CSS Mastery

Advanced Web Standards Solutions

Andy Budd
with Cameron Moll
and Simon Collison

CSS Mastery:
Advanced Web Standards Solutions

Copyright © 2006 by Andy Budd, Cameron Moll, and Simon Collison

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-614-2
ISBN-10 (pbk): 1-59059-614-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit www.apress.com.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is freely available to readers at www.friendsofed.com in the Downloads section.

Product numbers for the images used in Tuscany Luxury Resorts are as follows:
FAN1003579, FAN1003613, FAN1006983, and DVP0703035.

Credits

Lead Editor
Chris Mills

Technical Reviewer
Molly Holzschlag

Editorial Board
Steve Anglin
Dan Appleman
Ewan Buckingham
Gary Cornell
Jason Gilmore
Jonathan Hassell
Chris Mills
Dominic Shakeshaft
Jim Sumser

Project Manager
Denise Santoro Lincoln

Copy Edit Manager
Nicole LeClerc

Copy Editor
Liz Welch

Assistant Production Director
Kari Brooks-Copony

Production Editor
Kelly Winquist

Compositor and Artist
Diana Van Winkle, Van Winkle Design

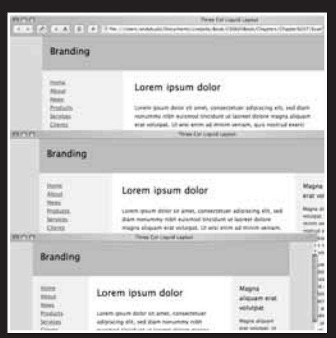
Proofreader
April Eddy

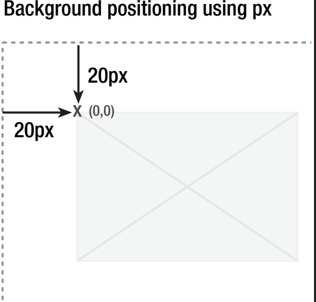
Indexer
John Collin

Interior and Cover Designer
Kurt Krames

Manufacturing Director
Tom Debolski

7 LAYOUT





One of the major benefits of CSS is the ability to control page layout without needing to use presentational markup. However, CSS layout has gained a rather undeserved reputation of being difficult, particularly among those new to the language. This is partly due to browser inconsistencies, but mostly due to a proliferation of different layout techniques available on the Web. It seems that every CSS author has their own technique for creating multicolumn layouts, and new CSS developers will often use a technique without really understanding how it works. This “black box” approach to CSS layout may get quick results, but ultimately stunts the developer’s understanding of the language.

All these CSS layout techniques rely on three basic concepts: positioning, floating, and margin manipulation. The different techniques really aren’t that different, and if you understand the core concepts, it is relatively easy to create your own layouts with little or no hassle.

In this chapter you will learn about

- Horizontally centering a design on a page
- Creating two- and three-column float-based layouts
- Creating fixed-width, liquid, and elastic layouts
- Making columns stretch to the full height of the available space

Centering a design

Long lines of text can be difficult and unpleasant to read. As modern monitors continue to grow in size, the issue of screen readability is becoming increasingly important. One way designers have attempted to tackle this problem is by centering their designs. Rather than spanning the full width of the screen, centered designs span only a portion of the screen, creating shorter and easier-to-read line lengths.

Centered designs are very fashionable at the moment, so learning how to center a design in CSS is one of the first things most developers want to learn. There are two basic methods for centering a design: one uses auto margins and the other uses positioning and negative margins.

Centering a design using auto margins

Say you have a typical layout where you wish to center a wrapper div horizontally on the screen:

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

To do this you simply define the width of your wrapper `div` and then set the horizontal margins to `auto`:

```
#wrapper {
  width: 720px;
  margin: 0 auto;
}
```

In this example I have decided to fix the width of my wrapper `div` in pixels, so that it fits nicely on an 800×600 resolution screen. However, you could just as easily set the width as a percentage of the body or relative to the size of the text using ems.

