Filters

Dynamically modify your website images with Photoshop-like filters.

Version 1.0



Written By David Leeper, author of the Filter JavaScript library. May, 2015.

TOC \o 2-3 \t "Heading, 4"

**The Filters PAGEREF \_Toc \h 4**

**Using Filters PAGEREF \_Toc1 \h 6**

Basic Usage PAGEREF \_Toc2 \h 6

Web Worker Usage PAGEREF \_Toc3 \h 7

Progress And Log Messages PAGEREF \_Toc4 \h 9

**The Code PAGEREF \_Toc5 \h 11**

Comparing Float Values PAGEREF \_Toc6 \h 11

Colors PAGEREF \_Toc7 \h 11

Creating Colors PAGEREF \_Toc8 \h 11

Comparing Colors PAGEREF \_Toc9 \h 11

Color Class Data Members And Color Normalization PAGEREF \_Toc10 \h 12

Converting Colors To And From Strings PAGEREF \_Toc11 \h 12

Color Math Functions PAGEREF \_Toc12 \h 13

Color Operations PAGEREF \_Toc13 \h 13

Views PAGEREF \_Toc14 \h 13

Creating Views And Working With Their Width And Height PAGEREF \_Toc15 \h 14

Getting And Setting Pixel Color In A View PAGEREF \_Toc16 \h 14

Converting View Images To And From Strings PAGEREF \_Toc17 \h 14

Drawing An Image In A CanvasView PAGEREF \_Toc18 \h 15

Masks, Factors, And Bias PAGEREF \_Toc19 \h 15

Filters PAGEREF \_Toc20 \h 16

# The Filters

|  |  |  |
| --- | --- | --- |
| **Top Row** | **Middle Row** | **Bottom Row** |
| Original | Translate | Pic Used For Blending |
| Equalize | Rotate | Blend: CROSS |
| Threshold | Scale Bigger | Blend: ADDITIVE |
| Grayscale | Scale Smaller | Blend: ADDITIVE ALPHA |
| Detect Edges | Erosion | Blend: Multiplied |
| Invert | Bilinear Interpolate | Motion Blur |

|  |  |  |
| --- | --- | --- |
| **Top Row** | **Middle Row** | **Bottom Row** |
| Original | Find All Edges | Mean |
| Blur 1 | Sharpen 1 | Assign Channel Value: Red |
| Blur 2 | Sharpen 2 | Assign Channel Value: Green |
| Find Horizontal Edges | Edges | Assign Channel Value: Blue |
| Find Vertical Edges | Emboss 1 | Assign Channel Value: Alpha |
| Find 45 Degree Edges | Emboss 2 | Assign Channel Value: RGBA |



# Using Filters

## Basic Usage

Filters uses images displayed in canvas tags. The general usage pattern is one canvas serves as the original and a second canvas displays the result of the filter. An example is provided below.

1 <html xmlns="http://www.w3.org/1999/xhtml">

2 <head>

3 <title>My Web Page</title>

4

5 <script type="text/javascript" src="./filters.js"></script>

6 <script type="text/javascript">

7

8 function equalize(inView){

9 var filters = new Filters();

10 var oCanvasEqualize = new

11 CanvasView(document.getElementById(

12 "CanvasEqualize"));

13

14 filters.equalize(inView, oCanvasEqualize);

15 }

16

17 function initDraw() {

18 oCanvasOriginal = new

19 CanvasView(document.getElementById(

20 "CanvasOriginal"));

21

22 oCanvasOriginal.drawImage("pic.png", equalize);

23 } // initDraw

24 </script>

25 </head>

26 <body onLoad="initDraw();">

27 <canvas id="CanvasOriginal" width=201; height=300;

28 style="position: absolute; left: 0px; top: 0px;”>

29 </canvas>

30 <canvas id="CanvasEqualize" width=201; height=300;

31 style="position: absolute; left: 210px; top: 0px;”>

32 </canvas>

33 </body>

34 </html>

The code is explained below.

Line 5: The filters.js script is included.

Lines 27-32: The canvas tags are declared

Line 26: initDraw() is called after the page loads.

Lines 17-23: The canvas containing the original image is loaded into a CanvasView object. The original image, pic.png, is loaded. Once the image is loaded a function named equalize() is called, receiving the CanvasView object as a parameter.

Lines 8-15: A Filters object is created and the destination canvas is loaded into a CanvasView object. The equalize filter is called, passing the original CanvasView and destination CanvasView as parameters.

