The City College of New York

IMAGE PROCESSING WITH OPENGL

BY KLAUDIO VITO AND TARAS FERLEY

Capstone II | Prof. George Wolberg | December 20, 2016

Table of contents

1. Introduction

The Capstone II course aimed at familiarizing the student with OpenGL to implement image processing operations. From the OpenGL website, “OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms. OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment.” [1] OpenGL works by instructing the Graphics Processing Unit (GPU) directly, rather than using the Central Processing Unit (CPU).

In this report, we will provide an introduction of how OpenGL performs its operations behind the scenes and how to create a channel between C++ and OpenGL. Furthermore, we will discuss the implementation of three filters: *blur*, *convolve*, and *correlate* using a CPU and a GPU approach. Finally, we will analyze the timing related to each implementation approach to understand the efficiency of instructing the GPU directly.

The Graphic User Interface (GUI) component of this project is built using Qt; a cross platform development framework written in C++. The final GUI programmed for this course is shown in the following figure.

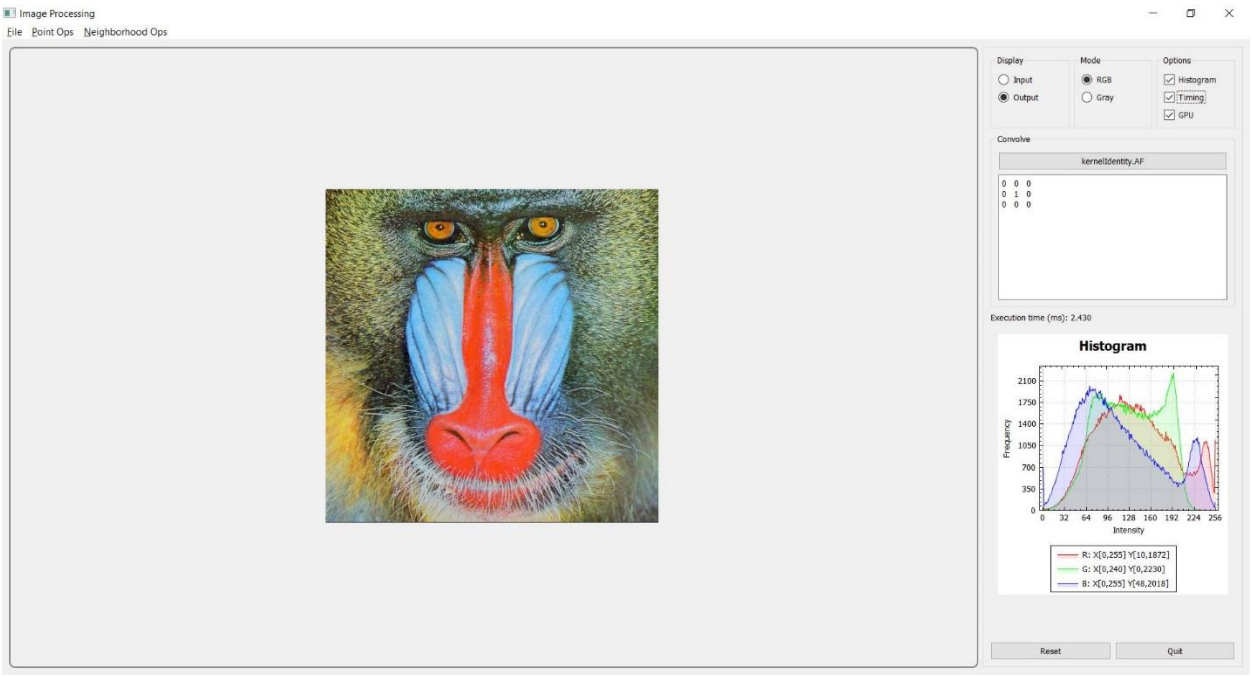


Figure 1 – GUI Application

2. The GUI

As previously stated, the GUI part of this program is written using the Qt Framework. Qt has many advantages; the most important advantage is that the developer writes code once and targets multiple platforms. The GUI of the program has three main components: the menu bar, the image display, and the control panel. The menu bar has three menus on its own: File, Point Ops, and Neighborhood Ops. The File menu allows the user to open an image, save an image, or quit the program. The Point Ops menu show the filters that are computed on a single pixel of the image. The point operations include the following filters: Threshold, Clip, Quantization, Gamma Correction, Contrast Enhancement, Histogram Stretch, and Histogram Match. The neighborhood operations are filters that act on a neighborhood of pixels. These filters include: Error Diffusion, Blur, Sharpen, Median, Convolve, Blur w/ Single Pass, and Correlation. In this report, I will focus on the blurring, convolution, and correlation filters. The following figure shows the menu contents.

Figure 2 – The Menu Bar

The second component in the GUI is the image display. It takes most of the interface because it is the most important part: this is where we see the filters applied to the input image. The third component is the control panel, which is composed of five widgets: Display, Mode, Option, Filter Box, Histogram, and Exit Buttons. The Display widget has two radio buttons where the user can choose to display the input or the filtered output image. The Mode widget also has two option,

where the user can display the image in color (RGB) or grayscale (Gray). The Histogram has three checkboxes: Histogram, which allows the user to display or hide the histogram of the current image, Timing, which allows the user to time a specific operation on the image, and GPU, which applies the current filter using the GPU approach. The next widget is used to display the individual widgets for each filter, for example filters, or checkboxes. The Histogram widget is used to display the histogram of the image. The Exit Buttons widget contains two buttons, Reset and Exit. When clicked, the Reset button will place the filter options to their initial values. The Exit button will close the program. The following figure shows how the control panel looks.

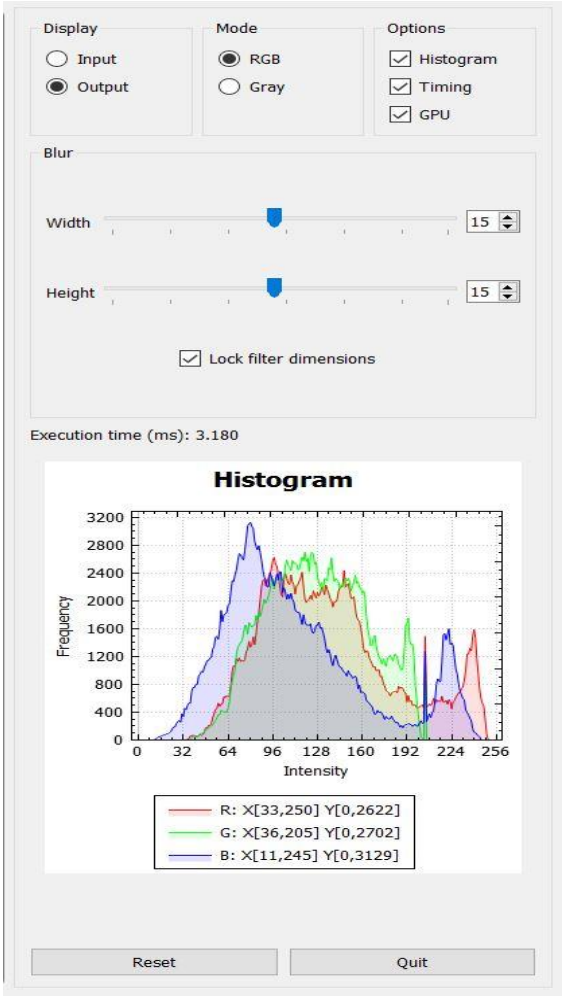


Figure 3 – The Control Panel

3. OpenGL

In the introduction of this report we stated that program was written such that it can apply the image processing operations to an image using two approaches: the CPU and the GPU approaches. There are six major elements in a graphics system:

1. Input devices
2. Central Processing Unit (CPU)
3. Graphics Processing Unit (GPU)
4. Memory
5. Frame Buffer
6. Output devices [2]

The elements in a graphics system can be visualized by the following figure.

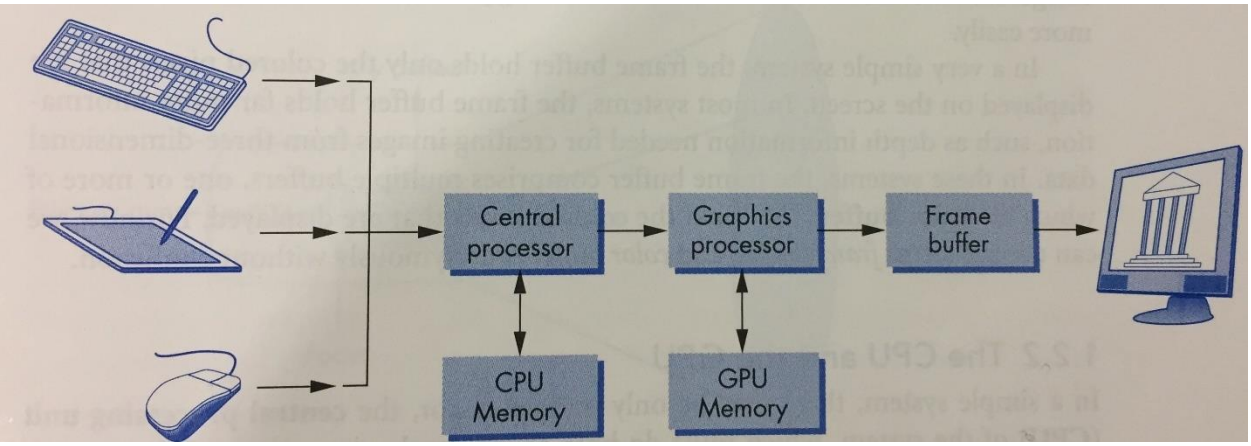


Figure 4 – A Graphics System [2]

The image that we see at the output device is an array of picture elements (pixels) produced b by the graphics system. This array is called a raster. The frame buffer is a part of memory where the pixels are stored. [2]

The main function of the processor is to take specifications of graphical primitives, such as lines or triangles, and assign values to the pixels in the frame buffer. In earlier graphics systems,

the frame buffer was part of the standard memory and could be accessed directly by the CPU, but in modern systems we have a special graphics processing unit (GPU) which carries out specific graphics functions. The main characteristics of the GPU are that it contains modules for graphical operations and it has a high degree of parallelism with over 100 processing units. [2]

Graphics systems that use GPU have pipeline architecture, which is composed of four major components:

1. Vertex Processor
2. Primitive Assembler
3. Rasterizer
4. Fragment Processor [2]

The following figure shows the geometric pipeline architecture of a modern graphics system.

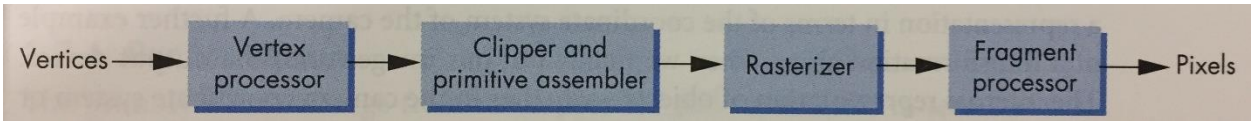


Figure 5 – Geometric Pipeline [2]

Every imaging process starts with a set of objects that comprise a set of graphical primitives. The graphical primitive that we used in this program is a triangle. We need three vertices to compose a triangle, and we need two triangles to compose a rectangular image. Consider the following figure.

