crazy layouts

I’ve been playing with the idea of creating crazy, magazine style, layouts since I Attended Jen Simmons and Rachel Andrew workshops at An Event Apart last year.

I came up with this crazy idea, build magazine layouts like those on the printed edition of Kinfolk Magazine and see how far we can push layouts while still remaining usable at small to medium scale sites as bespoke, art directed, designs.

This post is a refresher on the SCSS, CSS and Javascript I will use along with ideas of how these technologies may be used in designing print-like publications. Future posts in this series (published occasionally) will cover specific designs, from Kinfolk and elsewhere.

Font loading and fontface observer

Loading local web fonts has always been a pain. Paul Irish posts in 2009 and 2010 was one of the first people to articulate how to add web fonts to your page.

FontSquirrel’s Web Font Generator is still one of the best ways to get your fonts ready for web deployment, particularly when you have only one format of the font face and not all of the ones you need.

After a while I decided to bring it in house and adopted the following SCSS as part of my standard SCSS mixin library.

```
@mixin font-declaration($font-family,  $font-file-name, $weight:normal, $s
  @font-face {
    font-family: '#{$font-family}';
    src: '#{$font-family}'urlme}.eot');
    src: url('#{$font-file-name}.eot?#iefix')
          format('embedded-opentype'),
    url('#{$font-file'#{$font-file-name}.eot?#iefix'url  url('#{$font-file
 '#{$font-file-name}.woff'urltf') format('truetype'),
    url('#{$font-file-n') format('urlnt-family}') format('svg');
    font-weight: $weigh') format('url('#{$font-file-name}.svg##{$font-fam
    font-weight: $weight;
    font-style: $style;
  }
}
```

The `font-declaration` mixin takes the following as parameters:

- `$font-family`: The name we'll give to the font
- `$font-file-name`: The path of the font relative to where the stylesheet lives
- `$weight`: The weight of the font, a value of 100 to 700 or keywords (bold, bolder). The default is normal
- `$style`: is the font italics? Default is normal

With this mixin available we can then define our fonts. Each call to `@include font-declaration` will install one font face for the project. It's important to understand why we install 4 different font faces for the Noto Sans font.

If a browser cannot find a font face for bold, italics or bold-italics it will synthesize the font it needs with unacceptable that will vary between browsers. Instead we make sure that browsers will find the needed fonts to present font variations in style and weight. Later we'll make sure this works by assigning the fonts to given styles.

```
/* Monospaced font for code samples */
@include font-declaration('notomono_regular',
  '../fonts/notomono-regul'notomono_regular'l, normal);
/* Regular font */
@include font-declaration('notosans_regular',
  '../fonts/notosans-regular-webfont', normal, normal);
'notosans_regular'/* Bold font */
@include'notosans_bold',
 '../fonts/notosans-bold-webfont', 700 , normal);
/* Italic Font */
'notosans_bold'/* Italic Font */
@includecs',
 '../fonts/notosans-italic-webfont', normal , italic);
/* bold-italic font */
@include f'../fonts/notosans-italic-webfont'/* bold-italic font */
```

```
@include font-declaration('notosans_bolditalic',
 '../fonts/notosans-bolditalic-webfont', 700 , italic);
```

If the fonts loaded we want to make sure that the browsers pick up which fonts we want to use. Otherwise it doesn't matter because our default when fonts are not loaded is Verdana, making Verdana bold shouldn't be too much of a problem.

I've also created three versions of the body selector. They first one, plain body will match the body element before the loader script runs. The other two will be explained when we look at the loader script.

```
.fonts-loaded strong,
.fonts-loaded b {
  font-family: "notosans_bold";
}

"notosans_bold".fonts-loaded em,
.fonts-loaded i : "notosans_italics";
}

.fonts-loaded s"notosans_italics".fonts-loaded strong em,
.fonts-loaded strong i,
.fonts-loaded b em,
.fonts-loaded b i {
  font-family: notosans_bolditalic;
}

body {
  font-family: Verdana, sans-serif;
  font-size: 16px;
  line-height: 1.375;
}

/*
  This will match if the fonts failed to load.
  It is identical to the default but doesn't have to be
*/
.fonts-failed body {
  font-family: Verdana, sans-serif;
  font-size: 16px;
  line-height: 1.375;
}

/* This will match when fonts load successfully */
.fonts-loaded body {
  font-family: notosans_regular, verdana, sans-serif;
  font-size: 16px;
  line-height: 1.375;
}
```

Now that we’re done with the SCSS/CSS parts we can start working on the loader. I use [Font Face Observer]() to make font loading easier with a promise-based interface that is fairly easy to use as we’ll see later in this section.

