OpenEyes - EyeDraw

Editors: G W Aylward, M S Cross
Version: 1.0
Date issued: 17 April 2018

Target Audience

General Interest	
Healthcare managers	
Ophthalmologists	
Developers	✔

Amendment Record

Issue	Description	Author	Date
0.9	Draft	G W Aylward	3 Nov 2012
1.0	Draft	M S Cross	17 April 2018

Table of Contents

Introduction

This document describes the use of EyeDraw, the OpenEyes drawing package. It first provides an overview of the different components of EyeDraw, and then describes in detail how to add new drawing elements and contribute to the development of EyeDraw. Finally, this document details how to use EyeDraw to create a drawing on an HTML page.

Overview

Drawings

A drawing represents an EyeDraw instance; it consists of one HTML canvas element displaying one or more doodles.

Doodles

Doodles are the individual components that make up drawings; they have built in knowledge of what they represent, and how to behave when manipulated.

All doodles inherit default properties from the Doodle superclass. The properties that are defined in the Doodle constructor method are listed in Appendix One, with the default values.

Development

The following section describes in detail how to add new doodles to EyeDraw.

New Doodle classes

All doodles have their own class, saved in the src/ED/Doodles folder. Create a new doodle class under the correct subspecialty - for example Oph for ophthalmology doodles or Ent for Ears, Nose and Throat doodles, and name the file appropriately. The contents of an existing doodle class may provide a useful template when creating a new doodle.

In the new doodle class, create an object constructor function. This must define the class name (this.className) and any additional parameters that are unique to the doodle class. A list of the default doodle parameters can be found in Appendix One; these are inherited from the doodle superclass and do not need to be defined.

The savedParameterArray should contain all of the parameters of the doodle class that the user can edit within the drawing, and therefore need to be saved. This includes any default doodle parameters and those that are unique to the doodle class.

List all of the parameters that you want to appear in the doodle pop out in the controlParameterArray object. This is used to define the label text that you want to appear in the doodle pop out.

Finally, call the doodle superclass constructor.

The example below details the corneal suture doodle class (CornealSuture.js):

```
ED.CornealSuture = function(_drawing, _parameterJSON) {
    // Set classname
    this.className = "CornealSuture";

    // Derived parameters
    this.removed = false;

    // Saved parameters
    this.savedParameterArray = ['radius', 'rotation','removed'];

    // Control parameters
    this.controlParameterArray = {'removed' : 'Removed'};

    // Call superclass constructor
    ED.Doodle.call(this, _drawing, _parameterJSON);
```

```
}
```

The corneal suture doodle has a removed property that is initially set to false. The property appears in the controlParameterArray, and so the user will be able to edit this property in the doodle pop out. The user can also edit the radius and rotation of the corneal suture doodle within the drawing; all three of these “editable” parameters are listed in the savedParametersArray.

Next, you will need to set the doodle superclass and the constructor:

```
ED.CornealSuture.prototype = new ED.Doodle;
ED.CornealSuture.prototype.constructor = ED.CornealSuture;
ED.CornealSuture.superclass = ED.Doodle.prototype;
```

Then, override the methods of the doodle superclass to define the unique behaviour and design of the new doodle. The main doodle methods that you will need are:

- setHandles
- setPropertyDefaults
- setParameterDefaults
- dependentParameterValues
- draw
- describe or groupDescription
- snomedCode

The use of each of these methods is detailed below.

You may also need to create new methods, specific to the doodle class. For example, the rhegmatogenous retinal detachment doodle class (RRD) has its own method to calculate whether the macula is detached or spared, i.e. whether the doodle covers a particular point in the drawing canvas space.

Doodle handles

To control different doodle behaviours or properties, handles must take one of the mouse dragging modes defined in EyeDraw:

Mode	Description	Example
ED.Mode.Scale	Scale the doodle size	Blot haemorrhage
ED.Mode.Arc	Extend doodles in a circular fashion, by increasing the angle of arc	Laser demarcation
ED.Mode.Rotate	Rotate the doodle around its origin	PCIOL
ED.Mode.Apex	Change the X- and Y-coordinates of the doodle apex point	Anterior segment

Mode	Description	Example
ED.Mode.Handles	Change the X- and Y-coordinates of handle points that are saved in an array	Subretinal fluid

Define the mode of each handle that you will use in the setHandles() method. For example, the rhegmatogenous retinal detachment doodle (RRD) has three control handles: two that change the angle or arc, and one to adjust the apex point that defines the posterior extent of the detachment. When defining a new handle,

```
ED.RRD.prototype.setHandles = function() {
    this.handleArray[0] = new ED.Doodle.Handle(null, true, ED.Mode.Arc, false);
    this.handleArray[1] = new ED.Doodle.Handle(null, true, ED.Mode.Arc, false);
    this.handleArray[2] = new ED.Doodle.Handle(null, true, ED.Mode.Apex, false);
}
```

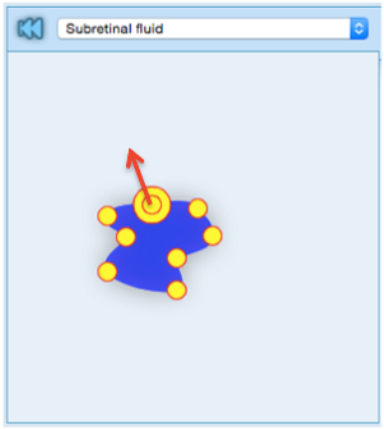
The Apex and Handles modes are very similar: if only a single control handle is needed, the apex point may be used to control and store the location of a single handle (as in the example above). Otherwise, use the handles mode and create an array to store the location points for each handle.

