CHALMERS

DomScape: 3D web development

An integrated visualization of code and layout

Master of Science Thesis in the Interaction Design Programme

EELKE BOEZEMAN

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, November 2010

DomScape: a 3D front-end web-development tool
An investigation into how DomScape compares with Firebug

EELKE BOEZEMAN

Examiner: MORTEN FJELD

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
A three-dimensional visualization of a web page by DomScape

Department of Computer Science and Engineering
Göteborg, Sweden. November 2010

Abstract

This study has investigated DomScape, a front-end web-development tool that visualizes web pages as three-dimensional structures by combining code with presentation. An experiment was performed to compare DomScape with Firebug, a popular web development tool. On three of four tasks DomScape was faster than Firebug. The results were likely due to the more unified approach of DomScape, which makes it easier to select and interpret elements and to understand how an element contributes to the layout of the web page.

Acknowledgements

I am indebted to my supervisor, Anna Gryszkiewicz, who has been very enthusiastic and dedicated. She has given great feedback and spent a lot of time helping me with the structure and contents of the report. I also want to thank Morten Fjeld, for taking the time and effort as an examiner to this project. Special thanks go to Alejandro Valenzuela Roca, who helped me a great deal with implementing DomScape in OpenGL using his 3D engine, motorJ.

Table of contents

Introduction

In this thesis I introduce DomScape, a new kind of front-end web development tool that visualizes web pages in three dimensions. Web pages are normally rendered in two dimensions - the third dimension is not used (see Figure 1). Traditional web development tools supplement the conventional two-dimensional view with information panels that contain HTML and CSS code. DomScape does it differently. Instead of extending the conventional two-dimensional view, it combines the code structure with code visualization in one, single three-dimensional visualization.

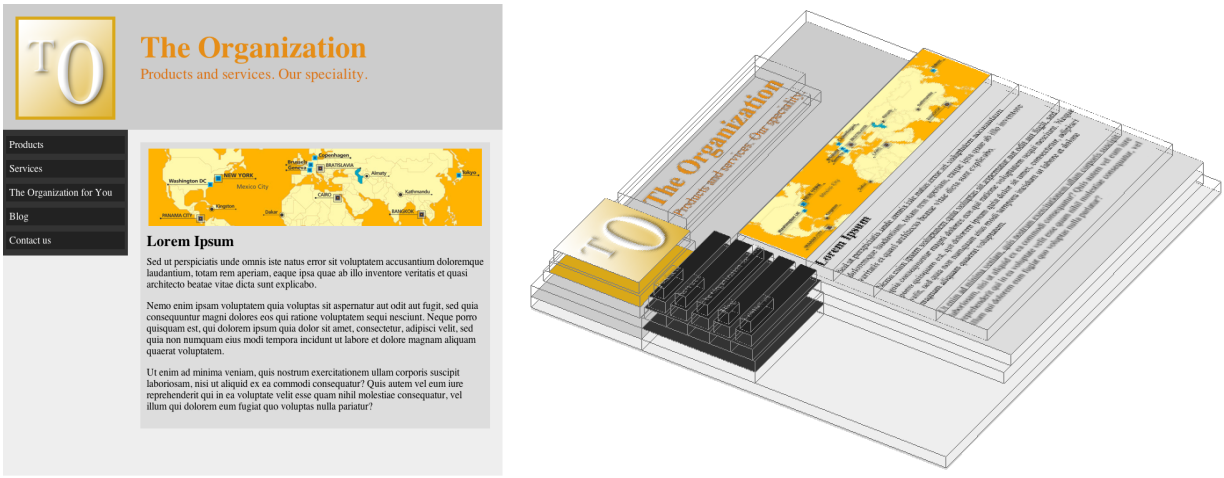

Figure 1. Two visualization of the same web page. On the left is displayed the conventional, two-dimensional visualization used by browsers and web development tools. On the right the web page is displayed as it is visualized by DomScape, using all three dimensions.

We are used to the conventional, two-dimensional visualization provided by browsers. It is what we, as end-users, expect to see. But is it also the best visualization for web developers - people that have to understand the relationship between the code and the layout? DomScape provides an interface that explicitly visualizes the code structure so that it is easier for web developers to understand the relationship between the code and the visualization of the code.

By employing all three dimensions, the visualization is in some way the same, but also very different from the two-dimensional visualization. The web page becomes a three-dimensional object that looks like a landscape and that can be zoomed in on and viewed from all angles. Prolonging the ‘landscape’ metaphor, DomScape allows the user to step in a helicopter, fly over the landscape and inspect the hills and valleys to understand the structure that gives rise to the landscape. On the other hand, browsers and traditional web development tools use the two-

dimensional visualization, which is essentially more like a flat, satellite view of the landscape, in which the hills and valleys have become indistinguishable.

In following sections I will go into detail of how DomScape works and how DomScape compares to traditional web development tools. I end with an hypothesis that provides the red line throughout the rest of this document. To start with, I will put DomScape into context of its intended use: front-end web development.

What is the Web?

A short history

Although most people freely interchange the terms ‘Internet’ and the ‘Web’, they are actually not the same. The Web, short for World Wide Web, is a the name of a system of interconnected documents that runs over the Internet. The Internet has been around for much longer than the Web. When the Web started to get traction among consumers, only then the Internet became as ubiquitous as it is now. The Web was the killer application the Internet was waiting for.

The roots of the Internet can be traced back to the 60s with inception of ARPAnet, a network to connect scientists at different universities in the United States. In the 90s the network changed names as it connected to European networks and the world spanning network was named the Internet (Internet History, 2010).

Figure 2. Berners-Lee ran the first Web server on this NeXT computer at CERN, Switzerland in 1991 (image courtesy of Wikimedia Commons)

Around the same time that the Internet was molded into its current form, Sir Tim-Berners Lee wrote a proposal for a project, “to link and access information of various kinds as a web of nodes in which the user can browse at will” (Berners-Lee and Cailliau, 2010). The project

proposed a system of interconnected documents that the user could access through ‘browsers’ that could be pointed at documents hosted by any server on the Internet.

Because it was open and free of license fees, flexible and easy to implement, the Web gained popularity at a very fast pace. First only in the science community, but later business and consumers started catching on. Currently, almost two billion people in the world are connected to the Internet (Internet World Stats, 2010).

In order to implement the proposal for the Web, Berners-Lee had to invent a number of new technologies and specifications. HTML, short for HyperText Markup Language, was the language he proposed to describe and lay out the content of Web documents. The first specifications of HTML were written by Berners-Lee in 1991 (Berners-Lee, 1992). Over the years the HTML specification went through several iterations. Since 1996 the World Wide Web Consortium (W3C) maintains, with input from browser vendors, the HTML specification. The last published HTML specification is from an astounding 11 years ago when W3C published the HTML 4.01 specification. The first HTML 5 specification working draft was published in January, 2008 and the last one as recent as June, 2010. It is to a large extent supported by most modern modern browsers.

As HTML began to be more and more popular, problems started to arise with nature of HTML code. HTML could contain both content and markup which leads to ‘spaghetti’ code that is hard debug, maintain, and scale. A separation between structure and presentation was proposed by Robert Cailliau (Petrie, 1997). The structure would remain in the HTML document, but the presentation of that structure would be handled by CSS (Cascading Style Sheet) documents. The CSS specification has also been maintained by the W3C and the current specification is CSS 2.1. Although CSS 3 is not officially a standard yet, and still under active development, it is supported by most modern browsers.

In order to access the ‘Web of documents’ Berners-Lee described a new kind of program, called the browser. The browser is responsible for retrieving and rendering the HTML documents and attached CSS documents. How the browser should render the code is almost completely defined by the HTML and CSS specification - with a few exceptions (Meyer, 2006). Although most browsers have failed to completely and correctly implement the specification, there has been a lot of improvement in recent years with the arrival of new browsers such as Mozilla Firefox and Google Chrome.

After the browser has loaded the HTML and CSS code, it constructs a Document Object Model (DOM) of this code in memory. The browser is not required to generate a DOM, but if the browser wants to execute javascript in the web page the DOM is required. Javascript is the programming language that can be used by the web developer to add dynamic functionality to a web page. The DOM acts as wrapper around the HTML and CSS code so that javascript code can access it and alter it. Because it encompasses the entire web page, it is used in the name ‘DomScape’.

HTML and CSS in more detail

HTML is a strict-hierarchical markup language. It does not contain any functionality, but rather defines the structure of the content on a web page. The structure of a web page is created by predefined 'tags'. The HTML specification defines over 50 tags to lay out text, include images, create interactive forms and links to other HTML documents or other files. It also provides means to semantically structure the text through headers, paragraphs, lists, quotes, tables, and other items.

An HTML tag, is term of one or more letters, surrounded by brackets on each end. Content can be tagged by placing it between a beginning and ending tag of the same kind. For example, the title of a page can be tagged as such by placing it between "header" tags, i.e. '`<h1>The title of the page</h1>`'. Besides content, a tag can also contain other tags. Web pages often are structures with a deep hierarchy of elements. The example HTML code in Figure 3 has 8 levels of elements.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
  <body>
    <div id="page">
      <div id="header">
        <div id="inner-header">
          <div id="header-logo">
            <div id="header-logo-inner">
              <img width="100%" src="website/logo.png">
            </div>
          </div>
          <div id="header-text">
            <h1>The Organization</h1>
            <span>Products and services. Our speciality.</span>
          </div>
        </div>
      </div>
      <div id="bottom">
    </div>
  </body>
</html>
```

Figure 3. It is part of the code responsible for the web page displayed in Figure 1, visualized by Firebug.

Cascading Style Sheets documents define how the HTML code is to be presented. CSS documents are attached or directly embedded in the HTML document and define how the tags in that document should be rendered. CSS can be used to define what color an element has, what typography settings apply to its contents, the width and height of the element, where the element should be positioned on the page, how it should deal with other overlapping elements and much more. The following example defines that all content between h1-tags should be rendered red and very large.

