HTTP to screen

Vincent Sanders

Intelligent design?

- Started as trivial declarative tagged markup
- Exploded and fragmented into many dialects
- W3C attempted some standardisation
- Best we have now is living standards

Anatomy of a browser

- User Interface

Anatomy of a browser

- User Interface
- Browser Engine

Anatomy of a browser

- User Interface
- Browser Engine
- Render Engine

Anatomy of a browser

- User Interface
- Browser Engine
- Render Engine
- Support elements
  - Protocol handlers

Anatomy of a browser

- User Interface
- Browser Engine
- Render Engine
- Support elements
  - Protocol handlers
  - URL handling

Anatomy of a browser

- User Interface
- Browser Engine
- Render Engine
- Support elements
  - Protocol handlers
  - URL handling
  - JavaScript

Anatomy of a browser

- User Interface
- Browser Engine
- Render Engine
- Support elements
  - Protocol handlers
  - URL handling
  - JavaScript
  - Plotting

Anatomy of a browser

- User Interface
- Browser Engine
- Render Engine
- Support elements
  - Protocol handlers
  - URL handling
  - JavaScript
  - Plotting
  - Persistent storage

Browser Engine

- Orchestrates all the other components

Image Renderers

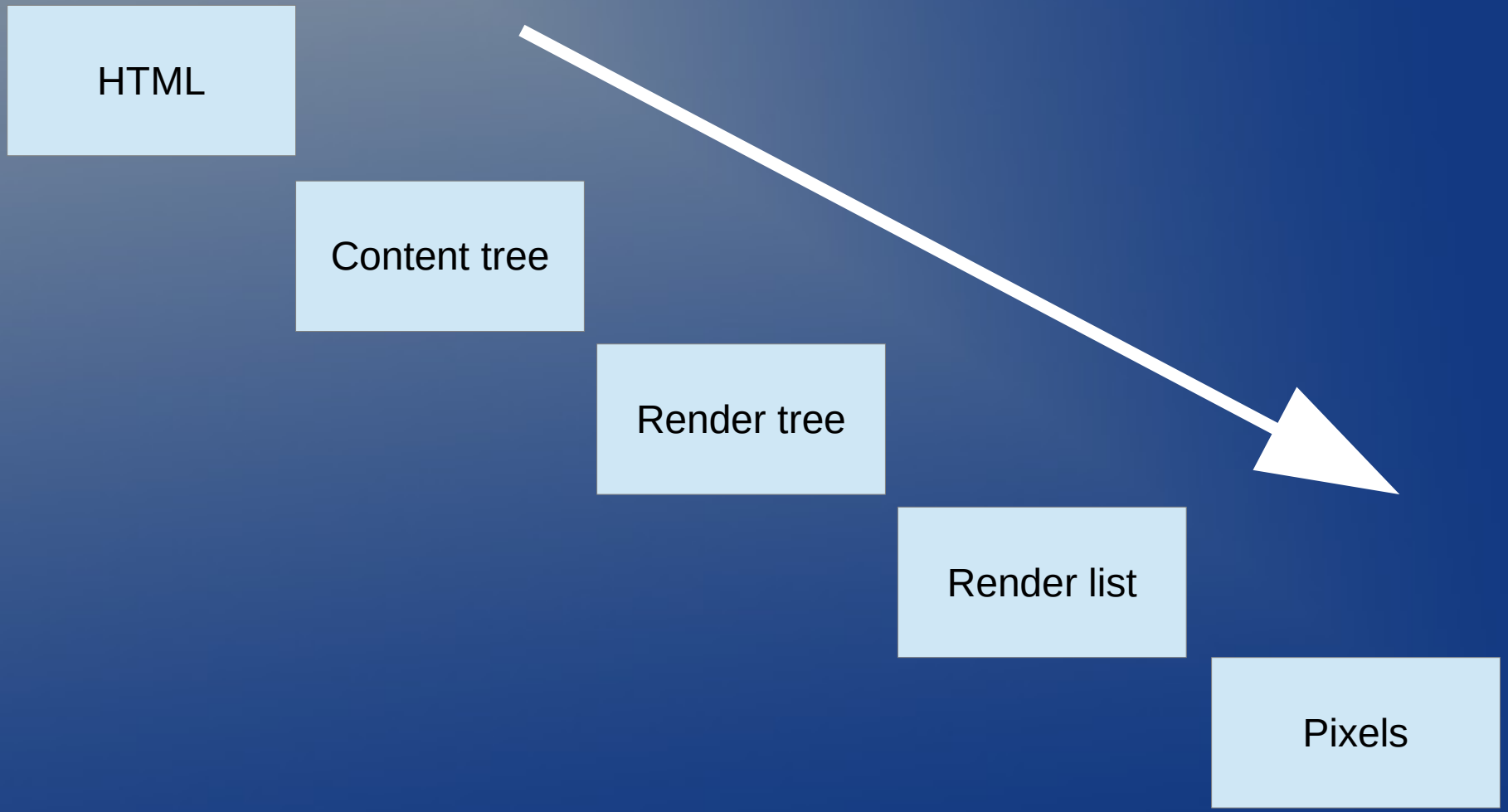
- Process raw data from protocol handler
- Convert to representation suitable for plotting
- Has to be massively robust

Other Renderers

- Other formats like PDF
- Converts to representation suitable for plotting
- Must be robust