Load `fontfaceobserver.js` in a script tag in the page you want to use the fonts in. Then load the script below

```
var mono = new FontFaceObserver('notomono_regular');
var sans = new FontFaceObserver('notosans_regular');
var italic = new FontFaceObserver('notosans_italics');
var bold = new FontFaceObserver('notosans_bold');
var bolditalic = new FontFaceObserver('notosans_bolditalic');

var html = document.documentElement;

html.classList.add('fonts-loading');

Promise.all([
  mono.load(), sans.load(), bold.load(), italic.load(), bolditalic.load()
]).then(() => {
  html.classList.remove('fonts-loading');
  html.classList.add('fonts-loaded');
  console.log('All fonts have loaded.');
}).catch(() =>{
  html.classList.remove('fonts-loading');
  html.classList.add('fonts-failed');
  console.log('One or more fonts failed to load')
});
```

The script has three sections in the first section we assign an instance of Font Face Observer (indicated by the new `FontFaceObserver(font-name)` method) to each of the fonts we want to use. The name used here must match the name of the font we loaded in CSS. I've also created a shortcut for `document.documentElement` that I will use to assign and remove classes to the document.

Before calling the promise I assign the class `fonts-loading` to the `html` element.

The second part is a `Promise.all` call to the load method for each font I'm using.

The third part is the continuation of the promise call. If the call succeeds (meaning all fonts loaded) the `then` method gets called and does three things: it removes the `font-loading` class from the HTML element, add the `font-loaded` class to the HTML element and logs the result to the console.

If one or more of the fonts fail to load the `catch` method will trigger and do the following: remove the `font-loading` class from the HTML tag, add the `fonts-failed` class to the HTML element and log the result to console.

I know this seems like a lot of work but think about the benefits we get:

- We can be certain that the fonts loaded (or didn't load)
- If we need to we can dictate how long should take to load (I didn't do it in this example)
- We can style the content both before and after the fonts have loaded using something like Meownica's font style matcher

Now I've loaded the fonts… what's next?

Grid and item placement

I'm super excited for CSS Grids. If you've been around the web development block for a few years you know that there are a ton of different float based grids and, most if not all of them, require complicated calculations and make even smaller changes a dicey proposition.

CSS Grids are built into the CSS parser so the browser itself is doing all the work and doing it better and faster than float-based implementations. They are also easier to work with media queries.

Rather than typing all the components of a grid manually I've gone back to SCSS and created a mixin to generate a grid. By default it was 12 columns of equal width and a gap between columns of 10px.

```scss
@mixin generate-grid($columns: 15, $gap: 10px) {
  display: grid;
  grid-gap: #{$gap};
  grid-template-columns: repeat(#{$columns}, 1fr);
  grid-template-rows: 1fr;
}
```

The grid specification introduces another length unit to CSS: The flexible length unit.

> A flexible length or is a dimension with the fr unit, which represents a fraction of the free space in the grid container.
>
> The distribution of free space occurs after all non-flexible track sizing functions have reached their maximum. The total size of such rows or columns is subtracted from the available space, yielding the free space, which is then divided among the flex-sized rows and columns in proportion to their flex factor.
>
> — CSS Grid level 1: 7.2.3. Flexible Lengths: the fr unit

If you leave the default values, the SCSS above will produce the following CSS file

```
.grid-demo {
  display: grid;
  grid-gap: 10px;
  grid-template-columns: repeat(15, 1fr);
  grid-template-rows: 1fr;
}
```