This works on all modern browsers. However, IE 5.*x* and IE 6 in quirks mode doesn't honor auto margins. Luckily, IE misunderstands `text-align: center`, centering everything instead of just the text. You can use this to your advantage by centering everything in the `body` tag, including the wrapper `div`, and then realigning the contents of the wrapper back to the left:

```
body {
  text-align: center;
}

#wrapper {
  width: 720px;
  margin: 0 auto;
  text-align: left;
}
```

Using the `text-align` property in this way is a hack—but a fairly innocuous hack that has no adverse effect on your code. The wrapper now appears centered in IE as well as more standards-compliant browsers (see Figure 7-1).

Figure 7-1. Centering a design using auto margins

There is one final thing that needs to be done in order for this method to work smoothly in all browsers. In Netscape 6, when the width of the browser window is reduced below the width of the wrapper, the left side of the wrapper spills off the side of the page and cannot be accessed. To keep this from happening, you need to give the body element a minimum width equal to or slightly wider than the width of the wrapper element:

```
body {
  text-align: center;
  min-width: 760px;
}

#wrapper {
  width: 720px;
  margin: 0 auto;
  text-align: left;
}
```

Now if you try to reduce the width of the window below the width of the wrapper `div`, scroll bars will appear, allowing you to access all of the content.

Centering a design using positioning and negative margins

The auto margin method of centering is by far the most common approach, but it does involve using a hack to satisfy IE 5.*x*. It also requires you to style two elements rather than just the one. Because of this, some people prefer to use positioning and negative margins instead.

You start as you did before, by defining the width of the wrapper. You then set the `position` property of the wrapper to `relative` and set the `left` property to `50%`. This will position the left edge of the wrapper in the middle of the page.

```
#wrapper {
  width: 720px;
  position: relative;
  left: 50%;
}
```

However, you don't want the left edge of the wrapper centered—you want the middle of the wrapper centered. You can do this by applying a negative margin to the left side of the wrapper, equal to half the width of the wrapper. This will move the wrapper half its width to the left, centering it on screen:

```
#wrapper {
  width: 720px;
  position: relative;
  left: 50%;
  margin-left: -360px;
}
```

Your choice of centering technique comes down to personal taste. However, it is always useful to have several techniques up your sleeve, as you never know when one may come in handy.

Float-based layouts

There are a few different ways of doing CSS-based layout, including absolute positioning and using negative margins. I find float-based layouts the easiest method to use. As the name suggests, in a float-based layout you simply set the width of the elements you want to position, and then float them left or right.

Because floated elements no longer take up any space in the flow of the document, they no longer appear to exert any influence on the surrounding block boxes. To get around this, you will need to clear the floats at various points throughout the layout. Rather than continuously floating and clearing elements, it is quite common to float nearly everything, and then clear once or twice at strategic points throughout the document, such as the page footer.

Two-column floated layout

To create a simple two-column layout using floats, you need to start off with a basic (X)HTML framework. In this example the (X)HTML consists of a branding area, a content area, an area for the main navigation, and finally a page footer. The whole design is enclosed in a wrapper div, which will be horizontally centered using one of the preceding methods:

```
<div id="wrapper">
<div id="branding">
  ...
</div>
<div id="content">
  ...
</div>
<div id="mainNav">
  ...
</div>
<div id="footer">
  ...
</div>
</div>
```

The main navigation for this design will be on the left side of the page and the content will be on the right. However, I have chosen to put the content area above the navigation in the source order for usability and accessibility reasons. First, the main content is the most important thing on the page and so should come first in the document. Second, there is no point forcing screenreader users to trawl through a potentially long list of links before they get to the content if they don’t have to.

Normally when people create float-based layouts, they float both columns left, and then create a gutter between the columns using margin or padding. When using this approach, the columns are packed tightly into the available space with no room to breathe. Although this wouldn't be a problem if browsers behaved themselves, buggy browsers can cause tightly packed layouts to break, forcing columns to drop below each other.

