

Edge Detection

and its implementation in C++

Written by MEA331 (T. Guttormsen, A. Andersen, J. Holst, M. Keilow and E. Kjæhr)
MED3 2008, Aalborg University

Introduction

This essay will explore how it is possible to make computers able to detect edges in digital images or video, and display them for the user, through the use of algorithms. It will describe the math used for calculating the filters, kernels or models, used for edge detection, and how to implement these in C++ using OpenFrameworks.

Definition of an edge

An edge is described as “the outside limit of an object, area or surface; a place or part farthest away from the center of something” (Oxford American Dictionary). In digital images, an edge can be seen as local changes in color intensity, also called “brightness discontinues”.

In an image without edges, the colors would either be continuous in intensity (called a gradient) or a flat image of one color. If an image does contain edges, these would be shown as local gradients (if the edges are, more or less, blurred) or a sudden change in brightness or color intensity as shown in illustration 1a and b.

General purpose and usage

Edge detection is used in many applications for computer vision, using image processing. The purpose is to identify and separate objects from one another, and also recognition of these shapes. One could imagine having an application, where a camera captures a DVD cover, and identifies it by calculating its position, then the position of the barcode and finally, by reading its barcode and comparing it to a database, getting its title and other relevant data. Other times it might be relevant to display the edges in a useful manner, i.e. the windows of a building (counting them), the extremities of an object (dimensions, orientation, distance, etc.).

The Basics of Edge Detection

An edge in a gray scale image occurs when there is a transition in gray level over an amount of pixels. A perfect edge would be a transition from black to white over one pixel as shown on figure 1. In many images, edges like these won't occur (unless it's a binary image). The transition will be blurred spreading the transition over more pixels, resulting in a slope-like profile of the gray level transition seen on figure 2. Here the edge is spread over more pixels and will show as a wider edge instead of the 1-pixel transition that shows as a 1-pixel edge. We might say the thickness of the edge is defined by the length of the ramp that in the gray level profile.

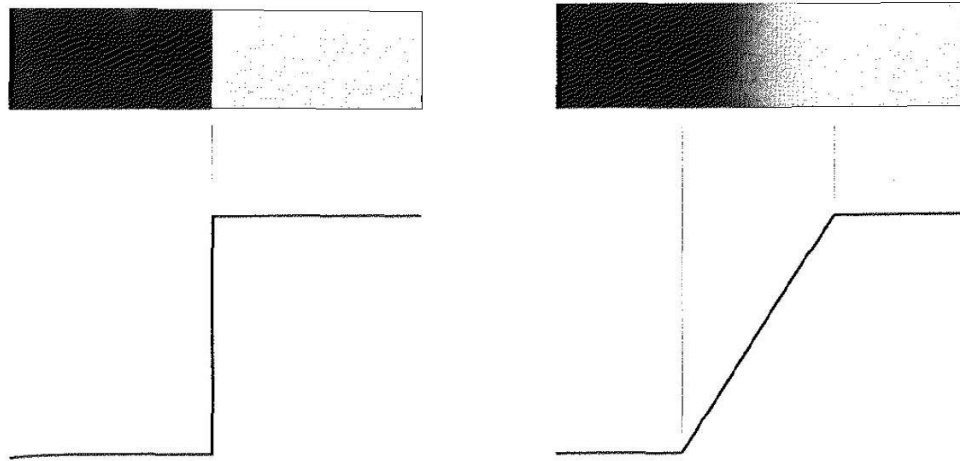


Figure 1a and b: description (a) and another description(b).

The graph shows the gray level profile of the edge with the grey levels shown as heights.