There are times when we want to create fluid grids, something beyond what our prior definition gives us. There is another way to define a grid where we tell the CSS parser to create columns of no less than minimal size and no larger than a maximum size (indicated in the mixing by `$min-size` and `$max-size`).

```
@mixin generate-fluid-grid($min-size: 100px, $max-size: 1fr, $gap: 10px)
/* autoprefixer: off*/
  display: grid;
  grid-gap: #{$gap};
  grid-template-columns: repeat(auto-fill, minmax(#{$min-size}, #{$max-si
  grid-template-rows: 1fr;
}
```

Note that the grid generation mixins do not deal with content alignment as specified in the [CSS Box Alignment]() specification. We can add those attributes when we build out the main CSS file.

Now that we've built the grid either as a static or fluid container we have to figure out how to place content in the grid. The `place-item` mixin does just that. it takes 4 values to represent column start and end (`$col_start`, `$col_end`)and row start and end (`$row_start`, `$row_end`).

```
@mixin place-item($col_start: 1, $col_end: 1, $row_start: 1, $row_end: 1)
  grid-column: #{$col_start} / #{$col_end};
  grid-row: #{$row_start} / #{$row_end};
}
```

The biggest advantage of using a mixin like this is that SASS doesn't evaluate the type of the variable so we could use it in the following hypothetical examples, taking from the grid specification

```
.three {
  grid-row: 2 / span 5;
  /* Starts in the 2nd row,
     spans 5 rows down (ending in the 7th row). */
}

.four {
  grid-row: span 5 / 7;
  /* Ends in the 7th row,
     spans 5 rows up (starting in the 2nd row). */
}
```

There is much more to CSS grid than I've covered here. Check the specification for more of what you can do with grids and other ways to work with them.

Multi column text

We've been able to do column-based layouts on the web for a while with different level of browser support. It wasn't until magazine style layouts became a real possibility for the web that I thought it good to revisit columns.

As with grid I've created to SASS mixins for columns. The first one gives a static number of columns (two by default) with a 1em gap between the columns.

```
@mixin column-attribs ($cols: 2, $gap: 1em, $fill: balance, $span: none){
  // How many columns?
  column-count: $cols;
  // Space between columns
  column-gap: $gap;
  // How do we fill the content of our columns, default is to balance */
  column-fill: $fill;
  // Column span, default is not to span columns */
  column-span: $span;
}
```

the second version, `fixed-columns-attribs`, uses fixed width columns to tell the browser: fit as many columns of this width with 1em gap in the space you have available.

```
@mixin fixed-column-attribs ($col-width: 15em, $gap: 1em, $fill: balance,
  // How many columns?
  column-width: $col-width;
  // Space between columns
  column-gap: $gap;
  // How do we fill the content of our columns, default is to balance */
  column-fill: $fill;
  // Column span, defaul is not to span columns */
  column-span: $span;
}
```

Both of these versions have the `column-span` property built into the mixin. This property is not widely supported (only Firefox and Mozilla browsers as far as I know). It's still a good idea to use it and future proof the code.

Hyphenation

Hyphenation is one of those strange things that makes text works better but where we need to be very careful when using it as browsers may not implement hyphenation the same way and the results may be different.

With that out of the way we can get started working with hyphenation on the web using CSS. We first define that the whole document (indicated by the body element) will be hyphenated. Then, for all the elements that we **don't** want hyphenated we reverse the rule, we say that hyphenation will be manual (as we did below) or that there will be no hyphenation (using the none value for hyphens).

I like to make sure that headings are not hyphenated. It's mostly a cosmetic choice but I don't want the content of my headings broken by a hyphen I'll add one more bit rule to make sure that there are no hyphens on any heading (h1 through h6).

```
body {
  hyphens: auto;
}

code, var, kbd, samp, tt. dir, abbr, acronym, blockquote,
q, textarea, input, option {
  hyphens: manual;
}

h1, h2, h3, h4, h5, h6 {
  hyphens: none;
}
```

Hyphenation is not supported at all in Chrome. For Chrome and older browsers, look at JavaScript solutions like Hyphenator.js and Bram Stein's Hypher.

Justificating text

Content derived in part from Advanced web typography: Justification & hyphenation by Elliot Jay Stocks.