For example, the subretinal fluid doodle requires eight handle points to edit the doodle shape. Using a for loop, eight handle-mode handles are added to the doodle's handle array.

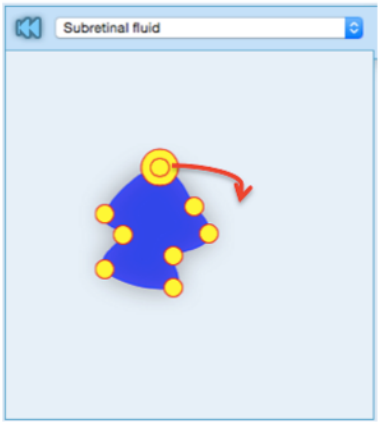
```
ED.SubretinalFluid.prototype.setHandles = function() {
    // Number of handles
    var numberOfHandles = 8;

    // Array of handles
    for (var i = 0; i < numberOfHandles; i++) {
        this.handleArray[i] = new ED.Doodle.Handle(null, true, ED.Mode.Handles, false);
    }

    // Allow top handle to rotate doodle
    this.handleArray[0].isRotatable = true;
}
```

The user also needs to be able to rotate the doodle. Even handles that do not use the rotate drag mode can be used to control the doodle rotation by setting the handle isRotatable property to true.

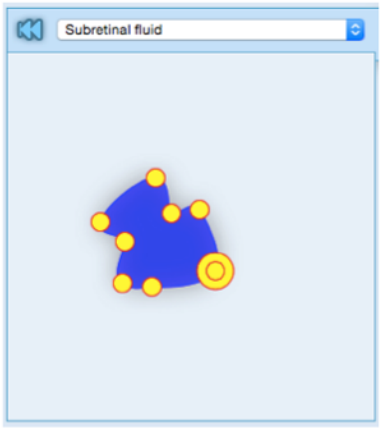
In the subretinal fluid doodle example that is given above, the first handle in the array is also rotatable. In the doodle, this will display as a handle with two rings; dragging the inner ring changes the control point location (handle mode), and the outer ring edits the doodle rotation.



Drag the inside handle ring to move the control point (handle mode)



Drag the outer handle ring to change the rotation (rotate mode)



Property defaults

You must define the initial properties of the doodle, if they vary from the superclass defaults (see Appendix One). To do this, use the setPropertyDefaults() method.

For example, the user should not be able to move or delete the Anterior Segment doodle, and there should not be more than one present in a single drawing (i.e. it is unique):

```
ED.AntSeg.prototype.setPropertyDefaults = function() {
    // Set default properties
    this.isDeletable = false;
    this.isMoveable = false;
    this.isUnique = true;
```

Then, you should also update the doodle simple parameters, if required. To do this you will need to edit the properties of the parameterValidationArray. Often, the minimum and maximum possible values of each parameter need to be specified. Use the setMinAndMax(_min, _max) range method, passing the possible minimum and maximum value.

For example, in the Anterior Segment doodle, the apex point is constrained so that the user cannot change the X-coordinate (i.e. both the minimum and maximum point is 0), and the Y-coordinate can only take a reduced range of values in the top half of the drawing canvas (-300 to -60 in the doodle plane). By default, the apex coordinates can take any value between -500 and +500.

```
    // Update component of validation array for simple parameters
    this.parameterValidationArray.apexX.range.setMinAndMax(0, 0);
    this.parameterValidationArray.apexY.range.setMinAndMax(-300, -60);
```

The default ranges for other simple parameters are listed in Appendix One. Using the setMinAndMax method, the minimum and maximum values can additionally be set for the origin, rotation, and the scale.

The movement of handle-mode control points can also be constrained. They can either be constrained between two coordinates to give a rectangular boundary, or by limiting the distance that the handle can be from the doodle origin (the length), and the clockwise angle of the handle from the origin, for when a circular boundary is required.

For example, in the subretinal fluid doodle, all of the handles are constrained within a segment of a circle, by defining a certain radii from the doodle origin to control the size of the shape, and the segment as an angles of the doodles circumference.

```
ED.SubretinalFluid.prototype.setPropertyDefaults = function() {
	// Create ranges to constrain handles
	this.handleVectorRangeArray = new Array();
	for (var i = 0; i < this.numberOfHandles; i++) {
		// Full circle in radians
		var cir = 2 * Math.PI;

		// Create a range object for each handle
		var n = this.numberOfHandles;
		var range = new Object;
		range.length = new ED.Range(+50, +290);
		range.angle = new ED.Range((((2 * n - 1) * cir / (2 * n)) + i * cir / n) % cir, ((1 * cir / (2 * n)) + i * cir / n) % cir);
		this.handleVectorRangeArray[i] = range;
	}
}
```

