OpenEyes - EyeDraw

Editors: G W Aylward
Version: 0.90:
Date issued: 1 November 2012

Target Audience

General Interest	
Healthcare managers	
Ophthalmologists	
Developers	✔

Amendment Record

Issue	Description	Author	Date
0.9	Draft	G W Aylward	1 Nov 2012

Table of Contents

Introduction

This document describes the use of EyeDraw, the OpenEyes drawing package. It describes in detail how to create a drawing on an HTML page using HTML elements and javascript. A companion document describes the EyeDraw widget, which makes the use of EyeDraw in web applications using the Yii framework easier

The latest version (2.0) of EyeDraw includes a number of improvements, including a notification system, easier initialisation of parameter values, animation, binding to HTML elements, synchronisation between drawings, and better error reporting.

Usage

For most of the sections below, a fully commented and working HTML file is provided in the with a name corresponding to the title of the section

Basic usage

An EyeDraw instance can be created using the following code which runs on page load (see the file testDoodle.html which contains the basic structure to run the examples which follow);

```
function init()
{
    drawingEdit = new ED.Drawing(document.getElementById('canvasEdit'),
ED.eye.Right, 'RPS', true, {graphicsPath:"../graphics/"});
}
```

There are four mandatory arguments;

- canvas: A reference to the canvas element linked to the drawing object
- eye: right or left eye
- IDSuffix: A string which uniquely identifies the drawing on the page
- isEditable: A boolean which specifies whether the user can edit the drawing or not

In addition, the constructor accepts an array of optional arguments:

- offset_x : Offset used to calculate pointer position (default value 0)
- offset_y: Offset used to calculate pointer position (default value 0)
- to_image: Boolean indicating whether to convert the drawing to an image (default value false)
- controllerFunctionName: Name of the main controller function (default value 'eyeDrawController')
- graphicsPath: Relative path to the folder containing graphics files (default value 'graphics/');

This code as it stands will generate the following error:

```
EYEDRAW ERROR! class: [ED.Drawing] method: [Constructor] message: [Expected
controller function: eyeDrawController does not exist]
```

This is because a main controller object is expected, and this needs to be provided, as in the following code;

```
function eyeDrawController(_drawing)
{
	// Function called for notification
	this.notificationHandler = function (_messageArray)
	{
		switch (_messageArray['eventName'])
		{
			case 'loaded':
				_drawing.addDoodle('AntSeg');
				break;
		}
	}
}
```

The name of the controller defaults to ‘eyeDrawController’, but this can be changed using an optional parameter for the ED.Drawing constructor call. The internal notification handler contains a switch case construct, which in this example will respond only to the ‘loaded’ notification, which is issued when all the image files required by EyeDraw for patterned fills, are successfully loaded. A full list of notifications will be found in Appendix one. In this example, once the images are loaded, a new doodle of the ‘AntSeg’ class is added to the drawing.

Initialisation of doodle parameters

The ‘AntSeg’ doodle has a derived parameter called ‘pupilSize’ which has three possible values (Large, Medium, Small). The initial value (Large) is defined as part of the doodle subclass, but this value can be set by passing an associative array as an optional parameter of the addDoodle function like this:

```
_drawing.addDoodle('AntSeg', {pupilSize:'Small'});
```

Any number of parameters (simple or derived) can be set using this mechanism.

Binding to HTML elements

Many real world examples in OpenEyes require that a parameter of a doodle reflect the value of an HTML element, and vice versa. For example, the following text element displays a possible value for the pupilSize parameter.

```
<input type="text" id="pupilSizeInput" value="Medium" />
```

Previous versions of EyeDraw required a great deal of glue code to syncronise the value of the parameter and the HTML element, but this can now be achieved using a simple binding, which is an associative array passed as a second optional parameter in the addDoodle command:

```
_drawing.addDoodle('AntSeg', {pupilSize:'Small'}, {pupilSize:'pupilSizeInput'});
```

Note that there is a conflict in the initial value of the pupilSize parameter between the initialisation array and the binding array, since the value of the bound element is set to ‘Medium’. In these circumstances, the binding array always takes precedence, so in this example, the value of the parameter is set to the initial value of the HTML element.

The binding function also provides animation (if the doodle allows it), so that changing the value of the bound element will result in the change occurring in the diagram with an animation.

Validation is also built in, so changing the value of the bound element to an invalid value will result in it being changed to a valid one, according to rules which vary according to the type of parameter. In this case, the value simply reverts to the value corresponding to the diagram.

Binding to deletable doodles

Sometimes there is a requirement to bind an element to a doodle which is deletable, and therefore might be deleted. The behaviour of the element under these circumstances needs to be defined, and this is done by passing an associative array to the drawing object linking the id of the element with the value corresponding to the absent doodle. For example, the 'CorticalCataract' doodle has a parameter indicating severity called 'grade', and this can have the value Mild, Moderate, or White, as in the following select element;

```
<select id="gradeSelect">
    <option value="None" selected="true">None</option>
    <option value="Mild" >Mild</option>
    <option value="Moderate">Moderate</option>
    <option value="White">White</option>
</select>
```

However, an additional option of 'None' is also required, corresponding to the absence of the doodle. The binding needs to support the following behaviour;

- Deleting the doodle should result in the value of the select element being set to 'None'
- Setting the select element to a value of 'None' should result in the doodle being deleted
- Setting the select element to a value other than none should create a doodle if one does not exist

All this behaviour is supported by the binding mechanism, though this has to be applied in a different way, to allow for the possibility of the initial conditions having no doodle present (so the addDoodle method cannot be used). An associative array is passed with the key corresponding to the Class name of the doodle to be bound to, and the value being another associative array linking parameters to the id of the bound element.