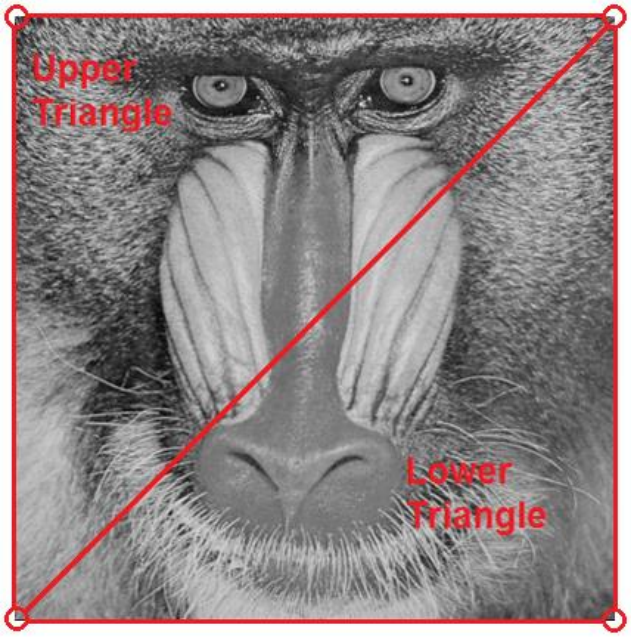


Figure 6 – Input Image to the Graphical System

We can see that the image contains four vertices. Each vertex is processed in the same manner to form the image in the frame buffer. These vertices form the upper and lower triangle of the image. Two triangles are enough to represent the image. Each block of the graphical system pipeline has specific tasks so that the filters are applied.

Vertex Processing

The main function of the vertex processor is to carry out coordinate transformation. The internal representation of objects must be represented in terms of the coordinate system of the display. A matrix is used to represent the changes in the coordinate system and successive changes are represented by concatenating (multiplying) individual matrices into a single matrix. [2] The vertex processing block is very useful when we work with three dimensional images. In our program, we will work with two dimensional images only and display them on a canvas, therefore

our vertex shader will only pass the coordinates to the next block. The following code snippet shows the vertex processor that we will use throughout this program.

```
1  #version 330
2
3  in  vec2    a_Position; // attribute variable: position vector
4  in  vec2    a_TexCoord; // attribute variable: texture coordinate
5  out vec2    v_TexCoord; // varying variable for passing texture coordinate to fragment shader
6
7  void main() {
8      gl_Position = vec4(a_Position, 0, 1);
9      v_TexCoord = a_TexCoord;
10 }
11
```

Figure 7 – The Vertex Shader

In the above figure, we can see in the first line that we declare the version of OpenGL that we are going to use, that is 3.30. The vertex shader takes two inputs and produces one output. The a_Position is an input attribute that defines the position vector of the vertex. The a_TexCoord is an input attribute containing the texture coordinate. The vertex shader will produce a varying texture coordinate called v_TexCoord in the above snippet. The main function of the vertex shader does not do much: it assigns the GL position to the input position, and sets the output texture coordinate to the input texture coordinate. This is simply passing the vertices to the next block in the pipeline, that is the Primitive Assembly,

Primitive Assembly

Because no imaging system can see the whole world we need to do some clipping. The objects in the clipping volume appear in the image, while the those that are outside are said to be clipped out. The clipping is done on a primitive-by-primitive basis rather than on a vertex-by-vertex basis therefore this block of the pipeline must assemble sets of vertices into primitive. In our program,

we use the triangle primitive. The output of this stage is a set of primitives whose projection can appear in the image. [2] In our case, the set of primitives is composed of two triangles.

Rasterization

The output primitives that emerge from the primitive assembler are still represented by their vertices and need to be converted to pixels for the frame buffer. In our case, the three vertices that define each triangle must be rasterized to determine which pixels in the frame buffer are inside the primitive. The output of the rasterizer is a set of fragments for each triangle. Each fragment is a potential pixel that carries some information with it: it contains the location of the pixel, the color as represented by the red, green, and blue components, and the depth or transparency. [2] These fragments are handled by the fragment shader.

Fragment Processing

The final block in the pipeline takes a fragment from the rasterizer and updates the pixel in the buffer. Texture mapping may alter the color of a pixel. Consider the following code snippet from the fragment shaders that we used.

```
#version 330

in  vec2      v_TexCoord;   // varying variable for passing texture coordinate from vertex shader

uniform float     u_Thr1;   // threshold value
uniform float     u_Thr2;   // threshold value
uniform sampler2D u_Sampler;    // uniform variable for the texture image

void main() {
    vec3 clr = texture2D(u_Sampler, v_TexCoord).rgb;

    clr = clamp(clr, u_Thr1, u_Thr2);

    gl_FragColor = vec4(clr, 1.0);
}
```

Figure 8 – The Fragment Shader

In figure 8, we see on line 3 that the fragment shader takes a texture coordinate as input. This is the coordinate of the fragment, or pixel, that this shader will handle and it comes from the output of the rasterizer. Depending on the operation that the shader will perform, it will take different inputs, but it will always get the texture image as input. The fragment shader in figure 8 is applying the threshold filter to the pixel therefore it takes the two threshold values as input as well. In the main function, the shader samples the corresponding pixel in the output image and puts its red, green, and blue components in a 3-element vector called "clr". This vector is then clamped between the two threshold values. The most important step is done on line 14. The "gl_FragColor" is the output of the fragment shader and it is a vector of four elements: the red value, the green value, the blue value, and the transparency value. This value is altered in the frame buffer which then displays the output image.

In our program, we implemented the GL pipeline using a base class by the name of "GLWidget.cpp". The initialization routine before the display loop is shown in the following figure.

```
43    void
44  GLWidget::initializeGL()
45  {
46      // initialize GL function resolution for current context
47      initializeGLFunctions();
48      // init vertex and fragment shaders
49      initShaders();
50      // init XY vertices in mesh and texture coords
51      initVertices();
52      // initialize vertex buffer and write positions to vertex shader
53      initBuffers();
54      // generate input texture name
55      glGenTextures(1, &m_inTexture);
56      // generate output texture name
57      glGenTextures(1, &m_outTexture);
58      // generate frame buffer
59      glGenFramebuffers(1, &m_fbo[PASS1]);
60      glGenFramebuffers(1, &m_fbo[PASS2]);
61      glGenTextures(1, &m_texture_fbo[PASS1]);
62      glGenTextures(1, &m_texture_fbo[PASS2]);
63      // set background color
64      glClearColor(1.0, 1.0, 1.0, 1.0);
65  }
```

Figure 9 – GLWidget initializeGL Function

In the initializeGL function on figure 9, we started by calling the “initializeGLFunctions()”. This function is used to start the GL function resolution for the current context. Next, we built the pipeline by initializing the vertex and fragment shaders, initializing the XY texture coordinates of each vertex, and initializing the frame buffer that will hold the pixels with their information. The pipeline needs the names of the input and output textures which are generated on lines 55 and 57 of figure 9. Furthermore, we generated the frame buffer using standard GL functions such as “glGenFramebuffers”. Finally, we set the background color by calling the “glClearColor” function and setting all its values to 1, meaning that the background is black.

The primitive assembler is told to compose triangles using the standard “glDrawArrays” function of OpenGL. This function takes three parameters: Glenum mode, Glint first, Glsizei count. The mode parameter tells the primitive assembler which type of primitive we want it to assemble given the array of vertices. The first parameter defines the starting index in the vertex array, and the count parameter defines the number of indices to be rendered. We used the following code to call the primitive assembler.

```
393     // draw triangles
394     glDrawArrays(GL_TRIANGLE_STRIP, 0, (GLsizei) m_numPoints);
395     glBindTexture(GL_TEXTURE_2D, 0);
396     glUseProgram(0);
```

Figure 10 – GLWidget paintGL Function

On line 394, we call the “glDrawArrays” and tell it to use GL_TRIANGLE_STRIP as the primitive type, start at index 0, and each vertex will be represented by “m_numPoints”. Then, we bind the texture using a GL_TEXTURE_2D target and the name of the texture is 0, which represents the default texture for each target texture. [3] The “glUseProgram” function is called to install the object as part of the current rendering state using the first (0^{th}) program handle.

4. Blur

Blurring is an image processing filter which uses a neighborhood of pixels to decide the output value of a specific pixel as the weighted average of its neighbors. We computed spatial blurring to remove small details in the input image, or for bridging small gaps in linear curves. The shape of an object is due to its edges. In blurring, we reduce the edge content and make the transition from one color to the other very smooth. The filter that we used for blurring is a mean filter which has some properties: it must be of odd size, the sum of all elements must add to one, and all the elements should be the same. [4] The following figure is an example of a weighted filter.

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Figure 11 – 3x3 Kernel

We can see that the above kernel satisfies all the properties mentioned above. The size of the kernel is clearly odd, it is a 3x3 filter. This means that there are 9 elements in total in the filter, each of which will have $1/9^{th}$ of the weight since they have to add up to one.

- ## CPU Implementation

The CPU implementation of the blurring function was done using one dimensional kernels. I will use an example of a 10x10 pixel input image, with a 3x3 kernel to show the concept of blurring using the CPU approach. Consider the following figure.

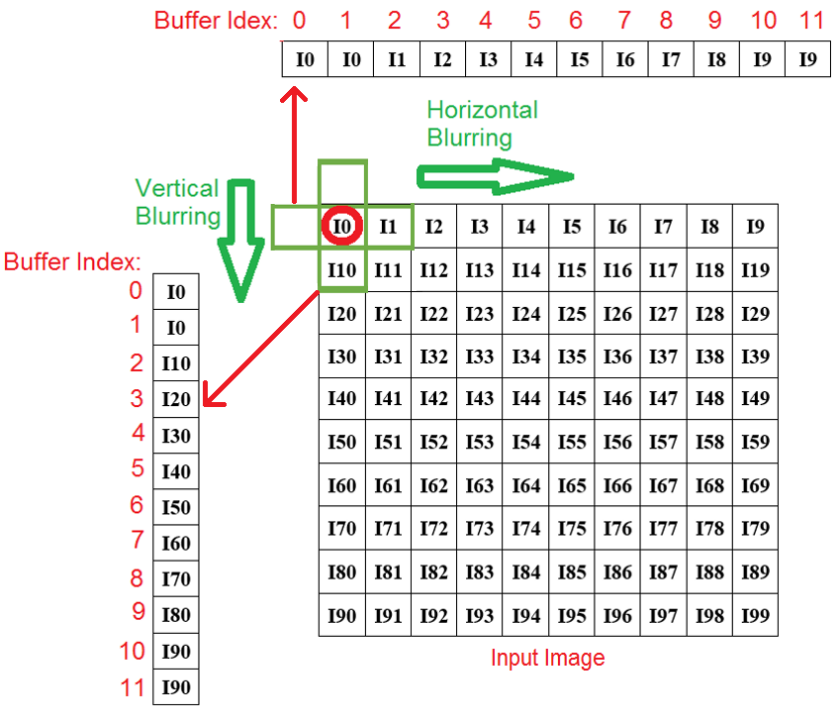


Figure 12 – Blurring (CPU Approach)

