“How browsers work”
Presentation notes

Domenico De Felice
Dublin, August 2015

Why?

Why is it useful to know the internals of a browser? And how this can help us?

Web browsers are likely the most widely used software in the world.

They present web resources and create a sandboxed environment in which web applications can run.

The way the browser achieves this is very complex and dictated by many different standards.

Some mechanisms are deceptive and/or counter-intuitive.

Understanding how the browsers work gives us useful insights to improve the efficiency of our website / web application and the general structure of the code.

Complexity

Of course it is not possible to explore all the details of how a browser works in a single presentation.

Also, each browser is implemented in its own way.

Mozilla Firerox, Google Chrome, Opera, Safari, Internet Explorer

---

General and modular approach

Nevertheless the browsers follow the same standards (IE ?)
and most of them share the same overall structure and modules/phases.

We will see how the browsers work in a more or less consistent way in these modules and this will give us a series of **insights** that we will be able to apply in our everyday work as web developers.

Browsers have two main modules:

The rendering engine, also called layout engine or simply web browser engine

Javascript interpreter

It is easy to understand what is the job of the javascript interpreter.

We will dwell on the rendering engine, where a big deal of the process takes place.

Mozilla Firefox uses Gecko, made by Mozilla
Both Safari and Google Chrome (until version 27) use Webkit, Chrome uses *Blink* after version 27

Components of a web site/web application

HTML: the content of the page/application

CSS: the style of the content

Javascript: the logic of the application, sometimes also for animations, etc.

other assets

What is the job of the rendering engine?

Starting from these files (HTML, CSS, JS, etc.), render the web page on the screen of the user.

This is done roughly in four phases:

1. Process the HTML to build the DOM, process the CSS to build the CSSOM
2. Combine DOM and CSSOM together into a render tree
3. Layout the render tree, computing geometries
4. Paint the render tree on the screen

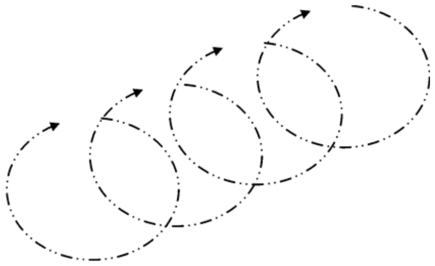
Optimizing this critical path will enable us to display the content to the user as soon as possible.

---

1. Process the HTML to build the DOM, process the CSS to build the CSSOM
2. Combine DOM and CSSOM together into a render tree
3. Layout the render tree, computing geometries
4. Paint the render tree on the screen

The important point to get here is that: **these phases are not strictly consequential**.

