Web Audio Visualizer Part II

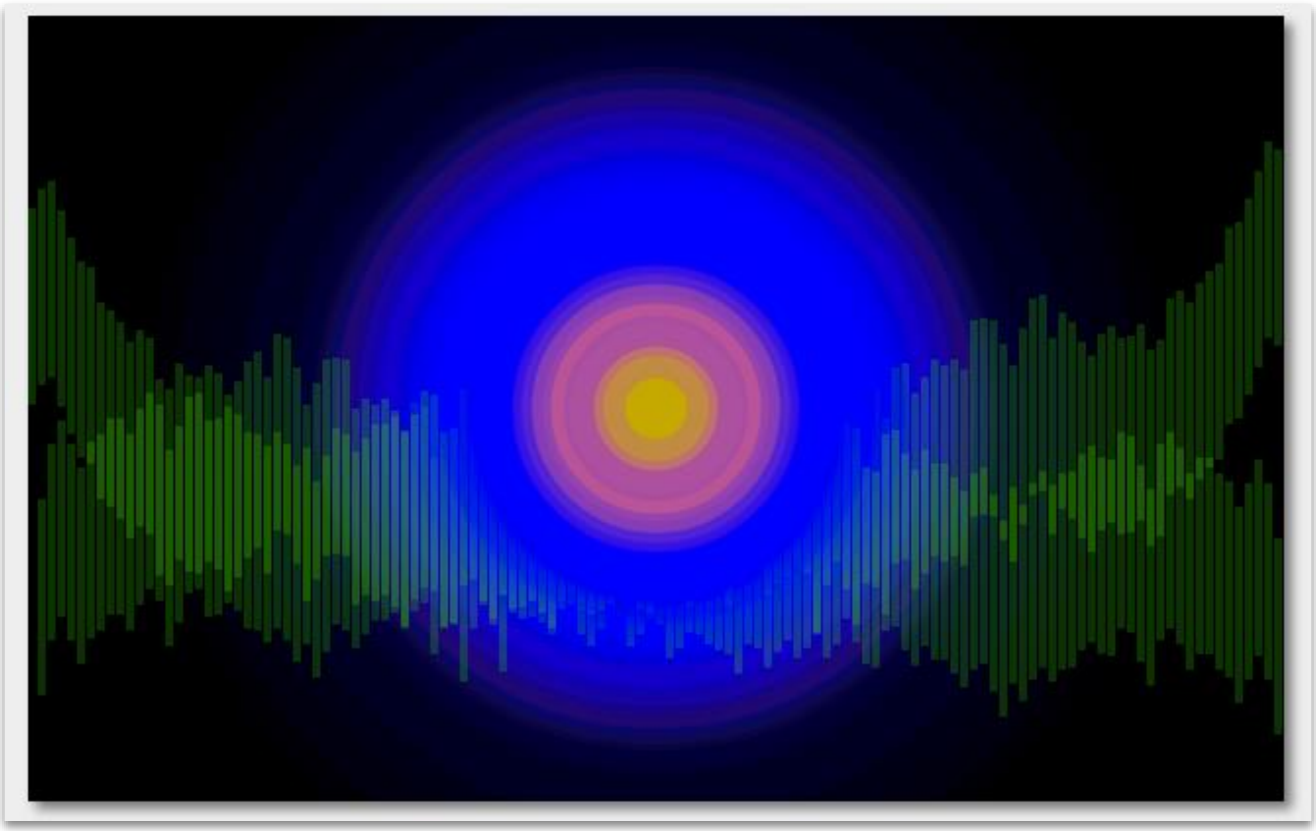
1) Doing pixel manipulations to the entire canvas.

We can grab the RGBA (red, green, blue, alpha) data for the entire canvas, manipulate it, and then copy the modified data back to the canvas. This allows (for example) Photoshop style filters, chroma key effects, edge detection for motion and object recognition applications, and more.

The methods we'll use here:

ctx.getImageData(x,y,width,height) - returns an ImageData object for the specified x,y,width height. This ImageData object has a .data property that is an 8-bit typed array of the rgba values of each pixel of the canvas.

ctx.putImageData(imageData,x,y) - copies an ImageData object onto the canvas at the specified x,y



The starting state of our visualization

A) Grab the Web Audio Visualizer done file (the one with the circles, or another working version you've customized) and add 2 *closure variables* at the top of the file where your ctx and canvas variables are:

```
 var invert = true, tintRed = true, noise = true, lines = true;
```

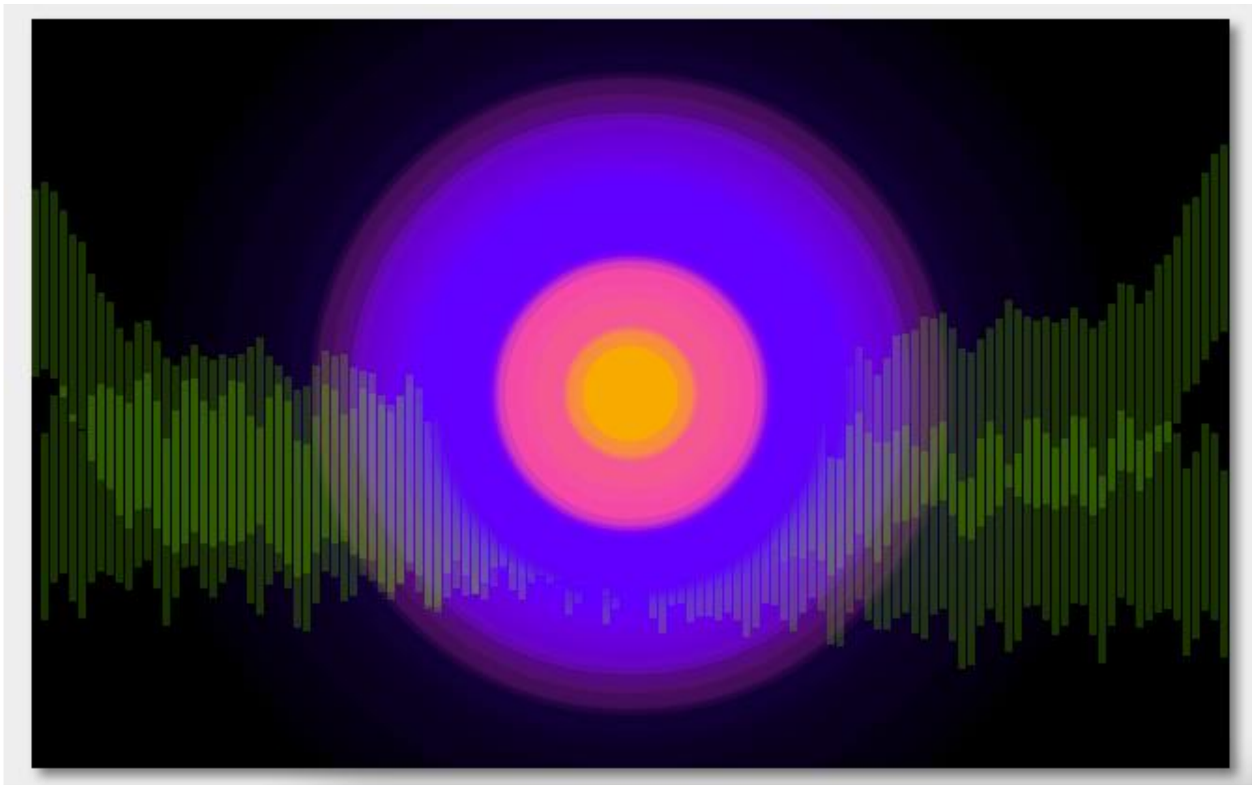
B) At the very end of the update() method (NOT inside the for loop), add a call to a function we're going to write called manipulatePixels().

