OpenEyes - Drawing Tool

Editors: G W Aylward
Version: 0.9:
Date issued: 19 February 2012

Target Audience

General Interest	
Heathcare managers	
Ophthalmologists	
Developers	✔

Amendment Record

Issue	Description	Authors	Date
0.9	Draft	G W Aylward	19 Feb 2012

Table of Contents

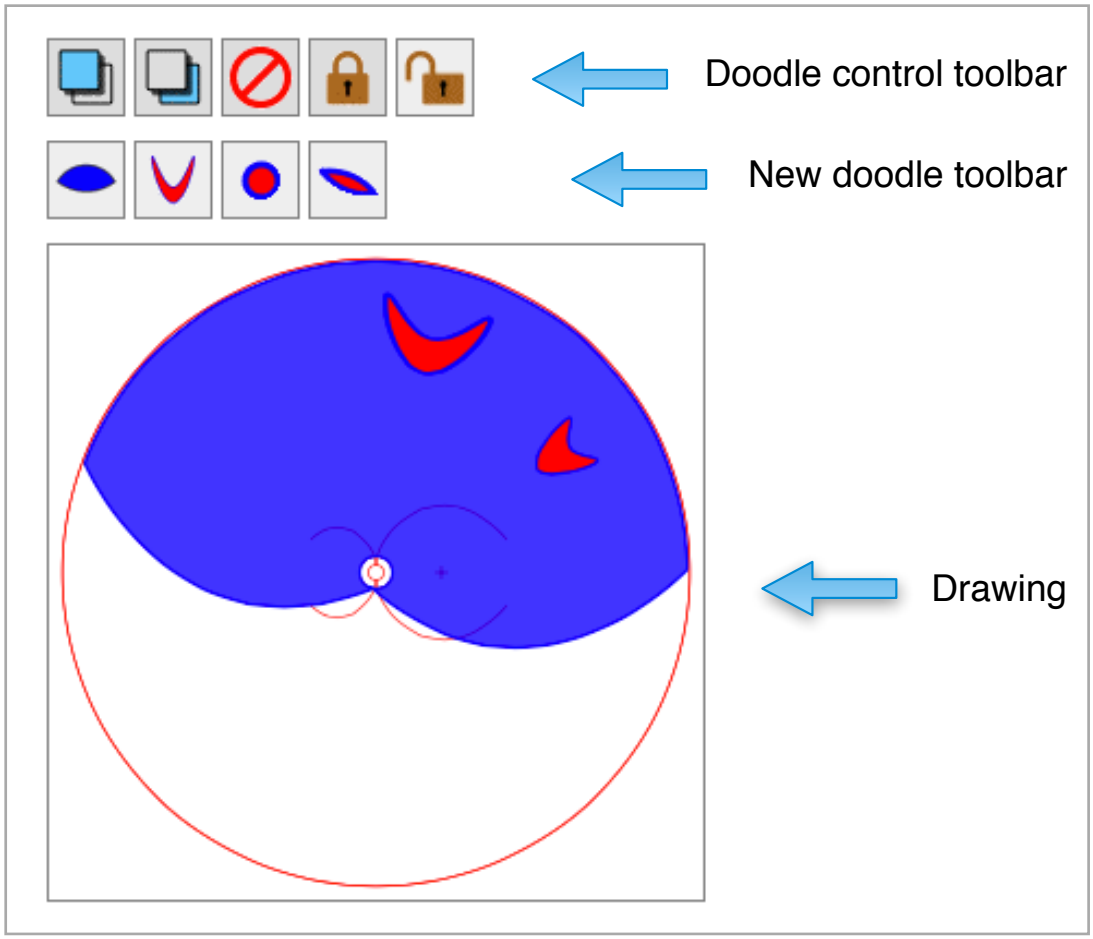
Introduction

The use of drawings is widespread within ophthalmology, and provides a useful method of entering clinical information on a patient. Ophthalmic EPRs have hitherto made use of 'off the shelf' drawing or painting packages which, although they are capable of producing a drawing, are not designed with ophthalmology in mind, and tend to be slow and difficult to use as a result. OpenEyes provides a dedicated drawing package (EyeDraw) with a built in knowledge of ophthalmology. It enables fast and intuitive entry of ophthalmology diagrams, combined with the ability to extract clinical information in text form, infer diagnostic codes, and interact with input fields in a form.

