CSS Layout Engine for Compound Documents

Mikko Pohja and Petri Vuorimaa
Telecommunications Software and Multimedia Laboratory,
Helsinki University of Technology
P. O. Box 5400, FI-02015 HUT, Finland
mikko.pohja@hut.fi and petri.vuorimaa@hut.fi

Abstract

XML is nowadays widely used to describe structural documents on the WWW. The layout of the documents can be defined by style sheets or by a language itself. XML languages can be combined to get the desired set of features for a document. Those documents are called compound documents. Also, multimedia applications can be defined using XML. In this paper, we have defined general requirements for a compound documents' CSS layout engine, which can be used in an XML user agent. The requirements relate to features of the layout engine, devices in which it can operate, and interfaces of the cooperating components. In addition, we describe our own implementation of a layout engine, which is done according to the requirements and operates in different devices.

1. Introduction

Extensible Markup Language (XML) [3] is designed to describe structural documents especially on the World Wide Web (WWW). One of the design principles of XML is to separate the content and its layout, which provides a clear and simple way to create and maintain structural documents. The layout of a given document can be defined by Cascading Style Sheets (CSS), Extensible Stylesheet Language (XSL), or it may have its own presentational semantics, e.g., Scalable Vector Graphics (SVG).

XML languages can be combined. The idea is to create languages for a specific purpose and use their elements as part of any XML document, when needed. For instance, a document could consist of XHTML, for content structuring, and XForms, for user interaction. A document, which consists of more than one XML specification is called a compound document. The compound documents set requirements for the user agents. They have to be able to handle functionalities of the all used languages and render the elements properly. The user agent consists of several components, which all have some certain duties (e.g., parsing, rendering, etc.) relating to the processing of a document. One of the critical components of an user agent is the layout engine.

The layout engine is in charge of the spatial dimension of a given document. It must be able to handle the compound documents, which may contain a diverse graphical content. All the content cannot be rendered by a single rendering method, but rather by the several different methods simultaneously. The layout engine of a novel user agent must be able to handle and combine all the rendering methods. In addition to spatial layout, temporal dimension of a document must be controlled to represent multimedia applications. The layout engine has to be integrated into a scheduler, which synchronizes the media elements of a application. In this paper, we represent a design and an implementation of a layout engine for X-Smiles XML browser [13].

The proposed layout engine can render any XML document styled by CSS. Moreover, it can provide a region for a graphic, which is laid out according to some other model. Overlay rendering method, in which all the graphics are blended at the pixel level, has been left as a future work, though. The languages styled by CSS can be combined with the other XML languages supported by X-Smiles (e.g., XForms and SVG). The multimedia presentations can be defined either by Synchronized Multimedia Integration Language (SMIL) [2] or by Timesheets [10, 6]. Also, we wanted to enable the layout engine to operate in the various environments (e.g., in digital television). The implementation was evaluated through the performance measurements.

The paper comprises as follows. Next Section gives an overview of a related work. Sections 3 and 4 discuss design principles and implementation of the layout engine, respectively. The results of the work are introduced in Section 5, while Section 6 defines further development of the layout engine. Finally, Section 7 concludes the paper.

2. Related Work

The scope of this paper is a CSS layout engine for an XML compound document user agent.

Even though the XML documents can be combined by their nature at the element level through namespaces, rendering of the compound documents is not defined explicitly. W3C has addressed the problem recently by forming a compound document formats working group [1]. The aim of the group is to produce recommendations on combining separate component languages, like XHTML, SVG, XForms, MathML, and SMIL, concerning user interface markups. They have already published the requirements for compounding documents by reference [1]. Vodafone, for one, has made a proposal how to integrate XHTML Mobile Profile (XHTMLMP) and SVG Tiny[2].

Some methods for the processing of the compound documents have been suggested in the literature, e.g., [4, 9]. Both approaches are general compound document user agents. The supported documents can include separate components by reference. Each component is taken care by a module. Thus, the user agents can be extended easily by implementing the needed modules. The components are laid out in separate regions in the document area. In addition to the general compound document user agents, there are some XML user agents, which can handle the XML compound documents.

XVM [7] is a component-based XML processing framework for the compound documents. It can be used on both server-side and client-side. XVM has a component for each supported XML vocabulary and it uses DOM to process XML data. The vocabularies are separated by namespaces. XVM can be used, for instance, as a base for XML browser. However, the presented XML browser prototype does not support compounding by inclusion and it has limited rendering engine.