In the Peripapillary Atrophy doodle, there are four handles that are each constrained within a range of coordinates;

```
ED.PeripapillaryAtrophy.prototype.setPropertyDefaults = function() {
	// Create ranges to constrain handles
	this.handleCoordinateRangeArray = new Array();

	var max = this.outerRadius * 1.4;
	var min = this.outerRadius;
	this.handleCoordinateRangeArray[0] = {
		x: new ED.Range(-max, -min),
		y: new ED.Range(-0, +0)
```

```
        };
        this.handleCoordinateRangeArray[1] = {
                x: new ED.Range(-0, +0),
                y: new ED.Range(-max, -min)
        };
        this.handleCoordinateRangeArray[2] = {
                x: new ED.Range(min, max),
                y: new ED.Range(-0, +0)
        };
        this.handleCoordinateRangeArray[3] = {
                x: new ED.Range(-0, +0),
                y: new ED.Range(min, max)
        };
    }
```

Finally, in the setPropertyDefaults() method, define the additional parameters for your doodle. This adds the parameter to the validation array, so that only appropriate data will be stored for each parameter.

Parameters can be simple or derived. Simple parameters are always numeric (i.e. of type int or float); all of the default doodle parameters are simple (e.g. originX, originY). Simple parameters cannot be bound to HTML elements, and therefore should not be used for parameters to be edited using the doodle pop out. Only derived parameters can be added to the control parameter array, and edited using the doodle pop out.

Derived parameters can be of type string (i.e. a value from a drop down list), int, float, bool, or freeText (i.e. any alphanumeric string, inputted using a text box input).

You should specify the possible list of values for drop down controls (*string* type). The range can be specified for the *Int* and *Float* type parameters, and additionally the precision for *Float*. If you wish the parameter to be animated, then set this property to true. You can adjust the speed of animation by changing the delta value.

Below are some of the parameter validation arrays for the Anterior Segment doodle:

```
    // Add complete validation arrays for derived parameters
        this.parameterValidationArray.pupilSize = {
                kind: 'derived',
                type: 'string',
                list: ['Large', 'Medium', 'Small'],
                animate: true
        };

        this.parameterValidationArray.pxe = {
```

```
                kind: 'derived',
                type: 'bool',
                display: true
        };

        this.parameterValidationArray.coloboma = {
                kind: 'derived',
                type: 'bool',
                display: true
        };
```

Parameter defaults

If you need to alter the initial values for parameters, these can be set in the setParameterDefaults() method. The Anterior Segment doodle uses the setParameterFromString method, specifying both the parameter and the value as a string:

```
ED.AntSeg.prototype.setParameterDefaults = function() {
        this.setParameterFromString('pupilSize', 'Large');
        this.setParameterFromString('pxe', 'false');
        this.setParameterFromString('cornealSize', 'Normal');
};
```

Dependent parameters

Dependent parameters are parameters whose value adjusts with that of another parameter. An associative array is used to define these dependencies, using the dependentParameterValues method.

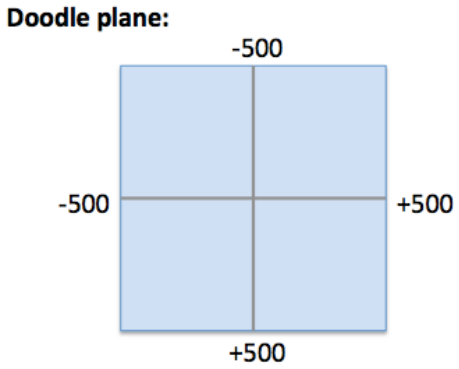
For example, the user can set the pupil size in the anterior segment doodle by dragging the apex-point control handle, or by using the pupilSize drop down in the doodle pop out. Therefore, when the user changes one of these parameters, the other must be adjusted accordingly so that they are in synch:

```
ED.AntSeg.prototype.dependentParameterValues = function(_parameter, _value) {
        var returnArray = {},
                returnValue;

        switch (_parameter) {
```

```
            case 'apexY':
                if (_value < -200) returnArray.pupilSize = ‘Large';
                else if (_value < -100) returnArray.pupilSize = 'Medium';
                else returnArray.pupilSize = 'Small';
                break;

            case 'pupilSize':
                switch (_value) {
                    case 'Large':
                        if (this.apexY < -200) returnValue = this.apexY;
                        else returnArray.apexY = -260;
                        break;
                    case 'Medium':
                        if (this.apexY >= -200 && this.apexY < -100) {
                            returnValue = this.apexY;
                        } else returnArray.apexY = -200;
                        break;
                    case 'Small':
                        if (this.apexY >= -100) returnValue = this.apexY;
                        else returnArray.apexY = -100;
                        break;
                }
                break;
        }
        return returnArray;
    };
```

Draw

Doodles are drawn in the ‘doodle plane’ consisting of a 1001 pixel square grid with central origin (i.e. -500 to +500), and negative Y upwards:

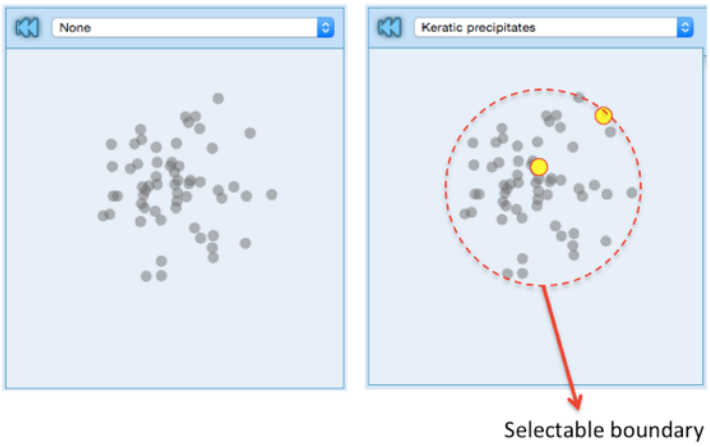


Using HTML5 Canvas, define the boundary path that outlines the “selectable” area of a doodle. Then call drawBoundary(_point), parsing the point parameter so that it can be used during a hit test

You will not need to call the canvas stroke() and fill() methods for the boundary path as they are called by the drawBoundary method.

Some doodles have an invisible boundary, that is, the boundary path outlining the area for the doodle is not part of the visible design for the doodle. This can be achieved by setting the opacity of the stroke colour to 0.

For example, the keratic precipitates doodle contains multiple small dots. An invisible circular boundary path is used to define the area that contains the dots, so that the user can select the doodle by clicking anywhere within the dot density.




```
// Keratic precipitates boundary path
ctx.beginPath();

// Define invisible circular boundary, of radius 200
var r = 200;
ctx.arc(0, 0, r, 0, Math.PI * 2, true);

// Close path
ctx.closePath();
```

```
// Set line attributes
ctx.lineWidth = 0;
ctx.fillStyle = "rgba(0, 0, 0, 0)";
ctx.strokeStyle = "rgba(0, 0, 0, 0)";

// Draw boundary path (also hit testing)
this.drawBoundary(_point);
```

If you do not require that the boundary shape is filled with a colour, set the doodle property isFilled to false when defining the property defaults or also set the fill colour opacity to 0.

Then define the non-boundary paths for the doodle, to make up the design of the doodle. For these paths, you will need to call ctx.stroke() and ctx.fill(), as appropriate.

The keratic precipitates doodle uses the drawSpot method, and is passed the parameters for the drawing context, the X-coordinate, Y-coordinate, spot radius, and fill style. Other useful doodle drawing methods are defined in Appendix Two.

```
// Non boundary paths
if (this.drawFunctionMode == ED.drawFunctionMode.Draw) {
    // Define colours
    var fill = "rgba(110, 110, 110, 0.5)";

    // Individual spot radius
    var dr = 10 * ((this.apexX + 20) / 20) / this.scaleX;

    var p = new ED.Point(0, 0);
    var n = 40 + Math.abs(Math.floor(this.apexY / 2));
    for (var i = 0; i < n; i++) {
        p.setWithPolars(r * ED.randomArray[i], 2*Math.PI * ED.randomArray[i*100]);
        this.drawSpot(ctx, p.x, p.y, dr, fill);
    }
}
```

If the doodle has control handles, next set the coordinates of each handle in the canvas plane and call the drawHandles() method.

The keratic precipitates doodle uses two control handles: one that is used to scale the doodle, and the second, defined by the apex point, used to control the spot size and number.

```
    // Define and set coordinates of handles (in canvas plane)
    var handle1 = new ED.Point(r * 0.7, -r * 0.7);
```

```
        this.handleArray[2].location = this.transform.transformPoint(handle1.x, handle1.y);
        var handle2 = new ED.Point(this.apexX, this.apexY);
        this.handleArray[4].location = this.transform.transformPoint(handle2.x, handle2.y);

        // Draw handles if doodle is selected
        if (this.isSelected && !this.isForDrawing) this.drawHandles(_point);
```

Finally, return value indicating a successful hit test:

```
        return this.isClicked;
    }
```

Description

Doodles also provide a textual report, describing their properties and parameters. Within OpenEyes, the descriptions for all doodles that are present within a drawing are added to a single report string and presented to the user.

If required, define the description for a doodle using the description() method. By default, the doodle description is an empty string:

```
ED.Doodle.prototype.description = function() {
    return "";
};
```

For example, circumferential buckle doodles will report the segment of the eye that the doodle covers in clock hours by calling the clockHourExtent method. Additional reporting methods that may be useful are detailed in Appendix Two. All doodles of the circumferentialBuckle class will individually report this description.

```
ED.CircumferentialBuckle.prototype.description = function() {
    var returnString = "";

    // Size description
    if (this.apexY <= -380) returnString = "280 circumferential buckle ";
    else if (this.apexY <= -350) returnString = "279 circumferential buckle ";
    else returnString = "277 circumferential buckle ";

    // Location (clockhours)
    if (this.arc > Math.PI * 1.8) returnString += "encirclement";
```

```
        else returnString += this.clockHourExtent() + " o'clock";

        return returnString;
    }
```

Doodles of the same class can also be described using a group report so that only one description is reported for multiple doodles of the class. To create a group report, use the groupDescription method instead of description():

```
    ED.RPEAtrophy.prototype.groupDescription = function() {
        return "RPE atrophy";
    }
```

To include some doodle specific information with a group description, use a combination of the description, groupDescription and groupDescription end methods. The group description will be called first, followed by the description method text repeated for each doodle of that class, and finally the group description end text appends the report.