This document describes EyeDraw in detail, gives installation instructions, and takes the reader through the steps required to embed a drawing in an HTML page.

EyeDraw Components

The following diagram shows the components of EyeDraw while in edit mode. There are two toolbars, and a drawing which is made up of a series of 'doodles', four in this case, each of which represent a component of the entity being drawn.



The user can select a doodle by clicking on it with the mouse. Selected doodles can be manipulated in various ways according to their properties (see below). They can also be deleted, moved to the front or to the back, or be locked by clicking one of the icons in the Doodle control toolbar. New doodles can be added to the diagram using the New doodle toolbar. The reader can see the drawing tool in action in the development section of the OpenEyes website here. In display mode, the editing components are removed, and only the doodles are displayed.

Installing the software

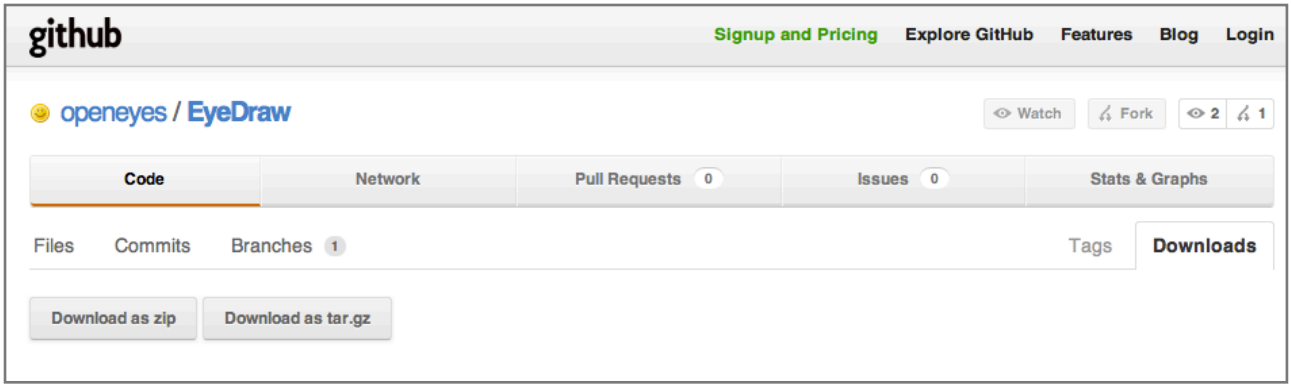
The software consists of a collection of javascript files which are loaded in an HTML page. In order to use and modify the software, the following prerequisites are required;

- A webserver running on your computer
- A text editor or IDE
- A standards compliant browser (e.g. Safari, Chrome, Firefox, or IE9)

To install the latest version of EyeDraw, follow these steps;

1. Download EyeDraw from GitHub

Point your browser here and click one of the download buttons as in the following screenshot;