We computed the blurred output image by blurring the rows and the columns separately. This was done for efficiency purposes. It is faster to compute blurring in two passes rather than all at once. We used a buffer to hold the values of the rows and columns temporarily until we compute the output values for each pixel. The buffer is of size 12 because we need to take care of the pixels at the borders. Since the filter is of size 3, then the padding will be its size divided by two, minus the center pixel. Hence, in the above example, the padding will be one on each end of the input image. After we populate the buffer with the appropriate input image pixel values, we calculate the values for the output image. The first output pixel will be the sum of the values in the first 3 indices of the buffer divided by 3, which is the kernel size, so that we can get the weighted average. This is done on each row until the end of the image. Similarly, we blur the input image vertically one column at a time until we reach the end of the image. The blurring is done from left to right,

and from top to bottom. The following code snippet shows the blurring function that we implemented.

```
47      int  sizeBuff = len + kernelSize - 1;                   //calculate buffer size
48      uchar* circBuff;                                        //define circular buffer
49      circBuff = new uchar[sizeBuff];
50      if (!circBuff) return;                                  //stop if circular buffer is null
51
52      int pad = kernelSize / 2;                               //calcutate extra space needed for edge pixels
53      int v = 0;
54      for (; v < pad; v++) circBuff[v] = *p1;                 //fill up buffer from 0 to extra
55
56      len += pad;
57      for (; v < len; v++, p1 += steps) circBuff[v] = *p1;    //fill up buffer from extra to new width/height for every step
58      pad += len;                                             //increase padding
59      p1 -= steps;                                            //go one step back
60      for (int v = len; v < pad; v++) circBuff[v] = *p1;      //fill up buffer's new extra space
61
62      int sum = 0;                                            //sum of buffer's intensities
63      for (int v = 0; v < kernelSize; v++) sum += circBuff[v];//calculate sum from 0 to size
64      *p2 = sum / kernelSize;                                 //assign average to output image
65      for (int i = kernelSize; i < sizeBuff; i++, p2 += steps) {
66          sum += (circBuff[i] - circBuff[i - kernelSize]);    //calculate sum from size to buffer size
67          *p2 = sum / kernelSize;                             //assign average to output image
68      }
69
70      delete circBuff;
```

Figure 13 – Blurring CPU Function (I)

In the above function, we initialized a buffer of size calculated by adding the image size to the kernel size minus one because we are positioned at the center pixel. We calculate the padding value, and fill the buffer with the value of the current input image pixel for all padding. Then, we fill the buffer with the rest of the pixels in the row or column that we are taking into consideration. The value of the output pixel is calculated as the sum of all values up to kernel size from the left and from the right of the current pixel. This sum is then divided by the kernel size to get the weighted average. This sort of filtering is done on the whole input image using a separate function. The following code snipped shows how the above filter is applied to each row and column.

```
27      int type;
28      ChannelPtr<uchar> p1, p2, p3;
29      for (int ch = 0; IP_getChannel(I1, ch, p1, type); ch++) {
30          IP_getChannel(temp, ch, p2, type);
31          for (int v = 0; v < h; v++, p1+=w, p2+=w) filter(p1, w, 1, width, p2);  //filter horizontally one step at a time
32          p2 -= (total - 1);                                                      //go one step back
33          IP_getChannel(I2, ch, p3, type);
34          for (int v = 0; v < w; v++, p2++, p3++)  filter(p2, h, w, height, p3);  //filter vertically every w steps
35      }
```

Figure 14 – Blurring CPU Function (II)

In the above figure, we can see that the we go through every channel (RGB) of the input image and filter horizontally and vertically. The results of the horizontal blurring are saved on a temporary image and then combined with the vertical blurring in the output image.

The results of blurring using the CPU approach are shown in the following figures.

Figure 15 – Horizontal Blurring (25x3 Kernel)

The above figure shows only the horizontal blurring, which we achieved by using a kernel of size 25x3 kernel. This means that the image is blurred horizontally about 8 times more than vertically.

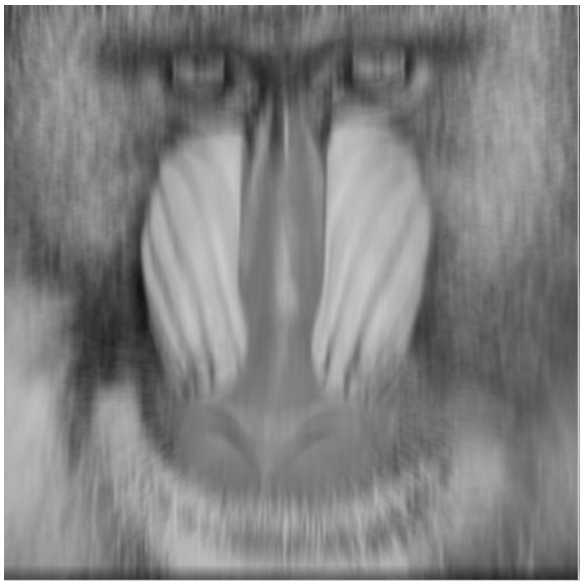


Figure 16 – Vertical Blurring (3x25 Kernel)

In this case, we are doing the opposite of figure 15. We are blurring vertically about 8 times more than horizontally by using a 3x25 kernel. We can clearly see the vertical blurring.

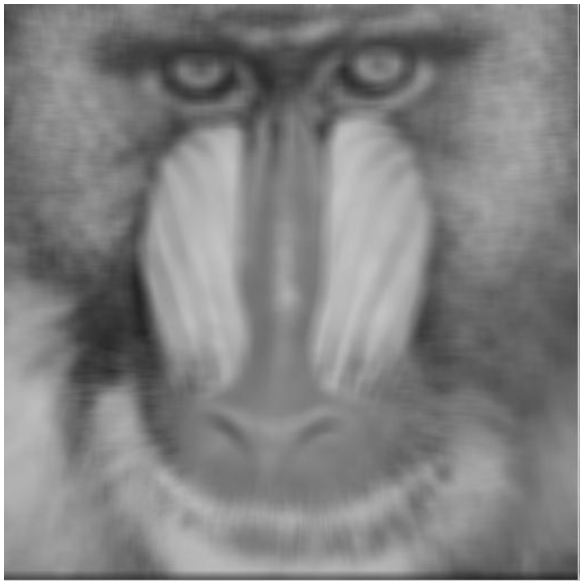


Figure 17 – Blurring (15x15 Kernel)

In this case, we are blurring equally on both directions using a 15x15 kernel.

- **GPU Implementation**

For the GPU implementation of the blurring filter, we are going to use single pass blurring, which means that we will blur both vertically and horizontally with one pass. In this case we need to functions to start the OpenGL pipeline: the initShader() and gpuProgram().

```
235  void
236  BlurSingle::initShader()
237  {
238      m_nPasses = 1;
239      // initialize GL function resolution for current context
240      initializeGLFunctions();
241
242      UniformMap uniforms;
243
244      // init uniform hash table based on uniform variable names and location IDs
245      uniforms["u_Wsize"  ] = WSIZE;
246      uniforms["u_Hsize"  ] = HSIZE;
247      uniforms["u_StepW"  ] = STEPW;
248      uniforms["u_StepH"  ] = STEPH;
249      uniforms["u_Sampler"] = SAMPLER;
250
251      // compile shader, bind attribute vars, link shader, and initialize uniform var table
252      g_mainWindowP->glw()->initShader(m_program[PASS1],
253          QString(":/hw2/vshader_blurSingle.glsl"),
254          QString(":/hw2/fshader_blurSingle.glsl"),
255          uniforms,
256          m_uniform[PASS1]);
257      m_shaderFlag = true;
258  }
```

Figure 18 – Blurring initShader Function

The “initShader” function is used to initialize the vertex and fragment shaders and declaring their parameters. The m_nPasses in this case is one since we want to apply the blurring in the horizontal and vertical direction at once. The “uniforms” UniformMap variable is used to hold the input parameters to the shaders. In our case, we have five paramters:

1. u_Wsize – The width of the kernel
2. u_Hsize – The height of the kernel
3. u_StepW – The horizontal step
4. u_StepH – The vertical step
5. u_Sampler – The input texture image

Next, we initialize the shaders, by calling the initShader function of the GLWidget base function. It takes the location of the vertex and fragment shader as a string indicating the names of the shaders as well. Furthermore, it takes the uniforms that we declared above.

The parameters of the fragment shader are passed using the “gpuProgram” function.

```
266     void
267   BlurSingle::gpuProgram(int pass)
268     {
269         int w_size = m_slider[0]->value();
270         int h_size = m_slider[1]->value();
271         if (w_size % 2 == 0) ++w_size;
272         if (h_size % 2 == 0) ++h_size;
273         glUseProgram(m_program[pass].programId());
274         glUniform1i(m_uniform[pass][WSIZE], w_size);
275         glUniform1i(m_uniform[pass][HSIZE], h_size);
276         glUniform1f(m_uniform[pass][STEPW], (GLfloat) 1.0f / m_width );
277         glUniform1f(m_uniform[pass][STEPH], (GLfloat) 1.0f / m_height);
278         glUniform1i(m_uniform[pass][SAMPLER], 0);
279     }
```

Figure 19 – Blurring gpuProgram Function

In this function, we get the width and height of kernel from the sliders in the GUI and make sure that they are of odd size. Then we tell OpenGL which program we are using and start passing the parameters. The first two parameters that we pass are of integer form and are the width and height of the kernel. Next, we pass the horizontal and vertical steps. The steps are defined as the distance from one pixel to the next. OpenGL expect a float number between 0 and 1 as the step and since our input images can have different widths and heights we need to specify the step size. For example, if the input image is 100x50, then the horizontal step will be $1/100^{th}$ and the vertical step will be $1/50^{th}$. Lastly, we pass the input texture. The vertex shader will simply pass the texture coordinates to the fragment shader. The fragment shader code is shown in the following figure.

```
1  #version 330
2
3  in      vec2      v_TexCoord;          // varying variable for passing texture coordinate from vertex shader
4  uniform int       u_Wsize;             // blur width value
5  uniform int       u_Hsize;             // blur height value
6  uniform float     u_StepW;             // horizontal step
7  uniform float     u_StepH;             // vertical step
8  uniform sampler2D u_Sampler;           // uniform variable for the texture image
9
10 void main() {
11
12     vec3 avg = vec3(0.0);              //average vector
13     vec2 tc  = v_TexCoord;             //texture coordinate
14     int  w2  = u_Wsize / 2;            //half width of the kernel
15     int  h2  = u_Hsize / 2;            //half height of the kernel
16
17     for(int i=-h2; i<=h2; ++i)
18         for(int j = -w2; j <= w2; ++j)
19             avg += texture2D(u_Sampler, vec2(tc.x + j*u_StepW, tc.y + i*u_StepH)).rgb;
20
21     avg /= (u_Wsize*u_Hsize);
22     gl_FragColor = vec4(avg, 1.0);     //put the average in the output image
23 }
```

Figure 20 – Blurring Fragment Shader

We can see that the input parameters to the fragment shader are declared in lines 3 – 8 as previously explained in the gpuProgram function. In the main function, we declare a 3-element vector called “avg” and initialize all its values to zero. This vector will hold the color information of each fragment. The 2-element vector called “tc” is simply a short name for the texture coordinate of the input fragment. The “w2” and “h2” integers are set to half the width and height of the kernel respectively since that is the number of pixels that from left to right, and from top to bottom, that we will use to form the neighborhood. The nested for loops, go from negative h2 to positive h2 and then from negative w2 to positive w2. This means that we will blur the rows first, from left to right, and then the columns, from top to bottom. We calculate the average by adding the sampled value of the input image at each coordinate starting at the top left corner and working our way to the bottom right corner. After we add all the values of the kernel elements, after the nested for loops we divide the value of the average by the size of the kernel. This average is then assigned to

the gl_FragColor with a transparency value of one, which means that it is fully opaque. The gl_FragColor will alter the value of the pixel in the frame buffer.

The following figure shows the result of blurring the input image with a 15x15 kernel using the GPU approach.

