Automated Webpage Evaluation

Ryan Tate, Gregory Conti, Edward Sobiesk
Army Cyber Center
U.S. Military Academy
West Point, New York 10996 USA
ryan.tate@usma.edu, gregory.conti@usma.edu, edward.sobiesk@usma.edu

ABSTRACT
Webpage evaluation and metrics have historically focused on page-level characteristics or on key words. We introduce an automated technique for graphically measuring specific elements on a webpage. Our technique provides a means to increase the fidelity of webpage analysis and introduces a novel metric focused on the number of pixels that certain elements occupy in a browser window. We implemented the technique as a Firefox extension and successfully tested it on Alexa's top 25 U.S. websites. The technique is fully automatable and consistently measures a customizable set of elements as they appear to users in the Firefox web browser. Importantly, the application allows for communication with and the incorporation of other browser-based tools or extensions. We discuss design considerations and creative solutions to technical implementation challenges. The application provides for a wide range of research opportunities that may require a new level of fidelity in webpage analysis and comparison.

Categories and Subject Descriptors
D.2.8 [**Software**]: Metrics - *product metrics*.
H.3.5 [**Information Storage and Retrieval**]: Online Information Services – *data sharing*, *web-based services*.
K.6.5 [**Management of Computing and Information Systems**]: Security and Protection - *invasive software.*

General Terms
Measurement, Security, Human Factors, Standardization

Keywords
web measurement, web content analysis, interfaces, interface design, user experience, webpage analysis

1. INTRODUCTION
Research techniques for evaluating webpage content on the World Wide Web (WWW) usually focus on the entire page, on metadata, or on key phrases. As webpages become more complex and increasingly integrate content from multiple sources, there must be reliable and automated means to measure specific HTML elements or content categories within the page. Such a capability can assist researchers and organizations in learning about common patterns within webpages, similarities of particular content embedded within webpages, usage characteristics, invasive advertising, the correlation of specific content with various rankings, and other research questions at a finer level of detail than simply considering the entire webpage, specified metadata, or key word searches.

This paper introduces an automated technique we developed for graphically measuring specific webpage content at a (customizable) granularity below page level. Our technique is implemented as an extension that runs in the Mozilla Firefox web browser. It is fully automated and works consistently on most popular webpages. Importantly, the tool uses a creative procedure that allows for communication with and the incorporation of other browser-based tools or extensions. The tool provides for a wide range of research opportunities that may reach a new level of fidelity in webpage analysis and comparison.

In this paper, we discuss the significant considerations involved with our automated technique and we describe and document the resulting application we created that measures specific content within a webpage. We also cover critical technical challenges we encountered in building the application as well as creative solutions to these challenges. We then demonstrate the application's operation. We conclude with related work and several straightforward, feasible adaptations that would make the application of value for many diverse purposes.

2. DESIGN CONSIDERATIONS
Designing an application to consistently measure specific elements within a webpage requires a clear picture of what is a useful metric for elements on a page, a clear definition of what constitutes measurable content on a page, and the technical means to accurately measure such content. We implemented a Mozilla Firefox browser extension that satisfies these requirements by determining the number of pixels that a browser displays for selective HTML content elements. The important first step toward developing that application was clearly determining an appropriate metric.

2.1 Graphical Content Measurement
There exist many methods to measure a webpage. They include page load time, popularity, user satisfaction, byte size, diversity of content, colors, motion, etc. In general, a good metric for this task should be contextually specific, quantifiable, and can be consistently inexpensively measured [1]. Our application proposes such a metric.

Our recommended metric for measuring webpage elements is computing the number of pixels that a web browser displays for each (HTML) element. While sounding simple, in today's web environment automating the computation of this metric's results

SIGITE/RIIT'13, October 10 - 12 2013, Orlando, FL, USA
2013 ACM 978-1-4503-2494-6/13/10
http://dx.doi.org/10.1145/2512209.2512220

actually involves many complex issues, as will be seen as we describe the actions and techniques of our application. Our metric of computing the number of pixels per element quantifies how much of a valuable, limited resource (the browser window) a particular element consumes. It is feasible to determine this number when operating from within a browser which must determine how to (consistently) display the many diverse HTML elements it receives from various sources.

