Complement canvas with HTML markup, Part 1: Blend the canvas API and HTML/CSS model

Hybrids that yield the best of both worlds

Kevin Moot
Ryan DeLuca

August 07, 2012

HTML canvas excels in many ways, including the great performance that comes with low overhead and direct pixel manipulation. However, canvas falls short in a few areas where HTML does exceedingly well: text rendering, SEO, accessibility, and device-independent markup. This article compares and contrasts the strengths of the traditional HTML model and the canvas API. Explore the idea of a hybrid HTML/Canvas application that uses the best aspects of both worlds. You'll also review different techniques for superimposing HTML elements on top of a canvas element.

View more content in this series

Introduction

In Part 1 of this two-part series, learn about creating applications that combine the advantages of the canvas API and the HTML/CSS model. The canvas API can be a great choice for web applications demanding high performance, low-overhead graphics. However, it can be short-sighted to architect an entire application around canvas without considering the usefulness of the traditional HTML/CSS model. Rather than taking it for granted, learn to leverage the many useful aspects of HTML/CSS which can be difficult, if not impossible, to achieve in canvas.

Frequently used abbreviations

- API: Application Programming Interface
- CSS: Cascading Style Sheets
- HTML: HyperText Markup Language
- UI: User Interface

This article examines a "hybrid" approach to designing applications that utilize traditional HTML components alongside canvas elements. The motivations for such an approach will be discussed, including why a pure canvas-based implementation can be far from ideal, along with an examination of the strengths and weaknesses of the traditional HTML model compared with the...
canvas API. Several examples also help you plan the layout and interaction between HTML and canvas elements when designing your application.

You can download the source code for the examples used in this article.

**Strengths of the HTML and CSS model**

The traditional web model using HTML and CSS excels in several scenarios. This section reviews the most useful benefits of the HTML model that aren't supported particularly well (or at all) in the canvas API—in particular, its robust text support and the benefits associated with semantically meaningful HTML markup.

**HTML's robust text support**

A major strength of HTML is its ability to easily annotate text with styling cues, ranging from formatting tags such as `<b></b>` to CSS rules such as `font-weight:bold`.

The canvas API, on the other hand, exposes a `fillText()` method that renders a text string as a bitmap. It is simple, low-level call with a number of limitations. One of the biggest limitations is that only one uniform style rule can be applied to the text string provided to `fillText()`. A "style rule" in this context can be thought of as a single rule that sets a font face, size, color, and so on (similar to a CSS rule).

**Text with uniform style**

Consider the text in Figure 1.

**Figure 1. Sample text**

Because the text contains a uniform style rule throughout—a red, italic, Arial font—you can easily render this text equally well using either HTML/CSS or canvas.

The code in Listing 1 and Listing 2 compares how to render this text using both HTML and CSS and canvas. See Related topics for a working example.

Listing 1 shows the text from Figure 1 rendered in HTML and CSS.

**Listing 1. Rendering text with a uniform style in HTML and CSS**

```html
<style>
  .uniform-style {
    font: italic 12px Arial;
    color: red;
  }
</style>
<span class="uniform-style">
  Variety is the spice of life!
</span>
```
Listing 2 shows the same text rendered using canvas.

**Listing 2. Rendering text with a uniform style in canvas**

```javascript
var canvas = document.getElementById('my_canvas');
var context = canvas.getContext('2d');
context.font = 'italic 12px Arial';
context.fillStyle = 'red';
context.fillText('Variety is the spice of life!', 0, 50);
```

**Text with dynamic style**

Consider the text in Figure 2.

**Figure 2. Text with dynamic style**

```
Variety is the spice of life!
```

Listing 3 illustrates the CSS rules necessary to render this text.

**Listing 3. Rendering text with a dynamic style in HTML and CSS**

```html
<style>
  .dynamic-style {
    font: 12px Arial;
    color: blue;
  }
  .dynamic-style strong {
    font-size: 18px;
    color: green;
  }
  .dynamic-style em {
    color: red;
    font-weight: bold;
    font-style: italic;
  }
</style>

<span class="dynamic-style">
  <strong>Variety</strong> is the <em>spice of life</em>!
</span>
```

Because no concept of HTML tags or CSS classes exist in the canvas API which would allow you to annotate the use of different font styles, rendering the same text block in the previous example is a much more complex matter.