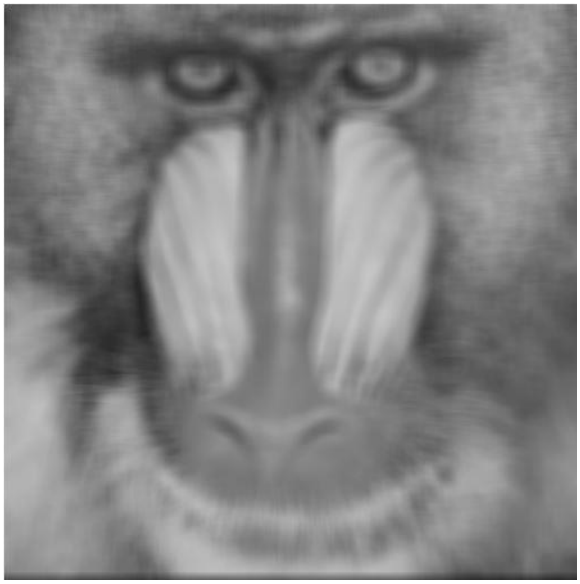


Figure 21 – Blurring (15x15 Kernel GPU)

We can see that the result is identical to the CPU approach. This is what we were expecting, because the only difference between the two approach is the efficiency and not the end result of the filter.

The timing analysis of the two approaches will be done in the Conclusion section of this report.

5. Convolve

Convolution is a mathematical operation which is fundamental to many common image processing operations. This operation is done by multiplying two arrays of values, usually of different sizes, but of the same dimensions, to produce a third array of values of the same dimensions. This operation is used in image processing to implement operators whose output pixel values are a combination of the input pixels. [5] Convolution can be used to perform several image processing operations such as Blurring, Sharpening, Edge Detection (Sobel), and Laplacian.

- **CPU Implementation**

The CPU implementation of the convolution operation is done using a two-dimensional filter as opposed to the one-dimensional filter used for blurring. The values of this kernel do not have to be the same. Actually, the different values of the filter produce the different results. The kernel is passed to the functions as an ".AF" file that we select through a file chooser.

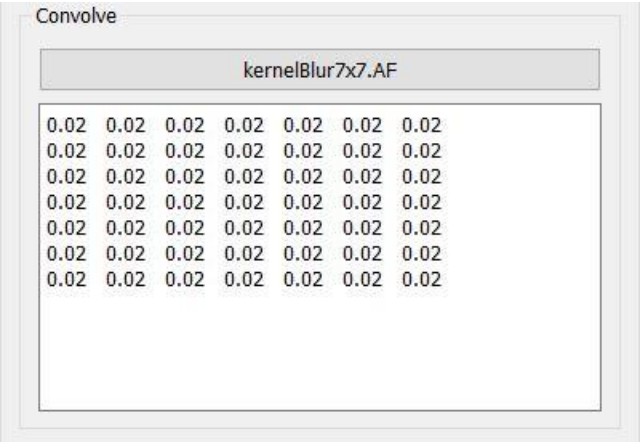


Figure 22 – Convolve Control Panel

In the control panel of the Convolve operation, we have a button which allows us to find the kernel that we are need and it will display the contents of the kernel on a text display.

The following example provides a visual representation of the convolution operation using the CPU approach.

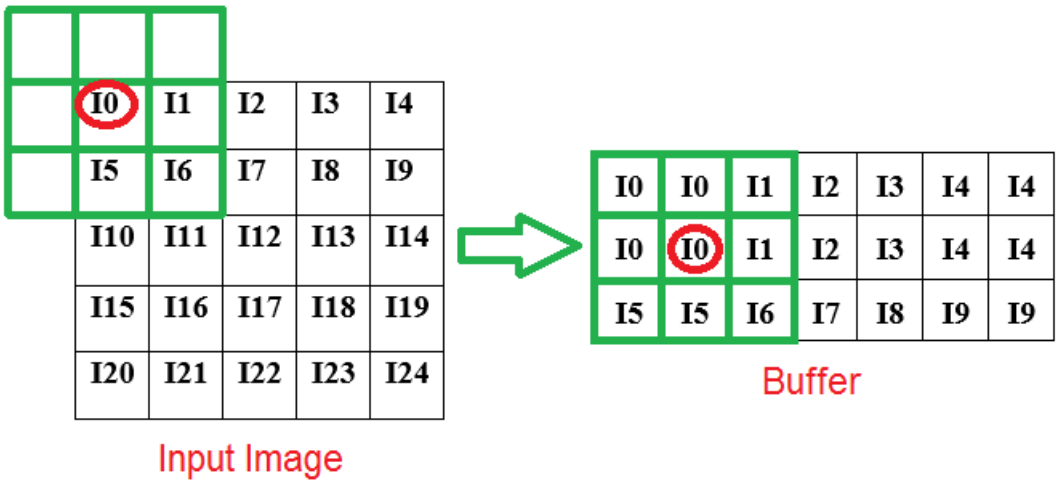


Figure 23 – Convolve (CPU Approach)

From the above figure, we can see that the output value of the first pixel can be determined using the following formula:

$$Out0 = I0 * K0 + I0 * K1 + I1 * K2$$
$$+ I0 * K3 + I0 * K4 + I1 * K5$$
$$+ I5 * K6 + I5 * K7 + I6 * K8$$

Where Out0 represents the value of the first output pixel, I_N represents the N^{th} input pixel, and K_N represents the N^{th} kernel value. We perform the convolution on the whole input image by sliding the kernel from left to right, and from top to bottom. The results of convolving the input image with different kernels is shown in the following figures.

```
1  IMPROC 7 7 AF
2  .02 .02 .02 .02 .02 .02 .02
3  .02 .02 .02 .02 .02 .02 .02
4  .02 .02 .02 .02 .02 .02 .02
5  .02 .02 .02 .02 .02 .02 .02
6  .02 .02 .02 .02 .02 .02 .02
7  .02 .02 .02 .02 .02 .02 .02
8  .02 .02 .02 .02 .02 .02 .02
9
```

Kernel

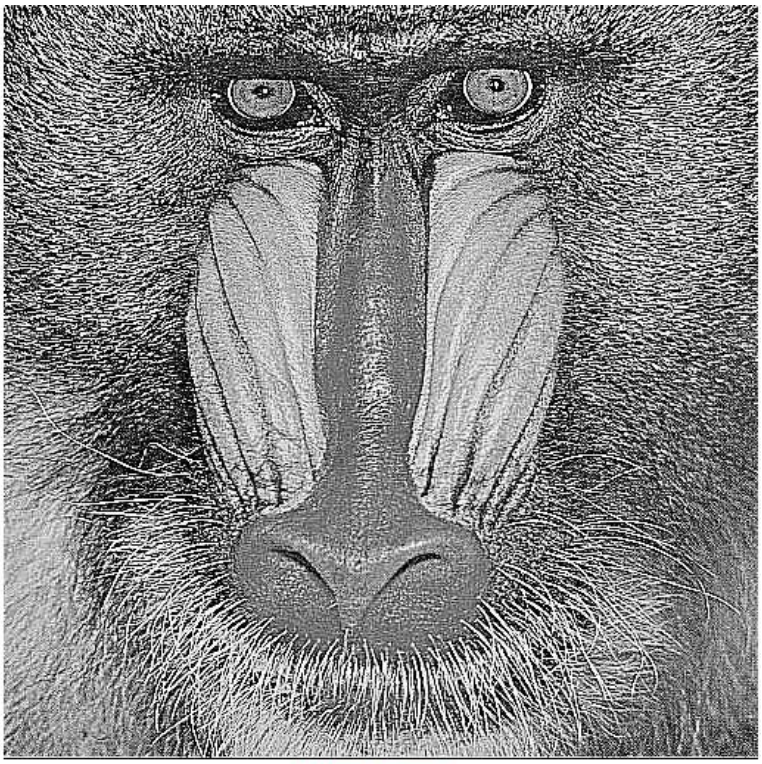
Output Image

Figure 24 – Convolve (Blur with 7x7 Kernel)

In the above figure, we can see that the image is blurred using a 7x7 unweighted filter. Since there are 49 elements in the kernel, then each of them has a value of 1/49 which is approximately 0.02.

```
1  IMPROC 3 3 AF
2  -1  -1  -1
3  -1   9  -1
4  -1  -1  -1
```

Kernel



Output Image

Figure 25 – Convolve (Sharpen 3x3 Kernel)

In this figure, we sharpened an image by convolving it with the appropriate filter. The kernel values add up to one, but there is a much higher weight on the central pixel, which produces the sharpening effect.

```
1  IMPROC 3 3 AF
2  -5 0 5
3  -5 0 5
4  -5 0 5
```

Kernel

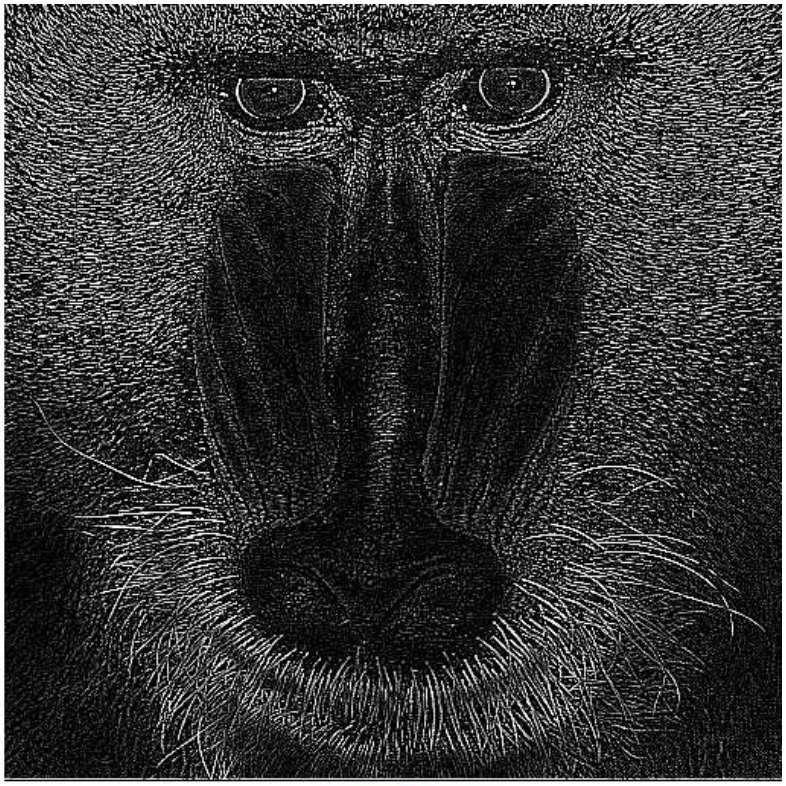
Output Image

Figure 26 – Convolve (Horizontal Edge Detection with 3x3 Kernel)

We applied an edge detection filter to the input image by convolving it with the appropriate kernel. We can see that the values of the kernel add up to zero, but we are eliminating the center column values, which means that the horizontal edge will be shown more than the vertical ones.

```
1  IMPROC 3 3 AF
2  -1 -1 -1
3  -1 8 -1
4  -1 -1 -1
```

