24

2006

24 WAYS

Credits

24 ways is the advent calendar for web geeks. For twenty-four days each December we publish a daily dose of web design and development goodness to bring you all a little Christmas cheer.

- *24 ways* is brought to you by Perch CMS
- Produced by Drew McLellan, Brian Suda, Anna Debenham and Owen Gregory.
- Designed by Paul Robert Lloyd.
- eBook published by edgeofmyseat.com and produced by Rachel Andrew.
- Possible only with the help and dedication of our authors.

2006

In March, the first tweets were tweeted; in August, jQuery 1.0 appeared. In its second year, 24 ways wrote responsible JavaScript and hinted at a mobile web, although mobile phones didn't yet have proper browsers. Using CSS3 in client work was still a pipedream. And in October, IE7 was officially released by Microsoft — no words.

1. Tasty Text Trimmer

Drew McLellan 24ways.org/200601

In most cases, when designing a user interface it’s best to make a decision about how data is best displayed and stick with it. Failing to make a decision ultimately leads to too many user options, which in turn can be taxing on the poor old user.

Under some circumstances, however, it’s good to give the user freedom in customising their workspace. One good example of this is the ‘Article Length’ tool in Apple’s Safari RSS reader. Sliding a slider left of right dynamically changes the length of each article shown. It’s that kind of awesomey magic stuff that’s enough to keep you from sleeping. Let’s build one.

THE SETUP

Let's take a page that has lots of long text items, a bit like a news page or like Safari's RSS items view. If we were to attach a class name to each element we wanted to resize, that would give us something to hook onto from the JavaScript.

Example 1: The basic page.

As you can see, I've wrapped my items in a `DIV` and added a class name of `chunk` to them. It's these chunks that we'll be finding with the JavaScript. Speaking of which …

OUR CORE FUNCTIONS

There are two main tasks that need performing in our script. The first is to find the chunks we're going to be resizing and store their original contents away somewhere safe. We'll need this so that if we trim the text down we'll know what it was if the user decides they want it back again. We'll call this `loadChunks`.

```
var loadChunks = function(){
  var everything = document.getElementsByTagName('*');
  var i, l;
  chunks  = [];
  for (i=0, l=everything.length; i<l; i++){
    if (everything[i].className.indexOf(chunkClass) >
-1){
      chunks.push({
        ref: everything[i],
        original: everything[i].innerHTML
      });
    }
  }
};
```

The variable `chunks` is stored outside of this function so that we can access it from our next core function, which is `doTrim`.

```
var doTrim = function(interval) {
  if (!chunks) loadChunks();
  var i, l;
  for (i=0, l=chunks.length; i<l; i++){
    var a = chunks[i].original.split(' ');
    a = a.slice(0, interval);
    chunks[i].ref.innerHTML  = a.join(' ');
  }
};
```

The first thing that needs to be done is to call `loadChunks` if the `chunks` variable isn't set. This should only happen the first time `doTrim` is called, as from that point the chunks will be loaded.

Then all we do is loop through the chunks and trim them. The trimming itself (lines 6-8) is very simple. We split the text into an array of words (line 6), then select only a portion from the beginning of the array up until the number we want (line 7). Finally the words are glued back together (line 8).

In essense, that's it, but it leaves us needing to know how to get the number into this function in the first place, and how that number is generated by the user. Let's look at the latter case first.

THE YUI SLIDER WIDGET

There are lots of JavaScript libraries available at the moment. A fair few of those are really good. I use the Yahoo! User Interface Library professionally, but have only recently played with their pre-build slider widget. Turns out, it's pretty good and perfect for this task.

Once you have the library files linked in (check the docs linked above) it's fairly straightforward to create yourself a slider.

```
slider = YAHOO.widget.Slider.getHorizSlider("sliderbg",
"sliderthumb", 0, 100, 5);
slider.setValue(50);
slider.subscribe("change", doTrim);
```

All that's needed then is some CSS to make the slider look like a slider, and of course a few bits of HTML. We'll see those later.

SEE IT WORKING!

Rather than spell out all the nuts and bolts of implementing this fairly simple script, let's just look at in it action and then pick on some interesting bits I've added.

Example 2: Try the Tasty Text Trimmer.

At the top of the JavaScript file I've added a small number of settings.

```
var chunkClass  = 'chunk';
var minValue  = 10;
var maxValue  = 100;
var multiplier  = 5;
```

Obvious candidates for configuration are the class name used to find the chunks, and also the some minimum and maximum values. The `minValue` is the fewest number of words we wish to display when the slider is all the way down. The `maxValue` is the length of the slider, in this case 100.

Moving the slider makes a call to our `doTrim` function with the current value of the slider. For a slider 100 pixels long, this is going to be in the range of 0-100. That might be okay for some things, but for longer items of text you'll want to allow for displaying more than 100 words. I've accounted for this by adding in a multiplier – in my code I'm multiplying the value by 5, so a slider value of 50 shows 250 words. You'll probably want to tailor the multiplier to the type of content you're using.

KEEPING IT ACCESSIBLE

This effect isn’t something we can really achieve without JavaScript, but even so we must make sure that this functionality has no adverse impact on the page when JavaScript isn’t available. This is achieved by adding the slider markup to the page from within the `insertSliderHTML` function.

```
var insertSliderHTML = function(){
  var s = '<a id="slider-less" href="#less"><img 
src="icon_min.gif" width="10" height="10" alt="Less 
text" class="first" /></a>';
  s +=' <div id="sliderbg"><div id="sliderthumb"><img 
src="sliderthumbimg.gif" /></div></div>';
  s +=' <a id="slider-more" href="#more"><img 
src="icon_max.gif" width="10" height="10" alt="More 
text" /></a>';
  document.getElementById('slider').innerHTML = s;
}
```

The other important factor to consider is that a slider can be tricky to use unless you have good eyesight and pretty well controlled motor skills. Therefore we should provide a method of changing the value by the keyboard.

I’ve done this by making the icons at either end of the slider links so they can be tabbed to. Clicking on either icon fires the appropriate JavaScript function to show more or less of the text.

IN CONCLUSION

The upshot of all this is that without JavaScript the page just shows all the text as it normally would. With JavaScript we have a slider for trimming the text excepts that can be controlled with the mouse or with a keyboard.

If you’re like me and have just scrolled to the bottom to find the working demo, here it is again:

Try the Tasty Text Trimmer

Trimmer for Christmas? Don’t say I never give you anything!

ABOUT THE AUTHOR

Drew McLellan is lead developer on your favourite small CMS, Perch. He is Director and Senior Developer at UK-based web development agency edgeofmyseat.com, and formerly Group Lead at the Web Standards Project. When not publishing 24 ways, Drew keeps a personal site covering web development issues and themes, takes photos and tweets a lot.

2. Faster Development with CSS Constants

Rachel Andrew 24ways.org/200602

Anyone even slightly familiar with a programming language will have come across the concept of constants – a fixed value that can be used through your code. For example, in a PHP script I might have a constant which is the email address that all emails generated by my application get sent to.

```
$adminEmail = 'info@example.com';
```

I could then use `$adminEmail` in my script whenever I wanted an email to go to that address. The benefit of this is that when the client decides they want the email to go to a different address, I only need change it in one place – the place where I initially set the constant. I could also quite easily make this value user defined and enable the administrator to update the email address.

Unfortunately CSS doesn't support constants. It would be really useful to be able to define certain values initially and then use them throughout a CSS file, so in this article I'm going to take a look at some of the methods we do have available and provide pointers to more in depth commentary on each. If you have a different method, or tip to share please add it to the comments.

SO WHAT OPTIONS DO WE HAVE?

One way to get round the lack of constants is to create some definitions at the top of your CSS file in comments, to define 'constants'. A common use for this is to create a 'color glossary'. This means that you have a quick reference to the colors used in the site to avoid using alternates by mistake and, if you need to change the colors, you have a quick list to go down and do a search and replace.

In the below example, if I decide I want to change the mid grey to `#999999`, all I need to do is search and replace `#666666` with `#999999` – assuming I've remember to always use that value for things which are mid grey.

```
/*
Dark grey (text): #333333
Dark Blue (headings, links) #000066
Mid Blue (header) #333399
```

```
Light blue (top navigation) #CCCCFF
Mid grey: #666666
*/
```

This is a fairly low-tech method, but if used throughout the development of the CSS files can make changes far simpler and help to ensure consistency in your color scheme.

I've seen this method used by many designers however Garrett Dimon documents the method, with more ideas in the comments.

GOING SERVER-SIDE

To truly achieve constants you will need to use something other than CSS to process the file before it is sent to the browser. You can use any scripting language – PHP, ASP, ColdFusion etc. to parse a CSS file in which you have entered constants. So that in a constants section of the CSS file you would have:

```
$darkgrey = '#333333';
$darkblue = '#000066';
```

The rest of the CSS file is as normal except that when you come to use the constant value you would use the constant name instead of adding the color:

```
p {
  color: $darkgrey;
}
```

Your server-side script could then parse the CSS file, replace the constant names with the constant values and serve a valid CSS file to the browser. Christian Heilmann has done just this for PHP however this could be adapted for any language you might have available on your server.

Shaun Inman came up with another way of doing this that removes the need to link to a PHP script and also enables the adding of constants using the syntax of at-rules . This method is again using PHP and will require you to edit an `.htaccess` file.

A further method is to generate static CSS files either using a script locally – if the constants are just to enable speed of development – or as part of the web application itself. Storing a template stylesheet with constant names in place of the values you will want to update means that your script can simply open the template, replace the variables and save the result as a new stylesheet file.

While CSS constants are a real help to developers, they can also be used to add new functionality to your applications. As with the email address example that I used at the beginning of this article, using a combination of CSS and server-side scripting you could enable a site administrator to select the colours for a new theme to be used on a page of a content managed site. By using constants you need only give them the option to change certain parts of the CSS and not upload a whole different CSS file, which could lead to some interesting results!

As we are unlikely to find real CSS constants under the tree this Christmas the above methods are some possibilities for better management of your stylesheets. However if you have better methods, CSS Constant horror stories or any other suggestions, add your comments below.

ABOUT THE AUTHOR

Rachel Andrew is a Director of edgeofmyseat.com, a UK web development consultancy and creators of the small content management system, Perch. She is the author of a number of books, most recently The Profitable Side Project Handbook and CSS3 Layout Modules, and is a regular columnist for A List Apart.

When not writing about business and technology on her blog at rachelandrew.co.uk or speaking at conferences, you will usually find Rachel running up and down one of the giant hills in Bristol.

3. Flickr Photos On Demand with getFlickr

Christian Heilmann 24ways.org/200603

In case you don't know it yet, Flickr is great. It is a lot of fun to upload, tag and caption photos and it is really handy to get a vast network of contacts through it.

Using Flickr photos outside of it is a bit of a problem though. There is a Flickr API, and you can get almost every page as an RSS feed, but in general it is a bit tricky to use Flickr photos inside your blog posts or web sites. You might not want to get into the whole API game or use a server side proxy script as you cannot retrieve RSS with Ajax because of the cross-domain security settings.

However, Flickr also provides an undocumented JSON output, that can be used to hack your own solutions in JavaScript without having to use a server side script.

- If you enter the URL http://flickr.com/photos/tags/panda you get to the flickr page with photos tagged "panda".

- If you enter the URL http://api.flickr.com/services/feeds/photos_public.gne?tags=panda&format=rss_200 you get the same page as an RSS feed.
- If you enter the URL http://api.flickr.com/services/feeds/photos_public.gne?tags=panda&format=json you get a JavaScript function called `jsonFlickrFeed` with a parameter that contains the same data in JSON format

You can use this to easily hack together your own output by just providing a function with the same name. I wanted to make it easier for you, which is why I created the helper `getFlickr` for you to download and use.

GETFLICKR FOR NON-SCRIPTERS

Simply include the javascript file `getflickr.js` and the style `getflickr.css` in the head of your document:

```
<script type="text/javascript"
src="getflickr.js"></script>
<link rel="stylesheet" href="getflickr.css" type="text/
css">
```

Once this is done you can add links to Flickr pages anywhere in your document, and when you give them the CSS class `getflickrphotos` they get turned into gallery links. When a visitor clicks these links they turn into loading messages and show a “popup” gallery with the connected photos once they were loaded. As the JSON returned is very small it won’t take long. You can close the gallery, or click any of the thumbnails to view a photo. Clicking the photo makes it disappear and go back to the thumbnails.

Check out the example page and click the different gallery links to see the results.

Notice that `getFlickr` works with Unobtrusive JavaScript as when scripting is disabled the links still get to the photos on Flickr.

GETFLICKR FOR JAVASCRIPT HACKERS

If you want to use `getFlickr` with your own JavaScripts you can use its main method `leech()`:

```
getFlickr.leech(sTag, sCallback);
```

`sTag`
: the tag you are looking for

`sCallback`
: an optional function to call when the data was retrieved.

After you called the `leech()` method you have two strings to use:

`getFlickr.html[sTag]`
: contains an HTML list (without the outer `UL` element) of all the images linked to the correct pages at flickr. The images are the medium size, you can easily change that by replacing _m.jpg with _s.jpg for thumbnails.

`getFlickr.tags[sTag]`
: contains a string of all the other tags flickr users added with the tag you searched for(space separated)

You can call `getFlickr.leech()` several times when the page has loaded to cache several result feeds before the page gets loaded. This'll make the photos quicker for the end user to show up. If you want to offer a form for people to search for flickr photos and display them immediately you can use the following HTML:

```
<form
onsubmit="getFlickr.leech(document.getElementById('tag').value,
'populate');return false">
  <label for="tag">Enter Tag</label>
  <input type="text" id="tag" name="tag" />
  <input type="submit" value="energize" />
  <h3>Tags:</h3><div id="tags"></div>
  <h3>Photos:</h3><ul id="photos"></ul>
</form>
```

All the JavaScript you'll need (for a basic display) is this:

```
function populate(){
  var tag = document.getElementById('tag').value;
  document.getElementById('photos').innerHTML =
getFlickr.html[tag].replace(/_m\.jpg/g,'_s.jpg');
  document.getElementById('tags').innerHTML =
getFlickr.tags[tag];
  return false;
}
```

Easy as pie, enjoy!

Check out the example page and try the form to see the results.

AVAILABLE IN GERMAN
webkrauts.de

ABOUT THE AUTHOR

Christian Heilmann grew up in Germany and, after a year working for the red cross, spent a year as a radio producer. From 1997 onwards he worked for several agencies in Munich as a web developer. In 2000 he moved to the States to work for Etoys and, after the .com crash, he moved to the UK where he lead the web development department at Agilisys. In April 2006 he joined Yahoo! UK as a web developer and moved on to be the Lead Developer Evangelist for the Yahoo Developer Network. In December 2010 he moved on to Mozilla as Principal Developer Evangelist for HTML5 and the Open Web. He publishes an almost daily blog at http://wait-till-i.com and runs an article repository at http://icant.co.uk. He also authored Beginning JavaScript with DOM Scripting and Ajax: From Novice to Professional.

4. Rounded Corner Boxes the CSS3 Way

Andy Budd 24ways.org/200604

If you've been doing CSS for a while you'll know that there are approximately 3,762 ways to create a rounded corner box. The simplest techniques rely on the addition of extra mark-up directly to your page, while the more complicated ones add the mark-up though DOM manipulation. While these techniques are all very interesting, they do seem somewhat of a kludge. The goal of CSS is to separate structure from presentation, yet here we are adding superfluous mark-up to our code in order to create a visual effect. The reason we are doing this is simple. CSS2.1 only allows a single background image per element.

Thankfully this looks set to change with the addition of multiple background images into the CSS3 specification. With CSS3 you'll be able to add not one, not four, but eight background images to a single element. This means you'll be able to create all kinds of interesting effects without the need of those additional elements.

While the CSS working group still seem to be arguing over the exact syntax, Dave Hyatt went ahead and implemented the currently suggested mechanism into Safari. The technique is fiendishly simple, and I think we'll all be a lot better off once the W3C stop arguing over the details and allow browser vendors to get on and provide the tools we need to build better websites.

To create a CSS3 rounded corner box, simply start with your box element and apply your 4 corner images, separated by commas.

```
.box {
  background-image: url(top-left.gif),
url(top-right.gif), url(bottom-left.gif),
url(bottom-right.gif);
}
```

We don't want these background images to repeat, which is the normal behaviour, so lets set all their background-repeat properties to no-repeat.

```
.box {
  background-image: url(top-left.gif),
url(top-right.gif), url(bottom-left.gif),
url(bottom-right.gif);
```

```
  background-repeat: no-repeat, no-repeat, no-repeat,
no-repeat;
}
```

Lastly, we need to define the positioning of each corner image.

```
.box {
  background-image: url(top-left.gif),
url(top-right.gif), url(bottom-left.gif),
url(bottom-right.gif);
  background-repeat: no-repeat, no-repeat, no-repeat,
no-repeat;
  background-position: top left, top right, bottom left,
bottom right;
}
```

And there we have it, a simple rounded corner box with no additional mark-up.

As well as using multiple background images, CSS3 also has the ability to create rounded corners without the need of any images at all. You can do this by setting the border-radius property to your desired value as seen in the next example.

```
.box {
  border-radius: 1.6em;
}
```

This technique currently works in Firefox/Camino and creates a nice, if somewhat jagged rounded corner. If you want to create a box that works in both Mozilla and WebKit based browsers, why not combine both techniques and see what happens.

ABOUT THE AUTHOR

Andy Budd is an internationally renowned web designer, developer and weblog author based in Brighton, England. He specialises in building attractive, accessible, and standards complaint web solutions as a Director of Clearleft. Andy enjoys writing about web techniques for sites such as digital-web.com and his work has been featured in numerous magazines, books, and websites around the world. He is the author of CSS Mastery: Advanced Web Standards Solutions.

5. Accessible Dynamic Links

Mike Davies 24ways.org/200605

Although hyperlinks are the soul of the World Wide Web, it’s worth using them in moderation. Too many links becomes a barrier for visitors navigating their way through a page. This difficulty is multiplied when the visitor is using assistive technology, or is using a keyboard; being able to skip over a block of links doesn’t make the task of finding a specific link any easier.

In an effort to make sites easier to use, various user interfaces based on the hiding and showing of links have been crafted. From drop-down menus to expose the deeper structure of a website, to a decluttering of skip links so as not to impact design considerations. Both are well intentioned with the aim of preserving a good usability experience for the majority of a website's audience; hiding the real complexity of a page until the visitor interacts with the element.

WHEN JAVASCRIPT IS NOT AVAILABLE

The modern dynamic link techniques rely on JavaScript and CSS, but regardless of whether scripting and styles are enabled or not, we should consider the accessibility implications, particularly for screen-reader users, and people who rely on keyboard access.

In typical web standards-based drop-down navigation implementations, the rough consensus is that the navigation should be structured as nested lists so when JavaScript is not available the entire navigation map is available to the visitor. This creates a situation where a visitor is faced with potentially well over 50 links on every page of the website. Keyboard access to such structures is frustrating, there's far too many options, and the method of serially tabbing through each link looking for a specific one is tedious.

Instead of offering the visitor an indigestible chunk of links when JavaScript is not available, consider instead having the minimum number of links on a page, and when JavaScript is available bringing in the extra links dynamically. Santa Chris Heilmann offers an excellent proof of concept in making Ajax navigation optional.

When JavaScript is enabled, we need to decide how to hide links. One technique offers a means of comprehensively hiding links from keyboard users and assistive technology users. Another technique allows keyboard and screen-reader users to access links while they are hidden, and making them visible when reached.

HIDING THE LINKS

In JavaScript enhanced pages whether a link displays on screen depends on a certain event happening first. For example, a visitor needs to click a top-level navigation link that makes a set of sub-navigation links appear. In these cases, we need to ensure that these links are not available to any user until that event has happened.

