Exhibit 9

Surfin' Safari

<< Safari Beta 3.0.3 Update WebCore Rendering II - Blocks and Inlines >>

Home
Surfin' Safari Blog
Planet WebKit
Project Goals
Keeping in Touch
Trac
Contributors Meeting

Working with the Code
Installing Developer Tools
Getting the Code
Building WebKit
Running WebKit
Debugging WebKit
Contributing Code
Commit and Review Policy
Security Policy

Documentation
Wiki
Projects
Code Style Guidelines
Major Objects in WebCore
Web Inspector
Web Developer Resources
Demos

Testing
Regression Testing
Leak Hunting
Writing New Tests
Getting a Crash Log

Bugs
Reporting Bugs
Bug Report Guidelines
Bug Prioritization
Test Case Reduction
Bug Life Cycle

Archives
June 2010
May 2010
April 2010
March 2010
February 2010
January 2010
December 2009
November 2009
October 2009
September 2009
July 2009
June 2009
May 2009
April 2009
March 2009
February 2009
December 2008

WebCore Rendering I – The Basics

Posted by DaveHyatt on Wednesday, August 8th, 2007 at 5:34 pm

This is the first of a series of posts designed to help people interested in hacking on WebCore's rendering system. I'll be posting these articles as I finish them on this blog, and they will also be available in the documentation section of the Web site.

The DOM Tree

A Web page is parsed into a tree of nodes called the Document Object Model (DOM for short). The base class for all nodes in the tree is `Node`.

`Node.h`

Nodes break down into several categories. The node types that are relevant to the rendering code are:

- Document – The root of the tree is always the document. There are three document classes, `Document`, `HTMLDocument` and `SVGDocument`. The first is used for all XML documents other than SVG documents. The second applies only to HTML documents and inherits from `Document`.
  The third applies to SVG documents and also inherits from `Document`.

  `Document.h`
  `HTMLDocument.h`

- Elements – All of the tags that occur in HTML or XML source turn into elements. From a rendering perspective, an element is a node with a tag name that can be used to cast to a specific subclass that can be queried for data that the renderer needs.

  `Element.h`

- Text – Raw text that occurs in between elements gets turned into text nodes. Text nodes store this raw text, and the render tree can query the node for its character data.

  `Text.h`

The Render Tree

At the heart of rendering is the render tree. The render tree is very similar to the DOM in that it is a tree of objects, where each object can correspond to the document, elements or text nodes. The render tree can also contain additional objects that have no corresponding DOM node.

The base class of all render tree nodes is `RenderObject`.

`RenderObject.h`

The `RenderObject` for a DOM node can be obtained using the `renderer()` method on `Node`.

```
RenderObject* renderer() const
```

The following methods are most commonly used to walk the render tree.

```
RenderObject* firstChild() const;
RenderObject* lastChild() const;
RenderObject* previousSibling() const;
RenderObject* nextSibling() const;
```

Here is an example of a loop that walks a renderer's immediate children. This is the most common walk that occurs in the render tree code.

October 2008
September 2008
June 2008
May 2008
April 2008
March 2008
February 2008
January 2008
December 2007
November 2007
October 2007
September 2007
August 2007
July 2007
June 2007
May 2007
April 2007
March 2007
February 2007
January 2007
December 2006
November 2006
October 2006
September 2006
August 2006
June 2006
May 2006
April 2006
March 2006
February 2006
January 2006
December 2005
November 2005
October 2005
September 2005
August 2005
July 2005
June 2005


```
for (RenderObject* child = firstChild(); child; child = child->nextSibling()) {
    ...
```

Creating the Render Tree

Renderers are created through a process on the DOM called *attachment*. As a document is parsed and DOM nodes are added, a method called `attach` gets called on the DOM nodes to create the renderers.

```
void attach()
```

The attach method computes style information for the DOM node. If the *display* CSS property for the element is set to *none* or if the node is a descendant of an element with *display: none* set, then no renderer will be created. The subclass of the node and the CSS display property value are used together to determine what kind of renderer to make for the node.

Attach is a top down recursive operation. A parent node will always have its renderer created before any of its descendants will have their renderers created.

