Instagram has a large collection of filters. In this assignment we'll try to create our own, copying features from a few of them in order to achieve the effect you see in the image above.

Steps to complete

The template code provides the general outline for how this filter will work. We'll call this filter the early bird filter because it looks a bit like the one by Instagram. Notice how the filter itself is a function called earlyBirdFilter() which takes an image as an input and returns an image as output. When you run the code the first time you should see the original picture of the boy with the Husky on the left and the rest of the grey canvas on the right. As you work through the code you'll be uncommenting each line in the earlyBirdFilter() function and implementing the sub-filters that make it up. Filters in social media apps are often combinations of simpler filters we have already examined.

Important: Please use the image we provided in order for us to be able to compare results and be able to mark you. Let's get to it!

Step 1: Inside earlyBirdFilter() function, uncomment the line that calls the sepiaFilter(). Implement the sepiaFilter() function. The basis of this filter is one of the simple filters we saw in class (e.g. invertFilter). Copy the code over and modify it to turn the image into sepia. A simple implementation involves the following conversion of each channel of all pixels.

```
1   newRed = (oldRed * .393) + (oldGreen *.769) + (oldBlue * .189)
2   newGreen = (oldRed * .349) + (oldGreen *.686) + (oldBlue * .168)
3   newBlue = (oldRed * .272) + (oldGreen *.534) + (oldBlue * .131)
```

We don't need to clamp the values of each channel because the p5 pixels array on an image is in fact a clamped array meaning that any value higher than 255 will be stored as 255. Remember to make sepiaFilter() return the resulting image. If you've done things right, you should be seeing the image below.

Step 2: Adding dark corners, also known as vignetting, will give our image a slightly older feel. Uncomment the next line in the earlyBirdFilter() that calls the darkCorners() function and implement the corresponding filter. The basis of this filter is a simple filter like the one above. What you'll need to do is adjust the luminosity/brightness of the pixel by scaling each colour channel. Pixels that are:

- up to 300 pixels away from the centre of the image – no adjustment (multiply each channel by 1)
- from 300 to 450 scale by 1 to 0.4 depending on distance
- 450 and above scale by a value between 0.4 and 0

Hint: You'll need to use the map() and constrain() functions in order to remap the distance of each pixel to a new variable called dynLum (for dynamic luminance) which will hold the scaling that will be required for each channel. If you've done things right you should see an image like the one below.

Step 3: Uncomment the next line in the earlyBirdFilter() that calls the radialBlurFilter() function and implement the corresponding filter. The basis of this filter is the blur filter we saw in class. However, this time we'll create a radial filter that blurs more as you move away from its centre. We'll also use a bigger kernel, as you have seen at the top of the template. Copy over the blur filter and the convolution function from the examples demonstrated in class. The main difference will be in how the new values are calculated. Just after the convolution call inside the blur function, we would need to update the each channel of the specific pixel as we're doing below for the red channel:

```
1    imgOut.pixels[index + 0] = c[0]*dynBlur + r*(1-dynBlur);
```

where c[0] is the red channel returned from the convolution, r is the red channel in the original image and dynBlur is a value we generated using the distance from the mouse. For each pixel we need to calculate the distance between it and the mouse on the colour image. We need to remap the distance from a range 100 to 300 to a new range from 0 to 1. We then need to constrain the returned value from 0 to 1 and save it in the dynBlur variable.

What the dynBlur variable allows us to do when used in the operation above is to say how much of the blur we want to use. When the pixels are up to 100 pixels from the mouse they are clear (the original image values are used and none of the ones returned from the convolution). As the distance from the mouse increases from 100 to 300, more of the blurred image is gradually used until 300 is reached - after which only the blurred image is used and none of the original (i.e. clear) image.

If you have done things right you should see something like the image below. I clicked on the face of the boy in the colour image. Notice the radial blur centered around there.

Step 4: The final touch: Let's add a border around the image. To do this we'll have to implement the borderFilter() function. Like all of the above filters this one will also take an image img as an input and then create a local buffer called buffer of the same size as the input image (Hint: you'll do this using the createGraphics() command). Draw the img onto the buffer. And draw a big, fat, white rectangle with rounded corners around the image. See the documentation about drawing rectangles with round corners here. This should create a rectangle like the one seen below. Make sure you return the buffer at the end of the function.

Draw another rectangle now, without rounded corners, in order to get rid of the little triangles so you end up with the image below.

Step 5: Can you further develop the sketch by implementing some logic to switch between different filter effects on the second image? On key press change between a set number of different "filters", perhaps try to combine different kernels (stored in the variable called "matrix") with different pixel color effects like the sepia color effect, for example grey scale is another effect. Make sure that the first effect visible when the sketch is loaded is the sepia effect with the radial blur and dark corners as per the above instructions, do not change the logic of applying the radial effect on mouse pressed. Beneath the images include written instructions about which key or keys to press. Include comments in the code about what you have done. Points awarded to ambitious learners.

Marking rubric

Step 1 – [2 points]: Sepia filter has been implemented successfully and images look very similar to the ones provided by the instructor.

Step 2 – [2 points]: Vignetting has been achieved using the map() and constrain() functions (code check required) and results look very similar to the ones provided by the instructor.

Step 3 – [3 points]: Radial blur has been achieved using the map() and constrain() functions (code check required). Clicking on the face of the boy in the colour image replicates the results provided by the instructor.

Step 4 – [2 points]: Borders recreated using the technique suggested by the instructor.

Step 5 – [3 points]: Did the student successfully and clearly implement functionality to switch from the main filter effect to some other filter effects?