C) Make manipulatePixels() look like this:

```
function manipulatePixels(){
        // i) Get all of the rgba pixel data of the canvas by grabbing the
        // ImageData Object
        // https://developer.mozilla.org/en-US/docs/Web/API/ImageData
        var imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);

        // ii) imageData.data is an 8-bit typed array - values range from 0-255
        // imageData.data contains 4 values per pixel: 4 x canvas.width x
        // canvas.height = 1024000 values!
        // we're looping through this 60 FPS - wow!
        var data = imageData.data;
        var length = data.length;
        var width = imageData.width;

        // iii) Iterate through each pixel
        // we step by 4 so that we can manipulate 1 pixel per iteration
        // data[i] is the red value
        // data[i+1] is the green value
        // data[i+2] is the blue value
        // data[i+3] is the alpha value

        for (var i = 0; i < length; i += 4){
                // iv) increase red value only
                if(tintRed){
                        // just the red channel this time
                        data[i] = data[i] + 100;
                }

        }

        // put the modified data back on the canvas
        ctx.putImageData(imageData, 0, 0);

}
```

Outside of the comments, there's not much code here - and be sure you understand what's going on.

Run this in Chrome, you should get a red tint on the circle pixels:

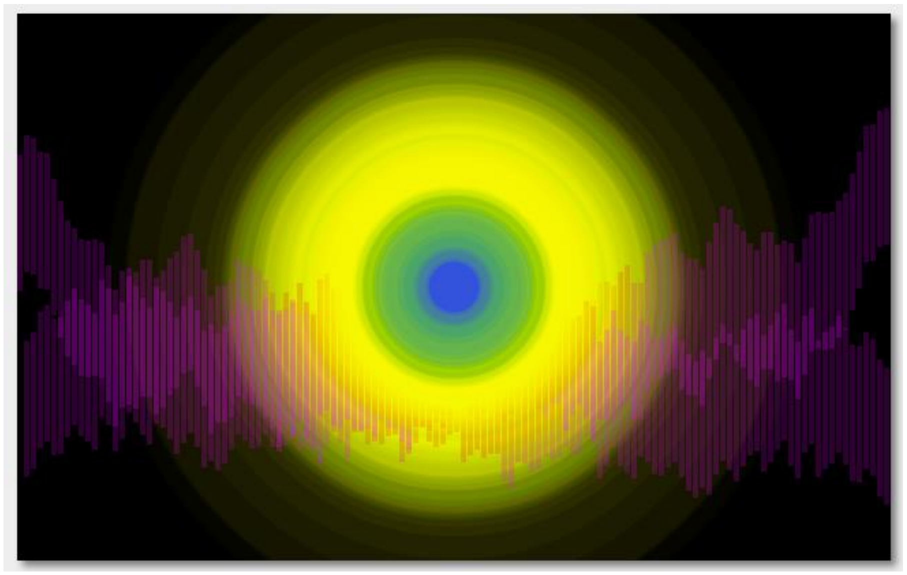


D) Next we'll invert the colors. Add the following right after the tintRed code:

```
// v) invert every color channel
if(invert){
        var red = data[i],  green = data[i+1],  blue = data[i+2];
        data[i] = 255 - red;                // set red value
        data[i+1] = 255 - green;            // set blue value
        data[i+2] = 255 - blue;             // set green value
        // data[i+3] is the alpha but we're leaving that alone
}
```

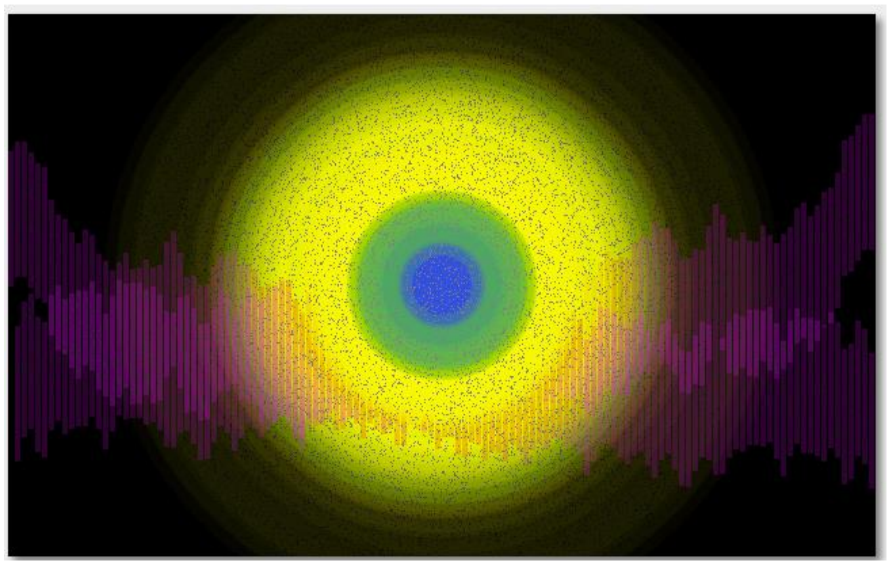
Set your tintRed closure variable to false before you run this. You should see the following:

Idea: Animating this invert effect could give a nice visual.



E) How about some noise?