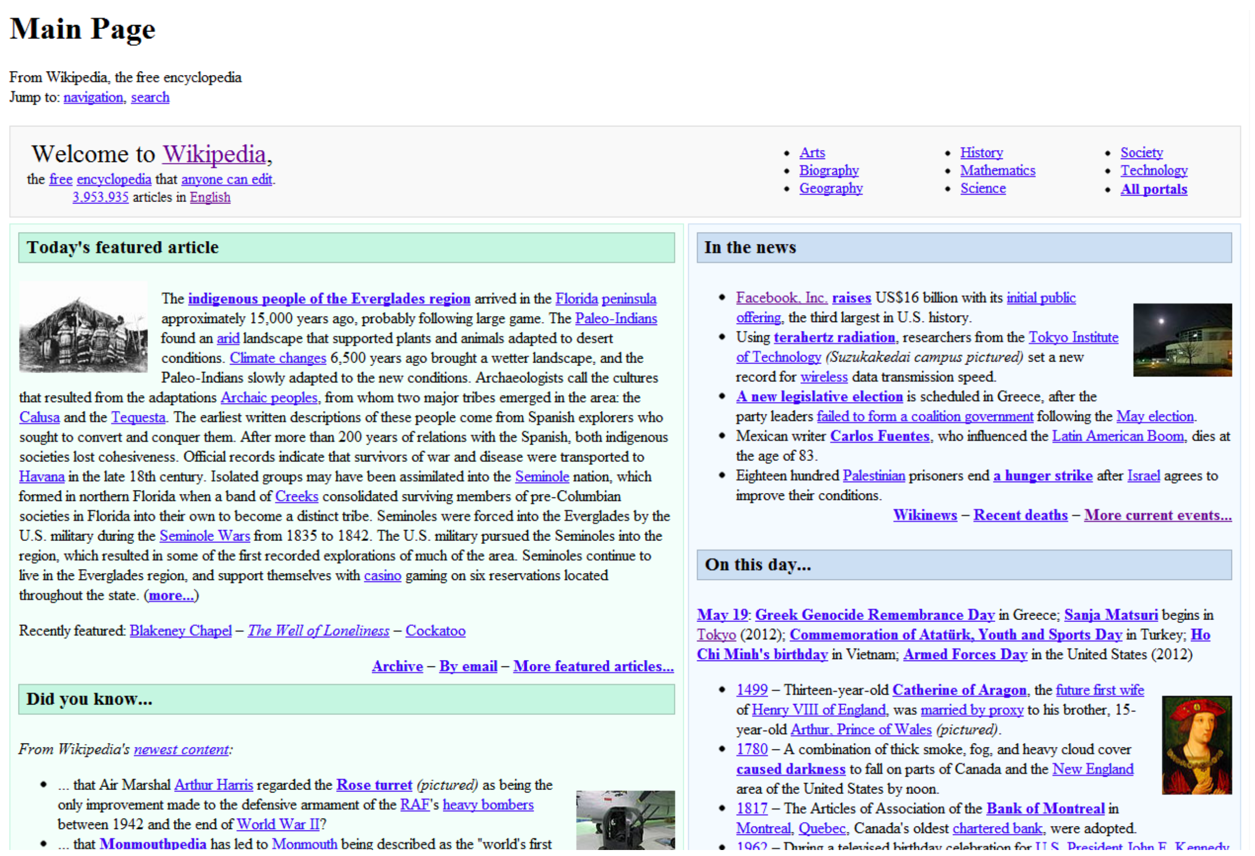
The engine indeed will try to display contents to the user as soon as possible. So phase 2 won't wait phase 1 to be complete before starting, but will start consuming output from phase 1 as soon as possible,



and so will do the other phases.

This gives us an anticipation of an insight that we will see later.

Since browsers try to show the content as soon as possible, in some situations it may happen that content is displayed before the style has been loaded: this is known as the **Flash of unstyled content** (**FOUC**) problem.
An example straight from Wikipedia:



Main Page

From Wikipedia, the free encyclopedia
Jump to: navigation, search

Welcome to Wikipedia,
the free encyclopedia that anyone can edit.
3,953,935 articles in English

- Arts
- Biography
- Geography
- History
- Mathematics
- Science
- Society
- Technology
- **All portals**

Today's featured article

The **indigenous people of the Everglades region** arrived in the Florida peninsula approximately 15,000 years ago, probably following large game. The Paleo-Indians found an arid landscape that supported plants and animals adapted to desert conditions. Climate changes 6,500 years ago brought a wetter landscape, and the Paleo-Indians slowly adapted to the new conditions. Archaeologists call the cultures that resulted from the adaptations Archaic peoples, from whom two major tribes emerged in the area: the Calusa and the Tequesta. The earliest written descriptions of these people come from Spanish explorers who sought to convert and conquer them. After more than 200 years of relations with the Spanish, both indigenous societies lost cohesiveness. Official records indicate that survivors of war and disease were transported to Havana in the late 18th century. Isolated groups may have been assimilated into the Seminole nation, which formed in northern Florida when a band of Creeks consolidated surviving members of pre-Columbian societies in Florida into their own to become a distinct tribe. Seminoles were forced into the Everglades by the U.S. military during the Seminole Wars from 1835 to 1842. The U.S. military pursued the Seminoles into the region, which resulted in some of the first recorded explorations of much of the area. Seminoles continue to live in the Everglades region, and support themselves with casino gaming on six reservations located throughout the state. (**more...**)

Recently featured: Blakeney Chapel – *The Well of Loneliness* – Cockatoo

Archive – **By email** – **More featured articles...**

Did you know...

From Wikipedia's newest content:

- ... that Air Marshal Arthur Harris regarded the **Rose turret** *(pictured)* as being the only improvement made to the defensive armament of the RAF's heavy bombers between 1942 and the end of World War II?
- ... that **Monmouthpedia** has led to Monmouth being described as the "world's first

In the news

- Facebook, Inc. **raises** US$16 billion with its initial public offering, the third largest in U.S. history.
- Using **terahertz radiation**, researchers from the Tokyo Institute of Technology *(Suzukakedai campus pictured)* set a new record for wireless data transmission speed.
- **A new legislative election** is scheduled in Greece, after the party leaders failed to form a coalition government following the May election.
- Mexican writer **Carlos Fuentes**, who influenced the Latin American Boom, dies at the age of 83.
- Eighteen hundred Palestinian prisoners end **a hunger strike** after Israel agrees to improve their conditions.