In these images there are no noise which produces a smooth transition. A gray level profile for a real image might look more like this:

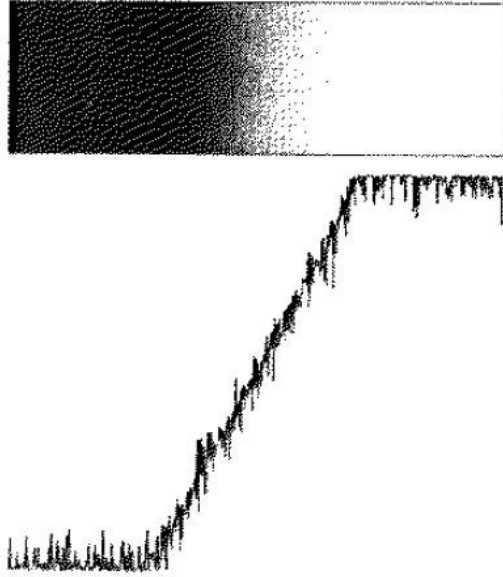


Figure 2: This illustrates that a transition is almost never perfect.

So we know that an edge is a large change in gray level value and is represented by a slope in the grey level profile. When we have a 2D image we will have these slopes in the x-direction and in the y-direction. In order to find these slopes (the edges) we use gradients. A gradient is the slope of the tangent to a point on the gray level profile and the magnitude of this gradient represents how steep the gradient (the tangent) is. How steep the gradient is represents the change in gray level values at any given point.

When we are using 2D images we will have a gradient in the x-direction and one in the y-direction. The gradient of a point is defined as the gradient vector $G = (g_x, g_y)$. This vector lies in the plane the x-gradient vector and the y-gradient vector spans (called the tangent plane).

The gradients represent the edges and by finding the magnitude of a gradient we can determine an edge.

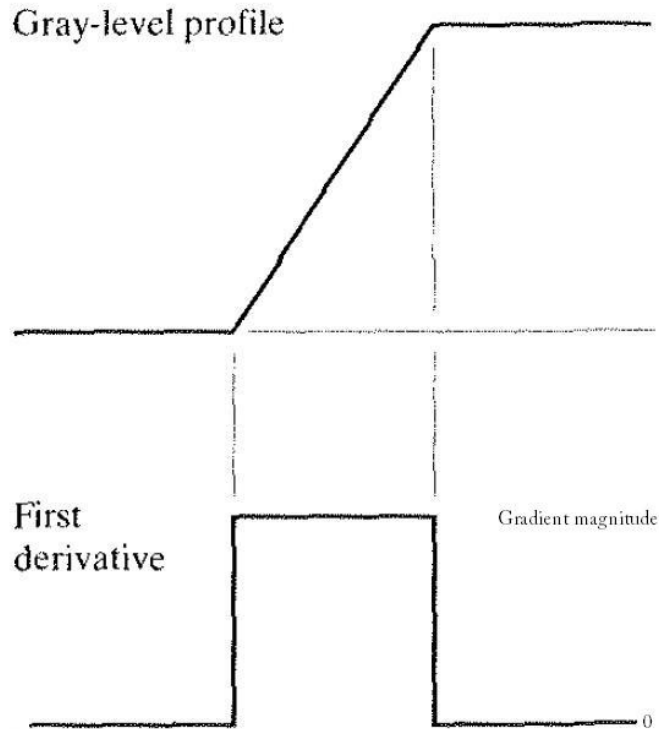


Figure 3: Showing the consequence of applying the first order derivative on a grayscale image.

In normal mathematics the gradients are found by the first order derivative. In an image however we can not use normal mathematics because the curves in the gray level profile are not continuous (they are pixel based: 1,2,3,4 ...). Therefore we need an approximation for the gradients. The magnitude of a gradient represents as mentioned before how steep the gradient is. This is how fast the values change. If we look at this on a pixel level and we want to find the gradient magnitude of a pixel in the x-direction, we take the pixel before and the pixel after the one were investigating and find the difference in these pixel values.

The same is done for the investigated pixel in the y-direction.

$$G = (g_x, g_y)$$

$$g_x(x,y) = f(x+1, y) - f(x-1, y)$$

$$g_y(x,y) = f(x, y+1) - f(x, y-1)$$

$$\text{Magnitude of } G = |g_x| + |g_y|$$

g_x and g_y can be found from using a kernel on an image. The pixel values are then added to form a picture with both horizontal and vertical edges.

1	0	-1
---	---	----

1
0
-1

Figure 4: Edges in y and x direction.

The kernels for basic edge detection are the Prewitt and the Sobel kernels discussed in the next section.