```
h1 {
 color: red;
 font-size: xx-large;
}
```

Box model

As defined by the CSS 2.1 standard, every element on the web page is rendered as a box. Because the HTML structure often is an hierarchy of many elements, a web page visualization consists of many boxes containing boxes - this is called the box model. Using the `width` and `height` properties the sizes of the boxes can be controlled. Boxes can be positioned using positioning properties such as `float` and `position`. Because of the flexibility of the box model it can be used to basically achieve any web page layout.

Normally, the box model remains hidden for the end-user, for whom it is only important that the contents are readable and understandable - not how the web page is constructed. However, if one is interested in the structure of the web page, it is easy to reveal the box model, by applying a colored border to every element in the page (see Figure 4).

Figure 4. Each element is made visible by applying a red border to it, revealing the box model of this web page. Wherever the lines seem thicker, the borders of multiple elements are displayed. The web page is the same as displayed in Figure 1.

Scope of this study

Because HTML and CSS are both quite complex, the current implementation of DomScape can only visualize a small fraction of the HTML elements and CSS properties. This study only focuses on the HTML box model and the CSS background-color and background-image property, as that is what is DomScape is capable of visualizing. DomScape currently does not support the CSS properties `float` and `position`, two properties that are commonly used to have more control over the position of elements.

Although most modern browsers support HTML 5 and CSS 3 to a large extent, I have decided not to focus on those versions. They introduce great new possibilities, but do not radically deviate from former specifications. The new specifications are largely backwards compatible. Although it is tempting to ride on the buzz that surrounds HTML 5 and CSS 3, it offers no real meaning to the scope of this thesis.

What is web development?

Web development is the activity that gives rise to web sites and web pages of varying degrees of scale, functionality and design. There are many different aspects to web development, ranging from high-level, user centered front-end design to low-level work on back-end server processes.

This thesis does not cover the entire variety of tasks that a web developer could be involved in. Instead, it focuses on one particular common web development task that is part of different front-end web development activities: the relationship between the structure of the HTML and CSS code and the layout of the visual representation of that code. For most web pages, this relationship is implemented with a box model. Other options are using a table layout or third party plug-ins (such as Adobe Flash), but these are not the subject of this thesis.

HTML and CSS offer a very flexible tool set to create an enormously wide variety of web pages. A particular design can be implemented in many different ways. To know how a particular layout has been implemented, the elements and the applied CSS must be analyzed in order to understand how they give rise to the layout.

When a web developer ‘themes’ a web site, he has limited control over the HTML code that is generated by the back-end. The task of the web developer is to implement a particular design by introducing new, and altering existing, CSS code. In other words, the developer cannot change the structure of the box model, but can change the appearance of it. As the HTML code is unknown to the web developer beforehand and can also be quite large, layered and complex, the web developer’s job can be non-trivial.

What is the current state of web development tools?

Over the years, web development tools have been created that help the web developer with the many aspects of web development. As broad as the term web development is, so wide and diverse is the selection of tools available to the web developer nowadays. There are tools to generate entire layouts, tools that assist the developer with typography, colors, images, and forms. Other tools specialize in validating existing HTML and CSS code, or to generate, minimize, or optimize it. There are tools that are web based, others are available as a browser plug-ins and there are tools that run as stand alone programs.

Needless to say, web developer of today has many options. However, if we focus on tools created to work with the box model and help the web developer understand the relationship between the code and layout, the selection becomes a lot smaller. Because of the nature of the task, all tools that offer such functionality run as a plug in in the browser or embed the browser's rendering engine inside a stand-alone application. Below the most popular solutions listed (Komenda, 2008). Because web-based (javascript) tools are limited in their capabilities, they are not covered in this section.

Stand-alone applications that embed a rendering engine:

- Adobe Dreamweaver (CS4 and CS5): WebKit
- Coda: WebKit
- KompoZer: Gecko
- Nvu: Gecko

Plug ins for browsers:

- Firebug: Firefox (see Figure 5)
- Web Developer: Firefox, Chrome
- Developer Tools (WebKit): Chrome
- Web Inspector (WebKit): Safari
- Opera Dragonfly: Opera
- IE Developer Toolbar: Internet Explorer

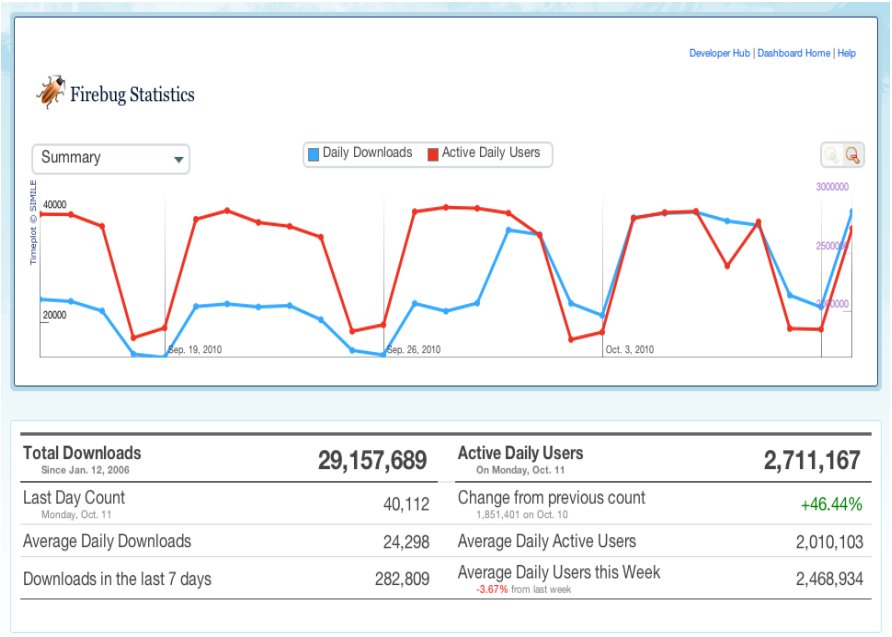

Figure 5. Firebug for Firefox is likely the most popular browser-based web development tool on the web. It is on average used by 2 million users a day. Source: Mozilla Foundation

Shared functionality

The tools are implemented differently, but do overlap to a large extent in the functionality that they offer to the developer. In this section I have listed the functionalities that are relevant to this project. Features such as Javascript debugging and load and connection statistics are left out.

Underwater view

An expert feature in old WordPerfect versions that ran on MS-DOS was the 'underwater view'. This feature splits the screen horizontally in two. In the top pane the text was normally displayed, but in the bottom pane all the markup tags were displayed as well. The underwater view enabled the experienced user to debug a document whenever it would produce awkward results when printed out.

Much in the same way the browser plug-ins split the browser window in two: the top pane is where the web page is displayed and in the lower pane the plug-in is loaded. The plug-in presents the user with another, but different perspective on the same code that gives rise to the visual representation. The plug-in pane contains two main interface elements: the DOM explorer panel and CSS properties panel (see Figure 6).

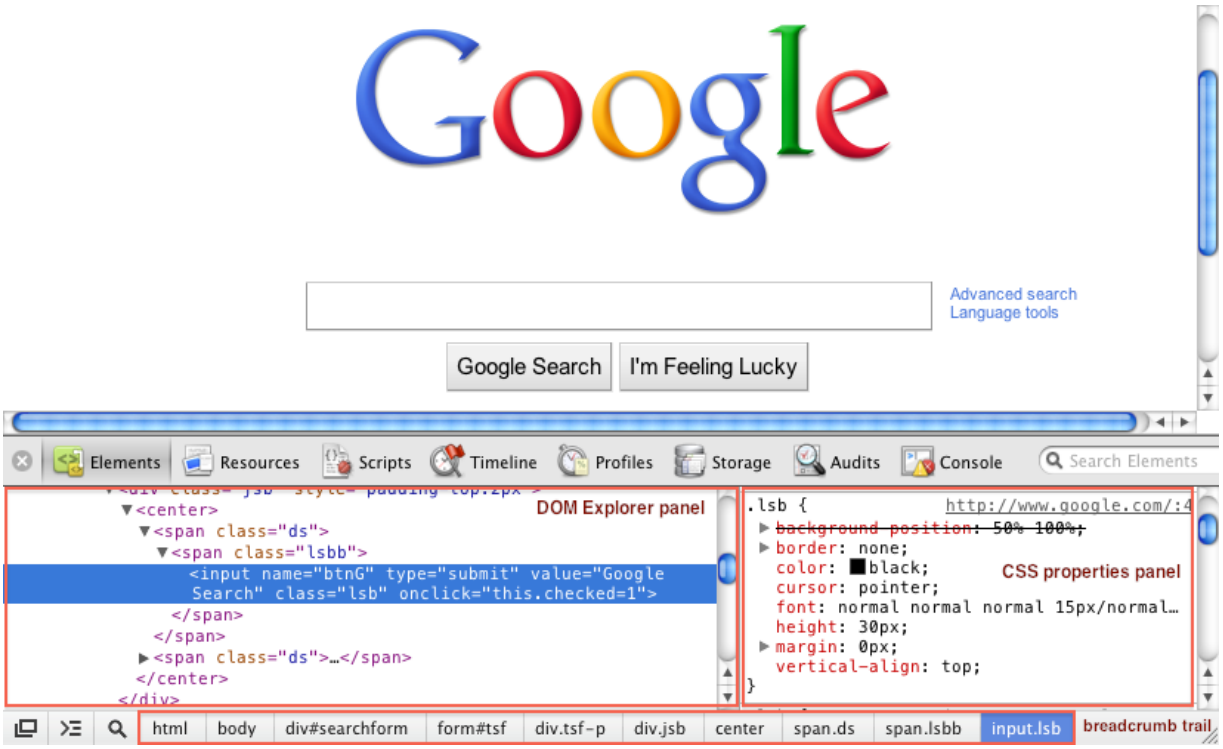

Figure 6. Chrome Developer Tools loaded on the Google homepage. The plug-in is loaded in the bottom panel and the web page is displayed in the top. In the plug-in panel, the DOM Explorer is displayed on the left and the CSS properties on the right. In the bottom the Breadcrumb trail is displayed.

DOM explorer panel

The HTML structure is displayed as a collapsible tree (see Figure 6). If the user is interested in

a particular element, he can explore the tree by expanding elements. The collapsible tree view offers an intuitive and clean view on the code.

Breadcrumb trail
To simplify traversing the DOM tree, a breadcrumb trail of the tree is provided. A horizontal list of ancestor elements makes it easy to see where in the tree the element is located. The elements are selectable to allow the user to quickly traverse the tree to a particular ancestor element (see Figure 6).

CSS properties panel
The CSS code that applies to a selected element is displayed in the style section, including default browser styles (see Figure 6). CSS allows style definitions to be overwritten by other style definitions (for example, because they were declared later). Overwritten style definitions are listed to help the user understand why those definitions no longer apply.

Select elements by mouse
All plug-ins allow the user to use the mouse to select elements directly in the web page and inspect their properties. When the mouse hovers over an element, a box is drawn around the edges of the element. In addition to that, the Chrome and Safari plug-ins highlight the element with a transparent blue layer.

When an element is selected (either by hovering or clicking, depending on the plug-in), the element is displayed in the DOM tree and the properties that apply to that element in the style section.

Box model shading
When an element is selected, either in the plug-in or on the page itself, not only that element is highlighted in the web page, but its padding and margin values are also visualized. In Firebug margin is visualized with a yellow transparent overlay and padding with a purple transparent overlay (see Figure 7). In the native Chrome and Safari plug-ins padding and margin are visualized with same kind of blue transparent overlay that is used to highlight the element.

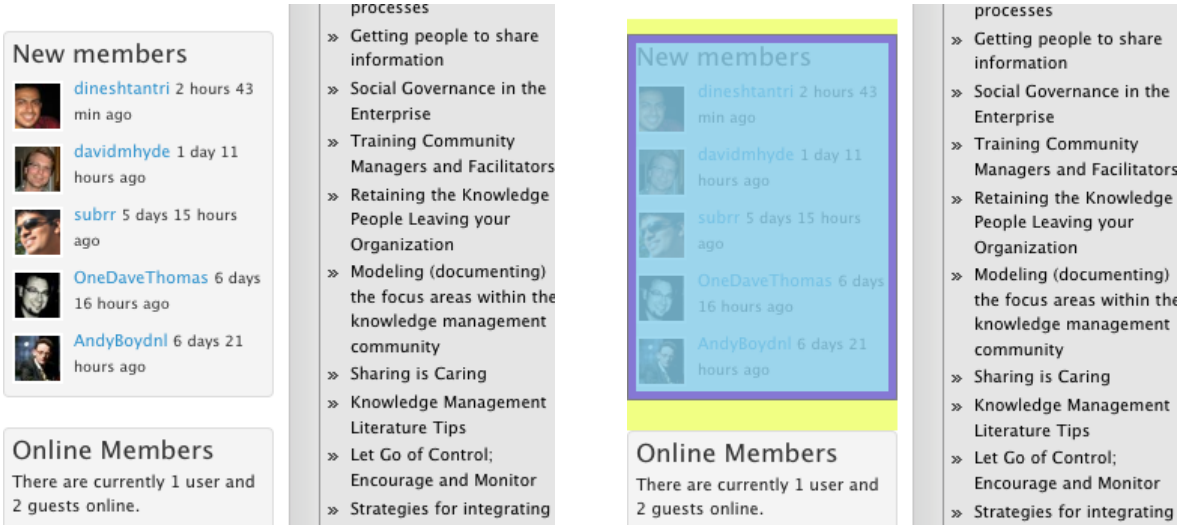

Figure 7. Two versions of the same web page. On the left is the normal view and on the right Firebug has visualized the margin (yellow) and padding (purple) that apply to on of the elements visualized. Web site: kmers.org

Instant editing
In all plug-ins it is possible to adjust the HTML and CSS code and see the effects visualized directly in the web page view. It is possible to delete, add and edit elements and all possible CSS properties. It is also possible to toggle certain CSS properties to see how that affects the web page's layout.

The changes are only stored in the current DOM model in the working memory of the browser. When the page is reloaded, the DOM gets rebuild and the changes are lost. This is however a helpful method to do quick prototyping.

The stand-alone applications outdo the browser plug-ins when it comes to editing the web page. They are more flexible and versatile and allow the user to save the code after any adjustments.

Search
A search field allows the user to search the HTML code. Search phrases are matched against tags, values and content. The search does not find any results in the attached style sheet documents.

DomScape: a different approach

DomScape is a web development tool, but takes an entire different approach compared to the traditional web development tools. These kind of tools maintain the original visual representation and add interface panels that provide the user with the code that applies to the current web page. DomScape attempts to provide the same level of functionality by creating a new kind of visualization that combines code structure with the visual representation of that structure.

