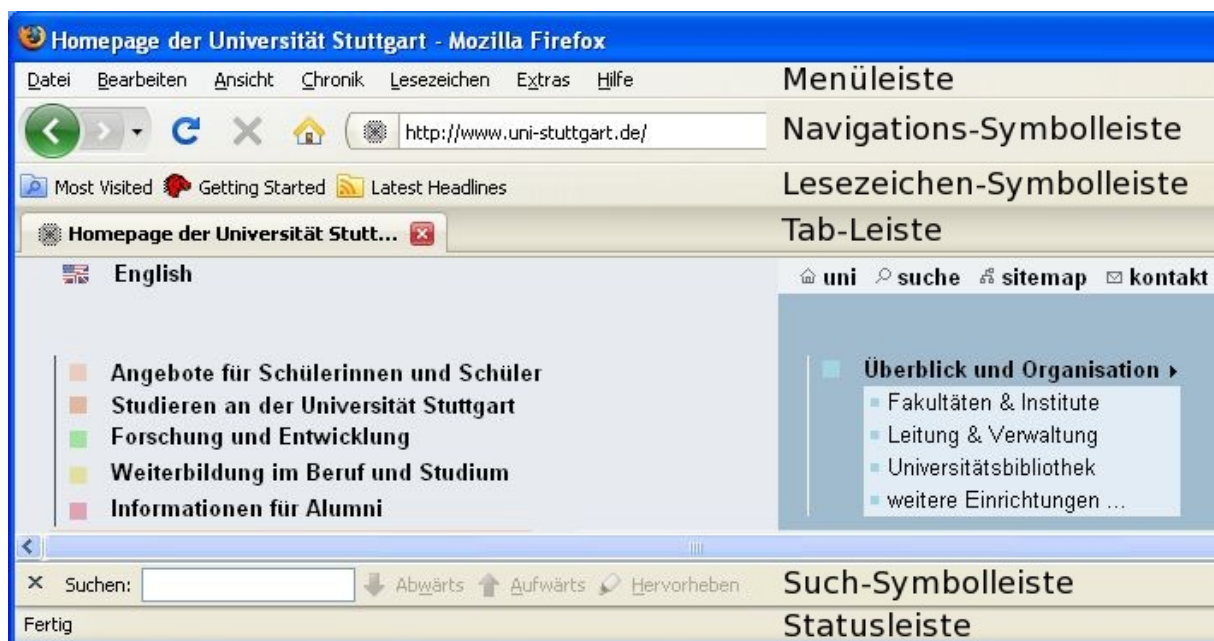


Abbildung 6: Leisten in Firefox in Originalansicht

Im unteren Bereich des Fensters findet man die Statusleiste, darüber die Such-Symbolleiste - diese ist aber standardmäßig unsichtbar, und wird nur aktiv wenn der Benutzer auf der Webseite nach einem Begriff sucht. Weitere Leisten können vom Benutzer manuell hinzugefügt oder mit einer Erweiterung installiert werden. Auch Sidebars können zu den Toolbars gezählt

3.3.1 Vergrößerung der Leisten

werden. Hier gibt es die Chronik- und die Lesezeichen-Sidebar, die ebenfalls standardmäßig nicht sichtbar sind.

Nimmt man nun das zugrunde liegende Konzept, dann muss jede der in Abbildung 6 dargestellten Leisten – mit Ausnahme der Such-Symbolleiste – immer dann vergrößert werden, wenn der Benutzer eine Zeit lang mit dem Mauszeiger über der Leiste verweilt. Sobald der Benutzer diese mit dem Mauszeiger wieder verlassen hat, sollte sie in den Originalzustand zurückkehren. Dasselbe gilt für manuell hinzugefügte oder installierte Leisten. Für die Such-Symbolleiste ist diese Art der Vergrößerung nicht notwendig, da der Benutzer diese Leiste sofort erkennen möchte, wenn er sie aktiviert. Sie soll also immer vergrößert dargestellt werden.

Zunächst aber müssen die Leisten so angepasst werden, dass sie zu jedem Zeitpunkt dem Konzept entsprechen. Unter anderem müssen alle Elemente einer Leiste in ihrer Breite begrenzt werden. Das wird erzielt, indem die aktuelle Breite eines jeden Elements bestimmt wird und die CSS-Eigenschaften „min-width“ und „max-width“ auf den ermittelten Wert gesetzt werden. Dadurch behalten die Elemente ihre Breite immer bei. Da der Benutzer ein Fenster eventuell in der Größe verändern könnte und sich dadurch auch der zur Verfügung stehende Platz ändert, werden bei jeder Änderung der Fenstergröße alte Begrenzungen verworfen und der Vorgang neu durchgeführt. Dieser Vorgang wird für alle Werkzeugleisten durchgeführt.

Die Anpassungen der Menüleiste gehen einen Schritt weiter. In der Originalansicht nutzt die Menüleiste nicht die gesamte Breite des Hauptfensters aus, sodass die Menütitel im vergrößerten Zustand nur zum Teil sichtbar wären. Damit das nicht passiert, wird die Menüleiste gestreckt. Das `<toolbar spring>`-Element, der flexible Zwischenraum rechts neben der Menüleiste, wird ausgeblendet. Genauso wie der Throbber (Grafik oben rechts, die anzeigt, dass eine Webseite geladen wird), der an der Stelle ohnehin überflüssig ist, da auch die Tabs einen Throbber anzeigen. Dadurch wird Platz frei und die Menütitel können auf die gesamte Breite des Fensters gestreckt werden. Die Änderungen an der Menüleiste verändern das Aussehen des Firefox dauerhaft, was eigentlich nicht im Sinne des Konzepts ist. Andererseits bringen die Änderungen Vorteile bei der Benutzung. Nach diesen vorbereitenden Maßnahmen entsprechen die Leisten dem Konzept und können vergrößert werden.

Die vom Fokus abhängige Vergrößerung der Leisten kann auf verschiedene Weise gelöst werden. Zum einen könnte man den Toolbar-Elementen – bei der Menüleiste wäre das ein `<toolbar>`-Element – Event-Listeners zuweisen, die jedes Mal, wenn der Mauszeiger über die Leiste bewegt wird, ein „mouseover“-Ereignis melden und eine Funktion aufrufen, die die Leiste schließlich vergrößern würde. Das Verlassen der Leiste würde entsprechend ein „mouseout“-Ereignis auslösen und die Leiste wieder in den Originalzustand versetzen. Leider werden die Ereignisse auch dann ausgelöst, wenn der Mauszeiger von einem Element einer Leiste auf ein anderes Element derselben Leiste wechselt. Bei einigen Elementen werden so-

3.3.1 Vergrößerung der Leisten

gar Ereignisse ausgelöst während man den Mauszeiger nur darüber bewegt. Der Grund dafür ist, dass das Element aus mehreren anderen Elementen besteht, was aber nicht immer zu erkennen ist. Man bräuchte dann einen Algorithmus, der überprüft ob der Mauszeiger sich tatsächlich über einer Leiste befindet bzw. diese tatsächlich verlassen hat. Dieser Algorithmus wäre sehr kompliziert, da er unzählige Spezialfälle berücksichtigen müsste.

Auch nur unter Verwendung von CSS ist diese Art der Vergrößerung realisierbar. Den Toolbar-Elementen müssten bestimmte CSS-Regeln zugewiesen werden, was bei „hover“, also wenn sich der Mauszeiger über den Leisten befindet, zur Vergrößerung der Leisten führen würde. Leider gibt es aber auch hier ein Problem. Die Leisten würden sofort vergrößern sobald sich der Mauszeiger über ihnen befindet. Auch bei der schnellsten Mausbewegung über eine Leiste, würde diese für kurze Zeit aktiv werden. Flackern wäre die Folge. Um aber dem Benutzer Zeit zu geben sein Ziel zu wählen bevor die Vergrößerung der Leiste einsetzt, wird eine zeitverzögerte Vergrößerung von Leisten benötigt, die aber nur unter Verwendung von CSS nicht machbar ist.

Die in der Firefox-Erweiterung verwendete Lösung bedient sich beider Methoden. Einerseits werden bei den Toolbar-Elementen die Attribute `onmouseover` und `onmouseout` gesetzt, was dazu führt, dass bei „mouseover“- und „mouseout“-Ereignissen die Funktion *CheckAndForward2* aufgerufen wird. Andererseits werden den Elementen CSS-Regeln zugewiesen, die bei „hover“ eine CSS-Eigenschaft der Leiste geringfügig ändern. Die Eigenschaft „border-right-color“ wird dann auf „rgb(0, 0, 1)“ gesetzt, sonst beträgt der Wert „rgb(0, 0, 0)“. Die zur selben Zeit aufgerufene Funktion *CheckAndForward2* überprüft, ob ein „mouseover“-Ereignis mit „rgb(0, 0, 1)“ bzw. ein „mouseout“-Ereignis mit „rgb(0, 0, 0)“ übereinstimmt. Wird auf diese Weise ein Ereignis bestätigt, wird im weiteren Verlauf, im Falle eines bestätigten „mouseover“-Ereignisses, das XUL-Attribut `ffzactive`⁴ der entsprechenden Leiste auf „true“ geschaltet, bei einem bestätigten „mouseout“-Ereignis wird das Attribut auf „false“ gesetzt. Beträgt der Wert des Attributs „true“, wird eine CSS-Regel gültig, die die Änderung der Schrift- und Icongröße der entsprechenden Leiste bewirkt. Nun befindet sich die Leiste in einem vergrößerten Zustand. Abbildung 7 zeigt den ganzen Ablauf in einem vereinfachten Schaubild. Die Zeichen über den Pfeilen sollen verdeutlichen, welche CSS-Regel bzw. JavaScript-Anweisung zu der Zustandsänderung führt.

⁴XUL erlaubt eigene Attribute zu verwenden

3.3.1 Vergrößerung der Leisten

XUL/JavaScript

1) `menubar.onmouseover = „check();“`

CSS

a) `menubar:hover { border-right-color: rgb(0, 0, 1); }`

b) `menubar[ffzactive="true"] { font-size: 30px; }`

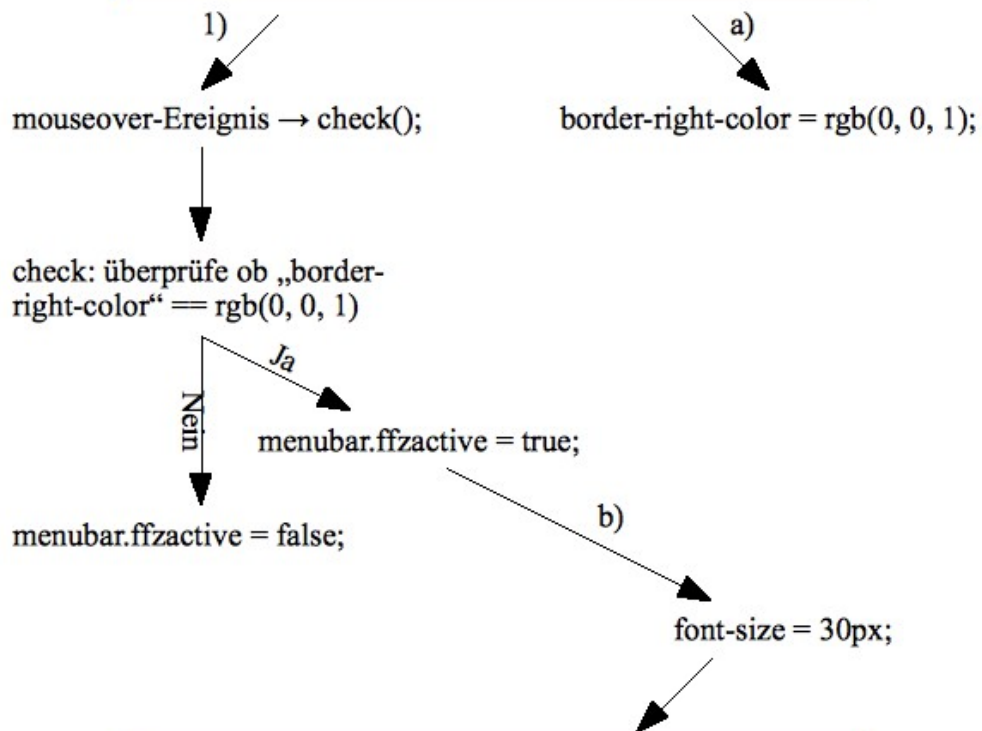


Abbildung 7: Ablauf bei der Vergrößerung einer Leiste

Verlässt der Mauszeiger eine Leiste, dann darf diese, der Erweiterung des Konzepts (siehe Abschnitt 3.2.2) zufolge, nicht in den Originalzustand versetzt werden, wenn sich der Mauszeiger nach unten bewegt. Deswegen überprüft die Funktion *Execute* vor dem Setzen des Attributs `ffzactive` welche Leisten aktiviert sind und welche Leisten durch die Zustandsänderung aktiviert oder deaktiviert werden müssen und führt dann die entsprechenden Anweisungen aus. Diese Funktion hat die Aufgabe den Zustand aller Leisten zu überwachen und inkonsistente Zustände zu verhindern (Abbildung 5).

