Class Notes: 26 September

CSS for positioning, images, concluding remarks

Positioning elements in 2-dimensional space

Before the end of the last century, many web-designers still relied upon HTML tables to control the placement of elements in 2-dimensional space. Sometimes this worked. But it was limiting. Some advantages to using tables:

- Well-known. Easy to write programs that auto generated tables, nested tables, etc.
- Somewhat intuitive, but not always—especially when they were rendered in a browser whose viewport was not of the expected dimensions.

A big problem with tables is they were elements that shared the same space as other HTML elements: limited to the container (document) in which they resided. Also, tables did not react well to being resized. As long as our expectations were low and the data somewhat predictable, this worked. But, as data requirements grew and people wanted more natural formatting, something needed to be done.

CSS added a set of properties and values to deal with “positioning,” hence CSS-P as it was called. Again: separate visual properties from semantic relations, as we said:

> As always: Think of HTML as the “composition,” the notes as the composer wrote them. Think of CSS as a particular performance of that composition.

What is a Position?

Ask: What determines the position of an entity? Position is always relative to some other object or objects.

HTML already provides parent/child (container/contained) relations, and this is a kind of position.

Ignoring left-right relations for the moment, we envision that children appear roughly beneath parents. Thus, HTML elements have a natural flow which follows gravity. But this is not enough.

Recall: all HTML elements reside within a single object, called the “document.” This document resides in another container, however, called a “browser.”

The elements and their relations contained with a document form the Document Object Model (DOM); the components of a Browser, which include the frame where HTML is rendered among other things, comprises another “model,” the Browser Object Model (BOM).

> (Models are mathematical/logical objects with a consistent semantics. Weather models, for example, help forecast the weather. We will talk a little about models from this week forward—especially as it relates to programming.)

CSS: the intermediary

CSS stands between these models, and, as such, can handle positioning by communicating with both the DOM and BOM (actually: CSS mediates communications between these containers).

CSS looks like a vertical arrangement of “rules:

```
selector { property: value; … }
```

Recall that CSS rules associate “selectors” with transformations, which appear as ordered pairs of properties and values. These **selectors** pick out HTML elements, these **property:value pairs** may delegate to the Browser to change a color or fix an element at a particular position.

Sometimes the position of an element is relative to its siblings … but other times it is not. Think of the column of Menu Items that appears along the left-hand side of the Elms class page.

Keep these ideas in mind as you review the slides for this week, and chapters 15 and 17 in your textbook. You should now be able to conceptualize for yourself the distinctions between :absolute, :fixed, and :relative.

Specific observations, suggestions, etc.

Remember that each HTML element is its own “box.” Boxes are either block-level or inline (we discussed this in greater detail last week).

You might begin by determining how much space a box requires (or will take-up) by computing its perimeter, or usually just its width.

Use borders, margins, padding, etc. to separate “boxes”.

Remember the parent-child relations, and, when appropriate group “boxes” by the consistent use of `<div>` (and sometimes `<span>`) elements. Naturally, each such division or span will most likely be associated with a class or with an id.

Reviewing our positioning schemes:

- Normal positioning—each “block” starts a new “line” (vertical flow).
- Relative positioning—moves element left, right, up, or down, of where it would have been without impacting the position of neighboring elements. In other words, siblings maintain their normal orientation.
- Absolute positioning—positions the element in relation to its containing element. The element is “taken out of the flow.” Thus, it does not impact the positioning of other elements. The “effect” is that these elements move as the user scrolls the page.
- Fixed positioning—a version of absolute positioning where the element is positioned in relation to the Browser’s display window. (This is different from absolute because the element in question is independent of its local container.) Fixed elements remain “fixed” and do no affect the positions of their neighbors. Think of text or any element that remains fixed as the user scrolls the text in the browser.

- Floating—removes the element from the “normal” flow, usually positioning it to the left or right of its “containing” box. The floated element becomes a block around which others flow.
    - Here’s where we might use the z-index property to determine which box appears on top.

Of these, the mechanics of “floating” is sometimes challenging, but often very useful in creating fluid presentations. Consider the next example (that we probably used in class).

Example of using a float to format a newsletter

Suppose we wanted to format a web-page as a two-column newsletter. Skipping some of the finer details, such as fonts, leading, justification and imaging sizing schemes, we certainly need a two column layout. Rather than use a `<table>` we might do something like the following:

```
<h1> Headline </h1>
<div class=”col_1_of_2”>
   <h2>Small Headline</h2>
   <p> text </p>
</div>

<div class=”col_2_of_2”>
   <h2>Small Headline</h2>
   <p> text </p>
</div>
```

And the CSS might look like:

```
.col_1_of_2 { float: left; width: 450px; margin: 10px; }
.col_2_of_2 { float: ?; width: 450px; margin: 10px; }
```

I intentionally left out the “float orientation” for the second rule. What do you think it might be? As a good habit of mind, compute the required width for these elements.

Useful Tutorials, etc.

As a reminder: use the CSS Intermediate and Advanced Tutorials which are available on the “dog site,” found under Helpful Resource (the first module on the Elms class page) to assist you in completing Project 2. At the end of each tutorial, you will find a dozen or so “examples” that will most likely provide you with near-fit code. Of course, we expect you to modify that code to your specification (do not plagiarize).

Images.

CSS3 prefers that images, i.e., <img> elements, be wrapped in <figure> elements as it provides a better model for controlling text runaround, etc. But, notwithstanding this, we need to discuss some preliminary issues that are related to the sizes, types, and placement options for your images.