The typical way of hiding links is to style the anchor elements, or its parent nodes with `display: none`. This has the advantage of taking the links out of the tab order, so they are not focusable. It's useful in reducing the number of links presented to a screen-reader or keyboard user to a minimum. Although the links are still in the document (they can be referenced and manipulated using DOM Scripting), they are not directly triggerable by a visitor.

Once the necessary event has happened, like our visitor has clicked on a top-level navigation link which shows our hidden set of links, then we can display the links to the visitor and make them triggerable. This is done simply by undoing the `display: none`, perhaps by setting the *display* back to `block` for block level elements, or `inline` for inline elements. For as long as this display style remains, the links are in the tab order, focusable by keyboard, and triggerable.

A common mistake in this situation is to use `visibility: hidden`, `text-indent: -999em`, or `position: absolute` with `left: -999em` to position these links off-screen. But all of these links remain accessible via keyboard tabbing even though the links remain hidden from screen view. In some ways this is a good idea, but for hiding sub-navigation links, it presents the screen-reader user and keyboard user with too many links to be of practical use.

MOVING THE LINKS OUT OF SIGHT

If you want a set of text links accessible to screen-readers and keyboard users, but don't want them cluttering up space on the screen, then style the links with `position: absolute; left: -999em`. Links styled this way remain in the tab order, and are accessible via keyboard. (The `position: absolute` is added as a style to the link, not to a parent node of the link – this will give us a useful hook to solve the next problem).

```
a.helper {
  position: absolute;
  left: -999em;
}
```

One important requirement when displaying links off-screen is that they are visible to a keyboard user when they receive focus. Tabbing on a link that is not visible is a usability mudpit, since the visitor has no visible cue as to what a focused link will do, or where it will go.

The simple answer is to restyle the link so that it appears on the screen when the hidden link receives focus. The anchor's `:focus` pseudo-class is a logical hook to use, and with the following style repositions the link onscreen when it receives the focus:

```
a.helper:focus, a.helper.focus {
  top: 0;
  left: 0;
}
```

This technique is useful for hiding skip links, and options you want screen-reader and keyboard users to use, but don't want cluttering up the page. Unfortunately Internet Explorer 6 and 7 don't support the `focus` pseudo-class, which is why there's a second CSS selector `a.helper.focus` so we can use some JavaScript to help out. When the page loads, we look for all links that have a class of helper and add in `onfocus` and `onblur` event handlers:

```
if (anchor.className == "helper") {
  anchor.onfocus = function() {
    this.className = 'helper focus';
  }
  anchor.onblur = function() {
    this.className = 'helper';
  }
}
```

Since we are using JavaScript to cover up for deficiencies in Internet Explorer, it makes sense to use JavaScript initially to place the links off-screen. That way an Internet Explorer user with JavaScript disabled can still use the skip link functionality.

It is vital that the number of links rendered in this way is kept to a minimum. Every link you offer needs to be tabbed through, and gets read out in a screen reader. Offer these off-screen links that directly benefit these types of visitor.

Andy Clarke and Kimberly Blessing use a similar technique in the Web Standards Project's latest design, but their technique involves hiding the skip link in plain sight and making it visible when it receives focus. Navigate the page using just the tab key to see the accessibility-related links appear when they receive focus.

This technique is also a good way of hiding image replaced text. That way the screen-readers still get the actual text, and the website still gets its designed look.

WHICH WAY?

If the links are not meant to be reachable until a certain event has occurred, then the `display: none` technique is the preferred approach. If you want the links accessible but out of the way until they receive focus, then the off-screen positioning (or Andy's hiding in plain sight technique) is the way to go.

ABOUT THE AUTHOR

Mike Davies works for Yahoo! Europe as a Web Developer with a focus on web accessibility. Online, he uses the moniker Isofarro and blogs about web accessibility and universality on isolani. His last project in his previous company (Legal & General) was presented at the launch of PAS 78 as a case study into the business benefits of web accessibility.

Photo: Neil Crosby

6. Hide And Seek in The Head

Peter-Paul Koch 24ways.org/200606

If you want your JavaScript-enhanced pages to remain accessible and understandable to scripted and noscript users alike, you have to think before you code. Which functionalities are required (ie. should work without JavaScript)? Which ones are merely nice-to-have (ie. can be scripted)? You should only start creating the site when you've taken these decisions.

SPECIAL HTML ELEMENTS

Once you have a clear idea of what will work with and without JavaScript, you'll likely find that you need a few HTML elements for the noscript version only.

Take this example: A form has a nifty bit of Ajax that automatically and silently sends a request once the user enters something in a form field. However, in order to preserve accessibility, the user should also be able to submit the form normally. So the form should have a submit button in noscript browsers, but not when the browser supports sufficient JavaScript.

Since the button is meant for noscript browsers, it must be hard-coded in the HTML:

```
<input type="submit" value="Submit form"
id="noScriptButton" />
```

When JavaScript is supported, it should be removed:

```
var checkJS = [check JavaScript support];
window.onload = function () {
  if (!checkJS) return;

document.getElementById('noScriptButton').style.display
= 'none';
}
```

PROBLEM: THE `LOAD` EVENT

Although this will likely work fine in your testing environment, it's not completely correct. What if a user with a modern, JavaScript-capable browser visits your page, but has to wait for a huge graphic to load? The `load` event fires only after all assets, including images, have been loaded. So this user will first see a submit button, but then all of a sudden it's removed. That's potentially confusing.

Fortunately there's a simple solution: play a bit of hide and seek in the `<head>`:

```
var checkJS = [check JavaScript support];
if (checkJS) {
  document.write('<style>#noScriptButton{display:
none}</style>');
}
```

First, check if the browser supports enough JavaScript. If it does, `document.write` an extra `<style>` element that hides the button.

The difference with the previous technique is that the `document.write` command is outside any function, and is therefore executed while the JavaScript is being parsed. Thus, the `#noScriptButton{display: none}` rule is written into the document before the actual HTML is received.

That's exactly what we want. If the rule is already present at the moment the HTML for the submit button is received and parsed, the button is hidden immediately. Even if the user (and the `load` event) have to wait for a huge image, the button is already hidden, and both scripted and noscript users see the interface they need, without any potentially confusing flashes of useless content.

In general, if you want to hide content that's not relevant to scripted users, give the hide command in CSS, and make sure it's given *before* the HTML element is loaded and parsed.

ALTERNATIVE

Some people won't like to use `document.write`. They could also add an empty `<link />` element to the `<head>` and give it an `href` attribute once the browser's JavaScript capabilities have been evaluated. The `<link />` element is made to refer to a style sheet that contains the crucial `#noScriptButton{display: none}`, and everything works fine.

Important note: The script needs access to the `<link />`, and the only way to ensure that access is to include the empty `<link />` element *before* your `<script>` tag.

ABOUT THE AUTHOR

Peter-Paul Koch (ppk for those In The Know) is a JavaScript guru residing in Amsterdam, the Netherlands. His cunning masterplan to get filthy rich consists of pimping his book, *ppk on JavaScript*, without which it's impossible to lead a succesful, or even happy, life.

7. Making XML Beautiful Again: Introducing Client-Side XSL

Ian Forrester 24ways.org/200607

Remember that first time you saw XML and got it? When you really understood what was possible and the deep meaning each element could carry? Now when you see XML, it looks ugly, especially when you navigate to a page of XML in a browser. Well, with every modern browser now supporting XSL 1.0, I'm going to show you how you can turn something as simple as an ATOM feed into a customised page using a browser, Notepad and some XSL.

WHAT ON EARTH IS THIS XSL?

XSL is a family of recommendations for defining XML document transformation and presentation. It consists of three parts:

- **XSLT 1.0 – Extensible Stylesheet Language Transformation**, a language for transforming XML

- **XPath 1.0 – XML Path Language**, an expression language used by XSLT to access or refer to parts of an XML document. (XPath is also used by the XML Linking specification)
- **XSL-FO 1.0 – Extensible Stylesheet Language Formatting Objects**, an XML vocabulary for specifying formatting semantics

XSL transformations are usually a one-to-one transformation, but with newer versions (XSL 1.1 and XSL 2.0) its possible to create many-to-many transformations too. So now you have an overview of XSL, on with the show…

SO WHAT DO I NEED?

So to get going you need a browser an supports client-side XSL transformations such as Firefox, Safari, Opera or Internet Explorer. Second, you need a source XML file – for this we're going to use an ATOM feed from Flickr.com. And lastly, you need an editor of some kind. I find Notepad++ quick for short XSLs, while I tend to use XMLSpy or Oxygen for complex XSL work.

Because we're doing a client-side transformation, we need to modify the XML file to tell it where to find our yet-to-be-written XSL file. Take a look at the source XML file, which originates from my Flickr photos tagged *sky*, in ATOM format.

The top of the ATOM file now has an additional `<?xml-stylesheet />` instruction, as can been seen on Line 2 below. This instructs the browser to use the XSL file to transform the document.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<?xml-stylesheet type="text/xsl"
href="flickr_transform.xsl"?>
<feed xmlns="http://www.w3.org/2005/Atom"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
```

YOUR FIRST TRANSFORMATION

Your first XSL will look something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:atom="http://www.w3.org/2005/Atom"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
  <xsl:output method="html" encoding="utf-8"/>
</xsl:stylesheet>
```

This is pretty much the starting point for most XSL files. You will notice the standard XML processing instruction at the top of the file (line 1). We then switch into XSL mode using the XSL namespace on all XSL elements (line 2). In this case, we have added namespaces for ATOM (line 4) and Dublin Core (line 5). This means the XSL can now read and *understand* those elements from the source XML.

After we define all the namespaces, we then move onto the `xsl:output` element (line 6). This enables you to define the final method of output. Here we're specifying `html`, but you could equally use XML or Text, for example. The encoding attributes on each element do what they say on the tin. As with all XML, of course, we close every element including the root.

The next stage is to add a template, in this case an `<xsl:template />` as can be seen below:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:atom="http://www.w3.org/2005/Atom"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
  <xsl:output method="html" encoding="utf-8"/>
  <xsl:template match="/">
    <html>
      <head>
        <title>Making XML beautiful again : Transforming
ATOM</title>
      </head>
      <body>
        <xsl:apply-templates select="/atom:feed"/>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

The beautiful thing about XSL is its English syntax, if you say it out loud it tends to make sense.

The / value for the match attribute on line 8 is our first example of XPath syntax. The expression / matches any element – so this `<xsl:template/>` will match against any element in the document. As the first element in any XML document is the root element, this will be the one matched and processed first.

Once we get past our standard start of a HTML document, the only instruction remaining in this `<xsl:template/>` is to look for and match all `<atom:feed/>` elements using the `<xsl:apply-templates/>` in line 14, above.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:atom="http://www.w3.org/2005/Atom"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
  <xsl:output method="html" encoding="utf-8"/>
  <xsl:template match="/">
    <xsl:apply-templates select="/atom:feed"/>
  </xsl:template>
  <xsl:template match="/atom:feed">
    <div id="content">
      <h1>
        <xsl:value-of select="atom:title"/>
      </h1>
      <p>
        <xsl:value-of select="atom:subtitle"/>
      </p>
      <ul id="entries">
        <xsl:apply-templates select="atom:entry"/>
      </ul>
    </div>
  </xsl:template>
</xsl:stylesheet>
```

This new template (line 12, above) matches `<feed/>` and starts to write the new HTML elements out to the output stream. The `<xsl:value-of/>` does exactly what you'd expect – it finds the value of the item specifed in its `select` attribute. With XPath you can select any element or attribute from the source XML.

The last part is a repeat of the now familiar `<xsl:apply-templates/>` from before, but this time we're using it inside of a *called* template. Yep, XSL is full of recursion…

```
<xsl:template match="atom:entry">
  <li class="entry">
    <h2>
      <a href="{atom:link/@href}">
        <xsl:value-of select="atom:title"/>
      </a>
    </h2>
    <p class="date">
      (<xsl:value-of
select="substring-before(atom:updated,'T')"/>)
    </p>
    <p class="content">
      <xsl:value-of select="atom:content"
disable-output-escaping="yes"/>
    </p>
    <xsl:apply-templates select="atom:category"/>
  </li>
</xsl:template>
```

The `<xsl:template/>` which matches `atom:entry` (line 1) occurs every time there is a `<entry/>` element in the source XML file. So in total that is 20 times, this is naturally why XSLT is full of recursion. This `<xsl:template/>` has been matched and therefore called higher up in the document, so we can start writing list elements directly to the output stream. The first part is simply a `<h2/>` with a link wrapped within it (lines 3-7). We can select attributes using XPath using `@`.

The second part of this template selects the date, but performs a XPath string function on it. This means that we only get the date and not the time from the string (line 9). This is achieved by getting only the part of the string that exists before the `T`.

Regular Expressions are not part of the XPath 1.0 string functions, although XPath 2.0 does include them. Because of this, in XSL we tend to rely heavily on the available XML output.

The third part of the template (line 12) is a `<xsl:value-of/>` again, but this time we use an attribute of `<xsl:value-of/>` called `disable output escaping` to turn escaped characters back into XML.

The very last section is another `<xsl:apply-template/>` call, taking us three templates deep. Do not worry, it is not uncommon to write XSL which go 20 or more templates deep!

```
<xsl:template match="atom:category">
  <xsl:for-each select=".">
    <xsl:element name="a">
      <xsl:attribute name="rel">
        <xsl:text>tag</xsl:text>
      </xsl:attribute>
      <xsl:attribute name="href">
        <xsl:value-of select="concat(@scheme, @term)"/>
      </xsl:attribute>
      <xsl:value-of select="@term"/>
    </xsl:element>
    <xsl:text> </xsl:text>
  </xsl:for-each>
</xsl:template>
```

In our final `<xsl:template/>`, we see a combination of what we have done before with a couple of twists. Once we match `atom:category` we then count how many elements there are at that same level (line 2). The XPath `.` means 'self', so we count how many `category` elements are within the `<entry/>` element.

Following that, we start to output a link with a `rel` attribute of the predefined text, `tag` (lines 4-6). In XSL you can just type text, but results can end up with strange whitespace if you do (although there are ways to simply remove all whitespace).

The only new XPath function in this example is `concat()`, which simply combines what XPaths or text there might be in the brackets. We end the output for this tag with an actual tag name (line 10) and we add a space afterwards (line 12) so it won't touch the next tag. (There are better ways to do this in XSL using the `last()` XPath function).

After that, we go back to the `<xsl:for-each/>` element again if there is another `category` element, otherwise we end the `<xsl:for-each/>` loop and end this `<xsl:template/>`.

A TOUCH OF STYLE

Because we're using recursion through our templates, you will find this is the end of the templates and the rest of the XML will be ignored by the parser. Finally, we can add our CSS to finish up. (I have created one for Flickr and another for News feeds)

```
<style type="text/css" media="screen">@import
"flickr_overview.css?v=001";</style>
```

So we end up with a nice simple to understand but also quick to write XSL which can be used on ATOM Flickr feeds and ATOM News feeds. With a little playing around with XSL, you can make XML beautiful again.

All the files can be found in the zip file (14k)

ABOUT THE AUTHOR

Ian Forrester heads up the BBC's Backstage, a developer/designer network like no other. He's well known for geek social events across the capital including London Geekdinners, BarCampLondon and recently the BBC Backstage London Christmas Bash. He's currently master minding plans BarCampLondon2, a series of backstage social events across the UK and something very special. Somehow, Ian finds time to blog at cubicgarden.com and think about user generated and xml pipelines at his new blog called flow *.

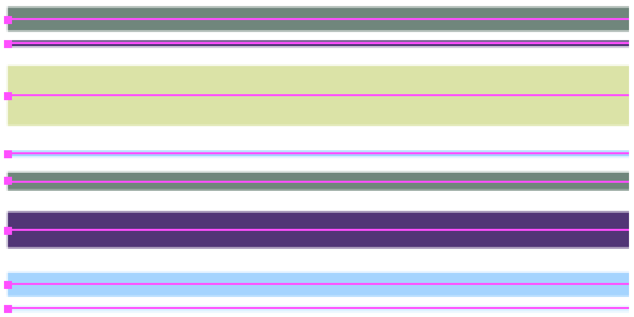
8. Random Lines Made With Mesh

Veerle Pieters

24ways.org/200608

I know that Adobe Illustrator can be a bit daunting for people who aren't really advanced users of the program, but you would be amazed by how easy you can create cool effects or backgrounds. In this short tutorial I show you how to create a cool looking background only in 5 steps.

STEP 1 – CREATE LINES

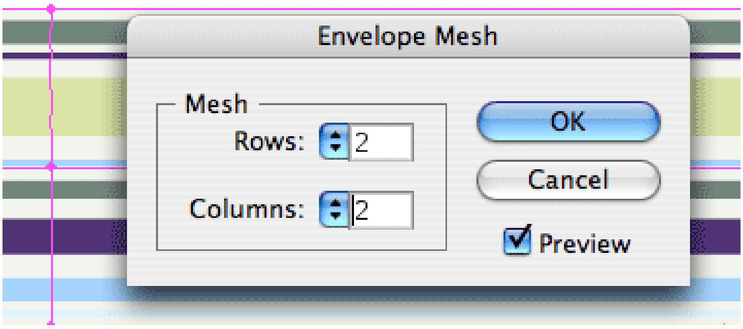


Create lines using random widths and harmonious suitable colors. If you get stuck on finding the right colors, check out Adobe's Kuler and start experimenting.

STEP 2 – CONVERT STROKES TO FILLS

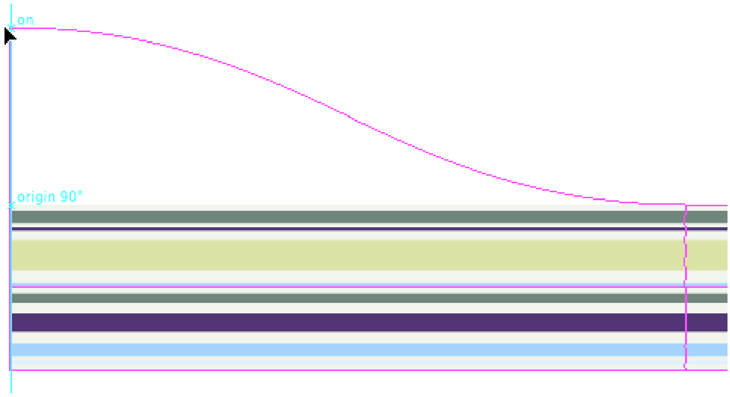
Select all lines and convert them to fills. Go to the Object menu, select Path > Outline Stroke. Select the Rectangle tool and draw 1 big rectangle on top the lines. Give the rectangle a suitable color. With the rectangle still selected, go to the Object menu, select Arrange > Send to Back.

STEP 3 – CONVERT TO MESH

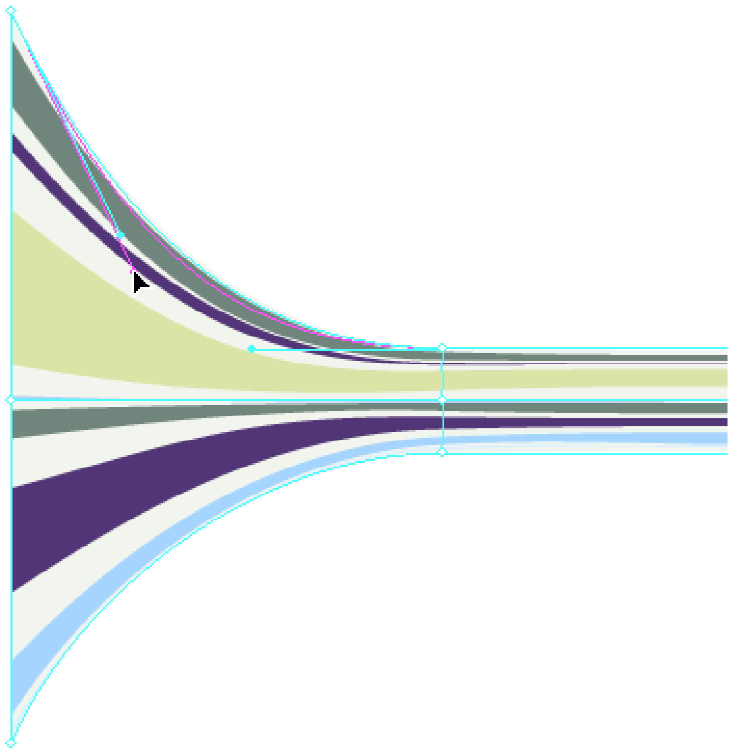


Select all objects by pressing the command key (for Mac users), control key (for Windows users) + the “a” key. Go to the Object menu and select the Envelope Distort > Make with Mesh option. Enter 2 rows and 2 columns. Check the preview box to see what happens and click the OK button.

STEP 4 – PLAY AROUND WITH THE MESH POINTS

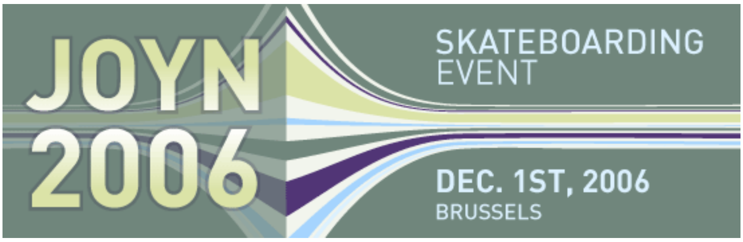


Play around with the points of the mesh using the Direct Selection tool (the white arrow in the Toolbox). Click on the top right point of the mesh. Once you're starting to drag hold down the shift key and move the point upwards.



Now start dragging the bezier handles on the mesh to achieve the effect as shown in the above picture. Of course you can try out all kind of different effects here.

THE FINAL RESULT



This is an example of how the final result can look. You can try out all kinds of different shapes dragging the handles of the mesh points. This is just one of the many results you can get. So next time you haven't got inspiration for a background of a header, a banner or whatever, just experiment with a few basic shapes such as lines and try out the 'Envelope Distort' options in Illustrator or the 'Make with Mesh' option and experiment, you'll be amazed by the unexpected creative results.

ABOUT THE AUTHOR

Veerle Pieters is a graphic/web designer based in Deinze, Belgium. Starting in '92 as a freelance graphic designer, Veerle worked on print design before focussing more on webdesign and GUI (since '96). She runs her own design studio Duoh! together with Geert Leyseele. Veerle has been blogging since 2003 and is considered number 39 on the list of "NxE's Fifty Most Influential 'Female' Bloggers".

9. Marking Up a Tag Cloud

Mark Norman Francis 24ways.org/200609

Everyone's doing it.

The problem is, everyone's doing it *wrong*.

Harsh words, you might think. But the crimes against decent markup are legion in this area. You see, I'm something of a markup and semantics junkie. So I'm going to analyse some of the more well-known tag clouds on the internet, explain what's wrong, and then show you one way to do it better.

DEL.ICIO.US

I think the first ever tag cloud I saw was on del.icio.us. Here's how they mark it up.