Wie schon erwähnt, braucht man für die Navigation mit der Tastatur diese Art des Verhaltens nicht. Allerdings beschränkt sich die Navigation mit der Tastatur in FirefoxZoom auf Firefox-spezifische Tastaturbelegungen und die in beinahe jeder Windows-Anwendung enthaltene Möglichkeit sich mit der Tastatur durch die Menüleiste zu bewegen. Diese wird mit [Alt] aktiviert, was in der Erweiterung zwangsläufig zur Vergrößerung führt. Die dahinter ablaufenden Vorgänge sind beinahe identisch mit den oben beschriebenen für die Maus. Mit den Pfeiltasten bewegt man sich anschließend durch die vergrößerten Menüs. Außerdem kann mit [Ctrl]+[L] bzw. [K] die Adress- oder Suchleiste aktiviert werden. Da diese nur fokussiert werden, wenn man eine Webadresse eingeben möchte oder nach etwas sucht, werden in dem Fall nur die relevanten Elemente dieser Leistenelemente vergrößert und nicht die gesamte Werkzeugleiste. Das bewirkt eine CSS-Regel, die bei Fokussierung aktiv wird und für die Vergrößerung sorgt – wie in Abschnitt 2.1.4 beschrieben. Möglichkeiten, wie eine Toolbar-Navigation umgesetzt werden könnte, werden in Abschnitt 3.4 erläutert.

3.3.2 Fisheye-Methode

Die folgende CSS-Regel sorgt dafür, dass die Schaltflächen in der Navigations-Symbolleiste sich nach der Fisheye-Methode verhalten. Dabei wird die begrenzte Ausdehnung des fokussierten Elements aufgehoben indem die maximale Breite des Elements auf einen sehr großen Wert gesetzt wird:

```
#nav-bar[ffzactive='true'] toolbarbutton:hover {
    min-width: 0px !important;
    max-width: 500px !important;
}
```

Diese und zwei ähnliche Regeln für Lesezeichen in der Lesezeichen-Symbolleiste und für Tabs in der Tab-Leiste befinden sich in der Datei „always.css“. Regeln für hinzugefügte oder installierte Leisten werden in der Funktion *UpdateStylesheet*, welche sich in der Datei „stylesheet.js“ befindet, dynamisch erstellt und registriert. Möchte man die Fisheye-Methode nicht, aber stattdessen die im ursprünglichen Konzept beschriebene Idee umgesetzt haben, dann kann das gewünschte Ergebnis durch das Auskommentieren oder Löschen dieser CSS-Regeln bzw. JavaScript-Anweisungen erreicht werden. Da den Schaltflächen, vor allem Lesezeichen, dann viel weniger Platz für die Beschriftungen zur Verfügung steht, kann mit der folgenden Regel die Ausrichtung von horizontal auf vertikal umgestellt werden:

3.3.2 Fisheye-Methode

```
toolbarbutton {
  -moz-box-orient: vertical;
}
```

Dadurch erscheint das Icon einer Schaltfläche in einer und die Beschriftung in einer anderen Zeile. Diese CSS-Regel ist nur von Nutzen, wenn es sich um Schaltflächen handelt, die einen Text enthalten.

3.3.3 Beeinflussung der Darstellung

Fast alle Änderungen der Benutzeroberfläche des Firefox können CSS-Regeln zugeschrieben werden, die entweder immer wirksam sind oder in bestimmten Situationen wirksam werden.

Die folgende, immer wirksame Regel ist dafür da, dass die Menütitel in der Menüleiste bei hohen Schriftgrößen nicht über die begrenzte Ausdehnung hinausgehen:

```
menu > label {
  min-width: inherit !important;
  max-width: inherit !important;
}
```

Die folgende Regel wird wirksam, wenn sich der Mauszeiger über der Menüleiste befindet, wenn ein Menü in der Menüleiste mit der Tastatur aktiviert wird oder wenn sich der Mauszeiger über der Tab-Leiste befindet. In diesen Fällen wird die Eigenschaft „border-right-color“ des entsprechenden Elements verändert. Diese Regel sorgt also dafür, dass die Fokussierung der Leisten erkannt wird (siehe Abschnitt 3.3.1):

```
#toolbar-menubar:hover,
menu[_moz-menuactive='true'],
.tabbrowser-tabs:hover {
  border-right-color: rgb(0, 0, 1) !important;
}
```

Die beiden Regeln befinden sich in der Datei „always.css“, welche bei Start der Erweiterung geladen wird. Darin befinden sich Regeln, die unabhängig von der Schriftgröße sind.

Alle von der Schriftgröße abhängigen Regeln und Regeln, die dynamisch erstellt werden müssen, werden durch den Aufruf der Funktion *UpdateStylesheet* erstellt und mit einer CSS-Datei registriert (siehe Abschnitt 2.1.2). Ändert sich die Schriftgröße, wird das alte Stylesheet entfernt und ein neues registriert.

Die folgende zur Erstellung der CSS-Datei verwendete, aber hier vereinfachte JavaScript-Anweisung addiert zu der String-Variable „style“ eine weitere CSS-Regel, welche für die Vergrößerung der Schriftgröße in der Navigations-Symboleiste sorgt:

```
style += "#nav-bar[ffzactive='true'] { font-size: "+fSize+"px; }";
```

Einen Sonderfall nehmen Mauszeiger und Icons ein, die nicht mit einer Chrome-Adresse eingebettet werden können. Diese Dateien werden in das Data-URI-Schema umgewandelt⁵ und

⁵Das Data-URI-Schema erlaubt es, beliebige Text- oder Binärdaten in Form einer URI auszudrücken.

in Variablen gespeichert. Zur Laufzeit werden sie abhängig von der Schriftgröße in die CSS-Regeln eingebunden.

Im folgenden Quellcode enthält die erste Variable die benötigte Größe eines Mauszeigers, abhängig von der aktuellen Schriftgröße. Bei Schriftgrößen 30 bis 39 wäre das die Zahl 30. Die zweite Variable enthält den Mauszeiger für diese Schriftgrößen. Die Anweisung erstellt mit Hilfe dieser Variablen eine CSS-Regel, die dafür sorgt, dass über allen Elementen der erwähnte Mauszeiger angezeigt wird:

```
var number = Math.round((fSize-5)/10)*10;
var arrow30 = „data:image/x-icon;base64,AAACAAEAHh4QAAAAAQDAAGAAAFgAAA-
CgAAAAeAAAAPAAAAAEABAAAAAAAAAWAIAAAAAAAAAAAAAAE...“
style += " * { cursor: url(„+eval(„arrow“+number+“;“)+“), auto;}“;
```

Nicht immer verhalten sich XUL-Elemente so, wie man es erwarten würde. Offensichtlich gibt es Elemente, die für hohe Schriftgrößen größer als ihre Elternknoten werden können. Das gilt für Menütitel der Menüleiste wie auch für Schaltflächen in der Toolbar der Druckvorschau. Diese Elemente müssen durch CSS-Regeln individuell angepasst werden um ein korrektes Verhalten zu erreichen. Ein korrektes Verhalten von sich aus ist deshalb für alle Elemente erwünscht, wenn man die Vergrößerung einer Leiste unabhängig von den enthaltenen Elementen implementieren möchte.

Eine weitere Schwierigkeit besteht darin, festzustellen, ob ein `<image>`-Element ein Bild enthält oder nicht. Buttons haben sehr häufig anonyme `<image>`-Elemente, die aber selten Bilder enthalten. Da die Vergrößerung der Bilder immer von einer CSS-Regel ausgelöst wird, die z.B. alle Bilder eines Dialogs vergrößert, werden auch die leeren `<image>`-Elemente vergrößert, was wieder nur durch das individuelle Ausschließen der Bilder von der Vergrößerung erreicht werden kann. Allgemein folgt daraus, dass viele Elemente an die dynamische Vergrößerung erst angepasst werden müssen, bevor sie problemlos verwendet werden können.

3.3.4 Tooltips

Firefox besitzt verschiedene Tooltips. Zum einen gibt es das Element `<tooltip>`, das im Regelfall das Element `<label>` als Kindknoten enthält, welches den Textinhalt des Tooltips aufnimmt. Das Erscheinungsbild dieses Tooltips lässt sich wie gewohnt mit CSS gestalten:

```
<toolbarbutton ... tooltip="tbb-tooltip">
<tooltip id="tbb-tooltip">
  <label value="Tooltip-Text" style="font-size: 30px;">
</tooltip>
```

Zum anderen existiert ein Tooltip, welcher von der Standard-Schriftgröße des Betriebssystems abhängt und mit CSS nicht verändert werden kann. Der Textinhalt dieses Tooltips wird in dem Attribut `tooltiptext` abgelegt:

```
<toolbarbutton ... tooltiptext="Tooltip-Text">
```

3.3.4 Tooltips

Das Konzept fordert, dass auch Tooltips vergrößert werden. Deshalb müssen Tooltips, die von der Schriftgröße des Betriebssystems abhängig sind, durch Tooltips ersetzt werden, die mit CSS gestylt werden können. Das betrifft den größten Teil der Tooltips in der Navigations-Symboleiste, allerdings nicht alle, da einige dynamisch erzeugt bzw. verändert werden. Beispielsweise kann sich die URL in der Adressleiste ändern. Wenn das passiert, wird das Attribut `tooltiptext` des Elements mit der neuen URL aktualisiert. Die Autovervollständigung der Adressleiste kann Tooltips mit einer URL oder einem Webseitentitel enthalten, wenn diese zu lang für das Pop-up sind. In solchen Fällen wird den Elementen das Attribut `tooltiptext` und damit ein Tooltip hinzugefügt, welches den vollständigen Titel bzw. die vollständige URL enthält.

Um auch auf dynamisch erstellte Tooltips reagieren zu können, wird eine Funktion benötigt, die erst dann den Tooltip ersetzt, wenn ein Element mit einem Attribut `tooltiptext` fokussiert wird. Die Funktion *OnPopup* wird mit jedem Tooltip aufgerufen. Enthält das fokussierte Element das Attribut `tooltiptext`, dann wird ein neues Attribut `tooltiptexthidden` mit dem Tooltip-Text als Wert erstellt. Das Attribut `tooltiptext` wird anschließend gelöscht. So wird verhindert, dass mehrere Tooltips gleichzeitig aufgerufen werden. Außerdem wird ein Attribut `tooltip` erstellt, das auf das `<tooltip>`-Element verweist, welches von nun an aufgerufen wird.

Eine Schaltfläche vor dem Aufruf der Funktion *OnPopup* und danach:

Davor:

```
<toolbarbutton ...   tooltiptext="Tooltip-Text">
```

Danach:

```
<toolbarbutton ...   tooltip="ffztip"
                    tooltiptexthidden="Tooltip-Text">
```

Der Tooltip, auf den das Attribut `tooltip` verweist, enthält das Attribut `onpopupshowing`, welches eine JavaScript-Anweisung enthält, die im Falle eines „`popupshowing`“-Ereignisses ausgeführt wird. Es wird der Tooltip-Text des fokussierten Elements in das `<label>`-Element des Tooltips geschrieben. Das bedeutet: Bei allen auf diese Weise modifizierten Elementen wird immer derselbe Tooltip aufgerufen, nur der Textinhalt wird bei jedem Aufruf aktualisiert:

```
<tooltip id="ffztip"
        onpopupshowing="this.firstChild.value = document
                        .tooltipNode.getAttribute('tooltiptexthidden');">
  <label value=""/>
</tooltip>
```

Leider ist es nicht möglich den neu erstellten Tooltip dann auch gleich automatisch anzeigen zu lassen. Deshalb wird dieser mit einer JavaScript-Anweisung aufgerufen. Beim nächsten Aufruf wird er dann automatisch angezeigt.