2. Copy to your webserver

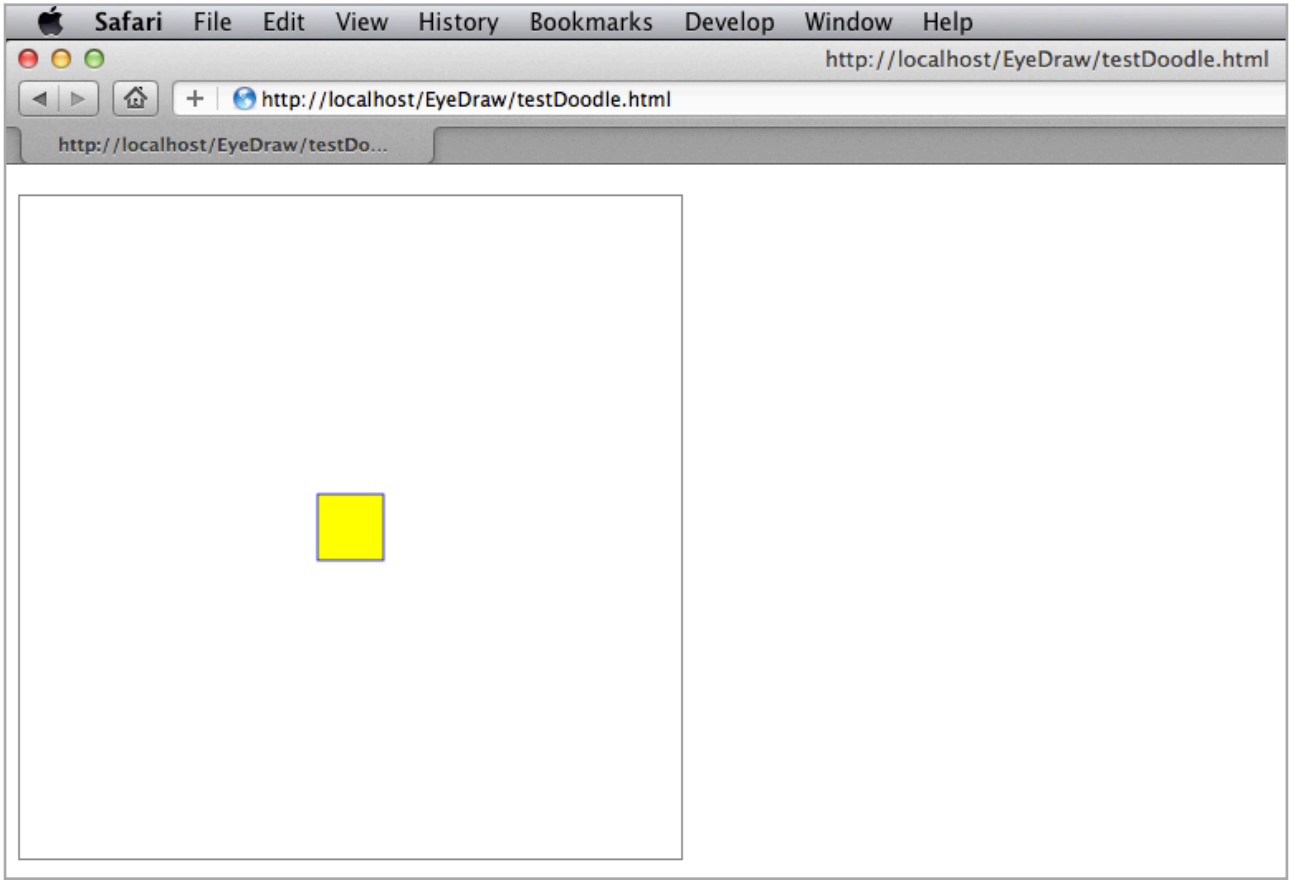
Decompress the downloaded file, and copy the contents to a folder called EyeDraw within the directory served by the webserver. If you use a Mac, this is the 'Sites' folder in the user home folder.

3. Test the setup

Point your browser to the file testDoodle.html within the EyeDraw folder using a URL such as the following (The exact text of the URL will depend on your webserver setup).

http://localhost/EyeDraw/testDoodle.html

You should see the following page;



The canvas element is the large square area surrounded by a grey border, and within that a single doodle (a simple square shape) is displayed.