Consider standardizing the size of images

When designing a site that uses images consider performance issues as well as display. Most of the time, you will use a "standard" set of image sizes, for example: small, medium, and large. Providing this information in the CSS stylesheet as opposed to specifying it on an image-by-image basis will

1) Placate the Validator;
2) Improve the overall speed with which the site loads, and
3) Likely result in a better design (one that's easier to manage and evolve over time).

Choosing the appropriate image type

Choose JPEG image types for images that are rich in color and/or greyscale, such as high-quality panchromatic (black and white) photographs. Note: JPEG is a lossy format, meaning that it reduces image size by a user-selectable percentage. This means that you will likely choose a working percentage of about 70 to 80% for most applications.

A variety of post-processing tools are available to you, and most of these allow you fine control over the JPEG image that is produced/exported by the tool.

In addition to file type, again, this is a place for you to set the dimensions of your image.

Choose PNG for low-resolution images that need to load quickly and will not be required to support animation. Note, PNG offers some advantages over GIF in that is lossless and it provides for transparency and layering …

But, if your images have "sharply defined edges" and/or will be animated, you would use GIF format.

Rather than attempt to summarize and discuss these in great detail, you will find a large number of web-sites that discuss these image types in great detail.

Some practical considerations ...

Prefer the "`float: direction`" pattern in CSS to specifying align: direction in `<img>` tags.

Consider adding margins to images to create some "buffer" between the end of the image and the beginning of text.

Again: whenever possible consolidate images to several "sizes," such as small, medium and large.

Likely pattern:

```
img.align-left { float: left; margin-right: 10px; }
img.align-right { float: right; margin-left: 10px; }
img.medium { width: 250px; height; 250px; }
```

Putting these "together" in a realistic application in HTML gives us something that we have not yet talked about:

```
<p><img src="..." alt=" ..."
   class="align-left medium">
… some test here …
</p>

<p><img src="..." alt="... "
   class="align-right medium">
… some text …
</p>
```

Do you see what it is? Note how we used "multiple" class names on the image tags above!

You might ask: does the order in which these names appear matter? Well, it does only when the rules associated with these classes reference the same properties! Then, it's the last in the list that takes preference … tricky, no? Think about this … it's usually best to avoid those kinds of dependencies. More on this later.

Background Images.

This is all out of the tutorials, etc. so we just summarize:

We can place an image behind any HTML element. Suppose we choose the “body” –

```
body { background-image url(“ . . . “) … }
```

The remaining complexity comes from choosing among a variety of “options.” We summarize just the most common (in their order of appearance). To do this briefly, I will introduce some new “syntax,” the listing of values that are separated by horizontal bars should be read to mean that any one of these symbols is allowed here and the bar (i.e., the “|”) is read “or”

```
background-repeat: repeat-x | repeat-y | repeat | no-repeat
```

This means that the background repeat for this element can be either `repeat-x`, `repeat-y`. `repeat`, or `no-repeat`.

```
background-attachment: fixed | scroll
```

The meaning of these values is self-evident.

The next property allows for nine different predefined settings as well as a way of specifying percentage offset from the top-level corner.

```
background-position:
```

Beginning with the nine: take the cross-product of { left, center, right} and { top, center, bottom } and you get nine items that look like:[1]

```
left top | left center | left bottom
center top | center center | center bottom
right top | right center | right bottom
```

(Again, read the “|” as the word “or”)

If this is not enough, you can resort to pairs of percentages, such as 0% 0% (which means top left) and 50% 50% which we assume centers both horizontally and vertically.

1 We should talk about cross-products here because these come up often in programming.

Finally, when specifying background images, the order in which the associated properties appear is fixed:

```
background-color, background-image, background-repeat, background-attachment,
background-position
```

Obviously, you should consult the CSS references when designing your own rules—no one should remember all of this.

Additional considerations.

A host of additional topics come to mind here. For instance: when images overlap text (or vice versa), the "relative difference in contrast" is critical. Both the text and the image cannot have the same relative contrast levels, such as high-high, low-low … But, CSS gives you no way of talking about image or text contrast. Again: you need to rely upon third-party tools to edit the images (most likely the images) to the desired contrast.

Bear in mind that we are dipping our toes in the shallow end of a very deep pool. We have done this to give you the ability to create and manage content in a Browser. But, this class is certainly not intended to prepare you for a career in web-design, rather it is an introduction to the broad topic of "computer programming." A reasonable question at this point is "how is any of this related to programming?"

Neither HTML nor CSS are "programming languages." But, they do introduce some essential ideas:

- Workflow: text is created in a "formal language," HTML, CSS, etc.
- That text is translated and interpreted by another program. Assuming that all went well,
- Something changes; usually this means that something appears different in the browser.

Programming is an activity that appears to have a similar workflow: You will create text that must conform to some "formal grammar," i.e., is determined by the rules of the language. Unlike what you've done to this point, however, the "meaning" of this programming text is much richer but not always so obvious.

Programs are "algorithms," a term we shall define in the following class. Algorithms give us a way of interacting with a world of data, which in our case is a world of objects that reside in an HTML document—remember the DOM? So, programs will allow us to transform or change these objects in response to changes that occur on objects that reside in the DOM … . Think a little about this. Now you should see why we've postponed talking about "forms" and some of the other topics in your textbook pending some understanding of algorithms.