Image Processing mini project

Diagonal Edge Detection in C++

Gustav Dahl, Study no. 20113263

Image Processing and Procedural Programming
Medialogy 3rd semester
Aalborg Universty October 2012

Task definition

The theme for the 3rd semester of Medialogy is Visual Computing. Here subjects about how humans and machines perceive images were taught. In the course **Image Processing** it was shown how it is possible to process and manipulate digital images in theory and concept, while the course **Procedural Programming** was about learning the `C++` programming language, as well as the OpenCV framework. To apply the knowledge about image processing in practice, each student were tasked with writing a small program that could process an image in a certain way. It was allowed to use the OpenCV framework to load in images, but the actual image processing algorithm should be written from scratch.

Each mini project was meant as an individual task, and everybody in the group received a different task. The following is the description of the task that I received.

> Topic #5: Diagonal Edge Detection
> Make a C/C++ program that can find diagonal edges in an image.
> Input: Greyscale image Output: Binary image where the diagonal edges are white (255) and the rest of the pixels black (0)

Table of content

Theory about Edge Detection 1

1.1 Edge definition

[Block, 2007] defines an edge as the apparent line around the borders of a two-dimensional object.

Another way to describe an edge is given by [Moeslund, 2012b]; he writes that an edge in an image is defined as a position where there is a significant change in gray-level values.

In other words, **an edge in an image is where the intensity changes dramatically.** A perfect edge would have to be a transition from e.g. black to white over just one pixel, but in the real world this rarely happen, unless it is a binary image where there are only black and white pixels.

1.2 Practical use of edge detection

Edges are typically used to define the boundary of an object. This reduces a lot of calculations needed to be done, either by the human brain or a computer, since it is only necessary to look at the outline and not the whole object. It allows for higher levels of abstraction.

The human brain makes use of this principle. To reduce the amount of information transported from the eyes to the brain, we perceive changes using the retinal ganglion cells. [Snowden et al., 2012] In machine vision this system is simulated using image edge detection. A typical example could be a robot that should be able to recognize and work with a specific object. The robot needs to know where the object is located, and this can be done using edge detection.

1.3 The concept of edge detection

When working with edges, one can think about it like gradients. The point of a gradient can be defined as the slope of the curve at the point. This corresponds to the slope of the tangent at the current point. [Moeslund, 2012b]

Having this in mind, edges will then be places where there are steep hills. Here, each point will have two gradients: one in the x-direction and another in the y-direction. These two gradients span a plane called the *tangent plane*. The gradient in is defined as a vector called $\vec{G}(g_x, g_y)$, where g_x is the gradient in the x-direction and g_y is the gradient in the y-direction. $\vec{G}(g_x, g_y)$ can be considered as the direction with the steepest slope. [Moeslund, 2012b]. Using the program ImageJ, this can be illustrated by creating a so-called *surface plot*, see figure 1.2.

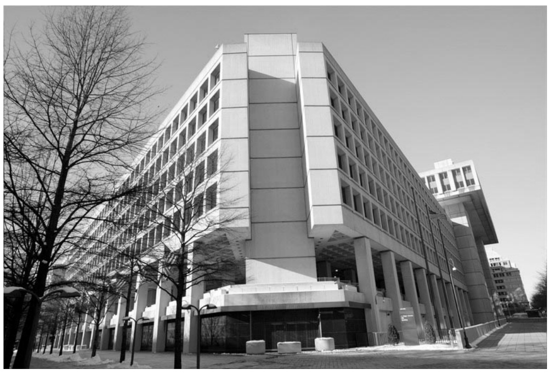


Figure 1.1. The original image seen in grayscale.