The Iris Hook doodle uses this technique, to include the specific clock hour of each iris hook doodle within the group description.

```
    ED.IrisHook.prototype.groupDescription = function() {
        return "Iris hooks used at ";
    }

    ED.IrisHook.prototype.description = function() {
        var returnString = "";
        returnString += this.clockHour();
        return returnString;
    }

    ED.IrisHook.prototype.groupDescriptionEnd = function() {
        return " o'clock";
    }
```

SNOMED CT coding

The presence of a doodle within a drawing can be used to drive SNOMED CT coding.

By default, doodles do not return a SNOMED CT code:

```
ED.Doodle.prototype.snomedCode = function() {
    return 0;
}
```

If coding is required, override the snomedCode() method to define the appropriate SNOMED CT code that should be returned for the doodle subclass.

For example, the posterior capsule doodle will return the SNOMED CT code for the *Posterior capsular opacification* entity (47337003), only if an opacity is present and a capsulotomy has not been performed; otherwise, the doodle will not return a SNOMED CT code:

```
ED.PosteriorCapsule.prototype.snomedCode = function() {
    var valueInt = 0;
    if (this.opacity>1 && this.capsulotomy=="None") valueInt = 47337003;
    return valueInt;
}
```

Cross section doodles

There is both a front-facing and cross sectional view of the Anterior Segment within OpenEyes. Cross sectional doodles are created using the same methods as front view doodles, as described above.

Synching parameters

If a front-facing and an associated cross-sectional doodle have the same parameters, they can be synched so that updating the parameter in one doodle view will update the other view. For example, when you change the pupil size in the anterior segment front-view (using the control parameter in the pop out or by dragging the handle), the pupil size also updates in the anterior segment cross section. Both the apexY and pupilSize parameters have been synched.

In the side view doodle class, add a linkedDoodleParameters object to specify which front-view doodle class it should be synched with, and which parameters. If the parameters are shared between the cross-sectional and front view doodles, list these in the source array. If the parameter is to be stored under a different name in the front view doodle (i.e. it is not synched with the front view), include it in the store array followed by the name of the front view parameter that will store the property. This is useful if the front view doodle also has a parameter of that name, but the values are not synched.

For example, the Anterior Segment Cross Section doodle is synched with the Anterior Segment doodle. Two properties are synched: the apexY and the colour. Additionally, the user can edit the apexX value for the side view, but this is not synched with the front view. Therefore, the apexX is included in the store array; it will be saved in the front view doodle as the csApexX parameter.

```
this.linkedDoodleParameters = {
    'AntSeg': {
            source: ['apexY', 'colour'],
            store: [['apexX', 'csApexX']]
    }
};
```

To work with doodles with synched parameters in a HTML web page or within OpenEyes, see the section on Synchronisation (p.29).

Saving cross section doodles

In OpenEyes, the parameters for the cross section doodles are stored in the associated front view doodle.

Add the parameters for the cross section doodle that need to be stored into the front view doodle, defining the parameters as they appear in the cross section doodle plus the “cs” prefix, and add these parameter to the saved parameter array. These parameters should also be included in the cross sectional doodle class in the linked parameter object store array. You do not need to do this for any parameters that are synched between the front and cross section doodles (i.e. those included in the cross section doodle linked parameter object source array).

For example, the Anterior Segment doodle stores the apexX value from the side view in the csApexX parameter. The cross section apexY value does not need to be saved in this instance as it is synched with the front view apexY parameter.

```
ED.AntSeg = function(_drawing, _parameterJSON) {
    // Set classname
    this.className = "AntSeg";

    // Derived parameters
    this.pupilSize = 'Large';

    // Other parameters
    this.pxe = false;
    this.coloboma = false;
    this.colour = 'Blue';
    this.ectropion = false;
    this.cornealSize = 'Normal';
    this.cells = 'Not Checked';
    this.flare = 'Not Checked';
    this.csApexX = 0;

    // Saved parameters
```

```
        this.savedParameterArray = [
                'pupilSize',
                'apexY',
                'rotation',
                'pxe',
                'coloboma',
                'colour',
                'ectropion',
                'cornealSize',
                'cells',
                'flare',
                'csApexX' // store of cross section apex x value
        ];

        // Parameters in doodle control bar (parameter name: parameter label)
        this.controlParameterArray = {
                'pupilSize':'Pupil size',
                'pxe':'Pseudoexfoliation',
                'coloboma':'Coloboma',
                'colour':'Colour',
                'ectropion':'Ectropion uveae',
                'cornealSize':'Corneal size',
                'cells': 'Cells',
                'flare': 'Flare'
        };

        // Call superclass constructor
        ED.Doodle.call(this, _drawing, _parameterJSON);
};
```

The front view doodle should also be used to report any features of the side view doodle in the description() method, add to SNOMED CT coding, and edit the control parameters (i.e. the cross section doodle should have an empty controlParameterArray; the control parameters should be part of the front view doodle and synched with the cross section, if relevant).

Compiling the EyeDraw JavaScript file

Grunt tasks are used to compile new doodles into the EyeDraw JavaScript file: from a controller, navigate to your EyeDraw directory and run *grunt*.

Running the *grunt watch* task will automatically compile any changes made every time that a doodle class file is saved.

Doodle icons

Within OpenEyes, doodles are added to a drawing using a toolbar. The toolbar contains an icon for each doodle.