1 <html xmlns="http://www.w3.org/1999/xhtml">

2 <head>

3 <script type="text/javascript" src="./filters.js"></script>

4 <script type="text/javascript">

5 function equalize(inView) {

6 var oCanvasView = new CanvasView(

7 document.getElementById("CanvasEqualize"));

8 var worker = new Worker('filters.js');

9

10 worker.addEventListener('message', function(e) {

11 var data = e.data;

12

13 switch (data.cmd) {

14 case 'ResultEqualizeFilter': {

15 oCanvasView.imageFromString(data.msg);

16 break; // switch

17 } // case

18 case 'Progress': {

19 if (data.msg) {

20 document.getElementById(

21 "CanvasEqualizeProgress").innerHTML =

22 data.msg + "%";

23 }

24 break; // switch

25 } // case

26 case 'Log': {

27 console.log(data.msg);

28 break; // switch

29 } // case

30 default: {

31 alert('Unknown command: ' + data.msg);

32 } // default

33 }; // switch

34 }, false);

## Web Worker Usage

Support for web workers, functions that run in the background, is built into Filters. Using web workers allows your site to remain quick and responsive to user input while the filters process. If you’re processing lots of filters, you’ll want to use web workers. An example is shown below.

The code is explained below.

35 worker.postMessage({

36 'type': 'Filter',

37 'cmd': 'Equalize',

38 'width': inView.getWidth(),

39 'height': inView.getHeight(),

40 'imageString': inView.imageToString()

41 });

42 } // equalize

43

44 function runFilters(inView) {

45 document.getElementById(

46 "CanvasOriginalProgress").innerHTML = "100%";

47 equalize(inView);

47 } // runFilters

48

49 function initDraw() {

50 var oCanvasView = new CanvasView(

51 document.getElementById("CanvasOriginal"));

52

53 oCanvasView.drawImage("pic.png", runFilters);

54 } // initDraw

55

56 </script>

57 </head>

58 <body onLoad="initDraw();">

59 <canvas id="CanvasOriginal" width=201; height=300;

60 style="position: absolute; left: 0px; top: 0px;">

61 </canvas>

62 <canvas id="CanvasEqualize" width=201; height=300;

63 style="position: absolute; left: 210px; top: 0px;">

64 </canvas>

65 <table style="top: 20px; left: 420px; position: absolute;">

66 <tr>

67 <td>1 Original</td>

68 <td id=“CanvasOriginalProgress">0%</td>

69 </tr>

70 <tr>

71 <td>2 Equalize</td>

72 <td id=“CanvasEqualizeProgress">0%</td>

73 </tr>

74 </table>

75 </body>

76 </html>

Line 3: The filters.js script is included.

Lines 59-64: The canvas tags are declared

Lines 65-74: A table is declared. This optional and will be used to show the progress of each filter.

Line 58: initDraw() is called after the page loads.

Lines 49-54: The canvas containing the original image is loaded into a CanvasView object. The original image, pic.png, is loaded. Once the image is loaded a function named runFilters() is called, receiving the CanvasView object as a parameter.

Lines 44-47: The progress for loading the original image is marked at 100% complete. The equalize() method is called to set up the web worker.

Lines 6-7: A CanvasView is created, referencing the canvas used to display the result of the filter.

Line 8: A new web worker is created. The worker uses the filters.js script.

Lines 10-34: A listener for the web worker is created. It handles the result, progress, and log messages sent by the worker. Notice on line 15 that the web worker returns the resulting image as a string, which we then load into the canvas with the imageFromString() method.

Lines 35-41: The parameters need to run the filter are passed to the web worker. Notice the image is passed to the web worker as a string using the CanvasView’s imageToString() method.

The results of the web worker in the browser.

## Progress And Log Messages

Support for progress notifications, and log, warn, and error messages are built in to the Filters class. To use this functionality, just set the progress, log, warn, and error members of the Filters class to functions that process these messages. These functions will receive the name of the filter and the message as parameters.

Note that web workers set theses functions automatically and pass the information back to their callers as messages. See lines 18-29 of the web worker example code above for an example of how to handle these messages.

1 var filters = new Filters();

2 filters.progress = function (name, msg) {…}

3 filters.log = function (name, msg) {…}

4 filters.warn = function (name, msg) {…}

5 filters.error = function (name, msg) {…}

An example of setting the functions manually is shown below.

# The Code

To use Filters, include filters.js in your HTML page. To import filters as a web worker, create a Worker and pass the string ‘fitlers.js’ as a parameter.

## Comparing Float Values

Comparing float values can be problematic due to the fact that two floats can be equal for several decimal places, but have unequal values in decimal places so small you don’t care about the difference.

To deal with this issue, Filters includes a function named isKindaSortaEqual(), which compares two numbers for equality within a given tolerance. An example is shown below.