DomScape creates a a three-dimensional visualization of the web page by assigning every element in a web page a value on the unused z-axis. The web page is visualized in a much different way that offers new possibilities in understanding the relationship between code and layout.

DomScape is not meant as replacement for traditional web development, but rather as an addition to existing solutions. However, in its current implementation DomScape is a stand-alone program. Ideally, it would be integrated into an existing tool, such as Firebug.

In this section I will go into detail how it DomScape works, what elements and properties it can visualize and what its limitations are.

How does it work?

DomScape is in essence a very simple concept. By analyzing the structure of the code, it is possible to assign every element a position in three-dimensional space in such a way that none of the elements intersect with each other. This creates an entirely new visualization of the web page, which is demonstrated in Figure 8. The camera is placed in such an angle and position that all elements are visible.

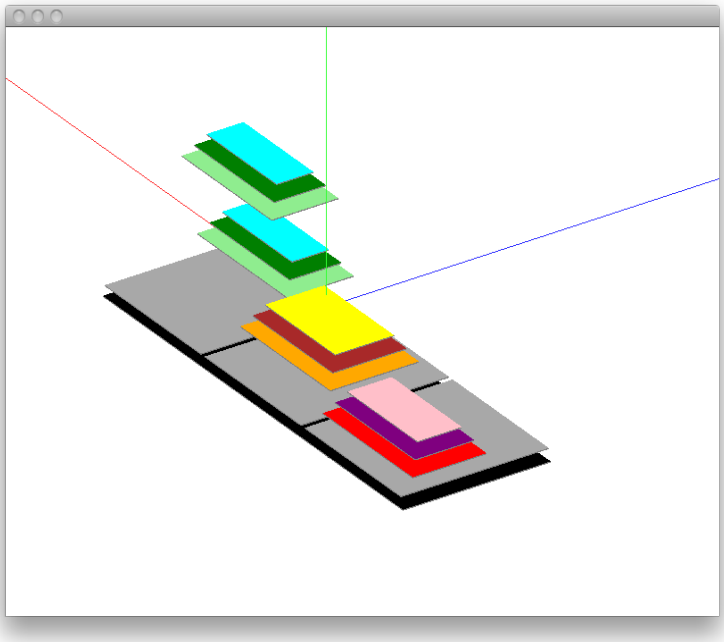

Figure 8. DomScape visualizes a simple web page that contains 16 elements in total

Note that the three-dimensional view is orthogonal - all depth cues other than occlusion have been removed. Elements do not get smaller if they are further in the distance, or bigger when they are right in front of the camera. It is important to preserve the visual likeliness with the conventional view which is two-dimensional and thus not contains any depth effects. When viewed exactly from above the web page looks exactly like it does in the conventional view. The

two-dimensional satellite view is an orthogonal projection of the three-dimensional structure.

User-controlled camera

The transformation from two to three dimensions happens when the user changes the perspective by controlling the position of the camera. As the camera leaves the orthogonal position and orbits around the structure the rest of the underlying structure is revealed to the user (see Figure 9).

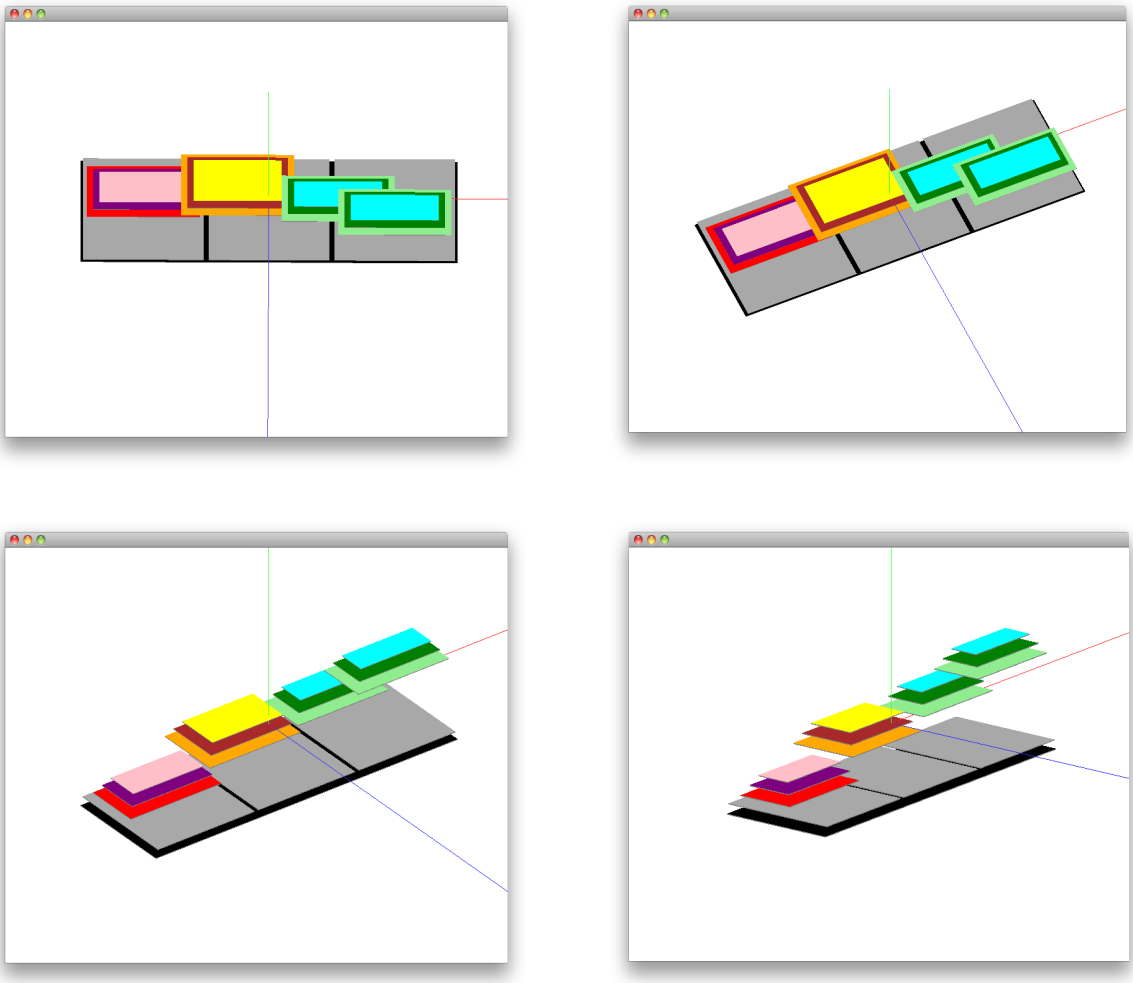

Figure 9. Four image stills of camera motion in DomScape. The top left image displays the web page in two dimensions, because the camera hovers exactly above the structure. The other images show the structure from increasing angles.

The user controls the position and angle of the camera by three distinct actions that offer in total six degrees of freedom:

- Zooming. By using the scroll button on the mouse the user can zoom in and out on the structure. Scrolling up means zooming in and scrolling down means zooming out.
- Panning. The position of the model on the screen can be changed horizontally and

vertically by holding the left mouse button and dragging the structure.

- Orbiting. The angle of the camera relative to the center of the model can be adjusted horizontally and vertically by holding the right mouse button and dragging the structure. Rolling the camera is not possible as this makes navigating the structure more confusing.

Parent-child relationship

The parent-child relationship between elements is a determining factor in the visual representation of the code structure. By default, a child element does not extend outside the boundaries of its parent element unless it is explicitly told to do so. The size and position of an element are thus depends to a large extent on its ancestors and descendants.

For this reason, the relationship between an element and its parent and potential children is visualized. The edges of an element are projected in its parent and the corners of the element and the element's projection are connected by lines to emphasize the parent-child relationship between the element and its parent. All the lines together form a sort of wire frame of the code structure (see Figure 10).

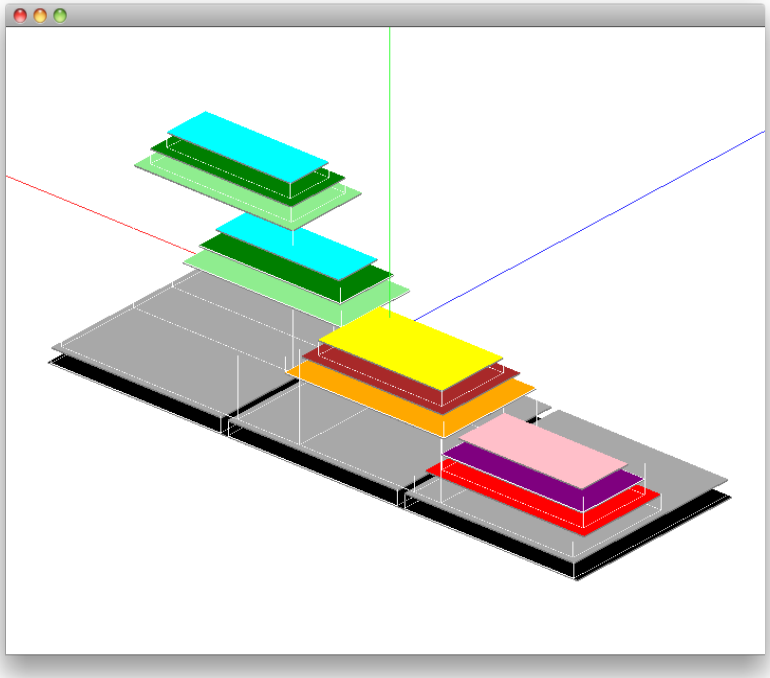

Figure 10. DomScape with lines that visualize parent-child relationships, creating a sort of wire frame for the code structure.

Margin and padding

Elements can be positioned using different CSS properties. In DomScape two of these properties, margin and padding, are visualized. The reason DomScape visualizes these

properties is because they are by far the most common CSS positioning properties. It is a very straight-forward concept that is easily understood and therefore easy to implement.

Margin is defined as the empty area around the *outside* of the element and padding is defined as the area around the edge *inside* the element. This is illustrated in Figure 11. The empty area inside the element, as defined by the padding value of the element, cannot be occupied by a child element (unless it is explicitly told to). The margin on the other hand, defines how much distance the element keeps to its parent element (which it keeps unless it is explicitly told not to).

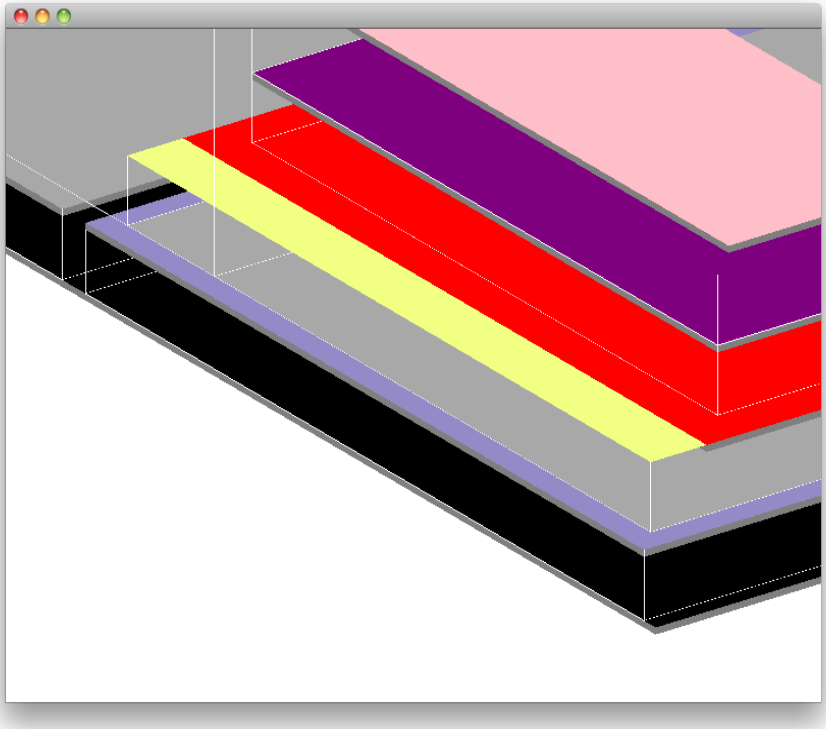

Figure 11. Padding and margin values are displayed by DomScape when the user hovers over an element; in this case the red element. This element has margin value that is visualized by the yellow rectangle. The element's parent, the grey element, has a padding value which is displayed by the thick purple border inside its edges.

Figure 11 shows how DomScape visualizes margin and padding. Margin is visualized with a semi-transparent yellow area. Padding is visualized with a semi-transparent purple area. The choice of these colors is not accidental but instead follows the convention set by the popular web development tool Firebug. DomScape is currently not capable of visualizing negative margin values.

Element selection

When the user hovers the mouse cursor over an element that element gets selected. A selected element is visualized by changing the color of the edges of the element and displaying the

element's margin and padding. When the user hovers over another element, that element is selected and the original element returns to its normal state.

Property pane

An element's properties, such as its width, height, margin, padding and background color are visualized by DomScape. However, it is common that web developers want to know the exact values of those properties. This is provided by the property pane that displays detailed information about the element that is currently selected by the user.

The property pane is not rendered in the three-dimensional visualization but is contained in a separate panel. Ideally, the panel would be integrated with the visualization. Currently however, the CSS properties are displayed by a console window that is positioned next to visualization window (see Figure 12).

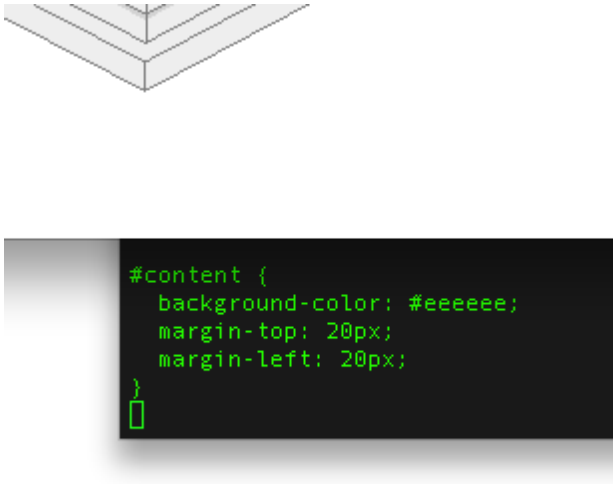

Figure 12. A console window is used to display CSS properties of the selected element. The window is positioned under and below the visualization window.

What are the limitations of DomScape?

In its current implementation, DomScape cannot visualize more than is described in the previous section. Especially relevant limitations are CSS properties that are commonly used to position elements, such as the `float` and `position` properties. DomScape is the first prototype, therefore the scope of possibilities of this implementation are rather limited.

DomScape cannot process HTML and CSS files, but instead uses its own pseudo-code XML-files to store the visualizations. Ideally, DomScape would be connected to a rendering engine, such as WebKit or Gecko, to tap into the HTML and CSS interpretation functionality that such frameworks offer.

Would DomScape work for any web page?

The concept of DomScape relies on the assumption that it is possible to generate a 3D visualization of any web page. This means that all elements in a web page need a value on the z-axis. In order to achieve this for any web page, an algorithm is needed that provides consistent results by using reasonable computational resources.

Besides those two constraints, there are three additional constraining factors that influence the design of the algorithm:

1. Elements cannot intersect in three-dimensional space. If elements overlap in the conventional, two-dimensional view, they should have different z-values.
2. When viewed orthogonally from the top, the 3D perspective should look *exactly* like the web page does in the conventional, two-dimensional view. Elements that are rendered on top in a web page thus have higher z-values than elements that render at the bottom.
3. Elements that are related to each other should be as close together as possible. This likely increases the usability of the tool.

The constraints demand that not just any z-value can be assigned to the elements. The solution to this problem lies within the HTML and CSS specifications, from which an algorithm can be deduced that satisfies all mentioned constraints.

Stacking context

Because of HTML's hierarchical structure elements that are high in the hierarchy often have many descendants. These elements all render in the same location unless they are specifically told not to. A web page of almost any size will thus contain a fair amount of overlap occurring between ancestors and descendants.

Elements can be also be displaced by applying certain CSS layout properties. This can make elements overlap that are not directly related to each other. So, how does the browser know which element is rendered on top? Browsers deal with overlap by employing the stacking context as defined in the CSS 2.1 specification (W3C, 2009). When two elements overlap the stacking context is used to determine what element is rendered on top of the other.

The stacking context consists of seven layers that each contains different kinds of elements. Every element in a web page belongs to one of these layers. Starting from back to front, the seven layers are rendered on top of each other. Elements in layer *n* always render on top of elements in layer *n - 1*.

1. The background and borders of the element that establishes the stacking context
2. Positioned descendants with negative `z-index`
3. Block-level descendants in the normal flow
4. Floating descendants and their contents
5. Inline-level descendants in the normal flow

6. Positioned descendants with `z-index set to auto` or `0`
7. Positioned descendants with `z-index` greater than 0

By default the first stacking context is established by the highest element in the HTML hierarchy, the `body` element or `html` element (dependent on the browser's implementation). If, however, an element is positioned, i.e. if the element is in layer 2, 6 or 7, the element will establish a *new* stacking context for all of its descendants. A web page can, and often does, contain more than one stacking context.

The z-algorithm

The stacking context offers an order of elements, which is non-numerical but can be nonetheless be translated into z-values. Consider the following algorithm that I have constructed.