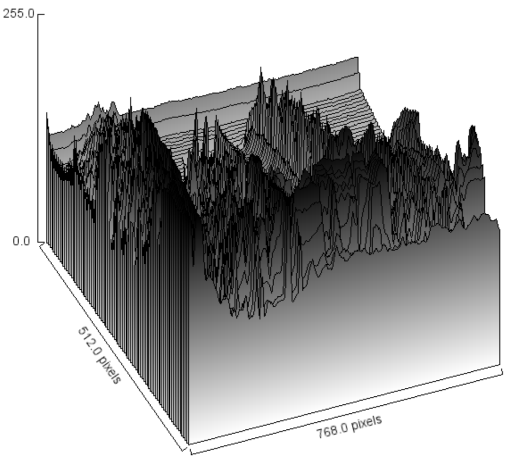


Figure 1.2. Surface plot of the same image, created with ImageJ.

The gradient has both a direction and a magnitude. The magnitude describes how steep the gradient is. It can be calculated by finding the length of the gradient vector, see equation 1.1. To achieve a faster implementation an approximation is often used, see 1.2.

$$\text{Magnitude} = \sqrt{g_x^2 + g_y^2} \tag{1.1}$$

$$\text{Approximated magnitude} = |g_x^2| + |g_y^2| \tag{1.2}$$

The following is mainly based on [Moeslund, 2012a].

Edge detectors consist of three steps:

- Noise reduction
- Edge enhancement
- Edge localization

The first step, **noise reduction**, can be done using a filter. Often an image contains an amount of noisy pixels with values that can change rapidly. These should not count as edges, and therefore a filter is used to reduce the noise, e.g. a mean or median filter is applied before the edge detection. However, there is a dilemma when choosing the size of the filter. A large filter will remove more noise from the image, but it will also remove some of the edges. A smaller filter, on the other hand, keeps more edges but also more noise.

The next step, **edge enhancement**, calculates the possible candidates for edges. This can be done in various ways. After this step it is time to decide what edges to keep using **edge localization**.

1.4 The Sobel filter

Multiple edge detectors exist. Among these are the Sobel and Canny filter. Sobel is the simplest of the two to implement and have therefore been chosen for this mini project. Its kernel weights row and column pixels in the center more than the rest. The Sobel filter is based on gray-level gradients, which is a measure of the steepness of what can be described as an image landscape (see figure 1.2). This is calculated for each individual pixel using the first-order derivative:

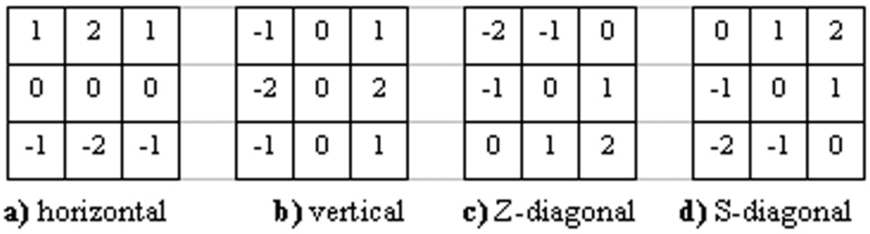
$$f'(x,y) = g(x,y)$$

Since the function of the image is not continuous but discrete, an approximation is used for the first-order derivative, as shown in equations 1.3 and 1.4.

$$g_x(x,y) \approx f(x+1,y) - f(x-1,y) \tag{1.3}$$

$$g_y(x,y) \approx f(x,y+1) - f(x,y-1) \tag{1.4}$$

Using correlation with the Sobel kernel can aid in finding either horizontal, vertical or diagonal edges in an image. This is done by applying the filter on the image via correlation. Depending on which kernel is used, lines will be more or less clear. The task for this mini project was to locate only diagonal edges. However, I chose to use all the kernels seen in figure 1.4 and combine them to get the most optimal image possible.



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

a) horizontal

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

b) vertical

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

c) Z-diagonal

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

d) S-diagonal

Figure 1.3. The different Sobel kernels focus on either horizontal, vertical or diagonal edges.

Implementation 2

As mentioned previously, OpenCV was used together with C++ to load in an image. OpenCV uses the matrix class denoted as *Mat* when working with images. The Mat type consists of various member fields and functions, such as its rows and columns.