What is kernels?

In order to do edge detection, it is necessary to have a method of going through an image systematically. This process is called ‘Correlation’, and is a method where you analyse a picture by applying a set of rules on an image, which will have some consequence for the output of each pixel. A kernel is basically like a looking-glass, or a band-pass filter through which the only thing that is visible is determined by the rules presented by the kernel applied (see figure 5).

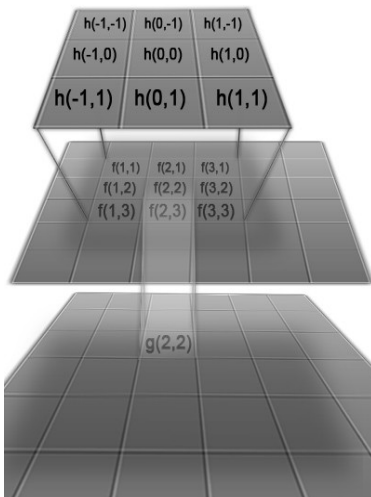


Figure 5: a 3x3 kernel applied to a 6x6 image.

When applied, the 3x3 kernel (h) is affecting the input image (f), and resulting in a specific output value, which will determine the output of that specific pixel. In kernels like the Sobels, the values produced can be positive, negative or neutral. If the result of the kernel applied in the 3x3 kernel in fig(5) results in a positive number, the output of g(2,2) would mean that the specific pixel g(2,2) would become lighter than earlier, resulting in a seemingly removal or highlighting of features. Whether the result is a highlighting or darkening of features is dependent on what kind of kernel you apply to it.

If the kernel is a 3x3 kernel, this also means that once applied to the edge of a picture, it will start comparing existing image data (f) to empty pixels outside of the image, as the kernel is too large for the area. As a solution to this, it is possible to apply special kernel sizes, such as a 3x2 kernel or 2x2 kernels.

Kernels used in edge detection

In this section the Sobel/Prewitt kernels will be explained.

These are typical examples of kernels that can be used to find edges in an image. They are very similar in structure and often looked at as if they were identical. Along with these, mean and median kernels will also be described.

The Sobel/Prewitt kernels are based on the grey-level gradients of pixel values in an image, originating from the first order derivatives of these particular gradients.

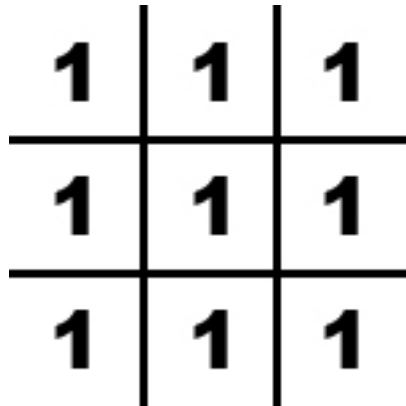
Before the actual edge detection kernel is applied, it is sometimes helpful to apply some type of averaging filter to the image. These are used to enhance the edge detection in an image. In this paper, mean and median filters will be described and explained.

Mean kernel

This type of kernel averages pixel values in an image. Effectively it blurs the image, erasing insignificant edges leaving only the most significant behind.

A mean kernel goes through the image pixels systematically starting at the upper most left corner. It sums these values and divides the result by the size of the kernel. In this

case the kernel is of the size 3x3 hence division by 9. The pixel at which the kernel acts will therefore become a mean of the surrounding neighbours.



1	1	1
1	1	1
1	1	1

Figure 6: the mean kernel

Median kernel

Instead of blurring an image, this filter will attempt to remove any ‘salt and pepper’ noise. The filter achieves its goal through execution of the following: inputting a series of pixel values, sorting them in ascending order and outputting the middle value. This way, extreme low- and high values are ignored, resulting in a more plausible result.

For the sake of the mini project, only a mean filter was used. We were solely interested in outputting the strongest edges.

Using either of the averaging filters, more suitable values will be available for the edge detecting kernels.

The various types of Sobel/Prewitt filters are shown below:

-1	-1	-1	-1	0	1
0	0	0	-1	0	1
1	1	1	-1	0	1

Figure 7a and b: the Prewitt kernel for vertical (a) and horizontal (b) edges.