Because the canvas API functions as a sequential state machine, reproducing the effect in Figure 2 requires what could be described as a "typewriter-like" approach. You must select a style to use, type the portion of text that needs to use this style, select a different style to use, type the next portion of text, and so on.

In general, this can be accomplished using the following algorithm:

1. Set the style to use for the next portion of text (18px Arial, green).
2. Render the next portion of text to the canvas (Variety).
3. Move a certain number of pixels to the right (60).
4. Repeat steps 1-3 for each portion of text.

Listing 4 shows the most basic, naive implementation of this algorithm.

**Listing 4. Rendering text with a dynamic style in canvas**

```javascript
context.font = '18px Arial';
context.fillStyle = 'green';
context.fillText('Variety', 0, 50);
context.translate(60, 0); // move 60 pixels to the right (a)

context.font = '12px Arial';
context.fillStyle = 'blue';
context.fillText('is the', 0, 50);
context.translate(35, 0); // move 35 pixels to the right (b)

context.font = 'italic bold 12px Arial';
context.fillStyle = 'red';
context.fillText('spice of life!', 0, 50); // (c)
```

Figure 3 illustrates how the three separate text blocks will appear as they are rendered in sequence.

**Figure 3. Text rendering in Canvas**

![Figure 3](image_url)

Later in this article, you'll see how to make some improvements to this approach by adding another layer of abstraction on top of the native canvas `fillText()` function. Regardless of the approach, the same typewriter algorithm will remain the core.

It should be noted that most text styles that are taken for granted in HTML/CSS, such as word wrapping, letter spacing, and line height, are not natively supported by the canvas API. If needed, it would be necessary to modify the algorithm and implement the functions yourself.

By now it should be evident why text rendering is a non-trivial problem for the canvas API, and why the traditional HTML model prevails when it comes to formatting text with a minimal amount of effort.

**HTML's meaningful semantics**

A fundamental of good web design is meaningful, semantically correct HTML structure. A number of benefits are derived from HTML markup that is written with proper structure and meaning in mind.

At its core, canvas is simply a graphics plotter, so the concept of meaningful markup does not exist, and the benefits of meaningful HTML markup will essentially be forfeited for applications...
written purely for canvas. For example, a canvas element will essentially appear as a black box of nothingness from the perspective of search engines and vision-impaired users.

Below, we'll examine several benefits derived from meaningful HTML markup.

**Search engine awareness**

Search engines create their search indexes based upon information read by automated search bots/spiders—when a search spider visits your web page, it parses the HTML to extract as much meaningful information as possible.

Because text that is programmatically rendered on the canvas is simply a bitmap in the end, the text will be completely ignored by the search spiders. Therefore, it is important to understand that if your application is developed exclusively in canvas, search spiders will be able to glean very little contextual information from your page.

Nonetheless, search engines will still be able to read any fallback text within a `<canvas>` element intended for non-HTML5 compliant browsers, as in Listing 5.

**Listing 5. Fallback text within a canvas element**

```html
<canvas>
  We are sorry, your browser doesn't support HTML5!
</canvas>
```

This is not a particularly novel issue for canvas—lack of search engine awareness has historically been a problem faced by applications written for browser plug-ins such as Flash, and Silverlight.

**Accessibility**

Vision-impaired users rely on text-to-speech tools (screen readers) to read content on web pages. Structured HTML markup makes it possible for screen reader tools to understand the difference between a header, a footer, a list, and so on. Applications that do not produce HTML markup, such as canvas-based applications, will leave no trace of content behind that is recognizable by screen readers.

Well-structured HTML also enables users to use standard keyboard shortcuts to navigate the page. For instance, the tab key will bring focus to the next link on the page, as in Figure 4.

**Figure 4. Using the tab key to bring focus to a link on the page**
Even browser plug-ins such as Flash and Silverlight have made great strides in providing accessibility by supporting commonly used keyboard shortcuts. Unfortunately, canvas provides no similar built-in compliance for accessibility.

**Devices and user agents**

Ultimately, it is up to the user agent to decide how your page will be rendered. Some devices may interpret the same HTML markup differently than others. For instance, on most mobile devices, tapping an `<input type="text">` element causes the device to bring up an on-screen keyboard so the user can enter input. Browsers conforming to the HTML5 spec provide for an even wider variety of available input types such as email, website, phone number, date, and so on.