Kernel



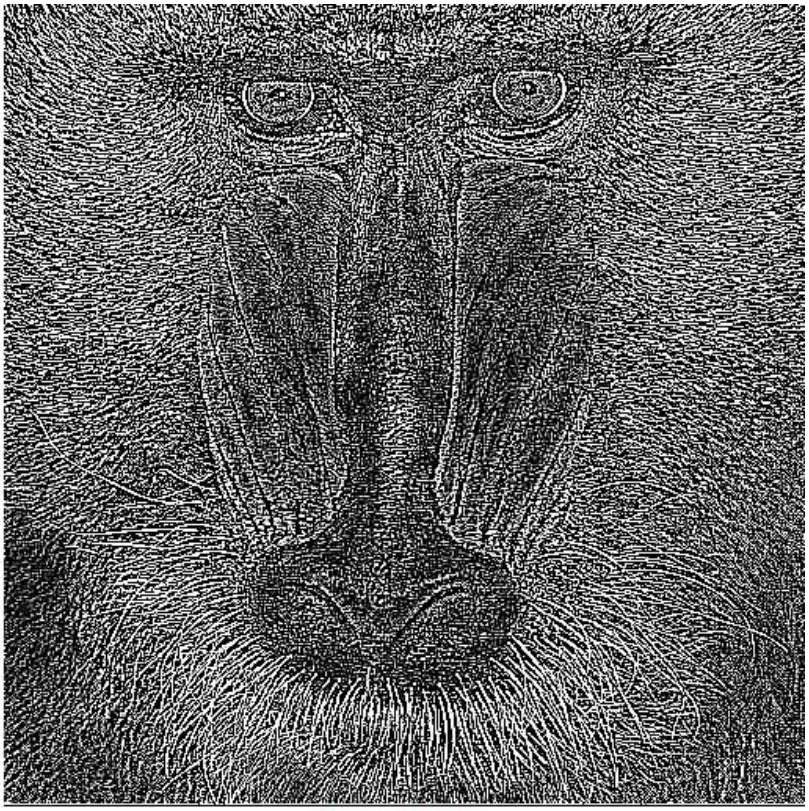
Output Image

Figure 27 – Convolve (Laplacian with 3x3 Kernel)

In this operation, we applied a 3x3 kernel whose values add up to zero, but the center value has a higher value than the rest of them. This produces the Laplacian effect. We can enhance the Laplacian effect by multiplying each value of the kernel by a constant.

```
1  IMPROC 3 3 AF
2  -5 -5 -5
3  -5 40 -5
4  -5 -5 -5
```

Kernel



Output Image

Figure 28 – Convolve (Laplacian with 3x3 Filter Multiplied by 5)

The enhanced Laplacian effect is accomplished by multiplying each value of the kernel by a constant. We can see that the effect is more visible then before since the central value has a much greater difference with the other elements compared to the previous kernel.

- **GPU Implementation**

The GPU implementation of the convolve operation is done similarly to the blurring function. We used the initShader function to initialize the vertex and the fragment shaders. Also, we defined the parameters that the shaders need in this function. The gpuProgram function was used to pass the parameters to the shaders. The convolve fragment shader get six inputs:

1. u_SizeW – The width of the kernel
2. u_SizeH – The height of the kernel
3. u_StepX – The horizontal step
4. u_StepY – The vertical step
5. u_Weights – The corresponding weights in the kernel
6. u_Sampler – The input texture image

In comparison to the blurring filter, the convolve filter takes one more parameter: the weights in the kernel. Previously, we did not need this parameter because we assumed that all elements of the kernel had the same value, but now it is different. The kernel's elements have different values.

```
216    void
217   Convolve::gpuProgram(int pass)
218    {
219        int w = m_kernel->width();
220        int h = m_kernel->height();
221        float weights[100];
222        int type;
223        ChannelPtr<float> p;
224
225        // get pointer to kernel values
226        IP_getChannel(m_kernel, 0, p, type);
227        //get kernel weights, left-right then top-bottom
228        int k = 0;
229        for (int i = 0; i < w; ++i)
230            for (int j = 0; j < h; ++j)
231                weights[k++] = *p++;
232
233        glUseProgram(m_program[pass].programId());
234        glUniform1i(m_uniform[pass][WSIZE], w);
235        glUniform1i(m_uniform[pass][HSIZE], h);
236        glUniform1f(m_uniform[pass][STEPX], (GLfloat) 1.0f / m_width);
237        glUniform1f(m_uniform[pass][STEPY], (GLfloat) 1.0f / m_height);
238        glUniform1fv(m_uniform[pass][WEIGHTS], 100, weights);
239        glUniform1i(m_uniform[pass][SAMPLER], 0);
240    }
```

Figure 29 – Convolve gpuProgram Function

The weights are passed to the to the fragment shader as a one-dimensional array of floats. The values of the array come from the m_kernel file which is seen as an image from the program. After the weights array is populated, we pass it to the fragment shader using the glUniform1fv function, which means that the variable is a 1 dimensional vector of floats.

```
#version 330

in      vec2      v_TexCoord;           // varying variable for passing texture coordinate from vertex shader

uniform int       u_SizeW;              //uniform int for kernel width
uniform int       u_SizeH;              //uniform int for kernel height
uniform float     u_StepX;              //input image horizontal step
uniform float     u_StepY;              //input image vertical step
uniform float     u_Weights[100];       //uniform array of float kernel values
uniform sampler2D u_Sampler;            //uniform variable for the texture image

void main() {
    vec3 conv = vec3(0.0);              //convolution vector
    vec2 tc  = v_TexCoord;              //texture coordinate
    int  w2  = u_SizeW / 2;             //half width of the kernel
    int  h2  = u_SizeH / 2;             //half height of the kernel
    int k = 0;                          //current kernel value

    //perform convolution: multiply texture value by kernel value
    for(int i=-h2; i<=h2; ++i)
        for(int j = -w2; j <= w2; ++j)
            conv += (texture2D(u_Sampler, vec2(tc.x + j*u_StepX, tc.y + i*u_StepY)).rgb) * u_Weights[k++];

    gl_FragColor = vec4(conv, 1.0);     //put the convolution in the output image
}
```

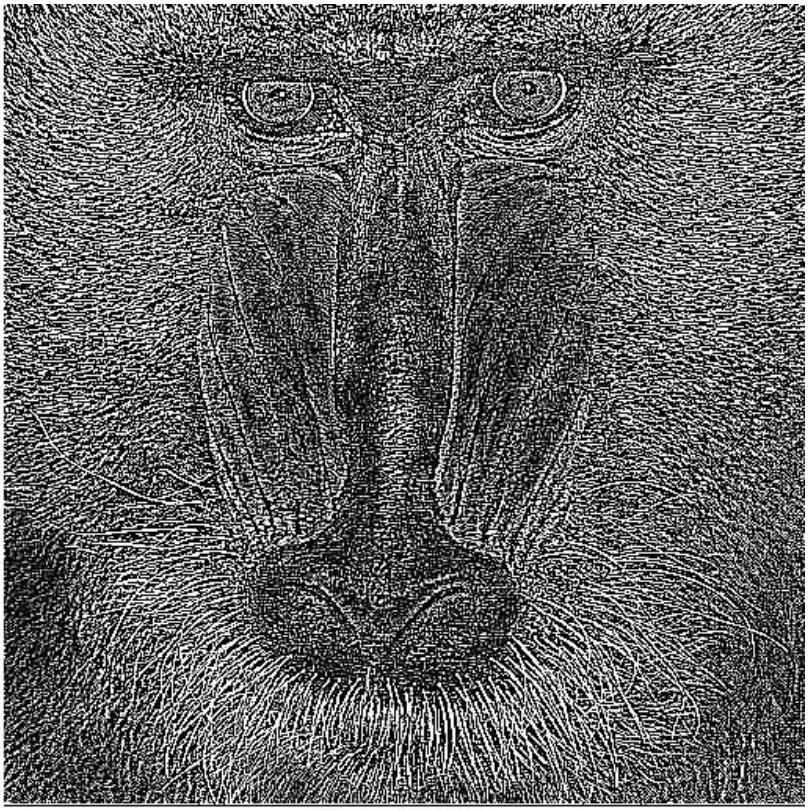
Figure 30 – Convolve Fragment Shader

The convolve fragment shader is very similar to the blurring fragment shader. The difference in this case is that the added convolved value is multiplied by the respective weight in the kernel rather than dividing the sum by the size of the kernel at the end. Finally, the convolved value is placed in the gl_FragColor which will alter the value of the pixel in the frame buffer.

The result of achieving the Laplacian effect on the input image is shown in the following figure.

```
1  IMPROC 3 3 AF
2  -5 -5 -5
3  -5 40 -5
4  -5 -5 -5
```

Kernel



Output Image

Figure 31 – Convolve (Laplacian with 3x3 Filter Multiplied by 5 GPU)

We can see that the result of applying a Laplacian filter to the input image using the GPU approach is identical to the result that we got by using the CPU implementation.

6. Correlate

Correlation is a method to determine the similarities between two images. We take into consideration an input image and a template image and perform the dot product of each pixel of the input image with every pixel of the template image. The best match between the two images is where the result is the brightest; i.e. closest to 1. The correlation number is calculated using the following formula:

$$Corr(u,v) = \frac{\sum T(x,y) \times I(x-u, y-v)}{\sqrt{\sum (I(x-u, y-v))^2}}$$

The correlation will be equal to 1 when $\sum T(x,y) \times I(x-u, y-v)$ equals $I(x-u, y-v)$. This means that the pixel values of the input image are the same as the pixel values from the template image. Usually, the correlation value does not usually equal exactly 1 because the template image is not an exact part of the input image. There are other factors that play a role in the pixel values, such as the angle from where the picture was taken, the brightness at the time it was taken, etc. Usually, the value that is closest to 1 will be the best guess of where the template image fits in the input image. In our case, however, we are using a template image that is a cropped version of the input image for simplicity.

The output image of this filter will be a combination of the input and template images. Once we find the location of the best match for the correlation, we proceed to add the pixels of the template image to the input image at that location. The pixels of that area will now be doubled. The pixels of the new image are then divided by two so that we obtain a darker image, except for the area where the template image fits. This area will have the original input image pixel values after the division. This will be the output image.

- **CPU Implementation**

The GUI of the correlation filter consists of two buttons and a display.

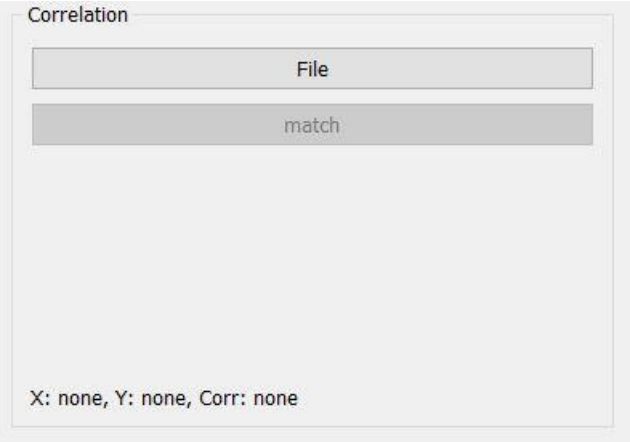


Figure 32 – Correlation Panel

From the above figure, we can see the "File" button which is used to select the template image. This image is a cropped version of the input image, which we cropped using an online cropping program: http://www169.lunapic.com/editor/?action=crop.

After the image is read, the correlation filter is applied. Consider the following figure.

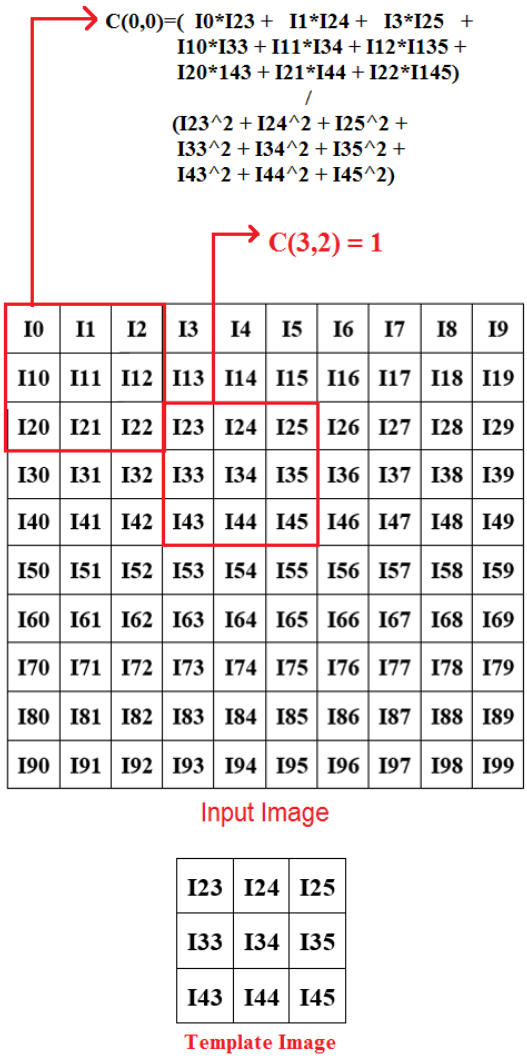


Figure 33 – Correlate (CPU Approach)

From the above figure, we can clearly see that at location (3,2) the correlation value will be one, since the nominator will be the same as the denominator in the correlation formula.