2.1 From color to grayscale

Even though the task was to work with grayscale images, I chose to load in a color image and convert it to grayscale manually. This was done using equation 2.1 and can be seen on figure 2.1 and 2.2.

$$I = W_R \cdot R + W_G \cdot G + W_B \cdot B \tag{2.1}$$

where I is the intensity, and W_R, W_G and W_B are weight factors for the red, green and blue channel. It should be noted that $W_R + W_G + W_B = 1$, so the values stay within one byte in the range of [0, 255]. Using various weight values, one can achieve a grayscale image that fits the human eye. A common standard of weight values, used within TV and image/video coding are listed in 2.2. [Moeslund, 2012b]

$$W_R = 0.299, \qquad W_G = 0.587, \qquad W_B = 0.114 \tag{2.2}$$

In OpenCV, the color channels are stored in the order of blue, green and red (and not red, green and blue as in many other places). Using a function named *ConvertColorImageToBlackWhite*, the image is converted from a three-channel color image to a one-channel grayscale image. This is done by looping through each pixel and assign the grayscale value using equation 2.1:

```cpp
Mat ConvertColorImageToBlackWhite(Mat colorImage)
{
  // new 8-bit unsigned grayscale image with only 1 channel
  Mat grayScaleImage(colorImage.rows, colorImage.cols, CV_8UC1);

  // Formula for converting from color to grayscale (3.3, p. 30 in ↩
     Introduction to Video and Image Processing book)
  // I = weightR * R + weightG * G + weightB * B

  // Common weight values used in TV production to calculate to grayscale
  float RedWeight = 0.299;
  float GreenWeight = 0.587;
  float BlueWeight = 0.114;

  // Iterate through all the pixels and apply the formula for grayscale
  for (int y = 0; y < colorImage.rows; y++) // rows
  {
    for (int x = 0; x < colorImage.cols; x++)
    {
      // [0] = blue channel
      // [1] = green channel
      // [2] = red channel

      // Calculate grayscale value
      float grayValue = colorImage.at<cv::Vec3b>(y, x)[0] * BlueWeight
        + colorImage.at<cv::Vec3b>(y, x)[1] * GreenWeight
        + colorImage.at<cv::Vec3b>(y, x)[2] * RedWeight;

      // Apply the grayscale value (0-255)
      grayScaleImage.at<uchar>(y, x) = grayValue;

    }
  }
  return grayScaleImage;
}
```

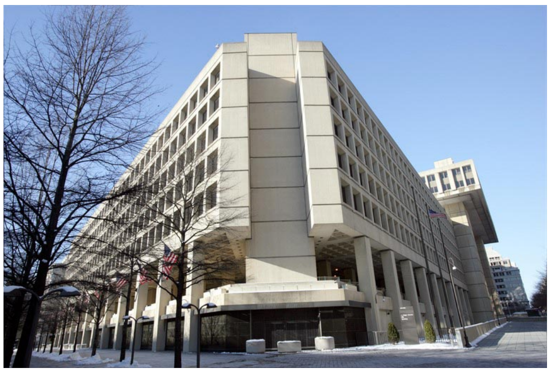


Figure 2.1. The original color image.

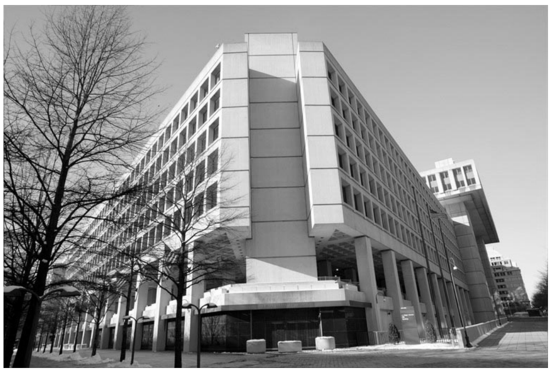


Figure 2.2. The new grayscale image.

2.2 Mean filter

To avoid unnecessary noise, a mean filter was applied. Using a kernel, it takes the average of the pixel values, which results in a blurred image (see figure 2.4). By doing this, all small edges are removed, leaving only the significant edges. In practice this is done by going through all pixels in the image and apply a kernel using correlation. It basically sums the values and divides the result by the size of the kernel.