This can happen on IE because IE/Win honors the size of an element's content, rather than the size of the element itself. In standards-compliant browsers, if the content of an element gets too large, it will simply flow out of the box. However, on IE/Win, if the content of an element becomes too big, the whole element expands. If this happens in very tightly packed layouts, there is no longer enough room for the elements to sit next to each other, and one of the floats will drop. Other IE bugs, such as the 3-pixel text jog bug and the double-margin float bug (see Chapter 9), can also cause float dropping.

To prevent this from happening, you need to avoid cramming floated layouts into their containing elements. Rather than using horizontal margin or padding to create gutters, you can create a virtual gutter by floating one element left and one element right (see Figure 7-2). If one element inadvertently increases in size by a few pixels, rather than immediately running out of horizontal space and dropping down, it will simply grow into the virtual gutter.

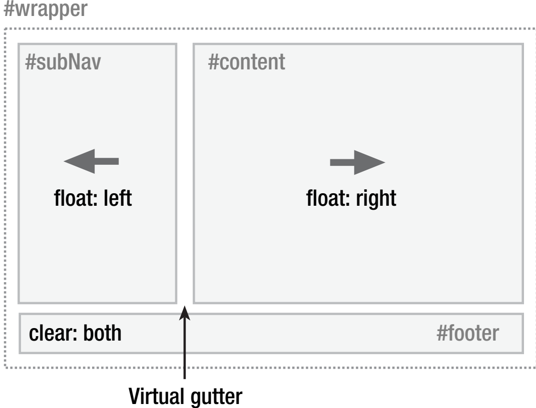


Figure 7-2. Creating a two-column layout using floats

The CSS for achieving this layout is very straightforward. You simply set the desired width of each column, then float the navigation left and the content right:

```
#content {
  width: 520px;
  float: right;
}

#mainNav {
  width: 180px;
  float: left;
}
```

Then, to ensure that the footer is positioned correctly below the two floats, the footer needs to be cleared:

```
#footer {
  clear: both;
}
```

The basic layout is now complete. Just a few small tweaks are required to tidy things up. First, the content in the navigation area is flush to the edges of the container and needs some breathing room. You could add horizontal padding directly to the navigation element, but this will invoke IE 5.*x*'s proprietary box model. To avoid this, add the horizontal padding to the navigation area's content instead:

```
#mainNav {
  padding-top: 20px;
  padding-bottom: 20px;
}

#mainNav li {
  padding-left: 20px;
  padding-right: 20px;
}
```


The right side of the content area is also flush to the right edge of its container and needs some breathing room. Again, rather than apply padding directly to the element, you can apply padding to the content and avoid having to deal with IE's box model problems:

```
#content h1, h2, p {
  padding-right: 20px;
}
```

And there you have it: a simple, two-column CSS layout (see Figure 7-3).

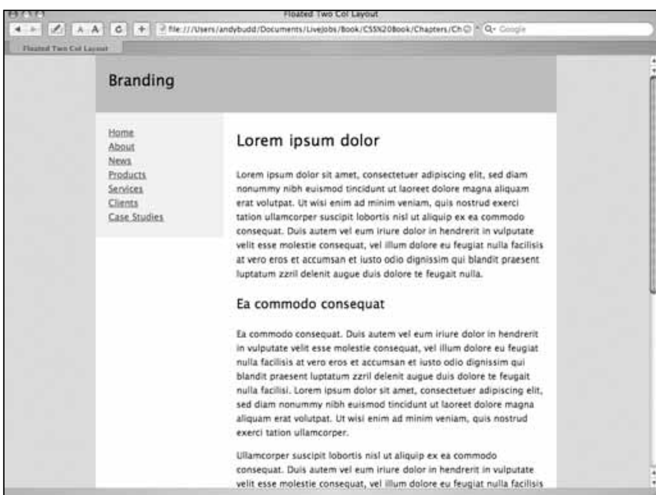


Figure 7-3. Floated two-column layout

Three-column floated layout

The HTML needed to create a three-column layout is very similar to that used by the two-column layout, the only difference being the addition of two new divs inside the content div: one for the main content and one for the secondary content.

```
<div id="content">
<div id="mainContent">
…
</div>
<div id="secondaryContent">
…
</div>
</div>
```

Using the same CSS as the two-column technique, you can float the main content left and the secondary content right, inside the already floated content div (see Figure 7-4). This essentially divides the second content column in two, creating your three-column effect.