HTML Renderer

- Where HTML gets converted to actual plotting



Main flow parsing

- Parse HTML into render tree

Main flow rendering

- Create render tree from content tree

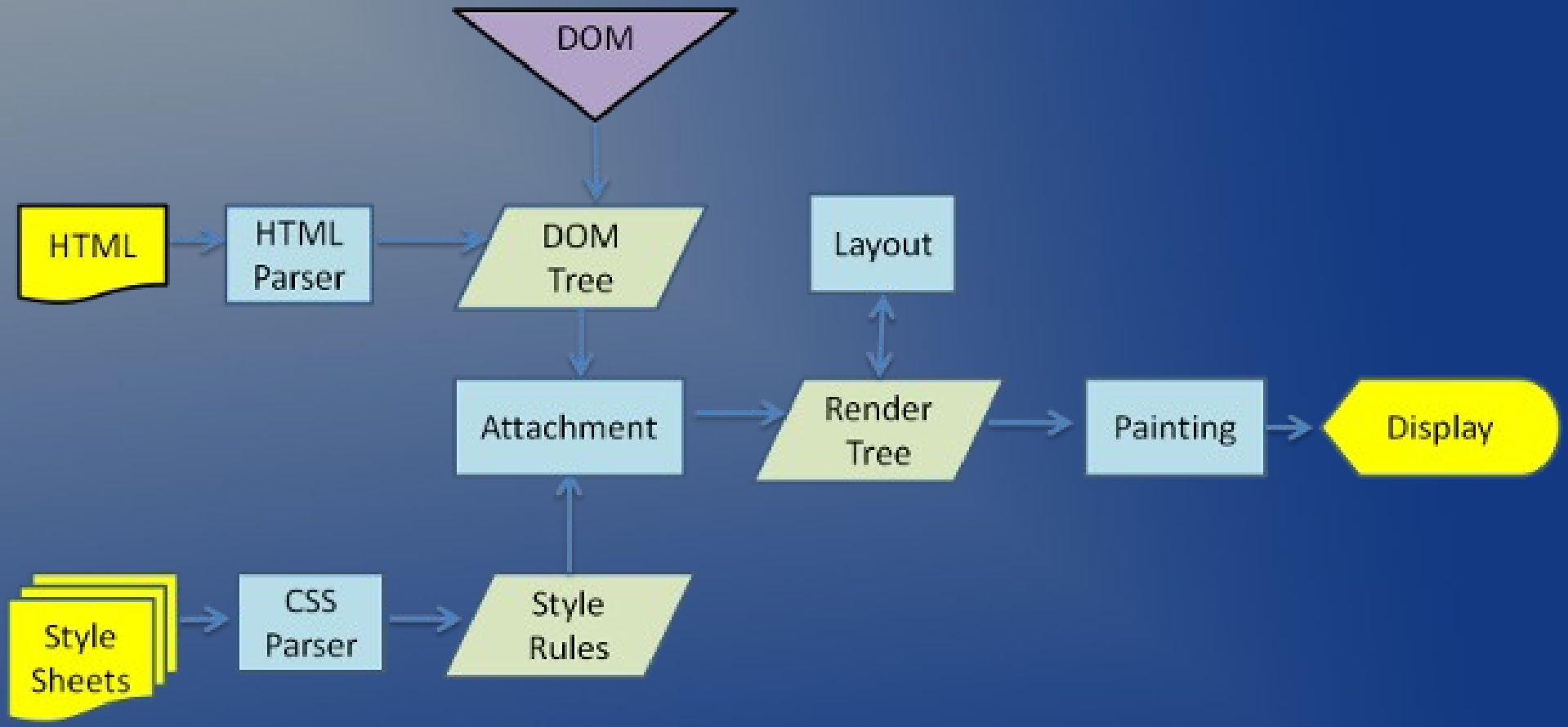
Man flow layout

- Convert the render tree to a render list

Main flow plotting

- Use the render list to put pixels on screen

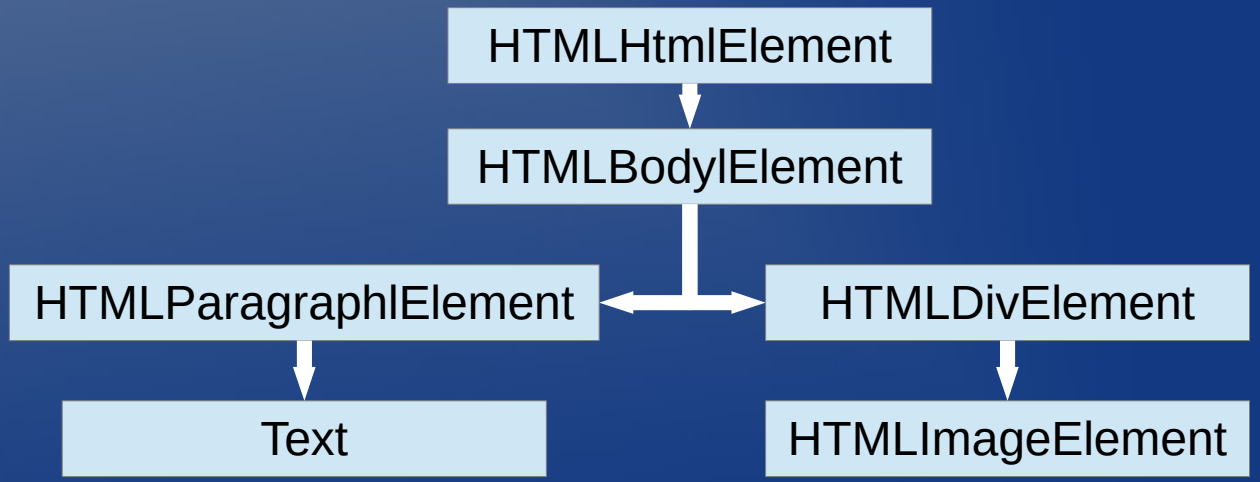
Example flow



HTML Parsing

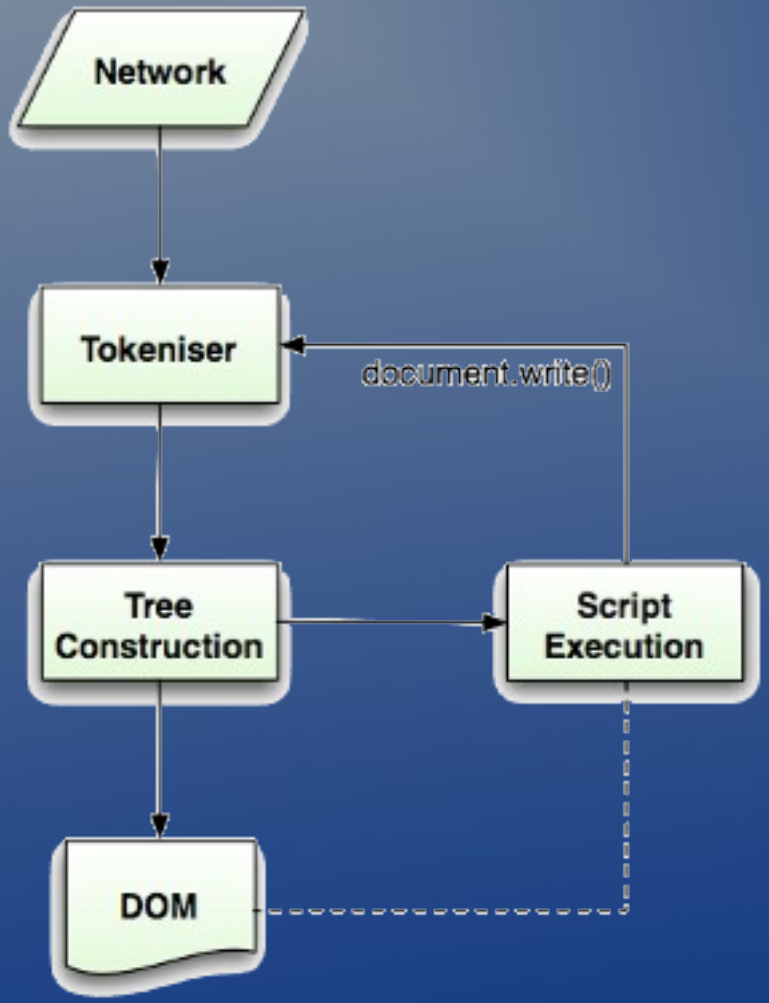
- Not context free grammar
- Another living specification

```
<html>
  <body>
    <p>
      Hello World
    </p>
    <div> <img src="example.png"/></div>
  </body>
</html>
```



HTML Parsing

- HTML parsing is re-entrant!



HTML Parsing

- HTML parsing is re-entrant!

HTML Parsing

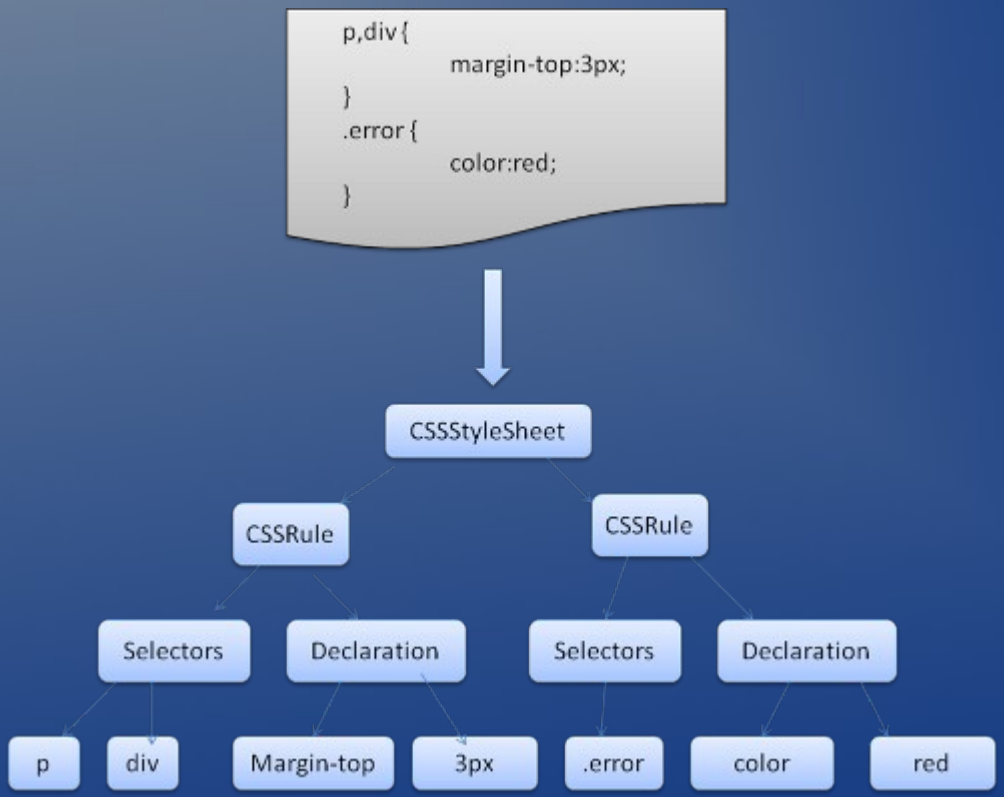
- When it is over there is still lots to do.

Parser Error Handling

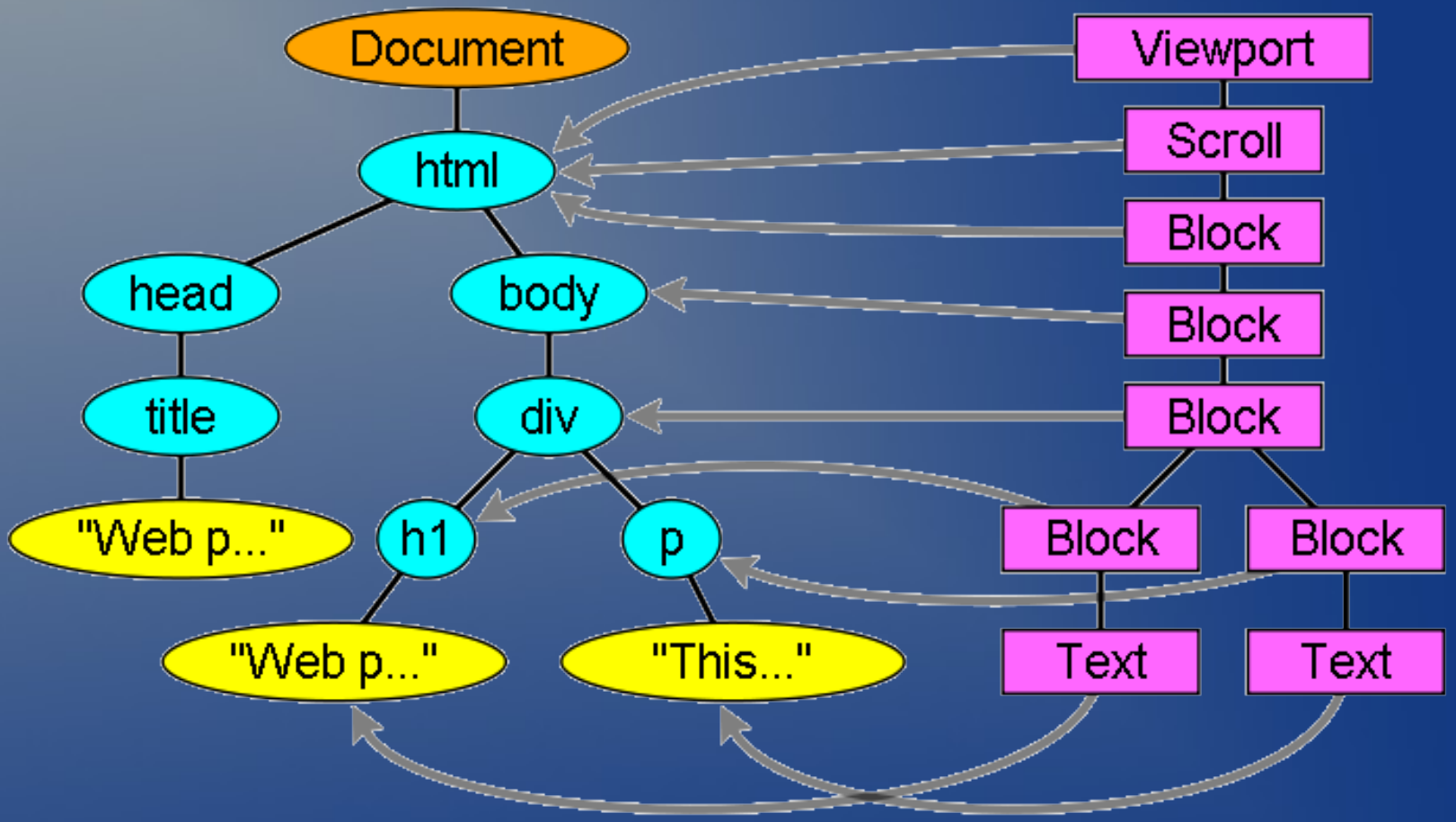
- Browsers never error out.