Wikinews – **Recent deaths** – **More current events...**

On this day...

May 19: **Greek Genocide Remembrance Day** in Greece; **Sanja Matsuri** begins in Tokyo (2012); **Commemoration of Atatürk, Youth and Sports Day** in Turkey; **Ho Chi Minh's birthday** in Vietnam; **Armed Forces Day** in the United States (2012)

- 1499 – Thirteen-year-old **Catherine of Aragon**, the future first wife of Henry VIII of England, was married by proxy to his brother, 15-year-old Arthur, Prince of Wales *(pictured)*.
- 1780 – A combination of thick smoke, fog, and heavy cloud cover **caused darkness** to fall on parts of Canada and the New England area of the United States by noon.
- 1817 – The Articles of Association of the **Bank of Montreal** in Montreal, Quebec, Canada's oldest chartered bank, were adopted.
- 1962 – During a televised birthday celebration for U.S. President John F. Kennedy

PHASE 1 -- Process the HTML to build the DOM, process the CSS to build the CSSOM

In this phase, the engine will start the parsing of the HTML.

The output of the parsing will be a tree called **Document Object Model** (DOM) or also **content tree**, where each HTML tag is represented by a node of the tree (**DOM node**).

The mapping between HTML tags and DOM nodes is not 1:1 but almost.
The root element of the tree is the <html> element.

The purpose of the DOM is at least dual:

- as an input tree for the next phases (to build the render tree)
- as an interface to connect the web page to scripts/programming languages:
    - indeed every DOM node implements an interface that defines how the structure of the tree can be accessed and manipulated
    - examples of common DOM methods are: .getElementById(), .createElement(), .removeChild()

Parsers

Parsers are heavily used in computer science.
They are a software components that take input data, usually text, and build a data structure to be used by the software.

The input data for a parser is most of the times static: it doesn't change during the parsing process.

The HTML parser is an exception. It is a **reentrant parser**: it means that its input can change dynamically while parsing.

This makes this phase more complex and will lead us to our real first insight. Let's see why.

Why HTML parsing is reentrant?

Because:

- **Javascript code is executed immediately when the parser reaches a <script> tag**,
- Javascript can modify the input, e.g. using document.write

The HTML parsing is synchronous

When the parser reaches a <script> tag,

- it **stops parsing**
- it fetches the script from the network, if it is external
- it yields control over to the Javascript engine to executes the code
- the parsing is resumed

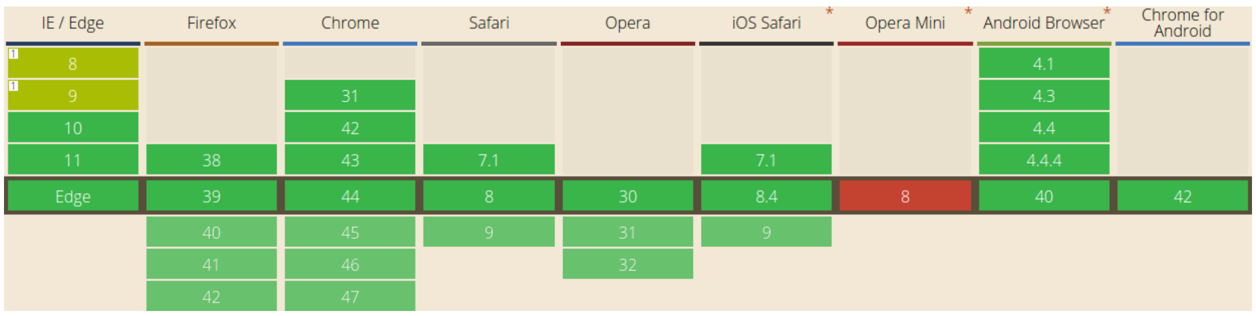
Since <script>s are blocking the parsing, moving them at the bottom of our web page (or just after the *above the fold* content) makes the page rendering faster.

---

defer and **async**

Two attributes of the <script> tag can also help when loading external scripts.