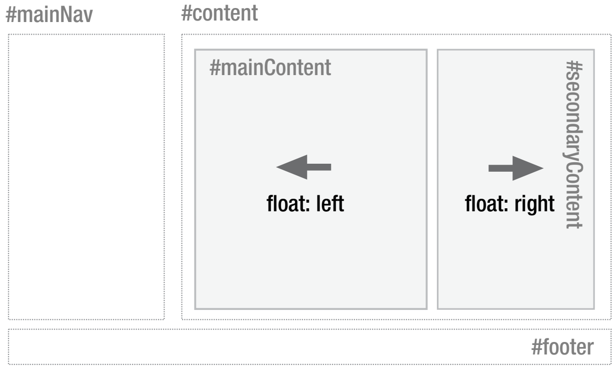


Figure 7-4. Creating a three-column layout by dividing the content column into two columns

As before, the CSS for this is very simple. You just set your desired widths and then float the main content left and the secondary content right:

```
#mainContent {
  width: 320px;
  float: left;
}


#secondaryContent {
  width: 180px;
  float: right;
}
```

You can tidy up the layout slightly by removing the padding from the content element and applying it to the content of the secondary content instead:

```
#secondaryContent h1, h2, p {
  padding-left: 20px;
  padding-right: 20px;
}
```

This leaves you with a nice and solid three-column layout (see Figure 7-5).

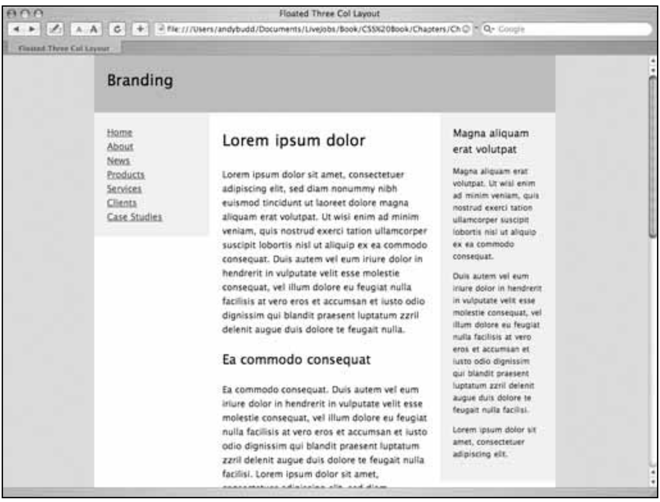


Figure 7-5. Three-column layout using floats

Fixed-width, liquid, and elastic layout

So far, all the examples have used widths defined in pixels. This type of layout is known as fixed-width layout, or sometimes "ice layout" due to its rigid nature. Fixed-width layouts are very common as they give the developer more control over layout and positioning. If you set the width of your design to be 720 pixels wide, it will always be 720 pixels. If you then want a branding image spanning the top of your design, you know it needs to be 720 pixels wide to fit. Knowing the exact width of each element allows you to lay them out precisely and know where everything will be. This predictability makes fixed-width layout by far the most common layout method around.

However, fixed-width designs have their downsides. First, because they are fixed, they are always the same size no matter what your window size. As such, they don't make good use of the available space. On large screen resolutions, designs created for 800×600 can appear tiny and lost in the middle of the screen. Conversely, a design created for a 1024×760 screen will cause horizontal scrolling on smaller screen resolutions. With an increasingly diverse range of screen sizes to contend with, fixed-width design is starting to feel like a poor compromise.

Another issue with fixed-width design revolves around line lengths and text legibility. Fixed-width layouts usually work well with the browser default text size. However, you only

have to increase the text size a couple of steps before sidebars start running out of space and the line lengths get too short to comfortably read.