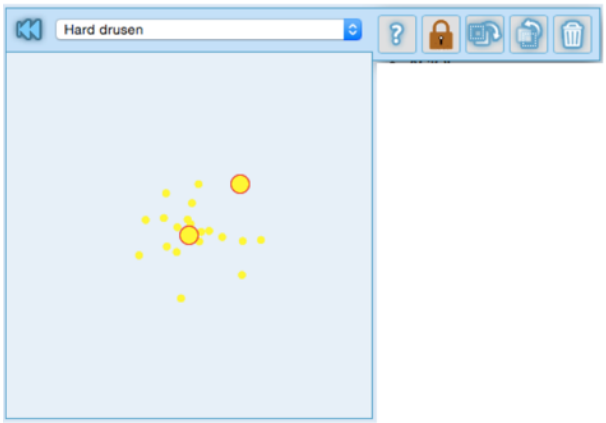
New doodle icons should be .png images, contain no background, and be named as the doodle class (e.g. SectorPRP.png).

Save the new icon to eyedraw/assets/img/new and run grunt from within the EyeDraw directory. The compass task will add the icon to a sprite (eyedraw/assets/img/sprites/icons) and include it in the oe-eyedraw.css file.

Cross-sectional doodles do not require their own icon; they will be added to the canvas with the front-view doodle.

Doodle display title

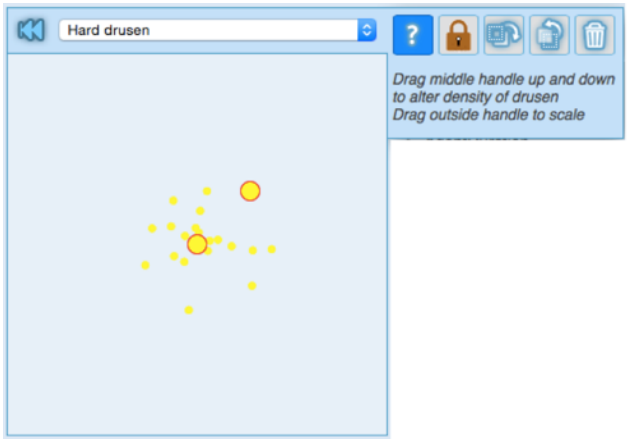
By default, the doodle classname is used as the doodle title in the EyeDraw toolbar and drop-down doodle selector element. This is often not very readable, and so you can create a title for the doodle that is to be presented to the user. For example, “Hard drusen” is used as a title for the HardDrusen doodle class:



Add a human-readable name for the doodle into the DoodleInfo php file (eyedraw/DoodleInfo.php). This will be used to create the doodle-selector drop down element.

Doodle help text

The EyeDraw pop-out “Help” button can be used to help the user know how to interact with a doodle:



Once you have completed your doodle, you should provide helper text that explains the functionality of the handles and so forth. Add the helper text for the doodle class to the Translations.js file located in eyedraw/src/ED/Misc/Translations.

FAQs

How can I test a doodle?

There are test HTML pages set up in EyeDraw (see eyedraw/tests).

The eyedraw/tests/other/AllDoodles.html page displays doodles alphabetically in a toolbar. Click on the button in the toolbar to add a doodle to the canvas below for testing; doodles will display with a blank button if an icon has not been created.

To add a new doodle to the test page, add the doodle class to the arrays in the doodles.js, doodleTitles.js and, if you have created an icon, images.js in the eyedraw/tests/other/js folder.

Why does my doodle jump when I try and move it?

Doodles can appear to “jump” on the canvas when they are first moved if there is a mismatch between a parameter value and the range. Ensure the initial parameter value is set within the defined range for that parameter.

How do I constrain the movement of a doodle within a circular boundary?

Use the dependentParameterValues() method to change the boundary range for the originY value as the originX changes, and vice versa. The equation of a circle is used to recalculate the minimum and maximum possible X values, given the current Y and vice versa.

For example, the posterior capsule doodle is constrained within the Anterior Segment:

```
ED.PosteriorCapsule.prototype.dependentParameterValues = function(_parameter, _value) {
	var returnArray = new Array();

	switch (_parameter) {
		case 'originX':
			// update boundary range for y coordinate using equation of circle
			var y = Math.sqrt(140*140 - _value*_value);
			this.parameterValidationArray['originY']['range'].setMinAndMax(-y,+y);

			// If being synced, make sensible decision about x
			if (!this.drawing.isActive) {
				var newY = this.parameterValidationArray['originY']['range'].max;
			} else {
				var newY = this.parameterValidationArray['originY']
['range'].constrain(this.originY);
```

```
                        }
                        this.setSimpleParameter('originY', newY);
                        break;

                case 'originY':
                        // update boundary range for x coordinate
                        var x = Math.sqrt(140*140 - _value*_value);
                        this.parameterValidationArray['originX']['range'].setMinAndMax(-x,+x);

                        // If being synced, make sensible decision about x
                        if (!this.drawing.isActive) {
                                var newX = this.originX;
                        } else {
                                var newX = this.parameterValidationArray['originX']
['range'].constrain(this.originX);
                        }
                        this.setSimpleParameter('originX', newX);
                        break;
        }

        return returnArray;
}
```

What if there is no corresponding front view for my cross sectional doodle? Will it work in OpenEyes?