## Colors

Colors are an important part of working with Filters. At a minimum, it’s a good idea to know how to create colors and compare them.

1 var float1 = 0.00000001;

2 var float2 = 0.00000002;

3 var tolerance = 0.005;

4

5 // The following line will return true.

6 isKindaSortaEqual(float1, float2, tolerance);

1 // Creating a yellow color.

2 var color = new Color(255, 255, 0, 1.0, false);

1 var color1 = new Color(255, 255, 10);

2 var color2 = new Color(255, 255, 0);

3

4 // The following line will return false.

5 color1.isEqual(color2);

### Creating Colors

The colors constructor takes a red, green, blue, alpha, and transparent value. The red, green, and Blue values range from 0 to 255. The alpha value ranges from 0.0 to 1.0. The transparent value is a boolean true or false. If transparent is true, the color will not be displayed regardless of the other color values.The alpha and transparent values are optional, with alpha defaulting to 1.0 and transparent defaulting to false.

### Comparing Colors

The color class provides the isEqual() method for comparing colors.

The isEqual() method is implemented using isKindaSortaEqual(). This allows you to specify a tolerance for the comparison, as shown below.

### Color Class Data Members And Color Normalization

The color class has the following data members which you can access:

* r - The red value of the color.
* g - The green value of the color.
* b - The blue value of the color.
* a - The alpha value of the color.
* transparent - The transparent value of the color (true or false).
* normalized - The normalized value of the color (true or false).

A normalized color has color values between 0.0 and 1.0. A color that is not normalized has color values between 0 and 255. The alpha value is always normalized, but the r, g, and b values may or may not be, based on the value of isNormalized.

To obtain the normalized or 255 version of a color, use the getNormalizedColor() and get255Color() methods. To translate individual values, such as the r value, you can use the xlate255ColorToNormalizedColor() and xlateNormalizedColorTo255Color() methods.

1 var color1 = new Color(255, 255, 10);

2 var color2 = new Color(255, 255, 0);

3

4 // The following line will return true.

5 color1.isEqual(color2, 10);

1 var colorNormalized = myColor.getNormalizedColor();

2 var color255 = myColor.get255Color();

3 var red = xlate255ColorToNormalizedColor(myColor.r);

4 var r = xlateNormalizedColorTo255Color(red);

### Converting Colors To And From Strings

It’s often useful to be able to convert colors to and from strings, especially in an HTML environment. To support this, the Color class provides the following methods.

* toString() Converts the color to a string of the format rgb(r,g,b). If the Color’s transparent member is true, the return string will contain only the word “transparent”.
* fromString() Converts a string in the format provided by toString() back to color values and assigns them to the color.
* toStringWithAlpha() Converts the color to a string of the format rgba(r,g,b,a). If the Color’s transparent member is true, the return string will contain only the word “transparent”.
* fromStringWithAlpha() Converts a string in the format provided by toStringWithAlpha() back to color values and assigns them to the color.
* toOpacityString() Converts the color to a string of the alpha value.
* fromOpacityString() Converts a string in the format provided by toOpacityString() back to color values and assigns them to the color.
* toFilterString() Converts the color to a string of the format filter:alpha(opacity=alphaValue), where alpha value is the color’s a value multiplied by 100.
* fromFilterString() Converts a string in the format provided by toFilterString() back to color values and assigns them to the color.

### Color Math Functions

You can assign, add, subtract, multiply, and divide a color by a number, as shown below.

You can also add and subtract two colors.

A color can be clamped to insure that it is within its minimum and maximum allowed values.

1 myColor.assignNumber(0);

2 myColor.addNumber(10);

3 myColor.subtractNumber(10);

4 myColor.multiplyNumber(2);

5 myColor.divideNumber(2);

1 myColor.addColor(otherColor);

2 myColor.subtractColor(otherColor);

1 myColor.clamp();

1 myColor.invert();

2 myColor.grayscale();

3 myColor.blend(otherColor, Color.blendOperation.CROSS);

4 myColor.blend(otherColor, Color.blendOperation.ADDITIVE);

5 myColor.blend(otherColor, Color.blendOperation.ADDITIVE\_ALPHA);

6 myColor.blend(otherColor, Color.blendOperation.MULTIPLIED);

### Color Operations

The Color class supports a few basic color manipulation methods. These are invert(), grayscale(), and blend(). The blend() method requires a second color to blend and has four variations: CROSS, ADDITIVE, ADDITIVE\_ALPHA, and MULTIPLIED.

Each of these operations, invert(), grayscale(), and blend(), have corresponding filters that work on the entire image.

