Slidy - a web based alternative to Microsoft PowerPoint

Dave Raggett

W3C/Volantis

dsr@w3.org
© 2006 Dave Raggett

Abstract

HTML Slidy is an open source Web-based alternative to Microsoft PowerPoint based upon XHTML, CSS and JavaScript, and which runs on a wide variety of browsers. I will introduce Slidy, and describe the challenges faced in developing an accessible cross platform browser-based editor for slide presentations.

Slidy is available at http://www.w3.org/Talks/Tools/Slidy/

Introduction

Slide presentations are a familiar part of life. In Victorian times, presenters were able to project images of glass slides and opaque objects using a device called an epidiascope. In the last century these were superseded by 35mm slides and the overhead projector. As computers and inkjet printers become more affordable, computer generated slides took over from hand drawn foils. This era came to an end with the spread of the video projector which displays slides directly from the computer. Today Microsoft PowerPoint is omnipresent. The first version was released in 1987 for the Apple Macintosh and used to produce black and white overhead transparencies. Microsoft bought the company that produced it (Forethought) and adapted it for use on Windows, and PowerPoint has been part of the Microsoft Office suite since 1990.

One of the problems with PowerPoint is the difficulty of making your slides available to others. The files tend to be very large and many people are suspicious

of binary email attachments. Another problem is that the format is proprietary to Microsoft so that you are dependent on one company for the software needed to produce and view the slides. This paper describes an alternative that is based upon HTML and which, unlike PowerPoint, works across platforms, and on most recent Web browsers such as Internet Explorer, Firefox, Opera and Safari. To share your presentation with others, you just need to give them the URI, either in email, or as a link on your website.

HTML Slidy

The Web has been used for presentations for some years. The W3C staff, for instance, have used a tool called slidemaker that splits a single HTML file into a number of files, with one file per slide, with graphical buttons for links to the previous and next slides, and the table of contents. Each time you edit the presentation, the tool has to be re-applied to update the slides. This encouraged people to search for alternatives. Opera Software provided one solution through the use of CSS to present HTML as a series of pages. Opera Show makes use of the CSS @media feature to apply page breaking rules when the Opera browser is in the projection mode (entered by pressing F11). Unfortunately Opera is one of the very few browsers to implement support for CSS in this way.

Another approach is to make use of Web page scripts to simulate paged media. This was pioneered by Tantek Çelik and soon followed by others. Eric Meyer produced the excellent S5 tool, and I independently came up with HTML Slidy. Both of these make use of JavaScript to hide and show HTML `div` elements that enclose the markup for each slide. The appearance of slides is specified via CSS.

The Slidy script and style sheet are freely available for anyone to use and to adapt under W3C's software licensing and document use policies. If people use their own markup and style sheets, but link to the Slidy script on the W3C website, then bug fixes and enhancements will be available to them automatically. This is one of the major benefits of Web applications, applications that are accessed with a Web browser over the net, as update and maintenance of such applications can be done without distributing and installing software on potentially very large numbers of clients. A second benefit is that by holding the user's data on the Web, you don't lose anything if your computer breaks down, or is mislaid or stolen. A third benefit is that Web applications are often free which compares very favourably with software for the Windows platform.

In the next section of the paper, I will introduce the features supported by Slidy. Subsequent sections will deal with recent work on developing a browser based

editor for creating and updating slide presentations, and the opportunities for using Slidy together with streaming audio for distributed presentations and later playback.

Features

A Slidy presentation is created as an HTML file, my personal preference is to use XHTML Strict. The document head contains a link to the Slidy script, and to the style sheet used to theme the slides. You can also use a `meta` element to specify a copyright statement, which is shown on the Slidy toolbar at the bottom of the window. The document body generally starts with the definition of any slide backgrounds, and the cover or title slide.

Each slide should be marked up with a `div` element with the class “slide”. You may include other class values using a space separated list on the class attribute. This allows you to use CSS rules to give slide specific styles. The font size is automatically adjusted by the Slidy script according to the width of the browser window. This is done by setting the *font-size* property on the `body` element. As a result, you shouldn't set the font-size on the body element in any accompanying style sheets. Instead, you are free to set relative font sizes on other elements, e.g.

```
div.slide.packed { font-size: 80% }
```

which reduces the font size on slides with the class value of “packed”. People vary in how much content they have on their slides. Some people have very little while others cram their slides with lots of detailed material. Slidy allows you to specify this via a `meta` element in the document head as an alterative to overriding the linked style sheet with a local rule.