```
<div class="alphacloud">
  <a href="/tag/.net" class="lb s2">.net</a>
  <a href="/tag/advertising" class=" s3">advertising</a>
  <a href="/tag/ajax" class=" s5">ajax</a>
  ...
</div>
```

Unfortunately, that is one of the *worst* examples of tag cloud markup I have ever seen. The page states that a tag cloud is “a list of tags where size reflects popularity”. However, despite describing it in this way to the human readers, the page’s author hasn’t described it that way in the markup. It isn’t a list of tags, just a bunch of anchors in a `<div>`. This is also inaccessible because a screenreader will not pause between adjacent links, and in some configurations will not announce the individual links, but rather all of the tags will be read as just one link containing a whole bunch of words. *Markup crime number one.*

FLICKR

Ah, Flickr. The darling photo sharing site of the internet, and the biggest blind spot in every standardista’s vision. Forgive it for having atrocious markup and sometimes confusing UI because it’s just so much damn fun to use. Let’s see what they do.

```
<p id="TagCloud">
   <a href="/photos/tags/06/" style="font-size:
14px;">06</a>
   <a href="/photos/tags/africa/" style="font-size:
12px;">africa</a>
   <a href="/photos/tags/amsterdam/" style="font-size:
14px;">amsterdam</a>
  ...
</p>
```

Again we have a simple collection of anchors like del.icio.us, only this time in a paragraph. But rather than using a class to represent the size of the tag they use an inline style. An inline style using a pixel-based font size. That's so far away from the goal of separating style from content, they might as well use a `<font>` tag. You could theoretically parse that to extract the information, but you have more work to guess what the pixel sizes represent. *Markup crime number two* (and extra jail time for using non-breaking spaces purely for visual spacing purposes.)

TECHNORATI

Ah, now. Here, you'd expect something decent. After all, the Overlord of microformats and King of Semantics Tantek Çelik works there. Surely we'll see something decent here?

```
<ol class="heatmap">
  <li><em><em><em><em><a href="/tag/
Britney+Spears">Britney
Spears</a></em></em></em></em></li>
  <li><em><em><em><em><em><em><em><em><em><a href="/tag/
Bush">Bush</a></em></em></em></em></em></em></em></em></em></li>

<li><em><em><em><em><em><em><em><em><em><em><em><em><em><a
href="/tag/
Christmas">Christmas</a></em></em></em></em></em></em></em></em></em><
  ...
  <li><em><em><em><em><em><em><a href="/tag/
```

```
SEO">SEO</a></em></em></em></em></em></em></li>

<li><em><em><em><em><em><em><em><em><em><em><em><em><em><em><em><a
href="/tag/
Shopping">Shopping</a></em></em></em></em></em></em></em></em></em></e
  ...
</ol>
```

Unfortunately it turns out not to be that decent, and stop calling me Shirley. It's not exactly terrible code. It does recognise that a tag cloud is a list of links. And, since they're in alphabetical order, that it's an ordered list of links. That's nice. However … *fifteen* nested `<em>` tags? **FIFTEEN?** That's emphasis for you. Yes, it is parse-able, but it's also something of a strange way of looking at emphasis. The HTML spec states that `<em>` is emphasis, and `<strong>` is for stronger emphasis. Nesting `<em>` tags seems counter to the idea that different tags are used for different levels of emphasis. Plus, if you had a screen reader that stressed the voice for emphasis, what would it do? Shout at you? *Markup crime number three.*

SO WHAT SHOULD IT BE?

As del.icio.us tells us, a tag cloud is a list of tags where the size that they are rendered at contains extra information. However, by hiding the extra context purely within the CSS or the HTML tags used, you are denying that context to some users. The basic assumption being made is that all users will be able to see the difference between font sizes, and this is demonstrably false.

A better way to code a tag cloud is to put the context of the cloud within the content, not the markup or CSS alone. As an example, I'm going to take some of my favourite flickr tags and put them into a cloud which communicates the relative frequency of each tag.

To start with a tag cloud in its most basic form is just a list of links. I am going to present them in alphabetical order, so I'll use an ordered list. Into each list item I add the number of photos I have with that particular tag. The tag itself is linked to the page on flickr which contains those photos. So we end up with this first example. To display this as a traditional tag cloud, we need to alter it in a few ways:

- The items need to be displayed next to each other, rather than one-per-line
- The context information should be hidden from display (but not from screen readers)
- The tag should link to the page of items with that tag

Displaying the items next to each other simply means setting the display of the list elements to `inline`. The context can be hidden by wrapping it in a `<span>` and then using the off-left method to hide it. And the link just means adding an anchor (with `rel="tag"` for some extra microformats bonus points). So, now we have a simple collection of links in our second example.

The last stage is to add the sizes. Since we already have context in our content, the size is purely for visual rendering, so we can just use classes to define the different sizes. For my example, I'll use a range of class names from `not-popular` through `ultra-popular`, in order of smallest to largest, and then use CSS to define different font sizes. If you preferred, you could always use less verbose class names such as `size1` through `size6`. Anyway, adding some classes and CSS gives us our final example, a semantic and more accessible tag cloud.

ABOUT THE AUTHOR

Mark Norman Francis is obsessed with HTML, semantics, code quality and doing things right. He is based in London, England and hopes one day to start blogging properly at marknormanfrancis.com.

10. Writing Responsible JavaScript

Drew McLellan 24ways.org/200610

Without a doubt, JavaScript has been making something of a comeback in the last year. If you're involved in client-side development in any way at all, chances are that you're finding yourself writing more JavaScript now than you have in a long time.

If you learned most of your JavaScript back when DHTML was all the rage and before DOM Scripting was in vogue, there have been some big shifts in the way scripts are written. Most of these are in the way event handlers are assigned and functions declared. Both of these changes are driven by the desire to write scripts that are responsible page citizens, both in not tying behaviour to content and in taking care not to conflict with other scripts. I thought it may be useful to look at some of these more responsible approaches to learn how to best write scripts that are independent of the page content and are safely portable between different applications.

EVENT HANDLING

Back in the heady days of Web 1.0, if you wanted to have an object on the page react to something like a click, you would simply go ahead and attach an `onclick` attribute. This was easy and understandable, but much like the `font` tag or the `style` attribute, it has the downside of mixing behaviour or presentation in with our content. As we're learned with CSS, there are big benefits in keeping those layers separate. Hey, if it works for CSS, it should work for JavaScript too.

Just like with CSS, instead of adding an attribute to our element within the document, the more responsible way to do that is to look for the item from your script (like CSS does with a selector) and then assign the behaviour to it. To give an example, take this oldskool `onclick` use case:

```
<a id="anim-link" href="#"
onclick="playAnimation()">Play the animation</a>
```

This could be rewritten by removing the `onclick` attribute, and instead doing the following from within your JavaScript.

```
document.getElementById('anim-link').onclick =
playAnimation;
```

IT'S ALL IN THE TIMING

Of course, it's never quite that easy. To be able to attach that `onclick`, the element you're targeting has to exist in the page, and the page has to have finished loading for the DOM to be available. This is where the `onload` event is handy, as it fires once everything has finished loading. Common practise is to have a function called something like `init()` (short for initialise) that sets up all these event handlers as soon as the page is ready.

Back in the day we would have used the `onload` attibute on the `<body>` element to do this, but of course what we really want is:

```
window.onload = init;
```

As an interesting side note, we're using `init` here rather than `init()` so that the function is assigned to the event. If we used the parentheses, the `init` function would have been run at that moment, and the *result* of running the function (rather than the function itself) would be assigned to the event. Subtle, but important.

As is becoming apparent, nothing is ever simple, and we can't just go around assigning our initialisation function to `window.onload`. What if we're using other scripts in the page that might also want to listen out for that event? Whichever script got there last would overwrite everything that came before it. To manage this, we need a script that checks for any existing event handlers, and adds the new handler to it. Most of the JavaScript libraries have their own systems for doing this. If you're not using a library, Simon Willison has a good stand-alone example

```
function addLoadEvent(func) {
  var oldonload = window.onload;
  if (typeof window.onload != 'function') {
    window.onload = func;
  } else {
    window.onload = function() {
      if (oldonload) {
        oldonload();
      }
      func();
    }
  }
}
```

Obviously this is just a toe in the events model's complex waters. Some good further reading is PPK's Introduction to Events.

CARVING OUT YOUR OWN SPACE

Another problem that rears its ugly head when combining multiple scripts on a single page is that of making sure that the scripts don't conflict. One big part of that is ensuring that no two scripts are trying to create functions or variables with the same names. Reusing a name in JavaScript just over-writes whatever was there before it.

When you create a function in JavaScript, you'll be familiar with doing something like this.

```
function foo() {
  ... goodness ...
}
```

This is actually just creating a variable called `foo` and assigning a function to it. It's essentially the same as the following.

```
var foo = function() {
  ... goodness ...
}
```

This name `foo` is by default created in what's known as the 'global namespace' – the general pool of variables within the page. You can quickly see that if two scripts use `foo` as a name, they will conflict because they're both creating those variables in the global namespace.

A good solution to this problem is to add just one name into the global namespace, make that one item either a function or an object, and then add everything else you need inside that. This takes advantage of JavaScript's variable scoping to contain you mess and stop it interfering with anyone else.

CREATING AN OBJECT

Say I was wanting to write a bunch of functions specifically for using on a site called ‘Foo Online’. I’d want to create my own object with a name I think is likely to be unique to me.

```
var FOOONLINE = {};
```

We can then start assigning functions are variables to it like so:

```
FOOONLINE.message = 'Merry Christmas!';
FOOONLINE.showMessage = function() {
  alert(this.message);
};
```

Calling `FOOONLINE.showMessage()` in this example would alert out our seasonal greeting. The exact same thing could also be expressed in the following way, using the *object literal* syntax.

```
var FOOONLINE = {
  message: 'Merry Christmas!',
  showMessage: function() {
    alert(this.message);
  }
};
```

CREATING A FUNCTION TO CREATE AN OBJECT

We can extend this idea bit further by using a function that we run in place to return an object. The end result is the same, but this time we can use *closures* to give us something like private methods and properties of our object.

```
var FOOONLINE = function(){
  var message = 'Merry Christmas!';
  return {
    showMessage: function(){
      alert(message);
    }
  }
}();
```

There are two important things to note here. The first is the parentheses at the end of line 10. Just as we saw earlier, this runs the function in place and causes its *result* to be assigned. In this case the result of our function is the object that is returned at line 4.

The second important thing to note is the use of the `var` keyword on line 2. This ensures that the `message` variable is created inside the scope of the function and not in the global namespace. Because of the way closure works (which if you're not familiar with, just suspend your disbelief for a moment) that `message` variable is visible to everything inside the function but not outside. Trying to read `FOOONLINE.message` from the page would return `undefined`.

This is useful for simulating the concept of private class methods and properties that exist in other programming languages. I like to take the approach of making **everything** private unless I know it's going to be needed from outside, as it makes the interface into your code a lot clearer for someone else to read.

ALL CHANGE, PLEASE

So that was just a whistle-stop tour of a couple of the bigger changes that can help to make your scripts better page citizens. I hope it makes useful Sunday reading, but obviously this is only the tip of the iceberg when it comes to designing modular, reusable code.

For some, this is all familiar ground already. If that's the case, I encourage you to perhaps submit a comment with any useful resources you've found that might help others get up to speed. Ultimately it's in all of our interests to make sure that all our JavaScript interoperates well – share your tips.

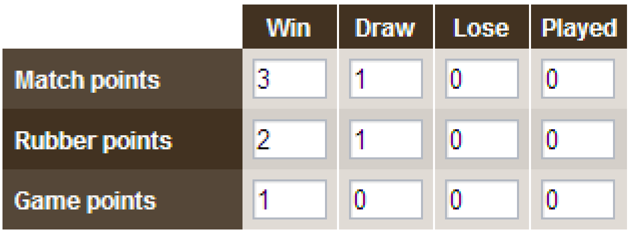
ABOUT THE AUTHOR

Drew McLellan is lead developer on your favourite small CMS, Perch. He is Director and Senior Developer at UK-based web development agency edgeofmyseat.com, and formerly Group Lead at the Web Standards Project. When not publishing 24 ways, Drew keeps a personal site covering web development issues and themes, takes photos and tweets a lot.

11. Showing Good Form

James Edwards 24ways.org/200611

Earlier this year, I forget exactly when (it's been a good year), I was building a client site that needed widgets which look like this (designed, incidentally, by my erstwhile writing partner, Cameron Adams):



	Win	Draw	Lose	Played
Match points	3	1	0	0
Rubber points	2	1	0	0
Game points	1	0	0	0

Building this was a challenge not just in CSS, but in choosing the proper markup – how should such a widget be constructed?

MMM … MARKUP

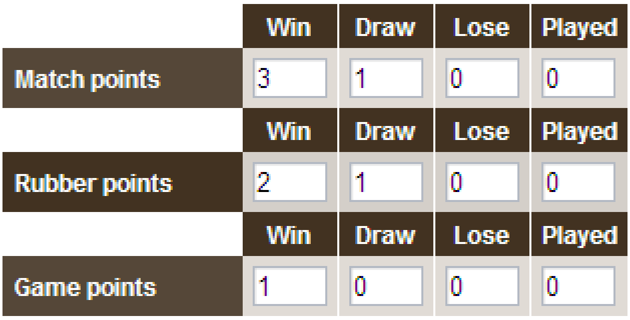
It seemed to me there were two key issues to deal with:

- The function of the interface is to *input information*, so semantically **this is a form**, therefore we have to find a way of building it using form elements: `fieldset`, `legend`, `label` and `input`
- We **can't use a table for layout**, even though that would clearly be the easiest solution!

Abusing tables for layout is never good – physical layout is not what table semantics mean. But even if this data *can* be described as a table, we **shouldn't mix forms markup with non-forms markup**, because of the behavioral impact this can have on a screen reader:

To take a prominent example, the screen reader JAWS has a mode specifically for interacting with forms (cunningly known as "forms mode"). When running in this mode its output *only includes relevant elements* – legends, labels and form controls themselves. Any other kind of markup – like text in a previous table cell, a paragraph or list in between – is simply ignored. The user in this situation would have to switch continually in and out of forms mode to hear all the content. (For more about this issue and some test examples, there's a thread at accessify forum which wanders in that direction.)

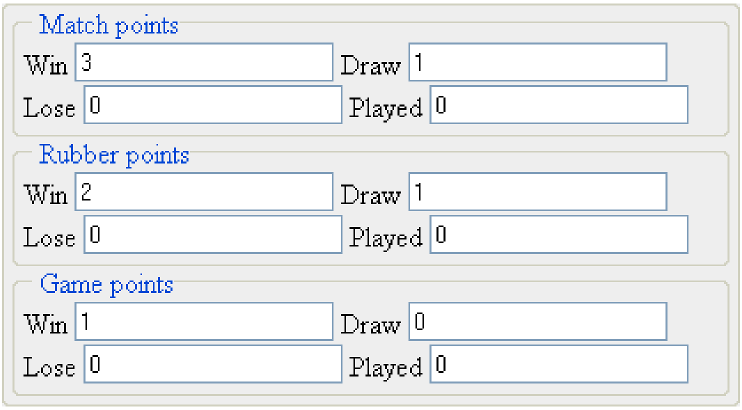
One further issue for screen reader users is implied by the design: the input fields are associated together in rows and columns, and a sighted user can visually scan across and down to make those associations; but a blind user can't do that. For such a user the row and column header data will need to be there at every axis; in other words, the layout should be more like this:



	Win	Draw	Lose	Played
Match points	3	1	0	0
	Win	Draw	Lose	Played
Rubber points	2	1	0	0
	Win	Draw	Lose	Played
Game points	1	0	0	0

And constructed with appropriate semantic markup to convey those relationships. By this point the selection of elements seems pretty clear: each row is a **fieldset**, the row header is a **legend**, and each column header is a **label**, associated with an input.

Here's what that form looks like with no CSS:



Match points

Win	3	Draw	1
Lose	0	Played	0

Rubber points

Win	2	Draw	1
Lose	0	Played	0

Game points

Win	1	Draw	0
Lose	0	Played	0

And here’s some markup for the first row (with most of the attributes removed just to keep this example succinct):