Another browser framework, similar to XVM, is presented in [15]. It is also based on the components, which are in charge for an XML vocabulary each. The layout issue is not discussed either in this approach. Basically the equivalent features can be also found from some XML application development tools, which provide a view for a semantical content of a document, e.g., Amaya [8].

XMLC [12] is an applet through which the XML + XSL documents can be displayed in any HTML browser with Java support. XMLC uses Java objects called Displets to render and provide functionality to each element in an XML document. The Displets are small rendering modules, which can contain both content, such as, text and images and other Displets. The Displets are positioned absolutely and they can be overlapped in the transparent or opaque mode. XMLC does not support CSS either.

The CSS layout is realized in several current web browsers. For instance, Mozilla[3], Opera[4], Konqueror[5], and Internet Explorer[6] have support for the general XML documents. However, they all were originally HTML browsers and were not designed to operate as complete XML user agent with all the rendering methods and support for the compound documents.

3. Requirements

We wanted to integrate the layout engine into an XML browser, which supports several XML specifications and operates in various devices. The requirements of the layout engine and the reasonings are discussed in Table 1. The requirements are divided into three groups, which are the technologies the engine must support, required operation environments, and application programming interfaces.

The layout engine must be able to handle the compound documents styled by CSS. In addition, it must reserve regions for other languages (e.g., SVG). That is, a module, which takes care of an external language, renders the elements in the given region. Overlay rendering is needed to complete the layout engine.

Multimedia presentations can be described by XML, too. In addition to the structure and the layout, the timing of the components of the presentation have to be defined, though. There are two options to control the temporal dimension of an XML document. One is SMIL [2] and other is Timesheets [10, 6].

The layout of the document must be synchronized with the DOM modifications. The easiest way would be just lay out and render the document again. Optimized alternative is to lay out and render only the areas, which need to be laid out (i.e., the areas that are damaged).

The layout engine has to be integrated into the modules and interfaces of an XML user agent. Usually, a core of the user agent controls the overall operation of the user agent. The core instantiates the layout engine and assigns the XML document to be rendered for it. The layout engine access the XML document through the general DOM interface. Dynamic changes of the XML document and the language specific functionalities are realized by the extended DOM elements. In addition, the layout engine has to operate with a CSS engine, which provides the style attributes for each element in a DOM.

Finally, the layout engine needs a user interface toolkit to render the text and the graphics. To operate also in the

1 http://www.w3.org/2004/CDF/
2 XHTMLMP+SVGT, http://lab.vodafone.com/public/XHTMLMP-SVGT-Recommendations.html

3 Mozilla, http://www.mozilla.org/
4 Opera, http://www.opera.com/
5 Konqueror, http://www.konqueror.org/
6 Internet Explorer, http://www.microsoft.com/windows/ie/default.mspx

Requirement	Reasoning
Technology support	
Must support CSS.	CSS is used to style the documents on the WWW.
Must be able to lay out the compound documents.	The XML documents may consist of several XML specifications. The layout engine must be able to handle all kind of combinations.
Supports temporal dimension of the documents.	To enable the multimedia presentations through XML, the layout engine must reflect to temporal changes of a document.
Supports dynamic DOM operations.	Since DOM can be modified dynamically, it is vital that the layout engine can reflect the changes efficiently. That is, the operation must be optimized in the dynamic cases.
Operation environments	
Operates in the desktop computers, in the mobile devices, and in the digital television set-top boxes.	The Web applications are accessed by the various devices nowadays.
Interfaces	
Cooperates with the other components of an XML user agent.	The layout engine needs other components to be able to operate. Those include XML processing, browser core, and CSS engine.

Table 1. The requirements of the layout engine.

digital televisions, the UI toolkit has to be compatible with the HAVi[7] toolkit.

4. Implementation

In this section, we introduce our implementation of an XML layout engine. The first author has implemented the layout engine discussed in this Section. The implementation is part of the X-Smiles XML browser [13]. The implementation supports most of the CSS Mobile Profile specification. Currently, it has a partial support for *float* and aligning content. Styling of the borders and the background have some deficiencies. Otherwise the specification is well supported. The implementation is discussed in more detail in the following Subsections.

4.1. Layout Engine

An overview of the layout engine is depicted in Figure 1. The layout engine consists of a renderer and a set of views. In addition, it cooperates with a CSS style context, which provides the styling attributes to the elements, and a DOM document. The renderer is created by an XMLCSS module, which handles the CSS styled documents in X-Smiles. Correct module for each media is defined by the Content Handler, which is assigned by the browser window.