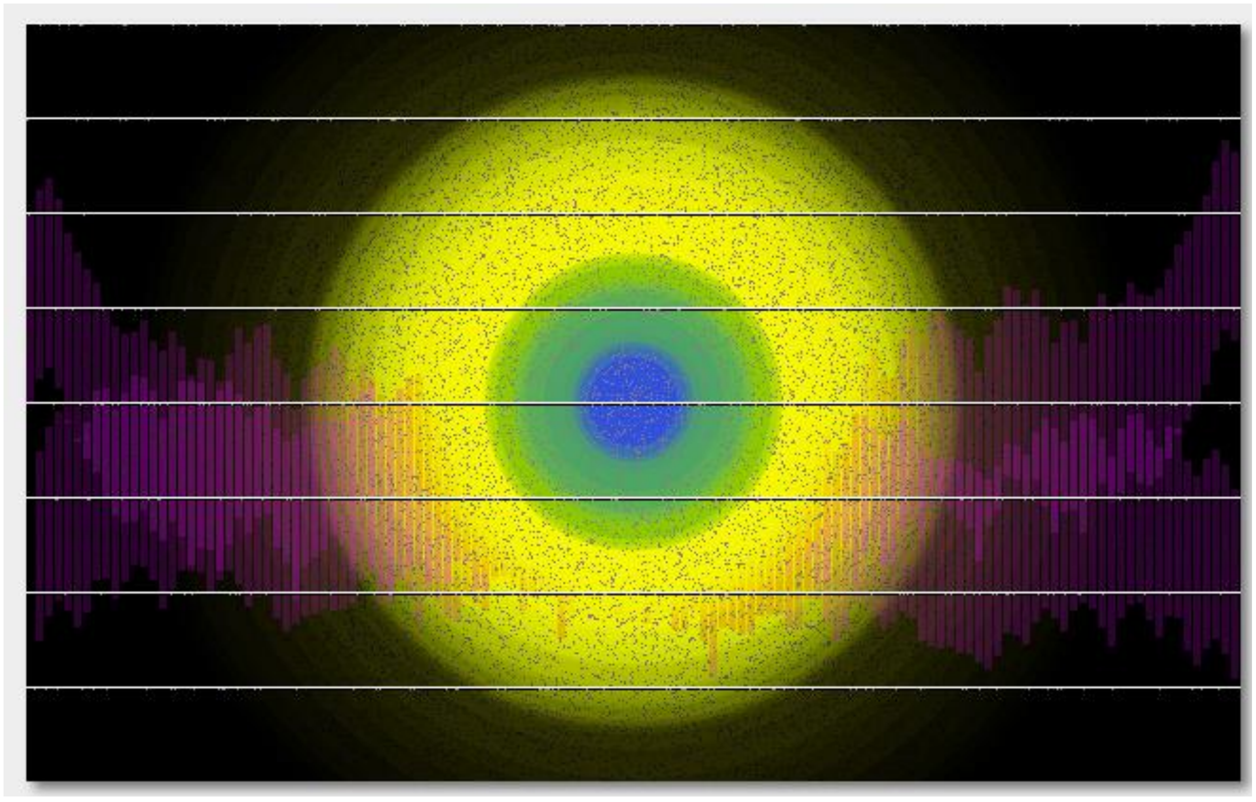
```
// vi) noise
if  (noise && Math.random() < .10){
            data[i] = data[i +1] = data[i+2] = 128;     // gray noise

            //data[i] = data[i +1] = data[i+2] = 255;  // or white noise
            //data[i] = data[i +1] = data[i+2] = 0;      // or black noise

}
```



To see noise even on the black areas we haven't drawn to, add this:
data[i+3] = 255; // alpha

F) How about some lines?

```
// vii) draw 2-pixel lines every 50 rows
if (lines){
        var row = Math.floor(i/4/width);
        if (row % 50  == 0){
                // this row
                data[i] = data[i +1] = data[i+2] = data[i+3] = 255;

                // next row
                data[i+ (width*4)] =
                data[i +(width*4) + 1] =
                data[i + (width*4) +2] =
                data[i + (width*4) +3] = 255;
        }

}
```



That’s enough for now. These effects often work well if animated, under user control (mouse, keyboard etc), and especially on bitmapped images like photographs and video feeds.

2) Image Processing Links

Image processing is a large field of study - here are some links you might find handy:

http://www.techrepublic.com/blog/how-do-i/how-do-i-convert-images-to-grayscale-and-sepia-tone-using-c/#

http://stackoverflow.com/questions/1061093/how-is-a-sepia-tone-created

http://html5doctor.com/video-canvas-magic/

https://archive.org/details/Lectures_on_Image_Processing

http://stackoverflow.com/questions/13932855/canvas-flip-half-the-image

http://www.html5rocks.com/en/tutorials/canvas/imagefilters/#toc-setup

http://lodev.org/cgtutor/filtering.html

Another application of this (when combined with the webcam) is edge detection, which lets us detect, recognize and track objects and motion, which leads to making a webcam controller that acts like the Kinect. There are JS libraries available that can do this.