Sometimes, the style sheet isn't quite enough when it comes to styling the slide's background. To support this, Slidy allows you to specify backgrounds with `div` elements with the class “background”. Using markup allows you to work around limitations in CSS, for instance, allowing you to stretch an image to just cover the width and height of the slide. You can also compose the background from a number of image and textual elements, using CSS to position them as desired. In most cases a single background is sufficient, but occasionally you may want to provide different backgrounds for different types of slides. Slidy enables you to do this by matching slides and backgrounds with common class values.

You are free to choose what markup you need for each slide. It is good practice to start the slide with a slide heading using an `h1` element. Slidy uses these headings to automatically construct a table of contents. This can be shown by clicking on the “contents” link on the toolbar at the bottom of the window, or by pressing the

"C" key. Users can navigate between slides in pretty much the same way as with PowerPoint. To advance to the next slide, you can press the space bar, click with the mouse, press the cursor right key or the page down key. To go back a slide, you can press the cursor left key or the page up key. The home and end keys take you to the first and last slides, respectively.

Slidy has evolved in response to end-user feedback, particularly amongst the W3C staff. One example is the support for incrementally revealing slide content. Users expressed a desire for this, but not for fancy build effects or slide transitions. An element is revealed incrementally if it has the class value of "incremental" or if its parent has. Users expressed a gripe with PowerPoint when moving forward or backward through incremental content in that there was no way to skip over incremental content to the next or previous slide. Slidy allows you to do this with the page up and page down keys. Cursor left and right always honour incremental content.

You can incrementally reveal overlapping images, but for the moment at least, you need to remember to use GIF transparency for this, as IE6 has buggy support for translucency in PNG images. This is due to be fixed in IE7. To make images scale with the browser window, you can use CSS to set percentage widths. I have also found that CSS positioning is simpler and more reliable across different browsers than using border-less tables. For resolution independent scaling, use [SVG](), but remember to provide a bitmapped image as a fallback. SVG support

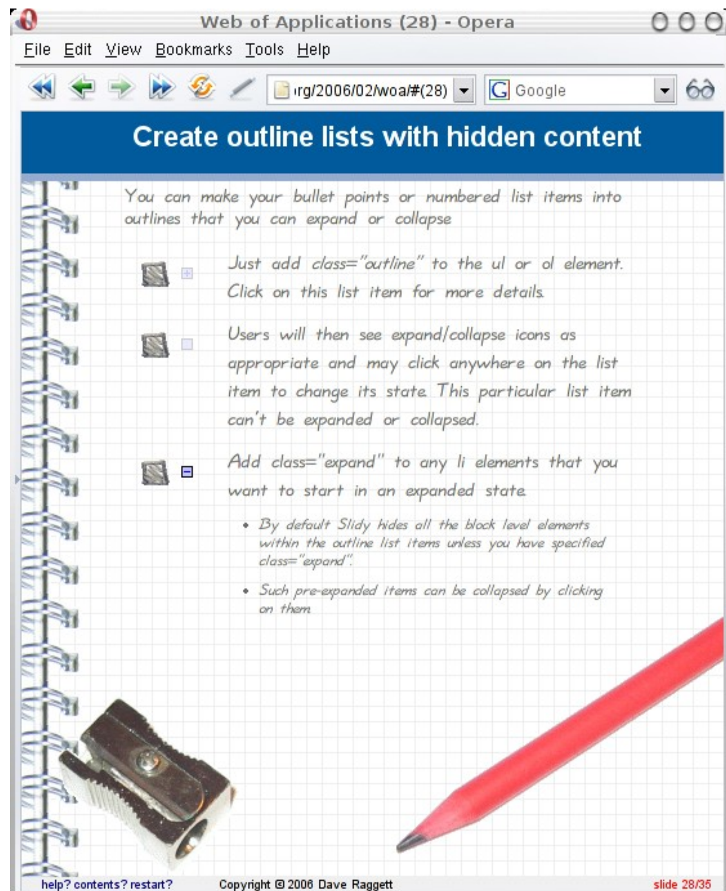

Figure 1: screen shot of slide describing how to create outline lists that expand and contract.

is built into Firefox 1.5 and Opera 8. On other browsers, you may be able to use the Adobe SVG plugin.