Das Ganze funktioniert mit allen Elementen, die ein `tooltiptext`-Attribut besitzen. Bei verschachtelten Elementen muss dasjenige Element dieses Attribut besitzen, welches beim „popup“-Ereignis die Quelle des Ereignisses ist. Sonst ist es nicht möglich auf das `tooltiptext`-Attribut zuzugreifen.

Einen Sonderfall stellen `<treechildren>`-Elemente dar. Diese Elemente enthalten den Inhalt eines hierarchischen Baums. Sie sind schwer zugänglich, weil sie im DOM-Baum als ein Element erscheinen. Es konnte nicht festgestellt werden wie man auf diese Tooltips zugreift, daher sind die Tooltips in solchen Elementen unverändert geblieben. Das trifft auf die Autovervollständigung der Suchleiste und auf die Sidebars zu. Um den Nachteil auszugleichen, wurde die begrenzte Breite der Sidebars aufgehoben. Außerdem wurde die Autovervollständigung der Suchleiste so modifiziert, dass die Breite des Pop-up mit der Schriftgröße ansteigt.

3.4 Navigation mit der Tastatur

Es gibt verschiedene Möglichkeiten, wie man mit der Tastatur durch die Schaltflächen von Werkzeuggestreifen navigieren kann:

- Tastenkombinationen – Darauf wird hier nicht näher eingegangen, weil das zu sehr an ein bestimmtes Ziel angepasst werden muss.
- Geradlinige Bewegung – Das Betätigen des Tabulators würde die Fokussierung von einem Element zum nächsten wechseln.
- Navigation mit Pfeiltasten – Durch jede Betätigung einer Pfeiltaste würde man dem Zielelement näher kommen.

Die Fokussierung von Schaltflächen mit dem Tabulator kann durch das Setzen der CSS-Eigenschaft `„-moz-user-focus“` auf den Wert `„normal“` ermöglicht werden. Was noch benötigt wird, ist eine Fähigkeit die Fokussierung zu erkennen und eine CSS-Regel oder JavaScript-Anweisung, die das entsprechende Element vergrößert. Um dem Konzept gerecht zu werden, müsste bei Fokussierung eines Elements die gesamte Leiste vergrößert werden. Auch die Fisheye-Methode könnte man anwenden, die dann für eine deutlichere Abhebung des fokussierten Elements sorgt. Der Vorteil dieser Methode ist, dass – im Gegensatz zur Navigation mit der Maus – nicht auf die Position des Mauszeigers geachtet werden muss.

Allerdings stellt sich die Frage, ob eine solche Navigation überhaupt Sinn macht. Die Navigations-Symbolleiste hat im Standardzustand acht Elemente. Geht man davon aus, dass die Tastatur auch zum Erreichen von anderen Werkzeuggestreifen verwendet werden soll, dann kann es vorkommen, dass der Benutzer zwei Dutzend mal den Tabulator betätigen muss bis er eine bestimmte Schaltfläche erreicht. Eine Navigation mit Pfeiltasten macht da mehr Sinn. Womit man bei einem Problem wäre, das auch die Kapitel 4, 5 und 6 haben (siehe Abschnitt 6.2).

3.5 Weitere Konzeptlösungen für Toolbars

Eine weitere Möglichkeit, wie Toolbars auch vergrößert werden können, ist die Fisheye-Methode, angewendet auf direkt übereinander liegende Leisten. Die Leiste, über welcher sich der Mauszeiger befindet, wird vollständig vergrößert. Je weiter vertikal eine andere Leiste von dem Mauszeiger entfernt liegt, desto weniger vergrößert wird diese. Dieses Verhalten hat den Vorteil, dass das Problem mit dem Wechseln der Leisten (siehe Abschnitt 3.2.2) verschwindet. Der Nutzen dieser Methode ist allerdings zweifelhaft. Abgesehen davon, dass auch Leisten vergrößert werden, die in dem Moment nicht interessant sind, hatte sich bei einer Implementierung dieser Variante gezeigt, dass die ständigen Berechnungen den Browser nur unnötig belasten. Außerdem ist bei schnellen Bewegungen ein Flackern unvermeidlich bzw. nur mit hohem Aufwand zu lösen.

Eine weitere Variante für die Vergrößerung von Toolbars ist die Methode, die auch bei der Lupeleiste verwendet wird (siehe Abschnitt 4.2.3.1). In diesem Fall würde sich die vergrößerte Leiste über die darunter liegende Leiste legen. Natürlich stellt sich dann die Frage, wie der Benutzer die Fokussierung von einer Leiste auf die darunter liegende Leiste wechselt, wenn diese von der vergrößerten Leiste überdeckt wird. Denkbar wären Lösungen wie das Drücken einer Taste, um die gerade vergrößerte Leiste zu deaktivieren, oder dass der „mouseover“-Effekt nur die Originalgröße der Leiste umfasst. Allerdings würde diese Lösung den Benutzer zu sehr verwirren. Auch bestimmte Gesten für die Aktivierung und Deaktivierung von Leisten sind möglich, z.B. das Ziehen der Toolbar nach unten zum Vergrößern und das Ziehen der Toolbar nach oben zum Verkleinern. Diese manuelle Vergrößerung von Elementen hat den Vorteil, dass das Ergebnis der eigenen Aktion dem Benutzer intuitiv verständlich ist, während bei einer automatischen Vergrößerung ein gewisser Überraschungseffekt auftreten kann. Allerdings dauert die manuelle Vergrößerung länger.

4 Vergrößerung von Webseitenelementen mit dem Lupenkonzept

4.1 Konzept

Webseitenelemente können nicht, wie in Kapitel 3, abhängig von der Position des Mauszeigers direkt vergrößert werden. Erstens würde eine direkte Vergrößerung zu verwirrenden Wechseln zwischen vergrößertem und unvergrößertem Zustand verschiedener Bereiche führen. Und zweitens ist es schwer semantisch zusammenhängende Elemente in Webseiten zu erkennen. Deshalb wurde für die Vergrößerung von Webseitenelementen das Lupenkonzept gewählt. Dabei wird das Auswählen eines Ausschnitts dem Benutzer überlassen. Das Element wird anschließend mit Hilfe eines zusätzlichen Objekts, der Lupe – das ist ein Fenster, das im Webbrowser eingeblendet wird – indirekt vergrößert. Das Layout der Webseite wird dabei nicht beeinflusst, die Webseitenelemente bleiben unverändert, lediglich in der Lupe wird das fokussierte Element vergrößert dargestellt. Allerdings ist es möglich, dass die Lupe einen Teil der Webseite überdeckt, der sonst sichtbar wäre. Da aber die meisten Webseiten ausreichend freien Platz für die Positionierung von Lupen zur Verfügung haben, ist das kein wesentlicher Nachteil. Viel unvorteilhafter ist die Tatsache, dass der Benutzer immer wieder gezwungen wird seine Fokussierung zwischen der Originalansicht und der Lupe zu wechseln.

Die im vorigen Kapitel erläuterten Konzeptanforderungen können z.T. auch auf dieses Kapitel übertragen werden. Beispielsweise müssen vergrößerte Elemente in der Lupe über dieselben Funktionalitäten verfügen wie die Elemente in der Originalansicht. Das Anklicken eines Links muss eine neue Seite öffnen, und auch das Ausfüllen und Abschicken eines Formulars muss so funktionieren, wie der Benutzer es erwarten würde.

Neue Lupen sollen über eine Tastenkombination oder per Doppelklick auf einen Bereich einer Webseite geöffnet werden können. Der rechte Rand einer Webseite eignet sich am Besten für die Positionierung einer Lupe, da dieser in vielen Webseiten nicht genutzt wird. Danach sollte der Benutzer die Position der Lupe verändern können. Im Allgemeinen sollte die Lupe über dieselben Funktionen verfügen wie jedes normale Fenster: Minimieren, Maximieren, Größe ändern, Verschieben und Schließen. Das Verhalten der Lupen - ob sie dem Benutzer beim Surfen folgen oder in bestimmten Situationen geschlossen werden sollen - sollte einstellbar sein. Um das vergrößerte Element in der Originalansicht schnell wiederfinden zu können, sollten das Element in der Originalansicht und die Lupe, die das Element vergrößert darstellt, gemeinsame Merkmale besitzen. Und da der Benutzer mehrere Lupen gleichzeitig öffnen könnte, um verschiedene Bereiche der Webseite zu vergrößern, muss man die Lupen unterscheiden können.

4.2 Umsetzung

4.2.1 Aufbau der Lupe

Der sichtbare Teil der Lupe ist zweigeteilt. Wie bei allen Bildschirmfenstern befindet sich über dem Dokumentenbereich eine Leiste (im folgenden als Lupenleiste bezeichnet). Es handelt sich um eine Symbolleiste mit Schaltflächen für die wichtigsten und häufigsten Lupenfunktionen. Darunter befindet sich der Browserbereich der Lupe (im folgenden als Lupenbrowser bezeichnet), in welchem der fokussierte Webseitenausschnitt vergrößert dargestellt wird (Abbildung 8).

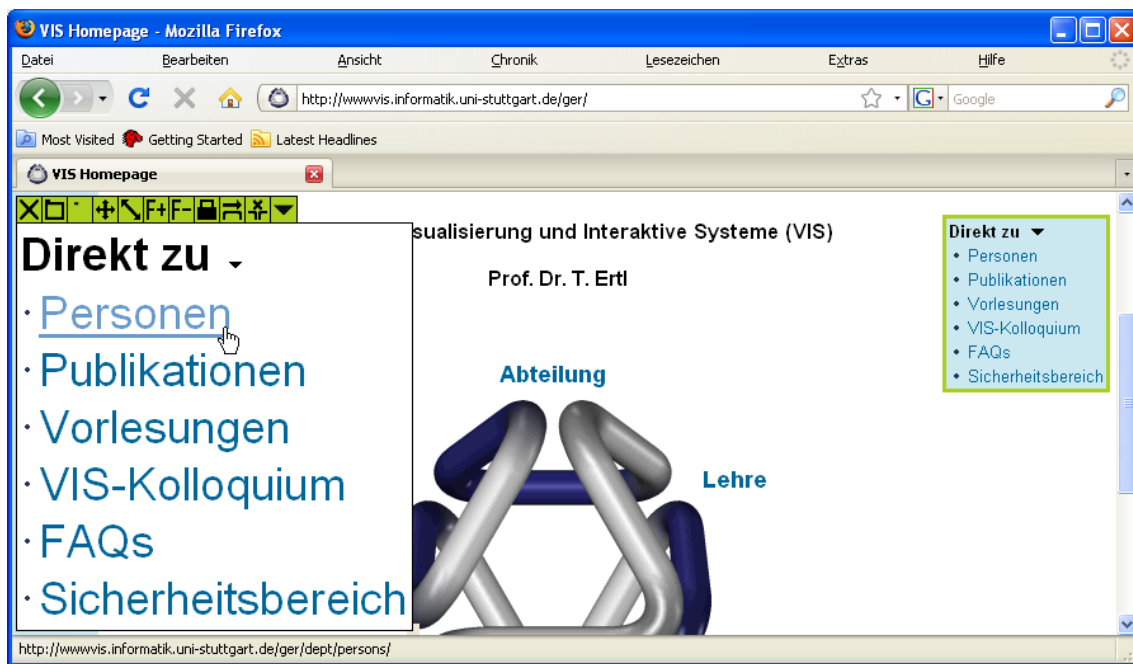


Abbildung 8: Lupe im Einsatz

Die mit XUL erstellte Benutzeroberfläche der Lupe wird bei Firefox-Start aus der Datei „overlay.xul“, in der sie als Quellcode vorliegt, in das Hauptfenster geladen. Die dabei geladene Lupe dient aber nur als Kopiervorlage für weitere Lupen und ist aufgrund des auf „true“ gesetzten Attributs `hidden` unsichtbar. Soll jedoch eine neue Lupe geöffnet werden, wird die Kopiervorlage dupliziert, die Kopie auf sichtbar gesetzt und dem DOM-Baum des Hauptfensters hinzugefügt. Der ganze Ablauf wird im Abschnitt 4.2.2.1 näher beschrieben.