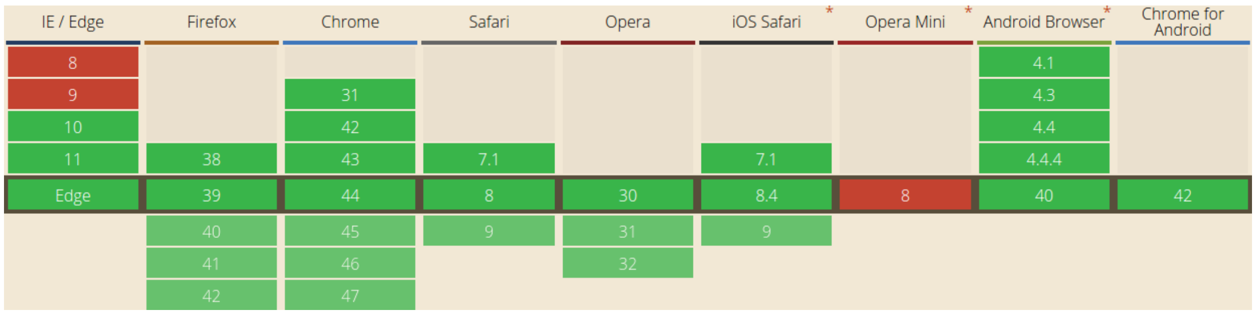
Marking a script with the **defer** attribute, will make the script run only when the DOM has been completely built.



IE / Edge	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
[1] 8							4.1	
[1] 9		31					4.3	
10		42					4.4	
11	38	43	7.1		7.1		4.4.4	
Edge	39	44	8	30	8.4	8	40	42
	40	45	9	31	9			
	41	46		32				
	42	47						

Source: caniuse.com

Marking a script with the **async** attribute, will make the script run only when it is available. without blocking the parsing.



IE / Edge	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
8							4.1	
9		31					4.3	
10		42					4.4	
11	38	43	7.1		7.1		4.4.4	
Edge	39	44	8	30	8.4	8	40	42
	40	45	9	31	9			
	41	46		32				
	42	47						

Source: caniuse.com

Curiosity

In both Gecko and Webkit, when the engine is blocked fetching and executing a script, a second thread starts parsing the document looking for external resources to load: it won’t modify the DOM but just start fetching external resources.

What about external stylesheets?

We saw what happens when the HTML parser meets scripts, what about styles?

Javascript blocked the parsing because it could modify the document.
CSS can’t modify the structure of the document, so it would seem we have no reasons to block the parsing.

Style is blocking

Actually the CSS itself is a blocking resource, for two reasons.

1- Scripts

Javascript could ask for style information that have not been parsed yet.

Browsers deal with this potential problem differently, but roughly:

Mozilla Firefox holds script execution until there are still stylesheets being fetched and parsed

WebKit pauses script execution when they try to access styles that may be affected by stylesheets that still need to be fetched or parsed.

This means that an external script in the head of the page on Firefox:

- blocks parsing
- fetch the external script file
- wait for previous stylesheet to be fetched, parsed and the CSSOM to be built
- execute the script
- resume parsing

As before, this can be avoided moving scripts as much to the bottom as possible.

---

2 - Rendering

The browser will hold the rendering process until all the CSSOM has been built.

This can be mitigated by:

- again, pushing scripts at the bottom of the page
- delivering the stylesheets as soon as possible:
    - splitting stylesheets by media type and media queries can help: they allow us to mark some CSS resources as non-render blocking
        - Note that non-render blocking resources will still be downloaded by the browser but with lower priority

---

Network

We talked about fetching files from the network.
Is there something we need to know about how the files are fetched?

HTTP/1.1

The RFC 2616, released in 1999, defines the HTTP protocol version 1.1.
In the section *8.1.4 Practical Considerations*, it says:

«A single-user client SHOULD NOT maintain more than 2 connections with any server or proxy.»

What does this mean to us?

Imagine having this external files in the head of your web page:

- my_style1.css
- my_style2.css
- my_style3.css
- my_style4.css

The download of file *my_style3.css*, for example, won't start until we first completed download of *my_style1.css* (assuming a sequential order of the downloads).

The limits are per domain. So if two file were on www.example.com and other two on assets.example.com the file could be downloaded at the same time (even if the two domains map to the same IP address).