```
e, f: element

E: collection of all elements
Ei: collection of all elements in the initial stacking context
F: collection of elements with calculated z

assign_z_to Ei

function assign_z_to: context C
source_sort C
z-index_sort C
layer_sort C

for every element e in C:
  z of e = 0

for every element f in F:
 if e overlaps with f
      e.z = 1 + f.z

  add e to F
  if e establishes context C'
    assign_z_to C'
```

In human language the algorithm does the following. It takes all elements in the initial stacking context, established by the `body` element, and sorts it in three ways. The result is that all element are ordered by the stacking layer they belong to; elements that are in layer two or seven are ordered by z-index; elements that are in the same stacking layer and have the same z-index (or no z-index), are ordered by their order in the source code.

The algorithm then iterates through the sorted elements - starting with the first element in the lowest layer - and assigns a z-value to each element. If an element overlaps with an earlier element, the element is given a higher z-value, because a later element should be rendered on top of an earlier element. Whenever an element is in layer 2, 6 or 7 it establishes a stacking context for its descendants. The z-values for this new context are calculated first, before the rest of the current context is assigned. Because the next element in the current context might overlap with the elements of the new context, the new context must be assigned first.

This algorithm provides a consistent and robust approach to make sure that the three constraints are never violated, because all web pages contain at least one stacking context and all elements belong to a certain stacking layer. This guarantees that a three-dimensional model can be created of *any* web page.

How do traditional tools and DomScape compare?

Traditional tools, such as Firebug, have been around for several years and offer a wealth of functionality that is invaluable to any front-end web developer. DomScape does not try to be a replacement for any of these tools, instead it is meant rather as a complementary functionality to these tools. On the other side, DomScape does duplicate functionality that is already available in traditional tools.

The current implementation of DomScape focuses visualizing the box model of a web page. By highlighting parent-child relationships and margin and padding values it attempts to make it easy for the user to understand why elements are positioned the way they are, i.e. the box model.

During the process of ‘theming’ understanding the relationship between code and visualization is crucial. Theming is the activity in which a web developer adopts a web site to a certain layout. The challenge to theming is that the web developer has to achieve that layout with limited or no control over the HTML code. Most or all adjustments are made using CSS. In theming tasks the web developer typically uses a web development tool, such as Firebug, to inspect and understand the relationship between the code and layout.

During the theming process the web developer uses the web development tool in particular way. The following list is an approximation of the atomic actions carried out by the web developer in the process of theming. This is deduced from my own experience with theming work.

1. Focus on a particular section of the web page that needs theming.
2. Select a visible element from that section by using the mouse.
3. Use the visual selector tool, the DOM explorer and the DOM breadcrumb trail to get insight into what elements are part of the section and how they are organized.
4. Keep an eye on the CSS properties to see how each element is styled and how that affects layout.
5. Adjust the CSS on one or more elements to change the layout.
6. When satisfied with the changes, adjust the code on the server to make the changes permanent.
7. Refresh web page to evaluate changes.

The web developer likely iterates through this process, gradually implementing the design with small changes each iteration. If the web developer would use DomScape to do this, the atomic actions are likely as follows.

1. Focus on a particular section of the web page that needs theming.
2. Adjust the perspective of the camera so that all elements of that section are visible.
3. Inspect how the elements are structured by looking at parent-child relationships.
4. Inspect individual elements to understand their CSS properties.
5. Adjust the CSS on one or more elements to change the layout.
6. When satisfied with the changes, adjust the code on the server to make the changes permanent.
7. Refresh web page to evaluate changes.

Because DomScape currently does not allow code adjustments, step 5 does not apply (yet). DomScape also does not visualize certain CSS position properties and lacks many other great functionalities that traditional web development tools offer. However, it is important to note that DomScape in its current form is simply a proof of concept. Only a sub-set of functionality is needed to investigate whether DomScape is an approach worthy of further investigation.

What are the benefits of DomScape?

The big difference between traditional web development tools and DomScape is the fact that DomScape visualizes the code structure and the visual appearance of the structure integrated in three dimensions. Traditional web development tools do not combine structure and visualization but instead display them in separate panels. This section elaborates on the assumed cognitive advantages that such an integration offers.

More accurate mental model

When a web developer works with a web page, he has in his mind an idea of how the code structure looks and how that causes the layout. In other words, he has internalized the box model of the web page. More accurately, the developer creates a mental model of the web page. A mental model is defined as a internal representation of a system or a structure. It is like blue print of how something works (Carroll and Olson, 1987).

When a web developer uses a traditional browser-based web development tool, such as Firebug, to get an overview of all elements involved in a particular section, he has to combine the code structure in the bottom panel and the visualization in the top panel to create a mental model of the page. his can offer benefits if the user must make decisions that also require an integration of the two types of information (Wickens and Hollands, 1999). Web pages consist of two types of information, code structure and its visual appearance, that are very much entangled with each other. If something changes in the code structure, it is likely to change visual appearance as well.

DomScape integrates the two related types of information into one visualization. This makes the visualization more complete, more accurate and better memorizable. Such a visualization is likely to allow the user to internalize a more correct mental model of the web page, which enables the user to make better predictions about potential, untested situations (Wickens and Hollands, 1999).

Higher visibility

In the DomScape visualization every element is visible and distinguishable from other elements. Every element has a unique position in three-dimensional space: even when elements overlap in the conventional, two-dimensional visualization. Because each element is visible and inspectable, it is relatively easy to determine what properties an element has and how the element contributes to layout of the web page.

People have often erroneous mental models of the inner workings of machines and other devices. Norman (1988) argues that this is partly due to a lack of ‘visibility’. A particular device has visibility if one can tell the state of the device and what the possibilities of actions are in that state, by merely looking at it.

I think that the DomScape visualization contains more visibility than visualizations offered by traditional web development tools. First, it provides a more complete picture because it combines two types of information into one visualization. Second, all elements are visible and inspectable. Third, the *entire* structure is visualized. There is no need for scrolling or opening new panels, expanding code blocks: what you see is what you get. I think that for these three reasons DomScape is better at conveying to the user why the web page is laid out the way it is.

Less memory strain

DomScape is capable of *simultaneously* visualizing relevant information about different objects. In traditional tools only one object can be selected and interpreted at the time. Reasoning about

several related objects using a traditional tool, requires the user to memorize the properties of previously inspected elements. DomScape externalizes a lot of that information in the visualization which alleviates the user's memory. Domscape burdens the short-term memory less than traditional web development tools because the developer has to remember less.

As a design principle Bohgard et al. (2009) state that "the design of an interface should not require that the [user] keeps important information active in his short-term memory ... in order to carry out a task." As the short-term memory is burdened with less data, more of its capacity can be used for solving problems. This can be especially relevant when a web page contains a lot of elements, layers (generations of elements) and many complex CSS position properties.

Parallel vs. serial search

The relationship between elements is very important in understanding the code structure and the layout. DomScape allows for the inspection of multiple elements at the same time. This makes it possible to interpret a group of elements and how they relate by merely looking at how the elements are positioned. As the user looks for items with particular properties, it is quite relevant how these properties are visualized and how they can be inspected.

The difference between DomScape and traditional web development tools might be connected to visual search. Basically, there are two types of search: serial and parallel. In serial search each item in inspected in turn, not simultaneously. In a letter-search task, Neisser (1963) observed that there is a linear relationship between the position of an element and the time it takes to find that element. As the number of items increases, the average times it takes to find the item increases as well.

Parallel search on the other hand, can occur when the target item appears to "pop out", which is possible when the target can be defined by a simple rule that sets it apart from other items. Parallel search is generally considered faster than serial search, because it allows more processing to occur simultaneously. Williams et al. (1997) have demonstrated that parallel search is more efficient, as it requires less eye movement than serial search.

Web pages often contain many elements. Traditional tools offer a visual selector that allows users to select elements in the web page on basis of their appearance. As an item is likely to both unique and easily identifiable, this approach can be considered to employ parallel search. However, in some situations the target element is not easily found, and the user has to resort to manually searching through (a particular section of) the code. This search is likely to be serial, because each element must be evaluated on basis of its written properties. In the code panel all elements are similarly visualized and the target item does not "pop out".

In DomScape the need for serial search is circumvented. Because the code structure is combined visually with the appearance of the structure, an element can be always identifiable by its visual properties. The user never has to resort to searching through code. This likely decreases the time needed to find the target element.

Hypothesis

Because of the assumed benefits of DomScape, it is expected that DomScape is faster than traditional tools on certain tasks on certain web pages. This is captured in the following hypothesis:

> DomScape is faster than Firebug for tasks that require the interpretation of CSS properties of HTML elements in structures that contain at least ten levels of overlapping elements.

Instead of comparing DomScape to all possible web development tools, it is easier to make a comparison between a specific web development tool and DomScape. For this reason, I have chosen to compare DomScape to Firebug, the mother of web-based web development tools. Firebug has been around for a long time (Hewitt, 2007) and seems to have the most intuitive interface compared to other similar tools. The hypothesis predicts that DomScape will be faster than Firebug. Speed of performance was chosen as a measurement, because it is easy to determine and simplifies designing tasks.

The hypothesis only predicts an outcome for tasks that require interpretation of CSS properties. This creates a rather limited task scope, but this is necessary considering the capabilities of potential experiment participants. The tasks should be performable by them, because otherwise it is likely too hard to find enough participants to properly test the hypothesis. For this reason, understanding the structure or reasoning about the structure is not included in the hypothesis as this requires a thorough understanding of HTML and CSS, something that not many people have. It must also be noted that due to time pressure it was not possible to invest a lot of resources in finding and selecting participants.

Although DomScape is currently not capable of visualizing complex CSS position properties, complexity can also be expressed in the amount of element overlap that occurs on a web page. It is not uncommon for larger web pages to have a relatively deep code structure with more than ten or twenty levels of elements in certain sections.

In the Method section I describe how I will test the hypothesis.

Method

In order to test the hypothesis formulated in the Introduction, an experiment is performed. In the experiment participants perform tasks in both Firebug and in DomScape. Performances are captured and compared. This chapter explains in detail how the experiment is performed and what choices were made for the design of the experiment, tasks and materials.

Quantitative study