https://github.com/mtschirs/js-objectdetect

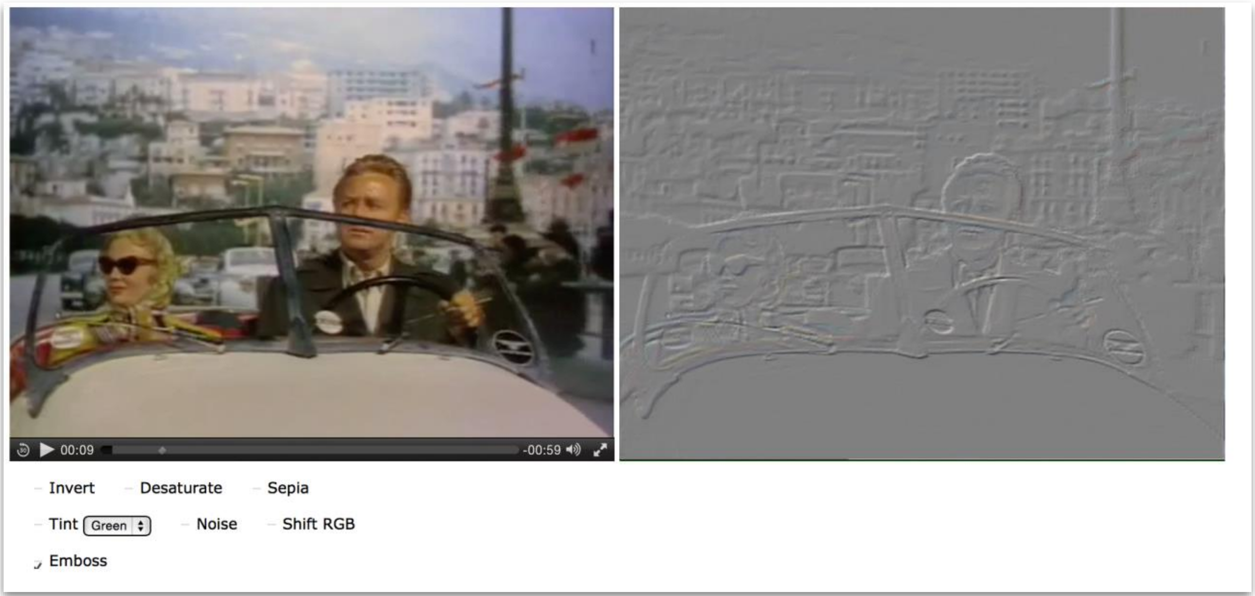
http://trackingjs.com/docs.html

3) A few demos - fun with Canvas and Video

So far our Audio Visualizer is performing pixel effects on procedurally generated drawings - but what if we wanted to do this on still images or even video?

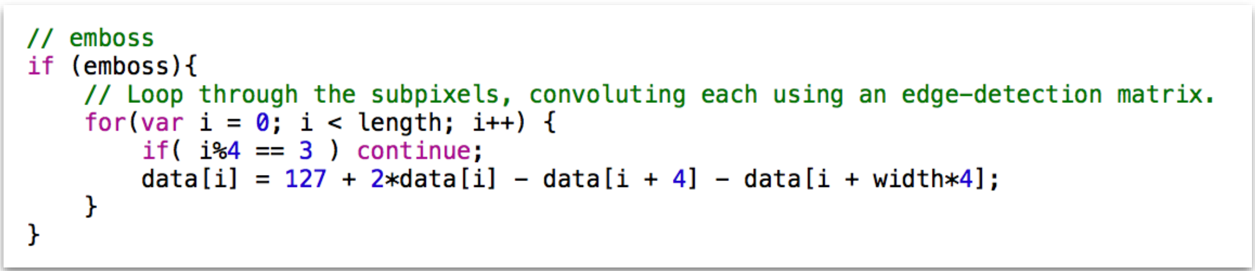
It's easy - just copy the image or video frame to the canvas, then grab the canvas image data, manipulate it, and then copy it back to the canvas.

A) The screenshot below is from:
http://igm.rit.edu/~acjvks/courses/2014-spring/450/code/getImageData-putImageData-demo/video-image-data-demo-3.html



Here we see some additional effects applied to a video, such as *Desaturate*, *Sepia*, and *Shift RGB* (I made that one up). Feel free to "view source" to check out the code.

The above effects look at only one pixel at a time. More interestingly, the *Emboss* effect looks at neighboring pixels.



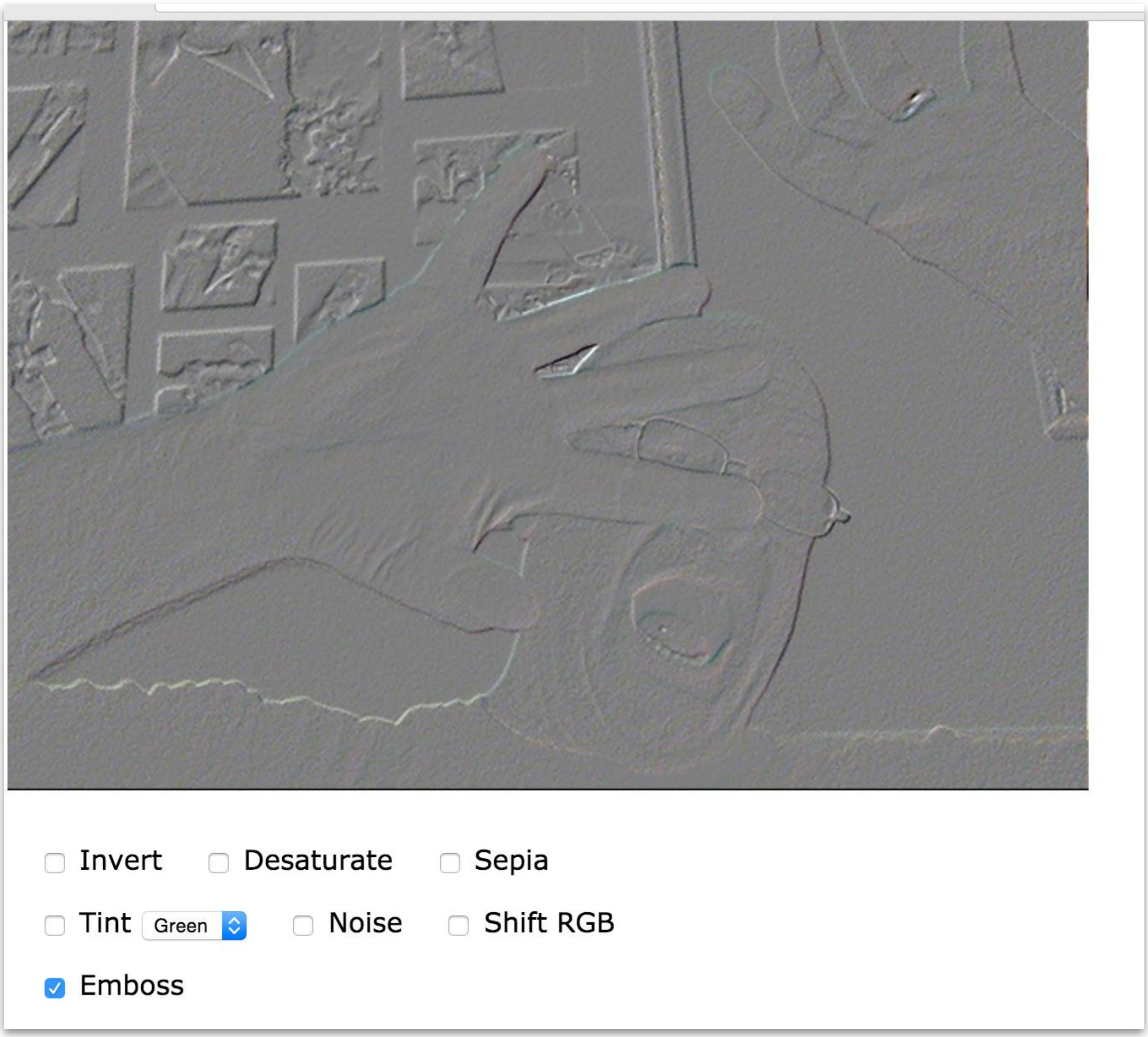
```
// emboss
if (emboss){
    // Loop through the subpixels, convoluting each using an edge-detection matrix.
    for(var i = 0; i < length; i++) {
        if( i%4 == 3 ) continue;
        data[i] = 127 + 2*data[i] - data[i + 4] - data[i + width*4];
    }
}
```

