Vertex Shader:

I defined the attributes a_Position and a_TextureCoord to read in pixel locations and the pixel's texture information, respectively. I then defined the uniform u_Resolution to define the resolution of the canvas, and the uniform u_FlipX that would control if the image is reflected. I lastly defined an output v_TextureCoord to pass the texture information from the vertex shader to the fragment shader. The texture information was not read directly into the fragment shader due to the clipping performed by the vertex shader.

In main(), I first computed clipSpace, which is the multiplier that will convert a coordinate vector from WebGL's coordinate system to the HTML canvas's coordinate system. After that I calculated gl_Position by creating an xy vector with x as u_FlipX (to invert the x coordinate for reflection, if applicable) and y as -1 for an initial vertex, then multiplied it by the clipSpace variable to convert it. The z coordinate was set to 0 and alpha set to 1. After this the texture information was passed to the fragment shader.

Fragment Shader:

The fragment shader was set to medium precision, as high precision made no appreciable difference to the output. I defined u_Image as a sampler2D because that data type has some background functionality for reading from textures so that I don't have to read individual pixels. I then defined the uniforms u_Kernel[9], u_KernelWeight, and u_Recolor. u_Kernel[9] is an array with 9 values to hold a convolution kernel, which are the values that will be used to calculate the blur and sharpen effects. u_KernelWeight is also used to calculate the blur and sharpen effects. u_Recolor acts as a flag for if a recolor effect is to be applied. The output outColor is the final color written to the canvas.

The first thing to happen in main() is the calculation for the size of one pixel after resizing and interpolation of the original image, resulting in the value onePixelSize. This was done using textureSize(), which will get the dimensions of a texture, and then dividing a vec2(1, 1) by the size to get the normalized texture dimensions. After this the colorSum was computed, which is the first step of convolution calculations. For a sliding window that iterates over each pixel, each window has all of the pixels around the target pixel and the target pixel itself has the color values multiplied by the kernel value in the same location relative to the center of the kernel. After this the sum is divided by the kernel weight (which is the sum of all of the kernel values) to get the average color value after multiplication. See the Convolution Example at the bottom of the document for an example. After convolving, the u_Recolor flag is checked and the output of the final convolution sum is either displayed as RGB (if there is no recoloring) or the colors are swapped and the output is converted to BRG. The final output of this is stored in outColor, which is rendered to the screen.

Global Variables:

I set three global variables for ease of reference and data handling. The first is the url of the picture to be used. The second is the name of the current convolution filter. This is initially set to "None" for no effect. Third is a map of convolution kernels under the variable filters. There are

three kernels, one to have no effect, one to compute a gaussian blur, and one to sharpen the image.

Main():

In main() I defined a new Image object named img, which calls for a source url (defined globally) and an onload() function definition. The onload() function is defined to run the function render() when called, which is where the rest of the program is for convenience's sake. After the initialization of img but before giving img the url, I run an if statement to set the crossOrign (CORS) string and check the CORS permissions of the image. Sites will check a CORS permissions string before allowing use of the requested image. However, some sites will only check that there is a CORS string and don't care about the contents, and will only check that there is a CORS string before giving permission to use the image. For this project I set an empty CORS string and am referencing an image that allows empty CORS strings.

Render():

In render() the first thing I do is retrieve the HTML canvas using the WebGL2 context. This is immediately followed by the initialization of the shaders using the initShaders() function provided in the cuon-utils.js, which is a file provided by the professor. After this I get the locations of all of the uniforms and attributes defined in the fragment and vertex shaders, checking each to make sure that the variable has properly linked to the location.

At this point I set up some buffer variables that the position and texture buffers will use. I have bufferSize set to 2, as I pull two values (x and y) from the buffers at a time. I have bufferType set to gl.FLOAT since all values being read from the buffers need to be float values. I have bufferNormalize set to false, as position values shouldn't be normalized to between 0 and 1 and color values cannot be guaranteed to have a low of 0 or a high of 1, so normalizing them could change the color unintentionally. I have bufferStride set to 0 to indicate where in the kernels to start reading from (index 0). I have bufferOffset set to 0 to indicate where in the texture/image to start reading from (index0).

Here I create a vertex array to put position values into. After that I call gl.bindVertexArray() to bind the vertex array to WebGL's framework and make it accessible. I then call gl.enableVertexAttribArray(a_Position) to link a_Position to the vertex array. Next I create a buffer in the variable positionBuffer, after which I bind it to gl.ARRAY_BUFFER which links it to the vertex array. Finally I call gl.vertexAttribPointer(), passing it a_Position, positionBuffer, and the buffer variables to connect a_Position to the buffer and set up for the buffer to be read. This series of steps creates the vertex array where I later pass in values that I need the shaders to work with, connects it to the positionBuffer, which is then connected to the shaders and set up so that a_Position in the shader can read from the buffer.

I perform a similar series of steps to set up the textureBuffer. I don't need to create a vertex array again, but i do call gl.createBuffer() to create the textureCoordBuffer and call gl.bindBuffer() to connect it to the shader. I also call gl.enableVertexAttribArray() and pass it a_TextureCoord so that the values in textureCoordBuffer can be read by a_textureCoord in the

shader. At this point I put vertices into the buffers using a Float32Array. The vertices form the two triangles that make up the canvas (and later the image). After this I call gl.vertexAttriPointer() and pass it a_TextureCoord and the buffer variables to connect the buffer to a_TextureCoord send the vertices to the shader.

