Painless CSS

Build Beautiful, Intuitive Websites

BILL MEI

This is a free sample of the book
Painless CSS: Build Beautiful, Intuitive Websites
by Bill Mei

Learn more at https://www.painlesscss.com/

Chapter 3: Rendering

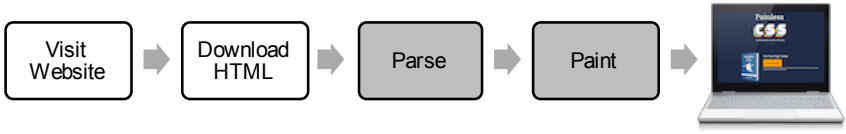
We learned previously that the only job of CSS is to make sure that things look nice on the screen. Thus, to really understand CSS, we should understand how things show up on the screen in the first place.

How a website loads

When you enter www.example.com into address bar of your browser, you're saying "Hey, www.example.com, give me your HTML file". The website responds by sending this file over the Internet, which your browser automatically downloads.

This downloaded HTML file is just words. After your browser downloads this file full of words, your browser somehow knows which individual pixels to turn on or off in your screen? How does that work? Here is a vastly oversimplified step-by-step process:

1. Browser downloads the HTML
2. Browser ***parses*** the HTML and CSS to figure out the structure of the page
3. Browser ***paints*** the page based on how it thinks the page should look



Parse means the browser takes our text file and, using the pre-defined rules of HTML language or CSS language, creates its own internal representation of the document that a machine can understand. It figures out the structure of how to put things on the page and gets an idea of what the page theoretically should look like.

Paint means the browser sends electrical signals to the screen to turn the pixels (the individual lights on your screen) on or off.

The combination of *parsing* and *painting* is also known as ***rendering***. If anything changes on the site, then the browser needs to re-render the page by re-parsing and re-painting the screen again to show you the updated content.

Let's break these down further to understand exactly what happens during the *parse* and the *paint* steps. We'll work backwards by talking about painting first.

Painting

Try this exercise: Open your text editor (installed in the *Introduction*), and type the following into a new file:

> *The quick brown fox jumps over a lazy dog*

Now save this file as `index.html` somewhere on your computer, and then double-click on this file to open it in your web browser.

What do you see?

Too easy right? But wait, there are some important questions I want to ask:

- How did the browser know to show your text at the top-left corner? Why not start at the bottom-left corner? Why isn't the text in the middle of the screen?
- How come the letters are coming in one after the other from left-to-right? Why not right-to-left? (Why don't we see `god yzal a revo spmuj xof nworb kciuq ehT`, instead?)
- Come to think of it, how do the letters know how to move out of each other's way? Why don't they just all try to show up at the same spot, resulting in a jumbled mess like this?

- What is a "letter" anyway? How does the browser know that an "o" should turn on the lights in a circle-shaped pattern, and an "l" should turn on the lights in a line-shaped pattern?

The short answer to these questions, is "that's just the way it is." It's just a convention that first programmers decided to follow, so now everyone follows it (like deciding whether to drive on the right side or left side of the road).

The longer answer is that things come one after the other from left-to-right because this is how the English language works, and the Internet was invented in the United States. Thus, the default direction is left-to-right. If you speak a right-to-left language such as Hebrew, you can manually change the direction so that the browser will render the text from right-to-left instead.

The even longer answer, is that this is the order that your GPU[11] uses to update your screen. Every time your computer needs to draw something new on the screen, it goes pixel by pixel starting from the top left, going left-to-right, and then top-to-bottom, all the way to the bottom right.

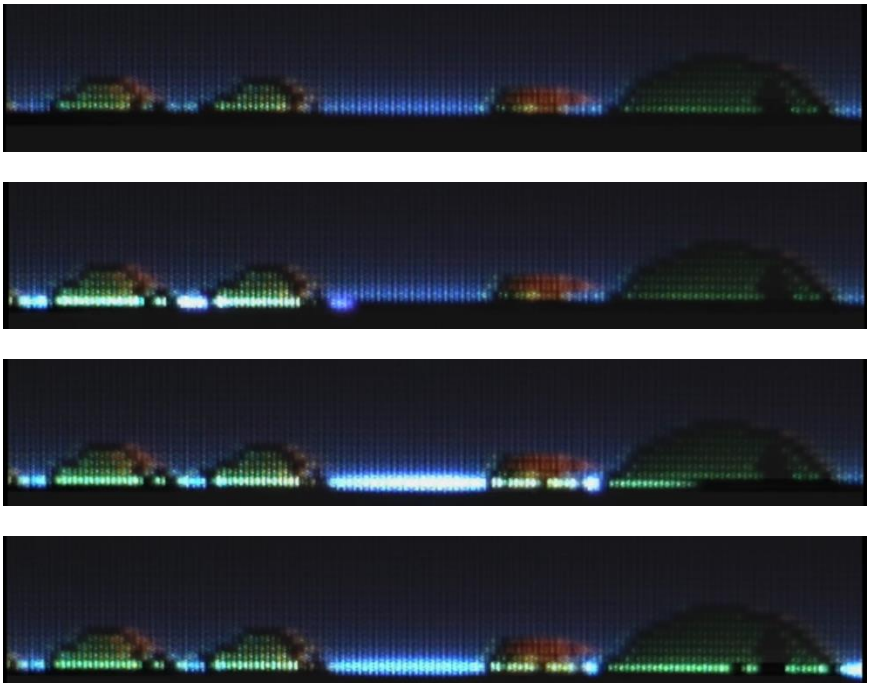
This painting happens so fast (the entire screen refreshes around 120 times per second), your eyes can't notice each individual pixel changing, which is why it appears to you that the whole screen is changing all at once.