Bevor es um die innere Struktur der Lupe geht, muss noch eine Besonderheit erläutert werden. Es ist ohne Weiteres nicht möglich, innerhalb eines Firefox-Fensters über einem Webdokument XUL-Elemente zu platzieren. Diese können zwar dem Hauptfenster hinzugefügt werden, sie würden sich dann aber eine Ebene unter dem Webdokument befinden und wären unsichtbar. Für solche Zwecke stellt XUL das Element `<panel>` zur Verfügung, das jede Art von Inhalt aufnehmen kann. Allerdings kommt es bei Maus-Interaktionen mit der Webseite zu

Problemen wenn das `<panel>` ein `<browser>`-Element enthält [MDC08], welches hier unentbehrlich ist. Es gibt aber eine weitere Methode, die von der Firefox-Erweiterung *CoolPreviews* übernommen wurde [COP08]. Nimmt man das Stapel-Element `<stack>` und platziert als unteres Element des Stapels ein `<browser>`-Element und eine Ebene darüber das Element, das dargestellt werden soll, dann verhält sich alles korrekt. Im folgenden vereinfachten Quellcode der Lupe befinden sich vier Stacks mit je einem unsichtbaren `<browser>` und je einem sichtbaren zweiten Element. Diese sind die Lupenelemente:

```
<box>
  <stack>
    <stack>
      <browser/>
      <image/> <!-- Anfasser -->
    </stack>
    <stack top="21">
      <browser/>
      <browser/> <!-- Lupenbrowser -->
    </stack>
    <stack top="21">
      <browser/>
      <colorpicker/> <!-- Farbwähler -->
    </stack>
    <stack top="0">
      <browser/>
      <hbox> <!-- Lupenleiste -->
        <image/> <!-- Erste Schaltfläche -->
        ...
        <image/>
      </hbox>
    </stack>
  </stack>
</box>
```

Das äußerste Stack ist dafür da, um die vier Lupenelemente versetzt stapeln zu können. Zunächst sieht es sicherlich so aus, als hätte man hier unnötige Elemente. Würde man aber eines davon weglassen, könnte das Ganze nicht mehr so funktionieren, wie man es möchte. Es stellt sich vielleicht auch die Frage, wieso man nicht bloß ein `<browser>`-Element nimmt und alle Lupenelemente oberhalb dieses platziert. Das würde zweifellos funktionieren, allerdings hätte man dann graue Flächen im Box-Element, wo leerer Bereich ist, wie z.B rechts neben der Lupenleiste.

Das erste Stack enthält den Anfasser⁶, welcher immer in der rechten unteren Ecke des Lupenbrowsers zu finden ist. Mit diesem Element lässt sich die Größe der Lupe verändern. Eine Ebene darüber liegt das Stack mit dem Lupenbrowser. Dieses ist etwas nach unten verschoben, da oberhalb des Lupenbrowsers die Lupenleiste liegt. Die Entfernung der oberen Kante eines Elements zum oberen Rand der Lupe ist in dem Attribut `top` im jeweiligen Stack zu finden. Das dritte Stack enthält das Element `<colorpicker>`, mit welchem man eine Farbe auswählen kann. Es liegt ebenfalls unterhalb der Lupenleiste, an derselben Stelle wie der

⁶Element zur Manipulation der Größe eines Fensters

4.2.1 Aufbau der Lupe

Browser, nur eine Ebene darüber. Wenn es eingeblendet wird, dann überdeckt es einen Teil des Lupenbrowsers. Das letzte Stack ist die Lupenleiste mit horizontal angeordneten `<image>`-Elementen, die für Schaltflächen stehen.

Zu der Lupe gehört außerdem ein Pop-up-Menü, welches über eine Schaltfläche in der Lupenleiste aufgerufen wird und schnellen Zugriff auf selten benötigte Funktionen bietet. Das dazugehörige `<menupopup>`-Element ist nicht Teilbaum der Lupen-Box. Es wird, wie die Lupe selbst, durch das Laden des Overlays ins Hauptfenster geladen. Im Gegensatz zu den Lupen, kommt es aber nur einmal im Hauptfenster vor. Sind mehrere Lupen eingeblendet, dann rufen alle dasselbe `<menupopup>`-Element auf. Da das Pop-up-Menü aber Eigenschaften der Lupe darstellt, die nur für die jeweilige Lupe gelten, sorgt vor dem Erscheinen des Menüs ein „popupshowing“-Ereignis dafür, dass eine Funktion aufgerufen wird, die die Menüeinträge des Menüs an die jeweilige Lupe anpasst.

4.2.2 Abläufe

4.2.2.1 Öffnen einer neuen Lupe

Soll eine Lupe geöffnet werden, wird zunächst eine neue Instanz des Objekts *lensObj* erzeugt. Darin befinden sich Referenzen auf wichtige Elemente der Lupe, wie das `<box>`-Element, die veränderlichen Schaltflächen in der Lupenleiste und das `<browser>`-Element. Somit kann auf diese Lupenelemente schnell und einfach zugegriffen werden. Außerdem speichert das Objekt wichtige Eigenschaften der Lupe, die zu Beginn auf Standardwerte gesetzt werden, und enthält Methoden um diese zu beeinflussen.

Als nächstes wird die Funktion *RandomColor* aufgerufen, die eine zufällige Farbe für die Lupenleiste generiert. So wird sichergestellt, dass Lupen unterscheidbar sind. Zu diesem Zeitpunkt befindet sich die Lupe noch nicht im Hauptfenster sondern existiert nur in einer Variable in dem neu instanziierten Objekt. Zunächst müssen Position und Größe der Lupe berechnet werden. Es wird davon ausgegangen, dass auf der rechten Seite im Browserbereich freier Platz ist und die Lupe die gesamte Höhe dieses Bereichs einnehmen soll. Die Höhe wird aus dem Wert der Eigenschaft `innerHeight` des *window*-Objekts berechnet, welche die Höhe des Hauptfensters enthält. Die Breite wird auf 300 Pixel gesetzt. Aus dem Wert der Eigenschaft `innerWidth` berechnet man die Position der Lupe auf der x-Achse. Die Position auf der y-Achse entspricht der des Browserbereichs. Anschließend werden die CSS-Eigenschaften „top“ und „left“ des `<box>`-Elements und die Eigenschaft „height“ des `<browser>`-Elements auf die ermittelten Werte gesetzt. Die Lupe wird absolut und außerhalb des normalen Textflusses positioniert, was mit der CSS-Deklaration „position:fixed“ erreicht wird. Das Layout der Webseite bleibt unverändert, zusätzlich bleibt die Lupe beim Scrollen an derselben Stelle. Zu guter Letzt wird die fertige Lupe dem DOM-Baum des Hauptfensters hinzugefügt.

Um eine bestimmte Lupe schnell finden zu können, werden die Objekte aller aktiven Lupen in einem Array gespeichert. Der Index eines Array-Elements ist die ID der jeweiligen Lupe, die im `<box>`-Element der Lupe im Attribut `id` gespeichert ist. Die ID besteht aus dem Wort „lens“ und einer fortlaufenden Nummer. Außerdem wird bei einigen Elementen, wie den Schaltflächen, von welchen ein schneller Zugriff auf das Objekt wichtig ist, die ID in dem Attribut `hiddenId` abgelegt. Mit Hilfe des Arrays kann man alle Lupen durchlaufen oder anhand der ID gezielt auf eine bestimmte zugreifen.

4.2.2.2 Laden eines Webseitenelements in die Lupe

Befindet sich eine geöffnete Lupe im Hauptfenster, dann kann darin ein Webseitenausschnitt vergrößert dargestellt werden. Das Auswählen eines Elements erfolgt durch das Bewegen des Mauszeigers über den entsprechenden Bereich. Das fokussierte Element erkennt man daran, dass dieses eine gestrichelte Umrandung in derselben Farbe erhält wie die Lupenleiste. Die Farbe soll es dem Benutzer leichter machen das vergrößerte Element einer Lupe im Dokument wiederzufinden. Verweilt der Benutzer eine Sekunde über demselben Bereich, bekommt dieses eine durchgezogene Linie als Umrandung und wird in die Lupe geladen. Der Ablauf wird im Folgenden genauer erklärt.

Der Ladevorgang einer Webseite im Hauptfenster löst ein „DOMContentLoaded“-Ereignis aus, was dazu führt, dass allen HTML-Elementen einer Webseite Event-Listeners zugewiesen werden. Öffnet der Benutzer später eine Lupe und bewegt den Mauszeiger über ein Element, wird ein „mouseover“-Ereignis ausgelöst, infolgedessen die Funktion *OnMouseOver* aufgerufen wird. Diese überprüft, ob das fokussierte Element in die Lupe geladen werden darf. Ein Element darf nicht in die Lupe geladen werden, wenn es dort bereits vergrößert dargestellt wird. Spricht nichts dagegen, bekommt das Element eine gestrichelte Umrandung und mit einer Verzögerung von einer Sekunde wird die Funktion *SetTimer* aufgerufen. Diese Funktion vergleicht die URL der aktuellen Webseite mit der aktuellen URL in der Lupe. Sind die URLs verschieden, wird zuerst die neue Webseite vollständig in der Lupe geladen. Auf diese Weise gelangt das Stylesheet des Dokuments in den Lupenbrowser. Das Stylesheet sorgt dafür, dass die Darstellung der vergrößerten Elemente der Originalansicht entspricht. Anschließend wird die Funktion *LoadIntoLens* aufgerufen, die veraltete Elemente aus dem Lupenbrowser entfernt und das neue Element dort einfügt. Hat sich die URL nicht geändert, dann bleibt das Stylesheet gleich und die Funktion *LoadIntoLens* wird sofort aufgerufen.

Für die Vergrößerung der Darstellung sorgt eine Firefox-eigene Zoom-Funktion, die durch das Setzen der Eigenschaft `textZoom` auf den Vergrößerungsfaktor aufgerufen wird. Dieser wird aus der Standardschriftgröße und der bevorzugten Schriftgröße berechnet.

4.2.2.3 Synchronisation

4.2.2.3 Synchronisation

Tastatureingaben, die der Benutzer in der Lupe macht, müssen erhalten bleiben wenn der Fokus auf ein anderes Element wechselt. Um das sicherzustellen, wird eine Funktion benötigt, die für jede Tastatureingabe und für jeden Mausklick in der Lupe eine identische Aktion im Hauptfenster hervorruft. Natürlich muss die Synchronisation auch in die andere Richtung funktionieren.

Jedem Element, das in die Lupe eingefügt wird, werden zwei Event-Listeners zugewiesen. Diese „hören“ auf einfache Mausklicks und Tastatureingaben. Dieselben Event-Listeners werden auch dem Originalelement im Hauptfenster zugewiesen. Gibt nun der Benutzer etwas in ein Formularfeld ein, wird dieses Element zusammen mit der Synchronisationsrichtung an die Funktion *PingPong* übergeben. Der Name der Funktion spiegelt ihren Zweck wider. Die Funktion leitet Tastatureingaben und Mausklicks vom Hauptfenster in die Lupe oder von der Lupe ins Hauptfenster, je nachdem wo die Aktionen gemacht wurden. Diese Funktion kann also in beide Richtungen synchronisieren. Als nächstes wird das übergebene Element im Zielfenster gesucht. Das geschieht indem die Attribute von jedem Element, das vom selben Typ ist wie das übergebene Element, mit den Attributen des übergebenen Elements verglichen werden. Sobald das Element gefunden ist, wird bei Tastatureingaben der Wert des Attributs `value` in das Zielelement kopiert. Bei Mausklicks wird an dem gefundenen Element die Funktion *click* ausgeführt, welche ein Mausklick simuliert. Bei Formularelementen wird die Funktion *submit* ausgeführt.

Einen besonderen Fall stellen Links dar. Klickt der Benutzer auf einen Link in der Lupe, dann öffnet sich dort eine neue Webseite. Dieses Verhalten ist aber nicht erwünscht. Die neue Webseite sollte sich im Hauptfenster öffnen, die Lupe sollte unverändert bleiben. Das wird erreicht, indem das Standardverhalten aller Links in der Lupe mit dem Setzen des `onclick`-Attributs auf den Wert „return false;“ unterdrückt wird. Dieses bewirkt, dass ein Klick auf einen Link in der Lupe keine Auswirkungen hat. Auch das Standardverhalten von Formularen wird unterdrückt indem hier das `onsubmit`-Attribut auf den Wert „return false;“ gesetzt wird.