```
<fieldset>
  <legend>
    <span>Match points</span>
  </legend>
  <label>
    <span>Win</span>
    <input value="3" />
  </label>
  <label>
    <span>Draw</span>
    <input value="1" />
  </label>
  <label>
    <span>Lose</span>
    <input value="0" />
  </label>
  <label>
    <span>Played</span>
    <input value="0" />
  </label>
</fieldset>
```

The `span` inside each `legend` is because legend elements are highly resistant to styling! Indeed they’re one of the most stubborn elements in the browsers’ vocabulary. Oh man … how I wrestled with the buggers … until this obvious alternative occurred to me! So the legend element itself is just a container, while all the styling is on the inner `span`.

OH YEAH, THERE WAS SOME CSS TOO

I'm not gonna dwell too much on the CSS it took to make this work – this is a short article, and it's all there in the demo [demo page, style sheet]

But I do want to touch on the most interesting bit – where we get from a layout with headers on every row, to one where only the top row has headers – or at least, so it appears to graphical browsers. For screen readers, as we noted, we need those headers on every row, so we should employ some cunning CSS to partly negate their visual presence, without removing them from the output.

The core styling for each `label span` is like this:

```
label span
{
  display:block;
  padding:5px;
  line-height:1em;
  background:#423221;
  color:#fff;
  font-weight:bold;
}
```

But in the rows below the header they have these additional rules:

```
fieldset.body label span
{
  padding:0 5px;
  line-height:0;
  position:relative;
  top:-10000em;
}
```

The rendered width of the element is preserved, ensuring that the surrounding `label` is still the same width as the one in the header row above, and hence a unified column width is preserved all the way down. But the element effectively has no height, and so it's effectively invisible. The styling is done this way, rather than just setting the `height` to zero and using `overflow:hidden`, because to do that would expose an unrelated quirk with another popular screen reader! (It would hide the output from Window Eyes, as shown in this test example at access matters.)

THE FINISHED WIDGET

It's an intricate beast allright! But after all that we do indeed get the widget we want:

- Demo page
- Style sheet

It's not perfect, most notably because the legends have to have a fixed width; this can be in `em` to allow for text scaling, but it still doesn't allow the content to break into multiple lines. It also doesn't look quite right in Safari; and some CSS hacking was needed to make it look right in IE6 and IE7.

Still it worked well enough for the purpose, and satisfied the client completely. And most of all it re-assured me in my faith – that there's never any need to abuse tables for layout. (Unless of course you think this content *is* a table anyway, but that's another story!)

ABOUT THE AUTHOR

James Edwards (aka brothercake) is a freelance web developer based in the United Kingdom, specialising in advanced JavaScript programming and accessible website development. He is an outspoken advocate of standards-based development, an active member of WaSP (The Web Standards Project), and creator of the Ultimate Drop Down Menu system - the first commercial DHTML menu to be WCAG compliant. James was also co-author of The JavaScript Anthology, published by SitePoint in 2006.

12. Compose to a Vertical Rhythm

Richard Rutter 24ways.org/200612

“Space in typography is like time in music. It is infinitely divisible, but a few proportional intervals can be much more useful than a limitless choice of arbitrary quantities.” So says the typographer Robert Bringhurst, and just as regular use of time provides rhythm in music, so regular use of space provides rhythm in typography, and without rhythm the listener, or the reader, becomes disorientated and lost.

On the Web, vertical rhythm – the spacing and arrangement of text as the reader descends the page – is contributed to by three factors: font size, line height and margin or padding. All of these factors must calculated with care in order that the rhythm is maintained.

The basic unit of vertical space is line height. Establishing a suitable line height that can be applied to all text on the page, be it heading, body copy or sidenote, is the key to a solid dependable vertical rhythm, which will engage and guide the reader down the page. To see this in action, I've created an example with headings, footnotes and sidenotes.

ESTABLISHING A SUITABLE LINE HEIGHT

The easiest place to begin determining a basic line height unit is with the font size of the body copy. For the example I've chosen 12px. To ensure readability the body text will almost certainly need some leading, that is to say spacing between the lines. A `line-height` of 1.5em would give 6px spacing between the lines of body copy. This will create a total line height of 18px, which becomes our basic unit. Here's the CSS to get us to this point:

```
body {
  font-size: 75%;
}
html>body {
  font-size: 12px;
}
p {
  line-height 1.5em;
}
```

There are many ways to size text in CSS and the above approach provides and accessible method of achieving the pixel-precision solid typography requires. By way of explanation, the first `font-size` reduces the body text from the 16px default (common to most browsers and OS set-ups) down to the 12px we require. This rule is primarily there for Internet Explorer 6 and below on Windows: the percentage value means that the text will scale predictably should a user bump the text size up or down. The second `font-size` sets the text size specifically and is ignored by IE6, but used by Firefox, Safari, IE7, Opera and other modern browsers which allow users to resize text sized in pixels.

SPACING BETWEEN PARAGRAPHS

With our rhythmic unit set at 18px we need to ensure that it is maintained throughout the body copy. A common place to lose the rhythm is the gaps set between margins. The default treatment by web browsers of paragraphs is to insert a top- and bottom-margin of 1em. In our case this would give a spacing between the paragraphs of 12px and hence throw the text out of rhythm. If the rhythm of the page is to be maintained, the spacing of paragraphs should be related to the basic line height unit. This is achieved simply by setting top- and bottom-margins equal to the line height.

In order that typographic integrity is maintained when text is resized by the user we must use ems for all our vertical measurements, including line-height, padding and margins.

```
p {
  font-size:1em;
  margin-top: 1.5em;
  margin-bottom: 1.5em;
}
```

Browsers set margins on all block-level elements (such as headings, lists and blockquotes) so a way of ensuring that typographic attention is paid to all such elements is to reset the margins at the beginning of your style sheet. You could use a rule such as:

```
body,div,dl,dt,dd,ul,ol,li,h1,h2,h3,h4,h5,h6,pre,form,fieldset,p,block
{
  margin:0;
  padding:0;
}
```

Alternatively you could look into using the Yahoo! UI Reset style sheet which removes most default styling, so providing a solid foundation upon which you can explicitly declare your design intentions.

VARIATIONS IN TEXT SIZE

When there is a change in text size, perhaps with a heading or sidenotes, the differing text should also take up a multiple of the basic leading. This means that, in our example, every diversion from the basic text size should take up multiples of 18px. This can be accomplished by adjusting the `line-height` and `margin` accordingly, as described following.

HEADINGS

Subheadings in the example page are set to 14px. In order that the height of each line is 18px, the `line-height` should be set to 18 ÷ 14 = 1.286. Similarly the margins above and below the heading must be adjusted to fit. The temptation is to set heading margins to a simple 1em, but in order to maintain the rhythm, the top and bottom margins should be set at 1.286em so that the spacing is equal to the full 18px unit.

```
h2 {
  font-size:1.1667em;
  line-height: 1.286em;
  margin-top: 1.286em;
  margin-bottom: 1.286em;
}
```

One can also set asymmetrical margins for headings, provided the margins combine to be multiples of the basic line height. In our example, a top margin of 1½ lines is combined with a bottom margin of half a line as follows:

```
h2 {
  font-size:1.1667em;
  line-height: 1.286em;
  margin-top: 1.929em;
  margin-bottom: 0.643em;
}
```

Also in our example, the main heading is given a text size of 18px, therefore the `line-height` has been set to 1em, as has the margin:

```
h1 {
  font-size:1.5em;
  line-height: 1em;
  margin-top: 0;
  margin-bottom: 1em;
}
```

SIDENOTES

Sidenotes (and other supplementary material) are often set at a smaller size to the basic text. To keep the rhythm, this smaller text should still line up with body copy, so a calculation similar to that for headings is required. In our example, the sidenotes are set at 10px and so their line-height must be increased to 18 ÷ 10 = 1.8.

```
.sidenote {
  font-size:0.8333em;
  line-height:1.8em;
}
```

BORDERS

One additional point where vertical rhythm is often lost is with the introduction of horizontal borders. These effectively act as shims pushing the subsequent text downwards, so a two pixel horizontal border will throw out the vertical rhythm by two pixels. A way around this is to specify horizontal lines using background images or, as in our example, specify the width of the border in ems and adjust the padding to take up the slack.

The design of the footnote in our example requires a 1px horizontal border. The footnote contains 12px text, so 1px in ems is 1 ÷ 12 = 0.0833. I have added a margin of 1½ lines above the border (1.5 × 18 ÷ 12 = 2.5ems), so to maintain the rhythm the border + padding must equal a ½ (9px). We know the border is set to 1px, so the padding must be set to 8px. To specify this in ems we use the familiar calculation: 8 ÷ 12 = 0.667.

HIT ME WITH YOUR RHYTHM STICK

Composing to a vertical rhythm helps engage and guide the reader down the page, but it takes typographic discipline to do so. It may seem like a lot of fiddly maths is involved (a few divisions and multiplications never hurt anyone) but good type setting is all about numbers, and it is this attention to detail which is the key to success.

ABOUT THE AUTHOR

Richard Rutter is a user experience consultant and director of Clearleft. In 2009 he cofounded the webfont service, Fontdeck. He runs an ongoing project called The Elements of Typographic Style Applied to the Web, where he extols the virtues of good web typography. Richard occasionally blogs at Clagnut, where he writes about design, accessibility and web standards issues, as well as his passion for music and mountain biking.

13. Revealing Relationships Can Be Good Form

Ian Lloyd 24ways.org/200613

A few days ago, a colleague of mine – someone I have known for several years, who has been doing web design for several years and harks back from the early days of ZDNet – was running through a prototype I had put together for some user testing. As with a lot of prototypes, there was an element of ‘smoke and mirrors’ to make things look like they were working.

One part of the form included a yes/no radio button, and selecting the Yes option would, in the real and final version of the form, reveal some extra content. Rather than put too much JavaScript in the prototype, I took a preverbial shortcut and created a link which I wrapped around the text next to the radio button – clicking on that link would cause the form to mimic a change event on the radio button. But it wasn’t working for him.

Why was that? Because whereas I created the form using a `<label>` tag for each `<input>` and naturally went to click on the text rather than the form control itself, he was going straight for the control (and missing the sneaky little `<a href>` I’d placed around the text). Bah! There goes my time-saver.

So, what did I learn? That a web professional who has used the Internet for *years* had neither heard of the `<label>` tag, nor had he ever tried clicking on the text. It just goes to show that despite its obvious uses, the `label` element is not as well known as it rightfully deserves to be. So, what’s a web-standards-loving guy to do? Make a bit more bleedin’ obvious, that’s what!

THE MOUSE POINTER TRICK

OK, this is the kind of thing that causes some people outrage. A dead simple way of indicating that the `label` does something is to use a snippet of CSS to change the default mouse cursor to a hand. It’s derided because the hand icon is usually used for links, and some would argue that using this technique is misleading:

```
label {
  cursor: pointer;
}
```

This is not a new idea, though, and you didn't come here for this. The point is that with something very simple, you've made the label element discoverable. But there are other ways that you can do this that are web standards friendly and won't upset the purists quite so much as the hand/pointer trick. Time to wheel in the JavaScript trolley jack …

OUR OLD FRIEND ADDEVENT

First things, first, you'll need to use the addEvent function (or your favourite variation thereof) that Scott Andrew devised and make that available to the document containing the form:

```
function addEvent(elm, evType, fn, useCapture)
{
  if(elm.addEventListener)
  {
    elm.addEventListener(evType, fn, useCapture);
    return true;
  }
  else if (elm.attachEvent)
  {
    var r = elm.attachEvent('on' + evType, fn);
    return r;
  }
  else
  {
    elm['on' + evType] = fn;
  }
}
```

FINDING ALL YOUR LABELS

Once you've linked to the addEvent function (or embedded it on the page), you can start to get your JavaScripting fingers a-flexing. Now, what I'm suggesting you do here is:

- Identify all the `label` elements on the page by working your way through the DOM
- Find out the value of the `for` attribute for each `label` that you uncover
- Attach a behaviour or two to each of those `label` elements – and to the `input` that the `label` relates to (identified with the for attribute)

Here's the technobabble version of the steps above:

```
function findLabels()
{
  var el = document.getElementsByTagName("label");
  for (i=0;i<el.length;i++)
  {
    var thisId = el[i].getAttribute("for");
    if ((thisId)==null)
    {
      thisId = el[i].htmlFor;
    }
    if(thisId!="")
    {
      //apply these behaviours to the label
      el[i].onmouseover = highlightRelationship;
      el[i].onmouseout = hideRelationship;
    }
  }
}
function highlightRelationship()
{
  var thisId = this.getAttribute("for");
  if ((thisId)==null)
  {
    thisId = this.htmlFor;
  }
  this.className="showRel";
  document.getElementById(thisId).className="showRel";
  //if (document.getElementById(thisId).type=="text")
document.getElementById(thisId).select();
}
function hideRelationship()
{
  var thisId = this.getAttribute("for");
  if ((thisId)==null)
  {
    thisId = this.htmlFor;
  }
  this.className="";
  document.getElementById(thisId).className="";
}
addEvent(window, 'load', findLabels, false);
```

Using the above script, you can apply a CSS class (I've called it `showRel`) to the elements when you hover over them. How you want it to look is up to you, of course. Here are a few examples of the idea. Note: the design is not exactly what you'd call 'fancy', and in the examples there is one `input` that looks broken but it is *deliberately* moved away from the `label` it relates to, just to demonstrate that you can show the relationship even from afar.

- Background colour changes on hover
- Background colour change + mouse pointer trick
- Background colour change + mouse pointer trick + text selection

Hopefully you'll agree that using an unobtrusive piece of JavaScript you can make otherwise 'shy' elements like the label reveal their true colours. Although you might want to tone down the colours from the ones I've used in this demo!

ABOUT THE AUTHOR

Ian Lloyd founded Accessify.com, a web accessibility site, back in 2002 and has been a member of the Web Standards Project since 2003, where he is part of the Accessibility Task Force. He has written or co-authored a number of books on the topic of standards-based web design/development, most recently co-authoring on Pro CSS for Apress. He lives in Swindon, UK, a place best known for its 'Magic Roundabout' and Doctor Who's Billie Piper. (It's not all bad, though.)

14. Styling hCards with CSS

John Allsopp 24ways.org/200614

There are plenty of places online where you can learn about using the hCard microformat to mark up contact details at your site (there are some resources at the end of the article). But there's not yet been a lot of focus on using microformats with CSS. So in this installment of 24 ways, we're going to look at just that – how microformats help make CSS based styling simpler and more logical.

Being rich, quite complex structures, hCards provide designers with a sophisticated scaffolding for styling them. A recent example of styling hCards I saw, playing on the business card metaphor, was by Andy Hume, at http://thedredge.org/2005/06/using-hcards-in-your-blog/. While his approach uses fixed width cards, let's take a look at how we might style a variable width business card style for our hCards.

Let’s take a common hCard, which includes address, telephone and email details

```
<div class="vcard">
  <p class="fn org">Web Directions North
    <a href="http://suda.co.uk/projects/X2V/
get-vcard.php?uri=http://north.webdirections.org/
contact/">
      <img src="images/vcard-add.png" alt="download
vcard icon"></a>
  </p>
```

1485 Laperrière Avenue

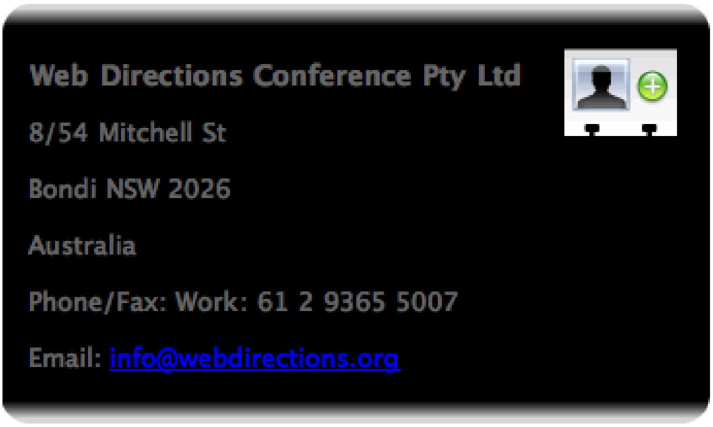
Ottawa ON K1Z 7S8

Canada

Phone/Fax: Work: 61 2 9365 5007

Email: info@webdirections.org

We’ll be using a variation on the now well established “sliding doors” technique (if you create a CSS technique, remember it’s very important to give it a memorable name or acronym, and bonus points if you get your name in there!) by Douglas Bowman, enhanced by Scott Schiller (see http://www.schillmania.com/projects/dialog/,) which will give us a design which looks like this


Web Directions Conference Pty Ltd

8/54 Mitchell St

Bondi NSW 2026

Australia

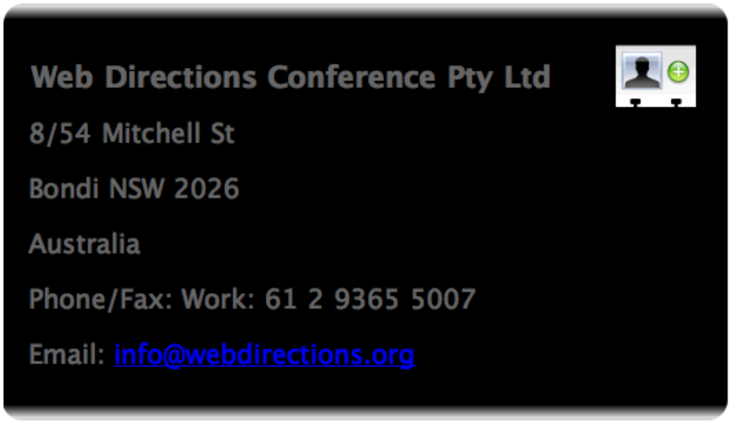
Phone/Fax: Work: 61 2 9365 5007

Email: info@webdirections.org

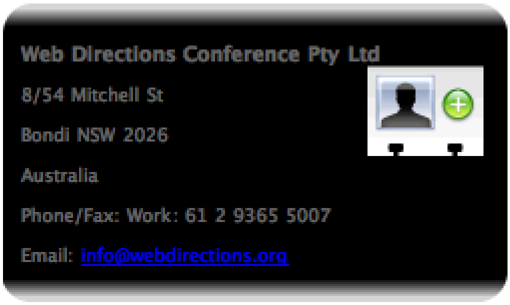
The technique, in a nutshell, uses background images on four elements, two at the top, and two at the bottom, to add each rounded corner.

We are going to make this design "fluid" in the sense that it grows and shrinks in proportion with the size of the font that the text of the element is displayed with. This is sometimes referred to as an "em driven design" (we'll see why in a moment).

To see how this works in practice, here's the same design with the text "zoomed" up in size


Web Directions Conference Pty Ltd
8/54 Mitchell St
Bondi NSW 2026
Australia
Phone/Fax: Work: 61 2 9365 5007
Email: info@webdirections.org

and the same design again, when we zoom the text size down


Web Directions Conference Pty Ltd
8/54 Mitchell St
Bondi NSW 2026
Australia
Phone/Fax: Work: 61 2 9365 5007
Email: info@webdirections.org

By the way, the hCard image comes from Chris Messina, and you can download it and other microformat icons from the microformats wiki.

Now, with CSS3, this whole task would be considerably easier, because we can add multiple background images to an element, and border images for each edge of an element. Safari, version 1.3 up, actually supports multiple background images, but sadly, it’s not supported in Firefox 1.5, or even Firefox 2.0 (let’s not mention IE7 eh?). So it’s probably too little supported to use now. So instead we’ll use a technique that only involves CSS2, and works in pretty much any browser.

Very often, developers add `div` or `span` elements as containers for these background images, and in fact, if you visit Scott Shiller’s site, that’s what he has done there. But if at all possible we shouldn’t be adding any HTML simply for presentational purposes, even if the presentation is done via CSS. What we can do is to use the HTML we have already, as much as is possible, to add the style we want. This can take some creative thinking, but once you get the hang of this approach it becomes a more natural way of using HTML compared with simply adding `divs` and `spans` at will as hooks for style. Of course, this technique isn’t always simple, and in fact sometimes simply not possible, requiring us to add just a little HTML to provide the “hooks” for CSS.

Let’s go to work

The first step is to add a background image to the whole vCard element.

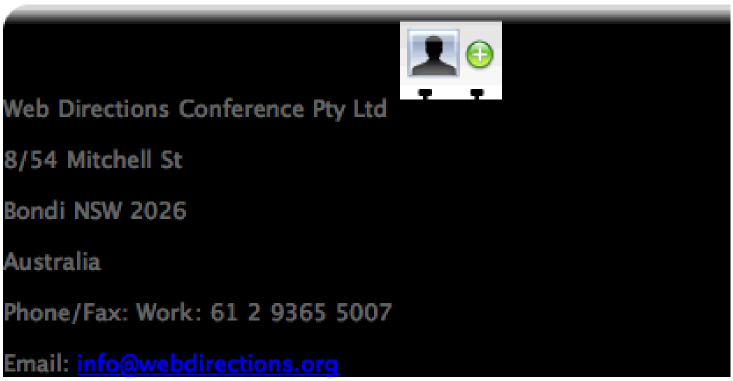
We make this wide enough (for example 1000 or more pixels) and tall enough that no matter how large the content of the vCard grows, it will never overflow this area. We can't simply repeat the image, because the top left corner will show when the image repeats.

We add this as the background image of the vCard element using CSS.

While we are at it, let's give the text a sans-serif font, some color so that it will be visible, and stop the image repeating.