CSS Parsing

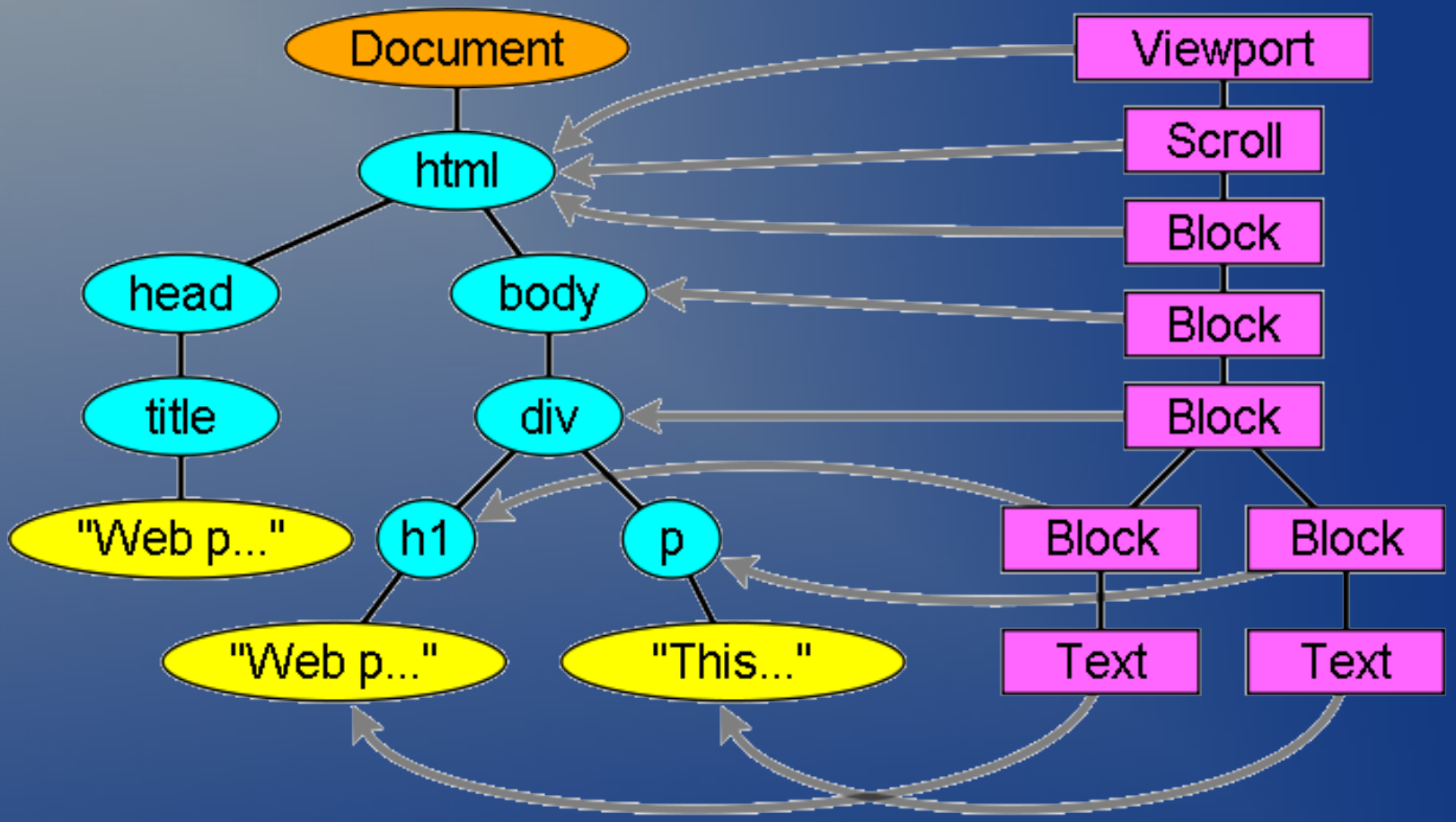
- Is context free
- Well specified



Render tree


Document
html
head
body
title
div
"Web p..."
h1
p
"Web p..."
"This..."
Viewport
Scroll
Block
Block
Block
Block
Block
Text
Text

Render tree


Document
html
head
body
title
div
"Web p..."
h1
p
"Web p..."
"This..."
Viewport
Scroll
Block
Block
Block
Block
Block
Text
Text

Layout

- Converts the render tree to a render list
- Recursive process
- Co-ordinates relative to top left of viewport
- Makes concrete decisions on where the render objects are placed.

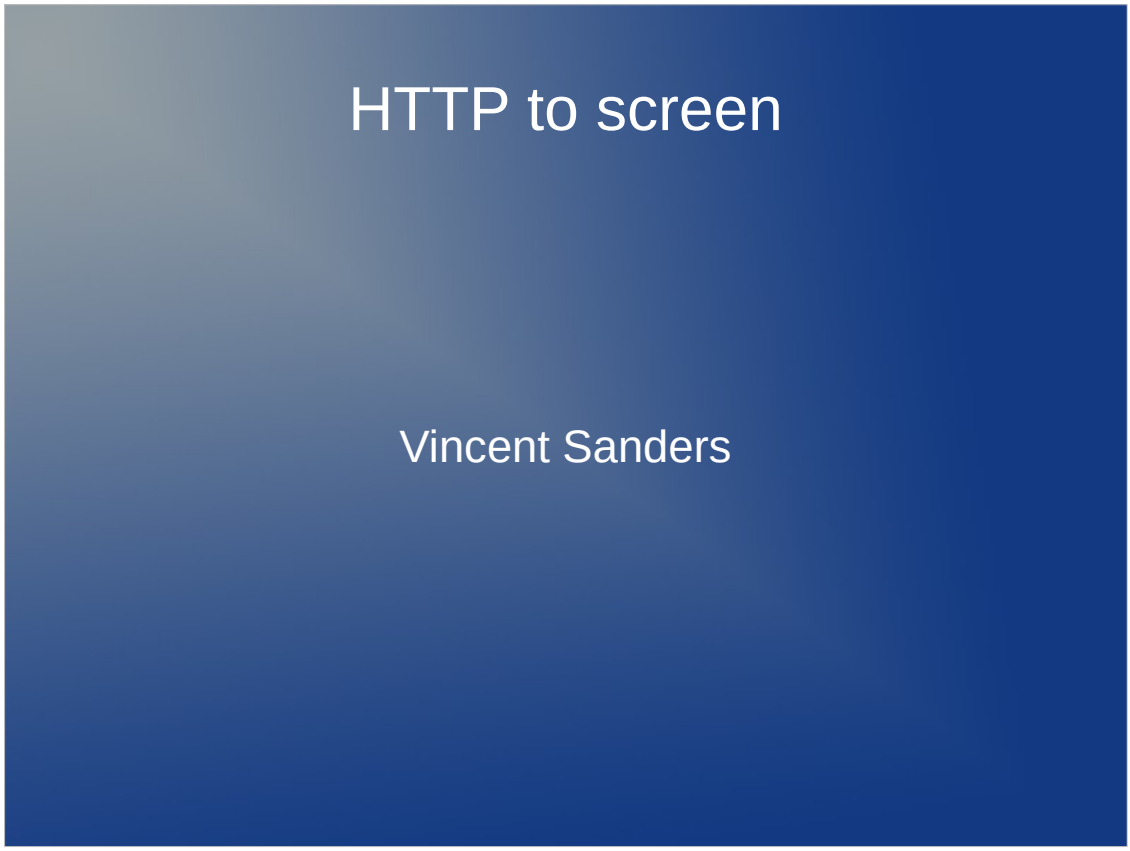
Plotting

- Putting the pixels on screen

But wait, there is more!

- Dynamic content

Any Questions?



Introduce self

Mention Cosworth sponsorship and are hiring.

Going to take a look at what it takes for a web browser to display something

Sir Tim invented the web in the early nineties and it was all simple text documents edited by hand.

Then everyone saw it was good and hacked and hacked and hacked.

Lots of features like images and CSS for styling and less usefully marquee. This was web 1.0

We ended up with “best viewed in” and were in danger of walled gardens

Then came web 2.0 and javascript and it got a whole lot worse.

W3C “standardisation” helped a bit and now we have living standards which are just a polite way of tracking what most browsers actually do.

Anatomy of a browser

- User Interface

The high level elements of a browser are relatively easy to identify, if rather challenging to implement

The user interface is all the “stuff” around the displayed web page.

Rather startlingly this is not specified formally anywhere. Convention causes most browsers to have the usual address bar, the back/forward/home etc. but this is because “that is how everyone does it”

NetSurf takes the approach of splitting these into numerous native toolkit “frontends”.

Other browsers seek to make their furniture common across as many OS and toolkits as possible.

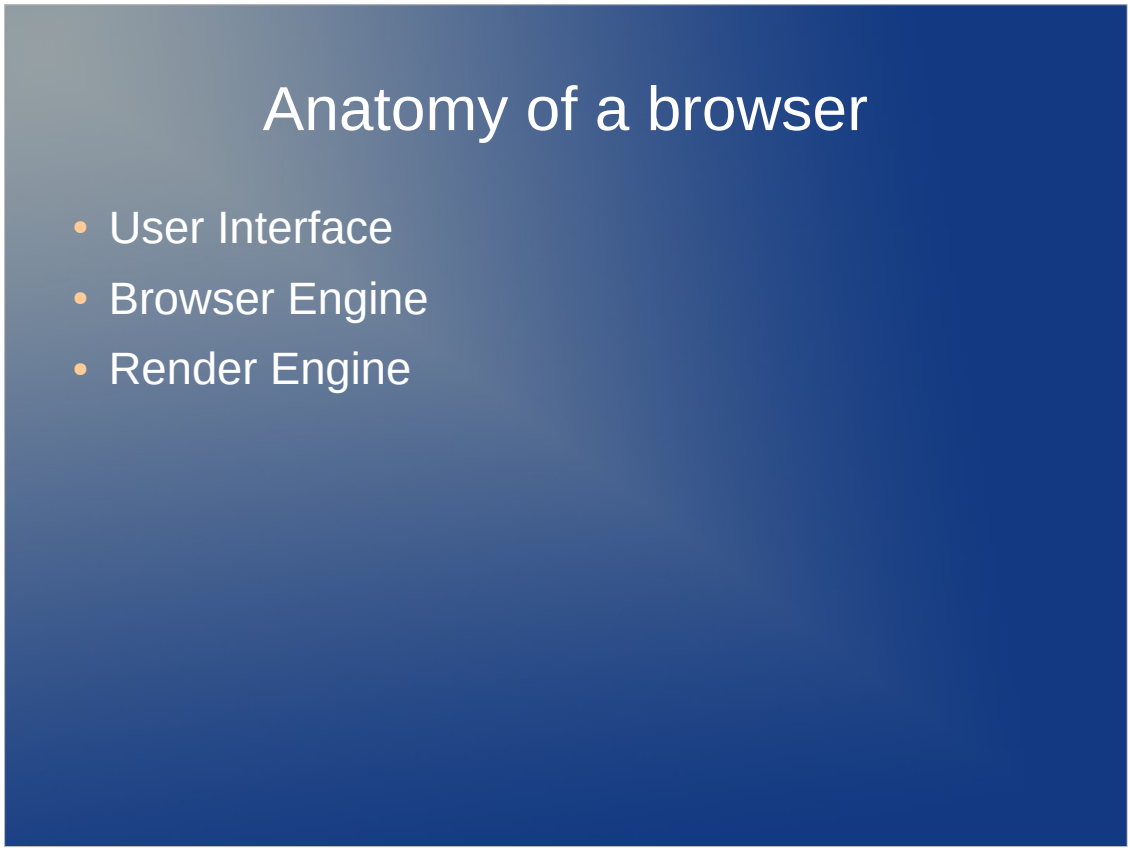
Pros and cons to both approaches native app vs familiar UI