4. Interact with the drawing

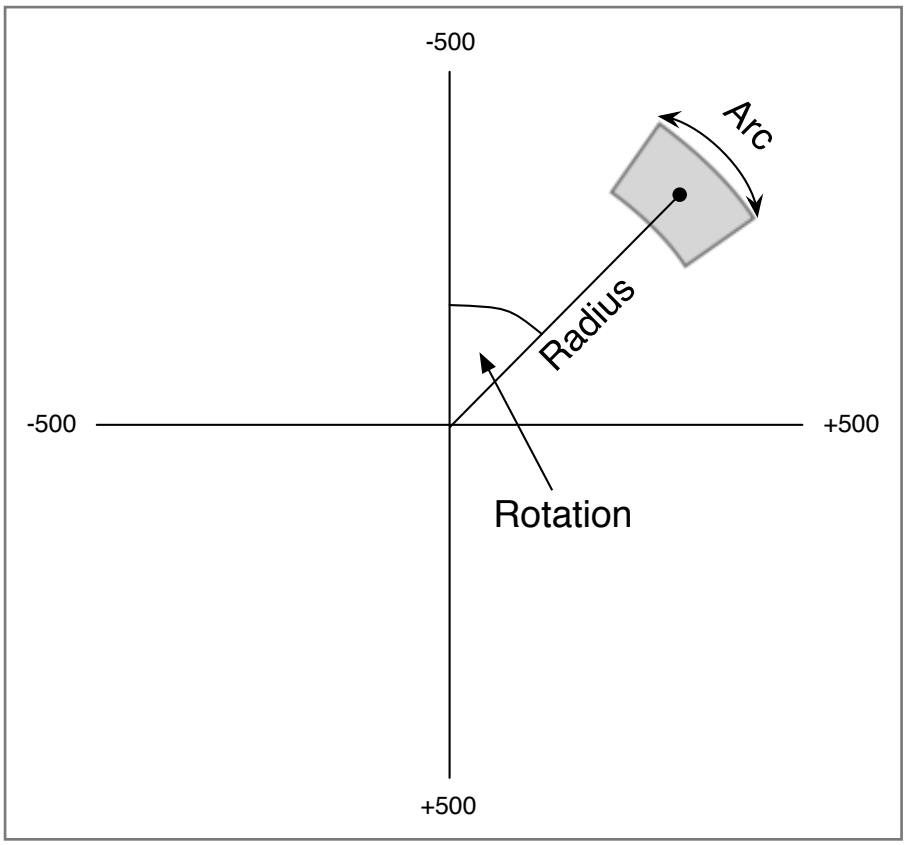
Click on the yellow square and it should become highlighted with a shadow around it. You should then be able to drag it around the drawing to re-position it, but at this stage, not much else.

Understanding Doodles

Each doodle is implemented as an object of a subclass of the 'doodle' class, the code for which can be found in the ED_Drawing.js file (a detailed software reference is available on the website here).

The 'draw' method of each doodle subclass is where the shape of each doodle is specified. As a minimum, this method should contain a 'boundary path', which is the path used for mouse detection when the user attempts to select or manipulate a doodle. The boundary path is usually part of the doodle and is visible, but for some doodles (for example those containing lines rather than shapes) may be invisible. Drawing is carried out in a virtual two dimensional, square shaped, plane called the 'doodle plane' which has a central origin with coordinates (0,0), and extends from -500 to +500 in both x and y directions.

The doodle is drawn in a square HTML canvas element of arbitrary dimension by means of affine transformations. Each doodle has a set of parameters which can be saved to persistent storage, and can be used in the drawing, some of which are shown in the following diagram.



Many of the parameters are used numerically in the doodle draw method, and others are applied by the affine transformation. The following table shows the complete list of parameters and their meaning.

The yellow square is the default doodle called on initialisation by the testDoodle.html page. The following section describes how to copy the template, and edit it in order to manipulate its properties to gain an understanding of how EyeDraw doodles behave.