However, you can use a slow-motion camera to see this happening. Watch this video to see a refresh happen, and also to learn what a "pixel" is in more detail: The Slow Mo Guys - How a TV Works in Slow Motion[12]

---

11 Graphics Processing Unit, the chip inside your computer which handles graphics

12 https://www.youtube.com/watch?v=3BJU2drrtCM&t=3m



The above images show what happens on old CRT screens. The first computers used CRT screens, and it was much easier to follow the hardware by writing software which paints in the same direction, instead of any other direction. On modern LCD and OLED screens, the hardware is so advanced that the computer can update the entire row all at once, instead of having to go pixel by pixel. Even though we no longer use CRT screens, the left-to-right, top-to-down order remains a standard convention, and we still follow it today.

Now that we understand what is happening during the *paint* operation, we can talk about the step that happens right before this: *parsing*.

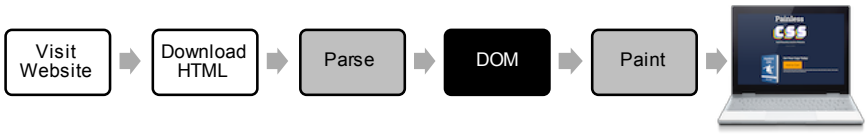
Parsing

After your browser downloads an HTML from the website you want to visit, it needs to *parse* this file to understand what exactly should be displayed.

The goal of parsing is for the browser to create its own internal representation of the content that a machine can understand. We call this internal representation the ***Document Object Model***, or ***DOM***.

Once the DOM is created, the browser can then hand it off to the hardware to perform the painting.

To summarize:



The HTML file that your browser has downloaded is just a text file. If you've encountered a file on your computer ending in `.txt`, such as one created with Notepad (on Windows) or TextEdit (on Mac), it's just like that. HTML files can end in `.html` or `.htm` too, but this is just a convention.[13] The ending of a file after the `.` is called a ***file extension***.

Here is the most basic possible HTML file:

`Source Code: chapter03/basic.html`

```
<html></html>
```

Recall that HTML is a markup language which defines the structure of a page using *tags*. Every HTML file starts with an `<html>` opening tag and ends with an `</html>` closing tag, and the whole website goes in between these tags. This way

---

[13] In fact, you could remove the `.html` at the end of the file and replace it with `.banana` and your browser will open the file just fine. Your computer has a list of all file extensions that it recognizes, such as `.mp3`, `.png`, `.txt`, `.html`, and so on. If your browser encounters a file extension that isn't in this list such as `.banana`, it tries its best to figure out what this file is by reading its contents.

However, if you mislead the browser by renaming your HTML file to use a file extension that already exists (such as `.mp3`), then your browser will try to open it as an audio file, but that won't work. We try to always follow conventions because this makes life easier for the browser, and also because if you see a file that ends with `.html` you'll immediately know "oh, that's an HTML file".

you say to the browser "everything in this file is HTML", as opposed to only plain text with no markup.

You can nest tags inside each other. For example, if you wanted to make some text inside a paragraph both bold *and* italics, you could write something like this:

Source Code: chapter03/quickbrownfox.html

```
<html>
  <p>
    The quick brown fox jumps over a
    <strong>
      <em>
        lazy
      </em>
    </strong>
    dog.
  </p>
</html>
```

This is the result:

> The quick brown fox jumps over a ***lazy*** dog.

Something extremely interesting happens when you start nesting tags inside other tags. The HTML tags get together and turn into a family—we could refer to the outer tags as ***parents*** and the inner tags as ***children***. You could even say that two child tags under the same parent are ***siblings***.

But like any family, when these tags live together under the same house, they don't always live together harmoniously, and we can get a lot of exciting drama when they get into conflicts with each other!

This family of tags is called the DOM, and the point of parsing is to read the HTML file to understand the family and build the DOM.