There are two major justification algorithms we come into contact with regularly: the Greedy algorithm that analyses only one line, and the far more advanced Knuth / Plass algorithm that looks at all lines in a paragraph, and which is used in advanced typesetting applications such as InDesign. Now, guess which one we have in browsers, ladies and gents? That's right: the crap one.

So we get justification but we get the crappy in most browsers but IE. The CSS3 spec introduced text-justify (in the CSS Text Module Level 3 which has been at the CR stage since 2013 because of no interoperable implementations), which allows us to control some aspects of justification. Sadly the spec doesn't specify which algorithm to use, which means that it's in the hands of the browser makers. Browser makers will not implement the Knuth / Plass algorithm because it involved taking a performance hit, so unsurprisingly the browsers have stuck with ol' faithful: Greedy.

Internet Explorer is the only browser that uses the Knuth / Plass algorithm, or an approximation of it, enabled through the text-justify: newspaper CSS property.

Some Javascript-based alternative solutions exist that attempt to give a better experience with justification on the web. One is Typeset, which implements the

Knuth / Plass algorithm. It's more of an experiment, the author has stated that it's not ready for production and likely will never be) but it's worth checking out as a work-in-progress and proof-of-concept. It's idea for the type of projects I want to work in.

Figures, captions and caption placement

Rather than using plain images I've taken to use the `figure` element and the `figcapion` child to provide captions.

The example below provides a basic mixin to work with figures. It takes three parameters:

- `$border-color` indicates the color for the figure border
- `$width` gives the width of the figure
- `$align` indicates the horizontal placement of the image. It takes one of `left`, `center` and `right`

```
// Builds figures with different alignments
@mixin figure($border-color, $width: 100%, $align: center) {
  figure {
    @if $border-color {
      border: 1px solid $border-color;
    }
    width: $width;

    img {
      max-width: $width;
      @if($align == left)  {
        float: left;
        margin: 2em 0;
      } @else if($align  == center) {
        margin: 2em auto;
      } @else {
        float: right;
        margin: 2em 0;
      }
    }

    figcaption {
      max-width: max-content;
    }
  }
}
```

The mixin uses some SASS tricks to make using the mixing less of a pain.

- It first tests if the `$border-color` variable and, if it exists, we add a border attribute to the figure element
- For the image element inside the figure:
    - We set the `max-width` attribute to the value of the `$width` parameter
    - Depending on the value of the `$align` we configure float and margin values
- we set `figcaption` to be as wide as the content from the parent. This way we'll have captions that are no wide than the associated image

One thing I haven't figure out yet is whether we can use this mixin to place captions in other places of the page rather than below the image. Some of my favorite layouts have the captions to the sides of the image and I'd like to replicate them.

Sure, I can always replace `figure` and `figcaption` with nested `div` elements but it looses some of the semantic richness of `figure` and children. More research is needed.

How we write our text on screen

I thought all there was to writing on the web was type the content, use the right tags and publish your content to the web. That was true as long as we only had to worry about western languages (those covered by Latin and Latin Extended character sets). However non western languages like arabic, hebrew, CJK (Chinese, Japanese and Korean) write completely differently either from the horizontal side or read from top to bottom.

We’ll explore some of these methods and ways in which we can use them in typographical and layout tricks

Writing modes

Writing modes are the basic way in which we tell CSS how to lay the content of our page.

```css
.foo {
  writing-mode: horizontal-tb;
}

.foo {
  writing-mode: vertical-rl;
}

.foo {
  writing-mode: vertical-lr;
}

.foo {
  writing-mode: sideways-rl;
}

.foo {
  writing-mode: sideways-lr;
}

.foo {
  writing-mode: inherit;
}
```

Possible Values

horizontal-tb
: Content flows horizontally from left to right, vertically from top to bottom. The next horizontal line is positioned below the previous line.

vertical-rl
: Content flows vertically from top to bottom, horizontally from right to left. The next vertical line is positioned to the left of the previous line.

vertical-lr
: Content flows vertically from top to bottom, horizontally from left to right. The next vertical line is positioned to the right of the previous line.

sideways-rl
: Content flows vertically from top to bottom and all the glyphs, even those in vertical scripts, are set sideways toward the right.

sideways-lr
: Content flows vertically from top to bottom and all the glyphs, even those in vertical scripts, are set sideways toward the left.

sideways-rl
: Content flows vertically from top to bottom and all the glyphs, even those in vertical scripts, are set sideways toward the right.

sideways-lr
: Content flows vertically from top to bottom and all the glyphs, even those in vertical scripts, are set sideways toward the left.