For a variety of reasons I have chosen to test the hypotheses by performing an experiment with quantifiable data. It is common to test new user interfaces on their usability. Potential end-users and experts are interviewed about a prototype to get an indication in what way the tool will be used and where potential pitfalls lie. The goal of this thesis however, is not to assess the usability of DomScape, but to test whether DomScape has certain benefits over traditional web development tools.

A quantified experiment design has certain benefits over a qualitative design. A quantitative is study is likely to be more objective than a qualitative study. In quantitative studies performance is determined with less interference and interpretation by the experimenter, as performance is determined by objective measurements (Hathaway, 1995). Results are generated by statistical measures and thus no evaluative interpretation of raw data is required, leaving no room for evaluator bias. As DomScape is my brainchild, it is obvious that I want it to succeed. However, more important than that is that it is judged on what it is now - not on what I want it to be.

Last but not least, time pressure also influences the choice for a quantitative approach. Analyzing and interpreting raw qualitative data can be very time-consuming. A quantitative study requires less time because participants' performances are distilled into variables, which only require a relatively simple statistical analyse.

Within-subject design

Because DomScape is expert software, not just anybody can participate in this study. Participants are required to have experience with web development. To reduce the number of required participants, the experiment has a within-subject design. In such a design, participants perform tasks in all conditions and thus less participants are needed (Preece et al., 2002).

There are downsides to within-subject design, which are all the result of the fact that participants perform tasks in both conditions. The participant's performance in the first condition likely influences performance in the second condition. However, the learning effect can be largely negated by alternating the order of the conditions between subjects and using different materials (Dix et al., 2003).

There are two conditions and each participant is in both conditions. In the first condition, the participant has to perform a set of tasks using Firebug, a traditional web development tool. In the second condition, the participant performs the same tasks using DomScape.

Minimizing the learning effect

The material has to be different in both conditions, as the learning effect would be very big if the material is the same in both conditions. The material has to be different, but it also has be to similar in order be able to compare the conditions for each participant. If there is too much difference between the materials, there is the risk that the materials influence the results more than the condition in which the material was used.

Half the participants start in the first condition and the other half of the participants starts in the second condition. The material used in the conditions is also alternated. Therefore, independent of what condition the participant is in, half the participants start with material A and the other half starts with material B (see Table 1).

Table 1. Condition and Material counter-balancing

	Condition	**Material**	**Condition**	**Material**
Participant 1	DomScape	A	Firebug	B
Participant 2	DomScape	B	Firebug	A
Participant 3	Firebug	A	DomScape	B
Participant 4	Firebug	B	DomScape	A
etc				

Material

Firebug and DomScape are front-end web development tools that deal with HTML and CSS code. In this experiment dummy web snippets are used that resemble a menu in a web page. A custom snippet was used, instead of a real snippet, because this allows for a tighter control on the properties of the material.

DomScape is in early prototype stages and can therefore visualize only a limited set of CSS properties, namely `margin`, `padding`, `width` and `height`. This severely limits the number of different web pages DomScape can visualize. Because DomScape currently cannot process

HTML and CSS files automatically, creating material in DomScape is very time-consuming. However, by using a small, custom HTML snippets, the material production time could be kept at an acceptable level.

The most important constraint on the material is that it matches the hypothesis, which states that the material must contain at least ten levels of elements. The material must also be complex enough so that it is not too easy to perform the tasks. If the task is too easy, the difference between both conditions will perhaps not be large enough to be significant. The materials should not be too complex otherwise participants might need too much time to perform certain tasks. This could annoy the participant, but also cause schedule issues. For these reasons, a custom web snippet was used, instead of a live production page.

Although the two materials contain the same content, they differ on various levels. Different colors are used, elements have different names, and the structure is slightly different (see Figure 13 and 14). Material A has more items in sub-menu section (“Meet your representative”, etc). Material B has more elements for each normal menu item (“Our products and services”, etc). The way margin and padding are used to space elements differs for each version (see Figure 15).

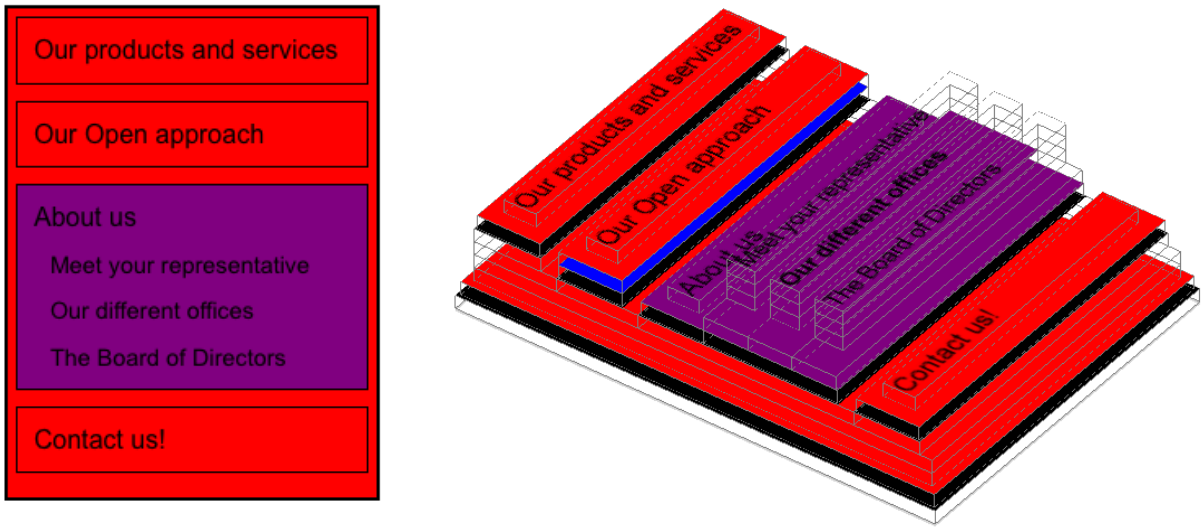

Figure 13. Material A visualized in both conditions, Firebug and DomScape respectively.

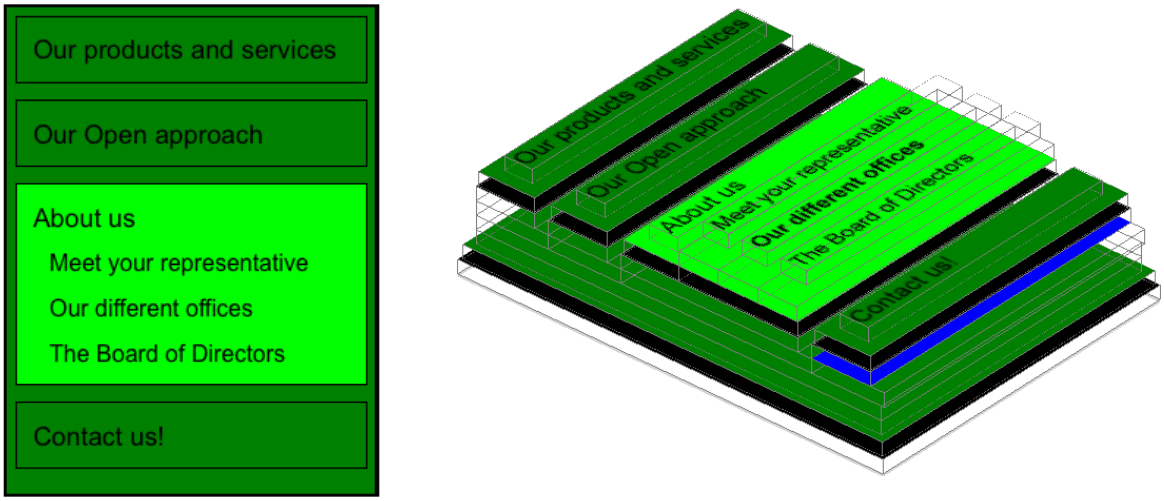

Figure 14. Material B visualized in both conditions, Firebug and DomScape respectively.

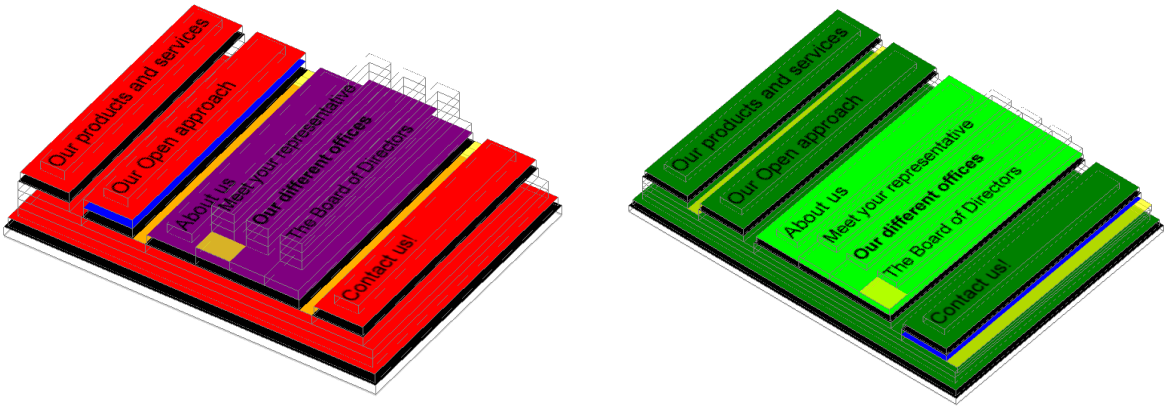

Figure 15. Material A (left) and B (right) have margin values specified for different elements.

Both materials contain many overlapping elements, which is illustrated by Figure 16. Material A contains 32 elements and material B 28. In both materials only seven elements have no overlapping children, the rest of the element is partially or completely overlapped.

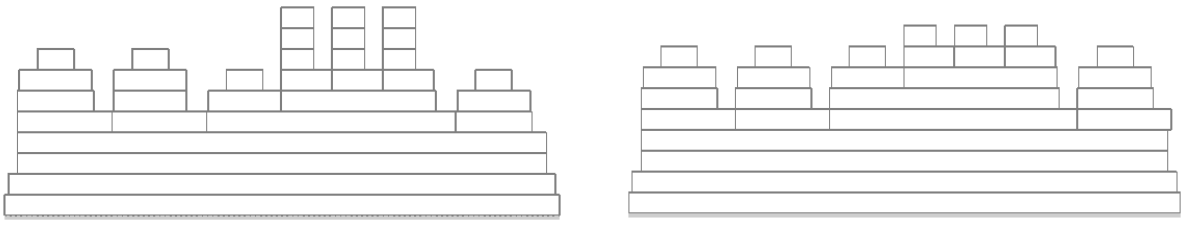

Figure 16. Side projections of the two materials, A and B. The maximum number of levels of elements is 10 and 11 respectively.

Tasks

There are four tasks that are used to test the hypothesis. The tasks all require the participant to interpret the properties of elements. The participant performs four tasks in each condition. Because the material is different for both conditions, some of the tasks also slightly differ in their language. What the task requires the participant to do, however, does not change per condition.

For three tasks performance is measured by the time it takes to complete the task. In one task the performance is based on the accuracy of the participant's answer. Table 2 displays the tasks that the participants performs in both conditions. The tasks are all designed to measure a particular aspect of inspecting and interpreting elements.

Table 2. Tasks that the participant performs in each condition.

	Task	**Performance measure**
1.	Find the overlapped, blue element.	Time to complete task
2.	Estimate how many elements this web page contains.	Accuracy of estimate
3.	Name the four elements that have a purple (A) / dark green (B) background.	Time to complete task
4.	Name all three elements that have a margin.	Time to complete task

The tasks are ordered in such a way that the order is not likely to influence the participant's performance on each task. For example, finding a blue element can be hard when the participant has not seen it yet, but very easy when it has already been found in an earlier task. Finding the overlapped, blue element is therefore the first task.

The tasks are also chosen to resemble situations that web developers normally find themselves in. Although it is unlikely that a web developer ever has to count elements, or that he knows in advance how many elements are responsible for a particular section of the layout, the tasks do require the same atomic actions: distinguishing elements from other elements on basis of its and other elements properties.

Interviews

Participants are interviewed before they participate in the experiment. The interview is used to get an indication of the experience the participants have. The interview consisted of four closed questions: one question about experience with front-end web development in general, two questions about experience with web development tools, and one question about experience with 3D applications, such as 3D games and 3D drawing programs. On each of the four questions the participant could chose from four options that indicated how much experience they have (ranging from no experience to very experienced). The interview has been added as Appendix A to this paper.

The interview also contains a short form of consent that stipulates that the participant participates voluntarily, can leave at any time and agrees that his or her data will be used anonymously in this study. At the end of the experiment an informal unstructured interview is carried out to get an impression of the satisfaction of participant with DomScape. The data for the interview is not analyzed in the results, but is used to interpret the performance of DomScape in this experiment.

Setup

The experiment takes place in a controlled lab environment - a single room that is quiet and only contains the necessary elements to perform the experiment. The room is closed during experiments and windows are obscured to reduce any interference from outside.

The participant sits behind a 23 inch screen with a keyboard and a mouse that are positioned on a normal desk at a normal height. The position of the mouse and its tracking speed are adjusted to the participants needs if necessary.

The experimenter sits on the right-hand side of the participant and controls the screen of the participant. The experimenter launches the application in each condition. Each task is read out loud to the participant by the experimenter. When the participant performs the task, the experimenter keeps track of the time needed to accomplish the task. The performance of each task is recorded on paper.

To help understand the performance of each participant, all screen activity and audio is recorded during the experiments using screen casting software Screenium. This was set up in such a way that it did not interfere with the performance of the applications.

Protocol

The experiment follows a strict protocol to minimize the effect of external conditions. Each experiment takes between 30 to 45 minutes to complete, depending on the speed of the participant.

- Participant comes in, welcome participant
- Ask to sit down, explain what will happen during this experiment
- Set up mouse position and speed
- Display interview which also which contains informed consent
- Explain condition 1
- Perform tasks condition 1
- Explain condition 2
- Perform tasks condition 2
- Informal unstructured interview about what they thought about the interface
- Give chocolate as time compensation, say goodbye

Participants

Participants were recruited inside the Interaction Design department of Chalmers University of Technology. Most participants were students, former students or teachers. All participant were known to the experimenter. The participants were selected on basis of their knowledge of HTML and CSS.

16 people participate in this experiment: 4 women and 12 men. The average age of the participants is 28 years. A little more than half of the participants has some experience with front-end web development, the rest indicates to have a bit more experience and has done web development projects recently. None of the participants consider themselves to have a lot of front-end web development experience.

Half of the participants has experience with stand-alone web development applications such as Adobe Dreamweaver. Six participants have never used a browser-native web development tool, such as Firebug. Two participants have some to a lot of experience and two participants have never used a tool like Firebug. Three participants indicate they have no experience at all with any web development tool.

Although none of the participants has ever used DomScape before, many participants are quite experienced with 3D interfaces such as 3D games and 3D drawing applications. Seven participants indicate they have a lot of experience with 3D interfaces. The other nine participants have got a little to some experience.

Results

In this study participants performed four tasks in two conditions. In the first condition participants used Firebug, a traditional web development tool, to perform the tasks. In the second condition participants used DomScape. The participants' performance in the three tasks was tested by measuring the time to complete the task. In one task participants were asked to estimate the number of elements. The first three tasks tested whether the participants performed faster in the DomScape condition than the Firebug condition. The estimation task tested whether participants could make more accurate estimations in DomScape than in Firebug.

The data that has been gathered during the study has been analyzed by using the t-test for paired samples, which was run for each task. Because the second task is measured with a different unit than the other tasks (percentage instead of time), it was not possible to make an cross-task comparison, by a MANOVA test for example. The participants are also in both conditions, which has to be considered by the test. For this reason it was not possible to use a normal t-test, but instead the t-test for paired samples was applied. The results are summarized in Table 3.

The results mean that DomScape is faster than Firebug on three different time-based tasks. On the first task participants performed the task on average more than six times faster in the DomScape condition. On the second time-based task DomScape was almost two times faster and on the last task DomScape was more than three times faster. These results confirms the hypothesis. Only the estimation task did not result in a significant difference between DomScape and Firebug.

Table 3. Analysed results of the experiment

Task	Condition	Mean	Standard deviation	Degrees of Freedom	t	p