## Views

Views are used to work with HTML canvas information. Filters provides two view classes: MinimalView and CanvasView.

MinimalView is a memory-only representation of image data that provides the minimal functionality needed to filter images. MinimalViews are used by web workers to run filters.

CanvasView will be used in your code to wrap an HTML canvas object.

### Creating Views And Working With Their Width And Height

The constructor for the MinimalView class takes a width and height parameter that defines the size of the image the view contains. The constructor for the CanvasView class takes a HTML canvas object that is wrapped by the view.

For both class, you can get and set the width and height using the getWidth(), setWidth(), getHeight(), and setHeight() methods.

Note that the setWidth() and setHeight() methods do not change the image data contained in a MinimalView class and will clear the image data contained in a CanvasView class.

1 var oMinimalView = new MinimalView(width, height);

2 var oCanvasView = new CanvasView(

3 document.getElementById("MyCanvas"));

1 var width = oCanvasView.getWidth();

2 var height = oCanvasView.getHeight();

3 oMinimalView.setWidth(100);

4 oMinimalView.setHeight(100);

1 var oColor = oCanvasView.getColor(xLoc, yLoc);

2 oCanvasView.setColor(xLoc, yLoc, otherColor);

3 oCanvasView.fill(oColor);

1 var imageData = oCanvasView.imageToString();

2 oCanvasView.imageFromString(imageData);

### Getting And Setting Pixel Color In A View

Use the getColor() and setColor() methods to get and set pixel colors in a view. Both take an X and Y coordinate that specifies the pixel in the view. The getColor() method returns the that pixel’s color or null if the coordinates are outside the view. The setColor() method takes a third parameter specifying the color to set at the given pixel in the view. You can also use the fill() method to fill a view with a single color.

### Converting View Images To And From Strings

Like colors, the images in a view can be converted to and from strings. Note that the string contains the view’s width and height and a view will only read in a string with the same width and height. The imageToString() and imageFromString() methods are used to do the conversion.

### Drawing An Image In A CanvasView

The drawImage() method is used to draw an image file, such as a .png file, in a CanvasView object. This method takes as parameters a URL, and an optional then function, and xOffset and yOffset. Once the canvas draws the view the then function will be called, passing the CanvasView object as a parameter.

## Masks, Factors, And Bias

The maskFilter() method of the Filters class is a generic filter that accepts a mask, a factor, and a bias to perform a variety of different types of filters. Understanding how these pieces work allows you to get the most out of the maskFilter() method.

A mask is a two dimensional array of numeric values that are multiplied against the red, green, and blue values of the image. Think of the mask as having its center overlay the current pixel. Every pixel covered by the mask array has its value adjusted by the numbers in the array cell that covers it. The values for all these colors are then merged together to get the value for the current pixel.

1 oCanvasView.drawImage("pic.png", function(inCanvasView) {…},

2 xOffset, yOffset);

The factor is used to multiply the red, green and blue values of the current pixel. The bias is added to the red, green and blue values of the current pixel. Masks, factors, and bias do not change the alpha value of any pixels.

The Masks class contains all the masks, factors, and bias values that come with Filters.

|  |  |  |
| --- | --- | --- |
| blur1 | blur1Factor | blur1Bias |
| blur2 | blur2Factor | blur2Bias |
| motionBlur | motionBlurFactor | motionBlurBias |
| findHorizontalEdges | findHorizontalEdgesFactor | findHorizontalEdgesBias |
| findVerticalEdges | findVerticalEdgesFactor | findVerticalEdgesBias |
| find45DegreeEdges | find45DegreeEdgesFactor | find45DegreeEdgesBias |
| findAllEdges | findAllEdgesFactor | findAllEdgesBias |
| sharpen1 | sharpen1Factor | sharpen1Bias |
| sharpen2 | sharpen2Factor | sharpen2Bias |
| edges | edgesFactor | edgesBias |
| emboss1 | emboss1Factor | emboss1Bias |
| emboss2 | emboss2Factor | emboss2Bias |
| mean | meanFactor | meanBias |

|  |  |  |
| --- | --- | --- |
| **Top Row** | **Middle Row** | **Bottom Row** |
| MotionBlur | Sharpen1 | Emboss1 |
| MotionBlur Factor \* 2 | Sharpen1 Factor \* 2 | Emboss1 Factor \* 2 |
| MotionBlur Factor \* 2 Bias - 20 | Sharpen1 Factor \* 2 Bias - 20 | Emboss1 Factor \* 2 Bias - 20 |
| MotionBlur Factor \* 4 | Sharpen1 Factor \* 4 | Emboss1 Factor \* 4 |
| MotionBlur Factor \* 4 Bias - 20 | Sharpen1 Factor \* 4 Bias - 20 | Emboss1 Factor \* 4 Bias - 20 |
| MotionBlur Factor \* 8 | Sharpen1 Factor \* 8 | Emboss1 Factor \* 8 |

You can create you own masks to develop custom filters. You can use custom values for factor and bias to modify the results of existing filters.

