Image Processing mini project

Diagonal Edge Detection in C++

Gustav Dahl, 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. Everything should be implemented from scratch; OpenCV should only be used to load in an image and nothing more.

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 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

1 Theory about Edge Detection

1.1 Edge definition

An edge in an image is basically a place in an image where there is a contrast between two points.

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

Another definition is given by [Moeslund, 2012b] who 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 The usefulness of edges

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. This system is used in the way a human perceives the world, using ganglion cell signal changes [Snowden et al., 2012] [1]. In machine vision this system is applied, e.g. if a robot needs to recognize and work with a specific object.

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 this 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 the end 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. Then $\vec{G}(g_x, g_y)$ can be considered as the direction with the steepst

[1] FiXme Note: bedre skrive + Perception bog

slope. [Moeslund, 2012b]. Using the program ImageJ, this can be illustrated by creating a so-called *surface plot*, see figure 1.1 and 1.2.

Figure 1.1. The original image seen in grayscale.

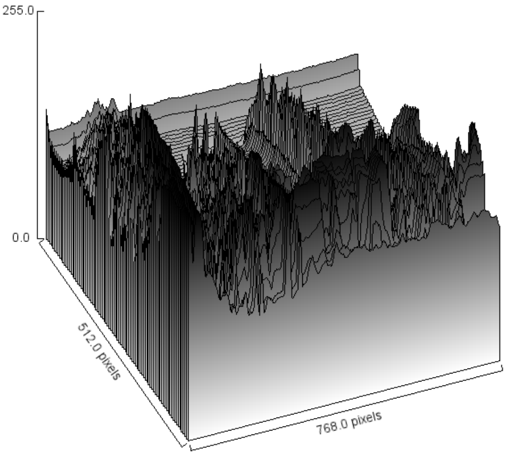

Figure 1.2. Surface plot point created using ImageJ.

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. After this step it is time to decide what edges to keep using **edge localization**.

1.4 The Sobel filter

Various 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. 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, an approximation is used for the first-order derivative, as shown in 1.1 and 1.2.

$$g_x(x, y) \approx f(x+1, y) - f(x-1, y) \tag{1.1}$$

$$g_y(x, y) \approx f(x, y+1) - f(x, y-1) \tag{1.2}$$

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. The task for this mini project was to locate 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.

$$\underset{\textbf{a)}\text{ horizontal}}{\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}} \quad \underset{\textbf{b)}\text{ vertical}}{\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}} \quad \underset{\textbf{c)}\text{ Z-diagonal}}{\begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix}} \quad \underset{\textbf{d)}\text{ S-diagonal}}{\begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}}$$