Webpages are primarily a visual presentation of information. One of the most important questions a web designer must answer is how much screen space to allocate to each element of a page. The larger any particular element appears, the greater the percentage of the presentation it occupies. Larger elements, such as a featured passage or featured advertisement, generally occupy more of a user's attention. Measuring elements by counting the number of pixels that they occupy in the browser is therefore contextually specific and quantifiable. In today's multi-source web environment, though, an automated calculation of element displayed pixel count is not entirely straightforward.

There must be a distinction between the pixels of the measurable content -- the message, such as an image or the words of a paragraph -- and the effects of style-rendered pixels immediately surrounding an element, such as border and padding. Pixels that fit within the context of a webpage's message are the pixels of an HTML element that lie inside all padding, border, and margin. Style-rendered pixels, on the other hand, are typically solid-colored "whitespace" pixels that provide a means to spatially arrange and emphasize certain elements on a webpage. This aligns with the classic distinction between style as in Cascading Style Sheets (CSS) and content in HTML design. For example, an identical image may appear on multiple webpages with different border and margin settings. Content pixels notably incorporate adjustments to height and width such as font size: they describe the pixels that users actually see and process. Based on this situation, it is important to define which elements count as content.

2.2 Defining Webpage Content

Ask a group of web users to define the content of a webpage and an inconsistent definition will inevitably emerge. What constitutes 'content' is relative to purpose. Therefore, a metric calculating content should allow for the measurement of selective or customizable categories of content. Our application is powerful enough to accurately describe certain sets of elements on a webpage such that one can automate tracking of them while easily redefining content categories. In some cases, this may amount to identifying certain HTML tags, such as all images.

Another method is to use CSS selectors for more precision. CSS selectors describe elements on a webpage by using HTML tag type, height and width attributes, background color, id or class attributes, and other descriptors. No matter the means, the end state is the ability to describe exactly which elements of a webpage are content such that our application can count the pixels of each. As will be shown in the next subsection, piggybacking on web browser capabilities can make this task much simpler.

2.3 Importance of the Web Browser

A webpage content measuring tool must determine how to display particular elements on the screen based on HTML and other code and how to classify elements based on CSS selectors. Building an application that is able to parse HTML, CSS, Javascript, and other webpage technologies, classify elements based on selectors, resolve overlap, boundary, and padding conditions, and finally count the pixels of each element is a significant undertaking. All of these tasks are essentially the job of web browsers. Despite very clear and accepted WWW standards, however, web browsers frequently display the same code differently. To remain contextually specific, it is important to capture content as users will actually see it. Using a popular existing web browser ensures that our application remains updated as standards and practices change. Therefore, we chose to make the web browser part of the application.

One option is to display a webpage within a browser while using an external program to capture elements on the screen. However, clearly defining the boundaries of elements and differentiating style from the true content would be difficult. We instead decided to build a browser extension (add-on). Our tool extends the popular Mozilla Firefox web browser because it brings portability across operating systems and is open-source with excellent documentation. By extending the browser, the browser itself becomes a key building block that makes creation of the tool much simpler. Browsers parse and analyze the HTML, CSS, and script code composing a webpage in order to properly render the page. Firefox exposes the various methods and properties associated with its rendering of webpages and their elements to extension code - making our application easier to build. Many of these HTML rendering methods are standardized across various popular browsers thanks to the World Wide Web Consortium (W3C) Document Object Model (DOM). In the next section, we describe how to use DOM methods to overcome difficult technical challenges to measuring webpage elements.

3. IMPLEMENTATION HIGHLIGHTS

Design considerations significantly guided implementation, but building the application brought significant technical challenges requiring creative solutions. Foremost, programmatic identification of content elements was difficult given the wide range of HTML code in practice on the web. Obtaining automatic and accurate measurements of displayed pixels for each content element required some modifications to built-in capabilities to separate rendered style and account for embedded windows (iframes). Dynamic and multi-sourced webpages presented a challenge in determining when a page was completely loaded. Integration with other extensions required working around the protections browsers enforce between different extension codebases for security and other purposes. And finally, the testing and debugging demanded a means for the programmer to visually confirm results.