1. Find the overlapped, blue element.	Firebug	65.56 s	50.16 s			
	DomScape	9.75 s	10.72 s			
				15	4.07	**< 0.001**
2. Estimate how many elements this web page contains.	Firebug	36.38 %	30.19 %			
	DomScape	21 %	17.55 %			
				15	1.75	**> 0.05**
3. Name the four elements that have a purple (A) / dark green (B) background.	Firebug	60.81 s	37.03 s			
	DomScape	35.06 s	25.64 s			
				15	3.29	**< 0.02**
4. Name all three elements that have a margin.	Firebug	74.69 s	45.65 s			
	DomScape	21.13 s	11.71 s			
				15	4.07	**< 0.001**

Discussion

In this chapter the results of the experiment are analyzed and interpreted. I focus first on how the conditions contributed to results. Following that, I go into detail about the tasks, materials and software that were used in the experiment. Then I analyze how the differences between participants might have contributed to the results. I conclude the findings by considering the implications for the hypothesis. I finish this chapter with a suggestion for further research.

Conditions

On three out of the four tasks DomScape was faster than Firebug. Only on the element estimation task DomScape and Firebug were not significantly different. In the following section I go into the results of each task, and analyze how the conditions might have contributed to those results.

Condition order

The study was performed with a within-subject design, which means that each participant performed the same set of tasks in both conditions. This creates the risk of a learning effect between conditions: participants are better at a certain task when they do it for the second time. In order to minimize the learning effect, the conditions had been counter-balanced. Half the participants started with the first condition and the other half started with the second condition.

Task 1. Find the hidden element

The first task required participants to find an element with a blue background that was completely overlapped by other elements and thus impossible to see in the two-dimensional visualization. Participants performed the task much faster in the DomScape condition than in the Firebug condition. On average participants were more than six times faster in DomScape, an average performance difference of nearly one minute.

A likely reason why DomScape was faster than Firebug is because participants were required to perform many more actions to complete the task in the Firebug condition. Using Firebug participants were forced to manually select each element in the code structure pane and inspect the element's properties in the CSS properties pane (see Figure 6 in the Introduction). In DomScape only three steps were required to perform the task: change the perspective of the camera, locate the element and read out its name.

In DomScape all elements are visualized. By merely changing the perspective it is possible to view and inspect each element. Whereas in Firebug participants were forced to serially process each element's properties, in DomScape the participant could perceive and process many more

elements at the same time.

The results of this task is an indication that the hypothesis holds. At tasks that require the inspection of CSS properties of overlapped elements DomScape is a much faster tool. It quickly reveals elements and their properties that stay hidden in Firebug.

Task 2. Estimate the number of elements
In the second task participants were asked to estimate the number of HTML elements that the web page contains. In the Firebug condition participants used the code structure pane to count the elements. In the DomScape condition participants changed the perspective so that elements were countable. The experiment showed that DomScape does not perform better than Firebug in counting elements tasks.

One of the reasons for this result might be that it was actually quite hard to distinguish certain elements from each other in DomScape. Horizontally adjacent elements without a margin would appear to be one single element - not separate elements (see Figure 17).

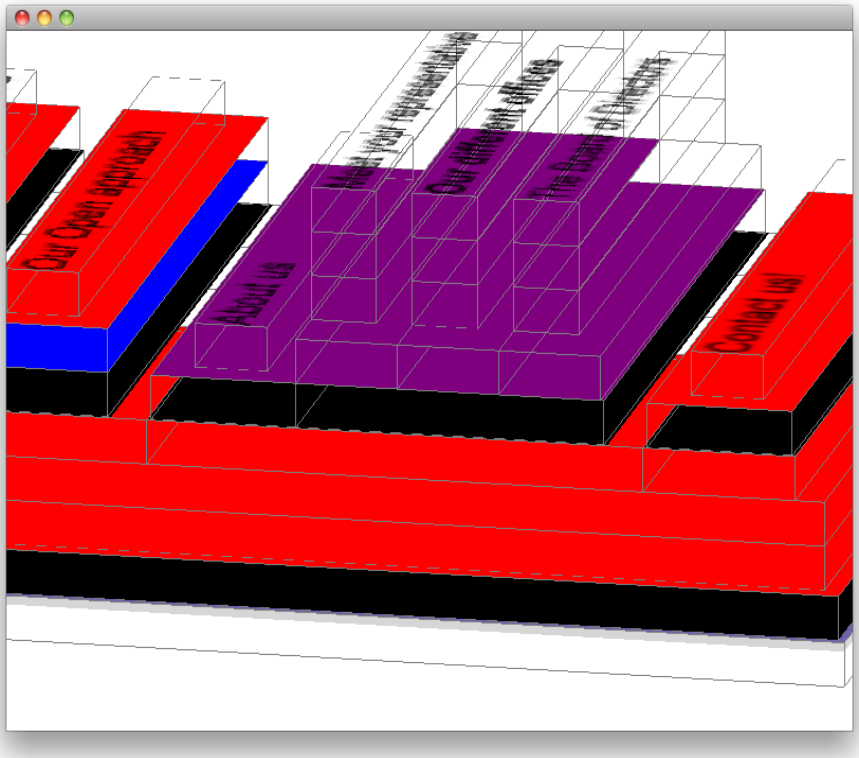

Figure 17. Participants were asked to find the four purple elements. There are four purple elements displayed in this figure, but it is required to hover over with the mouse to determine where each element begins and ends.

The way DomScape highlights the wire frame structure elements might also have contributed to the performance in that condition. The wire frame is constructed by drawing lines from each element to its parent (visible in Figure 17). It was confusing to some participants what to

count: the horizontal panes or the boxes (a result of drawing each element's wire projection in its parent). Both are part of how the elements are visualized, but understandably, to some participants this was not clear.

It is hard to say what the outcome of this task means for the hypothesis. This task was not designed particularly well, which I elaborate on the task analysis in the next section. It is not easy to argue that the results mean anything for the hypothesis, as the task contains so many flaws. However disappointing that may be, the task did prove to be very insightful as it pointed out several of the difficulties participants had while using DomScape.

Task 3. Find the four elements with a background color
In the third task, participants were asked to find and name all four elements with a certain background color (which was different in both materials). The results of this task demonstrate that participants were on average almost two times faster using DomScape than Firebug.

The difference in performance is likely due to the way elements are selected and interpreted in both conditions. In the Firebug condition, participants had to focus on two different and visually separate interface elements at the same time to complete the task. They used the visual selector tool to hover over elements in the visualization pane while at the same time they had to keep an eye on the CSS property pane to inspect the CSS properties of the hovered element (see Figure 6 in the Introduction). In DomScape however, the task required fewer actions. Participants first changed the perspective to see where the elements were located and then hovered over the elements to read out the names of the elements.

As Firebug requires the participant to pay attention to two different panels, one to visually select elements and the other to read out the CSS properties, the participant does not get direct feedback over what element is selected when he hovers the mouse cursor over a certain element. In DomScape participants have to look only in one place to select an element - where the mouse cursor is. Elements that are selected are highlighted. Naming out all four elements required the participant to merely pointing the mouse at the elements that contained the requested background color and reading out the name of the element via the properties pane.

Another reason for the performance difference might be that it is much easier to select elements in DomScape than in Firebug. Because DomScape allows users to change the perspective on the structure and zoom in and out, they control the size of the affordance areas. In Firebug however, some elements are only selectable by a line of one pixel wide. It is thus likely that it easier to select elements in DomScape and that this also contributed to the difference in performance between the two conditions.

Task 4. Find the three elements with a margin
In the fourth task, participants were asked to find and name all three elements with a margin. The experiment demonstrated that participants were on average more than three times faster in the DomScape condition than in Firebug condition.

The first, most obvious explanation of this result is that it was possible to ‘cheat’ in DomScape. Participants were able to highlight all elements with a margin by pressing the ‘m’ key on the keyboard. This allowed participants to instantly see which elements had margin (see Figure 18). All that was required then was to hover over the elements to read out the names.

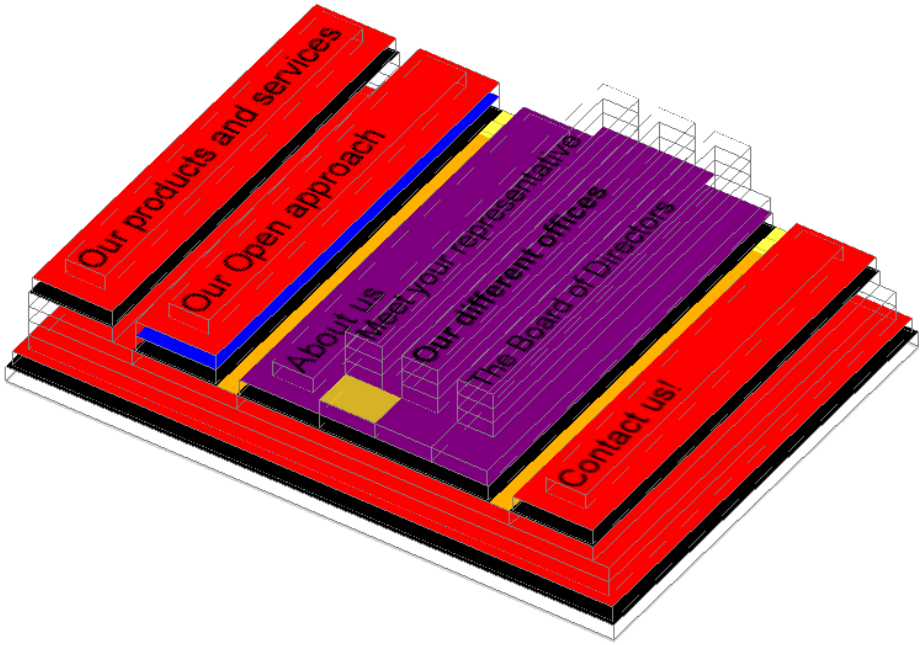

Figure 18. DomScape visualizing the margins (yellow)

Such functionality is not available in Firebug. Actually, Firebug only *visually* displays margin and padding properties when the mouse is hovered over the elements in the bottom code structure pane. The properties are *not* visualized when the mouse is hovered over the web page. The participants could thus not use the mouse to visually select elements, but instead were forced to pay attention to two different panels: the bottom code structure panel to select elements and top panel to visually inspect the properties of the selected element.

A second interpretation of the performance difference comes from the observation that two participants forgot that they could use DomScape to highlight all the margins. They both still performed faster in the DomScape condition. This suggests that perhaps only part of the performance difference on this task is due to the margin-highlight functionality of DomScape. As covered in the discussion of the third task, two other reasons might also contribute to why DomScape is faster in these kind of tasks. First, it is easier to select elements, because the areas of affordance are larger in DomScape. Second, because the structure and its appearance are combined into one visualization the participant only needs to focus on one single location to evaluate the elements.

A third reason why DomScape is faster might be that DomScape offers a more visual interface than Firebug. Because the entire code structure is visualized and the participant can navigate that structure visually, it is perhaps easier for the participant to remember which elements he has already inspected and which elements not. In Firebug elements are represented in plain

code and bear barely any visual distinction from each other. Navigating the code structure in this task is likely harder using Firebug than DomScape, which may have contributed to the slower performance of Firebug.

Besides the more memorable visual appearance of elements, DomScape visualizes the *entire* code structure. In one view the participant sees the whole structure. The benefit of this is that the participant can see all the possible selectable items. In Firebug, only part of the code structure is visualized in the code structure pane. There is simple not enough space to display more than a couple dozen of elements. To save space, if an element contains any child elements, it is collapsed by default. Viewing the entire code base thus also requires a number of mouse clicks to unfold collapsed elements. It is not surprising that the task takes longer in Firebug considering the amount of actions it requires to perform the task.

In retrospect, it is a pity that participants were allowed to 'cheat' in DomScape by using the keyboard shortcut to highlight all elements with margin. It makes the task so much easier that that it is the most obvious explanation for the performance difference. It is impossible to determine if that caused all the difference, but considering the performance difference in the rather similar third task, it is likely that other factors played part in it as well.

Conclusion

The performance difference between both conditions can be interpreted to be the result of three main benefits of DomScape over Firebug.

First, because there is vertical spacing between elements, they are better distinguishable from each other. Because elements are better distinguishable from each other, it is also easier to *interpret* what the properties of each element are and how the element contributes to the two-dimensional visualization.

Second, because DomScape visualizes with vertical spacing between elements, it is relatively easy to *select* that element. Sometimes, only a few pixels of an element are visible in the two-dimensional visualization. Using Firebug to select that element, requires precise mouse control. In DomScape however elements have larger areas of affordance, that make it easier to select an element. Because DomScape has zooming functionality the area of affordance can be increased if that is required, which is demonstrated by Figure 19.

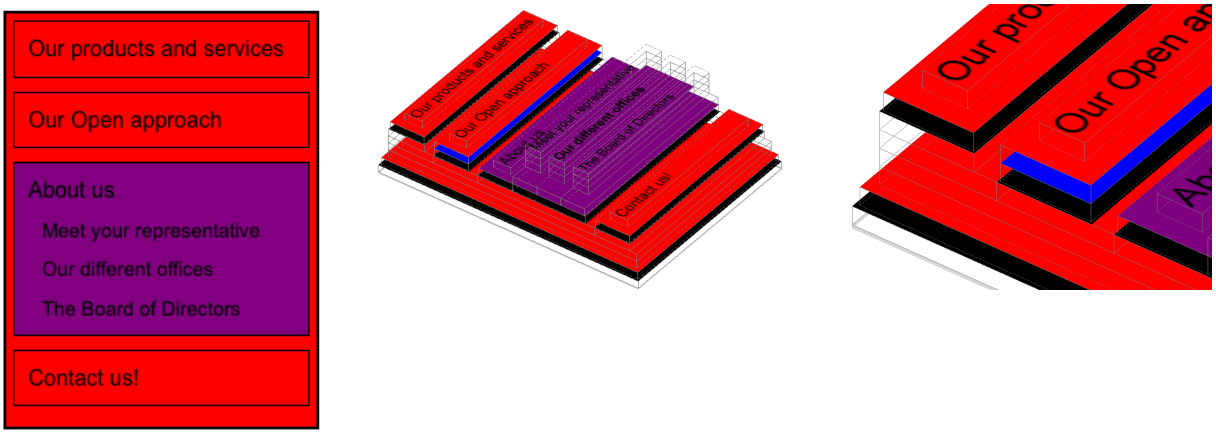

Figure 19. On the left the normal two-dimensional visualization, used by Firebug, offers minimal affordance when selecting certain elements, such as the element with a black background (hat cause the black borders). In DomScape the areas of affordance are much larger. Using the zooming functionality, as demonstrated on the right, the areas become as large as the user requires.

The task material contained elements that were in close vicinity to each other, which makes selecting elements challenging. The experiment showed that participants got confused when the mouse was moved a bit too far, another element would get selected and the code structure pane and the CSS properties pane ‘jumped’ to the new element. When that happens the user looses the context of the code structure and has to reposition the mouse until the right element is selected again. It is quite an unforgiving user-experience, when a wrong element is selected and the content of two panes complete change.

Third, in Firebug the code structure is visualized in a separate panel. This creates two places of interest for the user. In DomScape those two visualizations are combined into one, which creates a smoother experience. This means that there is no such thing as ‘jumping’ code visualizations. Because the code structure and the visual appearance of the elements are visualized together, both automatically stay in context with each other. The user focuses on one part of the interface, without having to divide his attention to visually separate locations in order to select and evaluate an element.

Tasks

This experiment contained four tasks. In this section I will analyze to what extent the tasks contributed to the results.

Order of tasks

The four tasks were performed in the following order (same for both conditions):

1. Find the hidden element.
2. Estimate the number of elements.
3. Find the four elements with a background color.
4. Find the three elements with a margin.