Destroying the Render Tree

Renderers are destroyed when DOM nodes are removed from the document or when the document gets torn down (e.g., because the tab/window it was in got closed). A method called `detach` gets called on the DOM nodes to disconnect and destroy the renderers.

```
void detach()
```

Detachment is a bottom up recursive operation. Descendant nodes will always have their renderers destroyed before a parent destroys its renderer.

Accessing Style Information

During attachment the DOM queries CSS to obtain style information for an element. The resultant information is stored in an object called a `RenderStyle`.

```
RenderStyle.h
```

Every single CSS property that WebKit supports can be queried via this object. RenderStyles are reference counted objects. If a DOM node creates a renderer, then it connects the style information to that renderer using the `setStyle` method on the renderer.

```
void setStyle(RenderStyle*)
```

The renderer adds a reference to the style that it will maintain until it either gets a new style or gets destroyed.

The `RenderStyle` can be accessed from a `RenderObject` using the `style()` method.

```
RenderStyle* style() const
```

The CSS Box Model

One of the principal workhorse subclasses of `RenderObject` is `RenderBox`. This subclass represents objects that obey the CSS box model. These include any objects that have borders, padding, margins, width and height. Right now some objects that do not follow the CSS box model (e.g., SVG objects) still subclass from `RenderBox`. This is actually a mistake that will be fixed in the future through refactoring of the render tree.

This diagram from the CSS2.1 spec illustrates the parts of a CSS box. The following methods can be used to obtain the border/margin/padding widths. The `RenderStyle` should not be used unless the intent is to look at the original raw style information, since what is actually computed for the `RenderObject` could be very different (especially for tables, which can override cell padding and have collapsed borders between cells).

```
int marginTop() const;
int marginBottom() const;
int marginLeft() const;
int marginRight() const;

int paddingTop() const;
int paddingBottom() const;
int paddingLeft() const;
```

```
int paddingRight() const;

int borderTop() const;
int borderBottom() const;
int borderLeft() const;
int borderRight() const;
```

The `width()` and `height()` methods give the width and height of the box including its borders.

```
int width() const;
int height() const;
```

The *client box* is the area of the box excluding borders and scrollbars. Padding is included.

```
int clientLeft() const { return borderLeft(); }
int clientTop() const { return borderTop(); }
int clientWidth() const;
int clientHeight() const;
```

The term *content box* is used to describe the area of the CSS box that excludes the borders and padding.

```
IntRect contentBox() const;
int contentWidth() const { return clientWidth() - paddingLeft() - paddingRight(); }
int contentHeight() const { return clientHeight() - paddingTop() - paddingBottom(); }
```

When a box has a horizontal or vertical scrollbar, it is placed in between the border and the padding. A scrollbar's size is included in the client width and client height. Scrollbars are not part of the content box. The size of the scrollable area and the current scroll position can both be obtained from the `RenderObject`. I will cover this in more detail in a separate section on scrolling.

```
int scrollLeft() const;
int scrollTop() const;
int scrollWidth() const;
int scrollHeight() const;
```

Boxes also have x and y positions. These positions are relative to the ancestor that is responsible for deciding where this box should be placed. There are numerous exceptions to this rule, however, and this is one of the most confusing areas of the render tree.

```
int xPos() const;
int yPos() const;
```

You can follow any responses to this entry through the RSS 2.0 feed. Both comments and pings are currently closed.

4 Responses to "WebCore Rendering I – The Basics"

Sam Weinig Says:
August 8th, 2007 at 6:15 pm

Awesome post Princess. You're the bees knees.

bartocc Says:
August 9th, 2007 at 10:54 am

This kind of documentation is very important and very helpful!
Thx for the post, I am looking forward for the coming ones.

Pingback from Around the Browsersphere #2:
September 5th, 2007 at 2:03 am

[...] Hyatt has posted five detailed articles on WebCore [...]

Pingback from Surfin' Safari - Blog Archive » Welcome to Planet WebKit:
December 3rd, 2007 at 10:38 pm

[...] WebKit, from descriptions of the many new features being added to technical discussions of engine internals to announcements of WebKit being used in entirely new places. But the web is a big place, and we [...]