## Filters

The Filters class is the heart of the filters package. Each filter is presented in the table below along with the parameters the filter expects.

| Filter | Parameters | Notes |
| --- | --- | --- |
| copy | inSourceView, inDestinationView | Copies the source view to the destination view. |
| histogram | inView | Provides a HistogramResult of the view. |
| sumHistogram | inHistogram | Accepts a HistogramResult and sums it, returning a new HistogramResult. |
| blend | inSourceView, inBlendView, inDestinationView, inBlendMode | Accepts a source and blend view. These are blended using the blend mode and the results placed in the destination view. |
| bilinearInterpolatePixel | inView, inX, inY | Performs a bilinear interpolation on a single pixel and returns a Color containing the result. |
| bilinearInterpolate | inSourceView, inDestinationView | Performs a bilinear interpolation on the source view, placing the results in the destination view. |
| equalize | inSourceView, inDestinationView | Equalizes the source view, placing the results in the destination view. |
| threshold | inSourceView, inDestinationView, inThreshold, inNewHigh, inNewLow, inThresholdAlpha | Performs a threshold on the source view, placing the results in the destination view.  The inThreshold color specifies the threshold. Colors with a higher value are replaced with the inNewHigh color. Colors with a lower value are replaced with the inNewLow color.  Each red, green, and blue component is checked separately. inThresholdAlpha is a boolean indicating if the alpha value should also be thresholded. |
| grayscale | inSourceView, inDestinationView | Grayscales the source view, placing the results in the destination view. |
| invert | inSourceView, inDestinationView | Inverts the source view, placing the results in the destination view. |
| assignChannelValue | inSourceView, inDestinationView, inChannels, inValue | Assigns a given value to the specified color channels (r, g, b, or a). The result is placed in the destination view.  The inChannels parameter is a string indicating which chamomiles should be affected. The string contains one or more of: r, g, b, or a.  The inValue is the new value. If the alpha channel is affected, inValue will be divided by 255 before applying it to that channel. |
| detectEdges | inSourceView, inDestinationView | Detects edges in the source view, placing the results in the destination view. |
| translate | inSourceView, inDestinationView, inOffsetX, inOffsetY, inFillColor | Moves the source view and places the result in the destination view.  The inOffsetX and inOffsetY parameters indicate how much to move the view. The inFillColor parameter is used to fill any space that opens up because of the move.  NOTE: The HTML 5 Canvas element has a translate method that translates drawings done AFTER the call to translate. The Filters translate method translates images drawn BEFORE the call to translate. |
| rotate | inSourceView, inDestinationView, inRotationPointX, inRotationPointY, inAngle, inFillColor | Rotates the source view and places the result in the destination view.  The inRotationPointX and inRotationPointY parameters indicate the center point of the rotation. The inAngle parameter indicates how much to rotate. The inFillColor parameter is used to fill any space that opens up because of the move.  NOTE: The HTML 5 Canvas element has a rotate method that rotates drawings done AFTER the call to rotate. The Filters rotate method rotates images drawn BEFORE the call to rotate. |
| scale | inSourceView, inDestinationView, inXScale, inYScale, inFillColor | Scales the source view and places the result in the destination view.  The inXScale and inYScale parameters indicate how much to scale the view. The inFillColor parameter is used to fill any space that opens up because of the move.  NOTE: The HTML 5 Canvas element has a scale method that scales drawings done AFTER the call to scale. The Filters scale method scales images drawn BEFORE the call to scale. |
| erosion | inSourceView, inDestinationView, inErosionColor, inThreshold, inTolerance, inNeighborColor, inReplacementColor | Performs the erosion operation on the source view and places the results in the destination view.  Each pixel has its neighboring pixels compared with the inNeighborColor. If the number of neighbors equaling the inNeighborColor color is greater than inThreshold, the original pixel is replaced with inReplacementColor.  The comparison between a neighboring pixel and inNeighborColor uses inTolerance to allow for a range of colors to be considered equal. |
| maskFilter | inSourceView, inDestinationView, inMask, inFactor, inBias | Applies inMask, inFactor, and inBias to the source view, placing the results in the destination view.  See the Mask class for masks, factors, and biases provided. |