1. Copy the template

Create a new file for your doodles called (for example) ED_MyDoodles.js in the same directory as the other javascript files and copy the text from Appendix 1 into it (For the latest version, copy the corresponding text at the beginning of the ED_General.js file). Using find and replace, change all instances of the word ‘Template’ to ‘MyDoodle’. Save the file.

2. Adjust the HTML file

Open the file testDoodle.html with your text editor and replace the line;

```
<script language="JavaScript" src="ED_General.js" type="text/javascript"></script>
```

with;

```
<script language="JavaScript" src="ED_MyDoodles.js" type="text/javascript"></script>
```

In the init method of the javascript element , replace the command;

```
drawingEdit.addDoodle('Template');
```

with

```
drawingEdit.addDoodle('MyDoodle');
```

3. Check it is still working

Refresh the page in the browser and you should get exactly the same square, but this time as an instance from your new MyDoodle subclass.

4. Examine the html file

The javascript in the testDoodle.html page runs when the page is loaded and creates the drawing. It consists of the following steps;

Assign a canvas element to a variable in order to pass it to a new EyeDraw drawing object

```
var canvas = document.getElementById('canvas');
```

Create the new drawing object, assigning it to a global variable for the page. The drawing object draws doodles within the canvas element, and handles all aspects of user interaction. with it.

```
drawingEdit = new ED.Drawing(canvas, ED.eye.Right, 'RPS', true);
```

There are four arguments for the drawing object constructor;

- The canvas element
- An enum indicating whether the drawing refers to the right or the left eye
- A three letter code in order to distinguish the canvas and related controls from other drawings on the same page
- A boolean indicating whether the drawing object is editable by the user or not

Tell the drawing object to preload any graphics images it requires (these are used for fill patterns)

```
drawingEdit.preLoadImagesFrom('graphics/');
```

Wait for successful loading, and then add a doodle to the drawing. Newly added doodles are selected by default, so the second line deselects it.

```
drawingEdit.onLoaded = function()
{
    drawingEdit.addDoodle('Template');
    drawingEdit.deselectDoodles();
}
```

5. Examine the javascript file

The code in the ED_MyDoodles.js file consists of a number of methods, most of which are currently empty. The most important is the draw method which is called whenever the drawing is refreshed, and if editable, whenever there is a mouse or touch (iOS) interaction with the canvas element.

```
ED.MyDoodle.prototype.draw = function(_point)
{
```

The optional -point parameter is an EyeDraw point object representing the position of the mouse in the canvas plane, and is used by the draw method of the superclass to detect a whether the point is within the boundary path;

```
    ED.MyDoodle.superclass.draw.call(this, _point);
```

The method then obtains the context of the canvas element and creates the boundary path, in this case a simple rectangle;

```
    ctx.beginPath();
    ctx.rect(-50, -50, 100, 100);
    ctx.closePath();
```

Note that the closePath() method is not necessary in this example since the rect method creates a closed path, but is necessary if other types of boundary path are used. The next few lines set the attributes of the path;

```
    ctx.lineWidth = 2;
    ctx.fillStyle = "yellow";
    ctx.strokeStyle = "blue";
```

The next line draws the path using the above attributes

```
    this.drawBoundary(_point);
```

There then follows an if block which is only called if the drawing method is being used to draw rather than detect a mouse hit. It can be used to embellish the drawing

```
    if (this.drawFunctionMode == ED.drawFunctionMode.Draw)
    {
    }
```

Finally a flag indicating a successful hit test is returned;

```
    return this.isClicked;
```

Modifying Doodle Behaviour

The way a doodle looks is determined by canvas commands within the draw method, and the way that a doodle behaves in response to user manipulation is determined by its properties, a full list of which, along with default values, is given in Appendix 2. The following sequence of changes illustrates the different ways that doodles can be made to behave.