Building a Firefox (or any browser) extension requires some initial understanding of an extension file structure but primarily involves Javascript use and a basic knowledge of the W3C DOM. The Mozilla Developer Network has tutorials and a repository of references available at https://developer.mozilla.org/en-US/docs/Building_an_Extension. The important tools for accessing a webpage document and the necessary browser methods are available through Mozilla's DOM API or XPCOM API. Using the DOM, an extension is able to access and dynamically change the content, structure, and presentation of a webpage much like Javascript embedded within a page but more so. In this section, we will focus on the important DOM methods necessary to implement the critical parts of the application. The source code of our tool is available at http://www.rumint.org/gregconti/publications/awe1.zip.

The basic algorithm for the application is to find and learn the position and sizes of all content elements displayed in the browser once a webpage fully loads. Key steps involve obtaining programmatic access to content elements, measuring their size and position on the screen, determining when to obtain and measure elements, integrating with other browser tools, and testing and debugging. Below, we describe creative ways to overcome the technical challenges these steps presented.

3.1 Identifying and Describing Content Elements

Research objectives will dictate which elements of a webpage will be treated as the measurable content. This aspect of the process is not automatable. A good technique is manual inspection of several different webpages of interest in order to identify patterns and common elements of interest. In most cases, this will be fairly straightforward. Consider Figure 1 as an example of a generic webpage as it would appear in a browser window. Research may require the ability to measure the images or Flash objects on a page, for example.

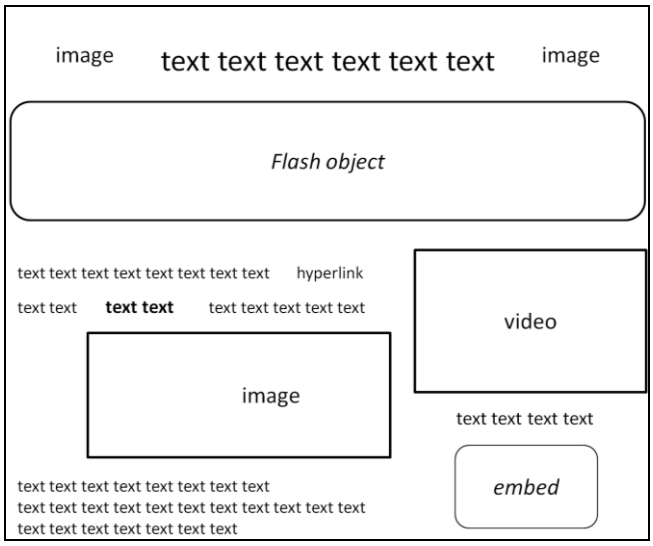

Figure 1. The appearance of a generic webpage with various HTML elements displayed in a browser window.

Existing tools like Mozilla Firefox's DOM and Style Inspector assist the researcher in identify particular elements. With a right-click on the webpage, the tool highlights selected elements within the page's HTML code in a window below the display of the page, revealing an element's tag name and other attributes and property values. The simplest method to classify elements as content is by tag type, but any programmatic method of distinguishing HTML elements using the DOM API is possible. One could view the DOM tree created from Figure 1 using the Firefox DOM and Style Inspector. In the DOM, an HTML document is basically a navigable and manipulatable tree of HTML elements. The tree reflects the structure of the HTML code for the webpage. It is possible to programmatically navigate the tree using DOM API methods on each element, such as childNodes and parentElement. The DOM tree includes more than HTML elements; it also allows access to text nodes.

In Figure 1, the paragraph below the Flash object has three types of text: plain text, a hyperlink, and bolded text. A browser renders each passage of text according to its parent (containing) HTML element in the DOM tree. These HTML elements dictate, for example, if the text should be block or inline and bold or italics. In every case, the text itself is a child of those elements known as a text node. The difference between an element and a text node will be important for classifying text. Text displayed on the screen may appear inside paragraph, heading, bold, list item, and many more HTML tags. In practice, web designers use nearly all block-level and inline tags as container elements for text. Rather than listing every possible text containing tag as content, it is simpler to just consider all DOM text nodes as content and selectively eliminate undesired parent element tag types. This method has important implications for measuring text as content, which we discuss in the next subsection. After the researcher determines which elements or nodes to consider as content, the next step is to describe them such that the application may locate them within the DOM tree of any webpage.