Another piece of user feedback concerns itself with the difficulty of dealing with structured markup in “what you see is what you get” (wysiwyg) document editors, most of which focus on paragraphs. As a result some users said that they found it hard to create the `div` element for each slide. My solution was to enhance the script to add the `div` element automatically when missing. This assumes that each such slide begins with an `h1` element.

A recently added feature provides support for outline lists with expanding and collapsing list items. You just need to add the class value “outline” to the `ul` or `ol` element. Slidy will then hide block-level elements within the list items. You can force Slidy to pre-expand a given list item by giving it the class value of “expand”. This feature makes use of CSS to show expand/collapse icons, by setting images as part of the `li` element's background, with *padding-left* used to offset the element's content. The same technique can be used to style list bullets, with the default bullets being suppressed through *list-style: none*.

A Browser-based Slidy Editor

While some users are comfortable with editing HTML, others would prefer not to have to learn the details of markup and prefer to use HTML editing tools. One of these is the open source Amaya browser/editor. As mentioned early, wysiwyg editors can make it harder for people to create structured markup. Another issue is the ease in which you can create and update files on your Web server. I therefore decided to work towards releasing a browser-based editor for Slidy, that would be usable in any modern Web browser, without the need for any plugins, and with the goal of being at least as usable for most purposes as PowerPoint.

Basic use of HTML Forms

My starting point was to look at what capabilities are available on any Web browser, and that leads to the basic use of HTML Forms. The idea is to allow you to edit one slide at a time, with the heading in a one field and the slide content in another. The form would include buttons to move forwards and backwards between slides, and the means to open an existing presentation or to create a new one.

The main problem is that the standard HTML form fields do not support rich text editing. You therefore need to make of plain text conventions that the server-side script can translate into the desired HTML markup. One such set of conventions is that for wiki markup. This identifies headings by surrounding the heading text by equals signs, with the number of such signs indicating the level of the heading. Hypertext links are represented by enclosing the URI and the caption in square brackets. Paragraph breaks are indicated by blank lines.

Problems start to creep in when you use lists. Each list item starts with an asterisk followed by a space and the list item content. Unfortunately the content is restricted to a single line as a line break marks the end of the list item. Nested items are indicated by starting the line with one or more spaces. This raises a usability problem for entering text with the HTML `textarea` element. When you type to the end of the line, the text may automatically wrap to the next line, or without being aware of it, you may press the Enter key to move to the next line. There is no visual indication either way of what happened. This causes usability problems for conventions relying on single line breaks.