It'd be interesting to see how the sideways writing modes (`sidewars-rl` and `sideways-lr`) work with western scrips meant to be written vertically in a horizontal line.

Direction

There are two concepts related to the direction property:

- **Inline direction** (ltr, rtl or tb) the direction in which we read. This changes based on the language we are working with

- **Block direction** is the flow in which we read (how the content is laid out, usually top to bottom)

If I'm understanding this correctly writing mode (discussed earlier) controls the block direction and the direction property controls t he horizontal flow of the text (lr or rl).

```
direction: ltr;
direction: rtl;
direction: inherit;
```

Posible values

ltr
: The initial value of direction (that is, if not otherwise specified). Text and other elements go from left to right.

rtl
: Text and other elements go from right to left

> For the direction property to have any effect on inline-level elements, the unicode-bidi property's value must be embed or override.

Text Orientation

Text orientation is where we can have fun mixing the way content is laid out. This is particularly interesting when mixing characters with different writing modes, we can roate English characters so they'll be vertical along Japanese Kanji.

```
.foo {
  text-orientation: mixed;
}

.foo {
  text-orientation: upright;
}

.foo {
  text-orientation: sideways-right;
}

.foo {
  text-orientation: sideways;
}
.foo {
  text-orientation: use-glyph-orientation;
}

.foo {
  text-orientation: inherit;
}
```

Possible values

mixed
: This keyword leads to have characters for horizontal-only scripts being turned 90°, while the glyphs for vertical scripts are laid out naturally.

upright
: This keyword leads to have characters for horizontal-only scripts to be laid out naturally (upright), as well as the glyphs for vertical scripts. Note that this keyword lead all characters to be considered as ltr: the used value of direction is ltr, whatever the user could try to set it to. sideways

  This keyword leads to have characters laid out like if they were laid out horizontally, but with the whole line rotated 90° to the right if writing-mode is vertical-rl or to the left, if it is vertical-lr.

sideways
: This keyword leads to have characters laid out like if they were laid out horizontally, but with the whole line rotated 90° to the right if writing-mode is vertical-rl or to the left, if it is vertical-lr.

use-glyph-orientation
: On SVG elements, this keyword leads to use the value of the deprecated SVG properties glyph-orientation-vertical and glyph-orientation-horizontal.

2D Transformations

2D transformations will let us play with the visual positioning of content. I can already rotate and control the flow of text but the transformations allow placement in different locations on the page without having to worry about the language or other writing mode considerations.

These are not the only transform functions available. There are also z-axis transformations, matrix and matrix3d and perspective functions. For this first round I've decided to remove them from my menu of options. Most of the CSS transform functions I'm interested in:

```
/* Function values */

.demo {
  transform: translate(12px, 50%);
}

.demo {
  transform: translateX(2em);
}

.demo {
  transform: translateY(3in);
}

.demo {
  transform: scale(2, 0.5);
}

.demo {
  transform: scaleX(2);
}

.demo {
  transform: scaleY(0.5);
}

.demo {
  transform: skew(30deg, 20deg);
}

.demo {
  transform: skewX(30deg);
```

```css
}

.demo {
  transform: skewY(1.07rad);
}

.demo {
  transform: rotate(0.5turn);
}

.demo {
  transform: rotateX(10deg);
}

.demo {
  transform: rotateY(10deg);
}
```