```
.vcard {
  background-image: url(images/vcardfill.png);
  background-repeat: no-repeat;
  color: #666;
  font-family: "Lucida Grande", Verdana, Helvetica,
Arial, sans-serif;
}
```

Which in a browser, will look something like this.



Web Directions Conference Pty Ltd

8/54 Mitchell St

Bondi NSW 2026

Australia

Phone/Fax: Work: 61 2 9365 5007

Email: info@webdirections.org

Next step we need to add the top right hand corner of the hCard. In keeping with our aim of not adding HTML simply for styling purposes, we want to use the existing structure of the page where possible. Here, we'll use the paragraph of class `fn` and `org`, which is the first child element of the `vcard` element.

```
<p class="fn org">Web Directions Conference Pty Ltd <img
src="images/vcard-add.png" alt="download vcard icon"></p>
```

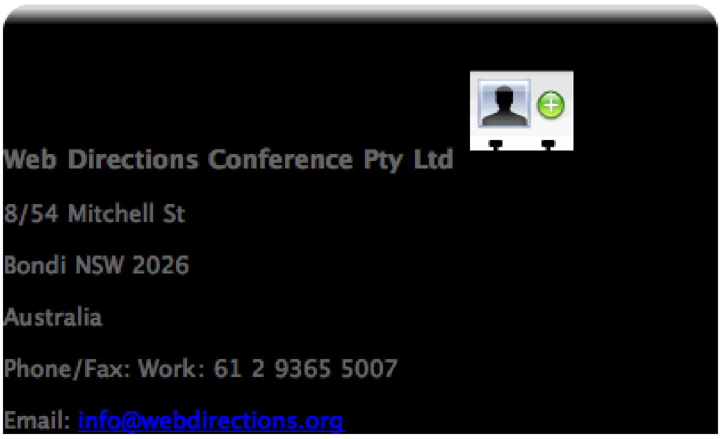
Here's our CSS for this element

```
.fn {
  background-image: url(images/topright.png);
  background-repeat: no-repeat;
  background-position: top right;
  padding-top: 2em;
  font-weight: bold;
  font-size: 1.1em;
}
```

Again, we don't want it to repeat, but this time, we've specified a background position for the image. This will make the background image start from the top, but its right edge will be located at the right edge of the element. I also made the font size a little bigger, and the weight bold, to differentiate it from the rest of the text in the hCard.

Here's the image we are adding as the background to this element.

So, putting these two CSS statements together we get



Web Directions Conference Pty Ltd

8/54 Mitchell St

Bondi NSW 2026

Australia

Phone/Fax: Work: 61 2 9365 5007

Email: info@webdirections.org

We specified a padding-top of 2em to give some space between the content of the `fn` element and the edge of the `fn` element. Otherwise the top of the hCard image would be hard against the border. To see this in action, just remove the `padding-top: 2em;` declaration and preview in a browser.

So, with just two statements, we are well under way. We've not even had to add any HTML so far. Let's turn to the bottom of the element, and add the bottom border (well, the background image which will serve as that border).

Now, which element are we going to use to add this background image to?

OK, here I have to admit to a little, teensie bit of cheating. If you look at the HTML of the hCard, I've grouped the email and telephone properties into a `div`, with a class of `telecommunications`. This grouping is not strictly requred for our hCard.

```
<div class="telecommunications">
  <p class="tel">Phone/Fax: <span class="tel"><span
class="type">Work</span>:
    <span class="value">61 2 9365 5007</span></p>
  <p class="email">Email: <a class="value"
href="mailto:info@webdirections.org">info@webdirections.org</a></p>
</div>
```

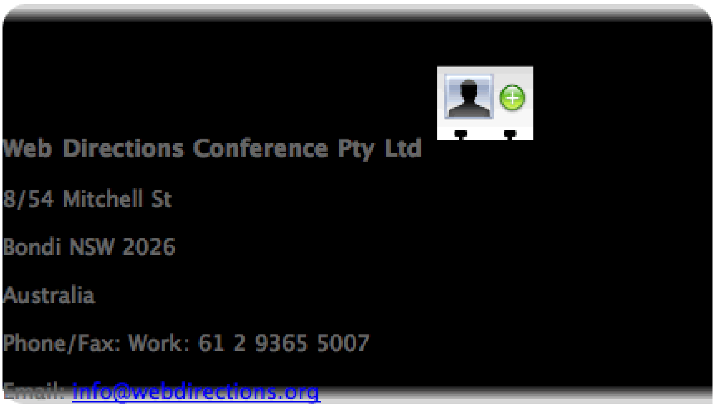
Now, I chose that class name because that is what the vCard specification calls this group of properties. And typically, I do tend to group together related elements using `divs` when I mark up content. I find it makes the page structure more logical and readable. But strictly speaking, this isn't necessary, so you may consider it cheating. But my lesson in this would be, if you are going to add markup, try to make it as meaningful as possible.

As you have probably guessed by now, we are going to add one part of the bottom border image to this element. We're going to add this image as the `background-image`.

Again, it will be a very wide image, like the top left one, so that no matter how wide the element might get, the background image will still be wide enough. Now, we'll need to make this image sit in the bottom left of the element we attach it to, so we use a backgound position of left bottom (we put the horizontal position before the vertical). Here's our CSS statement for this

```
.telecommunications {
  background-image: url(images/bottom-left.png);
  background-repeat: no-repeat;
  background-position: left bottom;
  margin-bottom: 2em;
}
```

And that will look like this


Web Directions Conference Pty Ltd
8/54 Mitchell St
Bondi NSW 2026
Australia
Phone/Fax: Work: 61 2 9365 5007
Email: info@webdirections.org

Not quite there, but well on the way. Time for the final piece in the puzzle.

OK, I admit, I might have cheated just a little bit more in this step. But like the previous step, all valid, and (hopefully) quite justifiable markup. If we look at the HTML again, you'll find that our email address is marked up like this

```
<p class="email">Email: <a class="value"
href="mailto:info@webdirections.org">info@webdirections.org</a></p>
```

Typically, in hCard, the value part of this property isn't required, and we could get away with

```
<a class="email"
href="mailto:info@webdirections.org">info@webdirections.org</a>
```

The form I've used, with the span of class value is however, perfectly valid hCard markup (hard allows for multiple email addresses of different types, which is where this typically comes in handy). Why have I gone to all this trouble? Well, when it came to styling the hCard, I realized I needed a block element to attach the background image for the bottom right hand corner to. Typically the last block element in the containing element is the ideal choice (and sometimes it's possible to take an inline element, for example the link here, and use CSS to make it a block element, and attach it to that, but that really doesn't work with this design).

So, if we are going to use the paragraph which contains the email link, we need a way to select it exclusively, which means that with CSS2 at least, we need a `class` or `id` as a hook for our CSS selector (in CSS3 we could use the `last-child` selector, which selects the last child element of a specified element, but again, as last child is not widely supported, we won't rely on it here.)

So, the least worst thing we could do is take an existing element, and add some reasonably meaningful markup to it. That's why we gave the paragraph a class of `email`, and the email address a class of `value`. Which reminds me a little of a moment in Hamlet

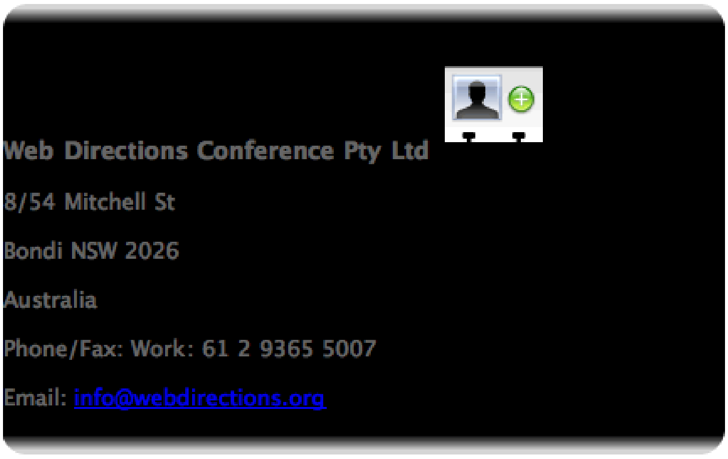
> The lady doth protest too much, methinks

OK, let's get back to the CSS.

We add the bottom right corner image, positioning it in the bottom right of the element, and making sure it doesn't repeat. We also add some padding to the bottom, to balance out the padding we added to the top of the hCard.

```
p.email {
  background-image: url(images/bottom-right.png);
  background-position: right bottom;
  background-repeat: no-repeat;
  padding-bottom: 2em;
}
```

Which all goes to make our hCard look like this


Web Directions Conference Pty Ltd

8/54 Mitchell St

Bondi NSW 2026

Australia

Phone/Fax: Work: 61 2 9365 5007

Email: info@webdirections.org

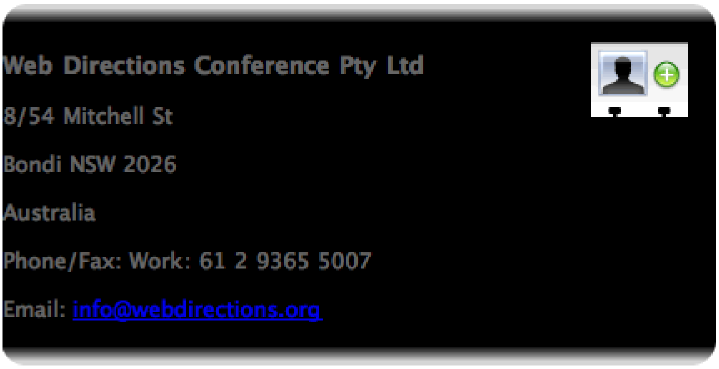
It just remains for us to clean up a little.

Let's start from the top. We'll float the download image to the right like this

```
.vcard img {
  float: right;
  padding-right: 1em;
  margin-top: -1em
}
```

See how we didn't have to add a class to style the image, we used the fact that the image is a descendent of the `vcard` element, and a descendent selector. In my experience, the very widely supported, powerful descendent selector is one of the most underused aspects of CSS. So if you don't use it frequently, look into it in more detail.

We added some space to the right of the image, and pulled it up a bit closer to the top of the hCard, like this



Web Directions Conference Pty Ltd

8/54 Mitchell St

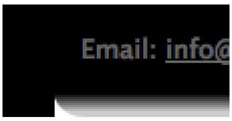
Bondi NSW 2026

Australia

Phone/Fax: Work: 61 2 9365 5007

Email: info@webdirections.org

We also want to add some whitespace between the edge of the hCard and the text. We would typically add padding to the left of the containing element, (in this case the `vcard` element) but this would break our bottom left hand corner, like this



That's because the div element we added this bottom left background image to would be moved in by the padding on its containing element.

So instead, we add left margin to all the paragraphs in the hCard

```
.vcard p {
  margin-left: 1em;
}
```

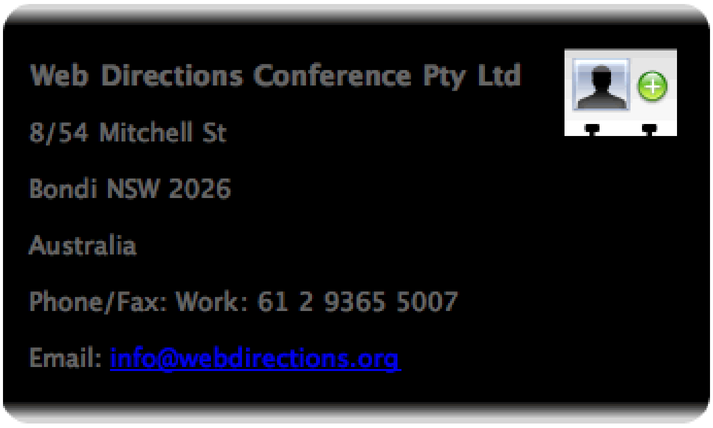
(there is the descendent selector again – it is the swiss army knife of CSS)

Now, we've not yet made the width of the hCard a function of the size of the text inside it (or "em driven" as we described it earlier). We do this by giving the hCard a width that is specified in em units. Here we'll set a width of say 28em, which makes the hCard always roughly as wide as 28 characters (strictly speaking 28 times the width of the letter capital M).

So the statement for our containing `vcard` element becomes

```
.vcard {
  background-image: url(images/vcardfill.png);
  background-repeat: no-repeat;
  color: #666;
  font-family: "Lucida Grande", Verdana, Helvetica,
Arial, sans-serif;
  width: 28em;
}
```

and now our element will look like this


Web Directions Conference Pty Ltd

8/54 Mitchell St

Bondi NSW 2026

Australia

Phone/Fax: Work: 61 2 9365 5007

Email: info@webdirections.org

We've used almost entirely the existing HTML from our original hCard (adding just a little, and trying as much as possible to keep that additional markup meaningful), and just 6 CSS statements.

Holiday Bonus – a downloadable vCard

Did you notice this part of the HTML

```
<a href="http://suda.co.uk/projects/X2V/
get-vcard.php?uri=http://north.webdirections.org/
contact/">
        <img src="images/vcard-add.png" alt="download
vcard icon"></a>
```

What's with the odd looking url

```
<a href="http://suda.co.uk/projects/X2V/
get-vcard.php?uri=http://north.webdirections.org/
contact/"
```

If you click the link, X2V, a nifty web service from Brian Suda, grabs the page at the URL, and if it finds a hCard, converts it to a vCard, and depending on how your system is setup, automatically downloads it and adds it to your address book (Mac OS X) or prompts you whether you'd like to save the vCard and add it to whatever application is the default vCard handler on your system.

What X2V does is take the actual HTML of your hCard, and with the magic of XSLT, converts it to a vCard. So, by simply marking up contact details using hCard, and adding a link like this, you automatically get downloadable vCard – and if you change your contact details, and update the hCard, there's no vCard file to update as well.

Technorati also have a similar service at http://technorati.com/contact so you might want to use that if you expect any kind of load, as they can probably afford the bandwidth more than Brian!

If you want to play with the HTML and CSS for this design, the code and images can be downloaded.

Hope you enjoyed this, and found it useful. If so, you might like to check out my microformats focussed blog, or get along to Web Directions North, where I'll be speaking along with Dan Cederholmn and Tantek Çelik in a 2 hour session focussed solely on microformats. And keep an eye out for my microformats book, from which this article has been adapted, coming in the spring of 2007.

A happy festive season, and all the best for 2007

John

Some hCard links

- The hCard entry at microformats.org
- The hCard Creator
- The hCard cheatsheet
- The hCard FAQ
- Ideas for authoring hCards

- Microfomatique – a blog about microformats
- Web Directions North – featuring a full 2 hour focussed microformats session

ABOUT THE AUTHOR

John Allsopp is a founder of Westciv, an Australian web software development and training company, which provides some of the best CSS resources and tutorials on the web. Westciv's software and training are used in dozens of countries around the World. The head developer of the leading cross platform CSS editor, Style Master, John has written on web development issues for numerous web and print publications and was one of the earliest members of the Web Standards Project.

15. A Message To You, Rudy - CSS Production Notes

Andrew Clarke

24ways.org/200615

When more than one designer or developer work together on coding an XHTML/CSS template, there are several ways to make collaboration effective. Some prefer to comment their code, leaving a trail of bread-crumbs for their co-workers to follow. Others use accompanying files that contain their working notes or communicate via Basecamp.

For this year's 24ways I wanted to share a technique that I has been effective at Stuff and Nonsense; one that unfortunately did not make it into the final draft of Transcending CSS. This technique, **CSS production notes**, places your page production notes in one convenient place within an XHTML document and uses nothing more than meaningful markup and CSS.

Let’s start with the basics; a conversation between a group of people. In the absence of *notes* or *conversation* elements in XHTML you need to make an XHTML compound that will effectively add meaning to the conversation between designers and developers. As each person speaks, you have two elements right there to describe what has been said and who has spoken: `<blockquote>` and its `cite` attribute.

```
<blockquote cite="andy">
  <p>This project will use XHTML1.0 Strict, CSS2.1 and
all that malarkey.</p>
</blockquote>
```

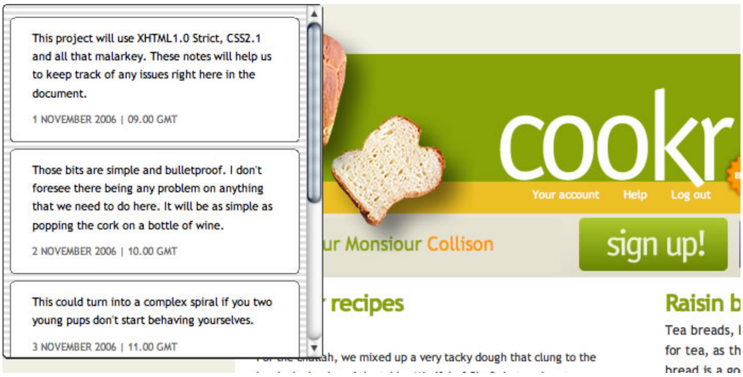
With more than one person speaking, you need to establish a temporal order for the conversation. Once again, the element to do just that is already there in XHTML; the humble ordered list.

```
<ol id="notes">
  <li>
    <blockquote cite="andy">
      <p>This project will use XHTML1.0 Strict, CSS2.1
and all that malarkey.</p>
    </blockquote>
  </li>
  <li>
    <blockquote cite="dan">
      <p>Those bits are simple and bulletproof.</p>
    </blockquote>
  </li>
</ol>
```

Adding a new note is as simple as adding a new item to list, and if you prefer to add more information to each note, such as the date or time that the note was written, go right ahead. Place your note list at the bottom of the source order of your document, right before the closing <body> tag. One advantage of this approach over using conventional comments in your code is that all the notes are unobtrusive and are grouped together in one place, rather than being spread throughout your document.

BASIC CSS STYLING

For the first stage you are going to add some basic styling to the notes area, starting with the ordered list. For this design I am basing the look and feel on an instant messenger window.



This project will use XHTML1.0 Strict, CSS2.1 and all that malarkey. These notes will help us to keep track of any issues right here in the document.

1 NOVEMBER 2006 | 09.00 GMT

Those bits are simple and bulletproof. I don't foresee there being any problem on anything that we need to do here. It will be as simple as popping the cork on a bottle of wine.

2 NOVEMBER 2006 | 10.00 GMT

This could turn into a complex spiral if you two young pups don't start behaving yourselves.

3 NOVEMBER 2006 | 11.00 GMT

```
ol#notes {
  width : 300px;
  height : 320px;
  padding : .5em 0;
  background : url(im.png) repeat;
  border : 1px solid #333;
  border-bottom-width : 2px;
  -moz-border-radius : 6px; /* Will not validate */
  color : #000;
  overflow : auto;
}
ol#notes li {
  margin : .5em;
  padding : 10px 0 5px;
  background-color : #fff;
  border : 1px solid #666;
  -moz-border-radius : 6px; /* Will not validate */
}
ol#notes blockquote {
  margin : 0;
  padding : 0;
}
ol#notes p {
  margin : 0 20px .75em;
  padding : 0;
}
ol#notes p.date {
  font-size : 92%;
  color : #666;
  text-transform : uppercase;
}
```

Take a gander at the first example.

You could stop right there, but without seeing who has left the note, there is little context. So next, extract the name of the commenter from the `<blockquote>`'s `cite` attribute and display it before each note by using generated content.