A median filter could also have been used, but this is more appropriate for images with the so-called *salt and pepper noise*.

In this case a 5x5 kernel (radius: 2) has been used, and therefore the values are divided by 25 (see figure 2.3).

1	1	1	1	1
1	1	1	1	1
1	1	①	1	1
1	1	1	1	1
1	1	1	1	1

Figure 2.3. A 5x5 mean filter.

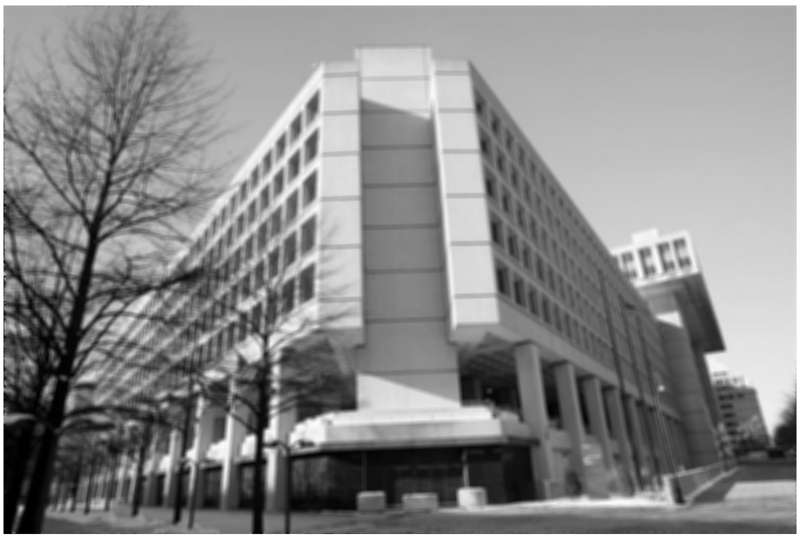


Figure 2.4. A mean filter results in a blurred image.

It should be noted that there is a problem associated with neighbourhood processing (which a mean filter is a part of) called *the border problem.* Since the kernel cannot be applied outside of the image, the pixels from the outer borders will not be touched by the filter. The bigger the radius of the filter is, the bigger the untouched border will be. For simplistic sake this problem has not been addressed in the following code:

```
Mat MeanFilter(Mat input)
{
  // 5x5 kernel

  // Make a temporary clone of the input image
  Mat mean = input.clone();

  // Loop through all pixels
  for (int y = 0; y < input.rows-2; y++)
  {
    for (int x = 0; x < input.cols-2; x++)
    {
      if (x - 2 < 0 || y - 2 < 0) // don't go out of bounds
        continue;

      // Apply the kernel
      mean.at<uchar>(y, x) = (
        input.at<uchar>(y-2, x-2) + input.at<uchar>(y-2, x-1)
        + input.at<uchar>(y-2, x) + input.at<uchar>(y-2, x+1)
        + input.at<uchar>(y-2, x+2) + input.at<uchar>(y-1, x-2)
        + input.at<uchar>(y-1, x-1) + input.at<uchar>(y-1, x)
        + input.at<uchar>(y-1, x+1) + input.at<uchar>(y-1, x+2)
        + input.at<uchar>(y, x-2) + input.at<uchar>(y, x-1)
        + input.at<uchar>(y, x) + input.at<uchar>(y, x+1)
        + input.at<uchar>(y, x+2) + input.at<uchar>(y+1, x-2)
        + input.at<uchar>(y+1, x-1) + input.at<uchar>(y+1, x)
        + input.at<uchar>(y+1, x+1) + input.at<uchar>(y+1, x+2)
        + input.at<uchar>(y+2, x-2) + input.at<uchar>(y+2, x-1)
        + input.at<uchar>(y+2, x) + input.at<uchar>(y+2, x+1)
        + input.at<uchar>(y+2, x+2)
        ) / 25;
    }
  }

  return mean;
}
```