To work around these issues, you could choose to use liquid or elastic layout instead of fixed-width layout.

Liquid layouts

With liquid layouts, dimensions are set using percentages instead of pixels. This allows liquid layouts to scale in relation to the browser window. As the browser window gets bigger, the columns get wider. Conversely, as the window gets smaller, the columns will reduce in width. Liquid layouts make for very efficient use of space, and the best liquid layouts aren't even noticeable.

However, liquid layouts are not without their own problems. At small window widths, line lengths can get incredibly narrow and difficult to read. This is especially true in multicolumn layouts. As such, it may be worth adding a `min-width` in pixels or ems to prevent the layout from becoming too narrow.

Conversely, if the design spans the entire width of the browser window, line lengths can become long and difficult to read. There are a couple of things you can do to help avoid this problem. First, rather than spanning the whole width, you could make the wrapper span just a percentage—say, 85 percent. You could also consider setting the padding and margin as percentages as well. That way, the padding and margin will increase in width in relation to the window size, stopping the columns from getting too wide, too quickly. Lastly, for very severe cases, you could also choose to set the maximum width of the wrapper in pixels to prevent the content from getting ridiculously wide on oversized monitors.

> *Be aware that IE 5.x on Windows incorrectly calculates padding in relation to the width of the element rather than the width of the parent element. Because of this, setting padding as a percentage can produce inconsistent results in those browsers.*

You can use these techniques to turn the previous fixed-width, three-column layout into a fluid, three-column layout. Start by setting the width of the wrapper as a percentage of the overall width of the window. In this example I have chosen 85 percent as it produces good results on a range of screen sizes. Next, set the width of the navigation and content areas as a percentage of the wrapper width. After a bit of trial and error, setting the navigation area to be 23 percent and the content area to 75 percent produced nice results. This leaves a 2-percent virtual gutter between the navigation and the wrapper to deal with any rounding errors and width irregularities that may occur:

```
#wrapper {
  width: 85%;
}

#mainNav {
  width: 23%;
```

```
  float: left;
}

#content {
  width: 75%;
  float: right;
}
```

You then need to set the widths of the columns in the content area. This gets a bit trickier as the widths of the content divs are based on the width of the content element and not the overall wrapper. If you want the secondaryContent to be the same width as the main navigation, you need to work out what 23 percent of the wrapper is in terms of the width of the content area. This is 23 (width of the nav) divided by 75 (width of the content area), multiplied by 100—which works out at around 31 percent. You will want the gutter between the content columns to be the same width as the gutter between the navigation and content areas. Using the same method, this works out to be around 3 percent, meaning that the width of the main content area should be 66 percent:

```
#mainContent {
  width: 66%;
  float: left;
}

#secondaryContent {
  width: 31%;
  float: right;
}
```

This produces a liquid layout that is optimal at 1024×780 but is comfortable to read at both larger and smaller screen resolutions (see Figure 7-6).

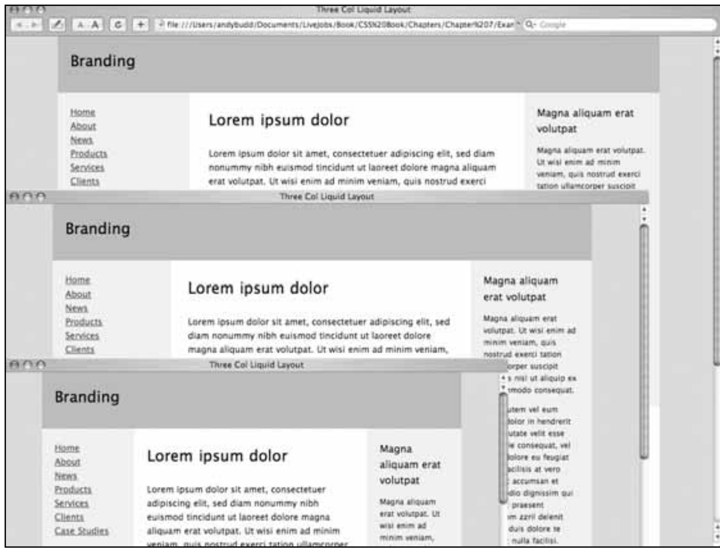


Figure 7-6. Three-column liquid layout at 800×600, 1024×768, and 1152×900

Because this layout scales so nicely, there isn't any need to add a max-width property. However, the content does start to get squashed at smaller sizes, so you could set a minimum width of 720px on the wrapper element if you liked.