```
ol#notes blockquote:before {
  content : " "attr(cite)" said: ";
  margin-left : 20px;
  font-weight : bold;
}
```

FUN WITH MORE DETAILED STYLING

Now, with all of the information and basic styling in place, it's time to have some fun with some more detailed styling to spruce up your notes. Let's start by adding an icon for each person, once again based on their `cite`. First, all of the first paragraphs of a `<blockquote>`'s that includes a `cite` attribute are given common styles.

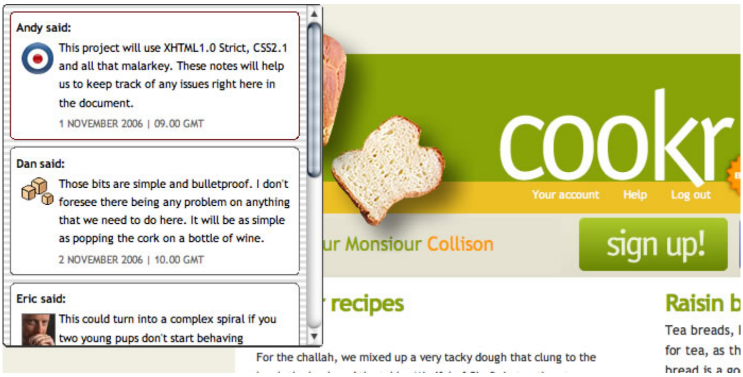
```
ol#notes blockquote[cite] p:first-child {
  min-height : 34px;
  padding-left : 40px;
}
```

Followed by an individual background-image.

```
ol#notes blockquote[cite="Andy"] p:first-child {
  background : url(malarkey.png) no-repeat 5px 5px;
}
```

If you prefer a little more interactivity, add a :hover state to each `<blockquote>` and perhaps highlight the most recent comment.

```
ol#notes blockquote:hover {
  background-color : #faf8eb;
  border-top : 1px solid #fff;
  border-bottom : 1px solid #333;
}
ol#notes li:last-child blockquote {
  background-color : #f1efe2;
}
```


Andy said:
This project will use XHTML1.0 Strict, CSS2.1 and all that malarkey. These notes will help us to keep track of any issues right here in the document.
1 NOVEMBER 2006 | 09.00 GMT

Dan said:
Those bits are simple and bulletproof. I don't foresee there being any problem on anything that we need to do here. It will be as simple as popping the cork on a bottle of wine.
2 NOVEMBER 2006 | 10.00 GMT

Eric said:
This could turn into a complex spiral if you two young pups don't start behaving

For the challah, we mixed up a very tacky dough that clung to the

Tea breads, I
for tea, as th
bread is a go

You could also adjust the style for each comment based on the department that the person works in, for example:

```
<li>
  <blockquote cite="andy" class="designer">
    <p>This project will use XHTML1.0 Strict, CSS2.1 and
all that malarkey.</p>
  </blockquote>
</li>
<li>
```

```
  <blockquote cite="dan">
    <p>Those bits are simple and bulletproof.</p>
  </blockquote>
</li>
ol#notes blockquote.designer { border-color : #600; }
```

Take a look at the results of the second stage.

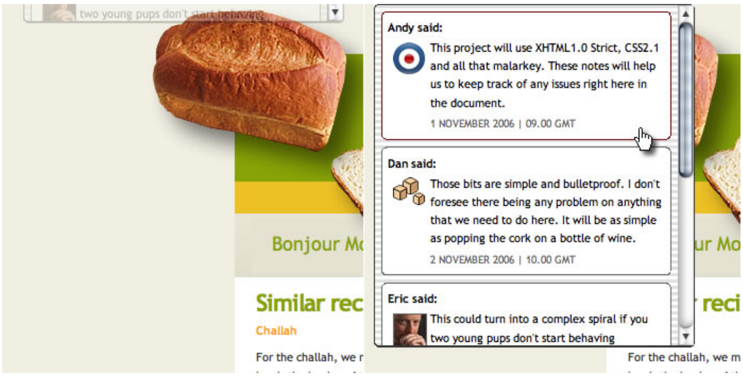
SHOW AND HIDE THE NOTES USING CSS POSITIONING

With your notes now dressed in their finest, it is time to tuck them away above the top of your working XHTML/CSS prototype so that you can reveal them when you need them, no JavaScript required. Start by moving the ordered list of notes off the top of the viewport leaving only a few pixels in view. It is also a good idea to make them semi-transparent by using the `opacity` property for browsers that have implemented it.

```
ol#notes {
  position : absolute;
  opacity : .25;
  z-index : 2000;
  top : -305px;
  left : 20px;
}
```

Your last step is to add `:hover` and `:focus` dynamic pseudo-classes to reposition the list at the top of the viewport and restore full opacity to display them in their full glory when needed.

```
ol#notes:hover, ol#notes:focus {
  top : 0;
  opacity : 1;
}
```


Andy said:
This project will use XHTML1.0 Strict, CSS2.1 and all that malarkey. These notes will help us to keep track of any issues right here in the document.
1 NOVEMBER 2006 | 09.00 GMT

Dan said:
Those bits are simple and bulletproof. I don't foresee there being any problem on anything that we need to do here. It will be as simple as popping the cork on a bottle of wine.
2 NOVEMBER 2006 | 10.00 GMT

Eric said:
This could turn into a complex spiral if you two young pups don't start behaving

Now it's time to sit back, pour yourself a long drink and bask in the glory of the final result. Your notes are all stored in one handy place at the bottom of your document rather than being spread around your code. When your templates are complete, simply dive straight to the bottom and pull out the notes.

A MESSAGE TO YOU, RUDY

Thank-you to everybody for making this a really great year for web standards. Have a wonderful holiday season.

Buy Andy Clarke's book Transcending CSS from Amazon.com

ABOUT THE AUTHOR

Andrew Clarke runs Stuff and Nonsense, a tiny web design company where they make fashionably flexible websites. Andrew's the author of Transcending CSS and Hardboiled Web Design and hosts the popular weekly podcast Unfinished Business where he discusses the business side of web, design and creative industries with his guests. He tweets as @malarkey.

16. Fast and Simple Usability Testing

Natalie Downe 24ways.org/200616

Everyone knows by now that they should test the usability of their applications, but still hardly anybody actually does it. In this article I'll share some tips I've picked up for doing usability tests quickly and effectively.

Relatively recent tools like Django and Ruby on Rails allow us to develop projects faster and to make significant changes later in the project timeline. Usability testing methods should now be adapted to fit this modern approach to development.

WHEN TO TEST

In an ideal world usability tests would be carried out frequently from an early stage of the project. Time and budget constraints lead this to be impractical; usability is often the first thing to get dropped from the project plan.

If you can only test at one stage in the project, whatever the size, the most valuable time is before your first public beta — leaving long enough to fix issues and not so late that you can't rethink your scope.

There are three main categories of usability test:

- Testing design mockups
- Testing a new working application
- Testing established applications

Each category requires a slightly different approach. For small modern web projects you are most likely to be testing a new working application. You will of course have already done functional tests so you won't be worried about the user breaking things. The main differences between the categories apply in how you word The Script.

Testing an established application is the most fun in my opinion. Humans are remarkably adaptable and rapidly develop coping strategies to work around usability issues in software they are forced to use. Uncovering these strategies may lead you to understand previously unspoken needs of your users. Often small changes to the application will have a dramatic affect on their everyday lives.

WHO TO TEST

When you have built a project to scratch your own itch, your intended audience will be people just like you. Test subjects in this case should be easy to find – friends, co-workers etc.

This is not always the case; your users may not be like you at all. When they are not, it's all the more important to run usability tests. Testing on friends, family and co-workers is better than not doing usability tests at all, but it can't be compared to testing on actual samples of your intended audience. People who would use the system will provide more genuine feedback and deeper insight.

Never let your test subjects put themselves in the shoes of your 'actual' users. For example, you should discourage comments like "Well, I would do this BUT if I was a bus driver I'd do that". Users are not qualified to put themselves in the position of others. Inaccurate data is often worse than no data.

Aim for five or six test subjects: any more and you probably won't learn anything new; any less and you're likely to be overwhelmed by issues stemming from people's individual personalities.

THE SCRIPT

The Script is a single side of A4 (or letter) paper, consisting of questions for your testers and reminders for yourself. Have a balance of task-based questions and expectation analysis. This helps maintain consistency across tests. Expectation analysis is more important for testing designs and new applications: “Where would you find X?”, “What would you expect to happen if you clicked on Y?”. In an established system users will probably know where these things are though it can still be illuminating to ask these questions though phrased slightly differently.

Task-based questions involve providing a task for the user to complete. If you are testing an established system it is a good idea to ask users to bring in tasks that they would normally perform. This is because the user will be more invested in the outcome of the task and will behave in a more realistic fashion. When designing tasks for new systems and designs ensure you only provide loose task details for the same reason. Don’t tell testers to enter “Chantelle”; have them use their own name instead. Avoid introducing bias with the way questions are phrased.

It’s a good idea to ask for users’ first impressions at the beginning of the test, especially when testing design mockups. “What are the main elements on the page?” or “What strikes you first?”.

You script should run for a maximum of 45 minutes. 30-35 minutes is better; after this you are likely to lose their attention. Tests on established systems can take longer as there is more to learn from them. When scheduling the test you will need to leave yourself 5 minutes between each one to collate your notes and prepare for the next. Be sure to run through the script beforehand.

Your script should be flexible. It is possible that during the test a trend will come to light that opens up whole new avenues of possible questioning. For example, during one initial test of an established system I noticed that the test subject had been printing off items from the application and placing them in a folder in date order (the system ordered alphabetically). I changed the script to ask future participants in that run, if they ever used external tools to help them with tasks within the system. This revealed a number of interesting issues that otherwise would not have been found.

RUNNING THE TESTS

Treat your test subjects like hedgehogs. Depending on your target audience they probably feel a little nervous and perhaps even scared of you. So make them a little nest out of straw, stroke their prickles and give them some cat food. Alternatively, reassure them that you are testing the system and that they can't give a wrong answer. Reward them with a doughnut or jam tart at the end. Try to ensure the test environment is relaxed and quiet, but also as close as possible to the situation where they would actually use the system.

Have your subjects talk out loud is very important as you can't read their minds, but it is a very unnatural process. To loosen up your subjects and get them talking in the way you want them to, try the Stapler Trick. Give them a stapler or similar item and ask them to open it, take the staples out, replace them, shut the stapler and staple some paper – talking all the time about what they see, what they expect to happen, what actually happens and how that matches up. Make them laugh at you.

Say how long the test will take up front, and tell your subject why you are doing it. After the test has been completed, conclude by thanking them for their time and assuring them that they were very useful. Then give them the sugary treat.

WHAT TO LOOK FOR

Primarily, you should look out for incidents where the user stops concentrating on her tasks and starts thinking about the tool and how she is going to use it. For example, when you are hammering in a nail you don't think about how to use a hammer; good software should be the same. Words like 'it' and 'the system' and are good indications that the test subject has stopped thinking about the task in hand. Note questioning words, especially where testers question their own judgement, “why can’t I find …”, “I expected to see …” etc. as this indicates that the work flow for the task may have broken down.

Also keep an eye on occasions where the user completely fails to do a task. They may need some prompting to unstick them, but you should be careful not to bias the test. These should be the highest priority issues for you to fix. If users recover from getting stuck, make a note of how they recovered. Prolonged periods of silence from the test subject may also require prompting as they should be talking all the time. Ask them what they are thinking or looking for but avoid words like ‘try’ (e.g. ‘what are you trying to do?’) as this implies that they are currently failing.

Be wary of users’ opinions on aesthetics and be prepared to bring them back to the script if they get side-tracked.

WRITING IT UP

Even if you are the only developer it’s important to summarise the key issues that emerged during testing: your notes won’t make much sense to you a week or so after the test.

If you are writing for other people, include a summary no longer than two pages; this can consist of a list or table of the issues including recommendations and their priorities.

Remember to anonymise the users in the report. In team situations, you may be surprised at how many people are interested in the results of the usability test even if it doesn't relate directly to something that they can fix.

TO CONCLUDE…

Some usability testing is better than none at all, even for small projects or those with strict deadlines. Make the most of the time and resources available. Choose your users carefully, make them comfortable, summarise your report and don't forget to leave a doughnut for yourself!

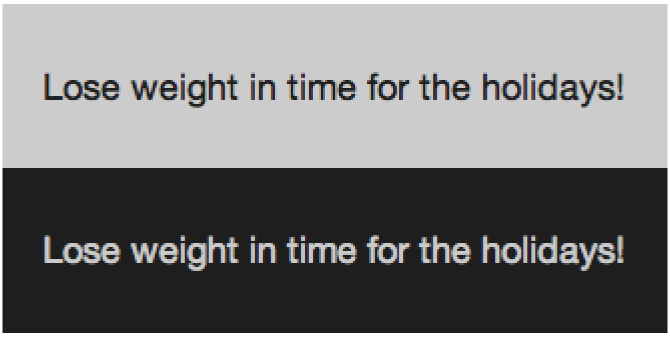
ABOUT THE AUTHOR

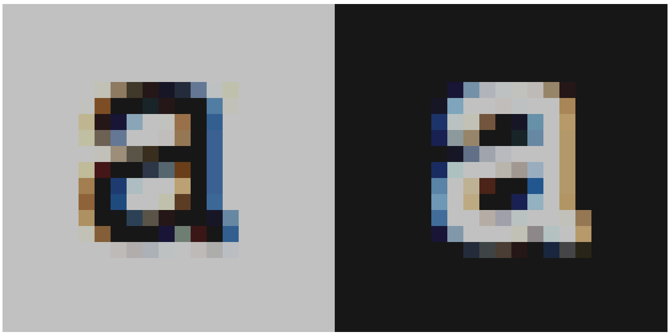
Natalie Downe is an excitable client-side web developer at Clearleft in Brighton, a perfectionist by nature and comes with the expertise and breadth of knowledge of a web agency background. Although front-end development and usability engineering are her first loves, Natalie still has fun dabbling with Python and poking the odd API. Natalie is also an experienced usability consultant and project manager.

17. Knockout Type - Thin Is Always In

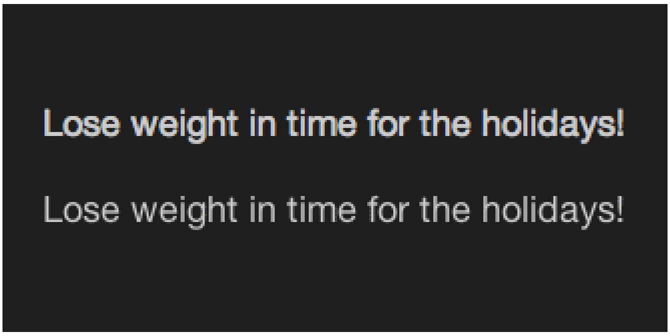
Shaun Inman 24ways.org/200617

OS X has gorgeous native anti-aliasing (although I will admit to missing 10px aliased Geneva — *sigh*). This is especially true for dark text on a light background. However, things can go awry when you start using light text on a dark background. Strokes thicken. Counters constrict. Letterforms fill out like seasonal snackers.


Lose weight in time for the holidays!
Lose weight in time for the holidays!

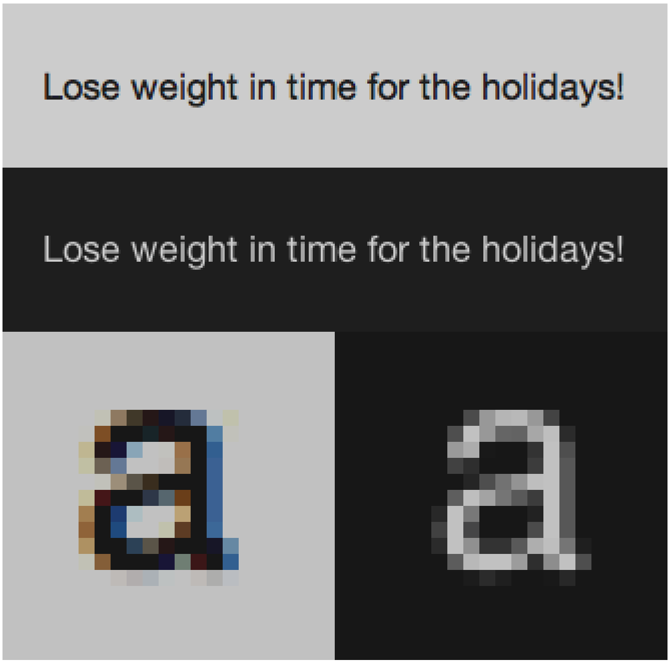


So how do we combat the fat? In Safari and other Webkit-based browsers we can use the CSS ‘text-shadow’ property. While trying to add a touch more contrast to the navigation on haveamint.com I noticed an interesting side-effect on the weight of the type.

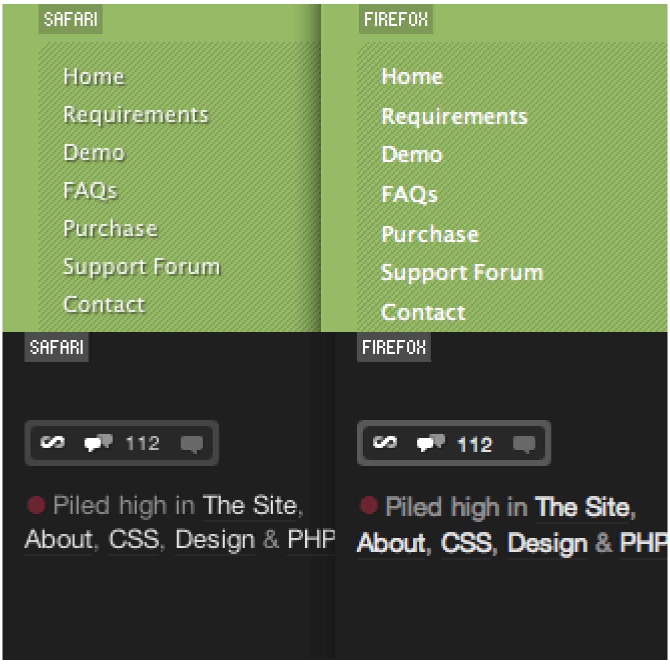

Lose weight in time for the holidays!
Lose weight in time for the holidays!

The second line in the example image above has the following style applied to it:

This creates an invisible drop-shadow. (Why is it invisible? The shadow is positioned directly behind the type (the first two zeros) and has no spread (the third zero). So the color, black, is completely eclipsed by the type it is supposed to be shadowing.)


Lose weight in time for the holidays!
Lose weight in time for the holidays!
a
a

Why applying an invisible drop-shadow effectively lightens the weight of the type is unclear. What is clear is that our light-on-dark text is now of a comparable weight to its dark-on-light counterpart.



You can see this trick in effect all over ShaunInman.com and in the navigation on haveamint.com and Subtraction.com. The HTML and CSS source code used to create the example images used in this article can be found here.

ABOUT THE AUTHOR

Shaun Inman designed and developed Mint, the curiously successful web site analytic tool. He passes the time (literally) tinkering on ShaunInman.com while nervously eyeing the dust gathering on Designologue.

18. Boost Your Hyperlink Power

Jeremy Keith 24ways.org/200618

There are HTML elements and attributes that we use every day. Headings, paragraphs, lists and images are the mainstay of every Web developer's toolbox. Perhaps the most common tool of all is the anchor. The humble `a` element is what joins documents together to create the gloriously chaotic collection we call the World Wide Web.

ANATOMY OF AN ANCHOR

The power of the anchor element lies in the `href` attribute, short for *h*ypertext *ref*erence. This creates a one-way link to another resource, usually another page on the Web:

```
<a href="http://allinthehead.com/">
```

The `href` attribute sits in the opening `a` tag and some descriptive text sits between the opening and closing tags:

```
<a href="http://allinthehead.com/">Drew McLellan</a>
```

"Whoop-dee-freakin'-doo," I hear you say, "this is pretty basic stuff" – and you're quite right. But there's more to the anchor element than just the `href` attribute.

THE THEORY OF RELATIVITY

You might be familiar with the `rel` attribute from the `link` element. I bet you've got something like this in the `head` of your documents:

```
<link rel="stylesheet" type="text/css" media="screen"
href="styles.css" />
```

The `rel` attribute describes the *rel*ationship between the linked document and the current document. In this case, the value of `rel` is "stylesheet". This means that the linked document is the stylesheet for the current document: that's its relationship.

Here's another common use of `rel`:

```
<link rel="alternate" type="application/rss+xml"
title="my RSS feed" href="index.xml" />
```

This describes the relationship of the linked file – an RSS feed – as "alternate": an alternate view of the current document.

Both of those examples use the `link` element but you are free to use the `rel` attribute in regular hyperlinks. Suppose you're linking to your RSS feed in the `body` of your page:

```
Subscribe to <a href="index.xml">my RSS feed</a>.
```

You can add extra information to this anchor using the `rel` attribute:

```
Subscribe to <a href="index.xml" rel="alternate"
type="application/rss+xml">my RSS feed</a>.
```

There's no prescribed list of values for the `rel` attribute so you can use whatever you decide is semantically meaningful. Let's say you've got a complex e-commerce application that includes a link to a help file. You can explicitly declare the relationship of the linked file as being "help":

```
<a href="help.html" rel="help">need help?</a>
```

ELEMENTAL MICROFORMATS

Although it's completely up to you what values you use for the `rel` attribute, some consensus is emerging in the form of microformats. Some of the simplest microformats make good use of `rel`. For example, if you are linking to a license that covers the current document, use the rel-license microformat:

```
Licensed under a <a href="http://creativecommons.org/
licenses/by/2.0/" rel="license">Creative Commons
attribution license</a>
```

That describes the relationship of the linked document as “license.”

The rel-tag microformat goes a little further. It uses `rel` to describe the final part of the URL of the linked file as a “tag” for the current document:

```
Learn more about <a href="http://en.wikipedia.org/wiki/
Microformats" rel="tag">semantic markup</a>
```

This states that the current document is being tagged with the value “Microformats.”

XFN, which stands for XHTML Friends Network, is a way of describing relationships between people:

```
<a href="http://allinthehead.com/" rel="friend">Drew
McLellan</a>
```

This microformat makes use of a very powerful property of the `rel` attribute. Like the `class` attribute, `rel` can take multiple values, separated by spaces:

```
<a href="http://allinthehead.com/" rel="friend met
colleague">Drew McLellan</a>
```

Here I’m describing Drew as being a friend, someone I’ve met, and a colleague (because we’re both Web monkies).

YOU SAY YOU WANT A REVOLUTION

While `rel` describes the relationship of the linked resource to the current document, the `rev` attribute describes the *rev*erse relationship: it describes the relationship of the current document to the linked resource. Here's an example of a link that might appear on `help.html`:

```
<a href="shoppingcart.html" rev="help">continue
shopping</a>
```

The `rev` attribute declares that the current document is "help" for the linked file.

The vote-links microformat makes use of the `rev` attribute to allow you to qualify your links. By using the value "vote-for" you can describe your document as being an endorsement of the linked resource:

```
I agree with <a href="http://richarddawkins.net/home"
rev="vote-for">Richard Dawkins</a>.
```

There's a corresponding `vote-against` value. This means that you can link to a document but explicitly state that you don't agree with it.

```
I agree with <a href="http://richarddawkins.net/home"
rev="vote-for">Richard Dawkins</a>
about those <a href="http://www.icr.org/"
rev="vote-against">creationists</a>.
```

Of course there's nothing to stop you using both `rel` and `rev` on the same hyperlink:

```
<a href="http://richarddawkins.net/home" rev="vote-for"
rel="muse">Richard Dawkins</a>
```

THE WISDOM OF CROWDS

The simplicity of `rel` and `rev` belies their power. They allow you to easily add extra semantic richness to your hyperlinks. This creates a bounty that can be harvested by search engines, aggregators and browsers. Make it your New Year's resolution to make friends with these attributes and extend the power of hypertext.

ABOUT THE AUTHOR

Jeremy Keith is an Irish web developer living in Brighton, England where he works with the web consultancy firm Clearleft. He wrote the books, DOM Scripting, Bulletproof Ajax, and most recently HTML5 For Web Designers.

His latest project is Huffduffer, a service for creating podcasts of found sounds. When he’s not making websites, Jeremy plays bouzouki in the band Salter Cane. His loony bun is fine benny lava.

19. The Mobile Web, Simplified

Cameron Moll

24ways.org/200619

A note from the editors: although eye-opening in 2006, this article is no longer relevant to today's mobile web.

Considering a foray into mobile web development? Following are four things you need to know before making the leap.

1. 4 billion mobile subscribers expected by 2010

Fancy that. Coupled with the UN prediction of 6.8 billion humans by 2010, 4 billion mobile subscribers (source) is an astounding 59% of the planet. Just how many of those subscribers will have data plans and web-enabled phones is still in question, but inevitably this all means one thing for you and me: A ton of potential eyes to view our web content on a mobile device.

2. Context is king

Your content is of little value to users if it ignores the context in which it is viewed. Consider how you access data on your mobile device. You might be holding a bottle of water or gripping a handle on the subway/tube. You're probably seeking specific data such as directions or show times, rather than the plethora of data at your disposal via a desktop PC.

The mobile web, a phrase often used to indicate "accessing the web on a mobile device", is very much a context-, content-, and component-specific environment. Expressed in terms of your potential target audience, access to web content on a mobile device is largely influenced by surrounding circumstances and conditions, information relevant to being mobile, and the feature set of the device being used. Ask yourself, What is relevant to my users and the tasks, problems, and needs they may encounter while being mobile? Answer that question and you'll be off to a great start.

3. WAP 2.0 is an XHTML environment

In a nutshell, here are a few fundamental tenets of mobile internet technology:

1. Wireless Application Protocol (WAP) is the protocol for enabling mobile access to internet content.

2. Wireless Markup Language (WML) was the language of choice for WAP 1.0.
3. Nearly all devices sold today are WAP 2.0 devices.
4. With the introduction of WAP 2.0, XHTML Mobile Profile (XHTML-MP) became the preferred markup language.
5. XHTML-MP will be familiar to anyone experienced with XHTML Transitional or Strict.

Summary? The mobile web is rapidly becoming an XHTML environment, and thus you and I can apply our existing "desktop web" skills to understand how to develop content for it. With WML on the decline, the learning curve is much smaller today than it was several years ago. I'm generalizing things gratuitously, but the point remains: Get off yo' lazy butt and begin to take mobile seriously.

I'll even pass you a few tips for getting started. First, the DOCTYPE for XHTML-MP is as follows:

```
<!DOCTYPE html PUBLIC "-//WAPFORUM//DTD XHTML Mobile
1.0//EN"
"http://www.openmobilealliance.org/tech/DTD/
xhtml-mobile10.dtd">
```

As for MIME type, Open Mobile Alliance (OMA) specifies using the MIME type `application/vnd.wap.xhtml+xml`, but ultimately you need to ensure the server delivering your mobile content is configured properly for the MIME type you choose to use, as there are other options (see Setting up WAP Servers).

Once you've made it to the `body`, the XHTML-MP markup is not unlike what you're already used to. A few resources worth skimming:

- Developers Home XHTML-MP Tutorial – An impressively replete resource for all things XHTML-MP
- XHTML-MP Tags List – A complete list of XHTML-MP elements and accompanying attributes

And last but certainly not least, CSS. There exists WAP CSS, which is essentially a subset of CSS2 with WAP-specific extensions. For all intents and purposes, much of the CSS you're already comfortable using will be transferrable to mobile. As for including CSS in your pages, your options are the same as for desktop sites: external, embedded, and inline. Some experts will argue embedded or inline over external in favor of reducing the number of HTTP connections per page request, yet many popular mobilized sites and apps employ external linking without issue.

Stocking stuffers: Flickr Mobile, Fandango Mobile, and Popurls Mobile. A few sites with whom you can do the View Source song and dance for further study.

4. “Cell phone” is so DynaTAC

If you’re a U.S. resident, listen up: You must rid your vocabulary of the term “cell phone”. We’re one of the few economies on the planet to refer to a mobile phone accordingly. If you care to find yourself in any of the worthwhile mobile development circles, begin using terms more widely accepted: “mobile” or “mobile phone” or “handset” or “handy”. If you’re not sure which, go for “mobile”. Such as, “Yo dog, check out my new mobile.”

More importantly, however, is overcoming the mentality that access to the mobile web can be done only with a phone. Instead, “device” encourages us to think phone, handheld computer, watch, Nintendo DS, car, you name it.

Simple enough?

ABOUT THE AUTHOR

Cameron Moll is the founder of Authentic Jobs, maker of Structures in Type, and author of *Mobile Web Design* (2007). He resides in Sarasota, Florida with his wife and five sons. Find him on Twitter at @cameronmoll.

Cameron is looking to share the principles of Cohesive UX in 2015. Please get in touch if you'd like to have him speak at your conference.

20. Intricate Fluid Layouts in Three Easy Steps

Nate Koechley 24ways.org/200620

The Year of the Script may have drawn attention away from CSS but building fluid, multi-column, cross-browser CSS layouts can still be as unpleasant as a lump of coal. Read on for a worry-free approach in three quick steps.

The layout system I developed, YUI Grids CSS, has three components. They can be used together as we'll see, or independently.

THE THREE EASY STEPS

1. Choose fluid or fixed layout, and choose the width (in percents or pixels) of the page.
2. Choose the size, orientation, and source-order of the main and secondary blocks of content.

3. Choose the number of columns and how they distribute (for example 50%-50% or 25%-75%), using stackable and nestable grid structures.

THE SETUP

There are two prerequisites: We need to normalize the size of an `em` and opt into the browser rendering engine's Strict Mode.

Ems are a superior unit of measure for our case because they represent the current font size and grow as the user increases their font size setting. This flexibility—the container growing with the user's wishes—means larger text doesn't get crammed into an unresponsive container. We'll use YUI Fonts CSS to set the base size because it provides consistent-yet-adaptive font-sizes while preserving user control.

The second prerequisite is to opt into Strict Mode (more info on rendering modes) by declaring a Doctype complete with URI. You can choose XHTML or HTML, and Transitional or Strict. I prefer HTML 4.01 Strict, which looks like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

INCLUDING THE CSS

A single small CSS file powers a nearly-infinite number of layouts thanks to a recursive system and the interplay between the three distinct components. You *could* prune to a particular layout's specific needs, but why bother when the complete file weighs scarcely 1.8kb uncompressed? Compressed, YUI Fonts and YUI Grids combine for a miniscule 0.9kb over the wire.

You could save an HTTP request by concatenating the two CSS files, or by adding their contents to your own CSS, but I'll keep them separate for now:

```
<link href="fonts.css" rel="stylesheet" type="text/css">
<link href="grids.css" rel="stylesheet" type="text/css">
```

Example: The Setup

Now we're ready to build some layouts.

STEP 1: CHOOSE FLUID OR FIXED LAYOUT

Choose between preset widths of 750px, 950px, and 100% by giving a document-wrapping div an ID of `doc`, `doc2`, or `doc3`. These options cover most use cases, but it's easy to define a custom fixed width.

The fluid 100% grid (`doc3`) is what I've been using almost exclusively since it was introduced in the last YUI released.

```
<body>
  <div id="doc3"></div>
</body>
```

All pages are centered within the viewport, and grow with font size. The 100% width page (`doc3`) preserves 10px of breathing room via left and right margins. If you prefer your content flush to the viewport, just add `doc3 {margin:auto}` to your CSS.

Regardless of what you choose in the other two steps, you can always toggle between these widths and behaviors by simply swapping the ID value. It's really that simple.

Example: 100% fluid layout

STEP 2: CHOOSE A TEMPLATE PRESET

This is perhaps the most frequently omitted step (they're all optional), but I use it nearly every time. In a source-order-independent way (good for accessibility and SEO), "Template Presets" provide commonly used template widths compatible with ad-unit dimension standards defined by the Interactive Advertising Bureau, an industry association.

Choose between the six Template Presets (`.yui-t1` through `.yui-t6`) by setting the class value on the document-wrapping `div` established in Step 1. Most frequently I use `yui-t3`, which puts the narrow secondary block on the left and makes it 300px wide.

```
<body>
  <div id="doc3" class="yui-t3"></div>
</body>
```

The Template Presets control two "blocks" of content, which are defined by two `div`s, each with `yui-b` ("b" for "block") class values. Template Presets describe the width and orientation of the *secondary* block; the *main* block will take up the rest of the space.

```
<body>
  <div id="doc3" class="yui-t3">
    <div class="yui-b"></div>
    <div class="yui-b"></div>
  </div>
</body>
```

Use a wrapping `div` with an ID of `yui-main` to structurally indicate which block is the main block. This wrapper—not the source order—identifies the main block.

```
<body>
  <div id="doc3" class="yui-t3">
    <div id="yui-main">
      <div class="yui-b"></div>
    </div>
    <div class="yui-b"></div>
  </div>
</body>
```

Example: Main and secondary blocks sized and oriented with `.yui-t3` Template Preset

Again, regardless of what values you choose in the other steps, you can always toggle between these Template Presets by toggling the class value of your document-wrapping `div`. It's really that simple.

STEP 3: NEST AND STACK GRID STRUCTURES.

The bulk of the power of the system is in this third step. The key is that **columns are built by parents telling children how to behave**. By default, two children each consume half of their parent's area. Put two units inside a grid structure, and they will sit side-by-side, and they will each take up half the space. Nest this structure and two columns become four. Stack them for rows of columns.

An Even Number of Columns

The default behavior creates two evenly-distributed columns. It's easy. Define one parent grid with `.yui-g` ("g" for grid) and two child units with `.yui-u` ("u" for unit). The code looks like this:

```
<div class="yui-g">
  <div class="yui-u first"></div>
  <div class="yui-u"></div>
</div>
```

Be sure to indicate the “`first`“ unit because the :first-child pseudo-class selector isn’t supported across all A-grade browsers. It’s unfortunate we need to add this, but luckily it’s not out of place in the markup layer since it is structural information.

Example: Two evenly-distributed columns in the main content block

An Odd Number of Columns

The default system does not work for an odd number of columns without using the included “Special Grids” classes. To create three evenly distributed columns, use the “`yui-gb`“ Special Grid:

```
<div class="yui-gb">
  <div class="yui-u first"></div>
  <div class="yui-u"></div>
  <div class="yui-u"></div>
</div>
```

Example: Three evenly distributed columns in the main content block

Uneven Column Distribution

Special Grids are also used for unevenly distributed column widths. For example, `.yui-ge` tells the first unit (column) to take up 75% of the parent's space and the other unit to take just 25%.

```
<div class="yui-ge">
  <div class="yui-u first"></div>
  <div class="yui-u"></div>
</div>
```

Example: Two columns in the main content block split 75%-25%

PUTTING IT ALL TOGETHER

Start with a full-width fluid page (`div#doc3`). Make the secondary block 180px wide on the right (`div.yui-t4`). Create three *rows* of columns: Three evenly distributed columns in the first row (`div.yui-gb`), two uneven columns (66%-33%) in the second row (`div.yui-gc`), and two evenly distributed columns in the thrid row.

```
<body>
  <!-- choose fluid page and Template Preset -->
  <div id="doc3" class="yui-t4">
    <!-- main content block -->
    <div id="yui-main">
      <div class="yui-b">
        <!-- stacked grid structure, Special Grid "b" -->
        <div class="yui-gb">
```

```
          <div class="yui-u first"></div>
          <div class="yui-u"></div>
          <div class="yui-u"></div>
        </div>
        <!-- stacked grid structure, Special Grid "c" -->
        <div class="yui-gc">
          <div class="yui-u first"></div>
          <div class="yui-u"></div>
        </div>
        <!-- stacked grid structure -->
        <div class="yui-g">
          <div class="yui-u first"></div>
          <div class="yui-u"></div>
        </div>
      </div>
    </div>
    <!-- secondary content block -->
    <div class="yui-b"></div>
  </div>
</body>
```

Example: A complex layout.

Wasn’t that easy? Now that you know the three “levers” of YUI Grids CSS, you’ll be creating headache-free fluid layouts faster than you can say “Peace on Earth”.

ABOUT THE AUTHOR

Nate Koechley is a Yahoo! frontend engineer and designer based in San Francisco's Mission district. When he's not helping design and build the open-source Yahoo! User Interface (YUI) Library he edits the YUIBlog, promotes accessibility, defines Yahoo! browser support policies, writes occasionally at his personal blog, and presents at conferences around the globe.

Photo: Dave Bullock

21. A Scripting Carol

Derek Featherstone 24ways.org/200621

We all know the stories of the Ghost of Scripting Past – a time when the web was young and littered with nefarious scripting, designed to bestow ultimate control upon the developer, to pollute markup with event handler after event handler, and to entrench advertising in the minds of all that gazed upon her.

And so it came to be that JavaScript became a dirty word, thrown out of solutions by many a Scrooge without regard to the enhancements that JavaScript could bring to any web page. JavaScript, as it was, was dead as a door-nail.

With the arrival of our core philosophy that all standardistas hold to be true: "separate your concerns – content, presentation and behaviour," we are in a new era of responsible development the Web Standards Way™. Or are we? Have we learned from the Ghosts of Scripting Past? Or are we now faced with new problems that come with new ways of implementing our solutions?

THE GHOST OF SCRIPTING PAST

If the Ghost of Scripting Past were with us it would probably say:

> You must remember your roots and where you came from, and realize the misguided nature of your early attempts for control. That person you see down there, is real and they are the reason you exist in the first place… without them, you are nothing.

In many ways we've moved beyond the era of control and we do take into account the user, or at least much more so than we used to. Sadly – there is one advantage that old school inline event handlers had where we assigned and reassigned CSS style property values on the fly – we knew that if JavaScript wasn't supported, the styles wouldn't be added because we ended up doing them at the same time.

If anything, we need to have learned from the past that just because it works for us doesn't mean it is going to work for anyone else – we need to test more scenarios than ever to observe the multitude of browsing arrangements we'll observe: CSS on with JavaScript off, CSS off/overridden with JavaScript on, both on, both off/not supported. It is a situation that is ripe for conflict.

This may shock some of you, but there was a time when testing was actually easier: back in the day when Netscape 4 was king. Yes, that's right. I actually kind of enjoyed Netscape 4 (hear me out, please). With NS4's CSS implementation known as JavaScript Style Sheets, you knew that if JavaScript was off the styles were off too.

THE GHOST OF SCRIPTING PRESENT

With current best practice – we keep our CSS and JavaScript separate from each other. So what happens when some of our fancy, unobtrusive DOM Scripting doesn't play nicely with our wonderfully defined style rules?

Lets look at one example of a collapsing and expanding menu to illustrate where we are now:

Simple Collapsing/Expanding Menu Example

We're using some simple JavaScript (I'm using jquery in this case) to toggle between a CSS state for expanded and not expanded:

JavaScript