2.3 Thresholding

Before the actual edge detection is used, a threshold is applied. This result in a binary image where all pixels are either black (0) or white (255); there is nothing in between. The threshold value can vary from image to image. In this example ImageJ was chosen to automatically find the best threshold value for the image of the building, which turned out to be 133. The code for doing the threshold is quite simple:

```
// optimal value was found using ImageJ
const int THRESHOLD_GRAYSCALE = 133;
Mat ThresholdBlackWhiteImage(Mat blackWhiteImage, int threshold)
{
  Mat image = blackWhiteImage.clone();

  // Loop through all pixels and set them to either 255 (white) or 0 ↩
      (black) using the threhold value
  for (int y = 0; y < image.rows; y++)
  {
    for (int x = 0; x < image.cols; x++)
    {
      if (image.at<uchar>(y, x) >= threshold)
```

```
            image.at<uchar>(y, x) = 255;
        else
            image.at<uchar>(y, x) = 0;
        }
    }

    return image;
}
```

The result can be seen in figure 2.10

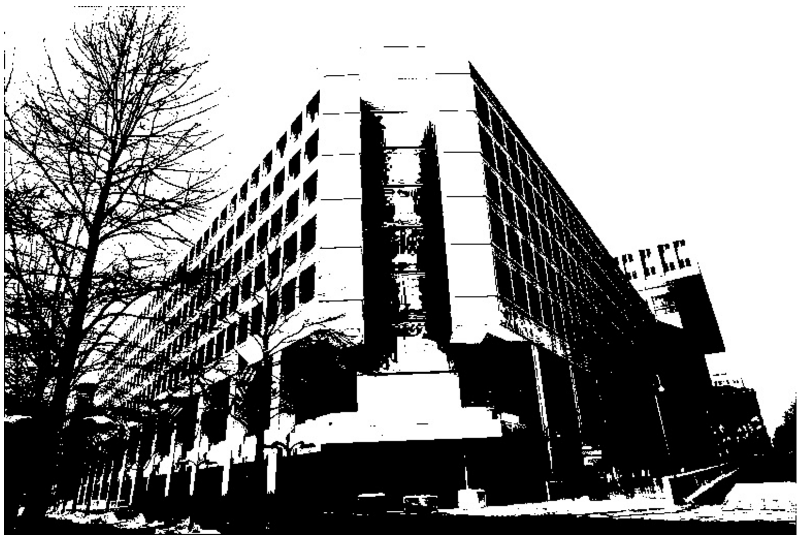


Figure 2.5. Using a threshold the image becomes binary.

2.4 The edge detection

As mentioned in section 1.3, edge detection consists of three basic steps, which I have put into a function called *SobelEdgeDetecting*. The first step is doing noise reduction - in this case, applying the before-mentioned mean filter.

When this is done, the Sobel kernel is applied on the who image. This is implemented with a nested for loop that goes through all the pixels, one by one, and applies the specified kernel. As mentioned earlier, I have chosen to implement not only a diagonal Sobel edge kernel, but also horizontal and vertical. This can be specified using an enumerator variable called SobelDirection.

Finally, it is time to decide what edges to keep and what to throw away. This is done using a simple threshold check, like in the threshold function defined earlier.

