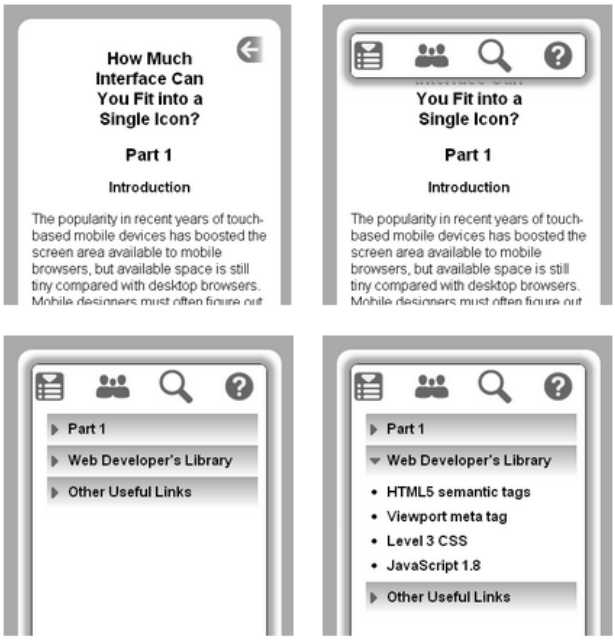
How much interface can you fit into a single icon? Part 2

This article shows how to use CSS3 transitions to create a series of expandable interface options. This is second part of a two-part article, we strongly recommend that you first read How much interface can you fit into a single icon? Part 1.

Introduction

The first part of this article discussed a simple way to make an application's controls available to users at all times, even in the midst of long scrolling pages. This part discusses how to expand that small control into a larger panel, making many different site-wide options available. We'll focus on implementing a nested accordion interface with which to access various navigation levels at once. We'll also discuss how to make the interface gracefully dismiss itself once it's no longer needed.

The application is implemented using a little bit of JavaScript, but with most of the work done by Level 3 CSS transitions. You can use a Webkit-based browser to view the effects described here in this sample page, which demonstrates the following interface flow:



Expanding the Panel

So far, the application simply toggled between the navigation item's default interface state and a *scrolling* class, which identified when the icon needed to be repositioned. To that we'll add a third expanded state, by adding a touch handler to the navigation item, which the application has already made available as *ui.nav*:

```
ui.nav.addEventListener( 'click', ui.expandNav );

ui.expandNav = function( event ) {
    var n = event.currentTarget;
    if ( ! n.className ) event.stopPropagation();
    n.className = 'expanded';
};
```

Here's the relevant CSS transition:

```
#icon {
  -webkit-transition  : all 1s ease-in-out;
  text-align          : justify;
  background-image    : url(img/icon_expand-nav.png);
  background-position : 100% 0%;
  background-repeat   : no-repeat;
  border              : thin solid transparent;
  width               : 10%;
}

#icon.expanded {
  -webkit-box-shadow  : 0em 0em 0.5em 0.5em #aaaaaa;
  background-color    : #ffffff;
  background-position : 200% 0%;
  border-color        : #000000;
  border-radius       : 0.5em;
  width               : 80%;
}
```


(C) Copyright Nokia 2013. All rights reserved.

The main transition shifts the `width` to expand from the size of an icon to cover most of the page. Note that the default class specifies the narrower width as a percentage rather than as the absolute size of the icon in pixels. Here, percentages offer greater flexibility to target different screen sizes, and account for the fact that transitions between fixed and flexible units don't work.

The `background-image` displays the default icon along the right edge. The `background-position` transition makes the icon fall away to the right when the panel is expanded, and slide back in when collapsed.

Additional properties provide an opaque white background and a border highlighted with a Level 3 CSS box shadow. The gray shadow is applied with no offset, but with blur and spread options that provide a backlit halo effect.

Read more about CSS3 Transitions.

Once you start testing the interface's responsiveness at this point, you may notice how well CSS transitions perform. You can interrupt or even reverse the course of a transition as it executes.

Collapsing the Panel

Most popup panels in desktop web interfaces feature an explicit control to dismiss them, usually resembling the letter **X**. In this mobile interface, that additional item might take up unnecessary space on the screen. Instead, this application implements a looser interface. Any tap on the screen that doesn't further expand the navigation bar causes it to collapse back down to its default icon.