Figure 5 shows how an `<input type="date">` element appears on an iOS device. The browser automatically displays a full-featured date picker; no additional code is required to implement the date picker other than defining the type of the input element as `date`.

**Figure 5. iPhone treatment of `<input type="date">`**

When writing semantically correct HTML, you are essentially making a contract with user agents, trusting that the user agent is able to present the content in the best way for users of that particular device.

In contrast, all content rendered in a canvas element appears consistently the same on all devices. You're breaking the pact with the user agent; no longer holding trust that the device knows best how to present your content.

**Hyperlink awareness**

At its core, the web is built around hyperlinks—it's hard to imagine a world without them. Beyond allowing users to navigate to links, modern user agents provide users with a wealth of additional contextual features: open pages in a new tab, copy link addresses, email links, and so on.

Although text can be rendered to a canvas, technically, no concept of hyperlinks exists.

Even if you took it upon yourself to simulate a hyperlink by rendering blue, underlined text, the user agent will be unaware that this is a true hyperlink and will be unable to offer any useful contextual function to the user.

**Strengths of the canvas API**

Considering the strengths of HTML, things might be looking grim for applications written for canvas. However, there are several benefits to architecting a solution around canvas. The major
advantages lie in canvas' high degree of graphical performance and potential for cross-device consistency.

**Graphical performance**

There are certain tasks that both HTML and canvas can accomplish, such as rendering images, text, and animations.

Because canvas is not burdened with the overhead associated with parsing HTML and maintaining a hierarchical document model, these tasks will invariably perform faster when done in canvas.

In HTML-centric applications, the predominant performance killer is the adding, removing, or updating of a node. These updates to the document model often force the browser to repaint or reflow the entire page, which can be a very computationally expensive process. Making updates to the document model many times per second, necessary in a realtime application, can slow the browser down to a crawl.

When speed is of the utmost importance, switching away from an HTML-centric architecture to a canvas-centric architecture can be the holy grail in achieving maximum performance. The high level of performance comes at the price of dealing with a low-level API rather than a robust markup language like HTML and the flexibility of CSS.

**Cross-device consistency**

Consistent, cross-platform appearance has always been a struggle with the traditional HTML/CSS model due to variances in browser layout engines. Although web pages can be made to look very similar across platforms, it is difficult to achieve a "pixel perfect" level of consistent design.

These issues are eliminated when graphics are plotted on a canvas. Canvas enables a pixel-level degree of control over the appearance of graphics and text, and a guarantee that output will be 100% consistent across all platforms.

**Rationale for canvas/HTML hybrids**

When creating a canvas-oriented application, features of the traditional HTML/CSS model can be utilized to overcome canvas' shortcomings.

As already noted, one of the biggest limitations of canvas is the difficulty in implementing robust, accessible UI components. When deciding on the best approach for implementing a UI, it can be a useful exercise to evaluate the requirements of the UI elements. Criteria to consider could include:

- What is the desired content of the UI element?
- How often must this content be updated?
- What level, if any, of interactivity is desired?

A strong requirement in any of these areas can greatly suggest one implementation or the other. For example, if the desired content is a simple textual display of some aspect of the application,
a basic HTML element would probably be most efficient and would not lead to any performance loss. If, however, you wanted the element to be animated or rich in frequently-updated content, using canvas might be a better approach. If ease of interactivity is a requirement, HTML elements typically provide the most natural approach, especially when paired with the functions provided by libraries such as jQuery.

Below, we will examine a number of situations that are well-suited toward a hybrid canvas/HTML implementation.

**Rapid prototypes**

Putting together a rapid prototype can potentially be accomplished much more quickly using HTML and CSS than canvas. For example, consider an application that features a menu. Because menus lend themselves well to an HTML/CSS structure, and you can leverage a wealth of existing libraries, such jQuery plug-ins, the amount of effort to implement a menu in HTML is trivial compared with its equivalent implementation in canvas. The performance drain associated with HTML elements is not as big of an issue with our prototype because it's intended as a proof-of-concept rather than a fully optimized system.

Aspects of the prototype that make use of HTML and CSS may later be implemented in pure canvas when the application is fully fleshed out. In the case of the menu example, converting the menu to canvas in the final product will noticeably improve performance because you'd be removing dependencies on the browser document model.