The general idea of the layout engine's operation is as follows. The XMLCSS module creates the layout engine

7 Home Audio Video Interoperability, http://www.havi.org/

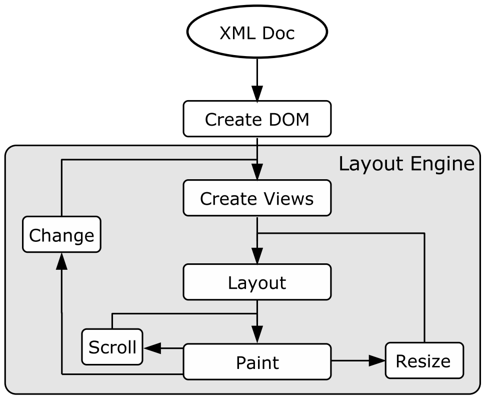


Figure 2. Flowchart of the layout engine.

and sets a document to be rendered. The document is accessed through the DOM interface by the layout engine (cf. Figure 2). The views are created according to the DOM elements. Every view creates its child views. When the views have been created, they are laid out and painted. The process is repeated partially due certain operations (e.g., scrolling the document, resizing the window, or modifying the content). The process is discussed in more detail in the next Subsections.

4.2. Views

The views are the visual presentation of the DOM elements. They are in charge both laying out and rendering the DOM elements. That keeps the layout engine more com-

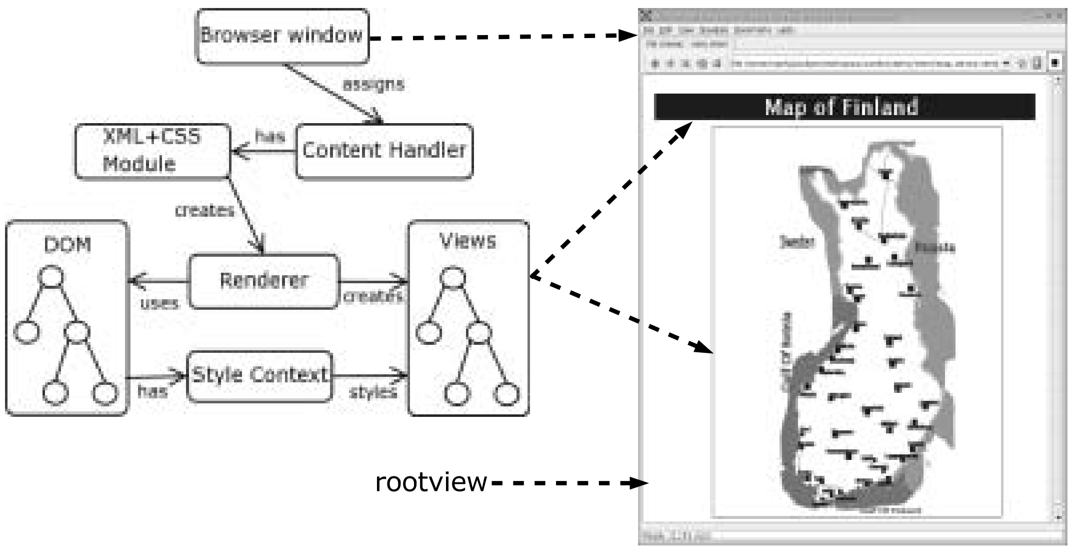


Figure 1. Overview of the components of the layout engine and relation to the actual browser.

pact compared to the solution where laying out and rendering are separated to the different components.

Basically, every element has its own view. In addition, there are some extra views to layout the document correctly and an element may have more than one view (e.g., a text element can be divided to several lines each of which is represented by a view). The views form a tree-like data structure just as the DOM.

The views can contain content and other views. A parent encloses its children also physically (i.e., the size of the parent extends over the children). A modification of a view affects to a whole sub tree underneath it. Generally speaking, there are two kinds of views, content views and branch views. The content views represent the contents of a DOM document (i.e., text, images, etc.), whereas the branch views represent the elements in a DOM. An element's view is decided by its CSS declarations (mainly by a display value). The content views take care of rendering the actual content of a document. An example of the view mapping and the laying out is given in Section 5.1. The rendering is done by Java's painting mechanisms. The painting is done recursively in a view tree. Only visible areas of a document are painted. If user scrolls a document, it is repainted (cf. Figure 2, the *Scroll* operation). Parent view calls the paint method to a child if the child intersects with the current visible area of the document. In the case of repaint and dynamic modifications, only damaged areas are painted again.

Every view contains information about its position and size. The position is view's absolute position in a flow and is set during the laying out of a document (cf. Section 4.3). The size of a view depends on the content it embeds and size requirements it has. A view may have minimum, maximum, and preferred size requirements. The requirements depend on the view and the content it embeds. The views can also contain a graphical content like borders and background.

