**Assignment 2**

**Makar Riabcev**

**Michał Czorniej**

**Part A**

**Code:**

#include <iostream>

#include <vector>

#include <opencv2/highgui.hpp>

#include <opencv2/imgproc.hpp>

#include <opencv2/opencv.hpp>

using namespace cv;

using namespace std;

Mat MyCanny(Mat src, int upperThreshold, int lowerThreshold, cv::Size kernel, double size);

Mat src, res3k, res5k, res7k, workImg, returnImg, imageChanged;

int scale = 1;

int delta = 0;

int ddepth = CV\_16S;

int main()

{

//read the image in grayscale

src = imread("C://Edge Detection.png", 0);

//clone the original image

res3k = src.clone();

res5k = src.clone();

res7k = src.clone();

//apply canny with different kernels

res3k = MyCanny(res3k, 10, 100,cv::Size(3,3),1.4);

res5k = MyCanny(res5k, 10, 100, cv::Size(5, 5), 1.4);

res7k = MyCanny(res7k, 10, 100, cv::Size(7, 7), 1.4);

//show the result

namedWindow("original");

imshow("original", src);

namedWindow("result of kernel 3", WINDOW\_NORMAL);

imshow("result of kernel 3", res3k);

namedWindow("result of kernel 5", WINDOW\_NORMAL);

imshow("result of kernel 5", res5k);

namedWindow("result of kernel 7", WINDOW\_NORMAL);

imshow("result of kernel 7", res7k);

waitKey(0);

return 0;

}

Mat MyCanny(Mat src, int upperThreshold, int lowerThreshold, cv::Size kernel, double size)

{

// Step 1: Noise reduction

GaussianBlur(src, workImg, kernel, 0, 0, BORDER\_DEFAULT);

// Step 2: Calculating gradient magnitudes and directions

Mat magX = Mat(src.rows, src.cols, CV\_32F);

Mat magY = Mat(src.rows, src.cols, CV\_32F);

Sobel(workImg, magX, CV\_32F, 1, 0, size);

Sobel(workImg, magY, CV\_32F, 0, 1, size);

Mat direction = Mat(workImg.rows, workImg.cols, CV\_32F);

divide(magY, magX, direction);

Mat sum = Mat(workImg.rows, workImg.cols, CV\_64F);

Mat prodX = Mat(workImg.rows, workImg.cols, CV\_64F);

Mat prodY = Mat(workImg.rows, workImg.cols, CV\_64F);

multiply(magX, magX, prodX);

multiply(magY, magY, prodY);

sum = prodX + prodY;

sqrt(sum, sum);

/// Step 3

Mat returnImg = Mat(src.rows, src.cols, CV\_8U);

returnImg.setTo(Scalar(0));

MatIterator\_<float>itMag = sum.begin<float>();

MatIterator\_<float>itDirection = direction.begin<float>();

MatIterator\_<unsigned char>itRet = returnImg.begin<unsigned char>();

MatIterator\_<float>itend = sum.end<float>();

for (; itMag != itend; ++itDirection, ++itRet, ++itMag)

{

const Point pos = itRet.pos();

float currentDirection = atan(\*itDirection) \* 180 / 3.142;

while (currentDirection < 0) currentDirection += 180;

\*itDirection = currentDirection;

if (\*itMag < upperThreshold) continue;

bool flag = true;

if (currentDirection > 112.5 && currentDirection <= 157.5)

{

if (pos.y > 0 && pos.x < workImg.cols - 1 && \*itMag <= sum.at<float>(pos.y - 1, pos.x + 1)) flag =

false;

if (pos.y < workImg.rows - 1 && pos.x>0 && \*itMag <= sum.at<float>(pos.y + 1, pos.x - 1)) flag =

false;

}

else if (currentDirection > 67.5 && currentDirection <= 112.5)

{

if (pos.y > 0 && \*itMag <= sum.at<float>(pos.y - 1, pos.x)) flag = false;

if (pos.y < workImg.rows - 1 && \*itMag <= sum.at<float>(pos.y + 1, pos.x)) flag = false;

}

else if (currentDirection > 22.5 && currentDirection <= 67.5)