-1	-2	-1	-1	0	1
0	0	0	-2	0	2
1	2	1	-1	0	1

Figure 8a and b: the Sobel kernels for vertical (a) and horizontal (b) edges.

All these kernels find horizontal and vertical edges in an image by getting a ‘stronger reaction’ in the respective direction. The values in the kernels are multiplied with the corresponding pixels values in the image, resulting in one final value for the pixel in question. Unlike the mean filter there is no division at the end of the calculation. The sole difference between Sobel and Prewitt lie in the weighting of the middle row/column, vertical and horizontal kernels respectively. Sobel uses a weighting of 2/-2 whereas Prewitt makes use of 1/-1. This results in smoothing since there is given more importance to the centre point.

For the mini project we were in need of finding diagonal edges in an image and looking at the horizontal/vertical edge kernels it was possible to ‘shift the attention’ to the diagonal directions.

We opted to use Sobel diagonal edge detectors since the smoothing was appreciated.

0	1	2	-2	-1	0
-1	0	1	-1	0	1
-2	-1	0	0	1	2

Figure 9a and b: the Sobel kernel for diagonal edges.

These kernels have the biggest impact if they go over diagonal edges and this results in diagonals appearing stronger after the kernel has gone through the image.

One of the mini project supervisors suggested that the group could take a look at the Laplacian filter. As a result of this, it was decided to implement the kernel in the application. The math behind the filter is more advanced than Sobel and Prewitt, and therefore it has been decided not give a detailed explanation of this.

C++ Implementation

Why openFrameworks?

When building the C++ program, there was many ways to solve trivial tasks, such as importing images, as is often the case with programming in general. The choice to use openFrameworks was heavily influenced by the fact that there was an expressed desire in the group to form a better understanding of openFrameworks and its use of the OpenCV library. The group has also chosen to work with openFrameworks in the 3rd semester project. Initially the implementation proved difficult, as openFramworks did not communicate well with the stand-alone OpenCV library. Through research and supervisor

guidance it was discovered that OpenFrameworks possessed all the needed functionality internally (although of course wrapping around OpenCV), and by sticking solely to this functionality progress was made quickly. Finally the choice to implement the code with openFramworks proved fruitful as it provided an easy way to create a gui for our application, that could then be used to manipulate the various settings in realtime.

Selection of C++ code explained

The following segment gives an overall description and explanation of the code written. The complete code is included at the end of the document.

In the setup() function we load in two source images; our original image to be used in the edge detection process (imageone), and lastly an illustration of the current working kernel (kernel).

```
void edgeDetect::setup(){  
    imageone.loadImage(picture+".jpg");  
    imageone.setImageType(OF_IMAGE_GRAYSCALE);  
  
    kernel.loadImage("sobelLeft.jpg");  
    kernel.setImageType(OF_IMAGE_GRAYSCALE);  
}
```

The three following lines declares three pointers pixel, pixel2, pixel3. Pixel links directly to the memory address of the loaded original image pixels, through the getPixels() function. Pixel2 and pixel3 are used to store the new pixels values after the kernel operations, by pushing the adjusted pixel values to imagetwo and imgethree. Note the setImageType() function that converts the images into a grayscale 8-bit image.

```
unsigned char * pixels = imageone.getPixels();  
unsigned char * pixels2 = imagetwo.getPixels();  
unsigned char * pixels3 = imgethree.getPixels();
```

Imageone (original) and imagetwo (edgedetected) are rendered on screen using openFrameworks function draw().

```
imageone.draw(50, 50);  
imagetwo.draw(w + 74, 50);
```

Three arrays are initiated to store the original (pixeldata[h][w]) and new pixel values; meandata[i][j] (the pixels subjected to a mean kernel) and edgedata[h][w] (one of the three edge detection kernels)

```
static int w = imageone.width;  
static int h = imageone.height;  
  
int pixeldata[h][w];  
float meandata[h][w];  
float edgedata[h][w];
```