1. Draw a different shape

Edit the ED_MyDoodles.js file and replace the canvas command to draw a square with the following commands which draw a triangle. Note that in this case the call to the closePath() method IS required;

```
ctx.moveTo(0, 50);
ctx.lineTo(-50, -50);
ctx.lineTo(50, -50);
```

2. Limit the range of movement

Add a line to the 'setPropertyDefaults' method so that it looks like the following code segment.

```
ED.MyDoodle.prototype.setPropertyDefaults = function()
{
    this.rangeOfOriginX = new ED.Range(-200, +200);
}
```

Once the page is refreshed, you should find that the horizontal movement of the square is limited to a vertical column two fifths of the width of the canvas. There are a number of calls which limit the range of several doodle parameters, and these are listed in the Appendix.

3. Alter the default position

If the desired default position is not central, then initial x and y coordinates can be defined in the 'setParameterDefaults' method. The following code will place the doodle in the upper half of the drawing;

```
ED.MyDoodle.prototype.setParameterDefaults = function()
{
    this.originY = -300;
}
```

4. Make it point to the centre

Many doodles have a natural orientation towards the centre of the picture. This property can be set using the following syntax;

```
ED.MyDoodle.prototype.setPropertyDefaults = function()
{
    this.isOrientated = true;
}
```

Once refreshed, the doodle should now change its orientation when it is moved so that it is always aligned with a radius to the centre of the drawing.

5. Add a control handle

Doodles can have an arbitrary number of handles which appear when selected, and allow aspects of the doodle to be changed. In order to display a simple handle to change the size (scale) of the doodle, edit the MyDoodle subclass to include the following method;

```
ED.MyDoodle.prototype.setHandles = function()
{
    this.handleArray[0] = new ED.Handle(null, true, ED.Mode.Scale, false);
}
```

This code adds a handle to the doodle's handle array. The handle constructor takes four arguments which are;

- The location in the doodle plane, in this case null since we are going to set the coordinates in the draw method
- A flag indicating whether the handle should be visible, in this case true
- An enum indicating what the moving the handle is meant to do, in this case Scale
- A flag indicating whether the handle should show an outer ring for rotation, set to false for now

Now uncomment the following line in the draw method, and alter the coordinates to put the handle on the top right of the doodle;

```
this.handleArray[0].location = this.transform.transformPoint(new ED.Point(100, -100));
```

Once refreshed and selected with the mouse, the doodle should now show a round yellow handle which can be dragged to resize the doodle.

6. Modify the control handle

An additional ring can be added to the control handle in order to allow rotation of the doodle as well as scaling. Make the last argument in the handle constructor true;

```
ED.MyDoodle.prototype.setHandles = function()
{
    this.handleArray[0] = new ED.Handle(null, true, ED.Mode.Scale, true);
}
```

You will also need to remove the automatic orientation created in step 4 above, which would otherwise conflict;

```
this.isOrientated = false;
```

You can also delete it since the default value of this property is false.

7. Embellish the drawing

So far we have drawn a boundary path which carries out the dual role of mouse hit detection and creating the drawing. By placing commands in the 'f (this.drawFunctionMode == ED.drawFunctionMode.Draw)' block, any number of additional drawing paths can be created. For example, the following code block will draw

a red circle in the centre of the triangle;

```
    ctx.beginPath();
    ctx.arc(0, 0, 30, 0, Math.PI*2, true);
    ctx.fillStyle = "red";
    ctx.fill();
```

Persistent storage

In a web application, EyeDraw will normally be embedded in a form. Drawing data is contained in a JSON string which is written to an <input> element whenever a change is made. On saving the form, the contents of the input element can be saved to persistent storage. When editing an existing drawing, or when in display mode, the web application should write the JSON element to an input element on the page.

If you are using the Yii framework, OpenEyes contains a widget which will do the all the work of setting up the required elements and javascript that should run on page load.