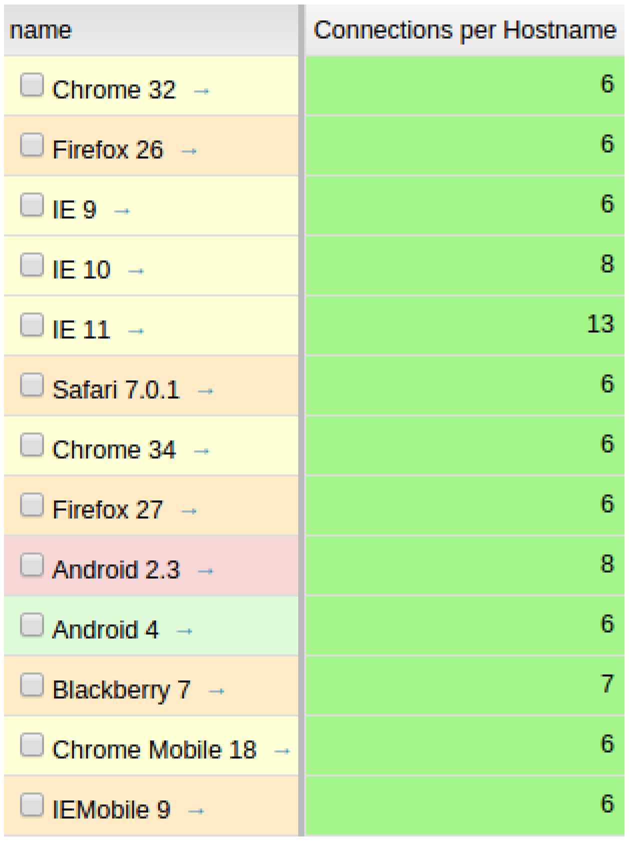
It is important to make as few HTTP requests as possible during page load and, to avoid the limit, it is good to distribute the files on different domains.

Anyway,

1) the specification said SHOULD NOT: so it was just a guideline and web browsers don't had to follow that; indeed most browsers have a limit that is greater than 2

2) in 2014 RFC2616 has been **replaced** by multiple RFCs (7230-7239) that, among other things, **removed the limit of two connections**.

This is the connections per hostname limit today according to browserscope.org:



name	Connections per Hostname
Chrome 32 →	6
Firefox 26 →	6
IE 9 →	6
IE 10 →	8
IE 11 →	13
Safari 7.0.1 →	6
Chrome 34 →	6
Firefox 27 →	6
Android 2.3 →	8
Android 4 →	6
Blackberry 7 →	7
Chrome Mobile 18 →	6
IEMobile 9 →	6

So it is higher but nevertheless it is good practice to reduce the number of HTTP requests, to load the webpage faster.

CSS

So we have seen roughly the main insights of the parsing of the HTML. Let's have a look at the CSS.

All the CSS of a page is parsed in a tree called the **CSS Object Model** (**CSSOM**).

It easy easy to see why the HTML page is represented by a tree: the code itself is a tree-like representation of the page contents.
But why also the CSS is represented by a tree?

The answer for this is in the name itself: **Cascading** Style Sheet.

Cascading

Every element in a page can be matched by more than one CSS rule, so how can the browser choose which property to apply to the element?

Ordering them by:

- origin
- weight
- **specificity**

CSS origin

The CSS can have different origins, in order of precedence:

- Author defined -- styles defined by the author of the page, i.e. by us
- User defined -- these are styles defined by the user of the browser
- User Agent -- these are the browser default styles

Weight
Each property can have a normal weight or a !important weight.

Specificity
This is what we need to pay more attention to.

Note: the weight (!important) has been introduced by the authors of CSS to let browsers users override web pages styles if they wanted. It was not intended for developers use! Using !important in our stylesheet is bad practice and a sign that we may be missing how specificity works.

Specificity

This is one of the main sources of confusion when working with CSS.

The specificity of a rule is represented by a group of four numbers **(a, b, c, d)**

a. The first number is 1 if the rule is inline (defined in the style attribute or in one of the obsolete style related attributes like e.g. bgcolor), 0 otherwise
b. The second number, is the number of the IDs appearing in the rule
    i. for example, div.class_name has 0 IDs, but #sidebar #widget .class_name has two IDs
c. The third number, is the number of classes, pseudo classes and attributes appearing in the rule
d. The fourth number, is the number of elements (tag names) and pseudo elements appearing in the rule

The final group (a, b, c, d) will be considered like a single number in a base that is greater or equal to the max number appearing in the group.