The following for loop is an integral part of any imageprocessing. It starts at the upper-left corner and moves through the image pixel by pixel, storing these graylevel values in the pixeldata array. Note the use of the pixels pointer.

```
for (int i = 0; i < h; i++) {  
    for (int j = 0; j < w; j++) {  
        pixeldata[i][j] = pixels[i * w + j];
```

The next part of the code does the actual edge detection. It first checks for the status of a boolean value mean, that determines whether the mean filter should be applied to the initial values or not (this is controlled by the use of keyboard input). Moving on, an if structure controls which of the three earlier explained edge detection kernels should be applied to the image. Lastly a threshold is used to emphasize the edges found.

```
if (mean) {  
    // 5 x 5 mean filter  
    meandata[i][j] =  
        (pixeldata[i-3][j-2] + pixeldata[i-1][j-2] +  
         pixeldata[i][j-2] + pixeldata[i+1][j-2] +  
         pixeldata[i+2][j-2] + pixeldata[i-2][j-1] +
```

```

    pixeldata[i-1][j-1] + pixeldata[i][j-1] +
    pixeldata[i+1][j-1] + pixeldata[i+2][j-1] +
    pixeldata[i-2][j] + pixeldata[i-1][j] +
    pixeldata[i][j] + pixeldata[i+1][j] +
    pixeldata[i+2][j] + pixeldata[i-2][j+1] +
    pixeldata[i-1][j+1] + pixeldata[i][j+1] +
    pixeldata[i+1][j+1] + pixeldata[i+2][j+1] +
    pixeldata[i-2][j+2] + pixeldata[i-1][j+2] +
    pixeldata[i][j+2] + pixeldata[i+1][j+2] +
    pixeldata[i+2][j+2]) / 25;

    pixels2[i * w + j] = (unsigned char)meandata[i][j];
} else {

    meandata[i][j] = pixeldata[i][j];
}

// Sobel diagonal edge detection

if (kernelText == "Sobel Right") {

    edgedata[i][j] =
        ((meandata[i-1][j-1]) * -2 + (meandata[i][j-1]) * -1 +
        (meandata[i+1][j-1]) * 0 + (meandata[i-1][j]) * -1 +
        (meandata[i][j]) * 0 + (meandata[i+1][j]) * 1 +
        (meandata[i-1][j+1]) * 0 + (meandata[i][j+1]) * 1 +
        (meandata[i+1][j+1]) * 2);
}

if (kernelText == "Sobel Left") {

    edgedata[i][j] =
        ((meandata[i-1][j-1]) * 0 + (meandata[i][j-1]) * 1 +
        (meandata[i+1][j-1]) * 2 + (meandata[i-1][j]) * -1 +
        (meandata[i][j]) * 0 + (meandata[i+1][j]) * 1 +
        (meandata[i-1][j+1]) * -2 + (meandata[i][j+1]) * -1 +
        (meandata[i+1][j+1]) * 0);
}

// Laplacian

if (kernelText == "Laplacian") {

    edgedata[i][j] =
        ((meandata[i-1][j-1]) * 0 + (meandata[i][j-1]) * -1 +
        (meandata[i+1][j-1]) * 0 + (meandata[i-1][j]) * -1 +
        (meandata[i][j]) * 4 + (meandata[i+1][j]) * -1 +
        (meandata[i-1][j+1]) * 0 + (meandata[i][j+1]) * -1 +
        (meandata[i+1][j+1]) * 0);
}

```

```

// Thresholding
if (meandata[i][j] < threshold) {
    meandata[i][j] = 0;
}

if (meandata[i][j] > threshold) {
    meandata[i][j] = 255;
}

pixels3[i * w + j] = (unsigned char)edgedata[i][j];
    }
}

```

The complete code:

main.cpp

```

#include "ofMain.h"
#include "edgeDetect.h"

//=====
int main(){

    ofSetupOpenGL(1024, 768, OF_WINDOW);           //
    <----- setup the GL context

    // this kicks off the running of my app
    // can be OF_WINDOW or OF_FULLSCREEN
    // pass in width and height too:

    ofRunApp(new edgeDetect);
}

```

edgeDetect.h

```

#ifndef _EDGE_DETECT
#define _EDGE_DETECT

#define OF_ADDON_USING_OFXOPENCV