From the implementation point of view there are three kind of views. The content specific views, like the image and the text views, keep up the spatial information and take care of the rendering of the content. The image view also fetches the image data from the source. The CSS specific views, like the block and the inline views, embed the content views and instruct the laying out according to the CSS specification. Finally, there are additional views to structure the content. For instance, when a paragraph is laid out, the text has to be divided into rows. This is done through the paragraph and the row views. The process is discussed in more detail in Section 5.1.

4.3. Laying out

The layout process has two main duties. Firstly, to calculate the sizes of the views, and secondly, to set the positions for the views. To set the size for a view, the size of the content it embeds has to be known (i.e., the process must be started from the leaves). On the other hand, to set the position to a view, its parent position has to be known (i.e., the process must be started from the root). Basically, both ends (leaves and root) can be used as a starting point for the process. Since the root view can be found unambiguously, it is natural starting point of the recursion.

In the recursion, the position of a view is set when going from root to leaves and the size when coming back (from leaves to root). Actually, the size of a parent is updated everytime one of its children's size is set. Thus, a parent can provide correct position for each of its children during the layout process. That way both sizing and positioning operations can be combined to a single layout call. Depending on the view, its children are positioned either horizontally or vertically. For example, a block's children are one on the other, whereas a row's children are one after the other.

The position of a view could be stored either relative to its parent or with the absolute coordinates. The advantage of the relative position is a remaining positioning of the successors even if the view is moved. However, that is very rare operation in CSS based layouts. In addition, during the painting one must use the absolute positions. Since the painting is very common operation (e.g., when scrolling the document or resizing the window etc.), it is more efficient to store view's position straight in the absolute coordinates than calculate it everytime from the relative position.

4.4. Dynamic DOM Operations

The DOM document can be modified after its creation by, e.g., scripts. Obviously, these modifications have to be delivered to the layout engine. The modifications are caught by the extended DOM elements. If something happens for an element, it updates its style and also its children's styles, because some of the CSS properties are inherited to the descendants. The view tree is modified according to the changes.

If the modifications have an effect only on the layout of the element in question, then the corresponding view tree is created, laid out, and, if on the screen, painted again (cf. Figure 2, the *Change* operation). The process is equivalent to the Mozilla's incremental layout [14]. However, if the changes have an effect also on the other elements (e.g., the size changes), then the whole affected view tree must be also laid out and painted again, but not created, though (cf. Figure 2, the *Resize* operation).

4.5. Temporal Control

The multimedia presentations can be formed either by SMIL [2] or by Timesheets [10, 6]. SMIL has some limitations [10, 11] to be used as a common multimedia declaration language for all XML documents. Especially, when a document consists of several XML languages. Also, the desire of keeping the content and the presentational aspects of a document separately does not realize when using SMIL. The original design principle of XML is better fulfilled if the content and the spatial and the temporal styling of a document are separated into the three functional sections. Those problems have been solved in Timesheets, which is counterpart of SMIL. Both technologies are implemented in X-Smiles.

The layout engine can be used as a media element in a SMIL application and a SMIL application can be embedded

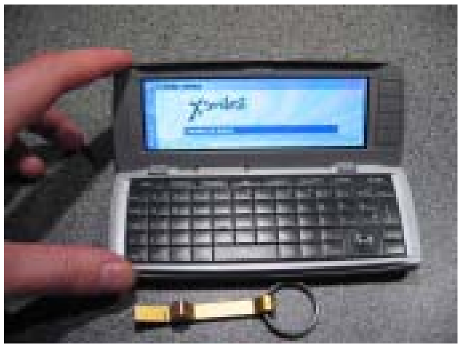


Figure 3. The layout engine running on a handheld device.

into the layout engine. In the both cases, the documents are compounded by reference.

Timesheets works the other way around. It is integrated into a host language's layout system, which makes it possible to control single elements inside a document (i.e., compound by inclusion). The Timesheets implementation is integrated into the layout engine. In short, the Timesheet module assigns temporal relations between the elements and sets styles for the elements in temporal manner. The activity of the elements and style variations are realized by pseudo-classes, which are supported by the layout engine.