Tastatureingaben und Mausklicks können also von einem Browser-Element an ein anderes weitergeleitet werden. Dabei bleiben die Lupen erhalten. Hätte man die Lupen mit HTML umgesetzt, würden sie nach jedem Klick auf einen Link verschwinden und müssten dem Dokument jedes Mal neu hinzugefügt werden.

4.2.3 Funktionen

4.2.3.1 Vergrößerung der Lupenleiste

Die Vergrößerung der Lupenleiste funktioniert auf den ersten Blick wie bei Firefox-Leisten. Bewegt man den Mauszeiger über die Lupenleiste, wird das fokussierte Element, also eine

4.2.3.1 Vergrößerung der Lupenleiste

Schaltfläche, in Breite und Höhe vergrößert, während die restlichen Elemente in der Reihe auf die Höhe des vergrößerten Elements gestreckt werden. Bewegt man nun den Mauszeiger auf ein benachbartes Element, wird das vorherige Element auf die ursprüngliche Breite verkleinert, während das neu fokussierte Element vollständig vergrößert wird. Der Unterschied zu der Vergrößerung der Firefox-Leisten liegt darin, dass der Lupenbrowser, der sich unterhalb der Lupenleiste befindet, bei Vergrößerung nicht nach unten verschoben wird.

Es stehen verschiedene Konzeptlösungen für die Vergrößerung einer Toolbar zur Auswahl und sie haben alle Vor- und Nachteile; wie im vorigen Kapitel gezeigt wurde. Der Lupenbrowser sollte - unabhängig davon, ob die Lupenleiste vergrößert ist oder nicht - immer an derselben Stelle bleiben. Das würde dem Benutzer erlauben den Mauszeiger von der Lupenleiste zu einem HTML-Element im Lupenbrowser zu bewegen ohne ein Springen des Elements zu verursachen, was ein Nachziehen des Mauszeigers zur Folge hätte. Das bedeutet, die Lupenleiste darf sich bei Vergrößerung nicht nach unten ausdehnen, wie das bei Firefox-Leisten der Fall ist. Eine Ausdehnung der Lupenleiste nach oben ist aber ebenfalls nicht erwünscht. Die Lupe sollte in ihren äußeren Maßen konstant bleiben, denn bei Verschieben der Lupe ist die Vergrößerung aktiv. Das kann zu Irritationen führen wenn die Aktion beendet wird, da sich dann mit der Verkleinerung der Lupenleiste die Position dieser oder die des Lupenbrowsers geringfügig ändert, während man beim Verschieben mit einer anderen Endposition der Lupe rechnet.

Durch das Trennen der beiden Elemente Lupenleiste und Lupenbrowser voneinander, wie in Abschnitt 4.2.1 beschrieben, kann die Lupenleiste über dem Lupenbrowser positioniert werden, sodass die Vergrößerung der Lupenleiste weder eine Verschiebung des Lupenbrowsers noch eine Änderung der Größe der Lupe zur Folge hat. Die Lupenleiste überdeckt lediglich einen Teil des Lupenbrowsers (Abbildung 9). Das hat nur minimale Nachteile, da der Benutzer in dem Moment nur an der Leiste interessiert ist.

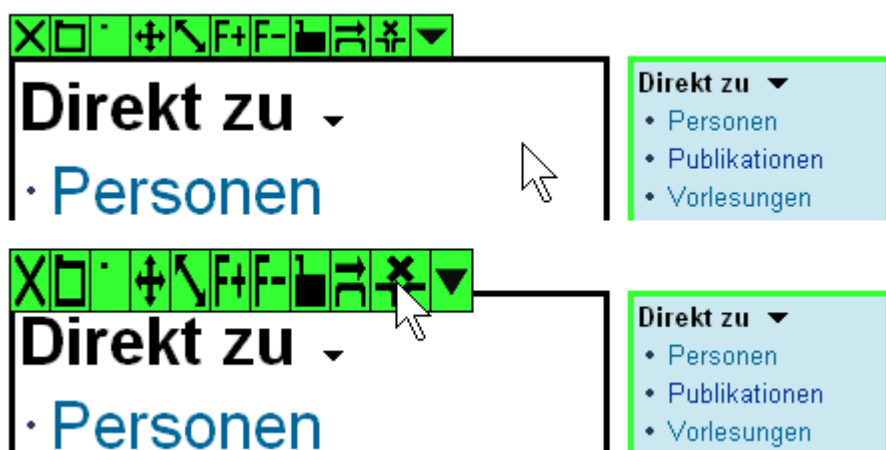


Abbildung 9: Lupenleiste unvergrößert und vergrößert

4.2.3.1 Vergrößerung der Lupenleiste

Die Lupenleiste muss sich also in der Stack-Hierarchie immer oberhalb des Lupenbrowsers befinden. Der Aufbau der Lupe sorgt zwar dafür, dass die Reihenfolge nach dem Öffnen einer Lupe korrekt ist. Leider kann diese Reihenfolge durcheinander kommen, denn zuletzt eingeblendete Webdokumente erscheinen immer oberhalb aller anderen Elemente. Blendet man z.B. das `<stack>`-Element des Lupenbrowsers kurz aus und gleich wieder ein, würde sich das Webdokument in der Stack-Hierarchie oberhalb der Lupenleiste befinden, was man allerdings erst merkt wenn man mit dem Mauszeiger auf die Lupenleiste geht. Dem Dokument einen höheren „z-Index“-Wert zuzuweisen würde daran nichts ändern. Damit das also nicht passiert, wird in solchen Fällen die gesamte Lupe für kurze Zeit ausgeblendet. Ein nicht gewünschter Effekt des Ganzen ist ein Flackern. Dieser Vorgang wird jedes Mal durchgeführt wenn eine Lupe aus dem minimierten Zustand zurückkehrt, wenn ein Lupe fokussiert wird oder wenn der Benutzer einen Tabwechsel macht.

4.2.3.2 Funktionen der Lupenleiste

Alle Bilder von Schaltflächen in der Lupenleiste sollen intuitiv verständliche Beschreibungen ihrer Funktionen darstellen. Die erste Schaltfläche entfernt die Lupe aus dem Hauptfenster. Tatsächlich wird das `<box>`-Element der Lupe im DOM-Baum gelöscht. Außerdem wird das JavaScript-Objekt, in dem das `<box>`-Element als Referenz gespeichert war, gelöscht und die Umrandung des fokussierten Elements entfernt. Ein Fehler, der nicht vollständig gelöst werden konnte, entsteht, wenn man eine Lupe schließt und an derselben Stelle eine neue öffnet. Offenbar wird das „mouseover“-Ereignis nicht beendet und wirkt auf die neue Lupe, obwohl der Mauszeiger evtl. an einer ganz anderen Stelle ist. Dieses Problem wird umgangen, indem eine neue Lupe um die Höhe der Lupenleiste zur letzten Lupe versetzt geöffnet werden.

Das Betätigen der zweiten Schaltfläche vergrößert die Lupe auf das Maximum. Die Lupenleiste erscheint also in der linken oberen Ecke und der Lupenbrowser nimmt den Rest des Browserbereichs ein, lediglich Platz für die beiden Bildlaufleisten rechts und unten wird übrig gelassen. Vor dem Vergrößern werden die Position und Größe der Lupe gespeichert. Anschließend werden die Position und Größe der Lupe im maximierten Zustand berechnet. Das Zuweisen der Positionswerte an das `<box>`-Element sowie der Größenwerte an das `<browser>`-Element geschieht durch das Setzen der CSS-Eigenschaften „top“ und „left“ bzw. „height“ und „width“. Ein nochmaliges Betätigen der Schaltfläche stellt den früheren Zustand wieder her. Dabei werden die zuvor gespeicherten Werte wieder übernommen.

Das Minimieren der Lupe ist die Funktion der dritten Schaltfläche. Dabei wird der Lupenbrowser durch das Setzen des Attributs `hidden` ausgeblendet. Entsprechend kann der Lupenbrowser wieder zum Vorschein gebracht werden. Da sich die Funktion der Schaltfläche ändert, müssen auch das Bild und der Tooltip-Text geändert werden. Das Bild wird geändert, indem die CSS-Eigenschaft „list-style-image“ auf die URL eines anderen Bilds gesetzt wird. Der Tooltip-Text befindet sich im Attribut `tooltiptexthidden`, der neue Wert wird aus der

lokalen Java-Properties-Datei geholt. Wie Tooltips funktionieren, wurde bereits beschrieben (siehe Abschnitt 3.3.4).

Mit Hilfe der nächsten Schaltfläche lässt sich die Lupe an eine beliebige Stelle innerhalb des Hauptfensters verschieben. Der Mauszeiger ändert sich über dieser Schaltfläche zu einem „Kreuz“. Das soll dem Benutzer die Funktion der Schaltfläche aufzeigen. Klickt man nun darauf, wird die Verschiebe-Funktion in Gang gesetzt. Die Funktion selbst steht hier nicht im Mittelpunkt, deshalb wird hier nicht näher darauf eingegangen. Erwähnenswert ist höchstens die Tatsache, dass die Lupe in ihren Maßen konstant bleibt. Man stelle sich vor, wie es wäre, wenn die Lupe nach dem Verschieben plötzlich ihre Größe ändern würde.

Die Änderung der Fenstergröße erfolgt im Regelfall durch das Ziehen des Anfassers auf die gewünschte Größe. Das ist auch bei der Lupe möglich. Die fünfte Schaltfläche kann ebenfalls dazu genutzt werden die Größe der Lupe zu verändern. Die Position der rechten unteren Ecke bleibt stabil, während die linke obere Ecke in ihrer Position frei verändert wird. Wieso diese Art der Lösung? Wenn man bedenkt, dass die Benutzer dieser Software Menschen mit einer Sehschwäche sind, dann fällt auf, dass die Ecke eines Fensters mit dem Mauszeiger schwerer einzufangen ist, als eine Schaltfläche, die eine ausreichende Größe hat. Sicher könnte man dem Problem mit einem größeren Anfasser entgegenwirken. Es gibt aber einen weiteren Punkt. Die Mauswege verkürzen sich mit dieser Methode. Wie zuvor wird auch hier nicht auf die Funktion selbst eingegangen.

Mit den folgenden beiden Schaltflächen kann der Textinhalt der Lupe vergrößert oder verkleinert werden. Dazu übergibt man einen Vergrößerungsfaktor an eine Eigenschaft des Browsers:

```
this.browser.markupDocumentViewer.textZoom = this.factor;
```

Für die Vergrößerung sorgt Firefox. Diese als *textZoom* bekannte Funktion kann u.a. mit den Tastenkombinationen [Ctrl][+] bzw. [Ctrl][-] genutzt werden.

Die nächste Schaltfläche zeigt ein Schloss, das offen oder geschlossen ist. Ist das Schloss offen, dann kann ein neues Webseitenelement in die Lupe geladen werden. Ist es geschlossen, ist das nicht möglich. Die Funktion soll das versehentliche Laden eines Webausschnitts in die Lupe verhindern. Der Ablauf dahinter ist trivial. Eine Variable im Lupen-Objekt speichert den Zustand. Ist der Zustand „geschlossen“, dann kann kein neues Element geladen werden. Da mehrere Lupen gleichzeitig geöffnet sein können, darf es maximal eine Lupe geben, die ein Element laden kann. Deswegen werden alle Lupen auf „geschlossen“ geschaltet, wenn eine neue Lupe geöffnet wird.

Die nächsten beiden Schaltflächen sind in ihren Funktionen ähnlich. Mit der ersten legt man fest, ob die Lupe dem Benutzer folgen soll, wenn dieser auf einen Link klickt. Mit der zweiten legt man fest, ob die Lupe bei Tabwechsel mitgehen soll. Beide Funktionen setzen - wie schon bei der letzten Funktion - nur eine Variable in einen bestimmten Zustand und ändern sonst nur

4.2.3.2 Funktionen der Lupenleiste

das Bild der Schaltfläche und den Tooltip-Text. Die eigentlichen Anweisungen werden abgearbeitet wenn der Benutzer auf einen Link klickt oder den Tab wechselt: Jedes Mal wenn eine Webseite fertig geladen wird, wird ein „DOMContentLoaded“-Ereignis ausgelöst, infolgedessen alle Lupen in dem aktuellen Tab danach geprüft werden, ob sie Links folgen dürfen oder nicht. Diejenigen, die ihnen nicht folgen dürfen, werden gelöscht. Ähnlich läuft es bei einem Tabwechsel ab. Das Ereignis „TabSelect“ wird ausgelöst, infolgedessen alle Lupen entfernt werden, die bei einem Tabwechsel geschlossen werden müssen.