```

```

#include "ofMain.h"
#include "ofAddons.h"

class edgeDetect : public ofSimpleApp{

public:

    void setup();
    void update();
    void draw();

    void keyPressed(int key);
    void keyReleased(int key);
    void mouseMoved(int x, int y );
    void mouseDragged(int x, int y, int button);
    void mousePressed(int x, int y, int button);
    void mouseReleased();

    ofImage imageone;
    ofImage imagetwo;
    ofImage imagethree;

    ofImage kernel;

};

#endif

```

edgeDetect.cpp

```

#include "edgeDetect.h"
using namespace std;

int threshold = 60;
string picture = "diagonals";
string kernelText = "Sobel Left";
bool mean = false;

//-----
void edgeDetect::setup(){

    imageone.loadImage(picture+".jpg");
    imageone.setImageType(OF_IMAGE_GRAYSCALE);

    kernel.loadImage("sobelLeft.jpg");
    kernel.setImageType(OF_IMAGE_GRAYSCALE);

}

```

```

//-----
void edgeDetect::update(){
    ofBackground(0,0,0);
}

//-----
void edgeDetect::draw(){

    ofSetupScreen();

    ofSetColor(0xFFFFFFFF);

    imagetwo = imageone;
    imagethree = imagetwo;

    unsigned char * pixels = imageone.getPixels();
    unsigned char * pixels2 = imagetwo.getPixels();
    unsigned char * pixels3 = imagethree.getPixels();

    static int w = imageone.width;
    static int h = imageone.height;

    imageone.draw(50, 50);
    imagetwo.draw(w + 74, 50);

    int pixeldata[h][w];
    float meandata[h][w];
    float edgedata[h][w];

    for (int i = 0; i < h; i++) {
        for (int j = 0; j < w; j++) {
            pixeldata[i][j] = pixels[i * w + j];

            if (mean) {
                // 5 x 5 mean filter
                meandata[i][j] =
                    (pixeldata[i-3][j-2] + pixeldata[i-1][j-2]
+ pixeldata[i][j-2] + pixeldata[i+1][j-2] + pixeldata[i+2][j-2] +
                    pixeldata[i-2][j-1] + pixeldata[i-1][j-1] +
pixeldata[i][j-1] + pixeldata[i+1][j-1] + pixeldata[i+2][j-1] +
                    pixeldata[i-2][j] + pixeldata[i-1][j] +
pixeldata[i][j] + pixeldata[i+1][j] + pixeldata[i+2][j] +
                    pixeldata[i-2][j+1] + pixeldata[i-1][j+1] +
pixeldata[i][j+1] + pixeldata[i+1][j+1] + pixeldata[i+2][j+1] +
                    pixeldata[i-2][j+2] + pixeldata[i-1][j+2]
+ pixeldata[i][j+2] + pixeldata[i+1][j+2] + pixeldata[i+2][j+2])
/ 25;

                pixels2[i * w + j] = (unsigned
char)meandata[i][j];
            } else {

```



```

        meandata[i][j] = pixeldata[i][j];
    }

    // Sobel diagonal edge detection
    if (kernelText == "Sobel Right") {
        edgedata[i][j] =
            ((meandata[i-1][j-1]) * -2 + (meandata[i]
[j-1]) * -1 + (meandata[i+1][j-1]) * 0
            + (meandata[i-1][j]) * -1 + (meandata[i]
[j]) * 0 + (meandata[i+1][j]) * 1
            + (meandata[i-1][j+1]) * 0 + (meandata[i]
[j+1]) * 1 + (meandata[i+1][j+1]) * 2);
    }

    if (kernelText == "Sobel Left") {
        edgedata[i][j] =
            ((meandata[i-1][j-1]) * 0 + (meandata[i][j-
1]) * 1 + (meandata[i+1][j-1]) * 2
            + (meandata[i-1][j]) * -1 + (meandata[i]
[j]) * 0 + (meandata[i+1][j]) * 1
            + (meandata[i-1][j+1]) * -2 + (meandata[i]
[j+1]) * -1 + (meandata[i+1][j+1]) * 0);
    }

    // Laplacian
    if (kernelText == "Laplacian") {
        edgedata[i][j] =
            ((meandata[i-1][j-1]) * 0 + (meandata[i][j-
1]) * -1 + (meandata[i+1][j-1]) * 0
            + (meandata[i-1][j]) * -1 + (meandata[i]
[j]) * 4 + (meandata[i+1][j]) * -1
            + (meandata[i-1][j+1]) * 0 + (meandata[i]
[j+1]) * -1 + (meandata[i+1][j+1]) * 0);
    }

    // Thresholding
    if (meandata[i][j] < threshold) {
        meandata[i][j] = 0;
    }
    if (meandata[i][j] > threshold) {
        meandata[i][j] = 255;
    }

    pixels3[i * w + j] = (unsigned char)edgedata[i]
[j];

}

}

imagetwo.update();
imgethree.update();