If there is no front view doodle, the side view cannot be saved in OpenEyes and it will not add to the drawing report.

However, sometimes it is not appropriate to display a front view for the clinical feature.

For example, there is a cornea cross section doodle but the cornea is not drawn in the front view Anterior Segment canvas. A front view Cornea doodle class (Cornea.js) does, however, exist. This doodle stores the side view parameters and adds to the report, but the draw method is empty so that it cannot be seen or clicked by the user:

```
ED.Cornea.prototype.draw = function(_point) {
    return false;
}
```

You will need to create a corresponding front view doodle for all cross section doodles but, as exemplified above, the front view doodle does not have to be drawn on the canvas so that it will not be visible to the user.

Usage

The following section describes in detail how to create a drawing on an HTML page using HTML elements and javascript.

For most of the sub-sections below, a fully commented and working HTML file is provided in the docs/examples folder, 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 BasicUsage.html);

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

	// Initialise drawing
	drawingEdit.init();
}
```

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:

- offsetX : Offset used to calculate pointer position (default value 0)
- offsetY: Offset used to calculate pointer position (default value 0)
- toImage: Boolean indicating whether to convert the drawing to an image (default value false)
- graphicsPath: Relative path to the folder containing graphics files (default value 'graphics/');

The above code will create an EyeDraw instance linked to the canvas object, and simple commands can be issued via the ‘drawingEdit’ variable. However, for more advanced interaction, a controller object should be instantiated using the following command. Place this before the init() call, in order to guarantee that the controller will receive the ‘ready’ notification.

```
	// Create a controller object for this drawing
	var controller = new eyeDrawController(drawingEdit);
```

A simple controller is illustrated with the following code.

```
function eyeDrawController(_drawing)
{
	// Register controller for notifications
	_drawing.registerForNotifications(this, 'notificationHandler', ['ready']);
```

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

The controller registers itself with the drawing object, and will then receive notifications to the specified handler function. The specific notifications are listed in the last argument of the registerForNotifications function, and if an empty array is passed, the controller will respond to all notifications. A full list of notifications will be found in Appendix Three. In this example, once the images are loaded, a new doodle of the 'AntSeg' class is added to the drawing.

The internal notification handler contains a switch case construct, which in this example will respond only to the 'ready' notification, which is issued when all the image files required by EyeDraw for patterned fills, are successfully loaded.

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 in this example 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 subclass allows it), so that changing the value of the bound element will result in a 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 with 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 that the addDoodle method cannot be used to create the binding). 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 'ready':
	_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 'ready':
	_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])
	{
		// Iterate through slave class names
		for (var slaveClassName in syncArray[idSuffix][className])
		{
			// Slave doodle (uses first doodle in the drawing matching the className)
			var slaveDoodle = slaveDrawing.firstDoodleOfClass(slaveClassName);

			// Check that doodles exist, className matches, sync is possible, and
```

```
master is driving it
                        if (masterDoodle && masterDoodle.className == className && master-
Doodle.isSelected && slaveDoodle && slaveDoodle.willSync)
                        {
                                // Get array of parameters to sync
                                var parameterArray = syncArray[idSuffix][className][slaveClass-
Name]['parameters'];

                                if (typeof(parameterArray) != 'undefined')
                                {
                                        // Iterate through parameters to sync
                                        for (var i = 0; i < parameterArray.length; i++)
                                        {
                                                // Check that parameter array member matches
changed parameter
                                                if (parameterArray[i] == _messageArray.object.para-
meter)
                                                {
                                                        // Sync slave parameter to value of master
                                                        slaveDoodle.setSimpleParameter(_mes-
sageArray.object.parameter, masterDoodle[_messageArray.object.parameter]);

slaveDoodle.updateDependentParameters(_messageArray.object.parameter);

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

                                                        // Refresh slave drawing
                                                        slaveDrawing.repaint();
                                                }
                                        }
                                }
                        }
                }
        }
}
```

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.

Appendix 1

Default doodle properties

Parameter	Description	Default value
isLocked	True if doodle is locked (temporarily unselectable)	FALSE
isSelectable	True if doodle can be selected in drawing	TRUE
isShowHighlight	True if doodle shows a highlight when selected	TRUE
willStaySelected	True if selection persists on mouseup	TRUE
isDeletable	True if doodle can be deleted	TRUE
isSaveable	True if doodle is to be included in saved JSON string	TRUE
isOrientated	True if doodle should always point to the centre	FALSE
isScaleable	True if doodle can be scaled	TRUE
isSqueezable	True if scaleX and scaleY can be independently modified (ie no fixed aspect ratio)	FALSE
isMoveable	True if doodle can be moved	TRUE
isRotatable	True if doodle can be rotated	TRUE
isDrawable	True if doodle accepts freehand drawings	FALSE
isUnique	True if only one doodle of this class is allowed in a drawing	FALSE
isArcSymmetrical	True if changing the arc does not change rotation	FALSE
addAtBack	True if new doodles are added to the back of the drawing (ie first in the doodle Array)	FALSE
isPointInLine	True if centre of all doodles with this property should be connected by a line segment	FALSE
snapToGrid	True if doodle should snap to a grid in doodle plane	FALSE
snapToQuadrant	True if doodle should snap to a specific position in a quadrant	FALSE
snapToPoints	True if doodle should snap to one of a set of specific points	FALSE
snapToAngles	True if doodle should snap to one of a set of specific rotation values	FALSE
snapToArc	True if doodle handle should snap to one of a set of specific arc values	FALSE
willReport	True if doodle responds to a report request (can be used to suppress reports when not needed)	TRUE
willSync	Flag used to indicate whether a doodle will synchronise with another doodle	TRUE