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

2 The code

```
#include <opencv2/highgui/highgui.hpp>
#include <iostream>

using namespace cv;
using namespace std;

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

const int THRESHOLD_GRAYSCALE = 133; // optimal value was found using ImageJ
const int THRESHOLD_SOBEL = 100; // found by experimenting

Mat ConvertColorImageToBlackWhite(Mat colorImage);
Mat MeanFilter(Mat input);
Mat ThresholdBlackWhiteImage(Mat blackWhiteImage, int threshold);
Mat SobelEdgeDetecting(Mat input, enum SobelDirection direction, bool useMeanFilterBeforeDoingEdgeDetecting, int threshold);
Mat AddTwoMatsTogether(Mat matA, Mat matB);
Mat Erosion(Mat input, int radius);



int main()
{
  // Program description
  cout << "Edge detection using the Sobel kernel (and OpenCV to load images)" << endl;
  cout << "By Gustav Dahl - Medialogy 3rd semester 2012 - Aalborg University\n\n";

  // Load the original color image
    Mat colorImage = imread("0_building.jpg");

  if (colorImage.empty())
    {
```

```
            cout << "Cannot load image!" << endl;
            return -1;
        }

    // "Loading" screen
    cout << "Processing image. Please wait..." << endl;


    // ---------- APPLY IMAGE PROCESSING ---------
    // Convert color image to grayscale
    Mat gray = ConvertColorImageToBlackWhite(colorImage);

    // Mean filter applied(black and white only)
    Mat mean = MeanFilter(gray);

    // Grayscale threshold
    Mat threshold = ThresholdBlackWhiteImage(gray, THRESHOLD_GRAYSCALE);

    // Erosion
    Mat erosion = Erosion(threshold, 1);

    // Finding outline using the eroded image, by subtracting the original grayscale from the eroded image
    Mat erosionOutline = threshold - erosion;

    // Edge detecting using the Sobel kernel
    Mat edge_diagonal_right = SobelEdgeDetecting(gray, Diagonal_Right, true, THRESHOLD_SOBEL);
    Mat edge_diagonal_left = SobelEdgeDetecting(gray, Diagonal_Left, true, THRESHOLD_SOBEL);
    Mat edge_vertical = SobelEdgeDetecting(gray, Vertical, true, THRESHOLD_SOBEL);
    Mat edge_horizontal = SobelEdgeDetecting(gray, Horizontal, true, THRESHOLD_SOBEL);

    // Combine the different kernels
    Mat vertical_plus_horizontal = AddTwoMatsTogether(edge_vertical, edge_horizontal);
    Mat diagonal_right_plus_left = AddTwoMatsTogether(edge_diagonal_right, edge_diagonal_left);
    Mat diagonal_plus_vertical_horizontal = AddTwoMatsTogether(vertical_plus_horizontal, edge_diagonal_right);

    // -------------------

    // Save the images
    imwrite("1_grayscale.jpg", gray);
    imwrite("2_meanFilter.jpg", mean);
    imwrite("3_threshold.jpg", threshold);
    imwrite("4_erosion.jpg", erosion);
    imwrite("5_erosionOutline.jpg", erosionOutline);
```

```
  imwrite("6_edge_diagonal_right.jpg", edge_diagonal_right);
  imwrite("7_edge_diagonal_left.jpg", edge_diagonal_left);
  imwrite("8_edge_vertical.jpg", edge_vertical);
  imwrite("9_edge_horizontal.jpg", edge_horizontal);
  imwrite("10_vertical_plus_horizontal.jpg", vertical_plus_horizontal↩
      );
  imwrite("11_diagonal_right_plus_left.jpg", diagonal_right_plus_left↩
      );
  imwrite("12_diagonal_plus_vertical_horizontal.jpg", ↩
      diagonal_plus_vertical_horizontal);

  // Show the images
  imshow("original color image", colorImage);
    imshow("grayscale", gray);
  imshow("meanFilter", mean);
  imshow("threshold", threshold);
  imshow("erosion", erosion);
  imshow("erosionOutline", erosionOutline);
  imshow("edge_diagonal_right", edge_diagonal_right);
  imshow("edge_diagonal_left", edge_diagonal_left);
  imshow("edge_vertical", edge_vertical);
  imshow("edge_horizontal", edge_horizontal);
  imshow("vertical_plus_horizontal", vertical_plus_horizontal);
  imshow("diagonal_right_plus_left", diagonal_right_plus_left);
  imshow("diagonal_plus_vertical_horizontal", ↩
      diagonal_plus_vertical_horizontal);
    waitKey(0);
}

Mat ConvertColorImageToBlackWhite(Mat colorImage)
{
  Mat grayScaleImage(colorImage.rows, colorImage.cols, CV_8UC1); // ↩
      new image with only 1 channel

  // 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++)
    {
      // 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;
}

Mat MeanFilter(Mat input)
{
  // 3x3 kernel size

  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;

      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;
}

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;
}

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

  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;


        // 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;

```

```
      // Map values from 0 to 255
      if (temp <= threshold)
        temp = 0;
      else
        temp = 255;


      edge.at<uchar>(y, x) = temp;
    }
  }
}
else if (direction == Diagonal_Left)
{ // Apply diagonal edge detecting LEFT
  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;

      // 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;

      // Map values from 0 to 255
      if (temp <= threshold)
        temp = 0;
      else
        temp = 255;


      edge.at<uchar>(y, x) = temp;
    }
  }
}
else if (direction == Vertical)
{
  // Apply diagonal edge detecting tVERTICAL
```

```
    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;

        // 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)) * -1
          + (input.at<uchar>(y, x-1)) * -2
          + (input.at<uchar>(y+1, x-1)) * -1
          + (input.at<uchar>(y-1, x)) * -0
          + (input.at<uchar>(y, x)) * 0
          + (input.at<uchar>(y+1, x+0)) * 0
          + (input.at<uchar>(y-1, x+1)) * 1
          + (input.at<uchar>(y, x+1)) * 2
          + (input.at<uchar>(y+1, x+1)) * 1
          );

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

        // Map values from 0 to 255
        if (temp <= threshold)
          temp = 0;
        else
          temp = 255;


        edge.at<uchar>(y, x) = temp;
      }
    }
  }
  else if (direction == Horizontal)
  {
    // Apply diagonal edge detecting HORIZONTAL
    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;

        // 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)) * -1
          + (input.at<uchar>(y, x-1)) * 0
          + (input.at<uchar>(y+1, x-1)) * 1
```

```
          + (input.at<uchar>(y-1, x)) * -2
          + (input.at<uchar>(y, x)) * 0
          + (input.at<uchar>(y+1, x+0)) * 2
          + (input.at<uchar>(y-1, x+1)) * -1
          + (input.at<uchar>(y, x+1)) * 0
          + (input.at<uchar>(y+1, x+1)) * 1
          );

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

        // Map values from 0 to 255
        if (temp <= threshold)
          temp = 0;
        else
          temp = 255;


        edge.at<uchar>(y, x) = temp;
      }
    }
  }
  else
  {
    // Error text
    putText(edge, "ERROR - Sobel type not defined!", Point(10, 50), FONT_HERSHEY_PLAIN, 2, Scalar(0, 0, 255), 4, 8, false);
  }

  // Threshold
  edge = ThresholdBlackWhiteImage(edge, 30);
  return edge;
}

Mat AddTwoMatsTogether(Mat matA, Mat matB) // should be same size!
{
  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;
}
```

```

// 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;
}
```

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.