```

```

        if (mean) {
            imagetwo.draw(50, 50);
        } else {
            imageone.draw(50, 50);
        }

        imagethree.draw(w + 74, 50);

        kernel.draw(w * 2 - 100 + 24, h + 75);

        // Text

        ofDrawBitmapString("Edge detector MEA331", 50, 40);

        if (mean) {
            ofDrawBitmapString("5x5 mean kernel applied to " +
picture + ".jpg", 50, h + 65);
        } else {
            ofDrawBitmapString("Original image (" + picture +
".jpg)", 50, h + 65);
        }

        ofDrawBitmapString("Edges found", w + 74, h + 65);
        ofDrawBitmapString("Threshold: " +
ofToString(threshold, 0), w * 2 - 100 + 24, h + 65);
        ofDrawBitmapString("Kernel: " + kernelText, w * 2 -
100 + 24, h + 245);

        ofDrawBitmapString("'left/right' image | 'up/down'
threshold | 'm' mean on/off | '1, 2, 3' kernel", 50, h + 245);

    }

    //-----
    void edgeDetect::keyPressed (int key){

        if (key == 'm') {
            if (mean) {
                mean = false;
            } else {
                mean = true;
            }
        }

        if(key == OF_KEY_UP) {
            threshold += 5;
            if(threshold > 255)threshold = 255;
        }else if(key == OF_KEY_DOWN) {
            threshold -= 5;
            if(threshold < 0)threshold = 0;
        }
    }

```

```

    }

    if(key == '1') {
        kernel.loadImage("sobelLeft.jpg");
        kernel.setImageType(OF_IMAGE_GRAYSCALE);
        kernelText = "Sobel Left";
    }
    if (key == '2') {
        kernel.loadImage("sobelRight.jpg");
        kernel.setImageType(OF_IMAGE_GRAYSCALE);
        kernelText = "Sobel Right";
    }
    if (key == '3') {
        kernel.loadImage("laplacian.jpg");
        kernel.setImageType(OF_IMAGE_GRAYSCALE);
        kernelText = "Laplacian";
    }

    if(key == OF_KEY_RIGHT) {

        if (picture == "diagonals") {
            picture = "miami";
        } else if (picture == "miami") {
            picture = "miami2";
        } else if (picture == "miami2") {
            picture = "diagonals";
        }

        imageone.loadImage(picture+".jpg");
        imageone.setImageType(OF_IMAGE_GRAYSCALE);
    }

    if(key == OF_KEY_LEFT) {

        if (picture == "diagonals") {
            picture = "miami2";
        } else if (picture == "miami2") {
            picture = "miami";
        } else if (picture == "miami") {
            picture = "diagonals";
        }

        imageone.loadImage(picture+".jpg");
        imageone.setImageType(OF_IMAGE_GRAYSCALE);
    }

}

//-----
void edgeDetect::keyReleased (int key){

}

```

```
//-----  
void edgeDetect::mouseMoved(int x, int y ){  
}  
  
//-----  
void edgeDetect::mouseDragged(int x, int y, int button){  
}  
  
//-----  
void edgeDetect::mousePressed(int x, int y, int button){  
}  
  
//-----  
void edgeDetect::mouseReleased(){  
}
```