Here I create the drop-down menu. Creating the menu here rather than in the HTML file allows for easy addition of behaviors. The document.createElement("select") creates the drop-down container, but not the elements. After creating the container I create a list of all of the options I want in the menu. I iterate over the list's size, and for each index I create an option object using document.createElement("option"), set the text using the string value from the list via createTextNode, and adding it to the drop-down container using appendChild(). The first option is set to be the current option selected, and is set as selected. Following the creation of the menu I set the drop-down's behavior. I set it so that whenever the drop-down's selected option changes, it sets the global variable of the current filter to the name of that option and then runs a function that handles drawing the images with the filters. Finally I add this to the HTML page by getting the drop-down container's placeholder via document.querySelect() and using appendChild() to attach the drop-down container to the HTML element.

Next I set up the texture to load the image into. I start by creating a blank texture using gl.createTexture() and assign it to the variable tex. Then I call gl.activateTexture(gl.TEXTURE0 + 0) to set the active, usable texture to be the first texture and only the first texture I link it to. Then I call gl.bindTexture(gl.TEXTURE_2D, tex) to link the empty texture to the active texture. After this I set four textures in WebGL's background using gl.texParameteri(). I set gl.TEXTURE_WRAP_S and gl.TEXTURE_WRAP_T to be gl.CLAMP_TO_EDGE so that parts of the image don't get clipped, and I set gl.TEXTURE_MIN_FILTER and gl.TEXTURE_MAG_FILTER to be gl.NEAREST for interpolation handling. Setting these four values allows for the code to work with any sided image without distorting the image. At this point I load the image into a texture object using texImage2D, setting the resolution to be as high as possible, colors to be RGBA, and the type of values contained in the image file to be UNSIGNED_BYTE. Finally, I call gl.bindBuffer on positionBuffer again to update the buffer and pass the texture into the buffer.

Next I resize the HTML canvas in order to not destroy the image ratio. I pass new vertex coordinates to the array buffer, however the values start at 0 and go to either the image's height or width (as is appropriate for that vertex) instead of 1. This primes the buffer for the change in canvas size. Next I check if the canvas is the correct size, and if not set canvas.width to the image width and canvas.height to the image height.

After resizing I make the initial call to the function handleSelection(). This is necessary as otherwise the webpage would not initially show an image, as the drop-down requires a change to call this function.

HandleSelection():

Next I define the function handleSelection(). This is the function that handles passing the convolution kernels to the shaders as well as setting u_FlipX and u_Recolor. This function is

defined inside of render() because it allows me to access variables local to render(), but is also a function that I can call which is necessary for the drop-down's behavior definition. First I set three variables, kernelName is "None", flip is 1, and color is 0. Next comes a switch statement that modifies these variables based on the drop-down's current selection. If either Blur or Sharpen are selected, kernelName is set to those values so that the relevant convolution filter is applied. Otherwise the convolution kernel will have no effect. If Reflect is the chosen option flip will be set to -1 to reflect the image. If Recolor is selected color is set to 1 to signal for recoloring. After the switch statement flip is passed to u_FlipX for the vertex shader to reflect the image or not based on selection, and color is passed to u_Recolor to set the vertex shader's flag for recoloring.

After this I call gl.viewport(), passing it the necessary values, to set the clipping space. By using this function a lot of the conversion from WebGL coordinate space to HTML coordinate space is handled without me having to read, calculate, and render each individual pixel. After this I pass the canvas width and height into u_Resolution using gl.uniform2f(). Following this I call gl.clear(gl.COLOR_BUFFER_BIT) to wipe the canvas clean and prepare for a new image to be drawn. Next I call gl.useProgram(gl.program) to load the WebGL program I loaded in render() as well as gl.bindVertex(vertexArray), since I'm in a new function. I then call gl.uniform(u_Image, 0) get the active texture unit and pass it into the u_Image variable in the shader.

At this point I get the kernel variables. First I pass the current kernel into u_Kernel by accessing the filters map and indexing the the currentFilter variable, and call gl.uniform1fv(). Next I calculate the kernel weight by iterating over the kernel with a for loop, and getting the sum of all of the elements. I then pass the weight to u_KernelWeight using gl.uniform1f().

Finally I call drawArrays(), telling it to draw gl.TRIANGLES since we are defining our texture to be two adjacent triangles. I also tell it that the start index is 0 and to take 6 vertices at a time since our texture coordinates are defined by the coordinates of the adjacent triangles.

Convolution Example (Grayscale):

Image:	Kernel:	Convolution Over (1, 1):	Convolution Sum: 9
0, 0, 0, 0	1, 2, 1	0*1, 0*2, 0*1	Final Value: 9 / 16 = 0.5625
0, 1, 1, 0	2, 4, 2	0*2, 1*4, 1*2	
0, 1, 1, 0	1, 2, 1	0*1, 1*2, 1*1	
0, 0, 0, 0	Kernel Weight: 16		