The task order was chosen in such a way that earlier tasks would minimally influence later tasks. The hidden element task therefore had to come first, because it would skew the results if participants would have been able to locate it in an earlier task. The estimation task was second because it was assumed that the more the participant would work with the material, the better he would become at this task.

In the third task participants were required to find four elements with a certain background color. This task was perhaps the most under influence of the effect of task order and it is likely that this increased the performance on the task in Firebug, but even more so in DomScape. The chance that the participant would notice an element having a certain background color in Firebug is likely smaller than in DomScape. In Firebug the participant needs to actually notice the CSS properties and then (subconsciously) remember the position of the element in the code, or the name of the element, to be able. In DomScape however, the participant only has to remember a general visual location in the code structure.

The fourth task suffers less from the effect of task order. Because margin is only visualized when the participant hovers over elements, the participant has a much shorter exposure to the position of the margin and the element that causes it. It is possible that during the performance of previous tasks, the participant has been exposed to at least one or more elements that contain margin. However, there is no reason to assume that this affected one condition more than the other.

Task 1. Find the hidden element

The first question that arises when analyzing the first task is to what extent it is relevant that a web development tool reveals completely hidden elements. How often do web developers perform such tasks in real-world situations? Not often, I would say, but there are variations on the same task that are actually quite common.

For example, consider the following situation. A developer has to change the background color of a certain section of a web page. What the developer does not know (yet) is that two overlapping elements, with the same background color, are responsible for the background

color of the section of interest. If the web developer wants to change the background color of the section, both elements need to be changed. Changing only one of the elements, would not change anything in the two-dimensional perspective.

Using Firebug, it initially seems to web developer as if only one element causes the background - there is no reason to assume otherwise. Upon changing that element's background color, the web developer notices that the background color has not changed at all. Apparently, another element in the same section also has a background color. The web developer would use Firebug again to find and change that element. If the web developer had been using DomScape, it would be much easier to interpret that there are two elements involved.

This again might seem like a trivial problem, and it might actually be, but the important thing to note is that these kind of problems are common when a web developer has to work with code that is not his own (nor of somebody else, e.g. when the code is automatically generated). Issues like these are relatively easy to solve, but in the end can add up to be unnecessarily time-consuming. This is why the task actually is relevant, although it does not seem so at first.

Task 2. Estimate the number of elements

If the first task raised any doubts about relevance, then the second task must sure do as well. How often is a front-end web developer interested in the number of elements in a web page? The answer is most likely: almost never. It is rarely important for a web developer to know how many elements a web page contains. That said, perhaps it should be important, because more elements increase the download time, the browser rendering time and the javascript execution time (Yahoo!, 2010).

It is obvious in this task and that complaint could be made about every task in the experiment. The goal in this experiment was not to evaluate DomScape in real world scenarios, but rather scenarios that would test the hypothesis. However, it must be admitted that even at that this task failed.

The hypothesis states that "DomScape is faster than Firebug". The way the task was designed, it fails at testing speed, but rather measures accuracy. The hypothesis also states that the task should "require interpretation of CSS properties of elements". It is hard to argue that estimating the total number of elements in a page is the same as inspecting and interpreting the properties of those elements.

The task for the participants was actually to make an *estimation* of the total number of elements. However, because the time available for the task was long enough most participants managed to *count* most of the elements. In Firebug, if the participant would have had less time, he would be forced to make an estimate on basis of incomplete information. It was assumed that because DomScape visualizes every element, the participants' estimates would be more accurate in that condition. Were the participants given less time, so that they were forced to estimate, the results might have been different.

Another issue with this task was that the DomScape explanation material did not properly explain what exactly an element is and how it is visualized. In the explanation material all elements had a background color. However, in the actual task material some of the elements had no background color and were therefore only visualized by the wire frame. This confused some participants as they had to adjust their understanding of what exactly an element is and how it is visualized.

The second task was likely the most flawed task of the experiment. It fails to say anything about the hypothesis, allowed participants to count instead of estimate and did not generate a significant difference. But, however flawed the task was, it did provide valuable insight into some of the issues with the current implementation of DomScape as highlighted in the evaluation of the conditions.

Task 3. Find the four elements with a background color
In this task, participants were required to explain why a certain section of the web page had a certain color. To make it easier for the relatively inexperienced participants, it was revealed that four elements contributed to the background color of that section. This task produced interesting results, because it demonstrated that DomScape is faster than Firebug, even though it is quite easy to perform the task in the latter as well.

This task was perhaps the closest to a real-world scenario. During front-end web evelopment work developers often have to deal with HTML and CSS code that has been generated by third party software and developers. It is quite common that such code is sub-optimal: it can contain inconsistent styling, duplications of CSS code and many redundant elements. Styling the web page, by altering the CSS, can therefore be difficult, as it is hard to understand code structure that is created by others.

The task does not seem to favor one condition over the other. If the task would have been closer to a real world scenario, the difference in performance between the two conditions might have been even larger. For example, if the amount of elements were not given in the task, this would make a minor difference to the performance in DomScape. Because in DomScape all elements are visible, it is relatively easy to deduce how many and what elements contribute to the background color. In Firebug however, the participant cannot interpret each element in the whole structure easily, and would thus have to deal with more uncertainty. This could result in a lower performance: more errors (not naming all elements that contribute to the background) and slower performance (the participant checks more elements in order to be sure about the answer).

Task 4. Find the three elements with a margin
In the fourth task, participants were asked to find and name all three elements that had a margin defined. Although it is not common at all for front-end web developers to have to count elements with a margin, the task does require participants to perform a very common sub-task of web development: inspecting elements.

In order to decide whether an element contains a margin participants had to go through a set of actions in each condition. What the task managed to highlight is how many actions are involved to perform the task in the Firebug condition. There is no reason to assume that the task favors one condition over the other.

Normally, a web developer would be interested in knowing why there is a certain amount of spacing between two sections on the web page. However, the task had to be simple enough so that even relatively inexperienced participants could perform it. If the task would have been formulated so that it would require the participant to interpret the relationship between elements, it would have been a task that is closer to real-world situations. The focus on interpreting the relationship between elements would be very interesting, but it would also test a different hypothesis than has been formulated for this study.

Materials

To make sure that the learning effect was a small as possible two materials were used and they were counter-balanced over the conditions. Half the participants used the first material in the first condition, and the second material in the second condition. For the other half of the participants, it was exactly the other way around.

The hypothesis states that DomScape outperforms Firebug on tasks that require the inspection and interpretation of CSS properties of elements in *structures that contain at least ten levels of overlapping elements*. The materials contained 10 and 11 levels of overlapping elements. This is a lot for such a small page, but it is not so many to suspect that it explicitly benefited DomScape. It is quite common for web pages to contain many overlapping elements in certain sections, such as menus.

The materials were meant to be equally challenging, but the experiment showed that this was not the case. In the first task, the hidden element could be found much easier when material A was used in the Firebug condition. In that condition, participants were forced to inspect each element in the code until the hidden element was found. The strategy most participants adopted was to start at the top of the code and then progress downwards. This strategy was much more successful with material A than B, because in material A the element is hidden on the top of the page and in material B it is hidden at the bottom. This illustrated by Figure 20.

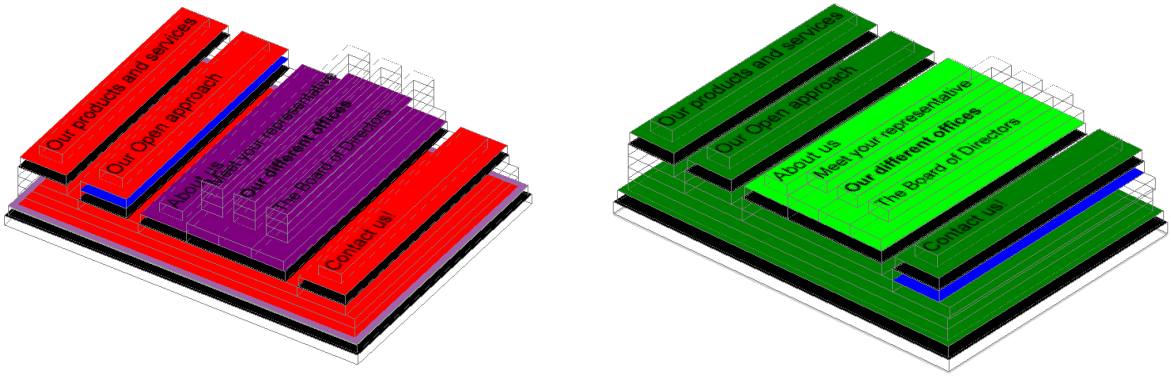

Figure 20. The hidden, blue element is positioned differently in each material, A and B respectively.

Another issue with this material was that padding was visualized purple. This confused some participants that mistook the padding for the blue element. This occurred only in the DomScape condition.

In the second task, the participants had to estimate the number of elements in the web page. In the Firebug condition, the participants used the code structure to count the elements. In the DomScape condition, it was harder to count the element because adjacent elements are hard to distinguish and appeared to some participants as being one single element. If the materials contained more spacing between the elements, it likely would have benefited the DomScape condition.

In the third task participants were asked to find all four elements that had a had a purple (material A) or dark green background color (material B). Figure 20 shows that this is much easier to determine in material B. In material A the purple elements were much closer together. All four all elements also border directly with one of the other three elements. This made it very hard to distinguish the elements. In material B the elements are farther apart, which made it easier to find these elements. Participants had issues with material A in both the DomScape and the Firebug condition, suggesting that this did not influence the results on the third task.

In the fourth task, which elements had margin differed for each material. However, none of the materials was harder in this respect. Both were equally challenging in either condition.

Software

During the experiments software issues occurred in both conditions. Firefox, the browser in which Firebug runs, became slow after using it for several consecutive hours. I only realized this after it happened the first time, upon which I restarted the browser to alleviate the slow performance of Firefox. During the rest of the experiment I kept an eye on the performance of Firefox and preemptively re-started Firefox several times. However, it is quite possible that a

number of participants had a less optimal performance because of the performance of Firefox. However, considering the nature of the tasks and relatively small performance deterioration it is unlikely that it had a large effect on the results of the experiment.

The DomScape prototype has a camera issue which likely has influenced the performance of the first two participants on the first task. The start position of the camera in DomScape is not exactly from the top. This revealed the position of the hidden, blue element even before the participants had started looking for it. In subsequent experiments the issue was taken care of by manually adjusting the camera so that the blue element would not show beforehand (see Figure 21). Considering the rather large difference in performance of the two conditions, it is not likely that this issue effected the results to a large extent.

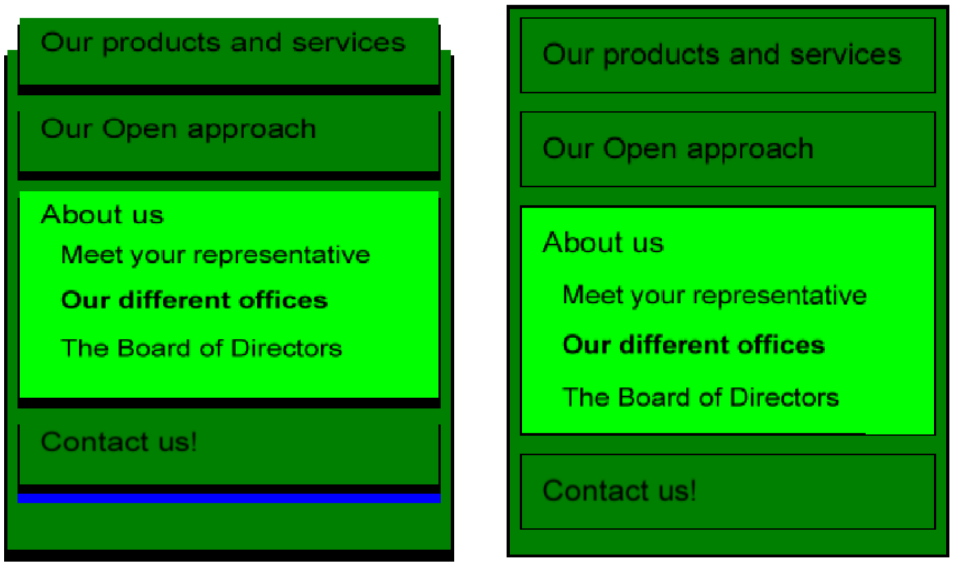

Figure 21. Two starting positions of material B using DomScape. On the left is the starting position displayed as it was used with the first two participants. It clearly reveals the location of the ‘hidden’ blue element. On the right it shows the adjusted starting position as it was used with the rest and majority of the participants.

Participants

Participants were recruited inside the Interaction Design department of Chalmers University of Technology. Most participants were students, former students or teachers. All participant were known to the experimenter.