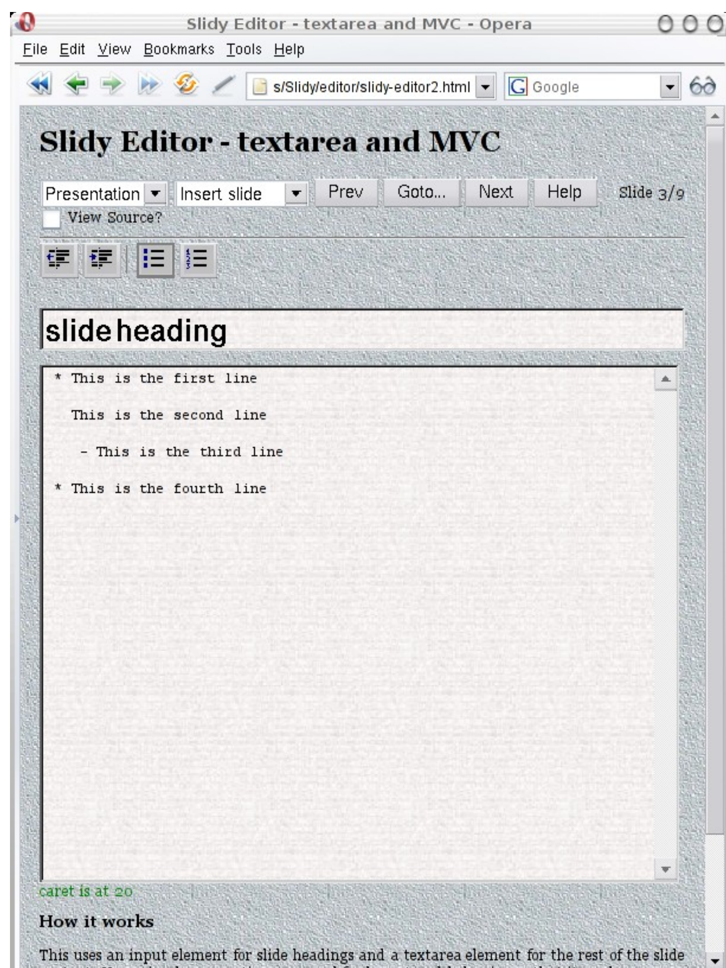

Figure 2: screen shot of plain-text editor with separate slide title and slide contents.

One solution is to rely on leading white space to indicate the level of nesting and on blank lines to indicate paragraph breaks. If the text flows onto the next line, that is treated as part of the current item. If you want to start a new paragraph or list item, you need to insert a blank line.

The corresponding server-side script can be written in a language such as Perl or Python. It needs to be able to open HTML files, to parse them into a DOM tree, and to be able to update them as appropriate and then write the file back. Parsing HTML is easy if it is well formed, but all to often it is not, for example, when the presentation has been edited using a plain text editor that knows nothing about HTML. My HTML Tidy tool can be used to correct such problems.

So far, so good. we can use HTML forms together with simple conventions for representing markup in plain text, and a matching server-side script. The snag is that the very users that don't like editing HTML markup, are unlikely to want to learn yet another markup language. They instead want some kind of “what you see is what you get” editing. How could this be provided with HTML forms?

The Model-View-Controller Design Pattern

Even though the HTML textarea element is limited to plain text, it should be practical to use JavaScript to override the default behaviour to provide a rough approximation to what you see is what you get. Think of this as what you see is roughly what you get (wysirwyg). Here is an example of a nested list:

```
* The first point
    - A subsidiary point
* The second point
```

At first glance, this looks rather like the wiki style conventions described earlier on. The difference is that when the text wraps to the next line it is automatically indented to the current level. When the Enter key is pressed, the next list item is started and the leading white space and bullet symbol inserted automatically. The reverse needs to happen when the user presses the Backspace key at the start of a list item or paragraph. To achieve this, the editor needs to override the default treatment of each keystroke.

An important step in realizing this is to separate the presentation from the markup being edited. This is where the model-view-controller design pattern is valuable. The view is the `textarea` element, the model is the markup, and the controller is some script that sits in the middle mediating between the model and the view. This requires support for some basic operations:

- A means to intercept and override the default behaviour for keystrokes including Tab, Backspace, Delete, Enter, Space, Cursor Left, Cursor Right, Cursor Up, Cursor Down, and other navigation keys.

- The means to determine the current position of the text editing caret, and to be able to set it, for example, after the user has clicked on the text.
- A means to be able to insert and remove text strings from the view.
- A means to determine the text selection when the user has selected a range of text, e.g. using a mouse drag select operation.
- A means to determine when to wrap text within the view. This requires a determination of how many fixed pitch characters fit on a line.

The above sound straightforward but in fact are quite tricky given limitations in the browser scripting interfaces and differences between browsers. An example is the difficulty in determining the caret position on Internet Explorer, which fails to provide a direct mechanism to achieve this. A workaround is to copy the text buffer for the textarea element, then to insert a short string, to search for it, and to then restore the buffer. This relies on the string not being already present in the buffer, but that is something you can test for. Another workaround, but this time for Firefox, was needed to measure the width of a character, and involved using a dummy element with its visibility set to hidden.

DesignMode and ContentEditable

Microsoft introduced rich editing support into Internet Explorer some years ago, but along with XMLHTTP, it has taken a while for people to learn how to exploit these powerful features. DesignMode is a property you can set on an HTML document to make it editable. This is usually done within an `iframe` element. Editing controls can then be placed in the document containing the `iframe` element. Design mode documents support an execCommand method that can be used to implement a range of styling operations as well as support for cut, copy and paste, undo/redo, and for inserting hypertext links and images etc. DesignMode is also supported by Firefox 1.5 and Opera 9. ContentEditable is currently only supported by Internet Explorer, and is like DesignMode, but can be applied to individual HTML elements. This avoids the need for a iframe and makes it easier to work with.