4.6. Integration into X-Smiles

X-Smiles suits well for the layout engine, because it is implemented in Java programming language. Java is a good alternative to develop a cross-platform software at the moment. Using Java, we could easily fulfill the operation environment requirement. Normally, restricted device Java environments have a limited set of functions, which has to be taken into account when implementing a software. The desired selection of Java environments in this work were Java 2 Standard Edition (J2SE) 1.2 or higher in the desktop computers, Personal Java 1.1 and Java 2 Micro Edition (J2ME) Personal Profile in the mobile devices, and Multimedia Home Platform (MHP) in the digital televisions set-top boxes. The layout engine can be run in all of those environments. The layout engine running on J2ME Personal Profile in the Nokia Communicator is depicted in Figure 3 and in the digital television environment in Figure 4. The digital television integration is discussed in more detail in [5].

The relation of the layout engine to the other major components of X-Smiles is depicted in Figure 5. The layout engine connects naturally closely to the XML processing since it is DOM's visual representation. The views, which correspond to a DOM element (there are views which do not, cf. Section 5.1), are stored to the respective element. At the

Figure 4. The layout engine running on a digital television.

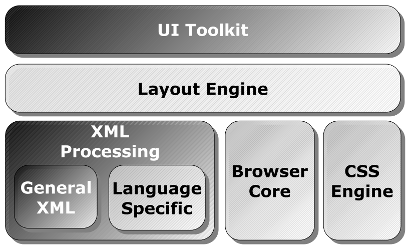


Figure 5. The components of an XML user agent.

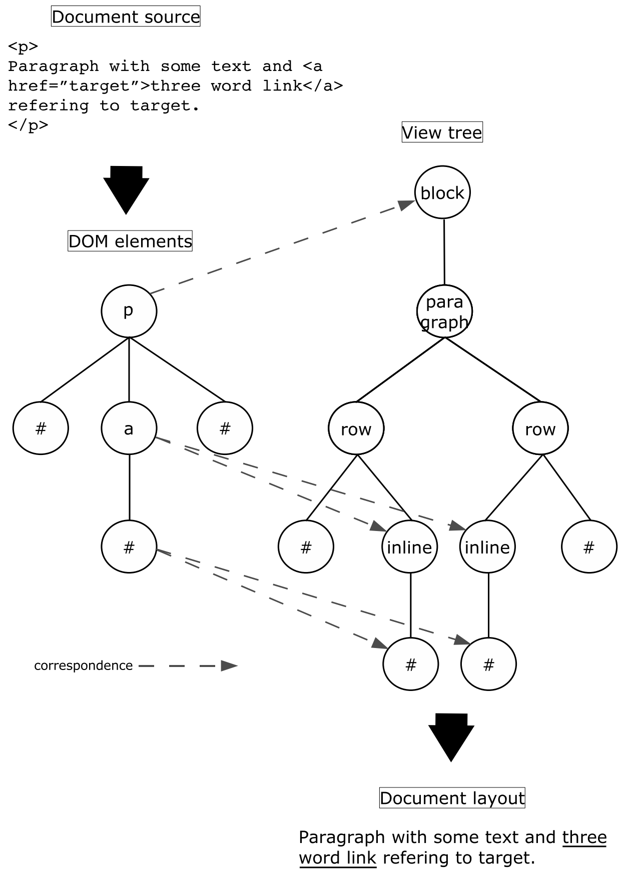


Figure 6. Text layout illustration.

same time, each view has reference to its DOM element. This way the interface can be traversed to the both directions. When the DOM tree has been modified dynamically, it takes care of the relayout and the repaint of the view tree as discussed in Section 4.4.

The browser core instantiates the layout engine and sets the document to be rendered for it. The layout engine is created separately for each document. The CSS engine is used through the DOM interface. The CSS engine creates the styles from a style sheet and the layout engine creates the views according to the styles.

The view tree is rendered through Java Abstract Window Toolkit (AWT) UI toolkit. AWT was chosen, because it enables using the layout engine in several platforms. AWT is basis for the other UI toolkits like Swing and HAVi, whose support was required in Section 3. Text, images, and graphics (e.g., background and borders) are drawn by Java's drawing methods. Widgets, such as form controls, are represented by components. Mouse events are delivered through AWT Events. The events are caught by the layout engine and transformed to DOM events.