Anatomy of a browser

- User Interface
- Browser Engine

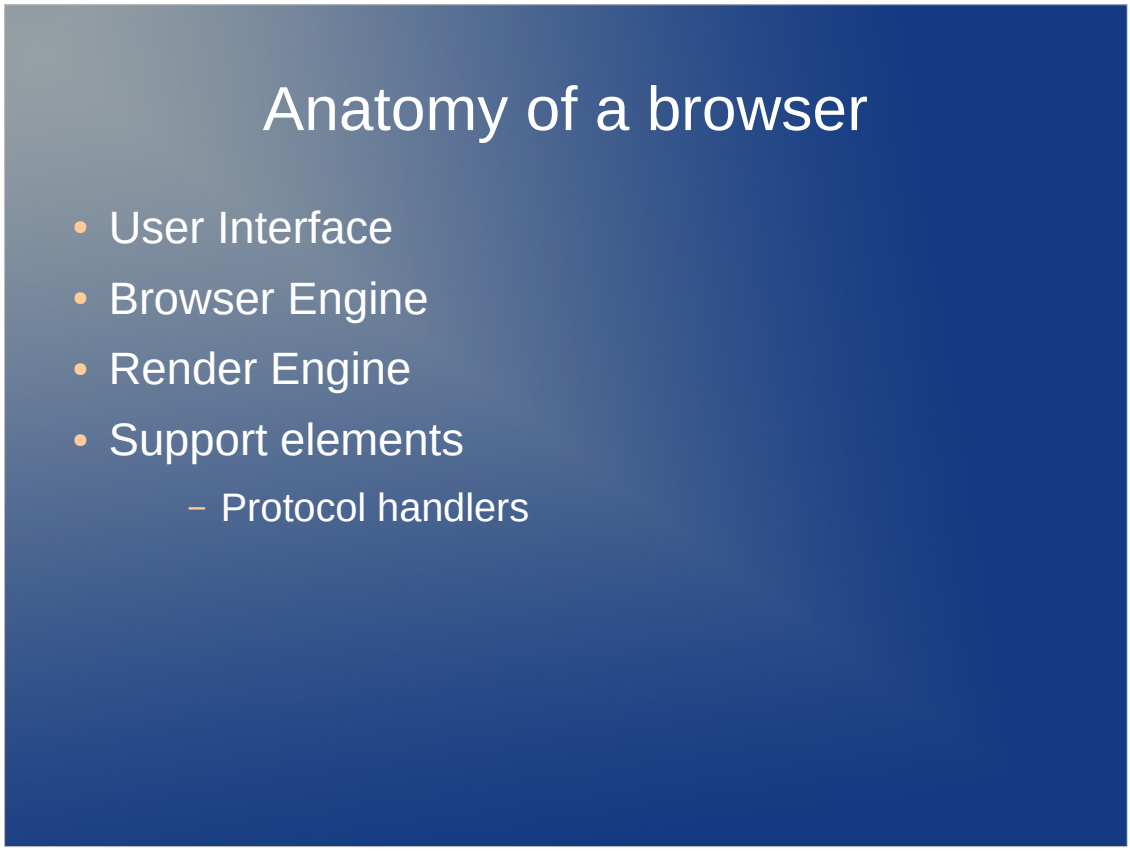
The Browser engine controls the interactions between the UI and the other browser elements

It initiates the fetches given a URI and manages the browsing contexts (tabs, windows etc)



responsible for displaying requested content.

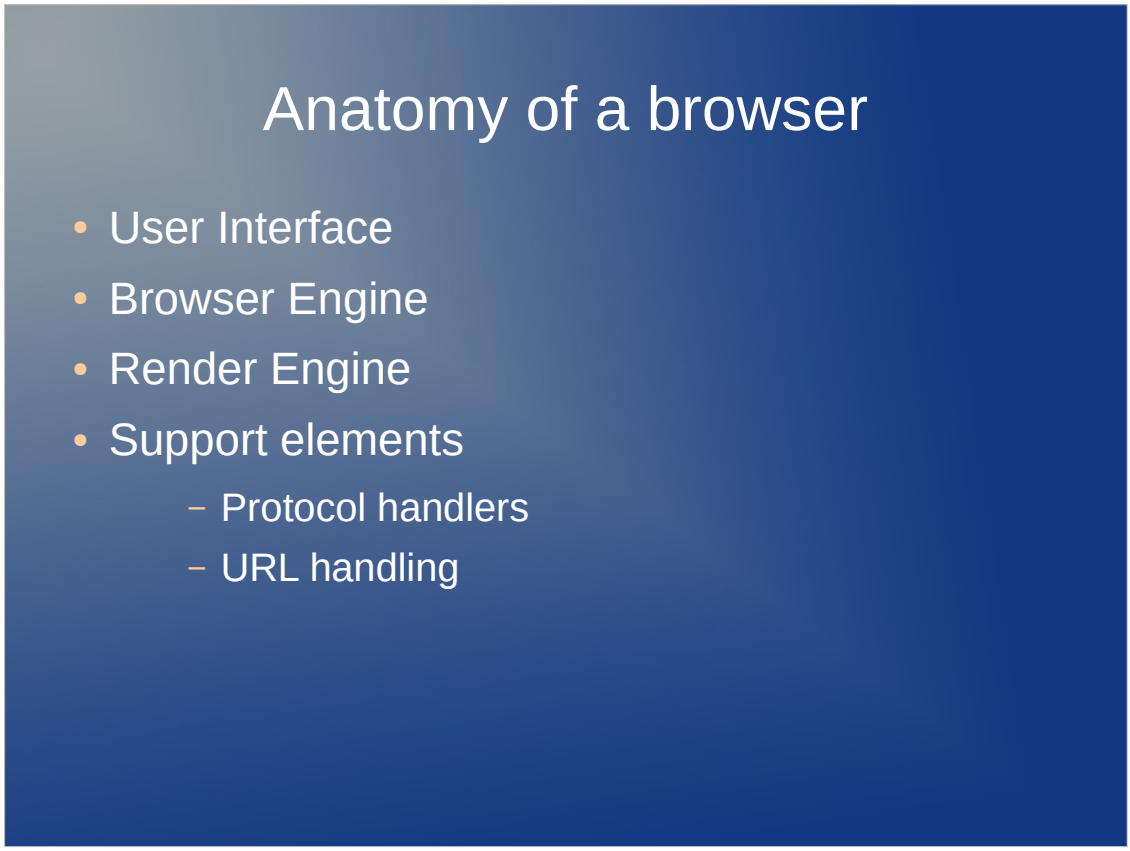
For example if the requested content is HTML, the rendering engine parses HTML and CSS, runs javascript and displays the parsed content on the screen.



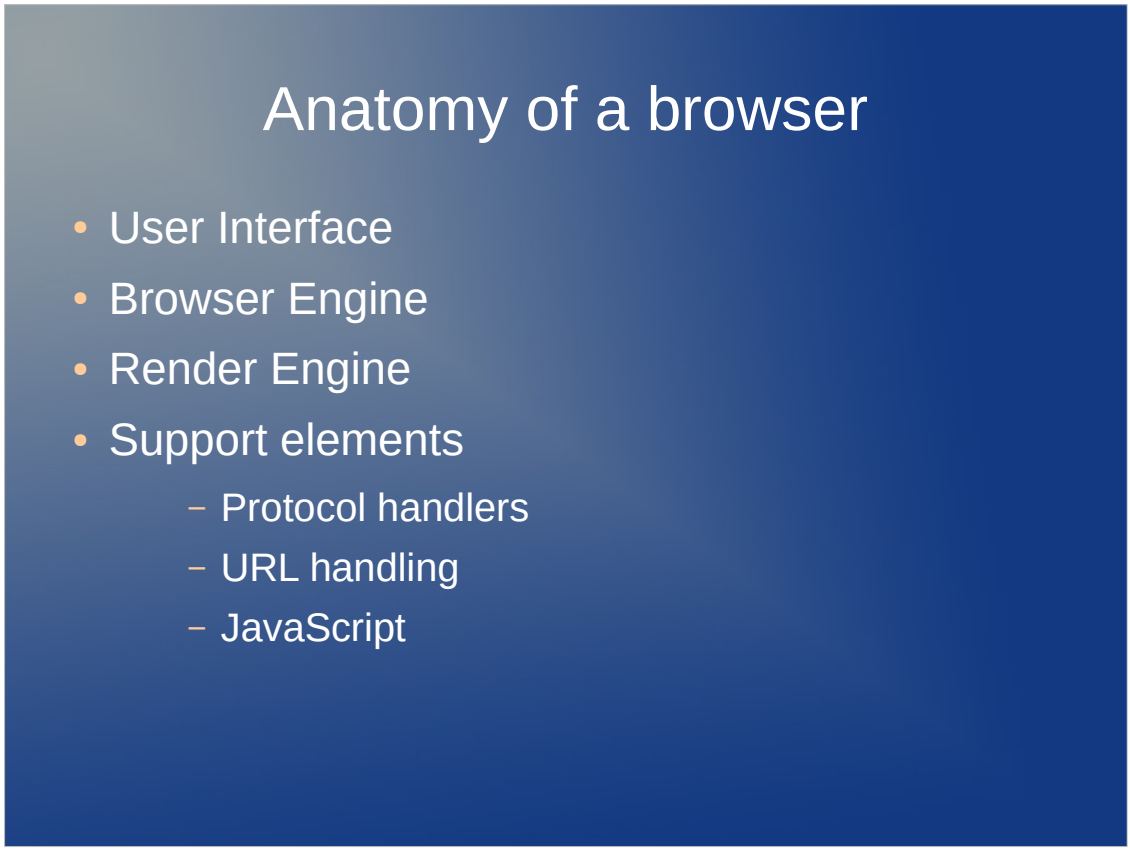
Support elements is a catch all for everything needed to support the other elements.

Protocol handlers for dealing with http etc. NetSurf chose to wrap curl but many other browsers implement their own.