Parameter	Description	Default value
isFilled	True if boundary path is filled as well as stroked	TRUE
showsToolTip	Shows a tooltip if true	TRUE

Default doodle simple parameters

Parameter	Type	Default Range	Default value	Default delta
originX	int	-500, +500	0	15
originY	int	-500, +500	0	15
width	int	100, +500	50	15
height	int	100, +500	50	15
radius	float	+100, +450	0	15
apexX	int	-500, +500	0	15
apexY	int	-500, +500	0	15
scaleX	float	+0.5, +4.0	+1	0.1
scaleY	float	+0.5, +4.0	+1	0.1
arc	float	Math.PI/12, Math.PI*2	Math.PI	0.1
rotation	float	0, 2*Math.PI	0	0.2

Appendix 2

Doodle methods for drawing

Method	Description
drawHandles(_point)	Draws selection handles and sets dragging mode which is determined by which handle and part of handle is selected. Or if a valid point object is passed, function will perform a hit test. _point: Optional point in canvas plane, passed if performing hit test
drawBoundary(_point)	Draws the boundary path or performs a hit test if a valid Point parameter is passed _point: Optional point in canvas plane, passed if performing a hit test
drawSpot(_ctx, _x, _y, _r, _colour)	Draws a circular spot with given parameters _ctx: Context of canvas _x: X-coordinate for origin of spot _y: Y-coordinate for origin of spot _r: Radius of spot _colour: String containing colour for spot fill
drawLaserSpot(_ctx, _x, _y)	Draws a laser spot with given parameters _ctx: Context of canvas _x: X-coordinate for origin of spot _y: Y-coordinate for origin of spot
xForY(_r, _y)	Returns the x coordinate of a point given its y and the radius _r: Radius to point _y: Y-coordinate of point

Point constructor class and methods

Method	Description
point(_x, _Y)	Point constructor. Represents a point in 2D space. _x: x-coordinate of the point _y: y-coordinate of the point
setWithPolars(_r,_p)	Sets properties of the point using polar coordinates _r: Distance from the origin _p: Angle in radians from North, going clockwise
distanceTo(_point)	Returns the distance between the point and another _point: an EyeDraw Point object
dotProduct(_point)	Returns the dot product of two points (treating points as 2D vectors) _point: an EyeDraw Point object
crossProduct(_point)	Returns the cross product of two points (treating points as 2D vectors) _point: an EyeDraw Point object
length()	Returns the length of the point, treated as a 2D vector
direction()	Returns the direction of the point treated as a vector, as an angle from North going clockwise
clockwiseAngleTo(_point)	Returns the inner angle in radians between two the passed vector of the same origin, going clockwise _point: an EyeDraw Point object
pointAtRadiusAndClockwiseAngle(_r,_phi)	Creates a new EyeDraw Point object at an angle _r: distance from the origin _phi: angle from the radius to the control point
tangentialControlPoint(_phi)	Creates an EyeDraw Point object on a tangent to the radius of the point, at an angle of phi from the radius. Useful for bezier curves. _phi: Angle from the radius to control point
bezierPointAtParameter(_t,_cp1,_cp2,_ep)	Creates a new EyeDraw Point object on a cubic Bezier curve at parameter t along curve _t: Proportion along curve (0-1) _cp1: Control point 1 _cp2: Control point 2 _ep: End Point

Doodle methods for reporting

Method	Description
orientation()	Calculates orientation based on x and y coordinates of the doodle origin
groupDescription()	Returns a string which, if not empty, determines the root descriptions of multiple instances of the doodle
description()	Returns a string containing a text description of the doodle
groupDescriptionEnd()	Returns a string which, if not empty, determines the suffix following a group description
snomedCode()	Returns the SNOMED CT code of the entity represented by the doodle
diagnosticHierarchy()	Returns a number indicating the position of the doodle in a hierarchy of diagnoses (from 0 to 9)
clockHour(_offset)	Returns the rotation converted to clock hours (1-12) _offset: Optional integer offset (1-11)
quadrant()	Returns the quadrant of a doodle based on origin coordinates
degrees()	Returns the rotation converted to degrees (0-360)
clockHourExtent()	Returns the extent of the doodle arc converted to clock hours (1-12)
degreesExtent()	Returns the extent of the doodle arc converted to degrees (0-360)
locationRelativeToDisc()	Returns the location relative to the disc, for posterior pole doodles
locationRelativeToFovea()	Returns the location relative to the fovea, for posterior pole doodles
calculateArc()	Calculates the arc in radians for doodles with a natural arc value

Appendix 3

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
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
parameterChanged	The value of a doodle's parameter has changed	doodle:The doodle parameter: Name of parameter value: The new value
ready	All required graphics files are successfully loaded, and the drawing object is ready to accept commands	Empty