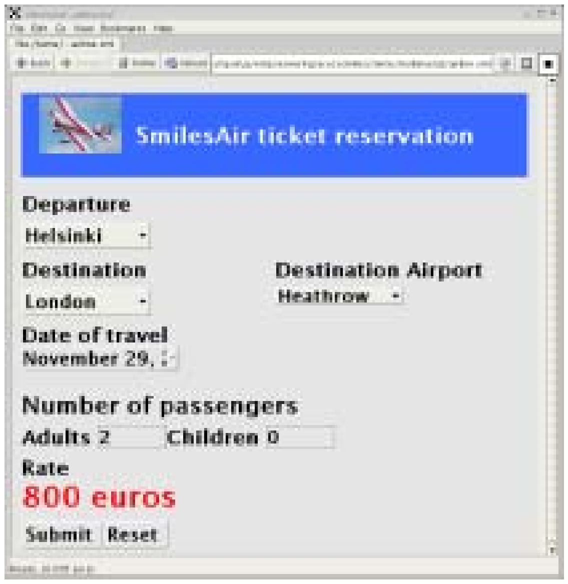
5. Results

The results of the implementation are discussed in this Section. First, a simple layout problem is shown and how the layout engine handles it. Second, two compound document examples are introduced. Third, we describe the performance measurements we made to evaluate the layout engine. Finally, there is a discussion about the measurements.

5.1. Text Layout Example

Usually, text is laid out as flow layout. However, it can preserve some preformation by the CSS declarations. Here, we describe an example how a paragraph with text and an inline element is processed in the layout engine. The process is depicted in Figure 6. The document source is defined in the upper left corner in the figure and the corresponding DOM beneath it. The nodes with "#" symbol in the DOM tree refer to the text in the source. The DOM elements are mapped to the views as follows.

Since it must be possible to divide textual content into the rows, there is a special view called paragraph view, which handles the text layout. As can be seen, the *p* ele-



SmilesAir ticket reservation

Departure
Helsinki

Destination
London

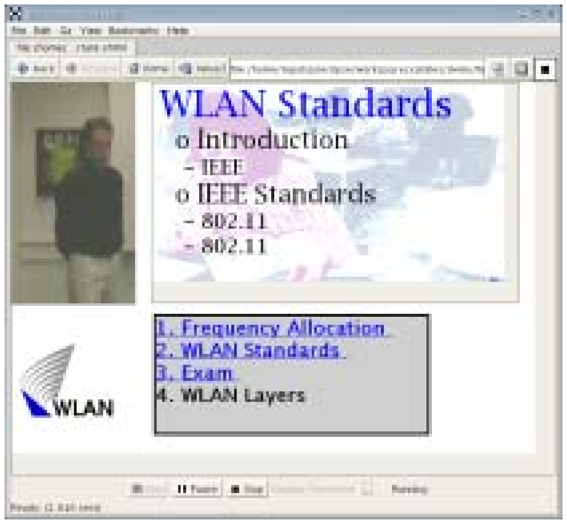
Destination Airport
Heathrow

Date of travel
November 29,

Number of passengers
Adults 2 Children 0

Rate
800 euros

Figure 7. Screenshot of the Airline ticket reservation system.



WLAN Standards
o Introduction
- IEEE
o IEEE Standards
- 802.11
- 802.11

1. Frequency Allocation
2. WLAN Standards
3. Exam
4. WLAN Layers

Figure 8. Screenshot of the Distance Education Application.

ment maps to the block view because it has *block* as a display value. When the text and the inline elements are added to the block view, a container view (the paragraph view in this case) is created for all the content. The paragraph view lays out the text. It divides the content to the rows according to the size requirements. The content views are children of the rows. The row division is always redone along with the laying out (e.g., when the browser window is resized). In the example, also the anchor element has been divided into the two rows. Thus, the DOM element corresponds to the two inline views. Also, its content node corresponds to the two content views. The other content has unambiguous DOM - view correspondence (the pointers are missing from the figure). The paragraph and the row views are additional views for laying out the content properly. They do not have the corresponce in the DOM.

5.2. Compound Document Examples

In this Subsection, the use of the layout engine in two different applications is shown. First, an airline ticket reservation system is realized with XHTML + XForms (cf. Figure 7). All the form components are XForms elements, which are compounded by inclusion. The layout is defined by CSS.

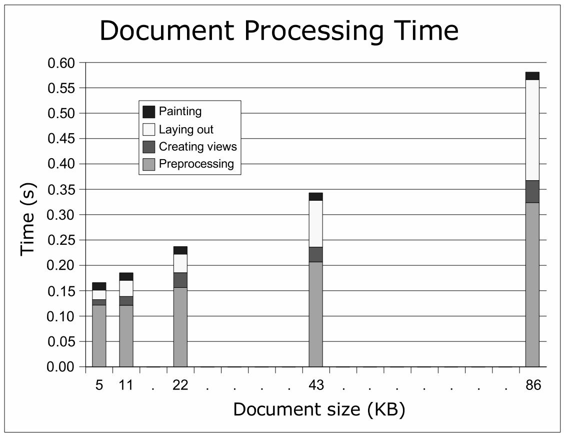
The other demonstration is a multimedia application. It describes how a distant education can be realized with the XML languages. The presentation comprises a video about a lecturer, a slide show next to the video, and an outline beneath the slide show (cf. Figure 8). The slide show is synchronized with the video and the lecture can be browsed through the links in the outline. The synchronization and the browsing has been done with Timesheets and the layout through XHTML + CSS.