The simplest method of describing content elements for an application is to identify elements by HTML tag name. The most straightforward approach is to traverse a document's tree with a recursive depth-first algorithm, beginning with the document (root) node and using the childNodes method. An element in the DOM is essentially an object that has many accessible properties which an application can evaluate in order to classify it as content. A more precise approach to obtaining content nodes is to pattern match specific elements using CSS selectors. A DOM element method called querySelectorAll returns a depth-first, pre-order search of all elements matching a comma separated list of CSS selectors. Describing content in terms of a combination of CSS selectors is a proven technique for many other works and all browsers support CSS selectors to be able to apply style rules. It is possible to describe content using CSS selectors based on HTML tag type, class name, or even certain properties. Either tree traversal with element property inspection or the obtaining of a list of elements from querySelectorAll provides access to the desired content on a webpage. The next step is to measure the pixels of each element in the content list.

3.2 Measuring Content Pixels

The content our application identifies includes any image, video, embed, or object element and any displayed text. Object and embed tags may include Flash that appears on a webpage. The first four elements each have HTML tag names (such as IMG) which our application uses to identify those elements as content while traversing a webpage's document tree. Most HTML elements occupy a rectangular space on a page which the browser has calculated. Measuring displayed text requires some additional manipulation of the document. In both cases, measuring the true content pixels of an element in a webpage (as opposed to style pixels) depends on several important definitions.

First, there must be a consistent means to classify exactly which pixels count as content and which do not. For example, HTML and CSS code may render an image on the browser window along with a border and a large margin. The basic premise is to draw a rectangle around the 'true content' pixels of each content element. Next, text on a webpage may appear in conjunction with significant whitespace depending on whether it falls inside a block-level or inline tag. And last, there must be a method for determining how to classify pixels from different elements on a webpage that may overlap and obscure each other.

3.2.1 Minimum Bounding Rectangles

Not every pixel rendered in a web browser will become the focus of a user's attention. Web browsers may render padding, border,

and margin to almost any HTML element using the CSS box model. These three properties allow web designers to creatively decorate or emphasize certain elements as well as spatially arrange them. Figure 2 shows the same generic webpage as Figure 1, but emphasizes the effects of padding, border, and margin which a browser renders based on CSS, style rules, scripting, or other methods. The markup in Figure 2 helps to show the effects of the CSS box model and where each element's pixels actually begin and end.

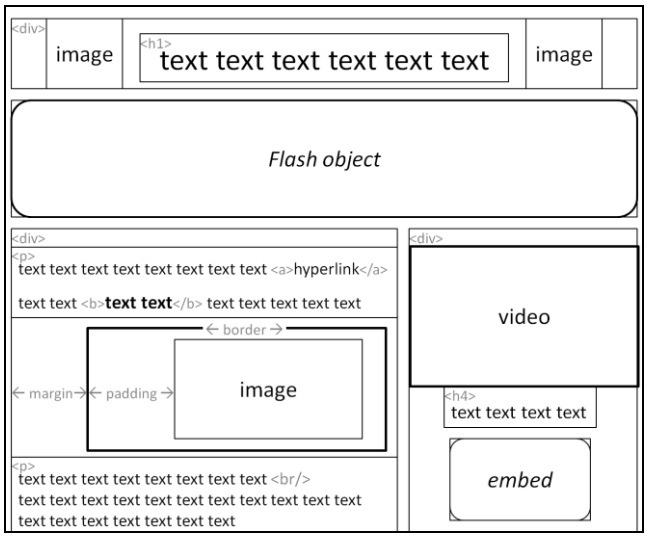

Figure 2. The generic webpage shown in Figure 1 with select markup of HTML element tag names. A thin border around many of the elements indicates an element's boundary. While each element has padding, border, and margin (possibly 0 pixels), only those of the large image element are annotated.

Our application discounts the pixels of all padding, border, and margin because they are aspects of style - not the element itself. For example, a web browser may display an image with a large border, but the file itself has no border. Moreover, the same image may appear on different pages with differing style renderings and some browsers may even display those style rules differently. The style of a webpage plays an important role, but we distinguish it from the content of a page. This definition of content narrows the selected pixels to those that form the actual message that users take away from the page – the message contained in the words and images of the page. The result is essentially a minimum bounding rectangle that surrounds an image or object element but not its padding. Figure 3 depicts these content rectangles (in blue) for elements in the same webpage as Figures 1 and 2.