A number of people have demonstrated how to use these features to implement simple HTML editors. Some examples include WidgEditor and FCKeditor, both of which are open source and intended to be included as part of other Web applications. There are also non-free commercially supported equivalents. The prospects for a wysiwyg editor for Slidy seem very promising. What are some of the pitfalls? Clearly, the solution needs to be accessible. This means that it should be possible to drive the editor from the keyboard without the use of a mouse, trackpad or other pointing device. In practice this constrains the use of images as controls. It is very simple to add an event handler to an image for mouse clicks, but by itself this doesn't allow the control to be activated by the keyboard. A solution is to enclose the image in a button element, as this then means that it will become part of the tabbing sequence. Another idea is to use keystroke event handlers to implement short cuts for activating the controls. A combination of the two approaches gives new users an obvious solution whilst offering experienced users a short cut.

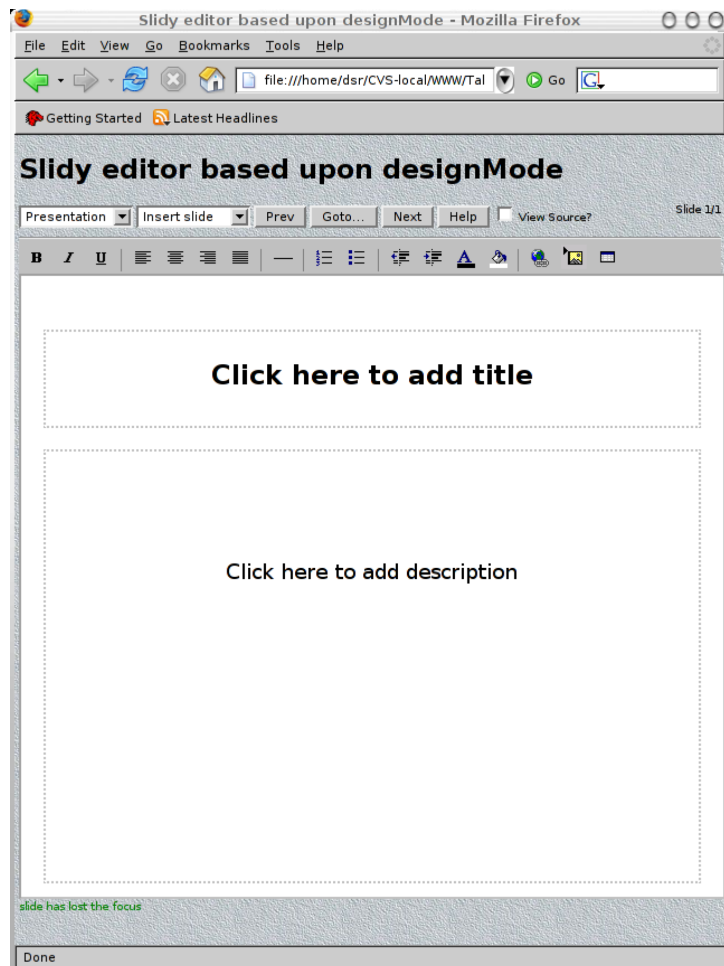

Figure 3: screen shot of design mode editor with template for a cover slide.

Another pitfall is the quality of the markup that is generated when using designMode and contentEditable. As an example, Firefox inserts a pair of <br> elements instead of inserting a paragraph <p> element. It is very hard to place a paragraph within a list item, and you tend to get a separate paragraph with a style attribute that indents the paragraph. One solution is to work around this when it comes time to exporting the edited markup. This involves applying a number of clean up

rules. The markup can also be messed up when the user performs a copy and paste operation that doesn't respect the nesting of the document structure.

Another solution is to apply the model-view-controller design pattern and to completely override the browser's built-in behaviour. This is harder to implement. but offers much greater control. I have been exploring both approaches.