The drawing object needs to know what value of the bound element is associated with a deleted doodle, and this is passed using the addDeleteValues method, which takes an associative array as a parameter, in this case with a single key:value pair linking the element id and the value to be associated with the deleted doodle:

```
case 'loaded':
    _drawing.addDoodle('AntSeg');
    _drawing.addBindings({CorticalCataract:{grade:'gradeSelect'}});
    _drawing.addDeleteValues({gradeSelect:'None'});
    break;
```

The binding should now behave as expected.

Synchronisation

Two or more drawings can be set up so that parameters are synchronised between them. First of all, set up two canvas elements by adding the following to the HTML.

```
<canvas id="canvasEdit2" class="ed_canvas_edit" width="300" height="300"
tabindex="2"></canvas>
```

Another drawing is instantiated using the following additional line in the init code:

```
var drawingEdit2 = new ED.Drawing(document.getElementById('canvasEdit2'),
ED.eye.Left, 'LPS', true);
```

Remove the binding code and revert to a simple addDoodle command so that an AntSeg doodle is loaded into each drawing

```
case 'loaded':
    _drawing.addDoodle('AntSeg');
    break;
```

Synchronisation information is held in an associative array, which should be defined in the controller function as follows. The array is effectively saying to the RPS drawing, “Sync any ‘AntSeg’ doodle with an ‘AntSeg’ doodle in the LPS drawing.

```
var syncArray = new Array();
if (_drawing.IDSuffix == 'RPS')
{
    syncArray['LPS'] = {AntSeg:['AntSeg']};
}
```

The synchronisation itself is handled by a method of the doodle subclass, but also requires code in the controller to iterate through the syncArray and call the appropriate methods. The notification handler is set up to respond to any changes in doodle parameters, by adding the case ‘parameter’. In this example, one drawing is defined as the master and one as the slave, enabling one way synchronisation.

```
// Iterate through syncArray
for (var idSuffix in syncArray)
{
    // Iterate through each specified className
    for (var className in syncArray[idSuffix])
    {
        // Get array of specified slave doodle class names
        var slaveClassNameArray = syncArray[idSuffix][className];

        // Iterate through it,
        for (var i = 0; i < slaveClassNameArray.length; i++)
        {
            // Define which drawing is slave
            var slaveDrawing = drawingEdit2;

            // Master doodle
            var masterDoodle = _messageArray['object'].doodle;

            // Slave doodle (uses first doodle in the drawing matching the
className)
            var slaveDoodle =
slaveDrawing.firstDoodleOfClass(slaveClassNameArray[i]);

            // If master is being driven, both are defined, and slave doodle is set to
sync, enact sync for the changed parameter
            if (masterDoodle && slaveDoodle && slaveDoodle.willSync &&
masterDoodle.drawing.isActive)
```

```
                        {
                                // Sync is handled by a method of the slave doodle object
                                slaveDoodle.syncParameter(_messageArray.object.parameter,
masterDoodle[_messageArray.object.parameter]);

                                // Update any bindings associated with the slave doodle
                                slaveDrawing.updateBindings(slaveDoodle);
                        }
                }
        }
}
```

Two way synchronisation is achieved by adding a syncArray to the other drawing:

```
if (_drawing.IDSuffix == 'LPS')
{
        syncArray['RPS'] = {AntSeg:['AntSeg']};
}
```

The only other change required is an additional line of code to correctly identify which is the slave drawing:

```
var slaveDrawing = idSuffix == 'RPS'?drawingEdit1:drawingEdit2;
```

Synchronisation can be switched off by setting the willSync property of the slave doodle to false.

Notifications

Any object can register to receive notifications from EyeDraw by calling the registerForNotifications method of a drawing object. This takes three arguments;

- The object to be registered
- The name of the object's method to be called with a notification
- An array containing a list of notifications that the object is interested in

The following example creates a new object which will respond to a change in the value of a doodle's parameter and send the contents of the received message to the console.

```
var test = new listener;
function listener(_drawing)
{
        // Register this object for notifications
        _drawing.registerForNotifications(this, 'handler', ['parameter']);

        // Function called for notification
        this.handler = function(_messageArray)
        {
                console.log(_messageArray);
        }
}
```

The message array consists of three items:

- The name of the event
- The currently selected doodle (or null if none selected)
- An additional object, the definition of which depends on the event (See Appendix one)

Appendix 1

Notifications

Event name	Description	Object properties
doodleAdded	A new doodle has been added	The newly added doodle
doodleDeleted	The selected doodle has been deleted	The class name of the deleted doodle
flipHor	The selected doodle has been flipped around a horizontal axis	Empty
flipVer	The selected doodle has been flipped around a vertical axis	Empty
hover	The mouse has been held over a doodle for the hover time	A point object containing the canvas coordinates of the pointer
keydown	A key has been pressed	the keycode of the pressed key
loaded	All required graphics files are successfully loaded	Empty
mousedown	The pointer has been clicked in the drawing's canvas	A point object containing the canvas coordinates of the pointer
mousedragged	The pointer has been dragged within the drawing's canvas	A point object containing the canvas coordinates of the pointer
mouseout	The pointer has left the drawing's canvas	A point object containing the canvas coordinates of the pointer
mouseover	The pointer has entered the drawing's canvas	A point object containing the canvas coordinates of the pointer
mouseup	The pointer has been unclicked	A point object containing the canvas coordinates of the pointer
moveToBack	The selected doodle has been moved to the back	Empty
moveToFront	The selected doodle has been moved to the front	Empty
parameter	The value of a doodle's parameter has changed	doodle:The doodle parameter: Name of parameter value: The new value