4.2.3.3 Pop-up-Menü der Lupenleiste

Die letzte Schaltfläche in der Lupenleiste ruft ein Pop-up-Menü auf, das vier Einträge enthält. Der erste Eintrag zeigt an, ob alle Textinhalte in der Lupe in einer Schriftgröße dargestellt werden. Ein wesentlicher Punkt des Konzepts ist eine einheitliche Schriftgröße bei Vergrößerung. Diese Einstellung kann jedoch durch einen Klick auf diesen Eintrag geändert werden. Die Textinhalte sehen dann aus wie im Original, nur um einen Vergrößerungsfaktor erhöht. Für Veränderungen am Webdokument in der Lupe wird in den <head>-Bereich ein Stylesheet eingefügt. Dieses Stylesheet kann auch für weitere Änderungen verwendet werden, wie für Schrift- und Hintergrundfarben. Möglicherweise bevorzugt der Benutzer helle Zeichen über einem dunklen Hintergrund [CON99]:

```
* {  
  font-size: 12px !important;  
  background-color: black !important;  
  color: yellow !important;  
}
```

Der nächste Eintrag im Menü zeigt an, ob Bilder im Webdokument vergrößert werden. Auch hier kann durch einen Klick auf den Eintrag die Einstellung geändert werden. Dann wird eine Funktion aufgerufen, die alle Bilder durchgeht und diese um den Vergrößerungsfaktor vergrößert bzw. verkleinert. Ändert man den Zustand, dann gilt dieser nur für die jeweilige Lupe und geht verloren, wenn man diese schließt. Möchte man Einstellungen dauerhaft ändern, dann macht man das in den Optionen. Der letzte Eintrag im Menü ist ein Schnellzugriff auf die Optionen. Dabei wird gleich der entsprechende Tab für die Lupeneinstellungen geöffnet.

Durch einen Klick auf den dritten Eintrag im Menü ruft man den Farbwähler auf. Damit lässt sich die Farbe der Lupenleiste verändern, wenn z.B. zwei Lupen ähnliche Farben haben. Das Element wird über dem Lupenbrowser eingeblendet. Es stellt alle wichtigen Farben zur Auswahl zur Verfügung. Wird eine Farbe ausgewählt, blendet das <colorpicker>-Element wieder aus und die Farben der Lupenleiste und der Umrandung im Hauptfenster werden auf den neuen Wert gesetzt.

4.2.3.4 Öffnen einer Lupe

Es ist möglich eine Lupe mit einem vorab ausgewählten Element zu öffnen. Man gehe folgendermaßen vor: Halte [Ctrl] gedrückt und wähle mit Hilfe der Umrandungen ein Element aus.

Ein Doppelklick auf einen freien Bereich innerhalb dieses Elements öffnet eine neue Lupe mit dem gewählten Element. Hier werden nur vorhandene Funktionen genutzt: Erstelle ein neues Lupe-Objekt und lade das gewählte Element in die Lupe. Damit nicht jeder Doppelklick zu einer neuen Lupe führt, funktioniert das Ganze nur, wenn [Ctrl] gedrückt ist.

4.3 Alternative Umsetzung - HTML statt XUL

Eine Alternative zur Umsetzung der Lupe mit XUL wäre eine Umsetzung mit HTML gewesen. Die erste Umsetzung, die als Quelle und Anregung dieser Arbeit diente und von Dipl. Inf. Christiane Taras erstellt wurde, hatte eine solche Lösung. Beide Methoden haben Vor- und Nachteile. Die XUL-Methode hat z.B die Schriftvergrößerung schon integriert (TextZoom), während bei der HTML-Methode eine Vergrößerungsfunktion hätte entwickelt werden müssen. Leider konnte eine weitere Funktion von Firefox (FullZoom), die Vergrößerung von Bildern unterstützt, nicht genutzt werden. Diese verursachte einen Fehler, da bei FullZoom in einer Lupe auch die XUL-Elemente vergrößert wurden. Ein weiterer Vorteil von XUL ist, dass die Lupen nicht verschwinden, wenn der Benutzer auf einen Link klickt. Bei der HTML-Methode müssen Lupen, die erhalten bleiben sollen, in diesem Fall neu eingefügt werden.

Auf der anderen Seite müssen Lupen bei der XUL-Methode kurz ausgeblendet werden, wenn der Benutzer einen Tabwechsel macht. Ein weiterer Punkt für die HTML-Methode ist, dass sich alle Elemente – sowohl das Originalelement als auch die Lupe mit dem vergrößerten Element – im selben Dokument befinden. Dadurch kann auch auf Elemente zugegriffen werden kann, die in einem anderen Teilbaum als das fokussierte Element liegen. Bei der XUL-Methode ist das nicht möglich, weil nur das fokussierte Element in den Lupenbrowser kopiert wird. Alle Elemente, die mit diesem assoziiert sind, aber in einem anderen Teilbaum liegen, können wie z.B. im Fall eines Pop-up-Menüs nicht angezeigt werden. Auch Stylesheets liegen gleich alle vor. Bei der XUL-Methode muss zuvor die ganze Webseite im Lupenbrowser geladen werden. Dann wird der Inhalt gelöscht und das fokussierte Element hinein kopiert. Nur auf diese Weise kann das Aussehen des vergrößerten Elements in der Lupe dem Original angepasst werden. Dieser Vorgang dauert natürlich deutlich länger als bei der HTML-Methode.

5 Dialoge und Fenster

Bevor ein Benutzer mit Sehbehinderung auf das Erscheinen eines Fensters reagieren kann, muss das Fenster vergrößert werden. Wir unterscheiden zwischen zwei Fensterarten: Dialogfenster werden meist vom Betriebssystem oder einer laufenden Anwendung eingeblendet um bestimmte Eingaben oder Bestätigungen vom Benutzer einzuholen, z.B. ein Warnhinweis oder eine Aufforderung zur Passworteingabe. Sie sind meist von kleiner Größe, einfach aufgebaut und enthalten nur die gängigsten Elemente wie Textfelder, Bilder und Checkboxen [DIA09]. Die andere Kategorie umfasst alle Fenster, die der Benutzer aufruft, etwa ein Fenster für die Einstellungen einer Anwendung. Diese haben einen komplexeren Aufbau, sind meist größer als Dialoge und können alle Typen von Elementen enthalten.

5.1 Konzept

Dialogfenster sollen gleich bei Erscheinen vollständig vergrößert werden. Dieses Vorgehen entspricht dem Konzept, bei Fokussieren eines Elements nicht nur dieses, sondern auch alle logisch eng zusammenhängenden Elemente zu vergrößern, denn gerade bei Dialogen hängen alle Elemente semantisch stark zusammen. Ein weiterer Punkt für vollständiges Vergrößern ist die Tatsache, dass Dialoge auch bei hohen Schriftgrößen noch ausreichend Platz für die Darstellung auf dem Bildschirm haben. Und ein Punkt für sofortiges Vergrößern ist die Tatsache, dass der Benutzer bei Erscheinen eines Dialogs ein Interesse an diesem hat, umso mehr wenn das Dialogfenster modal ist, also bei Erscheinen den Benutzer daran hindert in anderen Fenstern derselben Anwendung zu arbeiten. Deswegen muss auch verhindert werden, dass ein Dialogfenster nach Vergrößerung außerhalb des Sichtbereichs des Benutzers erscheint und damit Anwendungen blockiert. Hat der Dialog nicht genug Platz auf dem Bildschirm, sollte er im Originalzustand bleiben und später, aufgrund seiner Struktur, in Teilen vergrößert werden.

5.2 Erweiterung des Konzepts

Das oben beschriebene Konzept sollte nicht nur auf Dialoge beschränkt sein, sondern auf alle Fenstertypen angewendet werden. Schließlich möchte der Benutzer auch gewöhnliche Fenster vergrößern können, z.B. eines für die Einstellungen der laufenden Anwendung. Außerdem kann es vorkommen, dass ein vom Benutzer aufgerufenes Fenster so klein ist wie ein Dialogfenster oder der Vergrößerungsfaktor ist so gering, dass das Fenster auch im vergrößerten Zustand ausreichend Platz auf dem Bildschirm hat. In diesem Fall könnte dieses ebenfalls sofort und vollständig vergrößert werden. Dass bei einem gewöhnlichen Fenster dann auch Elemente vergrößert werden, die miteinander in keinem Zusammenhang stehen, kann ignoriert werden, schließlich nehmen die Elemente keinen Platz weg. Im Grunde sollte es zunächst nur von

der Größe des Fensters abhängen, ob das Fenster gleich vergrößert wird oder erst einmal im Originalzustand bleibt.

Um den Prozess zu vereinfachen, könnte man immer jedes Fenster vergrößern, allerdings würde der Inhalt in vielen Fällen nicht auf den Bildschirm passen. Dieses Problem könnte man in den Griff bekommen, wenn man den Fenstern Bildlaufleisten hinzufügt. Leider steigt dadurch die Unübersichtlichkeit sehr stark an und das würde nicht ins Konzept passen.

Anstatt nur die Höhe und Breite eines Fensters zu betrachten und das Fenster nur dann zu vergrößern, wenn die Fenstermaße die Maximalmaße nicht überschreiten, könnte auch die Fläche des vergrößerten Fensters abgeschätzt und mit der Anzeigefläche des Bildschirms verglichen werden. Es ist möglich, dass durch das Vergrößern nur die Breite des Fensters die Maximalgröße übersteigt. Wenn aber die Höhe des Fensters kleiner ist als die Höhe des Bildschirms, dann könnte man das Fenster in den Maßen soweit anpassen, dass bei gleichbleibender Fläche des Fensters dieses im Folgenden auf den Bildschirm passt. Und da in das Layout nicht eingegriffen wird, bleibt dieses unverändert. Die Änderungen machen sich hauptsächlich nur dadurch bemerkbar, dass Textfelder gezwungen werden früher umzubrechen. So könnte man die Anzahl der Fenster, die sofort und vollständig vergrößert werden, erhöhen, ohne dass die Qualität der vergrößerten Fenster abnimmt. Nachdem ein Fenster vergrößert wurde, muss darauf geachtet werden, dass es nicht zu klein für den Inhalt wird. Ist das der Fall, muss es der Größe des Inhalts angepasst werden.

Für den Fall, dass ein Fenster zu groß ist, um gleich vollständig vergrößert zu werden, wird eine Vergrößerungsmethode benötigt, die sich gut in das vorhandene Konzept einfügt. Eine automatische und von der Position des Mauszeigers abhängige Vergrößerung von Elementen in Fenstern könnte man sehr gut für größere Elemente umsetzen, wie Menüs. Und auch für Box-Elemente, die andere Elemente zusammenfassen, würde eine solche Lösung funktionieren. Allerdings gibt es in den Fenstern auch kleinere Elemente, die nicht zusammengefasst sind, wie einzelne Checkboxen oder Textfelder. Diese müssten bei Fokussierung einzeln vergrößert werden. Bei einer provisorischen Umsetzung mit Firefox hat sich allerdings gezeigt, dass eine solche Lösung zum starken Flackern neigt. Deshalb wurde beschlossen, Auswahl und Vergrößerung der Elemente dem Benutzer zu überlassen. Dieser soll nach Wunsch ein Element fokussieren und durch einen Mausklick vergrößern können. Es soll also möglich sein sowohl einzelne als auch zusammengehörende Elemente eines Fensters auszuwählen und zu vergrößern. Mit dieser Methode wird es dem Benutzer ermöglicht das zu vergrößern, was für ihn entweder zusammengehört oder einfach wichtig ist. Unwichtige Elemente bleiben klein und nehmen kein Platz weg.

5.3 Umsetzung des Konzepts

Damit bei Erscheinen eines neuen Fensters bestimmte Ereignisbehandlungsroutinen ausgeführt werden, wird ein Event-Handler registriert, was dazu führt, dass in solchen Fällen die Funktion *onOpenWindow* aufgerufen wird. Diese nutzt das XPCOM-Interface *nsIWindow-Mediator* um alle geöffneten Fenster durchzugehen und für jedes Fenster, das kein Browserfenster ist, die Funktion *CheckSize* aufzurufen. *CheckSize* schätzt die Fläche des Fensters im vergrößerten Zustand und führt dem Konzept entsprechende Anweisungen aus. Um zu verhindern, dass die Funktion *CheckSize* für ein Fenster mehrmals aufgerufen wird, wird an jedem Wurzelement eines Fensters ein Attribut gesetzt, an dem man erkennen kann, ob das Fenster diesen Schritt schon durchlaufen hat.