Template based editing

What should a Slidy editor look like? An obvious starting point is to look at the user interface for PowerPoint. This essentially allows you to edit presentations one slide at a time. When it comes to creating a new slide you are invited to pick from a smaller number of template slides. Like Slidy, you can also pick the backgrounds through the slide theme and customize them as needed. I therefore decided to copy this approach. For the designMode editor, the templates are defined as separate XHTML files along with their own style sheets. On the title slide, users are invited to supply a title and a description. On the template for a regular bullet list, there are fields for the slide's title and another for the list.

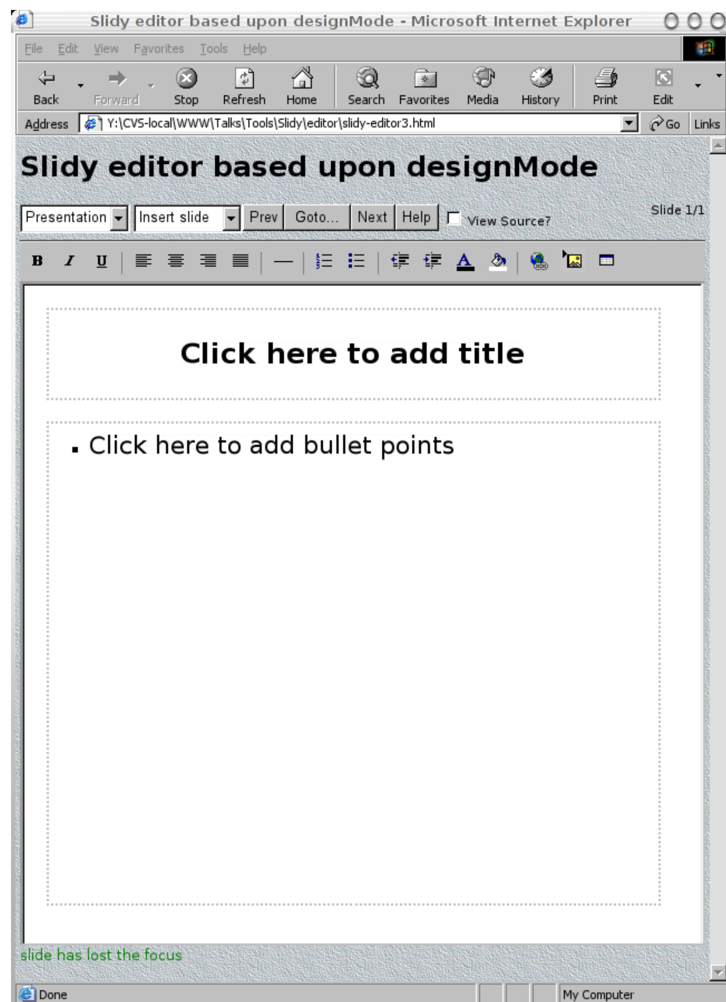

Figure 4: screen shot of design mode editor with template for slide with title and bullet list

On a raw designMode implementation, it is very easy to accidentally delete the heading or the list or other fields in the template. That creates usability problems. Internet Explorer's contentEditable

provides a solution as you can only edit the contents of elements for which the contentEditable attribute has been set to true. It is unfortunate that no other browsers at present support this feature. I hope to workaround this by careful handling of the Backspace and Delete keystrokes, as well as the cut operation. To support this, I use the contentEditable attribute in the markup for template slides. This is not yet a part of the W3C standards for XHTML, but this is under consideration. It may be the case, that rather than using an attribute, people may feel more comfortable about specifying this as a new CSS property, but we shall have to wait and see.

Another consideration is the frequency with which changes are synchronized with the Website. Some possibilities include, when you save the presentation as a whole, each time you move from one slide to another, or dynamically as you type. The latter could be supported through AJAX, using HTTP keep-alive for reduced latency. This is something the end-user feedback will be needed to decide on.

PowerPoint also provides integrated support for importing and editing graphics. For the moment, the Slidy editor is limited to inserting graphics that have already been uploaded to a Website. For the future, I hope to explore the use of JavaScript and SVG as a means to implement browser-based editing of graphics. It may be possible to constrain this in such a way as to exploit VML on Internet Explorer given its similar functionality to SVG Tiny, and one possibility is to exploit AJAX to access server-side translation services that map between the two.