Elastic layouts

While liquid layouts are useful for making the most of the available space, line lengths can still get uncomfortably long on large resolution monitors. Conversely, lines can become very short and fragmented in narrow windows or when the text size is increased a couple of steps. If this is a concern, then elastic layouts may be a possible solution.

Elastic layouts work by setting the width of elements relative to the font size instead of the browser width. By setting widths in ems, you ensure that when the font size is increased the whole layout scales. This allows you to keep line lengths to a readable size and is particularly useful for people with reduced vision or cognitive disorders.

Like other layout techniques, elastic layouts are not without their problems. Elastic layouts share some of their problems with fixed-width layouts, such as not making the most use of the available space. Also, because the whole layout increases when the text size is increased, elastic layouts can become much wider than the browser window, forcing the appearance of horizontal scroll bars. To combat this, it may be worth adding a max-width of 100% to the body tag. max-width isn't currently supported by IE 6 and below, but it is supported by standards-complaint browsers such as Safari and Firefox.

Elastic layouts are much easier to create than liquid layouts as all of the HTML elements essentially stay in the same place relative to each other; they just all increase in size. Turning a fixed-width layout into an elastic layout is a relatively simple task. The trick is to set the base font size so that 1em roughly equals 10 pixels.

The default font size on most browsers is 16 pixels. Ten pixels works out at 62.5 percent of 16 pixels, so setting the font size on the body to 62.5% does the trick:

```
body {
  font-size: 62.5%;
}
```

Because 1em now equals 10 pixels at the default font size, we can convert our fixed-width layout into an elastic layout by converting all the pixel widths to em widths:

```
#wrapper {
  width: 72em;
  margin: 0 auto;
  text-align: left;
}

#mainNav {
  width: 18em;
  float: left;
}
```

```
#content {
  width: 52em;
  float: right;
}

#mainContent {
  width: 32em;
  float: left;
}

#secondaryContent {
  width: 18em;
  float: right;
}
```

This produces a layout that looks identical to the fixed-width layout at regular text sizes (see Figure 7-7), but scales beautifully as the text size is increased (see Figure 7-8).

Figure 7-7. Elastic layout at the default text size

Figure 7-8. Elastic layout after the text size has been increased a few times

Elastic-liquid hybrid

Lastly, you could choose to create a hybrid layout that combines both elastic and liquid techniques. This hybrid approach works by setting the widths in ems, then setting the maximum widths as percentages:

```
#wrapper {
  width: 72em;
  max-width: 100%;
  margin: 0 auto;
  text-align: left;
}

#mainNav {
  width: 18em;
  max-width: 23%;
  float: left;
}

#content {
  width: 52em;
  max-width: 75%;
  float: right;
}

#mainContent {
  width: 32em;
  max-width: 66%;
  float: left;
}

#secondaryContent {
  width: 18em;
  max-width: 31%;
  float: right;
}
```

On browsers that support `max-width`, this layout will scale relative to the font size but will never get any larger than the width of the window (see Figure 7-9).

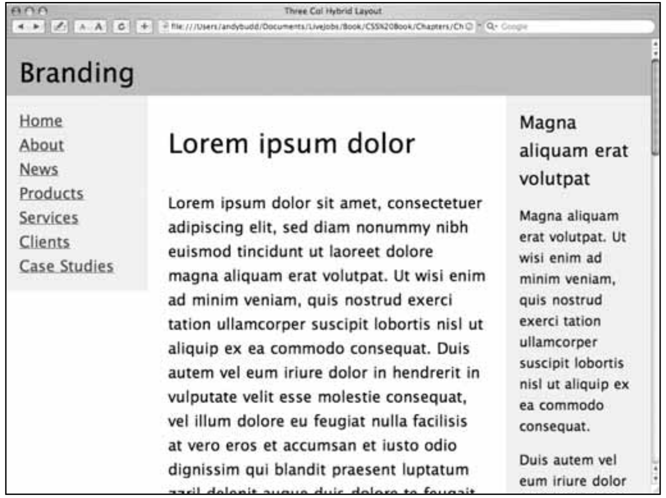


Figure 7-9. The elastic-liquid hybrid layout never scales larger than the browser window.