For years RFC2616 was the http reference but now we have RFC7230 to RFC7235 and http2 in RFC7240. Given the thousands of pages of spec one can immediately understand the complexity here



URL handling (there is a living standard for that) https://url.spec.whatwg.org/ complete with utf-8 and IDNA handling nightmares. fix DNS? No! Lets add a hideously complex encoding instead

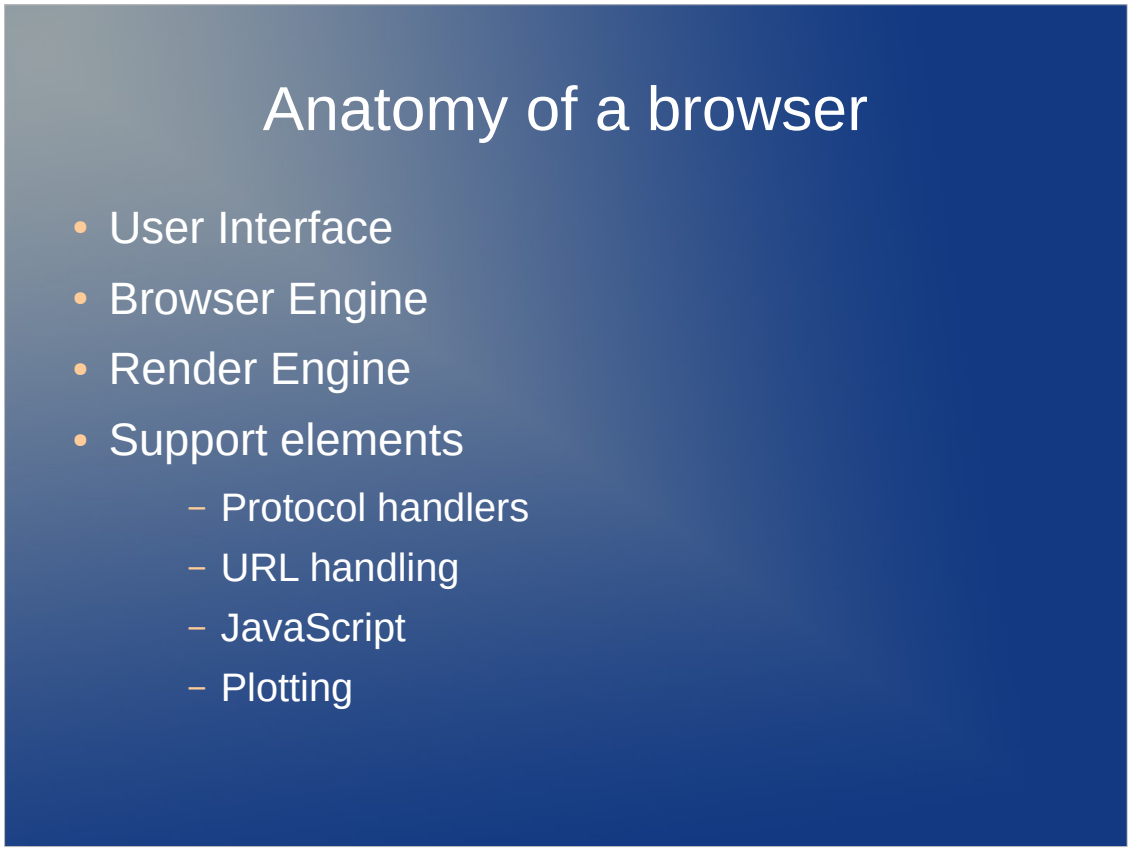


JavaScript engine. Limited options until recently. Mozilla had Spidermonkey and Google had V8. Recently both of those have become implementation details of their browsers and are no longer easily usable libraries.

Duktape is a nice library NetSuf is using with a great upstream, focus on size rather than speed but that fits.

The engine is bolted to the browser through interfaces defined in WebIDL throughout the specs...all over the place. Extracted by scraping the specs

>400 interfaces (classes) >4000 operations and attriutes (methods and properties) NetSurf uses a tool to generate c to glue the js interfaces to the DOM



Plotting and compositing. The render component needs to be able to get pixels on screen and there needs to be a library for it. In NetSurf its intimately tied to the UI so uses cairo on GTK and GDI on windows. Firefox and Chrome have similar components

Anatomy of a browser

- User Interface
- Browser Engine
- Render Engine
- Support elements
  - Protocol handlers
  - URL handling
  - JavaScript
  - Plotting
  - Persistent storage

Persistent data storage, a browser needs to store cookies, page scroll offsets, favicons and all manner of state. Add in javascript local storage and websql and you need a robust storage solution.

Browser Engine

- Orchestrates all the other components

This component is largely ignored when speaking about browsers. The render engines gets all the glory.

This component causes objects to be fetched using the protocol handlers, ensures the correct renderer component is used I.e html renderer for text/html or jpeg renderer for binary/jpeg often performs the content snooping spec to derive the correct mime type

Yes it is terrifying how many web servers give completely incorrect mime types out often icons are pngs, jpegs are gifs etc.

It maps the UI to the browsing contexts so the contents of each window or tab is what its supposed to be and the UI can show all the right things about the page like bookmark status and security certificates.

Also ensures periodic things like cache management happen.

Image Renderers

- Process raw data from protocol handler
- Convert to representation suitable for plotting
- Has to be massively robust

There are several image renderers, one for each supported image mime type

One of these for each image type gif, png, jpeg, bmp webp. Well ok maybe not webp ;-)

These renderers get fed a lot of garbage. Some tests while developing NetSurf renders showed that over 10% of the webs images have one issue or another. Browsers cannot return errors so we just have to do the best we can.

For even reasonable performance these renderers need to employ several methods to get the pixels plotted quickly.

The source image formats are efficient for transfer but rarely for plotting so often renderers use uncompressed internal representations to make plotting rapid.

This introduces the problem that raw bitmaps are huge, so images only get converted when used and that data cached and discarded when the images are not being plotted

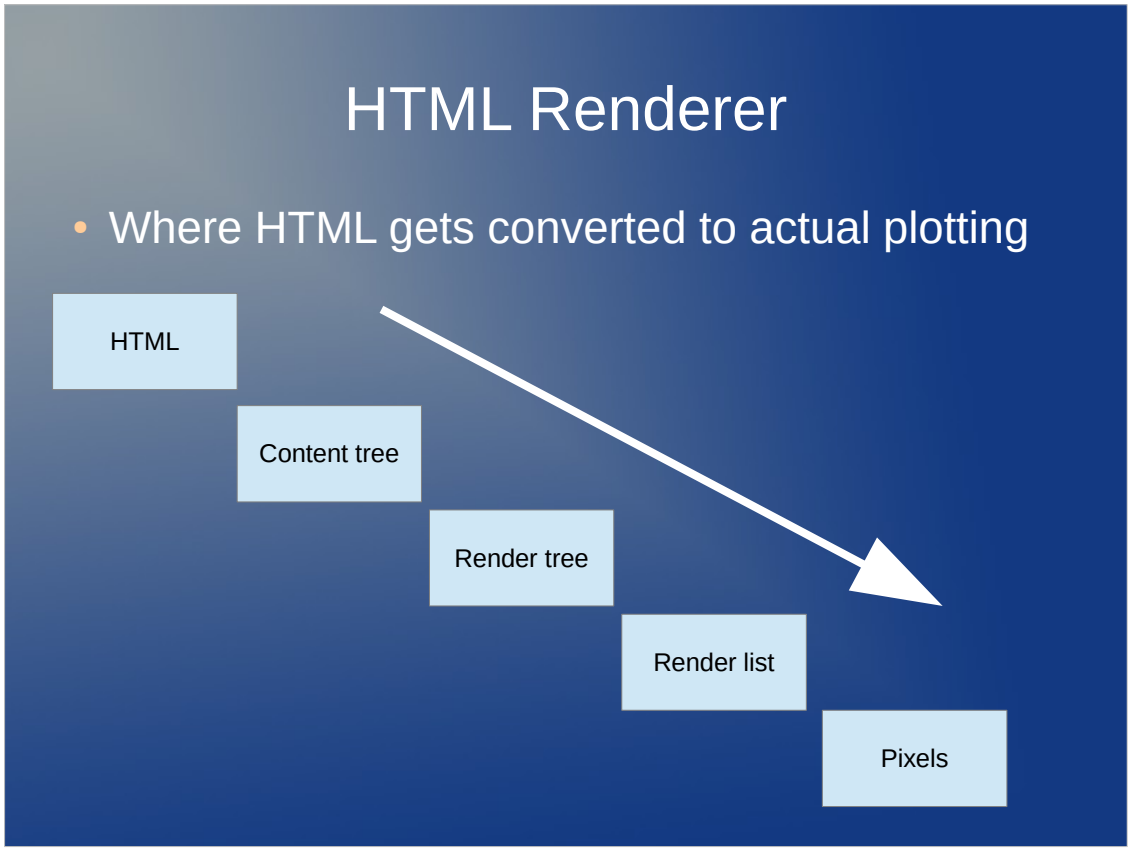
Other Renderers

- Other formats like PDF
- Converts to representation suitable for plotting
- Must be robust

These renderers handle everything that is not html. So pdf or plugins

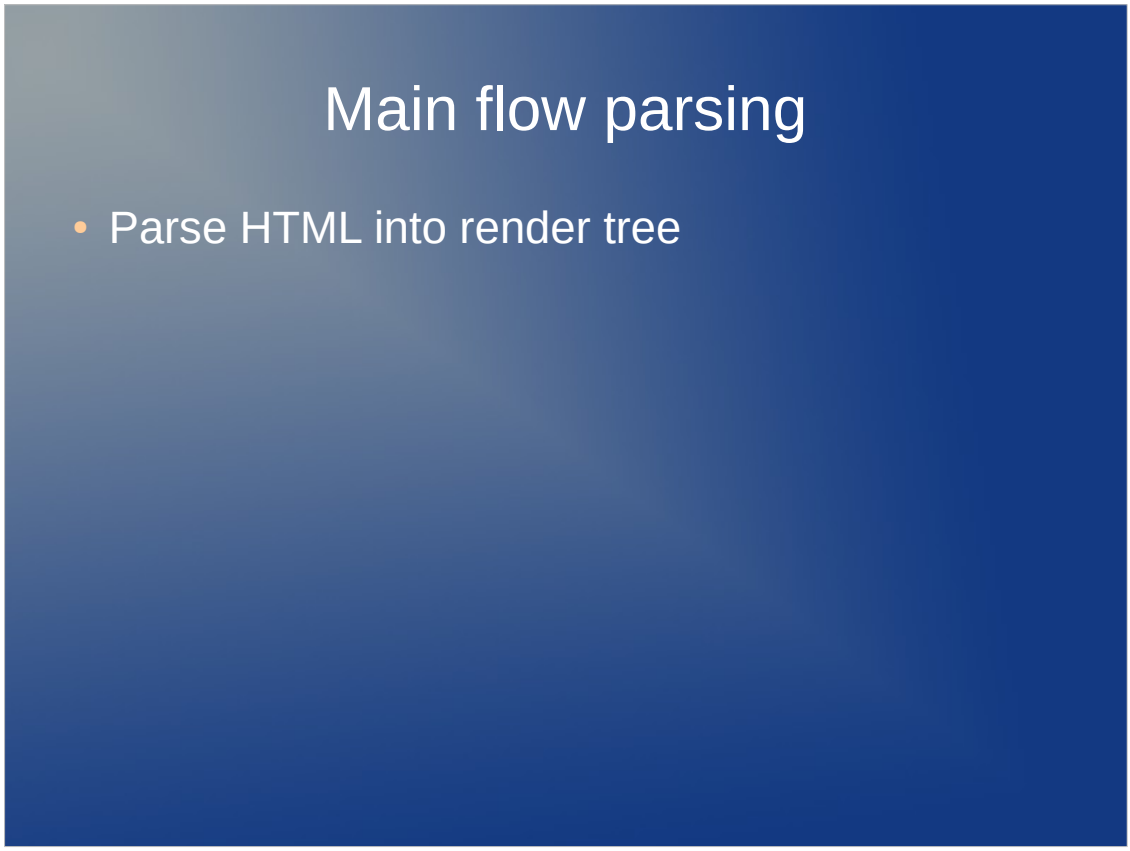
These are just like image renderers but may use other renderers for images etc.