Example:

HTML
```
<div id="sidebar">
    <div id="widget" class="class-div">
        <span class="span-class" style="color: red">Hello, world!</span>
    </div>
</div>
```

CSS
```
.span-class {      /* Specificity (0, 0, 1, 0) */
    color: green;
}

#sidebar #widget {    /* Specificity (0, 2, 0, 0) */
    color: orange;
}

#sidebar {    /* Specificity (0, 1, 0, 0) */
    color: yellow;
}

#sidebar .class-div {    /* Specificity (0, 1, 1, 0) */
    color: blue;
}
```

The inline rule, will have a specificity of (1, 0, 0, 0).

What color will have the text?
The style with the highest specificity will be applied. In this case the inline style has the highest specificity: (1, 0, 0, 0) and so the text will be red.

Which one would be applied if the inline style was not there?
The text would be orange, because that rule has a specificity of (0, 2, 0, 0).

Most of the time that we find ourselves needing !important because we can't get our style applied, we just need to see what is the specificity of the rule being applied (the web browser developer tools are useful for that) and then use a greater specificity for our rule (for example, adding a class to the selector).

Summarizing,
the rules that apply to an element are sorted by:

1) origin
2) weight
3) specificity
4) order of definition

The specificity is taken in consideration only if the rule have the same origin and weight.
If also the specificity is the same, then the rule that has been defined for last is applied.

When parsing the CSS, the CSSOM tree is not the only data structure that will be built by the browser.

The CSS rule matching can be a heavy job.
To make it faster, each rule is added to one of several hash tables, according to the most specific selector of the rule.

There are hash tables for IDs, class names, tag names and a general hash table for rules that don’t fit in any other dictionary.

So when the browser needs to find which CSS rules apply to a specific elements, it doesn’t need to look in every rule but only in the hash tables.

This leads us to another important **insight**: since we start from an element, take its IDs, classes, tag name, etc and look for them in the various dictionaries, **we will always match rules by their rightmost selector**, that for this reason is also called the **key selector**.

At first this may seem a difficult concept to grasp but it is fundamental to understand how to write faster CSS rules.

Let’s see an example:

HTML

```
<div id=”container1”>
    … thousands of <a> elements here …
    <a> … </a>
    … thousands of <a> elements here …
</div>
<div id=”container2”>
   <a class=”a-class”>...</a>
</div>
```

Let’s say we want to select only the <a> in the container2.

This selector:

```
#container2 a {
    …
}
```

will be very expensive. Why? Because, reading the rule left-to-right it reads: take #container2 subtree and find all <a>s. But the browser reads it right-to-left, so it reads: take all <a>s and goes up in the DOM until you find a #container2.
This means that this rule will iterate on the DOM for ALL the <a> of the page, even the thousands that are inside #container1.

We can write a most efficient rule:

```
#container2 .a-class {
    ...
}
```

or just

```
.a-class {
   ...
}
```

if that is the only element in the page.

This insight can be leveraged also from javascript / jquery.

Instead of selecting an element with

```
$('#container .class-name');
```

we can achieve better performances by using:

```
$('.class-name', '#container');
```

In the first case, all the .class-name element of the page will be grabbed and then the DOM will be traversed up looking for the #container element.

In the second case, the #container element will be grabbed (constant time) and then .class-name element will be looked for only in its subtree.

When all the HTML has been parsed, the document will be marked as **interactive** and its state set to complete:

- deferred scripts will be executed
- a load event will be fired

So.. we have seen the most important parts of phase 1 (process the HTML to build the DOM, process the CSS to build the CSSOM).

Let's have a look at the second phase: combining DOM and CSSOM together into a render tree

---

PHASE 2 - Combine DOM and CSSOM together into a render tree

The nodes from the DOM and the CSSOM trees will be combined together in a new tree, the **render tree**.

Its structure is similar to the DOM tree but doesn't match it exactly: only elements to be displayed will be in the tree. For example, <head> and <script> nodes won't be there, and elements with style display: none won't be there as well.
Also, some DOM elements could be represented by more nodes in the render tree (like multi-line text nodes).

So it is a tree of the **visual** elements in the same order in which they will be rendered.

Each node of the render tree stores the CSS properties that apply to it.