**Development tools**

Development tools can be a natural fit for HTML and CSS given the ease of using HTML markup to display a dense amount of information. An example would be the addition of a console in the application for debugging purposes. The debug console might display statistics such as frames-per-second, a list of all the objects in the application and their coordinates, and so on.

Again, the performance hit of introducing HTML elements is not a big issue in this case because the tools are intended for developers only and will never be displayed to the end user.

**User interface overlays**

If the application requires a rich UI that combines text with form elements such as checkboxes and drop-downs, HTML is a natural solution. Reproducing these types of components in canvas can be very time-consuming, and your application will likely suffer from accessibility issues.

Conversely, the strengths of HTML can provide for a rich user experience. For example, a collection of HTML-based UI components can comprise a "heads up" display that floats above the canvas area.

To minimize the impact on the realtime performance of the application, be mindful of making as few changes to the document model as possible (element updates, insertions, and removals). If document updates are frequent, the performance of the application could slow to a crawl.
Companion user interfaces

When a UI element is intended to appear outside of the interactive, realtime component of the application, it can be a great opportunity to use HTML and CSS. For example, the UI element might only appear during an introductory start sequence or when the application is paused.

A load/save game interface is another great example where an HTML-based companion UI can be utilized. Because the game is paused while the user is saving their game, there's no impact on the realtime performance of the application. HTML/CSS would allow this interface to be developed rapidly, reaping all of the benefits of well-structured markup.

Implementing canvas/HTML hybrids

This section reviews several basic approaches for combining a canvas element with other HTML elements within a web page layout. While not exhaustive, the approaches outlined below provide several examples of how to best take advantage of the strengths of both HTML and canvas while minimizing their weaknesses. When planning the architecture of your application, consider which approach (or set of approaches) will best suit your goals.

The following scenarios assume two primary components:

- The canvas element, which contains primarily the graphics and animation component
- HTML elements, which contain different aspects of the UI such as menus, navigation elements, forms, and so on

Flowing a canvas element

The `<canvas>` tag is considered a block-level element, so the simplest approach for combining HTML and canvas elements is to simply insert a `<canvas>` element within the document body. Because the element will be relatively positioned by default, no special markup or CSS is necessary for the canvas element to flow along with the rest of the document.

Canvas as a foreground element

A `<canvas>` element can be absolutely-positioned on the page with a z-index such that it appears in the foreground.

This approach is suitable for canvas applications that will not be interacting with other HTML elements found on the page, similar to embedding a Flash application.

Because the canvas is the top-most element in the foreground, the canvas element will be able to capture all forms of user input (mouse clicks, touch events, and so on).

Figure 6 illustrates the method of positioning of a canvas as an absolutely-positioned foreground element.
Figure 6. Layering a canvas element above the document
The code in listing 6 demonstrates the CSS necessary to place the element as the top-most foreground layer, using a large z-index.

**Listing 6. CSS that positions the canvas as a foreground element**

```css
canvas {
    position: absolute;
    z-index: 100;
    bottom: 0px;
    right: 0px;
}
```

**Canvas as a background element**

A `<canvas>` element can also be used to display content on the background of a web page.

This approach can be used to produce rich, animated background scenes underneath standard HTML content. The background canvas scene will appear visible through any transparent areas within the document.
Complement canvas with HTML markup, Part 1: Blend the canvas API and HTML/CSS model.

Figure 7. Layering a canvas element below the document.
The code in Listing 7 shows how to position the canvas element as the bottom-most layer by setting a negative z-index.

**Listing 7. CSS to position the canvas as a background element**

```css
canvas {
  position: absolute;
  z-index: -1;
  top: 0px;
  left: 0px;
}
```

**Layering on top of canvas**

Another approach involves layering one or more HTML elements above a canvas element. As shown in Figure 8, this approach can be a strong contender for applications that require a rich set of UI components. In this case, a set of HTML-based UI components would float above the graphics layer provided by the canvas.
Figure 8. Layering HTML elements above a canvas
Each of the individual UI components can be arranged using CSS so that they appear at a certain position in the foreground above the canvas.

Listing 8 shows an example of how to position a UI component contained within a `<div>` element by setting a high z-index.