PDF tools never have bugs of course



Just like the other renderers this one converts the source data (HTML) into pixels

Referred to as the “main flow”



Parse the html to generate a content tree. Not strictly a DOM tree because browsers heavily augment this tree

Main flow rendering

- Create render tree from content tree

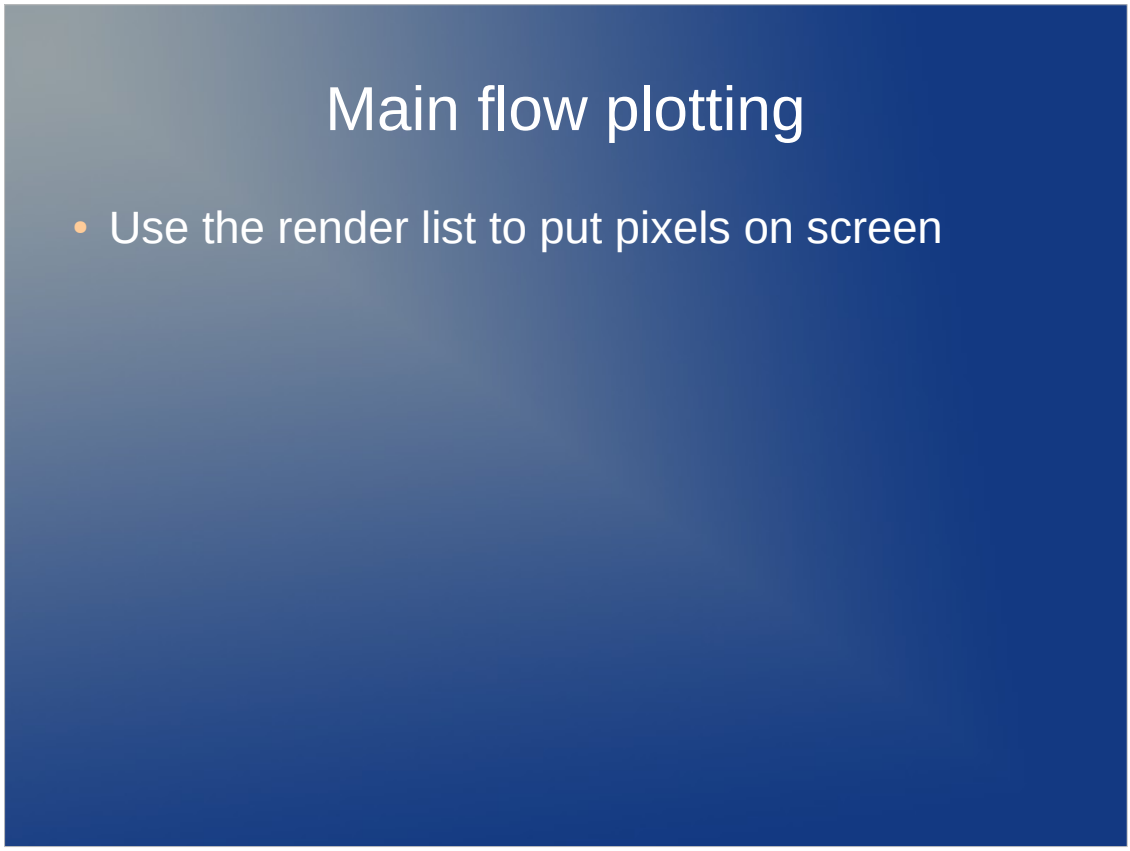
The content tree is used to build another tree, by parsing the style data, both in external CSS files and in style elements. Styling information together with visual instructions in the HTML will be used to create the render tree.

The render tree contains rectangles with visual attributes like color and dimensions. The rectangles are in the right order to be displayed on the screen.

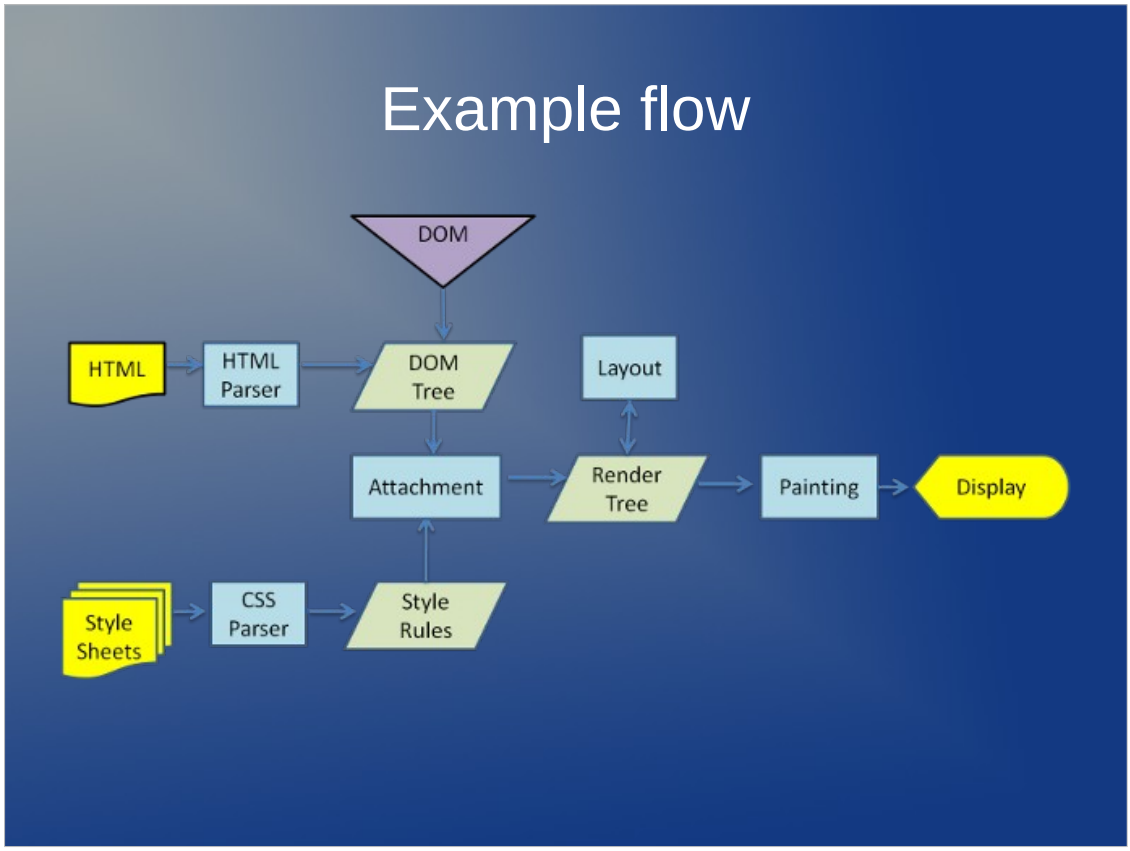
Man flow layout

- Convert the render tree to a render list

The render tree undergoes layout which gives it absolute positions within a viewport (browser window :-) to generate a render list of actual operations that need plotting.



The next stage is plotting (painting) the render list will be traversed and each node will be painted using the UI and you get cat pictures



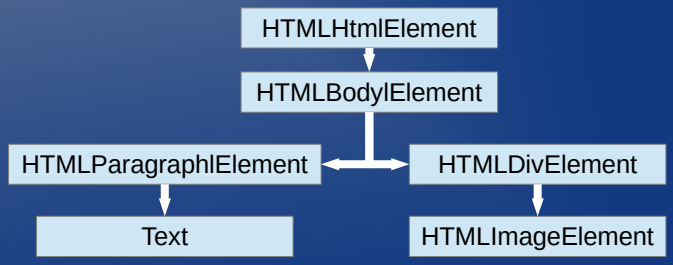
Flow diagram from webkit

All browsers internal HTML flow looks like this, the details differ considerably but the overall shape is the same.

HTML Parsing

- Not context free grammar
- Another living specification

```
<html>
  <body>
    <p>
      Hello World
    </p>
    <div> <img src="example.png"/></div>
  </body>
</html>
```



The parser takes the text from the protocol handler as it arrives over the network and turns it into a dom tree.

As each DOM node is inserted listeners are called to perform actions like causing css, images and scripts to be fetched.

The render tree constructor also sits and builds its tree.

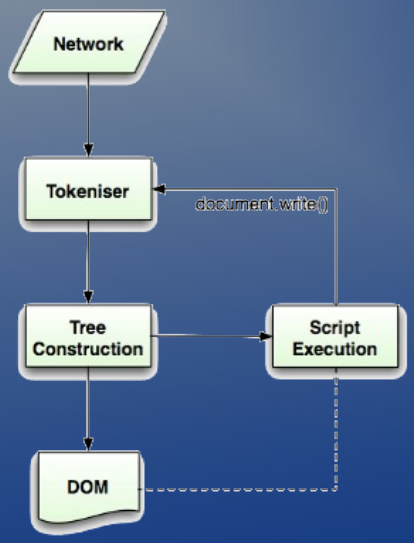
Why is this hard?

Because html has to forgive so much and still “work” the parser cannot be context free - actually it is much, much worse than this

Standards for tokenizing and tree construction but they are “loose” by definition because HTML accepts almost anything as input..

HTML Parsing

- HTML parsing is re-entrant!



When the parser inserts a script element into the dom tree that script has to be executed there and then.