Appendix 1

Template code for a new doodle

```
/**
 * An example to be used as a template
 *
 * @class Template
 * @property {String} className Name of doodle subclass
 * @param {Drawing} _drawing
 * @param {Int} _originX
 * @param {Int} _originY
 * @param {Float} _radius
 * @param {Int} _apexX
 * @param {Int} _apexY
 * @param {Float} _scaleX
 * @param {Float} _scaleY
 * @param {Float} _arc
 * @param {Float} _rotation
 * @param {Int} _order
 */
ED.Template = function(_drawing, _originX, _originY, _radius, _apexX, _apexY,
_scaleX, _scaleY, _arc, _rotation, _order)
{
	// Call superclass constructor
	ED.Doodle.call(this, _drawing, _originX, _originY, _radius, _apexX, _apexY,
_scaleX, _scaleY, _arc, _rotation, _order);

	// Set classname
	this.className = "Template";
}

/**
 * Sets superclass and constructor
 */
ED.Template.prototype = new ED.Doodle;
ED.Template.prototype.constructor = ED.Template;
ED.Template.superclass = ED.Doodle.prototype;

/**
 * Sets handle attributes
 */
ED.Template.prototype.setHandles = function()
{
}

/**
 * Sets default dragging attributes
 */
```

```
ED.Template.prototype.setPropertyDefaults = function()
{
}

/**
 * Sets default parameters
 */
ED.Template.prototype.setParameterDefaults = function()
{
}

/**
 * Draws doodle or performs a hit test if a Point parameter is passed
 *
 * @param {Point} _point Optional point in canvas plane, passed if performing hit test
 */
ED.Template.prototype.draw = function(_point)
{
	// Call draw method in superclass
	ED.Template.superclass.draw.call(this, _point);

	// Get context
	var ctx = this.drawing.context;

	// Boundary path
	ctx.beginPath();

	// Template
	ctx.rect(-50, -50, 100, 100);

	// Close path
	ctx.closePath();

	// Set line attributes
	ctx.lineWidth = 2;
	ctx.fillStyle = "yellow";
	ctx.strokeStyle = "blue";

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

	// Put other drawing paths in this if block
	if (this.drawFunctionMode == ED.drawFunctionMode.Draw)
	{
	}

	// Coordinates of handles (in canvas plane)
	//this.handleArray[0].location = this.transform.transformPoint(new ED.Point(-50,
50));
```

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

        // Return value indicating successful hittest
        return this.isClicked;
    }
```

Appendix 2

List of doodle properties and their default values

Property	Default value	Description
addAtBack	FALSE	New doodles are added behind those already there
gridSpacing	100	Default grid spacing for doodles that snap to grid
isSelectable	TRUE	Whether the doodle can be selected by the user
isDeletable	TRUE	Whether the doodle can be deleted
isOrientated	FALSE	If true, the doodle always points to the centre
isScaleable	TRUE	The doodle can be scaled
isSqueezable	FALSE	The doodle can be scaled independently in X and Y axes
isMoveable	TRUE	The doodle can be moved
isRotatable	TRUE	The doodle can be rotated
isDrawable	FALSE	The doodle can accept freehand drawing
isUnique	FALSE	Only one doodle of this class is allowed per drawing
isArcSymmetrical	FALSE	If the arc is changed, the doodle remains symmetrical
isPointInLine	FALSE	The doodle is one point of many in a line
snapToGrid	FALSE	The doodle snaps to a grid
snapToQuadrant	FALSE	The doodle snaps to a quadrant
willReport	TRUE	The doodle produces a report when requested
rangeOfOriginX	-1000 to +1000	Permitted range of OriginX
rangeOfOriginY	-1000 to +1000	Permitted range of Originy
rangeOfScale	+0.5 to +4.0	Permitted range of Scale
rangeOfArc	Math.PI/6 to Math.PI*2	Permitted range of Arc
rangeOfApexX	-500 to +500	Permitted range of ApexX
rangeOfApexY	-500 to +500	Permitted range of ApexY
rangeOfRadius	100 to 450	Permitted range of Radius property