Sobald ein Fenster geladen ist, wird berechnet, wie groß seine wahrscheinliche Fläche nach Vergrößerung sein wird. Ist sie geringer als die Anzeigefläche des Bildschirms, wird das Fenster vollständig vergrößert. Das Vergrößern eines Elements erfolgt indem das Attribut `ffzwin` auf den Wert „big“ gesetzt wird, wodurch bestimmte CSS-Regeln auf das Element angewendet werden, was wiederum zur Vergrößerung des Elements führt (ähnlich wie im letzten Teil in Abbildung 7 dargestellt). Anschließend wird überprüft, ob das Fenster dennoch die Maximalmaße überschritten hat. Stellt sich heraus, dass das Fenster breiter ist als die maximal erlaubte Größe, wird das Fenster in der Höhe vergrößert und in der Breite dem Bildschirm angepasst. Dies geschieht durch das Anwenden der JavaScript-Methode *window.resizeTo()*. Ziel ist es dem Fenster die nötige Höhe zu geben, sodass der Inhalt doch noch hineinpasst. Passt das Fenster dann von der Höhe nicht mehr, wird es wieder in den Originalzustand versetzt. Im letzten Schritt wird das Fenster auf dem Bildschirm zentriert indem aus den Fenster- und Bildschirmmaßen die nötigen Koordinaten berechnet werden und die Methode *window.moveTo()* ausgeführt wird.

Ist ein Fenster zu groß um vollständig vergrößert zu werden, dann bleibt es im Originalzustand. Allerdings wird eine Funktion aufgerufen, die allen XUL-Elementen des Fensters Event-Listener zuweist. Die Ereignisse „mouseover“ und „mouseout“ sollen es ermöglichen zu erkennen über welchem Element sich der Mauszeiger befindet. Das ist nötig um dem Benutzer zu zeigen, welches Element fokussiert ist. Dies geschieht, indem um das Element eine rot gestrichelte Umrandung angezeigt wird (Abbildung 10). Ein Mausklick auf ein fokussiertes Element löst ein „click“-Ereignis aus, was dazu führt, dass das Element vergrößert wird. Auch hier wird das Vergrößern durch das Setzen des `ffzwin`-Attributs ausgelöst. Möchte man das Element wieder verkleinern, ist das Vorgehen analog. Man fokussiert das Element, wobei nun eine grün gestrichelte Umrandung angezeigt wird, und betätigt die Maustaste (für detaillierte Angaben zu Tastaturbelegungen siehe Anhang A, Seite 51). Das Attribut wird erneut modifiziert, die CSS-Regeln werden ungültig und das Element kehrt in den Originalzustand zurück.

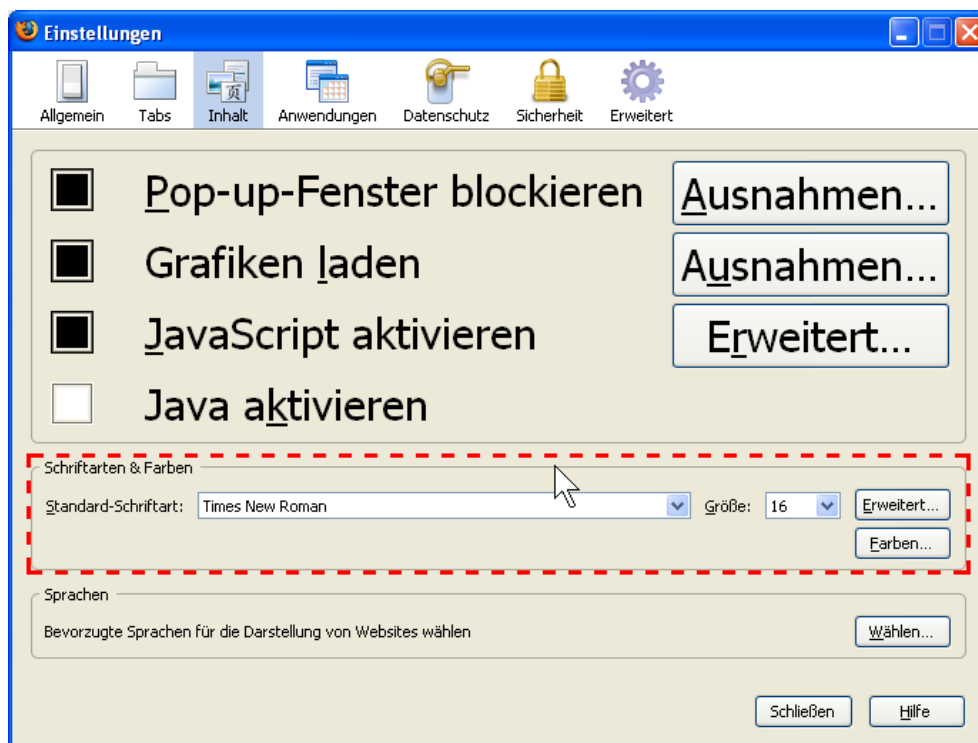


Abbildung 10: Ein vergrößertes Element über einem fokussierten Element

Um eine möglichst hohe Flexibilität zu gewährleisten, kann der Benutzer auch bei vergrößerten Fenstern einzelne Elemente vergrößern oder verkleinern. Wird beispielsweise kein vollständig vergrößertes Fenster gebraucht, dann kann man dieses wieder in den Originalzustand versetzen und dann nur diejenigen Elemente vergrößern, die man braucht. Um das zu tun, werden Funktionen benötigt, die im folgenden Abschnitt erläutert werden.

5.4 Zusätzliche Funktionen

5.4.1 Speicherfunktion

Um zu verhindern, dass Einstellungen, die der Benutzer an einem Fenster getätigt hat, verloren gehen und der Benutzer gezwungen wird bei jedem Aufruf eines Fensters wieder alles neu einzustellen, wird eine Funktion benötigt, die sich den Zustand des Fensters merkt. Bei jedem Start eines gespeicherten Fensters muss dann der Zustand wiederhergestellt werden.

Da die Speicherfunktion nicht Teil des Konzepts, wurde sie nur in beschränktem Maße umgesetzt. Bei Fensterstart wird allen nicht-anonymen XUL-Elementen ein Attribut mit einer fortlaufenden Nummer zugewiesen. Wenn der Benutzer einen Speichervorgang initiiert, werden die Nummer des Elements und relevante Attribute als ein String gespeichert. Ein Speichereintrag für ein Fenster könnte folgendermaßen aussehen:

5.4.1 Speicherfunktion

```
677/448/306/135/39,true,width: 231px ! important; height: 97px ! important; /
```

Die Einträge sind durch Schrägstriche getrennt. Die ersten beiden Zahlen stehen für die Fenstergröße, die nächsten beiden für die Koordinaten auf dem Bildschirm. Von da an hat jeder Eintrag dasselbe Muster: Durch Kommas getrennt sind Nummer des Elements, Wert des `ffzwin`-Attributs und Wert des `style`-Attributs.

Bei Start eines Fensters wird überprüft, ob ein Speichereintrag existiert. Ist das der Fall, werden anhand der gespeicherten Nummern die modifizierten Elemente identifiziert und die Attribute entsprechend geändert.

Leider funktioniert die Speicherfunktion nicht immer. Die Nummerierung kann sich in bestimmten Fenstern von Aufruf zu Aufruf unterscheiden. Die Gründe sind vielfältig, haben aber dieselbe Ursache: in solchen Fällen handelt es sich um dynamische Fenster, d.h. es sind Fenster, deren Elemente sich ändern können oder deren Elemente erst eingebunden werden, wenn erstmals auf sie zugegriffen wird. Die Speicherfunktion funktioniert am besten mit vollkommen statischen Fenstern.

5.4.2 Verändern der Größe von Elementen und Fenstern

Box-Elemente wie `<groupbox>`, `<hbox>` und `<vbox>` wachsen nur bis zu einem bestimmten Grad mit dem Inhalt, wenn dieser vergrößert wird. Deshalb kann es vorkommen, dass solche Elemente zu klein für ihren Inhalt werden. Die dann eingeblendeten Bildlaufleisten tragen nicht unbedingt zur Verständlichkeit bei. Deswegen wurde eine Funktion implementiert, mit der der Benutzer diese Elemente nach Wunsch in der Größe verändern kann. Zunächst geht man mit dem Mauszeiger auf das Element und hält die Taste [Ctrl] sowie die Maustaste gedrückt. Es wird ein „mousedown“-Ereignis ausgelöst, welches den Vorgang zur Größenänderung startet. Wenn man nun den Mauszeiger zieht, wird die Größe des Elements verändert indem die CSS-Eigenschaften „width“ und „height“ im `style`-Attribut des Elements modifiziert werden. Auch Fenster lassen sich so in der Größe ändern. Dazu wird allerdings die bereits angesprochene Methode `window.resizeTo()` verwendet.

5.4.3 Weitere Funktionen

Um dem Benutzer eine Auswahl an hilfreichen Funktionen zur Verfügung zu stellen, hat jedes Fenster ein Kontextmenü. Neben der schon erwähnten Speicherfunktion, enthält das Kontextmenü eine Funktion, die es dem Benutzer erlaubt das Fenster zu maximieren. Das wäre zwar auch mit der Vergrößerungsfunktion möglich, doch diese Funktion verkürzt die Aktion auf einen Mausklick. Die maximalen Bildschirmmaße werden aus den Eigenschaften `window.screen.availHeight` und `window.screen.availWidth` ermittelt.

5.4.3 Weitere Funktionen

Eine weitere Funktion erlaubt es dem Benutzer alle Elemente im Fenster zu vergrößern. Das kann nützlich sein, wenn das Fenster bei Start nicht vergrößert wurde, obwohl das möglich gewesen wäre. Im Grunde wird das nur `ffzwin`-Attribut am Wurzelement des Fensters gesetzt.

Die letzte Funktion macht alle Änderungen, die der Benutzer an den Elementen und dem Fenster durchgeführt hat, rückgängig. Hier werden für alle Elemente die beiden Attribute `ffzwin` und `style` zurückgesetzt. Im letzten Schritt wird die Größe des Fensters dem Inhalt angepasst.

6 Vergrößerung von Webseitenelementen ohne Lupen

Die Aufgabenstellung in Abschnitt 1.3 fordert eine Vergrößerungstechnik mit der man Bereiche von Webseiten ohne Lupen oder Lupen-ähnliche Werkzeuge vergrößern kann. Dieses Kapitel beschreibt eine Umsetzung, bei der die Vergrößerung direkt im Browserbereich stattfindet. Die Methode ist an die Vergrößerung von Fensterelementen in Kapitel 5 angelehnt. Dabei fokussiert der Benutzer ein Element und betätigt die Maustaste, was zur Vergrößerung des Elements führt.

6.1 Umsetzung

Um ein Element zu fokussieren, hält man die Tasten [Ctrl] und [Alt] gedrückt und bewegt den Mauszeiger über das Element. Dabei bekommt das Element eine rote Umrandung. Betätigt man nun die Maustaste, wird das Element vergrößert.

Folgende CSS-Regeln sorgen für die Änderungen:

```
*[ffz='big'],
*[ffz='big'] * {
  font-size: 30px !important;
}

*[ffz='big'],
*[ffz='big'] * {
  line-height: 1 !important;
}

*[ffz='big'] {
  overflow: auto !important;
  max-height: none !important;
  max-width: none !important;
}

*[ffz='big'],
*[ffz='big'] * {
  height: auto !important;
  width: auto !important;
}
```

Die erste Regel setzt die Schriftgröße auf die eingestellte Größe. Damit die Textlinien nicht übereinander geraten, wird mit der zweiten Regel festgelegt, dass der Zeilenabstand „einzeilig“ sein soll. Die letzten beiden Regeln sorgen dafür, dass Bildlaufleisten auftauchen, wenn der Inhalt zu groß für das Element ist und dass die Elementgröße veränderlich wird.