Unfortunately, we were unable to complete the CPU implementation of the correlation function.

- **GPU Implementation**

In the GPU implementation, we had to adjust the GLWidget.cpp file so that we could pass the template image to the shader program. We would like to recognize that Dong Liang helped us a great deal in the implementation of this filter. We added a function called “setTemplateTexture” to pass the template image to the shader.

```
void
GLWidget::setTemplateTexture(QImage &image)
{
    // convert jpg to GL formatted image
    QImage qImage = QGLWidget::convertToGLFormat(image);

    // init vars
    int w = qImage.width();
    int h = qImage.height();

    // bind texture
    glActiveTexture(GL_TEXTURE2);
    glBindTexture(GL_TEXTURE_2D, m_TemplateTexture);

    // set the texture parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    // upload to GPU
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w, h, 0, GL_RGBA, GL_UNSIGNED_BYTE, qImage.bits());
    //    glBindTexture(GL_TEXTURE_2D, 0);
}
```

Figure 34 – Correlate setTemplateTexture

The function takes a QImage as parameter and converts it into the GL format, then we proceed with passing the image to the shader using the standard OpenGL functions as shown in figure 34.

In the Correlate.cpp file, we pass the necessary parameters to the shader. These parameters are: u_StepX, u_StepY (the horizontal and vertical step sizes for the input image), u_SizeW_t, u_SizeH_T, u_StepX_T, u_StepY_T (the width and height of the template image, also the

horizontal and vertical step size of the template). We pass the sampler and the template as well. Furthermore, we pass the square root of the sum of the template image pixels. This is a float data type that is used for normalization. Using these variables, we perform the following fragment shader function:

```
void main() {
    vec4 corr = vec4(0.0);
    vec4 sum = vec4(0.0);
    vec2 tc  = v_TexCoord;
    int  sizeW  = u_SizeW_T / 2;
    int  sizeH  = u_SizeH_T / 2;
    int count = 0;

    for(int i=-sizeH; i<=sizeH; ++i) {
        for(int j=-sizeW; j<=sizeW; ++j) {
            float val = texture2D(u_Sampler, vec2(tc.x + j*u_StepX, tc.y + i*u_StepY));
            float val_T = texture2D(u_Sampler_T, vec2(0.5 + j*u_StepX_T, 0.5 + i*u_StepY_T));
            corr +=  val * val_T;
            sum += val*val;
        }
    }
    gl_FragColor = vec4(corr.rgb/sqrt(sum.rgb)/u_Sqrt_Sum_T, 1.0);
}
```

Figure 35 – Correlate Fragment Shader

In the above function, we sample the value of the input image and call it “val”. The value of the sampled texture image is placed in a variable called “val_T”. The correlation is the Cartesian product of the value of the input and template images. We also calculated the Cartesian product of the input image alone. The value of the correlation is ten divided by the square root of the sum, and then divided by the input square root that we passed to this function. The output of this function is a weird image composed of the correlation values for each pixels. We are interested in the position where the value is the highest, i.e. closest to 1. The correlation is then matched using a button and the “match” function that we wrote in the Correlate.cpp file. This means that we are reading from the fragment shader and applying a function through C++. The function takes the

image composed of the correlation values, scans through it, and when it finds the highest pixel value, it adds the template image. The final output is the half the intensity of the input image plus the template image at the position where the correlation is the highest. We get the following result for correlation.

Figure 36 – Correlate Result

From the result figure, we can see that the template, which is the left eye of the mandrill, is shown brighter on the image, while the rest of the image is darker because we are displaying only half of the pixel intensities. Also, on the panel to the left, we can see that the eye is shown below the match button, which was clicked to perform the correlation filter. We notice that the correlation value is 0.99 which is very close to 1 and a good place to decide where the template image fits in the input image. Furthermore, we can see that the correlation occurs at position (131, 35). More importantly, it only took my machine 23.9ms to compute the correlation filter.

7. Conclusion

This report showed the program completed during the Capstone II course. The main focus of this report was the show the approaches that we took to implement neighborhood operations to an image. There were two approach, the CPU approach where we instruct the CPU to perform the necessary operation on the image, and the GPU approach where we instruct the GPU directly to handle the operations. It was shown that the GPU approach has the advantage of parallelism since it handles multiple fragments (or pixels) at the same time while the CPU approach handles the pixels one at a time. We expect the GPU version of the program to work faster than the CPU version. The following timing analysis will show the results. My machine has a Intel Core i5-6200U CPU 2.3GHz Central Processing Unit and a Intel HD Graphics 520 Graphics Processing Unit.

Blur Timing

Kernel Size	CPU Time (ms)	GPU Time (ms)	CPU / GPU Ratio
3x3	22.8	2.7	8.4 times faster
25x3	23.7	4.05	5.8 times faster
3x25	23.5	4.11	5.7 times faster
15x15	23.7	4.2	5.6 times faster

Table 1 – Blur Timing

From table 1, we can see that the GPU version is about 6 times faster than the CPU version on my computer for the blurring operation. This is not a very distinct difference between the two approaches. This happened because I performed blurring using one-dimensional kernel with two passes in the CPU approach which speeds up the operation. Let's continue with the timing analysis for the convolve operation.

Convolve Timing

Kernel Size	CPU Time (ms)	GPU Time (ms)	CPU / GPU Ratio
3x3	70	1.7	41.2 times faster
7x7	802	6.1	131.5 times faster

Table 2 – Convolve Timing

In the case of convolution, the timing analysis shows a significant between the GPU and the CPU approaches. We can see that for a 7x7 Kernel I could achieve a speed of about 132 times faster using the GPU approach. This looks much better than the blur timing.

Correlate Timing

Unfortunately, we were not able to finish the correlation function using the CPU approach therefore we cannot compare the two approaches for this filter, but we can say that the GPU approach for correlation took approximately 24ms to complete.

The challenge that we faced during this project was to understand the pipeline of how OpenGL works to control the GPU. This challenge was overcome since we were able to implement and understand the logic behind the image processing filter. A challenge that remains, is the implementation of the correlation filter using the CPU approach. We are confident that we can overcome this challenge as well since we understand the logic and the theory behind the correlation filter. Overall, this was a very challenging but also fun project which taught us many new things regarding the inner capabilities of the machine that we tested the filters.

8. References

[1] OpenGL, "OpenGL," [Online]. Available: open.gl.

[2] E. Angel and D. Shreiner, Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL, 6th Edition, Boston, Massachusetts: Addison-Wesley of Pearson Education Inc., 2012.

[3] "glBindTexture," open.gl, [Online]. Available: https://www.opengl.org/sdk/docs/man/html/glBindTexture.xhtml.

[4] "Concept of Blurring," Tutorials Point, [Online]. Available: https://www.tutorialspoint.com/dip/concept_of_blurring.htm.

[5] "Convolution," [Online]. Available: http://homepages.inf.ed.ac.uk/rbf/HIPR2/convolve.htm.

9. Appendix

- **HW_blur.cpp**

```
void
filter(ChannelPtr<uchar>p1, int len, int steps, int kernelSize, ChannelPtr<uchar> p2);
void
HW_blur(ImagePtr I1, int width, int height, ImagePtr I2)
{
ImagePtr temp;                                          //use temporary image
IP_copyImageHeader(I1, temp);
IP_copyImageHeader(I1, I2);
int w = I1->width();
int h = I1->height();
int total = w * h;

if (width % 2 == 0) width += 1;                         //make sure width is odd
if (width > w)  width  = 2 * ((int) w / 2) - 1;//make sure slider width is less than input width

if (height % 2 == 0) height += 1;                               //make sure height is odd
if (height > h) height = 2 * ((int) h / 2) - 1;    //make sure slider height is less than input height

int type;
ChannelPtr<uchar> p1, p2, p3;
for (int ch = 0; IP_getChannel(I1, ch, p1, type); ch++) {
        IP_getChannel(temp, ch, p2, type);
        //filter horizontally one step at a time
        for (int v = 0; v < h; v++, p1+=w, p2+=w) filter(p1, w, 1, width, p2);
        p2 -= (total - 1);                                      //go one step back
        IP_getChannel(I2, ch, p3, type);
        //filter vertically every w steps
        for (int v = 0; v < w; v++, p2++, p3++)  filter(p2, h, w, height, p3);
}
}

void
filter(ChannelPtr<uchar>p1, int len, int steps, int kernelSize, ChannelPtr<uchar> p2)
{
//if filter size is 1
if (kernelSize == 1) {
```

```
        for (int v = 0; v < len; v++, p1 += steps, p2 += steps) *p2 = *p1;//input = output image
                return;                                 //no need to continue
}

int  sizeBuff = len + kernelSize - 1;                           //calculate buffer size
uchar* circBuff;                                        //define circular buffer
circBuff = new uchar[sizeBuff];
if (!circBuff) return;                                  //stop if circular buffer is null
int pad = kernelSize / 2;                       //calcutate extra space needed for edge pixels
int v = 0;
for (; v < pad; v++) circBuff[v] = *p1;         //fill up buffer from 0 to extra

len += pad;
//fill up buffer from extra to new width/height for every step
for (; v < len; v++, p1 += steps) circBuff[v] = *p1;
pad += len;                                             //increase padding
p1 -= steps;                                            //go one step back
for (int v = len; v < pad; v++) circBuff[v] = *p1;
        //fill up buffer's new extra space
        int sum = 0;                                    //sum of buffer's intensities
        for (int v = 0; v < kernelSize; v++) sum += circBuff[v];//calculate sum from 0 to size
        *p2 = sum / kernelSize;                         //assign average to output image
        for (int i = kernelSize; i < sizeBuff; i++, p2 += steps) {
                sum += (circBuff[i] - circBuff[i - kernelSize]);//calculate sum size to buffer size
                *p2 = sum / kernelSize;         //assign average to output image
        }
        delete circBuff;
}
```

- **fshader_blur.glsl**

```
#version 330

in              vec2    v_TexCoord;             // varying variable for passing texture
coordinate from vertex shader
uniform int     u_Wsize;                        // blur width value
uniform int             u_Hsize;                        // blur height value
uniform float   u_StepW;                        // horizontal step
uniform float   u_StepH;                        // vertical step
uniform         sampler2D u_Sampler;            // uniform variable for the texture image

void main() {

        vec3 avg = vec3(0.0);                   //average vector
        vec2 tc  = v_TexCoord;                          //texture coordinate
        int  w2  = u_Wsize / 2;                         //half width of the kernel
        int  h2  = u_Hsize / 2;                 //half height of the kernel

for(int i=-h2; i<=h2; ++i)
        for(int j = -w2; j <= w2; ++j)
                avg += texture2D(u_Sampler, vec2(tc.x + j*u_StepW, tc.y + i*u_StepH)).rgb;
        avg /= (u_Wsize*u_Hsize);
        gl_FragColor = vec4(avg, 1.0);          //put the average in the output image
}
```