Determination of an element's minimum bounding rectangle is simple for most HTML elements because the element method getBoundingClientRect provides an element's left, top, width, and height within a browser window. A window is a browser window object (a frame) that displays a webpage document once the document is loaded. The getBoundingClientRect element method includes any border and padding the browser renders along with the element. It is possible to remove those attributes by first obtaining the final computed list of style rules from the window method getComputedStyle. The list contains the number of padding and border (and scrollbar) pixels for all elements. iframes, which are separate documents with their own windows displayed within the main browser window, complicate the absolute positioning of their elements. Many webpages use iframes to display external content or for other reasons, and often even nest them. Since bounding rectangle measurements are relative to a document's window, absolute element rectangle positions – and whether they are completely visible – depend upon the determination of each iframe's offset within the main window. Offset calculations must accumulate the offset values of each iframe, to include recursively calculating all nested iframes and applying the final offsets to their elements. Elements may also overlap and occlude each other for other reasons.

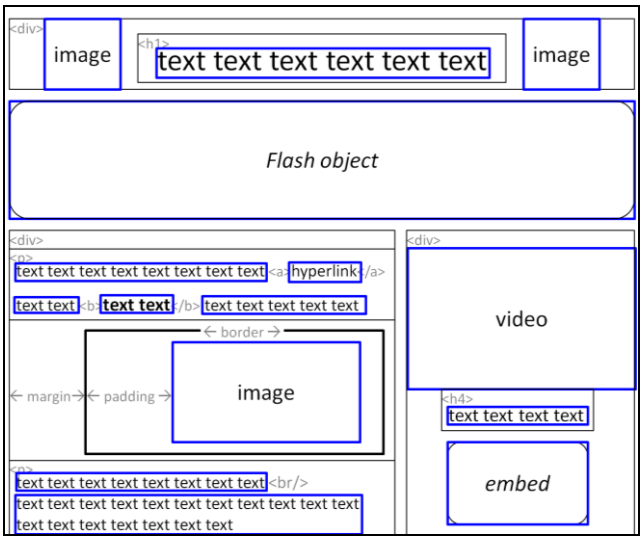

Figure 3. This figure depicts our application identifying content (blue) regions from the generic webpage of Figures 1 and 2. Content includes image, video, embed, object elements and displayed text. Content regions are based on bounding rectangles which ignore padding, border, and margin.

The complexity of HTML and CSS may result in the overlap or occlusion of other elements. It is necessary to determine how much of a bounding rectangle is actually visible. This begins with finding the bounding ancestor element. Our application accomplishes this by evaluating each ancestor element in a bottom-up algorithm using each element's parentElement method, stopping when one of the ancestor's own bounding rectangles restricts it or when reaching the HTML body or iframe element. Restriction is based on the overflow property or, if desired, the edge of the browser window. Those restrictions essentially reduce the size of the bounding rectangle. An element's opacity, visibility, z-index, and display properties may also affect the visibility of an element's rectangle. The z-index property can cause overlap between elements, for example.

Webpage designers usually avoid accidental overlap between elements, but it happens by design or browser inconsistencies as well. Default HTML element stacking order provides that elements appearing last in the code will appear on top unless they are positioned outside the normal flow and given a different stack order (z-index). For implementation simplicity, we assumed the default stack order of elements by traversing the document tree in a depth-first manner. In the case of overlap between like elements (i.e. content-content), this assumption does not affect the correct classification of overlapping pixels. However, there is a problem where content and non-content elements stack according to a different order. A solution is to clear or reclassify all pixels of non-content elements encountered during a stack order traversal of a document's elements.

3.2.2 Recording Content Pixels

To record the final content bounding rectangles, our application uses an HTML canvas that it dynamically inserts over the main document. The canvas element allows web designers to draw and animate graphics, such as rectangles. The canvas functions as a built-in 2D hashtable of pixels. Automated drawing of the final bounding rectangles of each content element essentially labels the content pixels in their absolutely positioned locations and assists with visually confirming the results. The overall calculation of content pixels amounts to a single pass over the canvas while counting pixels within the labeled rectangles. The final measurement of all content is basically the sum of the areas of all blue rectangles in Figure 3.