The `section` tag that covers the entire screen serves as a handy target for a fallback touch handler, one that removes the navigation element's className to return it to its default interface state:

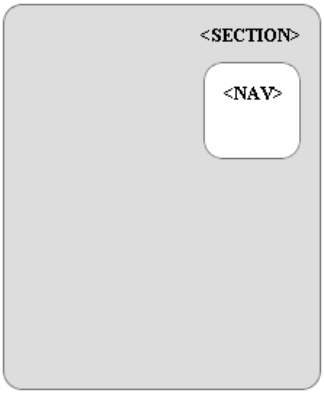
```
ui.page = document.querySelector('section');
ui.page.addEventListener( 'click', ui.defaultNav );
```

For this mechanism to work, the handler that initially expands the navigation panel must run `stopPropagation()`, to keep the event from percolating up the DOM tree and causing it to collapse again.

```
ui.expandNav = function( event ) {
    if ( ! event.currentTarget.className ) event.stopPropagation();
    event.currentTarget.className = 'expanded';
};
```

If the panel is already expanded, tapping anywhere within it sends the event up to the default touch handler by default, which collapses the panel. Regardless of whether you tap on a destination option or on an inactive portion of the screen, the panel dismisses itself. Only for handlers that call `stopPropagation()` does it remain open.

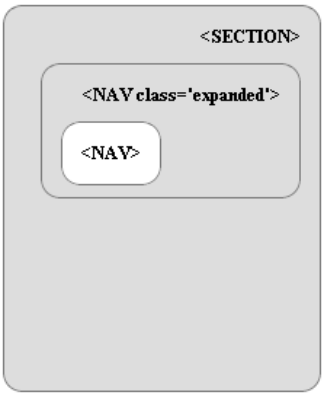
This simple wireframe diagram illustrates how this passive interface works:



Tapping any gray area collapses the `nav` element, controlled by a handler attached to the `section` tag. Tapping the white portion of the screen is the only way to override this default function.

Once you do so, tapping the `nav` in its expanded state passes the event to the default handler, which collapses it. Only by tapping newly revealed nested interface elements can you override this default function:

(C) Copyright Nokia 2013. All rights reserved.



Adding More Options

The rest of the interface consists of repeating the same basic mechanism: expanding panels to successively reveal nested items that can also be set to expand.

A handy way to distribute icons within the expanded panel is to rely on text justification. The icons are represented here as an additional series of nested `nav` items, followed by a hidden `span` that forces them to justify:

```
<nav id="icon">
  <nav id="toc"></nav>
  <nav id="share"></nav>
  <nav id="find"></nav>
  <nav id="pref"></nav>
  <span class="force"> </span>
</nav>
```

With the outer `nav` set to justify (`text-align:justify`) and the inner ones set to arrange horizontally like text (`display:inline-block`), the appended `span` is styled to take up a full line of its own, thus causing the previous items to distribute evenly:

```
span.force { margin-left : 100%; }
```

Note that we've already seen a transition applied when toggling the top-level navigation panel. An additional nested transition affects elements *within* the panel as the top-level element toggles its className, resulting in a simultaneous fade and slide-in effect:

```
#icon > nav {
  -webkit-transform   : translateX(300px);
  -webkit-transition  : all 1s;
  background-position : 1000% 50%;
  background-repeat   : no-repeat;
  background-size     : 80%;
  display             : inline-block;
  opacity             : 0;
  height              : 48px;
  width               : 48px;
}

#icon.expanded > nav {
  -webkit-transform   : translateX(0);
  background-position : 0% 50%;
  opacity             : 1;
}
```

Each navigation option, in turn, receives its own handler to expand it:

```
ui.opts = document.querySelectorAll('#icon > nav').toArray();
ui.opts.forEach(function(l){ l.addEventListener('click', ui.expandOpt) });
```

In the example above, the `toArray()` method is a local modification to the `NodeList` object the `querySelectorAll()` function releases. It allows you to run the JavaScript 1.6 `forEach()` function easily over the resulting array of elements:

```
NodeList.prototype.toArray = function() {
    for(var arr=new Array(),i=0,l=this.length;i<l;i++){arr.push(this[i])}
    return(arr);
};
```

Read more about JavaScript support in Nokia Browser 7.3.