```
$(document).ready(function(){
  TWOFOURWAYS.enableTree();
});
var TWOFOURWAYS = new Object();
TWOFOURWAYS.enableTree = function ()
{
  $("ul li a").toggle(function(){
    $(this.parentNode).addClass("expanded");
  }, function() {
    $(this.parentNode).removeClass("expanded");
  });
  return false;
}
```

CSS

```
ul li ul {
  display: none;
}
ul li.expanded ul {
  display: block;
}
```

At this point we've separated our presentation from our content and behaviour, and all is well, right?

Not quite.

Here's where I typically see failures in the assessment work that I do on web sites and applications (Yes, call me Scrooge – I don't care!). We know our page needs to work with or without scripting, and we know it needs to work with or without CSS. All too often the testing scenarios don’t take into account combinations.

TESTING IT OUT

So what happens when we test this? Make sure you test with:

- CSS off
- JavaScript off

Use the simple example again.

With CSS off, we revert to a simple nested list of links and all functionality is maintained. With JavaScript off, however, we run into a problem – we have now removed the ability to expand the menu using the JavaScript triggered CSS change.

Hopefully you see the problem – we have a JavaScript and CSS dependency that is critical to the functionality of the page. Unobtrusive scripting and binary on/off tests aren’t enough. We need more.

This Ghost of Scripting Present sighting is seen all too often.

Lets examine the JavaScript off scenario a bit further – if we require JavaScript to expand/show the branch of the tree we should use JavaScript to hide them in the first place. That way we guarantee functionality in all scenarios, and have achieved our baseline level of interoperability.

To revise this then, we'll start with the sub items expanded, use JavaScript to collapse them, and then use the same JavaScript to expand them.

HTML

```
<ul>
  <li><a href="#">Main Item</a>
    <ul class="collapseme">
      <li><a href="#">Sub item 1</a></li>
      <li><a href="#">Sub item 2</a></li>
      <li><a href="#">Sub item 3</a></li>
    </ul>
  </li>
</ul>
```

CSS

```
/* initial style is expanded */
ul li ul.collapseme {
  display: block;
}
```

JavaScript

```
// remove the class collapseme after the page loads
$("ul ul.collapseme").removeClass("collapseme");
```

And there you have it – a revised example with better interoperability.

This isn't rocket surgery by any means. It is a simple solution to a ghostly problem that is too easily overlooked (and often is).

THE GHOST OF SCRIPTING FUTURE

Well, I'm not so sure about this one, but I'm guessing that in a few years' time, we'll all have seen a few more apparitions and have a few more tales to tell. And hopefully we'll be able to share them on 24 ways.

Thanks to Drew for the invitation to contribute and thanks to everyone else out there for making this a great (and haunting) year on the web!

ABOUT THE AUTHOR

Derek Featherstone is a web developer and experienced accessibility consultant based in Ottawa, Canada where he runs Further Ahead. He serves as the Lead for the WaSP Accessibility Task Force. He is insane and thinks that somehow he'll manage to find time to train for an IronMan triathlon amidst work and family life with wife and three children. Insane.

22. Photographic Palettes

Dave Shea 24ways.org/200622

How many times have you seen a colour combination that just worked, a match so perfect that it just seems obvious?

Now, how many times do you come up with those in your own work? A perfect palette looks easy when it's done right, but it's often maddeningly difficult and time-consuming to accomplish.

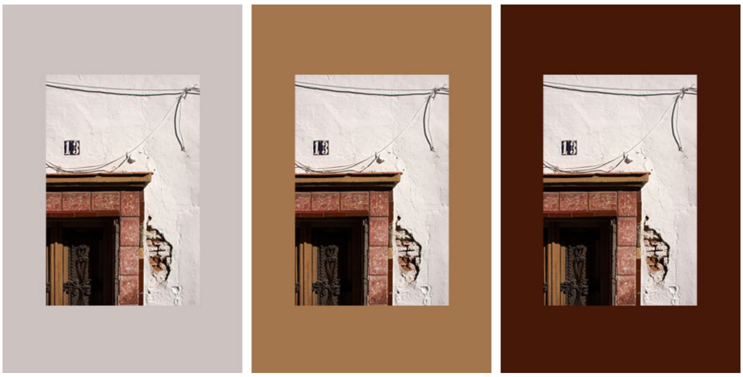
Choosing effective colour schemes will always be more art than science, but there are things you can do that will make coming up with that oh-so-smooth palette just a little a bit easier. A simple trick that can lead to incredibly gratifying results lies in finding a strong photograph and sampling out particularly harmonious colours.

PHOTO SELECTION

Not all photos are created equal. You certainly want to start with imagery that fits the eventual tone you're attempting to create. A well-lit photo of flowers might lead to a poor colour scheme for a funeral parlour's web site, for example. It's worth thinking about what you're trying to say in advance, and finding a photo that lends itself to your message.

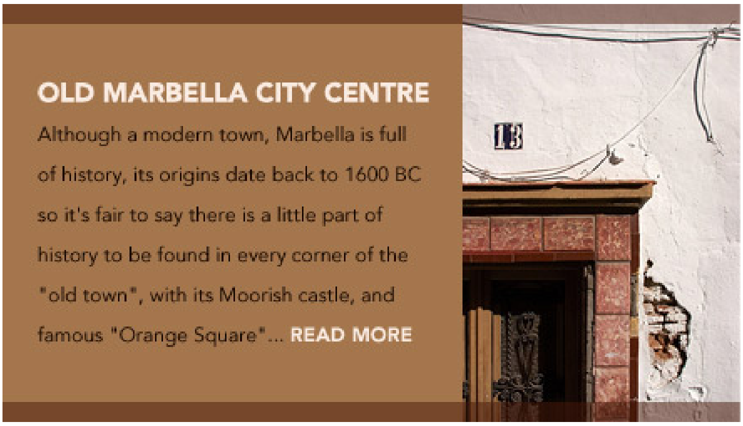
As a general rule of thumb, photos that have a lot of neutral or de-saturated tones with one or two strong colours make for the best palette; bright and multi-coloured photos are harder to derive pleasing results from. Let's start with a relatively neutral image.

SAMPLING



In the above example, I've surrounded the photo with three different background colours directly sampled from the photo itself. Moving from left to right, you can see how each of the sampled colours is from an area of increasingly smaller coverage within the photo, and yet there's still a strong harmony between the photo and the background image. I don't really need to pick the big obvious colours from the photo to create that match, I can easily concentrate on more interesting colours that might work better for what I intend.

Using a similar palette, let's apply those colour choices to a more interesting layout:

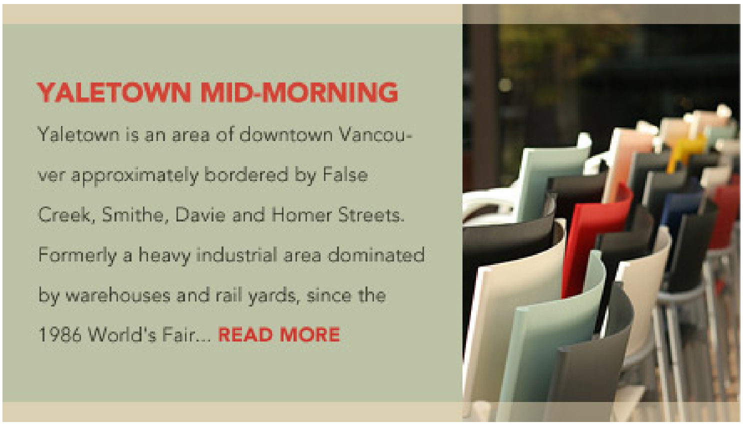


OLD MARBELLA CITY CENTRE

Although a modern town, Marbella is full of history, its origins date back to 1600 BC so it's fair to say there is a little part of history to be found in every corner of the "old town", with its Moorish castle, and famous "Orange Square"... READ MORE

In this mini-layout, I've re-used the same tan colour from the previous middle image as a background, and sampled out a nicely matching colour for the top and bottom overlays, as well as the two different text colours. Because these colours all fall within a narrow range, the overall balance is harmonious.

What if I want to try something a little more daring? I have a photo of stacked chairs of all different colours, and I'd like to use a few more of those. No problem, provided I watch my colour contrast:



YALETOWN MID-MORNING

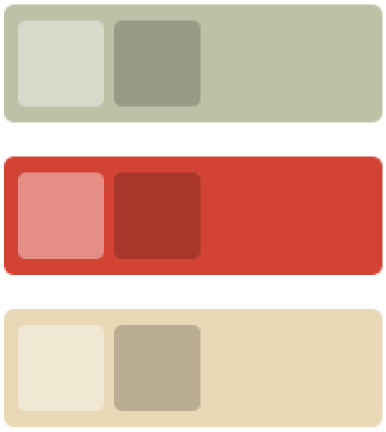
Yaletown is an area of downtown Vancouver approximately bordered by False Creek, Smithe, Davie and Homer Streets. Formerly a heavy industrial area dominated by warehouses and rail yards, since the 1986 World's Fair... READ MORE

Though it uses varying shades of red, green, and yellow, this palette actually works because the values are even, and the colours muted. Placing red on top of green is usually a hideous combination of death, but if the green is drab enough and the red contrasts well enough, the result can actually be quite pleasing. I've chosen red as my loudest colour in this palette, and left green and yellow to play the quiet supporting roles.

Obviously, there are no hard and fast rules here. You might not want to sample absolutely **every** colour in your scheme from a photo. There are times where you'll need a variation that's just a little bit lighter, or a blue that's not in the photo. You might decide to start from a photo base and tweak, or add in colours of your own. That's okay too.

TONAL VARIATIONS

I'll leave you with a final trick I've been using lately, a way to bring a bit more of a formula into the equation and save some steps.



Starting with the same base palette I sampled from the chairs, in the above image I've added a pair of overlaying squares that produce tonal variations of each primary. The lighter variation is simply a solid white square set to 40% opacity, the darker one is a black square at 20%. That gives me a highlight and shadow for each colour, which would prove handy if I had to adapt this colour scheme to a larger layout.

I could add a few more squares of varying opacities, or adjust the layer blending modes for different effects, but as this looks like a great place to end, I'll leave that up to your experimental whims. Happy colouring!

ABOUT THE AUTHOR

Dave Shea is the Founder of Bright Creative, and co-organizer of Web Directions North. He blogs sometimes at Mezzoblue and Flickrs a bit more often than that. Oh, and there's other stuff too.

23. Cheating Color

Jason Santa Maria 24ways.org/200623

Have you ever been strapped to use specific colors outlined in a branding guide? Felt restricted because those colors ended up being too light or dark for the way you want to use them?

Here's the solution: throw out your brand guide.

gasp!

OK, don't throw it out. Just put it in a drawer for a few minutes.

BRANDING GUIDES BE DAMNED

When dealing with color on screen, it's easy to get caught up in literal values from hex colors, you can cheat colors ever so slightly to achieve the right optical value. This is especially prevalent when trying to bring a company's identity colors to a screen design. Because the most important idea behind a brand guide is to help a company maintain the visual integrity of their business, consider hex numbers to be guidelines rather than law. Once you are familiar enough with the colors your company uses, you can start to flex them a bit, and take a few liberties.

This is a quick method for cheating to get the color you really want. With a little sleight of design, we can swap a color that might be part of your identity guidelines, with one that works better optically, and no one will be the wiser!

COLOR IS A WILY BEAST

This might be hard: You might have to break out of the idea that a color can only be made using one method. Color is fluid. It interacts and changes based on its surroundings. Some colors can appear lighter or darker based on what color they appear on or next to. The RGB gamut is additive color, and as such, has a tendency to push contrast in the direction that objects may already be leaning—increasing the contrast of light colors on dark colors and decreasing the contrast of light on light. Obviously, because we are talking about monitors here, these aren't hard and fast rules.

CHEAT AND FEEL GOOD ABOUT IT

On a light background, when you have a large element of a light color, a small element of the *same color* will appear lighter.

Enter our fake company: Double Dagger. They manufacture footnotes. Take a look at Fig. 1 below. The logo (Double Dagger), rule, and small text are all `#6699CC`. Because the logo so large, we get a good sense of the light blue color. Unfortunately, the rule and small text beneath it feel much lighter because we can't create enough contrast with such small shapes in that color.

Now take a look at Fig. 2. Our logo is still `#6699CC`, but now the rule and smaller text have been cheated to `#4477BB`, effectively giving us the same optical color that we used in the logo. You will find that we get a better sense of the light blue, and the added benefit of more contrast for our text. Doesn't that feel good?

FIG. 1

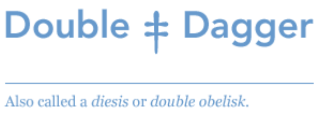
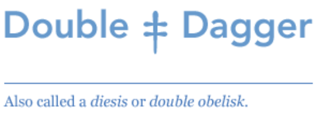


FIG. 2



Conversely, when you have a large element of a dark color, a small element of the *same color* will appear darker.

Let's look at Fig. 3 below. Double Dagger has decided to change its identity colors from blue to red. In Fig. 3, our logo, rule, and small text are all `#330000`, a very dark red. If you look at the rule and small text below the logo, you will notice that they seem dark enough to be confused with black. The dark red can't be sustained by the smaller shapes. Now let's look at Fig. 4. The logo is still `#33000`, but we've now cheated the rule and smaller text to `#550000`. This gives us a better sense of a red, but preserves the dark and moody direction the company has taken.

FIG. 3

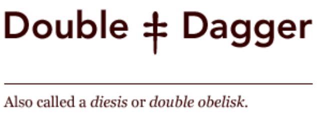
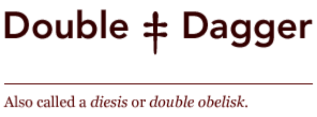


FIG. 4



But we've only touched on color against a white background. For colors against a darker background, you may find lighter colors work fine, but darker colors need to be cheated a bit to the lighter side in order to reach a good optical equivalent. Take a look below at Fig. 5 and Fig. 6. Both use the same exact corresponding colors as Fig. 1 and Fig. 2 above, but now they are set against a dark background. Where the blue used in Fig. 1 above was too light for the smaller elements, we find it is just right for them in Fig. 5, and the darker blue we used in Fig. 2 has now proven too dark for a dark background, as evidenced in Fig. 6.

FIG. 5

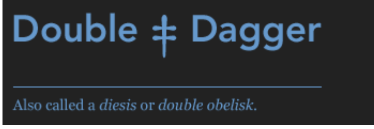
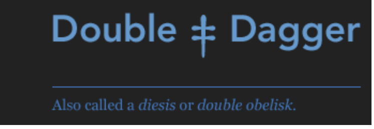


FIG. 6



Your mileage may vary, and this may not be applicable in all situations, but consider it to be just another tool on your utility belt for dealing with color problems.

ABOUT THE AUTHOR

Jason Santa Maria is a graphic designer from sunny Brooklyn, NY. He currently works as Creative Director for Happy Cog Studios, a web design consultancy, and A List Apart, an online magazine for people who make websites. He maintains a personal site where discussion of design, film, and sock monkeys can often be observed. His work has garnered him awards and pleasantries ranging from firm handshakes to forceful handshakes with a little hitting. Ever the design obsessif, Jason is known to take drunken arguments to fisticuffs over such frivolities as kerning and white space.

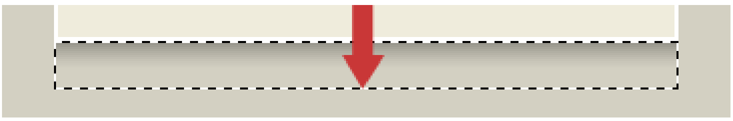
24. Gravity-Defying Page Corners

Dan Cederholm 24ways.org/200624

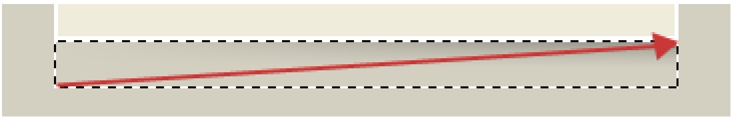
While working on Stikkit, a "page curl"
came to be.
Not being as crafty as Veerle, you see.
I fired up Photoshop to see what could be.
"*Another copy is running on the network*"
… oopsie.

With license issues sorted out and a concept in mind
I set out to create something flexible and refined.
One background image and code that is sure to be lean.
A simple solution for lazy people like me.

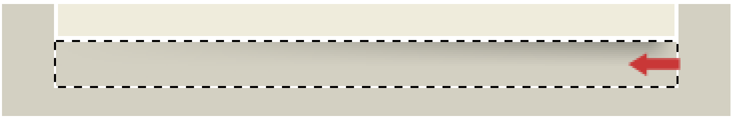
The curl I'll be showing isn't a curl at all.
It's simply a gradient that's 18 pixels tall.
With a fade to the left that's diagonally aligned
and a small fade on the right that keeps the illusion
defined.



Create a selection with the marquee tool (keeping in mind a reasonable minimum width) and drag a gradient (black to transparent) from top to bottom.



Now drag a gradient (the background color of the page to transparent) from the bottom left corner to the top right corner.

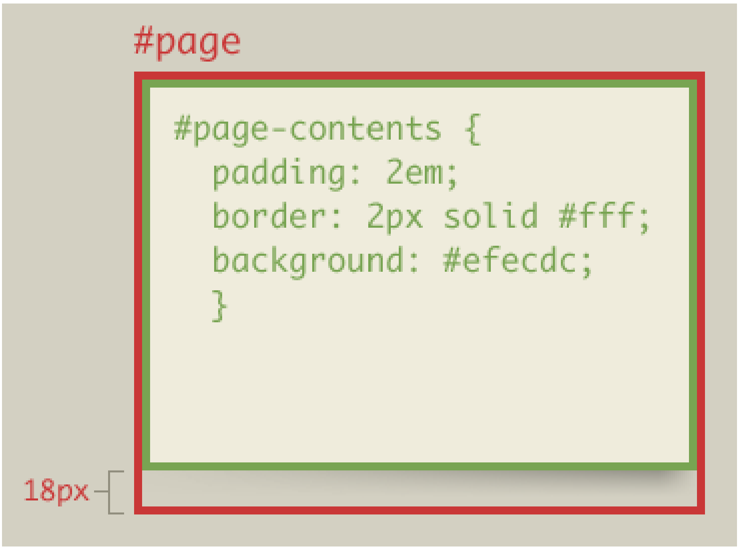


Finally, drag another gradient from the right edge towards the center, about 20 pixels or so.

But the top is flat and can be positioned precisely
just under the bottom right edge very nicely.
And there it will sit, never ever to be busted
by varying sizes of text when adjusted.

```
<div id="page">
  <div id="page-contents">
    <h2>Gravity-Defying!</h2>
    <p>Lorem ipsum dolor ...</p>
  </div>
</div>
```

Let’s dive into code and in the markup you’ll see
“is that an extra div?” … please don’t kill me?
The #page div sets the width and bottom padding
whose height is equal to the shadow we’re adding.



```
#page-contents {
  padding: 2em;
  border: 2px solid #fff;
  background: #efecdc;
  }
```

The #page-contents div will set padding in ems
to scale with the text size the user intends.
The background color will be added here too
but not overlapping the shadow where #page’s padding
makes room.

A simple technique that you may find amusing
is to substitute a PNG for the GIF I was using.
For that would be crafty and future-proof, too.
The page curl could sit on *any* background hue.

I hope you've enjoyed this easy little trick.
It's hardly earth-shattering, and arguably slick.
But it could come in handy, you just never know.
Happy Holidays! And pleasant dreams of web three point oh.

ABOUT THE AUTHOR

Dan Cederholm is a web designer and author based in Salem, Massachusetts. He's the founder of SimpleBits, a tiny web design studio. He's writes and speaks (in non-poetic form) about interface design during the day, and plays the ukulele and drinks wine at night.

Photo: Scott Beale / Laughing Squid