3.2.3 Capturing Text

Figure 3 depicts the inclusion of text pixels as content. As Section 3.1 introduced, capturing text as content is more complicated than capturing a bounding rectangle around other HTML elements. To capture the text of a document, a creative and effective technique is to insert an arbitrary span element into the document as the immediate parent of all text node descendents of the root body element. DOM methods createElement, insertBefore, and appendChild provide the means to achieve the desired effect. For example, an h1 tag containing text becomes the grandparent of the text node – with a span taking its place as the immediate parent. With a few steps to ensure the span does not alter browser text rendering, spans create an anchor for obtaining a bounding rectangle for any passage of text.

Since the span is an inline element, it helps minimize whitespace pixels when calculating the number of pixels that text actually occupies in a browser window. Inserted spans should include a special class attribute that will classify them as content and have the padding, margin, and border set to 0px through inline style (inline style rules take precedent). It is also helpful to surround each contiguous set of non-whitespace text (text that is not a carriage return or newline, for example) with its own span. This may result in several inserted spans within a single, original text-containing element, but this further reduces whitespace and makes measurements more accurate to 'true content.' Not every text node is content; the application should ignore text nodes within elements such as style and video that are not normally visible in the browser. These additional measures assist in more accurately measuring the number of pixels of the text and prevent the possible double-counting of text that appears inside multiple elements.

3.3 Initiating Measurements

The web is dynamic. Designers competing for user attention create flashing, animated, and interactive webpages. Many popular websites use various scripts. This environment complicates the decision of when to measure a webpage because the measurements may change over time. It may change by design, through user interaction, scripting, or simply because of lags in page loading. The researcher may be interested in the change of measurements over time or with user interaction, but for simplicity our application currently measures a webpage at a single point in time shortly after page load.

Browser extensions are able to listen for certain events associated with webpages, such as a document "load" which indicates when a document and all of its resources have fully loaded. However, this event may fire before embedded documents (iframes) load, as they depend upon user or location parameters that dynamic pages detect in various ways. Our application measures a webpage five seconds after the base document's "load" event. We empirically confirmed the 5 second delay was sufficient for every webpage in our dataset to fully load all iframe documents and for scripted elements to "settle" (excluding user interaction).

3.4 Integration with Other Browser Tools

Integration of another developer's external application may provide the best means of determining content or for further sub-classifying content. There are thousands of browser extensions that fulfill various purposes, but fundamental browser security demands a separation that makes communication between them difficult. An innovative solution is to modify elements of the document by dynamically inserting an arbitrary class attribute to elements in a live document and therein share information between extensions. Since all extensions have access to the document HTML code and its current state, this technique safely bridges the security barrier. For example, we modified Adblock Plus [2], a popular ad blocking extension, to label advertising elements as a particular type of content by inserting a unique class name for those elements in the live document. Our content measuring application included this class name in the list of CSS selectors describing the content and added those elements to the tracking canvas with a different color to distinguish them as a unique subclass of content.

3.5 Testing/Confirmation

Using an HTML canvas to track content pixels provides the key ability to visually confirm that rectangles align properly with content elements. It is also possible to insert arbitrary properties and values into elements of a document, such as the dimensions of bounding rectangles, to permit a manual inspection of values when viewing a document's source code.

4. DEMONSTRATION AND ANALYSIS

We evaluated our tool using Alexa's top 25 U.S. websites and found, through manual confirmation, that it accurately measured content elements (as defined in Section 3). We placed a corpus of screenshots and archived websites online at http://www.rumint.org/gregconti/publications/awe2.zip. Figures 4 and 5 demonstrate how the tool works on two popular webpages using the definition of content in Section 3. Thin blue rectangles surround each content element and the figure captions list the total number of pixels for each page.

As discussed above, it is possible to further subcategorize content. Red rectangles surround a subcategory of content in Figure 5. We modified, with permission, the code-base of Adblock Plus version 2.2.1, a popular open source ad blocker available at http://adblockplus.org. Rather than blocking them as Adblock Plus normally would, our modified version of Adblock Plus labeled advertising elements using the technique discussed in Section 3.4. Figure 5 demonstrates the ability of using external tools to guide content classification and the potential of creating subcategories of content to provide greater fidelity with webpage research. Our Adblock Plus example also illustrates the power of integration because our application can seamlessly adjust when Adblock Plus updates its list of ad sites.