{

if (pos.y > 0 && pos.x > 0 && \*itMag <= sum.at<float>(pos.y - 1, pos.x - 1)) flag = false;

if (pos.y < workImg.rows - 1 && pos.x < workImg.cols - 1 && \*itMag <= sum.at<float>(pos.y + 1, pos

.x + 1)) flag = false;

}

else

{

if (pos.x > 0 && \*itMag <= sum.at<float>(pos.y, pos.x - 1)) flag = false;

if (pos.x < workImg.cols - 1 && \*itMag <= sum.at<float>(pos.y, pos.x + 1)) flag = false;

}

if (flag)

{

\*itRet = 255;

}

}

// Step 4

bool imageChanged = true;

int i = 0;

while (imageChanged)

{

imageChanged = false;

printf("Iteration %d\

", i);

i++;

itMag = sum.begin<float>();

itDirection = direction.begin<float>();

itRet = returnImg.begin<unsigned char>();

itend = sum.end<float>();

for (; itMag != itend; ++itMag, ++itDirection, ++itRet)

{

//Point pos = itRet.pos();

Point pos = itRet.pos();

if (pos.x<2 || pos.x>src.cols - 2 || pos.y<2 || pos.y>src.rows - 2)

continue;

float currentDirection = \*itDirection;

// Do we have a pixel we already know as an edge?

if (\*itRet == 255)

{

\*itRet = (unsigned char)64;

if (currentDirection > 112.5 && currentDirection <= 157.5)

{

if (pos.y > 0 && pos.x > 0)

{

if (lowerThreshold <= sum.at<float>(pos.y - 1, pos.x - 1) &&

returnImg.at<unsigned char>(pos.y - 1, pos.x - 1) != 64 &&

direction.at<float>(pos.y - 1, pos.x - 1) > 112.5 &&

direction.at<float>(pos.y - 1, pos.x - 1) <= 157.5 &&

sum.at<float>(pos.y - 1, pos.x - 1) > sum.at<float>(pos.y - 2, pos.x) &&

sum.at<float>(pos.y - 1, pos.x - 1) > sum.at<float>(pos.y, pos.x - 2))

{

returnImg.ptr<unsigned char>(pos.y - 1, pos.x - 1)[0] = 255;

imageChanged = true;

}

}

if (pos.y < workImg.rows - 1 && pos.x < workImg.cols - 1)

{

if (lowerThreshold <= sum.at<float>(Point(pos.x + 1, pos.y + 1)) &&

returnImg.at<unsigned char>(pos.y + 1, pos.x + 1) != 64 &&

direction.at<float>(pos.y + 1, pos.x + 1) > 112.5 &&

direction.at<float>(pos.y + 1, pos.x + 1) <= 157.5 &&

sum.at<float>(pos.y - 1, pos.x - 1) > sum.at<float>(pos.y + 2, pos.x) &&

sum.at<float>(pos.y - 1, pos.x - 1) > sum.at<float>(pos.y, pos.x + 2))

{

returnImg.ptr<unsigned char>(pos.y + 1, pos.x + 1)[0] = 255;

imageChanged = true;

}

}

}

else if (currentDirection > 67.5 && currentDirection <= 112.5)

{

if (pos.x > 0)

{

if (lowerThreshold <= sum.at<float>(Point(pos.x - 1, pos.y)) &&

returnImg.at<unsigned char>(pos.y, pos.x - 1) != 64 &&

direction.at<float>(pos.y, pos.x - 1) > 67.5 &&

direction.at<float>(pos.y, pos.x - 1) <= 112.5 &&

sum.at<float>(pos.y, pos.x - 1) > sum.at<float>(pos.y - 1, pos.x - 1) &&

sum.at<float>(pos.y, pos.x - 1) > sum.at<float>(pos.y + 1, pos.x - 1))

{

returnImg.ptr<unsigned char>(pos.y, pos.x - 1)[0] = 255;

imageChanged = true;

}

}

if (pos.x < workImg.cols - 1)