Participants were selected on the basis of their HTML and CSS knowledge, but most participants also had experience with web development applications. Half of the participants had used stand-alone web development applications such as Adobe Dreamweaver before and two-thirds had used a browser-based web development tools, such as Firebug, in the past. Only three participants indicate they had no experience at all with any web development tool.

Most of the participants were thus experienced with Firebug. The results would likely have been more in favor of DomScape if only participants were selected that had no experience with any web development tool. However, these kind of participants are very hard to find, as most people that have knowledge of HTML and CSS, also have used web development tools.

Although none of the participants had ever used DomScape before, many participants were quite experienced with 3D interfaces such as 3D games and 3D drawing applications. Seven participants indicated that they have a lot of experience with 3D interfaces. The other nine participants indicated they have a little to some experience. This is probably due to the population from which the participant were selected: Interaction Design naturally attracts people that have used 3D drawing programs or like to play computer games or even develop them in the Interaction Design Game Design track.

It is quite possible that the participants in this experiment are more familiar with 3D interfaces than the average front-end web developer. Navigating a 3D camera can take some time to get used to. It is thus likely that experience of participants influenced the performance of the DomScape condition.

It is a possibility that participants (subconsciously) tried to favor the participant by trying a bit harder in the DomScape condition than in the Firebug condition. However, there were no signs during the experiments that suggest that this has happened. Participants also complied with a set of rules for the experiment, one of which explicitly mentions that the participant does not try to influence the outcome of the experiment in any way.

Implications for the hypothesis

In this study an experiment was performed to test the following hypothesis.

> DomScape is faster than Firebug for tasks that require the interpretation of CSS properties of HTML elements in structures that contain at least ten levels of overlapping elements.

The experiment consisted of four tasks that were performed in each of the two conditions. On three out of four tasks DomScape was notably faster than Firebug: finding a hidden element (almost six times faster), finding elements with a certain background color (almost two times faster) and finding elements with a margin (more than three times faster). On the element estimation task there was no difference measured between DomScape and Firebug.

As has been pointed out in this chapter, there are many factors that might have contributed to the performance differences between DomScape and Firebug. The tasks, materials and software used all contained anomalies that make the results harder to interpret. However, the performance difference between DomScape and Firebug is quite large. It would be hard to argue that these results are mainly due to the issues with tasks, materials or software.

I have also argued that three of the four tasks are relevant to the hypothesis. The estimation task was quite flawed and seems irrelevant to the hypothesis. On basis of the relevance of the tasks and the results that the tasks produced, I conclude that the hypothesis holds: DomScape is faster than Firebug on tasks that require inspection and interpretation of CSS properties of HTML elements in structures that contain many overlapping elements.

What is next?

What do these results mean for DomScape? First of all, this study has demonstrated that DomScape is more than just an idea - it actually has merit. DomScape performed faster than Firebug on several tasks. Although the results look promising, because DomScape is such a new concept, there are a lot of uncertainties surrounding it. Future research will be needed to further investigate how DomScape can help the web developer understand the relationship between the code and the presentation of a web page.

The experiment in this study has focused on the inspection and interpretation of the properties of elements. It did not say anything about interpreting the relationship between elements. One of the assumed benefits of DomScape however, is that it allows the user to understand the structure more easily. It would be very interesting to investigate whether participants are not only faster at finding, selecting and interpreting elements, but also if they are better capable of understanding the relationship between elements and how the elements together give rise to the layout.

A better understanding of the cognitive processes that are involved during the usage of DomScape is also required. Research that focuses on perception, memory and reasoning can provide valuable insights. It will be especially interesting how those findings could be used to improve the application. Interesting will be what the effect is of the top-down approach of DomScape. To what extent is the performance of web developers influenced by the mental concept they have of a web page. How does DomScape contribute to that?

The experiment has already revealed a number of points for improvement. For example, some participants clearly struggled with adjacent elements that were not distinguishable enough. Participants have suggested that the 3D navigation would improve if the center-point of the camera could be adjusted. It was also suggested that DomScape could be used not only to interpret existing web pages, but also to build new ones. It would also be interesting to see whether DomScape could visualize of how CSS selectors apply to the code. Other suggestions were: mouse gestures and mouse acceleration, better depth cues such as shadow casting from child elements on parent elements, the ability to focus on particular sections of the web site (and blur out the others), better visual integration by displaying property panels inside the visualization.

One of the down sides of the current implementation, is that DomScape currently does not work

with real web pages. It would be very interesting to see how the three-dimensional visualization holds up on real data. Does it still make sense? In its current implementation DomScape only visualizes a marginal subset of CSS properties. Will it be able to visualize structures that use certain complex CSS positioning properties such as the position, float and overflow attribute?

Usability research is much needed as well. DomScape is an entirely new way of visualizing HTML code structure. Because it is such a new concept, there are many interface possibilities (good and bad) to explore. The design space is very big and largely unexplored. DomScape has only touched the surface of the possibilities that lie in visualizing web pages in three dimensions. Above all, I think that DomScape has demonstrated that unconventional web page visualizations are an interesting and promising topic of investigation.

Conclusion

This study has investigated DomScape, a new approach to visualizing web pages. DomScape visualizes any web page as a three-dimensional structure by combining the code with the presentation. The resulting visualization is complete, accurate, and visual. The user can navigate the visualization by controlling a camera that orbits the structure. Through zooming, panning and orbiting, the user can easily change the perspective, focus on particular sections or take a distant perspective to view the structure as a whole.

To test how DomScape compares to traditional, two-dimensional front-end web development tools, an experiment was performed. In one condition participants used DomScape and in the other Firebug, a popular browser-based web development tool. There were four different tasks and DomScape produced a faster performance in three of them. This let me to conclude that the hypothesis was confirmed: DomScape is faster than Firebug at tasks that require inspection and interpretation of CSS properties of HTML elements in structures that contain at least ten overlapping elements.

The reason DomScape performed faster than Firebug is assumed to be due to different reasons:

- Every element is clearly and distinctively visualized. This makes it easier to understand the relationship between each element and structure as a whole and how an element contributes to the layout.
- Multiple elements and their properties are visualized simultaneously. This is less burdensome on the participant's memory. As the short-term memory is burdened with less data, more of its capacity can be used for solving problems.
- It is easier to select elements, because the user can control the size of the areas of affordance.
- Because DomScape combines code and presentation, there is only one point of focus for the user. DomScape requires fewer user actions for the same tasks.

This study was conducted to test how DomScape compares to traditional web development tools. It has been demonstrated that DomScape has benefits over Firebug in certain situations. Future research will have to test how DomScape holds up in real-world scenarios. More research is also needed to get a better understanding of why DomScape works, from a cognitive perspective. Usability research will be essential to building an application that suits the needs of its end-users.

References

"Berners-Lee, Tim; Cailliau, Robert (November 12, 1990). "WorldWideWeb: Proposal for a hypertexts Project". Retrieved October 8, 2010. http://www.w3.org/Proposal.html

Bohgard, M., Karlsson, S., Lovén, E., Mikaelsson, L., Mårtensson, L., Osvalder, A., Rose, L. & Ulfvengren, P. (red.) (2008) *Work and technology on human terms.* Stockholm: Prevent

Carroll, J. M., & Olson, J. R. (1987) *Mental models in human-computer interaction*. Washington, DC: Natinol Academy Press.
Norman, D. A. (1988). *The psychology of everyday things*. New York: Harper & Row.

Dix, A., Finley, J., Abowd, G., & Beale, R. (2003) *Human- computer interaction (3rd ed.)*. Upper Saddle River, NJ: Prentice-Hall.

Hathaway, R. S. (1995) Assumptions underlying quantitative and qualitative research: implications for institutional research. *Research in Higher Education* 36(5):535–562.

Hewitt, J. (2007) Ajax Debugging with Firebug. *Dr. Dobb 's Journal*, 393:22-26.

Hollands, J. & Wickens, C. D. (1999). *Engineering Psychology and Human Performance*. Englewood Cliffs, NJ: Prentice-Hall.

Internet History One-Page Summary. http://www.livinginternet.com/i/ii_summary.htm Retrieved, October 8, 2010.

"Inventor of the Week Archive: The World Wide Web". Massachusetts Institute of Technology: MIT School of Engineering. Retrieved October 8, 2010. http://web.mit.edu/invent/iow/berners-lee.html

Komenda, Klaus. “Collection of Web Developer Tools, per Browser” February 16, 2008. http://www.klauskomenda.com/archives/2008/02/16/collection-of-web-developer-tools-per-browser/

Meyer, Eric (2006). “CSS: The Definitive Guide”, Third Edition. O'Reilly & Associates

Neisser, U. 1963. Decision time without reaction time. *American Journal of Psychology, 76,* 376-385.

Petrie, Charles; Cailliau, Robert (November 1997). "Interview Robert Cailliau on the WWW

Proposal: "How It Really Happened."". Institute of Electrical and Electronics Engineers. Retrieved 18 August 2010. http://www.computer.org/portal/web/computingnow/ic-cailliau

Preece J., Rogers Y., Sharp H. (2002) Interaction Design - Beyond Human Computer Interaction - 2nd Edition. Wokingham: Addison-Wesley

Raggett, Dave (1998). Raggett on HTML 4. Retrieved October 7, 2010. http://www.w3.org/People/Raggett/book4/ch02.html

"Tags used in HTML". World Wide Web Consortium. November 3, 1992. Retrieved October 8, 2010. http://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Tags.html

W3C (2009). “Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification”. http://www.w3.org/TR/CSS21/ Retrieved October 11, 2010.

Williams, D. E., Reingold, E. M., Moscovitch, M., & Behrmann, M. (1997). Patterns of eye movements during parallel and serial visual search tasks. *Canadian Journal of Experimental Psychology, 51,* 151-164.

World Internet Usage Statistics. Retrieved, October 8, 2010. http://www.internetworldstats.com/stats.htm

Yahoo!. Best Practices for Speeding Up Your Web Site. Retrieved November 5, 2010. http://www.w3.org/People/Raggett/book4/ch02.html

Appendix A: Interview

Participants were asked to fill out the following questions before they participated in the experiment.

A couple of questions regarding this experiment

This form contains a couple of basic questions that give an indication of your level of experience.

Please note that if you fill out the form you agree with the following:
1. the computer input (mouse movement and keyboard strokes) and audio are recorded
2. the data from this experiment will be used anonymously
3. you are free to leave the experiment at any time
4. you are rewarded with chocolate for your time :)

* Required

What is your first name? *

What is your gender? *
- ○ Male
- ○ Female

What is your age? *

How much experience do you have with front-end website development? *
- ○ 1. No experience at all
- ○ 2. Little experience, e.g. did a bit web development
- ○ 3. Some experience, e.g. have done some projects
- ○ 4. A lot of experience, e.g. have been working on front-end web development projects recently

Have you ever used any of these web development tools? *
- ☐ Browser plug-in, such as Firebug or the native plug-ins in Chrome, Safari, IE or WebKit
- ☐ DreamWeaver, Coda, or any other stand-alone, front-end web development application
- ☐ Other:

How experienced do you consider yourself with web-development browser plug-ins? *

Such as Firebug or the native plug-ins in Chrome, Safari, IE or WebKit
- ○ 1. No experience at all
- ○ 2. Little experience, e.g. used it a couple of times
- ○ 3. Some experience, e.g. have done some projects with it
- ○ 4. A lot of experience, e.g. have been working a lot with it (recently)

How experienced do you consider yourself with 3D interfaces, such as 3D games? *
- ○ 1. No experience at all
- ○ 2. Little experience, e.g. played a couple of 3D games
- ○ 3. Some experience, e.g. played 3D games recently
- ○ 4. A lot of experience, e.g. created own 3D games, plays a lot of 3D games, etc