6.2 Navigation mit der Tastatur

Das in Abschnitt 3.4 beschriebene Verfahren, die Verwendung von Tabulator, erscheint hier wenig sinnvoll, denn eine Webseite kann Hunderte von Elementen enthalten. Eine Funktion, bei der der Benutzer Dutzende Male den Tabulator betätigen muss, um ein bestimmtes Element zu erreichen, wäre von wenig Nutzen. Eine Methode, die möglicherweise funktionieren könnte, ist die Navigation mit den Pfeiltasten einer Tastatur. Allerdings stellt sich die Frage, auf welcher Ebene man die Elemente fokussiert. Auch hier gilt: Semantische Zusammenhänge sind in Webseiten schwer zu erfassen. Bewegt man sich also standardmäßig auf der Blattebene des DOM-Baums, dann hat man evtl. Elemente, die sehr klein sind. Bewegt man sich im DOM-Baum zu hoch, dann hat man evtl. zu große Elemente. Die Entscheidung dem Benutzer zu überlassen ist auch nicht ideal. Nicht jeder Benutzer hat Kenntnis vom Aufbau einer Webseite und kann abschätzen, ob er sich im DOM-Baum nach oben oder nach unten bewegen muss, um zu einem bestimmten Element zu gelangen. Außerdem kann bei der Menge an Elementen die Navigation frustrierend werden und keinesfalls erträglich für jemanden, der ohnehin schlecht sieht.

Beim Laden einer Webseite müsste der gesamte DOM-Baum analysiert und ein Stellen-Transitions-Netz erstellt werden. Anhand dieses müsste es dann möglich sein, für jedes fokussierte Element und für jede Tastatureingabe festzustellen, welches Element als nächstes fokussiert werden soll. Das setzt voraus, dass das Programm erkennen kann, welches Element sich neben einem anderen befindet, womit man wieder bei obigem Problem wäre: Auf welcher Ebene wählt man ein Element aus? Abgesehen davon, im DOM-Baum benachbarte Elemente müssen nicht unbedingt auch auf dem Bildschirm nebeneinander liegen. Ein weiteres Problem würde bei der Vergrößerung entstehen, da sich durch die veränderte Größe auch die Position eines Elements ändern kann. Elemente, die in der Originalansicht nebeneinander lagen, können nach Vergrößerung weit weg voneinander sein. Und der Graph müsste dann von neuem erstellt werden.

7 Fazit

Das Ergebnis der vorliegenden Arbeit ist die Firefox-Erweiterung FirefoxZoom, die das Konzept aus Abschnitt 1.2 umsetzt. Dabei wurde für jeden Aufgabenbereich aus Abschnitt 1.3 eine Lösung entwickelt und implementiert. Die Erfahrungen mit dem Konzept waren überwiegend positiv. Es konnten Lösungen für alle Probleme gefunden werden. Von Fall zu Fall musste das Konzept ergänzt werden, einige Konzeptkriterien mussten vereinzelt missachtet werden, um die Bedienung der Anwendung zu vereinfachen. Das lag selten an dem Konzept und öfter an der Tatsache, dass das Konzept viel Interpretationsraum für konkrete Lösungen lässt. Viele Konzeptpunkte waren in konkreten Fällen nicht miteinander vereinbar und es konnte keine Lösung gefunden werden, die alle Punkte zufrieden stellt (siehe Abschnitt 3.2.2). Die Erarbeitung von Standard-Lösungen für gängige Elemente einer Bedienungsfläche und für besondere Probleme wird empfohlen.

Stark auffallend ist die Tatsache, dass das Konzept viele Probleme, die es bei der Navigation mit der Maus hat, mit der Tastatur nicht hat. Insofern könnte man davon sprechen, dass das Konzept besser für die Navigation mit der Tastatur geeignet ist, als für die Navigation mit der Maus. Andererseits ist es nicht möglich sich mit der Tastatur durch die Elemente einer Webseite oder eines XUL-Fensters zu bewegen. Hierfür liefert das Konzept keine Lösung, wofür es aber auch nicht gedacht ist. Ein Anwendungsbereich des Konzepts könnte die Bedienungsfläche von mobilen Geräten sein. Aufgrund der aufkeimenden Touchscreen-Technologien könnte das Konzept auch für die Bedienung mit dem Finger angepasst werden.

Auch was die Firefox-Erweiterung als Hilfe für sehbehinderte Menschen angeht, sind noch Verbesserungen möglich. Für einen flexibleren Einsatz des Konzepts, wird eine Implementierung empfohlen, bei der der Benutzer entscheidet, wie Elemente vergrößert werden sollen. Im Grunde sollten die Funktionen aus Kapitel 5 auf den gesamten Webbrowser angewendet werden. Der Benutzer sollte jedes Element fokussieren können um anschließend aus einer Reihe von Operationen auszuwählen, wie sich das Element verhalten soll, ob es z.B. immer vergrößert werden soll oder nur bei Fokus. Anschließend sollte eine Speicherfunktion den Zustand speichern. Bei nächstem Aufruf der Erweiterung würde der gespeicherte Zustand geladen werden. Natürlich müssten weitere Funktionen hinzugefügt werden, um die Bedienung zu verbessern.

Anhang A Tastenkombinationen in FirefoxZoom (deutsche Lokalisierung)

Im Browserfenster:

- STRG + ALT + Z – FirefoxZoom starten/beenden
- STRG + ALT + H – Schriftgröße erhöhen
- STRG + ALT + R – Schriftgröße reduzieren
- STRG + ALT + O – FirefoxZoom Optionen aufrufen
- STRG + ALT + L – Neue Lupe öffnen
- STRG gedrückt halten, mit dem Mauszeiger ein Element auswählen, Doppelklick – Öffnet eine neue Lupe mit dem ausgewählten Element
- STRG + ALT gedrückt halten, mit dem Mauszeiger ein Element auswählen, linke Maustaste klicken – Vergrößert/Verkleinert ausgewähltes Element

In aufgerufenen Fenstern:

- Rechtsklick – Ruft ein Kontextmenü auf, welches folgende Einträge enthält: Fenster maximieren, Zustand speichern, Alles vergrößern/verkleinern, Alles zurücksetzen
- STRG gedrückt halten, mit dem Mauszeiger ein Element auswählen, linke Maustaste gedrückt halten und ziehen – Verändert die Größe des Elements
- STRG gedrückt halten, mit dem Mauszeiger auf die rechte untere Ecke des Fensters gehen, linke Maustaste gedrückt halten und ziehen – Verändert die Größe des Fensters
- STRG + ALT gedrückt halten, mit dem Mauszeiger ein Element auswählen, linke Maustaste klicken – Vergrößert/Verkleinert ausgewähltes Element

Anhang B Verzeichnisstruktur der Erweiterung

```
+ firefoxzoom
|
+ - install.rdf
|
+ - chrome.manifest
|
+ - + content
|   |
|   + - overlay.xul (GUI: Lupe, Pop-up-Menüs, zusätzliche Menüeinträge)
|   |
|   + - firefoxzoom.js (Leisten, Tooltips)
|   |
|   + - lens.js (Lupen, direkte Vergrößerung)
|   |
|   + - listener.js (Event-Listener auf Einstellungsvariablen)
|   |
|   + - options.xul (GUI: Optionen-Fenster)
|   |
|   + - options.js (Funktionen für das Optionen-Fenster)
|   |
|   + - windows.js (neu geöffnete Dialoge und Fenster)
|   |
|   + - stylesheet.xul (dynamische Änderung der Darstellung: gl. Stylesheet)
|   |
|   + - common.js (gemeinsame Funktionen und Variablen)
|
+ - + locale
|   |
|   + - + de-DE (deutsche Bezeichnungen)
|   |   |
|   |   + - firefoxzoom.dtd
|   |   + - firefoxzoom.properties
|   |
|   + - + en-US (englische Bezeichnungen)
|   |   |
|   |   + - firefoxzoom.dtd
|   |   + - firefoxzoom.properties
|
+ - + skin
|   |
|   + - always.css (zur gesamten Laufzeit gültige CSS-Regeln)
|   |
|   + - {Bilder}
|   |
|   + - {Mauszeiger}
```

Literaturverzeichnis

- [BIT09] Erstmals arbeitet die Mehrheit der Deutschen am Computer, Zugriff am: 16.2.2009, URL: http://www.bitkom.org/de/presse/43408_41261.aspx
- [BIT209] PC-Ausstattung in Deutschland knackt erstmals 75-Prozent-Marke, Zugriff am: 16.2.2009, URL: http://www.bitkom.org/de/presse/30739_43365.aspx
- [BIT309] Drei von vier deutschen Haushalten haben Internetzugang, Zugriff am: 16.2.2009, URL: http://www.bitkom.org/de/presse/8477_56246.aspx
- [FKT08] Christiane Taras, Thomas Ertl, Fokus-und-Kontext-Techniken zur intelligenten Vergrößerung von graphischen Benutzungsoberflächen, 2008
- [XUL09] XML User Interface Language - Wikipedia, Zugriff am: 21.1.2009, URL: http://de.wikipedia.org/wiki/XML_User_Interface_Language
- [ANC09] Anonymous Content - MDC, Zugriff am: 16.2.2009, URL: https://developer.mozilla.org/En/XUL_Tutorial/Anonymous_Content
- [XPC09] XPCOM - MDC, Zugriff am: 6.1.2009, URL: <https://developer.mozilla.org/en/XPCOM>
- [MOZ09] Mozilla Application Framework in Detail - MDC, Zugriff am: 8.1.2009, URL: https://developer.mozilla.org/en/Mozilla_Application_Framework_in_Detail
- [FUE09] FUEL - MDC, Zugriff am: 4.1.2009, URL: <https://developer.mozilla.org/en/FUEL>
- [W3C09] Cascading Style Sheets, Level 2, Zugriff am: 17.2.2009, URL: <http://www.edition-w3c.de/TR/1998/REC-CSS2-19980512/kap05.html>
- [JSC09] JavaScript - Wikipedia, Zugriff am: 26.1.2009, URL: <http://de.wikipedia.org/wiki/Javascript>
- [ECM09] ECMAScript - Wikipedia, Zugriff am: 17.2.2009, URL: <http://en.wikipedia.org/wiki/ECMAScript>
- [DOM09] Document Object Model - Wikipedia, Zugriff am: 17.2.2009, URL: http://de.wikipedia.org/wiki/Document_Object_Model
- [WDM09] Document Object Model (DOM), Zugriff am: 17.2.2009, URL: <http://www.w3.org/DOM/>
- [JAS03] Tobias Hauser, JavaScript - Interaktives und dynamisches Webpublishing, Markt+Technik Verlag, 2003

- [FIR09] Firefox - Wikipedia, Zugriff am: 26.1.2009, URL: <http://de.wikipedia.org/wiki/Firefox>
- [NSC09] Netscape Communicator - Wikipedia, Zugriff am: 19.2.2009, URL: http://de.wikipedia.org/wiki/Netscape_Communicator
- [FFO09] Firefox erreicht 20 Prozent Marktanteil in Europa, Zugriff am: 19.2.2009, URL: <http://www.heise.de/newsticker/Firefox-erreicht-20-Prozent-Marktanteil-in-Europa--/meldung/68393>
- [ADO09] Firefox Add-ons, Zugriff am: 19.2.2009, URL: <https://addons.mozilla.org/de/firefox/>
- [XUL07] Jonathan Protzenko, XUL - Entwicklung von Rich Clients mit der Mozilla XML User Interface Language, Open Source Press, 2007
- [PFF07] Kenneth C. Feldt, Programming Firefox, O'Reilly Media, 2007
- [XPI09] XPI - Wikipedia, Zugriff am: 19.2.2009, URL: <http://de.wikipedia.org/wiki/XPI>
- [EXT09] Building an Extension - MDC, Zugriff am: 19.2.2009, URL: https://developer.mozilla.org/en/Building_an_Extension
- [MDC08] panel - MDC, Zugriff am: 15.12.2008, URL: <https://developer.mozilla.org/en/XUL/panel>
- [COP08] CoolPreviews, Zugriff am: 15.12.2008, URL: <https://addons.mozilla.org/de/firefox/addon/2207>
- [CON99] Heinrich Lindner, Friedrich-Wilhelm Röhl, Wolfgang Behrens-Baumann, Kontrastoptimierung im öffentlichen Bereich - Verbesserung der Orientierungsmöglichkeiten für Sehbehinderte, 1999
- [DIA09] Dialogfenster - Wikipedia, Zugriff am: 14.1.2009, URL: <http://de.wikipedia.org/wiki/Dialogfenster>