The script can then call document.write()

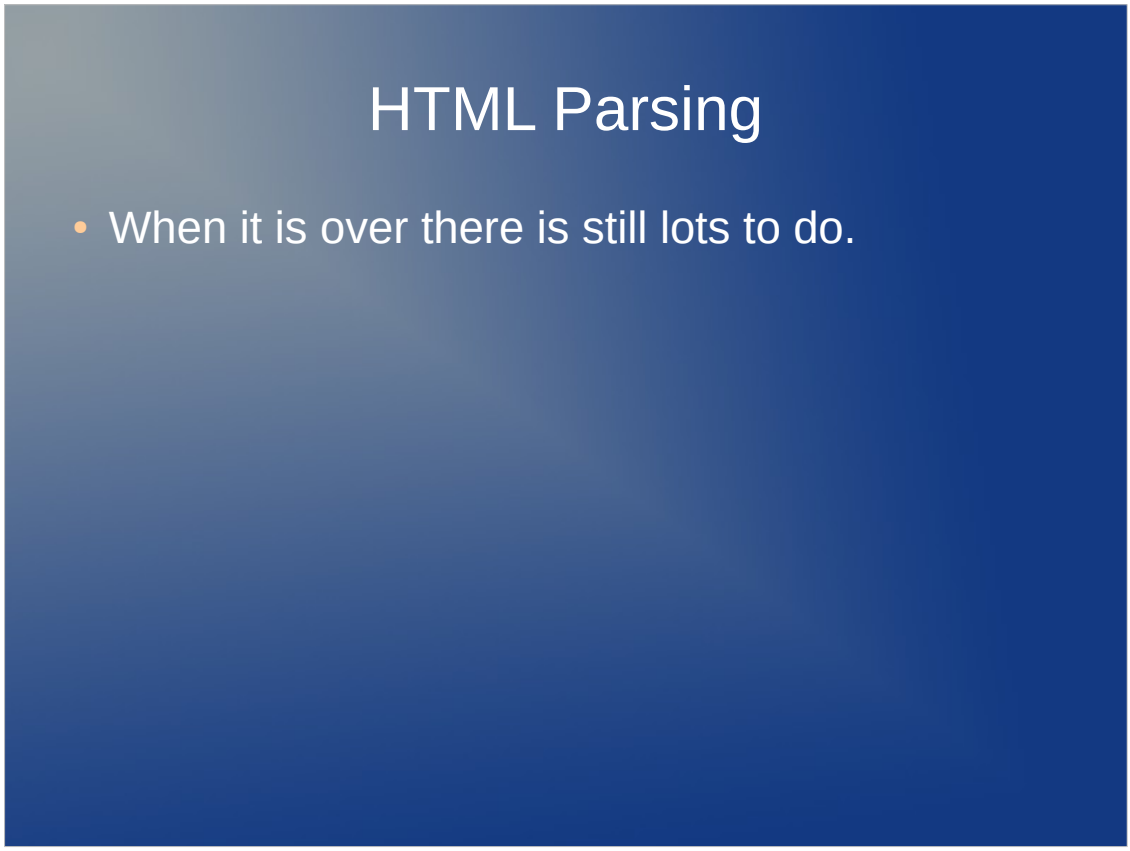
That inserts **text** directly into the parse stream. Oh and yes, you can then insert scripts in your scripts.

HTML Parsing

- HTML parsing is re-entrant!

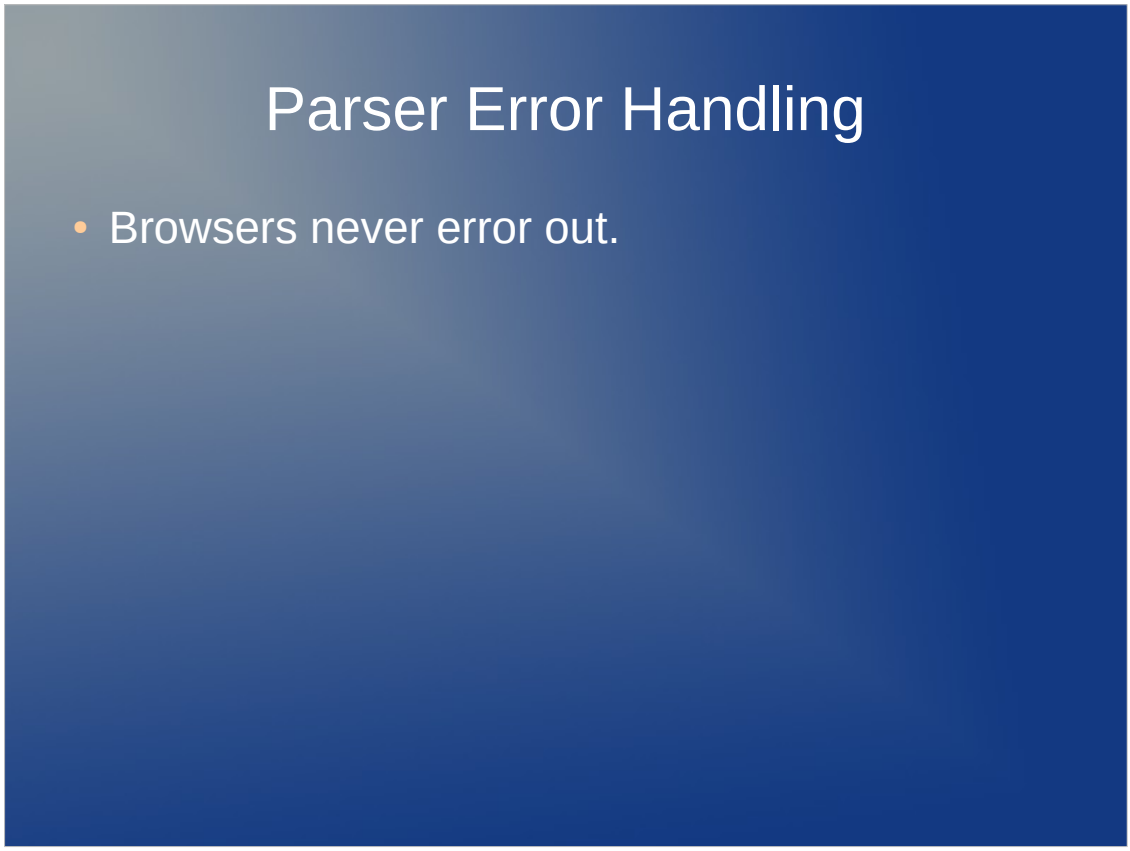
Turtles all the way down!

OK those are terrapins from a public domain image off wikipedia



When the parse completes there is still much to do.

Any defered javascript or css file loads must be completed, and events delivered for complete and the infamous ”onload”



Browsers never return an error to the user, they will try and deal with anything.

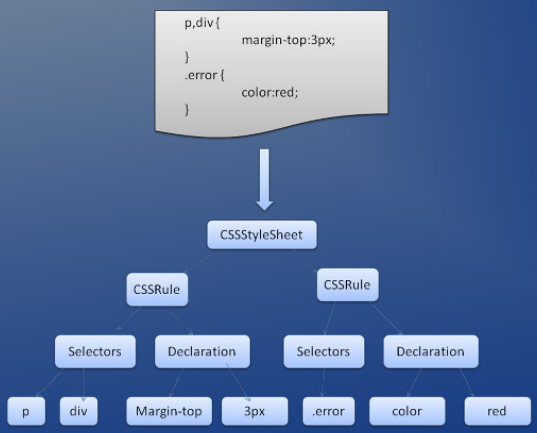
Error handling is quite consistent in browsers, but amazingly enough it hasn't been part of HTML specifications. Like bookmarking and back/forward buttons it's just something that developed in browsers over the years.

There are known invalid HTML constructs repeated on many sites, and the browsers try to fix them in a way conformant with other browsers.

The whatwg living specification does define some of these requirements but fundamentally it is all convention

CSS Parsing

- Is context free
- Well specified

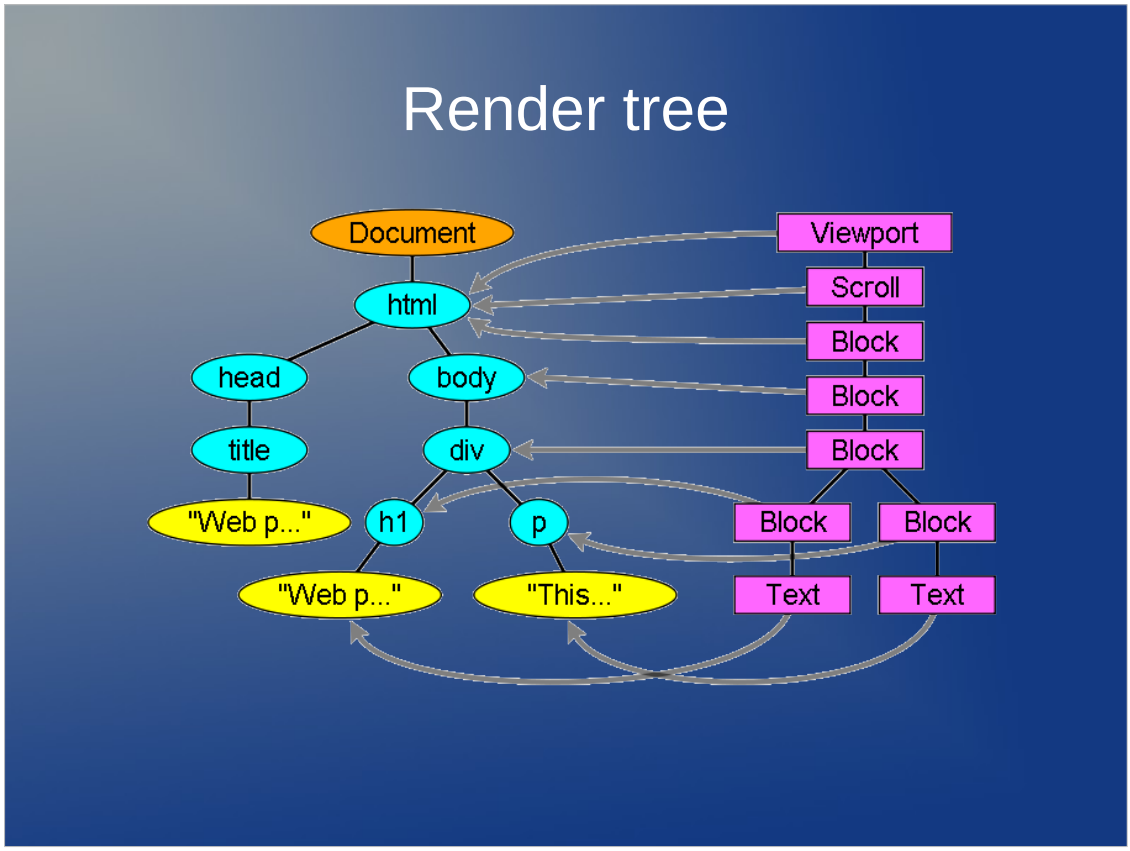


CSS is relatively well specified and can be parsed in a more traditional manner.

Grammar in BNF etc.

Still cannot rely on completely correct input but it is better

Ordering is hard. As noted scripts can be executed mid parse and if the css the script uses has not been loaded the script execution must wait until the stylesheet is available



Ok that is the DOM tree built. But while that was happening the render tree was also being constructed.

This tree is of visual elements in the order in which they will be displayed.

It is the visual representation of the document.

The purpose of this tree is to enable painting the contents in their correct order.