All this leave of with the following code. Note that I have chosen to omit all the direction checks to make it easier to read. In the appendix the code in all its length can be seen.

```
enum SobelDirection
{
  Diagonal_Right,
  Diagonal_Left,
  Vertical,
  Horizontal
};

// found by experimenting
const int THRESHOLD_SOBEL = 100;

Mat SobelEdgeDetecting(Mat input, enum SobelDirection direction, bool↩
    useMeanFilterBeforeDoingEdgeDetecting, int threshold)
{
  Mat edge = input.clone();

  // STEP 1: NOISE REDUCTION
  if (useMeanFilterBeforeDoingEdgeDetecting)
    edge = MeanFilter(edge);

  // Apply diagonal edge detecting RIGHT
  if (direction == Diagonal_Right)
  {
    for (int y = 0; y < input.rows-1; y++)
    {
      for (int x = 0; x < input.cols-1; x++)
      {
        if (x-1 < 0 || y-1 < 0) // don't go out of bounds
          continue;

        // STEP TWO: EDGE ENHANCEMENT
        // temp value is used to not get overflow (value cannot be ↩
           less than 0 or greater than 255)
        int temp = (
          (input.at<uchar>(y-1, x-1)) * -2
          + (input.at<uchar>(y, x-1)) * -1
          + (input.at<uchar>(y+1, x-1)) * 0
          + (input.at<uchar>(y-1, x)) * -1
          + (input.at<uchar>(y, x)) * 0
          + (input.at<uchar>(y+1, x+0)) * 1
          + (input.at<uchar>(y-1, x+1)) * 0
          + (input.at<uchar>(y, x+1)) * 1
          + (input.at<uchar>(y+1, x+1)) * 2
          );

        // Absolute value
        if (temp < 0)
          temp *= -1;

        // STEP THREE: EDGE LOCALIZATION
        // Map values from 0 to 255
        if (temp <= threshold)
          temp = 0;
        else
          temp = 255;
```

```


            edge.at<uchar>(y, x) = temp;
        }
      }
    }
      return edge;
}
```

Depending on which Sobel direction is chosen (right diagonal, left diagonal, vertical or horizontal), the output image will look as follows:

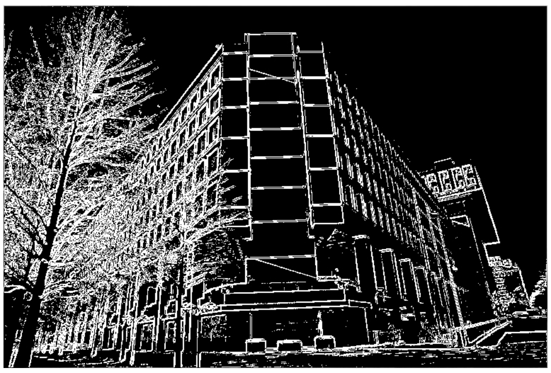


Figure 2.6. Right diagonal.

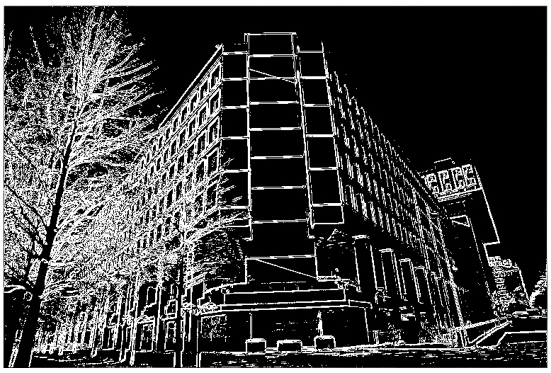


Figure 2.7. Left diagonal.

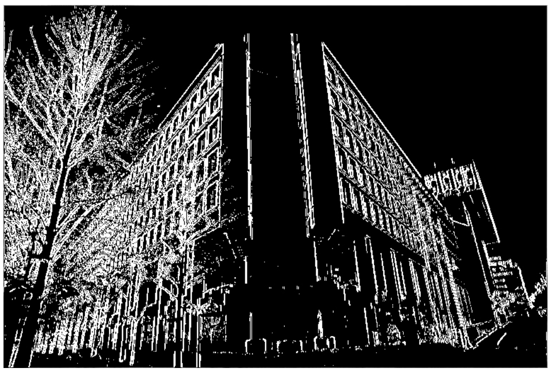


Figure 2.8. Vertical.

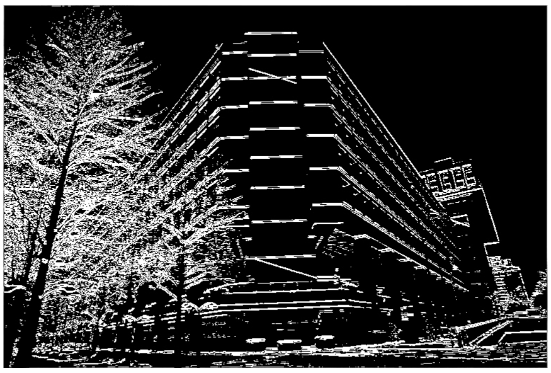


Figure 2.9. Horizontal.