This nodes are called differently in different browsers.
WebKit call them **renderers** (hence render tree).
Firefox calls them **frames** and the tree is called **frame tree**.
Sometime they are simply called **boxes** from the concept of CSS box, since each node represent a rectangular area that usually corresponds to the node’s CSS box.

Once we have our render tree containing all the visual elements with their relative styles, we can move to the third stage:

The **layout** phase, also called **flow** (or **reflow**, as we’ll see later)
We now have a tree containing all elements that need to be rendered, in the right order of rendering and with the associated style. What else is missing to be able to render the page?

The geometry of the nodes.

The geometry of a node is its **position** and **dimensions**: sometimes they are specified in the stylesheet, sometimes not. But in any case they need to be computed for all nodes to be able to correctly render them.

The browser traverses once again the render tree, starting from the root, and for each node computes the geometry.

All the relative units used in the stylesheet are converted to pixels.

Final stage

Now our render tree knows:

- which nodes are visible
- in which order they should be rendered
- their style (CSS)
- their geometry (position and dimensions)

This is enough to move to the final phase: rendering the web page.

This phase is also called painting (or **repaint**, as we'll see later)

All nodes are traversed, starting from the root, and their "paint" method is called: each node "knows" how to paint itself.

We went faster on these last stages.
The reason is that we are not interested in the details of how the layout and the paint are accomplished but we need to know that:

- they can be expensive (browser-blocking)
- **they can be retriggered**

Reflows and repaints are unavoidable, but Reducing their frequency will improve the speed of our web site.

What triggers reflows and repaints?

Javascript manipulation of the page; examples:

- DOM manipulation (e.g. adding or removing elements)
- Stylesheet manipulation (e.g. adding or removing CSS rules)

User interactions; examples:

- Triggering a :hover effect
- Scrolling the page
- Entering text in a input box
- Resizing the window

There’s not much we can do to avoid the reflows and repaints triggered by the user, but we can try to make them as less expensive as possible:

- optimizing our CSS and HTML

We have more control over the reflows and repaints triggered by our code.

Browsers are smart (1)

Browsers will try to do the minimal possible actions in response to a change.

Changing a property of an element, for examples, may just trigger a local repaint, i.e. a repaint of only the subtree of the render tree rooted in that element.

Hence, reflows and repaints can be **global** or **incremental**.

Applying changes to subtrees that are as small as possible will make the reflow/repaint faster.

In other words, try modifying elements that are deeper in the DOM and with a low height.

Reflows and repaints don’t have to happen together.

For example, if you change only the color of an element then **only a repaint is triggered**.

If you change the position of an element, both reflow and repaint are triggered.

What is more expensive, reflow or repaint?

We have to consider that the browser will use heavy caching to avoid recalculations.

A repaint requires the browser to search through all the elements to determine what is visible and what should be displayed.

A reflow recalculate the geometry for an element. It will recursively reflow also its children and sometimes also its siblings.

A reflow of course will trigger a repaint to update the webpage.

How to optimize CSS and HTML for minimal reflow

- The leaner the stylesheet, the faster the reflow
- The higher is the DOM (the HTML structure) and the more expensive reflows can be
- Some elements and display mode are more expensive than others:
    - inline CSS styles may trigger a further reflow
    - <table>s with automatic cell widths are expensive since the browser will need more than one pass to calculate the cell dimensions
    - using the flexbox display itself can be expensive, since the geometry of the flex items can change during the parsing

Anyway, this optimizations are not as important as the ones we can do in our javascript code.

Let’s see some javascript code

```
var foo = document.getElementById(‘foobar’);

foo.style.color = ‘blue’;
foo.style.marginTop = ‘30px’;
```

How many reflows and repaints do we trigger?

Browsers are smart (2)

Since reflows and repaints are expensive, the browser will accumulate all the DOM manipulation you do in a timeframe into a queue and will perform them in batch.

Let’s see some other code

```
var foo = document.getElementById(‘foobar’);

foo.style.color = ‘blue’;
var margin = parseInt(foo.style.marginTop);
foo.style.marginTop = (margin + 10) + ‘px’;
```

How many reflows and repaints do we trigger?

Why?
After the first change to the color property, the manipulation has been added to the accumulating queue.
Then, we ask for the style of that element.

The only way for the browser to be sure to give us the right answer, is triggering the batch execution of the queue.
In other words, we are forcing a premature repaint of the page.

How does this work?