Here's the code for the emboss effect for the above app:

And here's a good explanation of how the emboss effect is accomplished from:
http://html5doctor.com/video-canvas-magic/

Then I just loop through the pixels, like I did before. If the pixel happens to be for the alpha channel (every fourth number in the array), I can just skip it — I don't want to change the transparency. Otherwise, I'll do a little math to find the difference between the current pixel's color channel and the similar channels of the pixels below and to the right, then just combine that difference with the "average" gray value of 127. This has the effect of making areas where the pixels are the same color a flat medium gray, but edges where the color suddenly changes will turn either bright or dark.

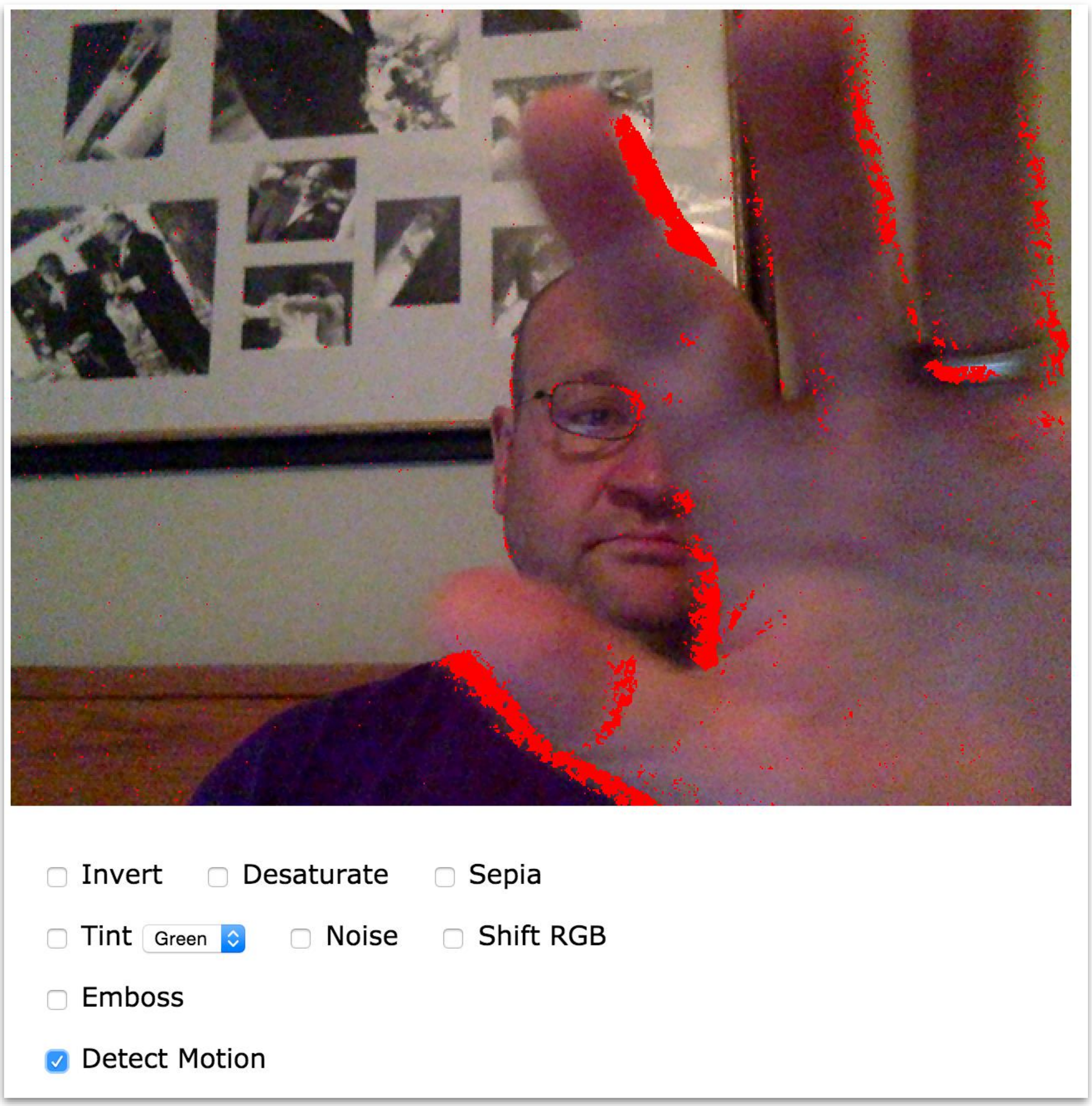
B) Turns out we can do this with webcam video too:



https://igm.rit.edu/~acjvks/courses/2014-spring/450/code/video/webcam-image-data-demo.html
(may be marked unsafe due to RIT's certificate. May need to accept the page to view).

C) We can also use pixel data for motion detection.

https://igm.rit.edu/~acjvks/courses/2014-spring/450/code/video/webcam-image-data-demo-2.html