- **HW_convolve.cpp**

```
void
HW_convolve(ImagePtr I1, ImagePtr Ikernel, ImagePtr I2) {
   IP_copyImageHeader(I1, I2);
   int w = I1->width();
   int h = I1->height();
   int total = w*h;
   int type, y, x, ch, i, j;
         ChannelPtr<uchar> p1, p2, end, cursor;
         end = p1 + total;

         // Kernel information
         int wKernel = Ikernel->width();
         int hKernel = Ikernel->height();

         // Output = Input if kernel is of size 1x1
         if (wKernel == 1 && hKernel == 1) {
                  IP_copyImage(I1, I2);
                  return;
         }
         ChannelPtr<float> pKernel;
         IP_getChannel(Ikernel, 0, pKernel, type);

         // Padding and Buffer information
         int wPadding = wKernel / 2;
         int hPadding = hKernel / 2;
   int sizeBuff = w + wPadding * 2;

   // buffer allocation
   uchar** buff = new uchar*[hKernel];
   for(i = 0; i < hKernel; i++)
                  buff[i] = new uchar[sizeBuff];

   for(ch = 0; IP_getChannel(I1, ch, p1, type); ch++) {
      IP_getChannel(I2, ch, p2, type);
      cursor = p1;                                                                    //
always start at first pixel in the row
      // populate buffer
      for(i = 0; i < hPadding; i++) {
```

```
        for(j = 0; j < wPadding; j++)
                                buff[i][j] = *cursor;                                   // left
padding
        for(j = wPadding; j < w+ wPadding; j++)
                                buff[i][j] = *cursor++;
        // in between padding
        cursor--;                                                                       //
move cursor one step back to accomodate for last pixel
        for(j = w+ wPadding; j < sizeBuff; j++)
                                buff[i][j] = *cursor;                                   // right
padding
    }

                cursor = p1;
        // move cursor back to first pixel in the current row
    for(i = hPadding; i < hKernel; i++) {
        for(j = 0; j < wPadding; j++)
                                buff[i][j] = *cursor;                                   // left
padding
        for(j = wPadding; j < w+ wPadding; j++)
                                buff[i][j] = *cursor++;
        // in between padding
        cursor--;                                                                       //
move cursor one step back to accomodate for last pixel
                        for(j = w+ wPadding; j < sizeBuff; j++)
                                buff[i][j] = *cursor;                                   // right
padding
                        cursor++;
        // move cursor to next buffer row
    }

    //perform convolution
    for(y = 0; y < h; y++) {
        for (x = 0; x < w; x++) {
            float conv = 0;
            for (i = 0; i < hKernel; i++) {
                for (j = 0; j < wKernel; j++)
                                                conv += buff[i][j+x] * (*pKernel++);// convolve
            }
```

```
            *p2++ = CLIP(conv, 0, MaxGray);                    // assign convoluted
0-255 value to output
                                pKernel -= (wKernel * hKernel);                    //
move kernelPtr back to first value in the first row
          }
          for (i = 0; i<hKernel -1; i++)
             for (j = 0; j<sizeBuff; j++)
                                    buff[i][j] = buff[i+1][j];                     //
move all row buffer one index up
          for(j = 0; j < wPadding; j++)
                                buff[hKernel -1][j] = *cursor;                     // left padding
          for(j = wPadding; j < w+ wPadding; j++)
                                buff[hKernel -1][j] = *cursor++;                   // in between
padding
          cursor--;                                                                //
move cursor one step back to accomodate for last pixel
                          for(j = w+ wPadding; j < sizeBuff; j++)
                                buff[hKernel -1][j] = *cursor;                     // right
padding
                          cursor++;
          //move cursor to next buffer row
                          if (y >= (h - hPadding - 2)) cursor -= w;          //position curser for
next iteration
       }
    }
          //take care of memory leakage
    for (i = 0; i < hKernel; i++) delete[] buff[i];
          delete[] buff;
}
```

- **fshader_convolve.glsl**

```
#version 330

in              vec2    v_TexCoord;             // varying variable for passing texture
coordinate from vertex shader

uniform int     u_SizeW;                        //uniform int for kernel width
uniform int             u_SizeH;                        //uniform int for kernel height
uniform float   u_StepX;                        //input image horizontal step
uniform float   u_StepY;                        //input image vertical step
uniform float   u_Weights[100];             //uniform array of float kernel values
uniform         sampler2D u_Sampler;                    //uniform variable for the texture
image

void main() {
        vec3 conv = vec3(0.0);                          //convolution vector
        vec2 tc  = v_TexCoord;                          //texture coordinate
        int  w2  = u_SizeW / 2;                         //half width of the kernel
        int  h2  = u_SizeH / 2;                         //half height of the kernel
        int k = 0;                                              //current kernel value

        //perform convolution: multiply texture value by kernel value
        for(int i=-h2; i<=h2; ++i)
                for(int j = -w2; j <= w2; ++j)
                        conv += (texture2D(u_Sampler, vec2(tc.x + j*u_StepX, tc.y +
i*u_StepY)).rgb) * u_Weights[k++];

        gl_FragColor = vec4(conv, 1.0);         //put the convolution in the output image
}
```

- **fshader_correlate.glsl**

```
#version 330

in vec2          v_TexCoord;  //varying variable for passing texture coordinate from vertex shader
uniform float     u_StepX;
uniform float     u_StepY;
uniform sampler2D u_Sampler;    // uniform variable for the texture image
uniform int       u_SizeW_T;
uniform int       u_SizeH_T;
uniform float     u_StepX_T;
uniform float     u_StepY_T;
uniform sampler2D u_Sampler_T;
uniform float     u_Sqrt_Sum_T;


void main() {
   vec4 corr = vec4(0.0);
   vec4 sum = vec4(0.0);
   vec2 tc  = v_TexCoord;
   int  sizeW  = u_SizeW_T / 2;
   int  sizeH  = u_SizeH_T / 2;
   int count = 0;

   for(int i=-sizeH; i<=sizeH; ++i) {
      for(int j=-sizeW; j<=sizeW; ++j) {
         float val = texture2D(u_Sampler, vec2(tc.x + j*u_StepX, tc.y + i*u_StepY));
         float val_T = texture2D(u_Sampler_T, vec2(0.5 + j*u_StepX_T, 0.5 + i*u_StepY_T));
         corr +=  val * val_T;
         sum += val*val;
      }
   }
   gl_FragColor = vec4(corr.rgb/sqrt(sum.rgb)/u_Sqrt_Sum_T, 1.0);
}
```

- **GLWidget.cpp**

```
// ========================================================================
// Computer Graphics Homework Solutions
// Copyright (C) 2015 by George Wolberg
//
// GLWidget.cpp - GLWidget class. Base class of homework solutions.
//
// Written by: George Wolberg, 2015
// ========================================================================

#include "MainWindow.h"
#include "GLWidget.h"

extern MainWindow *g_mainWindowP;

// shader ID
enum { PASSTHROUGH_SHADER };

// uniform ID
enum { SAMPLER };

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// GLWidget::GLWidget:
//
// GLWidget constructor.
//
//
GLWidget::GLWidget(QWidget *parent) : QGLWidget(parent),
m_imageFlag(false)
{}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// GLWidget::initializeGL:
//
// Initialization routine before display loop.
// Gets called once before the first time resizeGL() or paintGL() is called.
//
void
GLWidget::initializeGL()
```

```
{
	// initialize GL function resolution for current context
	initializeGLFunctions();

	// init vertex and fragment shaders
	initShaders();

	// init XY vertices in mesh and texture coords
	initVertices();

	// initialize vertex buffer and write positions to vertex shader
	initBuffers();

	// generate input texture name
	glGenTextures(1, &m_inTexture);

	// generate output texture name
	glGenTextures(1, &m_outTexture);

	glGenTextures(1, &m_TemplateTexture);

	// generate frame buffer
	glGenFramebuffers(1, &m_fbo[PASS1]);
	glGenFramebuffers(1, &m_fbo[PASS2]);
	glGenTextures(1, &m_texture_fbo[PASS1]);
	glGenTextures(1, &m_texture_fbo[PASS2]);

	glClearColor(1.0, 1.0, 1.0, 1.0);	// set background color

//	GLint value;
//
	glGetIntegerv(GL_MAX_FRAGMENT_UNIFORM_COMPONENTS, &value);
//	qDebug() <<
value;
//
	glGetIntegerv(GL_MAX_FRAGMENT_UNIFORM_VECTORS, &value);
//	qDebug() <<
value;

}
```

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// GLWidget::initVertices:
//
// Initialize XY vertices in quad and texture coords
//
void
GLWidget::initVertices() {

	// init geometry data
	// two triangles that form a quad
	m_points.push_back(vec2(-1.0f, -1.0f));
	m_points.push_back(vec2(-1.0f, 1.0f));
	m_points.push_back(vec2(1.0f, -1.0f));
	m_points.push_back(vec2(1.0f, 1.0f));


	// init texture coordinate
	for (int i = 0; i<m_points.size(); ++i) {
		m_texCoord.push_back((m_points[i] + vec2(1.0f, 1.0f)) / 2.0f);
	}
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// GLWidget::initVertexBuffer:
//
// Initialize vertex buffer.
//
void
GLWidget::initBuffers() {


	m_numPoints = (int)m_points.size();		// save number of vertices

	glGenBuffers(1, &m_vertexBuffer);
	// bind vertex buffer to the GPU and copy the vertices from CPU to GPU
	glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer);
	glBufferData(GL_ARRAY_BUFFER, m_numPoints * sizeof(vec2), &m_points[0],
GL_STATIC_DRAW);
```

```
	glGenBuffers(1, &m_texCoordBuffer);
	// bind color buffer to the GPU and copy the colors from CPU to GPU
	glBindBuffer(GL_ARRAY_BUFFER, m_texCoordBuffer);
	glBufferData(GL_ARRAY_BUFFER, m_numPoints * sizeof(vec2), &m_texCoord[0],
GL_STATIC_DRAW);

}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// GLWidget::setInTexture:
//
// Initialize texture for input image.
//
void
GLWidget::setInTexture(QImage &image)
{
	// convert jpg to GL formatted image
	QImage qImage = QGLWidget::convertToGLFormat(image);

	// init vars
	m_imageW = qImage.width();
	m_imageH = qImage.height();

	// bind texture
	glActiveTexture(GL_TEXTURE0);
	glBindTexture(GL_TEXTURE_2D, m_inTexture);

	// set the texture parameters
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
	// upload to GPU
	glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, m_imageW, m_imageH, 0,
GL_RGBA, GL_UNSIGNED_BYTE, qImage.bits());
	m_imageFlag = true;
}
```

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// GLWidget::setOutTexture:
//
// Initialize texture for output image.
//
void
GLWidget::setOutTexture(QImage &image) {
	// convert jpg to GL formatted image
	QImage qImage = QGLWidget::convertToGLFormat(image);

	// init vars
	m_imageW = qImage.width();
	m_imageH = qImage.height();

	// bind texture
	glActiveTexture(GL_TEXTURE1);
	glBindTexture(GL_TEXTURE_2D, m_outTexture);

	// set the texture parameters
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
	// upload to GPU
	glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, m_imageW, m_imageH, 0,
GL_RGBA, GL_UNSIGNED_BYTE, qImage.bits());
}

void
GLWidget::allocateTextureFBO(int w, int h)
{

	// bind texture
	glBindFramebuffer(GL_FRAMEBUFFER, m_fbo[PASS1]);

	glActiveTexture(GL_TEXTURE3);
	glBindTexture(GL_TEXTURE_2D, m_texture_fbo[PASS1]);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