### Listing 8. CSS to position an HTML component as a foreground element

```css
div.user-interface {
    position: absolute;
    z-index: 100;
    top: 50px;
    left: 50px;
}
```

Because the HTML-based UI components are on the top-most layer, they will respond by default to user input. For instance, if a certain element contains a UI element such as a checkbox, a scrollbar, or a hyperlink, a user's mouse click will be intercepted by these elements. Normally, a click on one of these elements would never fall through to the canvas surface below. However, in some cases, it might be desirable for user input to fall through to the canvas.

A particular HTML element can be configured to ignore user input by setting the pointer-events CSS property, as shown in Listing 9. Setting pointer-events to `none` causes the element to ignore all behaviors that would normally be associated with user input—all text, hyperlinks, and form elements within that `<div>` would be unselectable and essentially unusable.

### Listing 9. CSS to set the pointer-events property

```css
div.user-interface {
    position: absolute;
    z-index: 100;
    top: 50px;
    left: 50px;
    pointer-events: none;
}
```

The input events fall onto the layer below, where the `<canvas>` element could intercept the event. For example, an event listener might be in place on the `<canvas>` element that captures the mouse-move event.

The pointer-event's property is supported by all browsers with the exception of Internet Explorer (workarounds exist to mimic a similar behavior in Internet Explorer).

### Layering on top of canvas with an input surface

An alternate method of capturing user input is similar to the previous approach, but does not require the use of a `pointer-events:none` CSS property. In this approach, one additional layer is added on top of the canvas and HTML UI elements. The additional layer is added for the sole purpose of capturing user input.
This approach is suitable for scenarios where HTML elements exist for display purposes only. No interactivity with the HTML interface elements will be possible because the input surface would trap all events before they reach the underlying elements. Figure 9 shows an example.

**Figure 9. Layering HTML elements above a canvas with an additional input surface**

You can implement the input surface by creating an absolutely positioned `<div>` with the same width and height as the canvas. Leaving the `<div>` empty essentially leaves you with a transparent layer whose sole purpose is to capture user input.

To capture user input, simply attach all of your event listeners to this `<div>`. Because it is the top-most element, all user input will be trapped by this element rather than the underlying elements.

**Next steps**

Part 2 of this series will expand on the concepts covered in this article and walk through an example implementation of a hybrid canvas/HTML application. Step by step, you'll learn how to add a basic HTML UI on top of a canvas game, and how to create a UI element in canvas. Part 2 will also discuss considerations for animation and text rendering and illustrate how to harness the advantages of HTML and canvas.

Figure 10 illustrates the sample game concept built in Part 2.
Conclusion

In this article, you learned about the strengths and weaknesses of the HTML model compared with the canvas API, the reasons for building a hybrid canvas/HTML application, and different approaches for layering HTML and canvas elements.

Now that you’re thinking about how to select an architecture to best use the strengths of HTML and canvas, watch for Part 2 of this series, where you’ll get to use your new skills to implement a space shooter game.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Article source code</td>
<td>canvashtmlpt1sourcecode.zip</td>
<td>2KB</td>
</tr>
</tbody>
</table>
Related topics

- "Create great graphics with the HTML5 canvas" (developerWorks, February 2011): Read how to enhance your web pages with canvas, a simple HTML5 element that packs a punch.
- "HTML5 fundamentals, Part 4: The final touch - The Canvas" (developerWorks, July 2011): Learn more about the changes in HTML5 and the HTML5 canvas element.
- **Canvas Pixel Manipulation**: View this demo, from the Safari Dev Center, to learn about managing the canvas to develop effective visual assets.
- **WHATWG**: Explore this community of developers working with the W3C to fine-tune HTML5.
- **Canvas tutorial**: Check out this tutorial from the Mozilla developers to learn how to implement the canvas element in your HTML pages.
- **HTML5 Canvas reference**: Use the exercises in this reference from W3Schools.com to help hone your canvas knowledge.
- **jQuery Events API**: Learn more about methods used to register behaviors to take effect when the user interacts with the browser.
- **jQuery**: Get the popular JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development.
- **Modernizr**: Download the open-source JavaScript library that helps you build the next generation of HTML5 and CSS3-powered web sites.
- **developerWorks Web development zone**: Find articles covering various web-based solutions. See the Web development technical library for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- **IBM product evaluation versions**: Get your hands on application development tools and middleware products from DB2, Lotus, Rational, Tivoli, and WebSphere.

© Copyright IBM Corporation 2012
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)