5.3. Performance measurements

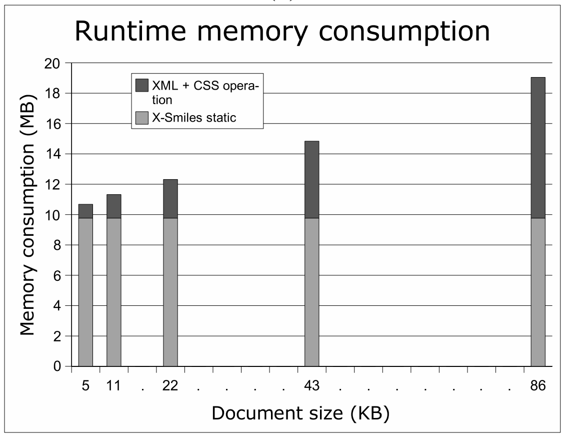
We did some performance measurements to evaluate the implementation. We measured the total processing time and the separate phases of the processing. In addition, we examined runtime memory consumption. The tests were done with five documents, which differed by size from each other. The four biggest documents were derived from the smallest document by multiplying its content 2, 4, 8, and 16 times. The tests were done on PC with 3.06 GHz Pentium 4 processor and 512 MB memory.

In the processing time evaluation, we measured the time of creating a view tree, laying out the tree, and painting the document. The total processing time includes also a preprocessing, which comprises fetching a document, parsing, and creating and assigning the styles. The results of the document processing time tests are depicted in Figure 9 (a).

The laying out takes most of time in the layout engine. Elapsed time also grows almost linearly according to the document size. View creation is not that time consuming, because it basically consists of just object creations in contrast to the laying out, which contains a lot of calculations. The painting time is basically constant in every case. This is a consequence of the fact that only the visible areas of a document are rendered. However, it is noteworthy, that painting is redone every time when part of a document becomes visible during scrolling it. In other words, the painting time in these tests represents only the painting of the first screen.



(a)



(b)

Figure 9. (a) The results of the processing time measurements and (b) the runtime memory consumptions of the documents.

The runtime memory consumptions are shown in Figure 9 (b). As can be seen, the X-Smiles' static memory consumption is fairly large and the share of XML + CSS processing is small with the smallest documents. The memory consumption grows linearly along with the document size. The growth curve is slowed down a little bit due optimization of the CSS engine's memory consumption.

5.4. Discussion

The results of the performance measurements arouse a question could the layout engine, especially the laying out, be optimized. The optimization should be realized same way as the painting is done currently. That is, the layout engine would lay out only the visible views at the time and continue laying out along with the viewing of a document. That would definitely speed up the layout process especially with large documents, but, on the other hand, slow down viewing the page. Since the absolute layout times are not very long at the moment, the trade off to slow down the viewing should not be remarkable, either. There would also be some problems to do the layout if all the views are not created. However, without closer look at the problem, it remains as an open question.

Another fact shown by the measurements is the workable implementation of the dynamic modifications. We have minimized the view creation and the laying out operations in the dynamic cases. Since they are the most time consuming operations in the layout engine according to the measurements, the implementation decisions can be found out reasonable.

6. Future Work

There are XML specifications, which do not use CSS layout model. The compound documents consist of a host language and one or more embedded languages. Some of the XML languages can be both hosts and embedded (e.g., XHTML, SVG, and X3D), whereas others can only be embedded (e.g., XForms, MathML, or XML Events). Depending on the combination of the languages, there are three different methods to render the documents. First, the layout of embedded language may depend on host language (e.g., SVG + XForms). In that case, the embedded elements are rendered using the host language's layout model. Second, a host language may provide a region for embedded elements to lay out themselves (e.g., XHTML + MathML). In the given region, embedded elements are rendered using their own layout model. Finally, the documents could be rendered using overlay rendering (i.e., alpha blending). The elements are mixed in a document and rendered using their own layout model on top of each other (e.g., SVG + XHTML). The layout engine should be able to handle all the three rendering methods.