Liquid and elastic images

If you choose to use a liquid or an elastic layout, fixed-width images can have a drastic effect on your design. When the width of the layout is reduced, images will shift and may interact negatively with each other. Images will create natural minimum widths, preventing some elements from reducing in size. Other images will break out of their containing elements, wreaking havoc on finely tuned designs. Increasing the width of the layout can also have dramatic consequences, creating unwanted gaps and unbalancing designs. But never fear—there are a few ways to avoid such problems.

For images that need to span a wide area, such as those found in the site header or branding areas, consider using a background image rather than an image element. As the `branding` element scales, more or less of the background image will be revealed:

```
#branding {
  height: 171px;
  background: url(images/branding.png) no-repeat left top;
}

<div id="branding"></div>
```

If the image needs to be on the page as an image element, try setting the width of the container element to 100% and the overflow property to hidden. The image will be truncated so that it fits inside the branding element but will scale as the layout scales:

```
#branding {
  width: 100%;
  overflow: hidden;
}

<div id="branding">
  <img src="images/branding.png" width="1600" height="171" />
</div>
```

For regular content images, you will probably want them to scale vertically as well as horizontally to avoid clipping. You can do this by adding an image element to the page without any stated dimensions. You then set the percentage width of the image, and add a max-width the same size as the image to prevent pixelization.

> *Remember that* max-width *only works in more modern browsers such as Safari and Firefox. If you are concerned about the image pixelating in older browsers, make the image as large as you will ever need it to be.*

For example, say you wanted to create a news story style with a narrow image column on the left and a larger text column on the right. The image needs to be roughly a quarter of the width of the containing box, with the text taking up the rest of the space. You can do this by simply setting the width of the image to 25% and then setting the max-width to be the size of the image—in this case 200 pixels wide:

```
.news img {
  width: 25%;
  max-width: 200px;
   float: left;
  padding: 2%;
}

.news p {
  width: 68%;
  float: right;
  padding: 2% 2% 2% 0;
}
```

As the news element expands or contracts, the image and paragraphs will also expand or contract, maintaining their visual balance (see Figure 7-10). However, on standards-complaint browsers, the image will never get larger than its actual size.

Figure 7-10. Giving images a percentage width allows them to scale nicely in relation to their surroundings.

Faux columns

You may have noticed that the navigation and secondary content areas on all these layouts have been given a light gray background. Ideally the background would stretch the full height of the layout, creating a column effect. However, because the navigation and secondary content areas don't span the full height, neither do their backgrounds.

To create the column effect, you need to create fake columns by applying a repeating background image to an element that does span the full height of the layout, such as a wrapper `div`. Dan Cederholm coined the term "faux column" to describe this technique.

Starting with the fixed-width, two-column layout, you can simply apply a vertically repeating background image, the same width as the navigation area, to the wrapper element (see Figure 7-11):

```
#wrapper {
  background: #fff url(images/nav-bg-fixed.gif) repeat-y left top;
}
```

Figure 7-11. Faux fixed-width column

For the three-column fixed width layout, you can use a similar approach. This time, however, your repeating background image needs to span the whole width of the wrapper and include both columns (see Figure 7-12). Applying this image in the same way as before creates a lovely faux two-column effect (see Figure 7-13).

Figure 7-12. Background image used to create the faux three-column effect

Figure 7-13. Faux three-column effect

Creating faux columns for fixed-width designs is relatively easy, as you always know the size of the columns and their position. Creating faux columns for fluid layouts is a little more complicated; the columns change shape and position as the browser window is

scaled. The trick to fluid faux columns lies in the use of percentages to position the background image.