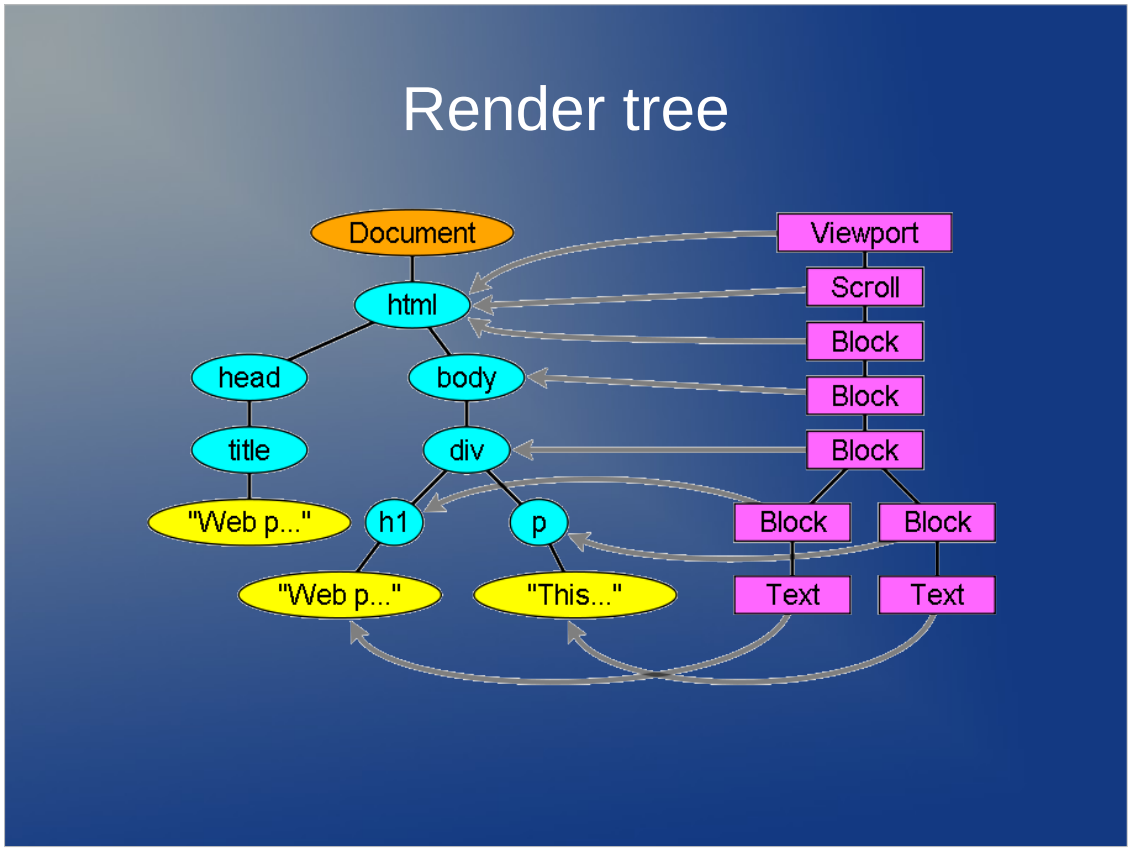
Firefox calls the elements in the render tree "frames".

WebKit uses the term renderer or render object

NetSurf refers to them as render boxes. Its all the same thing

.

A renderer object knows how to lay out and paint itself and its children.



The renderers correspond to DOM elements, but the relation is not one to one. Non-visual DOM elements will not be inserted in the render tree. An example is the "head" element.

Also elements whose display value was assigned to "none" will not appear in the tree (whereas elements with "hidden" visibility will appear in the tree).

Building the render tree requires calculating the visual properties of each render object. This is done by calculating the style properties of each element.

The style includes style sheets of various origins, inline style elements and visual properties in the HTML (like the "bgcolor" property).The later is translated to matching CSS style properties.

There is much, much more to how CSS and the render tree interoperate but I have a time limit.

Layout

- Converts the render tree to a render list
- Recursive process
- Co-ordinates relative to top left of viewport
- Makes concrete decisions on where the render objects are placed.

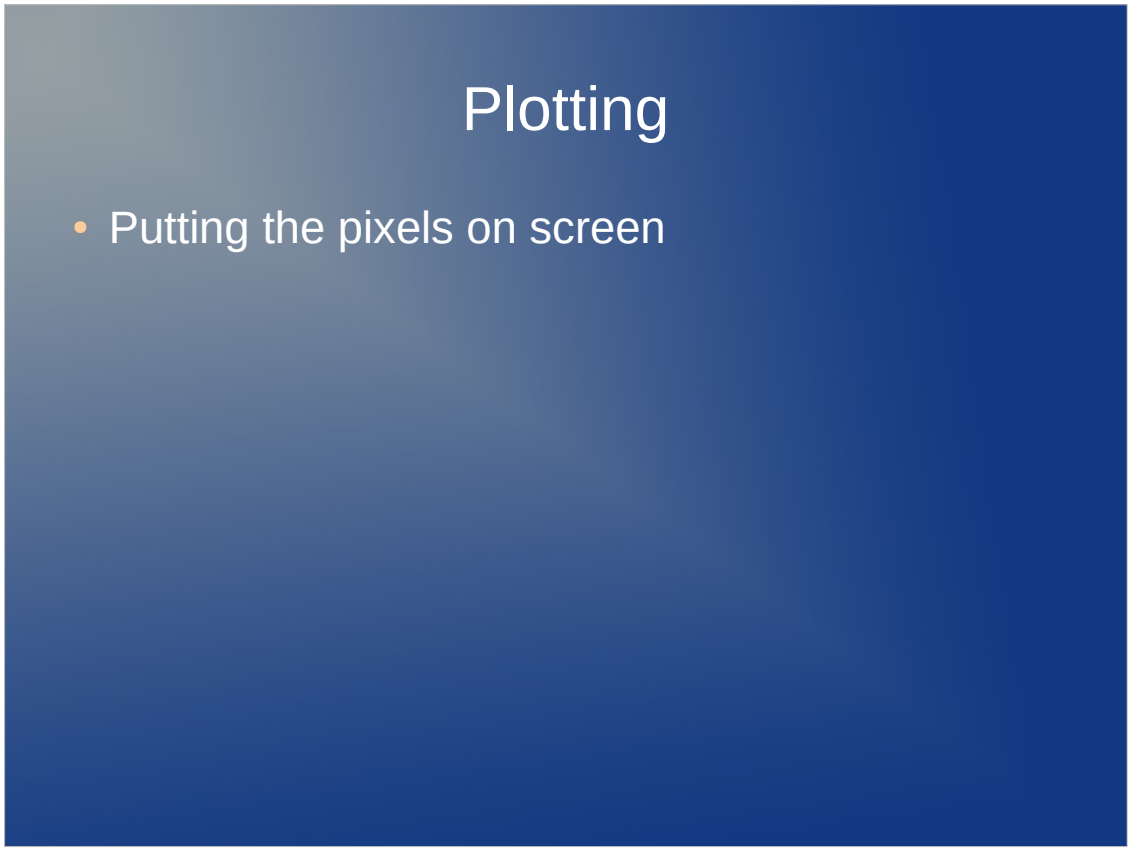
When the renderer is created and added to the tree, it does not have a position and size. Calculating these values is called layout or reflow.

HTML uses a flow based layout model, meaning that most of the time it is possible to compute the geometry in a single pass. Elements later ``in the flow'' typically do not affect the geometry of elements that are earlier ``in the flow'', so layout can proceed left-to-right, top-to-bottom through the document. There are exceptions: for example, HTML tables may require more than one pass

Layout is a recursive process. It begins at the root renderer, which corresponds to the <html> element of the HTML document. Layout continues recursively through some or all of the frame hierarchy, computing geometric information for each renderer that requires it.

The position of the root render object is 0,0 and its dimensions are the viewport–the visible part of the browser window.

All render objects have a "layout" method, each render object invokes the layout method of its children that need layout.

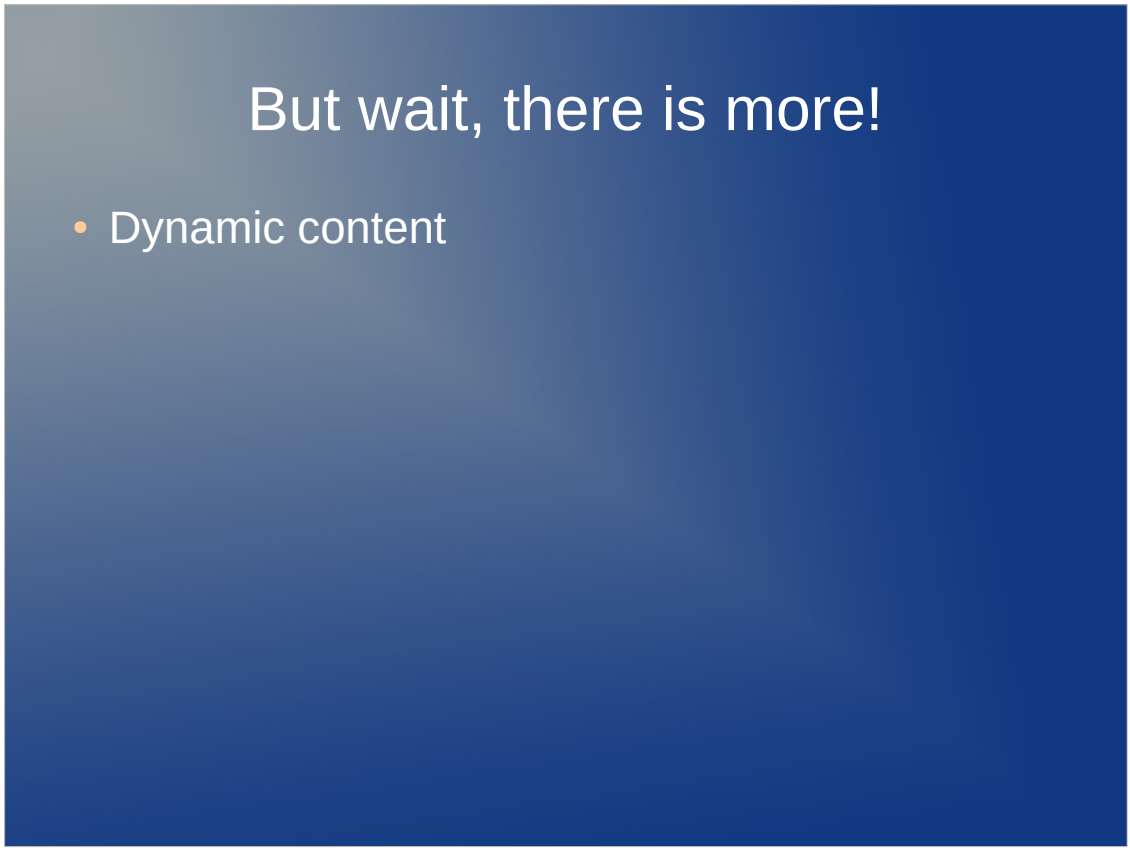


In the plotting or painting stage, the render tree is traversed and the renderer's "paint()" method is called to display content on the screen. Painting uses the UI infrastructure component.

CSS2 defines the order of the painting process. This is actually the order in which the elements are stacked in the stacking contexts. This order affects painting since the stacks are painted from back to front. The stacking order of a block renderer is:

background color
background image
border
children
Outline

Practical implementations use various acceleration techniques to make this faster. The state of the art has changed with modern GPUs meaning the paint stage has become less of a bottleneck.



That takes care of the main flow. But that is not the end of the story. Thanks to the wonders of dynamic content the DOM can be modified.

Modifying the DOM changes the render tree which causes (partial) layout that causes repaint.

Used to be small things like onclick which were user driven

But now we have canvas and webgl and timer driven redraw and the whole DOM can be re-written. The web is full of many horrors.

Any Questions?