A focus manager keeps track all the focusable content, like links and widgets, in a document. It is used, for instance, in the digital television environments and in the mobile phones, which do not have a pointing device (e.g., mouse). The navigation of a document is then realized either two dimensionally through the arrow keys or one dimensionally through the next/previous key (i.e., like the tab key). The navigation order must be decided by the focus manager. The order is not necessarily unambiguous, since the layout may change temporarily and the compounded documents maybe scrollable, etc. Even then, the navigation order must be equivalent to the visual order of the focus points all the time. Focus manager cooperates with the layout engine in two ways. Firstly, it fetches the focus points from the layout engine and, secondly, the layout engine highlights the current focus point given by the manager.

7. Conclusions

The layout engine is a critical component of a novel XML user agent. We have defined the requirements for such a layout engine. The requirements relate to the features of the layout engine, the operation environments, and the interfaces with other needed components. We have implemented a layout engine according to the requirements and integrated it into X-Smiles XML browser. We were able to fulfill the requirements.

We did performance measurements for the layout engine. The results show that there could be need for the optimization. However, as discussed above, it is not trouble-free to realize. The dynamic modifications are already optimized and the measurements indicates that the optimizations are directed to the correct operations.

To make the layout engine complete, there are still few features to implement. The most important is a full support for the compound documents. That requires implementation of the overlay rendering. Another important issue is the integration of a focus manager, which is needed mainly in the digital television environment, in the PDAs, and in the mobile phones.

Acknowledgments

The author Mikko Pohja would like to thank Nokia Foundation for providing support during the research. We would also like to thank Pablo Cesar and Mikko Honkala for valuable comments during the work.

References

[1] D. Appelquist, T. Mehrvarz, and A. Quint. Compound Document by Reference Use Cases and Requirements Version 1.0. Working draft, W3C, April 2005. Available at http://www.w3.org/TR/2005/WD-CDRReqs-20050404/.

[2] J. Ayars. Synchronized Multimedia Integration Language (SMIL 2.0). W3C recommendation, W3C, August 2001. Available at http://www.w3.org/TR/smil20/.

[3] T. Bray. Extensible Markup Language (XML) 1.0. W3C Recommendation, W3C, February 2004. Available at http://www.w3.org/TR/REC-xml/.

[4] J. Buchner. Hotdoc: a framework for compound documents. *ACM Comput. Surv.*, 32(1es):33, 2000.

[5] M. Honkala, P. Cesar, and P. Vuorimaa. A Device Independent XML User Agent for Multimedia Terminals. In *IEEE Sixth International Symposium on Multimedia Software Engineering*, pages 116–123, December 2004.

[6] T. Jalava, M. Honkala, M. Pohja, and P. Vuorimaa. Timesheets: XML Timing Language. Member submission, W3C, April 2005. Available at http://www.w3.org/Submission/xml-timing/.

[7] Q. Li, M. Y. Kim, E. So, and S. Wood. XVM: A Bridge between XML Data and Its Behavior. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 155–163. ACM Press, 2004.

[8] V. Quint and I. Vatton. Techniques for authoring complex XML documents. In *DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering*, pages 115–123. ACM Press, 2004.

[9] I. Satoh. Mobile agent-based compound documents. In *DocEng '01: Proceedings of the 2001 ACM Symposium on Document engineering*, pages 76–84. ACM Press, 2001.

[10] W. ten Kate, P. Deunhover, and R. Clout. Timesheets - Integrating Timing in XML. In *WWW9 Workshop: Multimedia on Web*, Amsterdam, Netherlands, May 2000.

[11] J. van Ossenbruggen, L. Hardman, J. Geurts, and L. Rutledge. Towards a Multimedia Formatting Vocabulary. In *Proceedings of the twelfth international conference on World Wide Web*, pages 384–393, Budapest, Hungary, May 2003.

[12] F. Vitali, L. Bompani, and P. Ciancarini. Hypertext Functionalities with XML. In *Markup Languages: Theory and Practice 2.4*, pages 389–410. MIT Press, 2001.

[13] P. Vuorimaa, T. Ropponen, N. von Knorring, and M. Honkala. A Java based XML browser for consumer devices. In *17th ACM Symposium on Applied Computing*, pages 1094–1099, Madrid, Spain, March 2002.

[14] C. Waterson. Notes on HTML Reflow. Technical report, Mozilla, December 2004. Available at http://www.mozilla.org/newlayout/doc/reflow.html.

[15] C. Xiaolu, C. Jing, G. Yongging, and S. Baile. XML Arouse the WEB Architecture Revolution. In *5th International Computer Science Conference on Internet Applications*, pages 461 – 466, Hong Kong, China, December 1999. Springer-Verlag, London, UK.