If you set a background position using pixels, the top-left corner of the image is positioned from the top-left corner of the element by the specified number of pixels. With percentage positioning, it is the corresponding point on the image that gets positioned. So if you set a vertical and horizontal position of 20 percent, you are actually positioning a point 20 percent from the top left of the image, 20 percent from the top left of the parent element (see Figure 7-14).

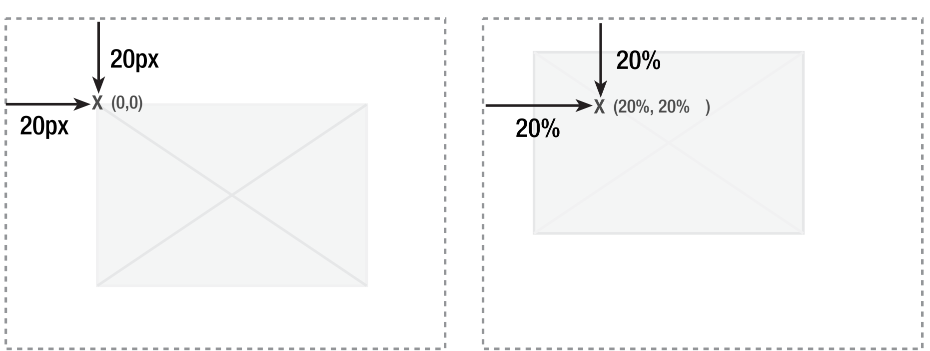


Figure 7-14. When positioning using percentages, the corresponding position on the image is used.

This is very useful as it allows you to create background images with the same horizontal proportions as your layout, and then position them where you want the columns to appear.

To create a faux column for the navigation area, you start by creating a very wide background image. In this example, I have created an image that is 2000 pixels wide and 5 pixels high. Next you need to create an area on the background image to act as the faux column. The navigation element has been set to be 23 percent of the width of the wrapper, so you need to create a corresponding area on the background image that is 23 percent wide. For a background image that is 2000 pixels wide, the faux column part of the image needs to be 460 pixels wide. Output this image as a GIF, making sure that the area not covered by the faux column is transparent.

The right edge of the faux column is now 23 percent from the left side of the image. The right edge of the navigation element is 23 percent from the left edge of the wrapper element. That means if you apply the image as a background to the wrapper element, and set the horizontal position to be 23 percent, the right edge of the faux column will line up perfectly with the right edge of the navigation element.

```
#wrapper {
  background: #fff url(images/nav-faux-column.gif) repeat-y 23% 0;
}
```

You can create the background for the secondary content area using a similar method. The left edge of this faux column should start 77 percent from the left edge of the image,

matching the position of the secondaryContent element relative to the wrapper. Because the wrapper element already has a background image applied to it, you will need to add a second wrapper element inside the first. You can then apply your second faux column background image to this new wrapper element.

```
#wrapper2 {
  background: url(images/secondary-faux-column.gif) repeat-y 77% 0;
}
```

If you have worked out your proportions correctly, you should be left with a beautiful three-column liquid layout with columns that stretch the height of the wrapper (see Figure 7-15).

Figure 7-15. Faux three-column layout

Summary

In this chapter you learned how to create simple two- and three-column fixed-width layouts using floats. You then learned how these layouts could be converted into liquid and elastic layouts with relative ease. You learned about some of the problems associated with liquid and elastic layouts and how liquid images and hybrid layouts can help solve some of these problems. Lastly, you saw how to create full height column effects on both fixed-width and liquid layouts, using vertically repeating background images. This chapter touched on some of the techniques used to create CSS-based layouts. However, there are a lot of techniques out there, enough to fill a whole book of their own.

One of the big problems developers face with CSS layouts is that of browser inconsistency. To get around browser rendering issues, various hacks and filters have been created. In the next chapter, you will learn about some of the better-known hacks and how to use them responsibly.