Depending on what is needed, one of the images can be chosen. Even though the diagonal images are quite good, it is possible to combine all the images, which I have done with the following code:

```
// both Mats should be same size!
Mat AddTwoMatsTogether(Mat matA, Mat matB)
{
  Mat output = matA.clone();

  for (int y = 0; y < matA.rows; y++)
  {
    for (int x = 0; x < matA.cols; x++)
    {
      output.at<uchar>(y, x) = matA.at<uchar>(y, x) + matB.at<uchar>(↩
          y, x);
    }

  }

  output = ThresholdBlackWhiteImage(output, THRESHOLD_GRAYSCALE);
  return output;
}
```

Combining all four images into one gives this result. Compared with the original image (figure 2.1), the edges are quite clear.

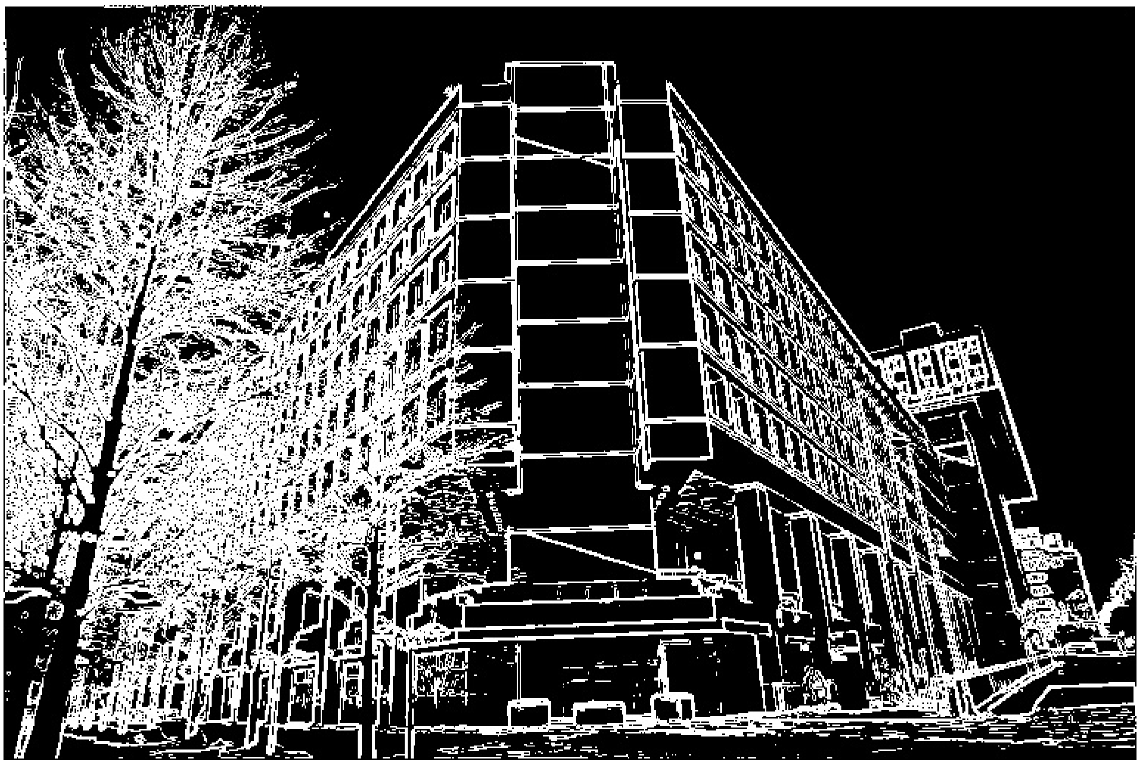


Figure 2.10. The final image where all Sobel directions are combined.

2.5 Erosion outline

Another way to find edges is using the concept of morphology, more specifically the erosion operation. Erosion basically makes an image smaller. Subtracting this from the original image results in an outline. Even though this is not as precise as the Sobel edge detection, it is interesting to compare the two.