When we change an element of the DOM, that element will be flagged as **dirty**. Sometimes it may also have the **children are dirty** flag, meaning that at least one of its children needs a reflow.

Once the time interval for manipulations aggregations runs out, all the dirty elements are reflowed and repainted.
Asking some properties from an element that has been marked as dirty force the browser to perform the reflow early.

Forced synchronous layout or **layout trashing**

Interleaving a lot of reads and writes to the DOM can lead us to what is called as layout trashing, a performance killer.

How can we avoid layout trashing?

- Reordering the commands to group together DOM reads and DOM writes
- Cache computed styles
- Don't change styles one by one, but use CSS classes
    - This should be normal practice, since CSS stylesheet is where styles belong
    - Anyway sometimes the styles need to be computed dynamically
- Manipulate elements outside the DOM
    - document fragments can help here and they are is widely supported by browsers
- When possible, set position to absolute or fixed to elements that are going to be animated; in this way the updated geometries of these elements won't affect other elements
- Elements that are not displayed can't cause any reflow or repaint: only display them when necessary
- Use window.requestAnimationFrame()
- Use a virtual DOM library

---

requestAnimationFrame

window.requestAnimationFrame() allows us to execute some code at the next reflow.

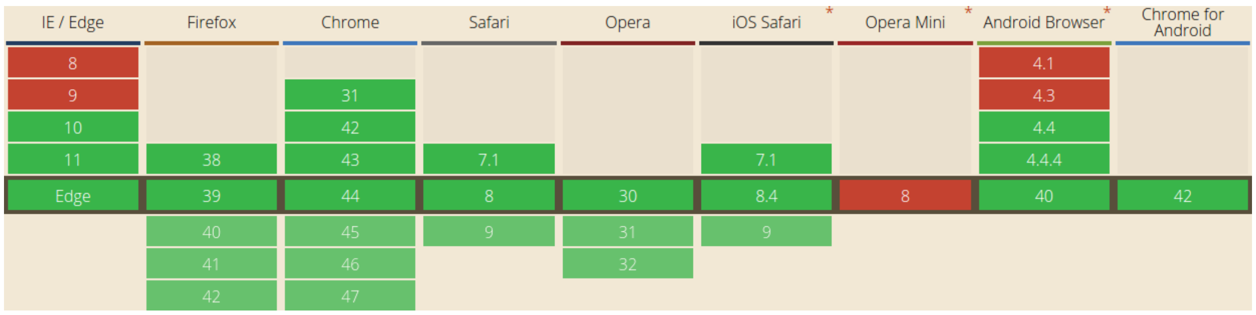
This is very useful because it allows us to interleave reads and writes and at the same time to execute them in the best way.

```
function doubleHeight(element) {
    var currentHeight = element.clientHeight;
    element.style.height = (currentHeight * 2) + ‘px’;
}

all_my_elements.forEach(doubleHeight);
```

Using window.requestAnimationFrame:

```
function doubleHeight(element) {
    var currentHeight = element.clientHeight;
    window.requestAnimationFrame(function () {
        element.style.height = (currentHeight * 2) + ‘px’;
    });
}

all_my_elements.forEach(doubleHeight);
```

RAF support



IE / Edge	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
🟥 8							🟥 4.1	
🟥 9		🟩 31					🟥 4.3	
🟩 10		🟩 42					🟩 4.4	
🟩 11	🟩 38	🟩 43	🟩 7.1		🟩 7.1		🟩 4.4.4	
🟩 Edge	🟩 39	🟩 44	🟩 8	🟩 30	🟩 8.4	🟥 8	🟩 40	🟩 42
	🟩 40	🟩 45	🟩 9	🟩 31	🟩 9			
	🟩 41	🟩 46		🟩 32				
	🟩 42	🟩 47						

Source: caniuse.com

Anyway javascript developers have been using a trick for a long time to achieve roughly the same result:

window.setTimeout(function () {...}, 0);

Setting a timeout with a zero interval will usually invoke the callback function at the next reflow.

Virtual DOM libraries

Virtual DOM libraries are gaining a lot of momentum lately, thanks to Facebook’s **React** library.

How does a virtual DOM works?

The developer performs the manipulations on a virtual DOM that will then:

- aggregate the manipulations together
- apply some heuristics to simplify the changes to apply to the DOM
- cache values
- execute the manipulation at the right moment to avoid a layout trashing

---

Thanks