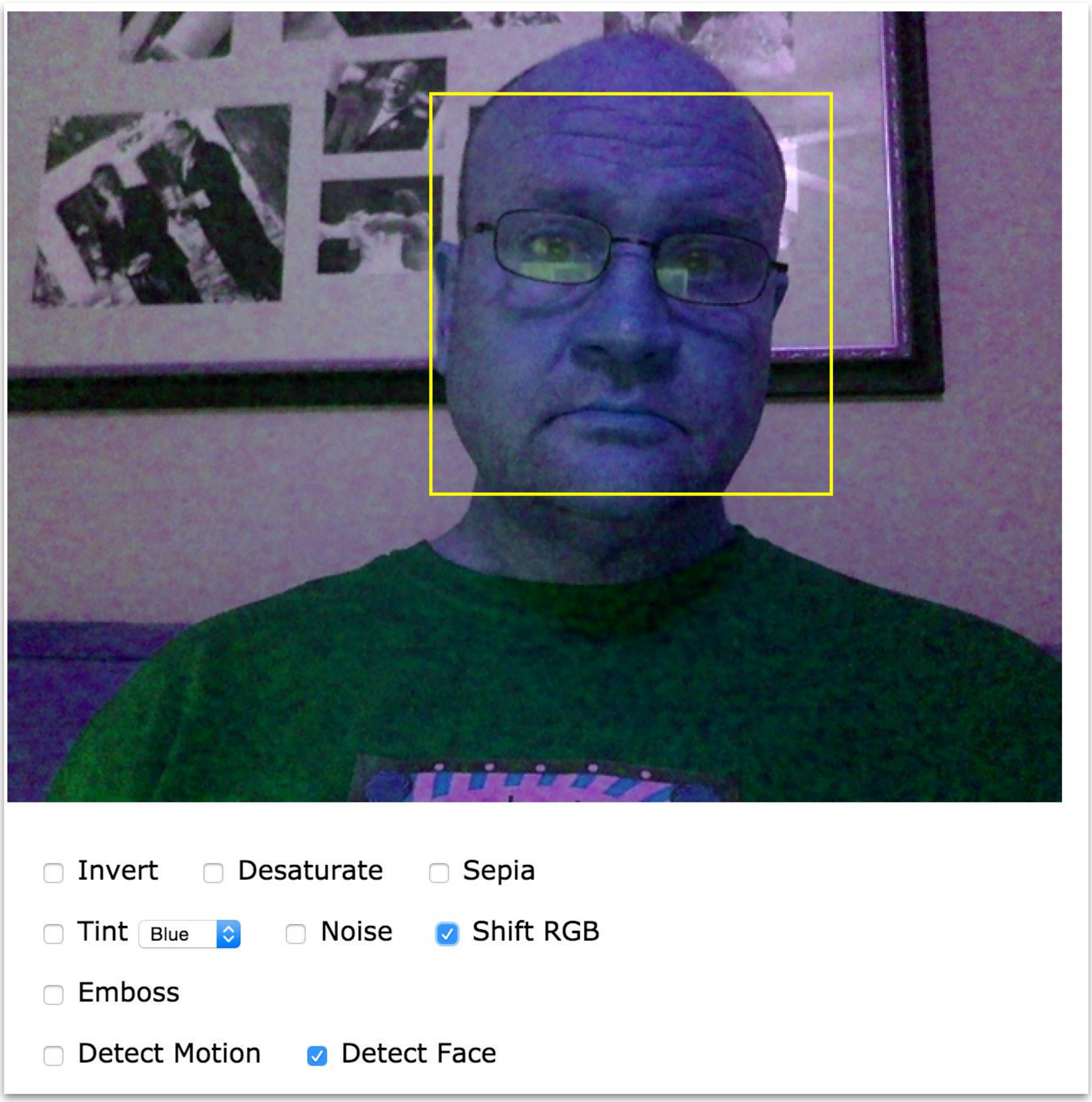
- ☐ Invert ☐ Desaturate ☐ Sepia
- ☐ Tint Green ☐ Noise ☐ Shift RGB
- ☐ Emboss
- ✓ Detect Motion

Feel free to view the source code. Here we just look at the current and previous ImageData captures and compare the pixel data. Wherever any of the color components has varied by 15 units or more, we color that entire pixel red. This has the effect of giving us motion detection for very little effort.

D) This captured pixel data can now be used for face and object detection.

https://igm.rit.edu/~acjvks/courses/2014-spring/450/code/video/webcam-image-data-demo-3.html

And to do this, we're going to go ahead and use an existing JS library rather than write this object detection code from scratch.



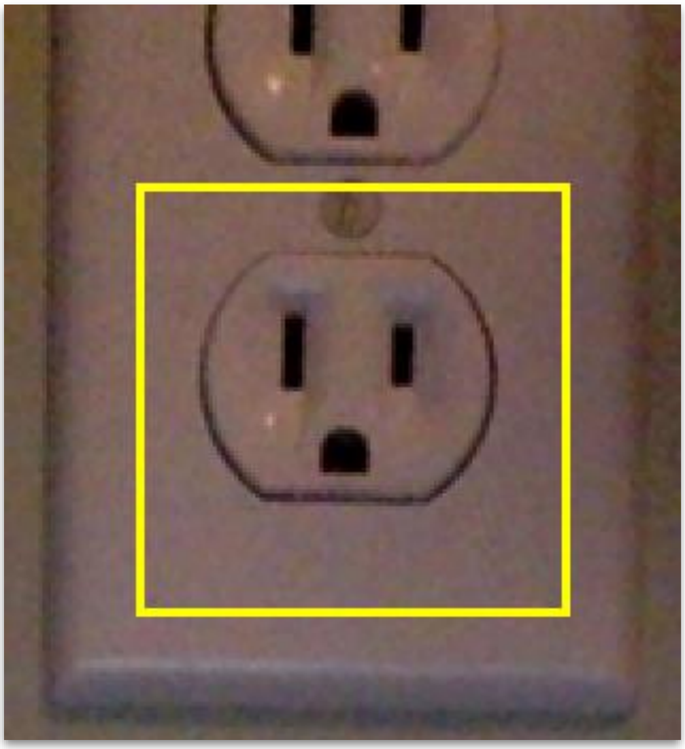
☐ Invert ☐ Desaturate ☐ Sepia

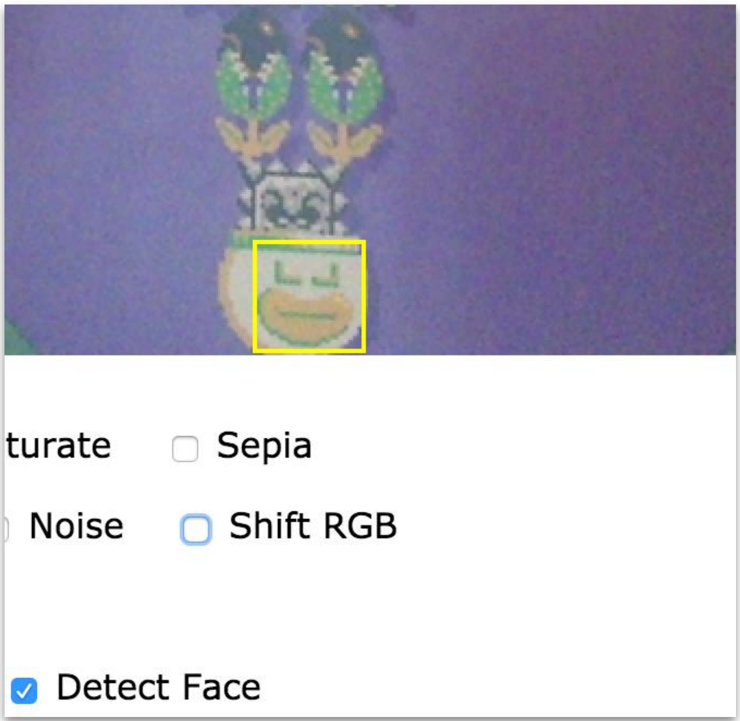
☐ Tint Blue ☐ Noise ☑ Shift RGB

☐ Emboss

☐ Detect Motion ☑ Detect Face

Turns out that wall outlets and Bowser Jr. Propeller Ship's have faces too:





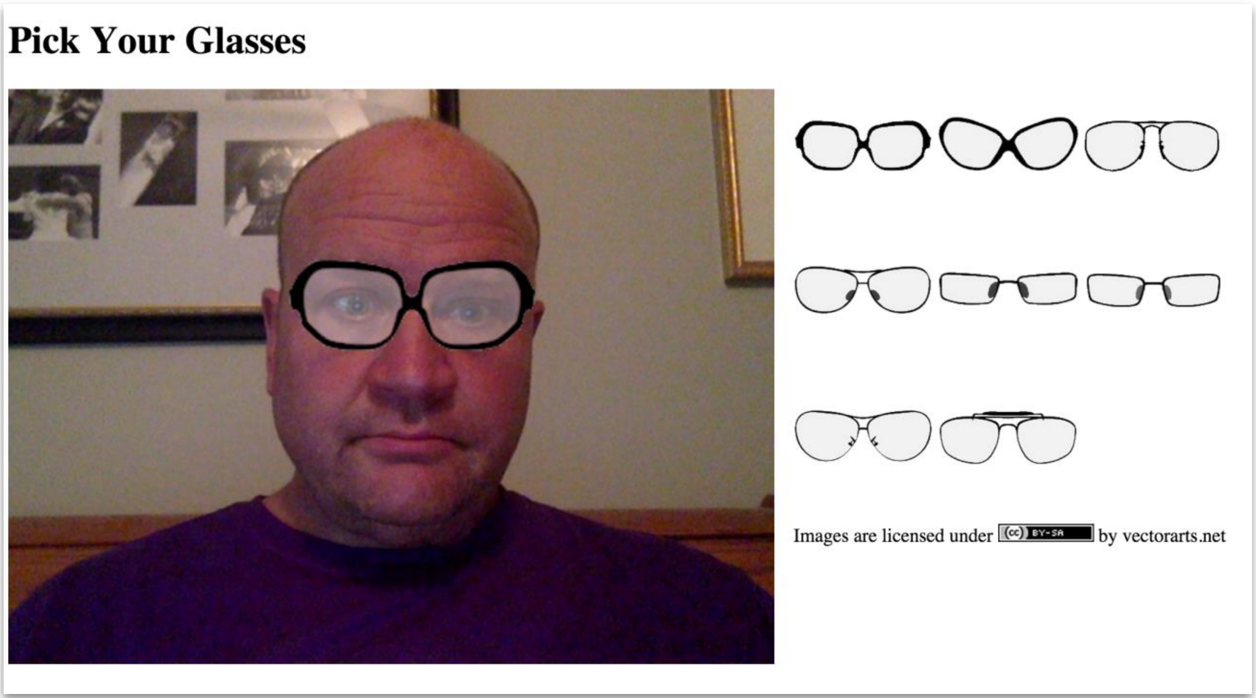
turate [ ] Sepia

Noise [ ] Shift RGB

[✓] Detect Face

E) Now we can see how a sunglass app (or mustache app) works. This also uses a JS library.

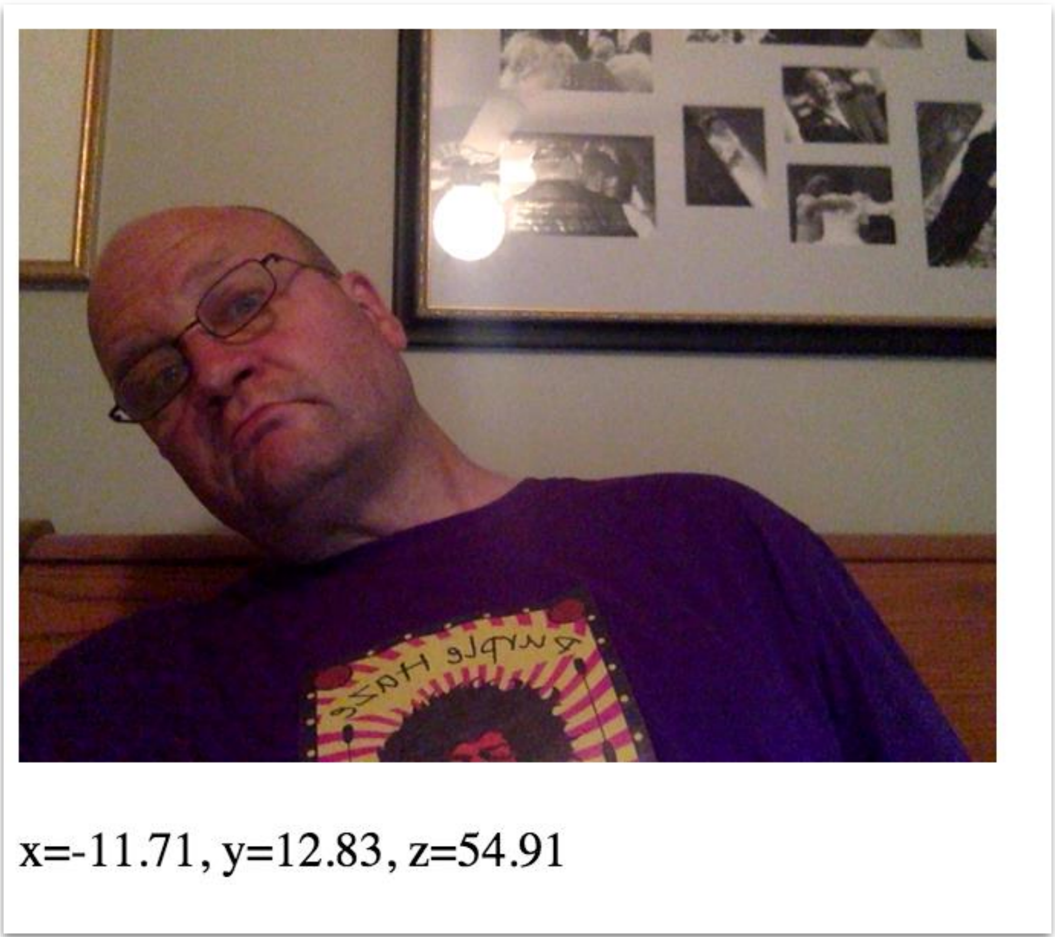
https://igm.rit.edu/~acjvks/courses/2014-spring/450/code/video/JS-Object-Detect/js-objectdetect-master/examples/example_sunglasses_jquery.htm