Our application currently has some restrictions which follow the simplifying assumptions we made in building it. For example, we excluded CSS background images from our definition of content because they frequently overlapped multiple elements and are

rendered in the browser through style rules rather than HTML elements. This is evident in Figure 4 where the Amazon logo image, implemented as a CSS background-image, has no bounding rectangle. CSS background images were the single dominant challenge that our application ignored, but a more robust algorithm could improve upon this shortcoming. Despite its limitations, Figures 4 and 5 demonstrate the potential power our application offers in providing a greater fidelity in analyzing and comparing webpages.

5. RELATED WORK

Our application does not replace traditional usability and user experience evaluation techniques, but potentially enhances them. Automated website measurement tools have partially resembled our own efforts. Commercial and open-source software as well as research tools provide an automated means to accomplish certain aspects of our tool. Ivory et al developed an automated tool that functions like a web browser and calculates 11 page-level metrics useful in comparing webpages and designs [3]. Those metrics provide a statistical analysis of webpage content like word count, body text %, page size in bytes, image % in bytes, and image count. This work most closely resembles our own, but our use of an existing browser provides a more accurate measurement platform. Other software programs allow users to manually measure pixels on a screen between two points, and several browser extensions (add-ons) allow users to manually highlight a single element in the browser. Frietas developed a Firefox extension that allows users to manually measure any element in pixels [4] and Firefox's DOM and Style Inspector tool assists in identifying elements on the screen; but neither tool can measure multiple elements automatically.

6. CONCLUSION AND FUTURE WORK

There are many useful metrics for comparing webpages on the World Wide Web, but they measure a page holistically, fail to measure pages within the context of the message that users see, or use methods that are not automatable. Our technique provides a means to increase the fidelity of webpage analysis and introduces a novel metric focused on the number of pixels that certain elements on a page occupy in a browser window. This method is customizable, provides user context in measuring the pixels that users actually see in a popular web browser, and is fully automatable. Several feasible extensions of the application will suit this technique for many different research objectives.

Promising future research areas include subcategorization of content, integration with other external tools, and general improvement of the application. We have demonstrated the utility of classifying content into various categories in Figure 5. Content may be more accurately measured through a content-specific weighting scheme, such as through element opacity. Measurements may also be taken over time to capture the dynamic nature of webpages. Finally, more accurate measurements demand the lifting of several simplifying assumptions discussed in Section 3. Our technique can potentially provide greater fidelity in research which may lead to increased understanding of common practices on the web and improved user experiences. As an automated tool, our application has the potential to improve search engine rating schemes and inform users of global trends with respect to certain elements on a webpage. Finally, a promising area of future work includes opportunities for a more general application of our metric as an automated tool for other purposes.

Figure 4. A screen capture of the Amazon homepage using our application to measure page content as defined in Section 3. There are 418,641 content element pixels out of 633,270 total pixels in this browser window.

Figure 5. A screen capture of the NY Times homepage using our application to measure page content as defined in Section 3 and a subcategory of content as discussed in Section 3.4. There are 361,571 content element pixels out of 633,270 total pixels in this browser window. 94,278 of the content pixels are the subcategory of advertising elements that Adblock Plus identified (shown with red rectangles).

7. REFERENCES

[1] Andrew Jaquith. 2007. Security Metrics: Replacing Fear Uncertainty, and Doubt. Addison Wesley, 2007.

[2] Wladimir Palant. 2007. Adblock plus Firefox extension. Available from http://adblockplus.org/en/firefox

[3] Melody Y. Ivory, Rashmi R. Sinha, and Marti A. Hearst. 2001. Empirically validated web page design metrics. Proceedings of the SIGCHI conference on Human factors in computing systems, March 2001, Seattle,Washington, pp 53-60.

[4] Kevin Freitas. 2011. MeasureIt Firefox extension. Available from https://addons.mozilla.org/en-US/firefox/addon/measureit/

The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Military Academy, the Department of the Army, the Department of Defense, or the United States Government.