The idea behind morphology is simple and works like neighbourhood processing (such as the mean and Sobel filter). Here the kernel is denoted as the *structuring element* and contains 0's and 1's. When doing erosion, you look at each of the 1's in the structuring element and see if the corresponding pixel is also a 1 (or 255, meaning it is white). If this is the case for all of the 1's in the structuring element, the structuring element is "fitting" the image. If this is not the case, the pixels are set to 0, which has the result of shrinking the overall image. [Moeslund, 2012b]

The erosion function is programmed in the following manner:

```cpp
// Uses a grayscale image
Mat Erosion(Mat input, int radius)
{
  Mat output = input.clone();

  for(int x = radius; x < input.cols-radius; x++)
  {
    for(int y = radius; y < input.rows-radius; y++)
    {
      bool pixelIsaccepted = true;
      for(int filterX = x - radius; pixelIsaccepted && filterX <= x +
          radius; filterX++)
      {
        for(int filterY = y - radius; pixelIsaccepted && filterY <= y
            + radius; filterY++)
        {
          if (input.at<uchar>(filterY,filterX) == 0)
          {
            pixelIsaccepted = false;
          }
        }
      }
      if (pixelIsaccepted == true)
        output.at<uchar>(y,x) = 255;
      else
        output.at<uchar>(y,x) = 0;
    }
  }

  return output;
}
```

The result of this image can be seen in figure 2.11. By subtracting the original image with the eroded image, an outline can be seen (figure 2.12). This does not give the diagonal edges, but it still provides a decent outline of the building, as well as the tree.

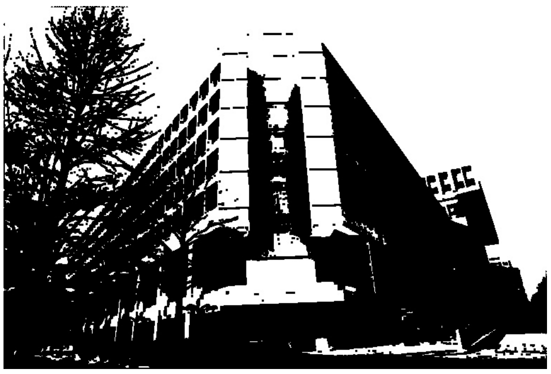


Figure 2.11. The image has become smaller due to the erosion operation.

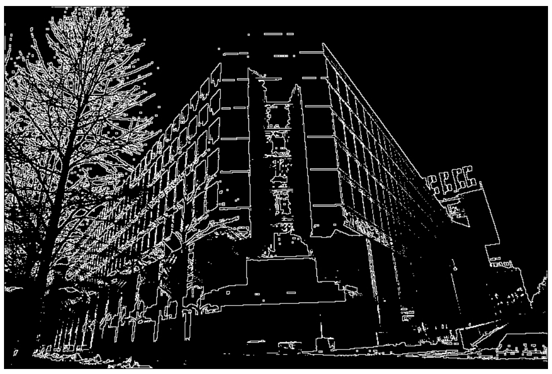


Figure 2.12. The outline based on the erosion image.

Bibliography

Image of building used throughout the report. URL `http://www.mccullagh.org/db9/1ds2-4/fbi-headquarters-building.jpg`.

Block, **2007**. Bruce Block. *The Visual Story, Second Edition: Creating the Visual Structure of Film, TV and Digital Media.* ISBN: 978-0240807799. Focal Press, second edition edition, 2007.

Moeslund, **October 2012a**. Thomas B. Moeslund. *Lecture AAU - Edge detection*, 2012.

Moeslund, **2012b**. Thomas B. Moeslund. *Introduction to Video and Image Processing - Building real systems and applications.* ISBN: 978-1447125020, Handbook. Springer, 2012.

Snowden et al., **2012**. Robert Snowden, Peter Thompson og Tom Troscianko. *Basic Vision: An Introduction to Visual Perception.* 978-0199572021. Oxford University Press, second edition edition, 2012.