F) The last application we'll look at is motion tracking, which could be used as a game controller, or as input to a visualizer (hint, hint - see project 1). This example is demoing a JS library.

https://igm.rit.edu/~acjvks/courses/2014-spring/450/code/headtracker/headtracker2.html

Here the x, y, and z of the subject's face is computed and displayed. Below we're not doing anything fancy, just showing the x/y/z values. Note that the x is negative because the subject is leaning left.



```
x=-11.71, y=12.83, z=54.91
```

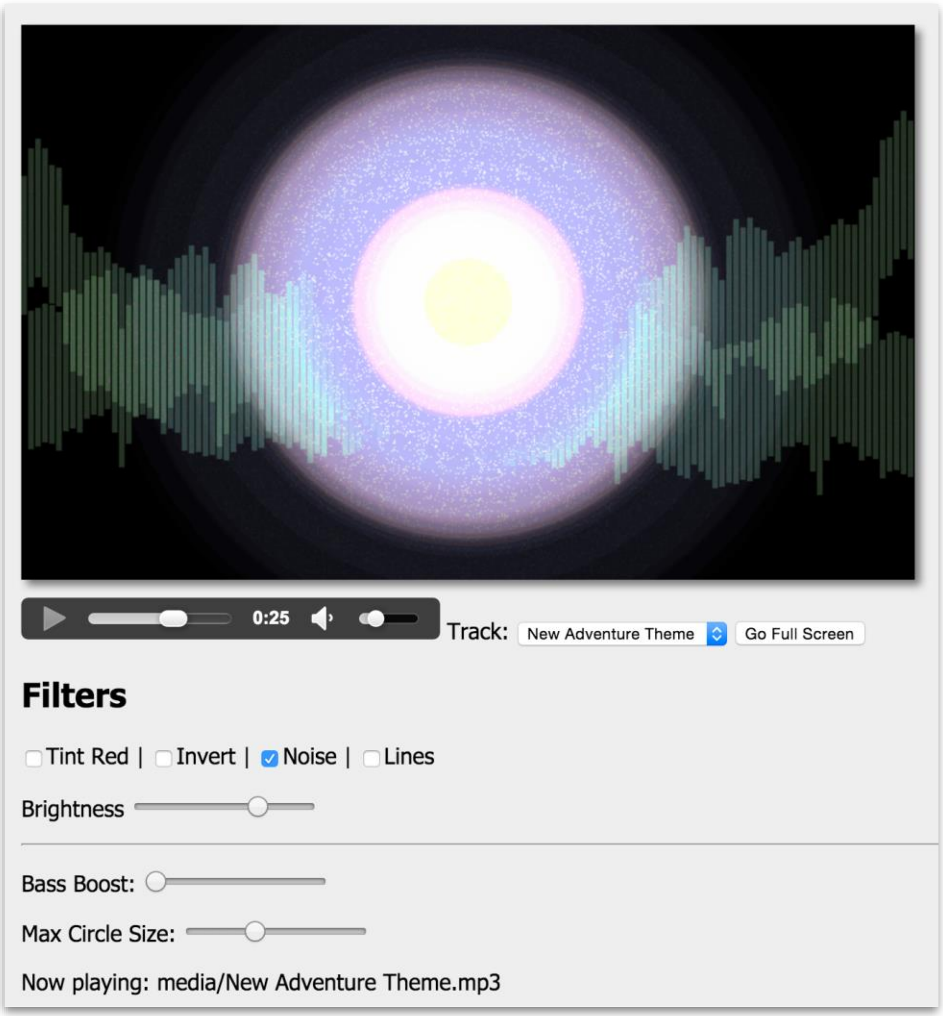
HW!

- Add all 4 of the above pixel effects from way back in section #1 to your completed Audio Visualizer-1 from last time.

- Add check boxes to change the various pixel effect boolean variables so that a user can toggle the effects on and off. Have the initial state of the effects be "off". See the Speed Circles Starter file for how to do checkboxes.

Bonus

- for bonus points, add an additional pixel effect that is also user controllable. Below is an example of a *Brightness* slider. (Ignore the *Bass Boost* slider for now) Don't use any from the video examples from above, come up with your own



Filters

Tint Red | Invert | [✓ Noise] | Lines

Brightness

Bass Boost:

Max Circle Size:

Now playing: media/New Adventure Theme.mp3

Audio Viz 2 Rubric

DESCRIPTION	SCORE	VALUE %
4 Pixel Effects – The four pixel effects are completed and functional.		**20**
Tint Checkbox – Tint checkbox is completed and functional. Effect works on canvas. Not checked by default.		**20**
Invert Checkbox – Invert checkbox is completed and functional. Effect works on canvas. Not checked by default.		**20**
Noise Checkbox – Noise checkbox is completed and functional. Effect works on canvas. Not checked by default.		**20**
Lines Checkbox – Lines checkbox is completed and functional. Effect works on canvas. Not checked by default.		**20**
(Bonus) Pixel Effect – For bonus points, add a pixel effect that is user controllable. This must be connected to the UI in a way that the user can control. The effect must manipulate the pixels in a visible way. Document your bonus effect so that we know what it is.		**+10% (bonus)**
Checkboxes Checked by Default – None of the checkboxes should be checked by default. The user should check them after the page loads.		**-10% (each)**
Errors Thrown – Any errors thrown in the console.		**-20% (this time)**
Additional Penalties – These are point deductions for poorly written code or improper code. There are no set values for penalties. The more penalties you make the more points you will lose.		
TOTAL		**100%**

Submission

Please submit a zip of your files to the dropbox by the due date.
In the submission comments, include a link to your work on Banjo.

If you did the bonus, please also include a description of your bonus effect in the submission comments in order to receive credit.