As is true for the top-level navigation item, the handler that expands the secondary options keeps the event from percolating up to the default handler that collapses the top-level item. It also collapses other options and resizes the box to accomodate different kinds of content:

```
ui.expandOpt = function(event) {
    event.stopPropagation();
    ui.opts.forEach( function(l) { l.className = '' } );
    event.currentTarget.className = 'expanded';

    (this.id == 'toc' || this.id == 'pref')                                  ?
      (ui.nav.style.height = (screen.availHeight - (ui.offset * 2)) + 'px' ) :
      (this.id == 'find' || this.id == 'share')                              ?
      (ui.nav.style.height = '110px' )                                       :
```

 (C) Copyright Nokia 2013. All rights reserved.

```
      alert('failed case')                                                    ;
}
```

Each nested interface panel can then be placed within a `div` following each option.

This CSS transition makes each panel simultaneously flip and fade in:

```
#icon > nav + div {
  -webkit-transition : all 0.5s;
  -webkit-transform  : scaleY(0);
  opacity            : 0;
}

#icon > nav.expanded + div {
  -webkit-transform  : scaleY(1);
  opacity            : 1;
}
```

At the same time, the fact that the top-level icon's `transition-property` is set to `all` means that the JavaScript handler's local customizations to its `height` property also transition smoothly.

In this example, the interface panels accompanying each secondary navigation option include a search field, a small set of sharing options, and tab-style navigation within the site.

Accordion Sliders

The additional level of accordion-style navigation to nested headings is implemented in much the same way as the options used to access it. A touch handler toggles the className, and makes sure other top-level headings are collapsed:

```
ui.heads = document.querySelectorAll('#accordion > dt').toArray();
ui.heads.forEach(function(l){ l.addEventListener('click', ui.expandHead) });

ui.expandHead = function( event ) {
    event.stopPropagation();
    var selected = event.currentTarget.className;
    ui.heads.forEach( function(l) { l.className = '' });
    selected ? (event.currentTarget.className = '' )
             : (event.currentTarget.className = 'expanded');
};
```

The rest is handled with CSS. In this case, the accordions are implemented as data lists, with `dt` tags representing top-level headings, and `dd` tags containing lists of nested headings.

The top-level headings rely on CSS3's support for multiple background images. The first, a left-aligned icon, is toggled between the two states, while the second, a background gradient, remains constant:

```
#accordion > dt {
    background-image:
      url(img/icon_head-collapsed_x13.png),
      -webkit-gradient(linear, center top, center bottom, from(#aaaaaa), to(white));
    background-repeat   : no-repeat  , no-repeat;
    background-position : 0.25em 50% , 0 0;
    background-size     : auto       , auto;
}

#accordion > dt.expanded {
    background-image:
      url(img/icon_head-expanded_x13.png),
      -webkit-gradient(linear, center top, center bottom, from(#aaaaaa), to(white));
}
```

Read more about CSS3 Gradients and Backgrounds.

The change above occurs abruptly, while the change to the adjacent sibling element relies on the same flip-in, fade-in transition we saw before:

```
#accordion > dt + dd {
    -webkit-transition : all 0.5s;
    -webkit-transform  : scaleY(0);
    opacity            : 0;
    max-height         : 0;
}

#accordion > dt.expanded + dd {
    -webkit-transform  : scaleY(1);
    opacity            : 1;
    max-height         : 300px;
}
```

In this case, manipulating the `max-height` property allows the dimensions of the area containing the nested headings to vary freely. Additional styling allows the nested headings to be more easily selected as a block element, so that the finger can tap outside the text:

```
#accordion li > a { display : block }
```

For top-level headings, however, tapping outside the link text causes subheadings to expand.

Conclusion

As you can see, applying CSS transitions to successive layers of interface elements is quite easy, and facilitates engaging mobile user experiences. CSS

(C) Copyright Nokia 2013. All rights reserved.

transitions (and related keyframe animations) help provide mobile users in particular visual hints informing them how an application is responding to their input. Without them, shifts from one interface state to another can be abrupt, and users can lose context.

Read more about CSS3 Keyframe Animations.

You can expand on the basic approach described here to allow users to access account information, related items of interest, links to mobile Web Apps, or anything else that might appear along the margins of a desktop-formatted web page. You can also implement the same interface options using a more familiar mobile "drill-down" navigation from one screen to another. The benefit of this alternative approach, however, may be to unify site-wide options within a single ubiquitous interface element, one that can be styled with its own look and feel that adds a distinctive character to your mobile web-based application.

Download

Nd_mobile_interface.zip is a zip file with HTML and images of this tutorial. CSS and JavaScript are embedded in index.html.

(C) Copyright Nokia 2013. All rights reserved.

 (C) Copyright Nokia 2013. All rights reserved.