Other opportunities for extending the Slidy editor include work on filters for importing PowerPoint presentations. This could leverage the work done on import filters for Open Office, an open source alternative to Microsoft Office. Such filters would need to run server-side due to their complexity, but would enable you to upload a PowerPoint file to a Website and to get back a URI for the corresponding Slidy presentation. The filter would need to unpack the images, and to generate the style sheet and markup files.

Slidy, AJAX and Streaming Audio

When people give presentations, what they say is just as important as what is shown on the slides. Unfortunately there is currently no easy way to capture the audio for later playback together with the slides. I intend to provide a cure for this using a browser plugin that supports audio streaming. Here is how it works.

The plugin provides a means to capture audio from the microphone pickup and to stream it to the server via the real-time streaming protocol (RTP). To reduce bandwidth and storage requirements, the audio can be encoded in an open

source codec such as iLBC or Speex. To start with, the Web page script uses AJAX to make an HTTP request to the Web server for a port for the RTP connection. This request also provides the URI for presentation and allocates files to log the slide transitions, and to store the audio samples sent via RTP.

The Web page script then passes the RTP port to the plugin and starts the capture mode in which audio is streamed to the server. The user is then able to start talking. When he or she moves to the next slide, the time at which this occurred is logged to the server via XMLHttpRequest. At the end of the presentation, the server is told to close the files and the streaming is terminated.

To replay the presentation at a later date, the plugin is used in playback mode. The process starts with the Web page script downloading the slide transition log file. This starts with the name of the audio file used for capture. This name is then passed to the Web server along with a request for a RTP port for the playback stream. The Web page then commands the plugin to start playback. The times and details of slide transitions are obtained from the log file. The Web page script then switches slides at the same point in the audio stream as happened in the original presentation. The user can then sit back and pay attention as the presentation unfolds.

But that's not all! The user may decide to skip backward or forward within the presentation. Slidy is then able to re-synchronize the audio stream by sending HTTP commands to the server to seek to the appropriate time from the start of the presentation, as given by the log file. Note that the approach is designed to work even if the user accesses the Internet through a Network Address Translation (NAT) device as is commonplace for home users.

Using Slidy for Distributed Presentations

A similar set up can be used to provide remote access to live presentations. In this case, the Web server provides a means for participants to connect to the presentation. This involves them in opening the link to the presentation, causing their bowser to download the slides. The Web page script then requests the RTP port for the audio playback and commands the audio plugin to initiate streaming from that port. At the same time, the script opens an HTTP connection to the server to listen for slide changes. This could take advantage of HTTP keep-alive to reduce the latency between the presenter moving to the next slide, and having this take effect on the participant's browsers. The presenter's set up is very

similar to that used for recording a presentation. The difference is that this time, the server copies the audio stream out to all of the participants, along with the slide transitions.

To allow the presenter to hear back from the participants, the audio is streamed in both directions. If only one participant is speaking at any given time, then the server can copy the audio directly without the need for decoding (for iLBC each audio frame is encoded independently). If several participants speak at the same time, the server can either pick one, or can pick the "top" few, in which case, the audio needs to be decoded, mixed together and re-encoded. It should be straightforward to provide people with an indication when they are speaking at the same time as others.

With audio capture and playback occurring simultaneously, there is a possibility of feedback and echoes. A simple way to avoid that is for everyone to use a headset of some kind. Another idea is to use a push to talk mechanism that cuts audio playback while the button is activated. A further possibility is for the plugin to support acoustic echo cancellation. An open source solution for this is available from Andre Adrian's web site, and another is integrated as part of Speex since version 1.1.9.

There is tremendous promise for using Slidy and AJAX as components of Web-based solutions for remote meetings. Other components will provide support for shared minute taking, for text-based chat between participants, and for meeting management as exemplified by W3C's Zakimbot. I am also looking forward to trying out ideas with integrating server-based speech recognition and synthesis services. If you are in a position to help with any of this, please contact me.

Acknowledgements

I would like to thank the many people who have contributed to Slidy, including my daughter who came up with the name. I am grateful to Canon and more recently Volantis for their financial support in my role as W3C Fellow whilst doing this work.