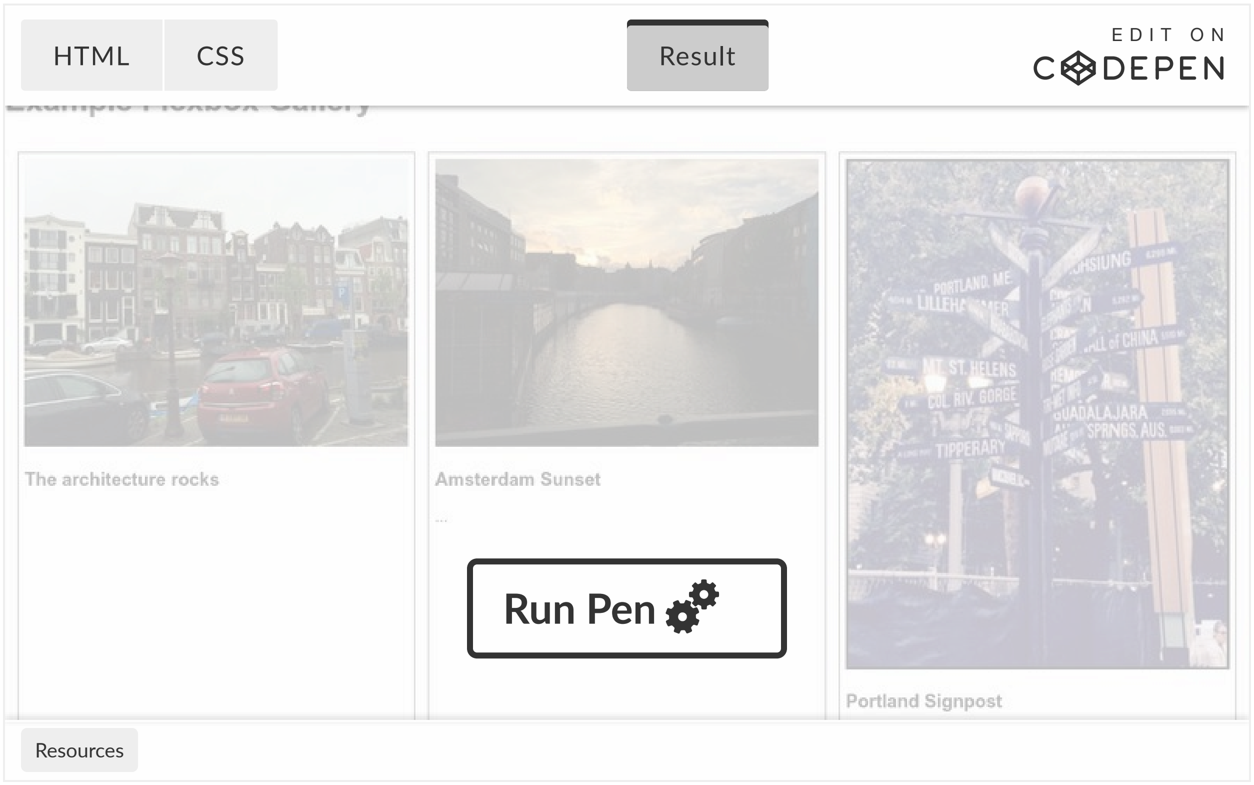
Note that there are three values for each property (`rotate`, `skew` and `translate`), this will allow us to work with both x and y values together or independently using the x or y values.

There may be times when we need to perform multiple transformations on an element to achieve the desired effect. We can chain the transformations effects like in the example below where the object will be moved 10 pixels in the X axis, rotated 10 degrees and moved 5 pixels horizontally (in the Y axis)

```css
/* Multiple function values */
.demo {
  transform: translateX(10px) rotate(10deg) translateY(5px);
}
```

Flexbox

Flexbox layouts are the easiest way to build repetitive content like navigation menus or image galleries.



To get the full efect, open the pen in [full view]() and play with making the browser window narrower. the images will resize themselves and will change sizes until they become a single column display.

Pay particular attention to the CSS and see how little code we need to accomplish this effect. Granted, we're using a small subset of the [CSS Flexible Box Layout Module Level 1]() specification but even this small subset gives us the following capabilities:

- As many equal width columns as there is space available
- Equal vertical height (all boxes are as tall as the tallest element)
- Fluid layout (the boxes will wrap if there is not enough space available)