```
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
	glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
	glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
	glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w, h, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);

	glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
		GL_TEXTURE_2D, m_texture_fbo[PASS1], 0);


	glBindFramebuffer(GL_FRAMEBUFFER, m_fbo[PASS2]);
	glActiveTexture(GL_TEXTURE4);
	glBindTexture(GL_TEXTURE_2D, m_texture_fbo[PASS2]);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
	glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
	glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
	glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w, h, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);

	glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
		GL_TEXTURE_2D, m_texture_fbo[PASS2], 0);
	glBindFramebuffer(GL_FRAMEBUFFER, 0);
}


void
GLWidget::setTemplateTexture(QImage &image)
{
	// convert jpg to GL formatted image
	QImage qImage = QGLWidget::convertToGLFormat(image);

	// init vars
	int w = qImage.width();
	int h = qImage.height();
```

```
	// bind texture
	glActiveTexture(GL_TEXTURE2);
	glBindTexture(GL_TEXTURE_2D, m_TemplateTexture);

	// set the texture parameters
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
	// upload to GPU
	glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w, h, 0, GL_RGBA,
GL_UNSIGNED_BYTE, qImage.bits());
	//    glBindTexture(GL_TEXTURE_2D, 0);
}

void GLWidget::setCorrOutTexture(QImage &image) {
	QImage qImage = QGLWidget::convertToGLFormat(image);

	// init vars
	int w = qImage.width();
	int h = qImage.height();

	glActiveTexture(GL_TEXTURE3);
	glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w, h, 0, GL_RGBA,
GL_UNSIGNED_BYTE, qImage.bits());
}
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// GLWidget::initShader:
//
// Initialize vertex and fragment shaders.
//
void
GLWidget::initShader(QGLShaderProgram &program, QString vshaderName, QString
fshaderName, UniformMap &uniformsMap, int *uniforms)
{
	// compile vertex shader
	bool flag = program.addShaderFromSourceFile(QGLShader::Vertex, vshaderName);
	if (!flag) {
```

```
                QMessageBox::critical(0, "Error", "Vertex shader error: " + vshaderName + "\n"
+
                        program.log(), QMessageBox::Ok);
                exit(-1);
        }

        // compile fragment shader
        if (!program.addShaderFromSourceFile(QGLShader::Fragment, fshaderName)) {
                QMessageBox::critical(0, "Error", "Fragment shader error: " + fshaderName +
"\n" +
                        program.log(), QMessageBox::Ok);
                exit(-1);
        }

        // bind the attribute variable in the glsl program with a generic vertex attribute index;
        // values provided via ATTRIB_VERTEX will modify the value of "a_position")
        glBindAttribLocation(program.programId(), ATTRIB_VERTEX, "a_Position");
        glBindAttribLocation(program.programId(), ATTRIB_TEXCOORD, "a_TexCoord");

        // link shader pipeline; attribute bindings go into effect at this point
        if (!program.link()) {
                QMessageBox::critical(0, "Error", "Could not link shader: " + vshaderName +
"\n" +
                        program.log(), QMessageBox::Ok);
                exit(-1);
        }


        // iterate over all uniform variables; map each uniform name to shader location ID
        std::map<QString, GLuint>::iterator iter;
        for (iter = uniformsMap.begin(); iter != uniformsMap.end(); ++iter) {
                QString uniformName = iter->first;
                GLuint  uniformID = iter->second;

                // get storage location
                uniforms[uniformID] = glGetUniformLocation(program.programId(),
                        uniformName.toStdString().c_str());
                if ((int)uniforms[uniformID] < 0) {
                        qDebug() << "Failed to get the storage location of " + uniformName;
                        exit(-1);
```

```
                }
        }
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// GLWidget::initShaders:
//
// Initialize vertex and fragment shaders.
//
void
GLWidget::initShaders()
{

        UniformMap uniforms;

        // init uniform hash table based on uniform variable names and location IDs
        uniforms["u_Sampler"] = SAMPLER;

        QString v_name = ":/vshader_passthrough";
        QString f_name = ":/fshader_passthrough";

#ifdef __APPLE__
        v_name += "_Mac";
        f_name += "_Mac";
#endif
        // compile shader, bind attribute vars, link shader, and initialize uniform var table
        initShader(m_program, v_name + ".glsl", f_name + ".glsl", uniforms, m_uniform);

        g_mainWindowP->imageFilter(THRESHOLD)->initShader();
        g_mainWindowP->imageFilter(CLIP)->initShader();
        g_mainWindowP->imageFilter(QUANTIZE)->initShader();
        g_mainWindowP->imageFilter(GAMMA)->initShader();
        g_mainWindowP->imageFilter(CONTRAST)->initShader();
        g_mainWindowP->imageFilter(HISTOSTRETCH)->initShader();
        g_mainWindowP->imageFilter(HISTOMATCH)->initShader();
        g_mainWindowP->imageFilter(ERRDIFFUSION)->initShader();
        g_mainWindowP->imageFilter(BLUR)->initShader();
        g_mainWindowP->imageFilter(SHARPEN)->initShader();
        g_mainWindowP->imageFilter(MEDIAN)->initShader();
        g_mainWindowP->imageFilter(CONVOLVE)->initShader();
```

```
	g_mainWindowP->imageFilter(BLURSINGLE)->initShader();
	g_mainWindowP->imageFilter(CORRELATE)->initShader();


}


void
GLWidget::setViewport(int w, int h, int ww, int hh)
{
	// compute orthographic projection from viewing coordinates
	m_projection.setToIdentity();
	m_projection.ortho(-1.0f, 1.0f, -1.0f, 1.0f, -1.0f, 1.0f);
	// center glwidget if needed when the image is smaller than screen
	int new_ww = w;
	int new_hh = h;
	if (w > ww || h > hh) {
		// compute aspect ratio of the image and screen
		float screenRatio = (float)ww / hh;
		float imageRatio = (float)w / h;
		if (imageRatio > screenRatio) {
			new_hh = (int)(((float)ww / w)*h);
			new_ww = ww;
		}
		else {
			new_ww = (int)(((float)hh / h)*w);
			new_hh = hh;
		}
	}
	resize(new_ww, new_hh);
	// move glwidget to the center
	int dx = (ww - new_ww) / 2;
	int dy = (hh - new_hh) / 2;
	move(dx, dy);

}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// GLWidget::resizeGL:
//
```

```
// Resize event handler.
// The input parameters are the window width (w) and height (h).
//
void
GLWidget::resizeGL(int w, int h) {
	// save window dimensions
	m_winW = w;
	m_winH = h;
	glViewport(0, 0, w, h);
	int imgW = g_mainWindowP->imageIn()->width();
	int imgH = g_mainWindowP->imageIn()->height();
	setViewport(imgW, imgH, g_mainWindowP->glFrameW(), g_mainWindowP-
>glFrameH());


}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// GLWidget::paintGL:
//
// Update GL scene.
//
void
GLWidget::paintGL()
{

	// clear canvas with background color
	glClear(GL_COLOR_BUFFER_BIT);

	// error checking (nothing to display)
	if (!m_imageFlag) return;

	// enable buffer to be copied to the attribute vertex variable and specify data format
	glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer);
	glEnableVertexAttribArray(ATTRIB_VERTEX);
	glVertexAttribPointer(ATTRIB_VERTEX, 2, GL_FLOAT, false, 0, NULL);

	// enable buffer to be copied to the attribute texture coord variable and specify data
format
	glBindBuffer(GL_ARRAY_BUFFER, m_texCoordBuffer);
```

```
        glEnableVertexAttribArray(ATTRIB_TEXCOORD);
        glVertexAttribPointer(ATTRIB_TEXCOORD, 2, GL_FLOAT, false, 0, NULL);

        glUseProgram(m_program.programId());      // passthrough glsl progam

        int selection = g_mainWindowP->gpuFlag() * 2 + g_mainWindowP->isInput();
        switch (selection) {
        case 0: // display out image generated by CPU
                glUniform1i(m_uniform[SAMPLER], 1);
                break;
        case 1:
        case 3: // display input image
                glUniform1i(m_uniform[SAMPLER], 0);
                break;
        case 2: // display rendered texture by GPU filter
                int n = g_mainWindowP->gpuPasses();
                if (n == 1) glUniform1i(m_uniform[SAMPLER], 3);
                else glUniform1i(m_uniform[SAMPLER], 4);

                break;
        }

        // draw triangles
        glDrawArrays(GL_TRIANGLE_STRIP, 0, (GLsizei)m_numPoints);
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// GLWidget::applyFilterGPU:
//
//  Apply selected filter to input image by render to the texture in GPU.
//
void
GLWidget::applyFilterGPU(int nPasses)
{

        for (int pass = 0; pass<nPasses; ++pass) {
                glBindFramebuffer(GL_FRAMEBUFFER, m_fbo[pass]);
                glViewport(0, 0, m_imageW, m_imageH);
                glClearColor(0.0, 0.0, 0.0, 1.0);         // set background color
                glClear(GL_COLOR_BUFFER_BIT);
```

```
                // enable buffer to be copied to the attribute vertex variable and specify data
format
                glBindBuffer(GL_ARRAY_BUFFER, m_vertexBuffer);
                glEnableVertexAttribArray(ATTRIB_VERTEX);
                glVertexAttribPointer(ATTRIB_VERTEX, 2, GL_FLOAT, false, 0, NULL);

                // enable buffer to be copied to the attribute texture coord variable and specify
data format
                glBindBuffer(GL_ARRAY_BUFFER, m_texCoordBuffer);
                glEnableVertexAttribArray(ATTRIB_TEXCOORD);
                glVertexAttribPointer(ATTRIB_TEXCOORD, 2, GL_FLOAT, false, 0, NULL);

                g_mainWindowP->gpuProgram(pass);
                glDrawArrays(GL_TRIANGLE_STRIP, 0, (GLsizei)m_numPoints);

        }
        glUseProgram(0);

        if (!g_mainWindowP->timeFlag())
                setDstImage(nPasses - 1);

        glDisableVertexAttribArray(ATTRIB_TEXCOORD);
        glDisableVertexAttribArray(ATTRIB_VERTEX);
        glBindFramebuffer(GL_FRAMEBUFFER, 0);
}

void
GLWidget::setDstImage(int pass)
{
        glViewport(0, 0, m_imageW, m_imageH);
        ImagePtr I = IP_allocImage(3 * m_imageW, m_imageH, BW_TYPE);
        ChannelPtr<uchar> p = I[0];
        glBindFramebuffer(GL_FRAMEBUFFER, m_fbo[pass]);
        glReadPixels(0, 0, m_imageW, m_imageH, GL_RGB, GL_UNSIGNED_BYTE, &p[0]);
        glBindFramebuffer(GL_FRAMEBUFFER, 0);

        // uninterleave image
        ImagePtr ipImage = IP_allocImage(m_imageW, m_imageH, RGB_TYPE);
        IP_uninterleave(I, ipImage);
```

```
        // flip over the the image
        ImagePtr temp;
        IP_copyImageHeader(ipImage, temp);
        int type;
        int total = ipImage->height() * ipImage->width();
        ChannelPtr<uchar> p1, p2, endd;
        for (int ch = 0; IP_getChannel(ipImage, ch, p1, type); ch++) {
                for (int i = 1; i <= ipImage->height(); i++) {
                        IP_getChannel(temp, ch, p2, type);
                        p2 = p2 + total;
                        p2 = p2 - i*ipImage->width();
                        for (int j = 0; j < ipImage->width(); j++) {
                                *p2++ = *p1++;
                        }
                }
        }

        g_mainWindowP->setImageDst(temp);
        glViewport(0, 0, m_winW, m_winH);
}
```