{

if (lowerThreshold <= sum.at<float>(Point(pos.x + 1, pos.y)) &&

returnImg.at<unsigned char>(pos.y, pos.x + 1) != 64 &&

direction.at<float>(pos.y, pos.x + 1) > 67.5 &&

direction.at<float>(pos.y, pos.x + 1) <= 112.5 &&

sum.at<float>(pos.y, pos.x + 1) > sum.at<float>(pos.y - 1, pos.x + 1) &&

sum.at<float>(pos.y, pos.x + 1) > sum.at<float>(pos.y + 1, pos.x + 1))

{

returnImg.ptr<unsigned char>(pos.y, pos.x + 1)[0] = 255;

imageChanged = true;

}

}

}

else if (currentDirection > 22.5 && currentDirection <= 67.5)

{

if (pos.y > 0 && pos.x < workImg.cols - 1)

{

if (lowerThreshold <= sum.at<float>(Point(pos.x + 1, pos.y - 1)) &&

returnImg.at<unsigned char>(pos.y - 1, pos.x + 1) != 64 &&

direction.at<float>(pos.y - 1, pos.x + 1) > 22.5 &&

direction.at<float>(pos.y - 1, pos.x + 1) <= 67.5 &&

sum.at<float>(pos.y - 1, pos.x + 1) > sum.at<float>(pos.y - 2, pos.x) &&

sum.at<float>(pos.y - 1, pos.x + 1) > sum.at<float>(pos.y, pos.x + 2))

{

returnImg.ptr<unsigned char>(pos.y - 1, pos.x + 1)[0] = 255;

imageChanged = true;

}

}

if (pos.y < workImg.rows - 1 && pos.x>0)

{

if (lowerThreshold <= sum.at<float>(Point(pos.x - 1, pos.y + 1)) &&

returnImg.at<unsigned char>(pos.y + 1, pos.x - 1) != 64 &&

direction.at<float>(pos.y + 1, pos.x - 1) > 22.5 &&

direction.at<float>(pos.y + 1, pos.x - 1) <= 67.5 &&

sum.at<float>(pos.y + 1, pos.x - 1) > sum.at<float>(pos.y, pos.x - 2) &&

sum.at<float>(pos.y + 1, pos.x - 1) > sum.at<float>(pos.y + 2, pos.x))

{

returnImg.ptr<unsigned char>(pos.y + 1, pos.x - 1)[0] = 255;

imageChanged = true;

}

}

}

else

{

if (pos.y > 0)

{

if (lowerThreshold <= sum.at<float>(Point(pos.x, pos.y - 1)) &&

returnImg.at<unsigned char>(pos.y - 1, pos.x) != 64 &&

(direction.at<float>(pos.y - 1, pos.x) < 22.5 ||

direction.at<float>(pos.y - 1, pos.x) >= 157.5) &&

sum.at<float>(pos.y - 1, pos.x) > sum.at<float>(pos.y - 1, pos.x - 1) &&

sum.at<float>(pos.y - 1, pos.x) > sum.at<float>(pos.y - 1, pos.x + 2))

{

returnImg.ptr<unsigned char>(pos.y - 1, pos.x)[0] = 255;

imageChanged = true;

}

}

if (pos.y < workImg.rows - 1)

{

if (lowerThreshold <= sum.at<float>(Point(pos.x, pos.y + 1)) &&

returnImg.at<unsigned char>(pos.y + 1, pos.x) != 64 &&

(direction.at<float>(pos.y + 1, pos.x) < 22.5 ||

direction.at<float>(pos.y + 1, pos.x) >= 157.5) &&

sum.at<float>(pos.y + 1, pos.x) > sum.at<float>(pos.y + 1, pos.x - 1) &&

sum.at<float>(pos.y + 1, pos.x) > sum.at<float>(pos.y + 1, pos.x + 1))

{

returnImg.ptr<unsigned char>(pos.y + 1, pos.x)[0] = 255;

imageChanged = true;

}

}

}

}

}

}

MatIterator\_<unsigned char>current = returnImg.begin<unsigned char>();

MatIterator\_<unsigned char>final = returnImg.end<unsigned char>();

for (; current != final; ++current)

{

if (\*current == 64)

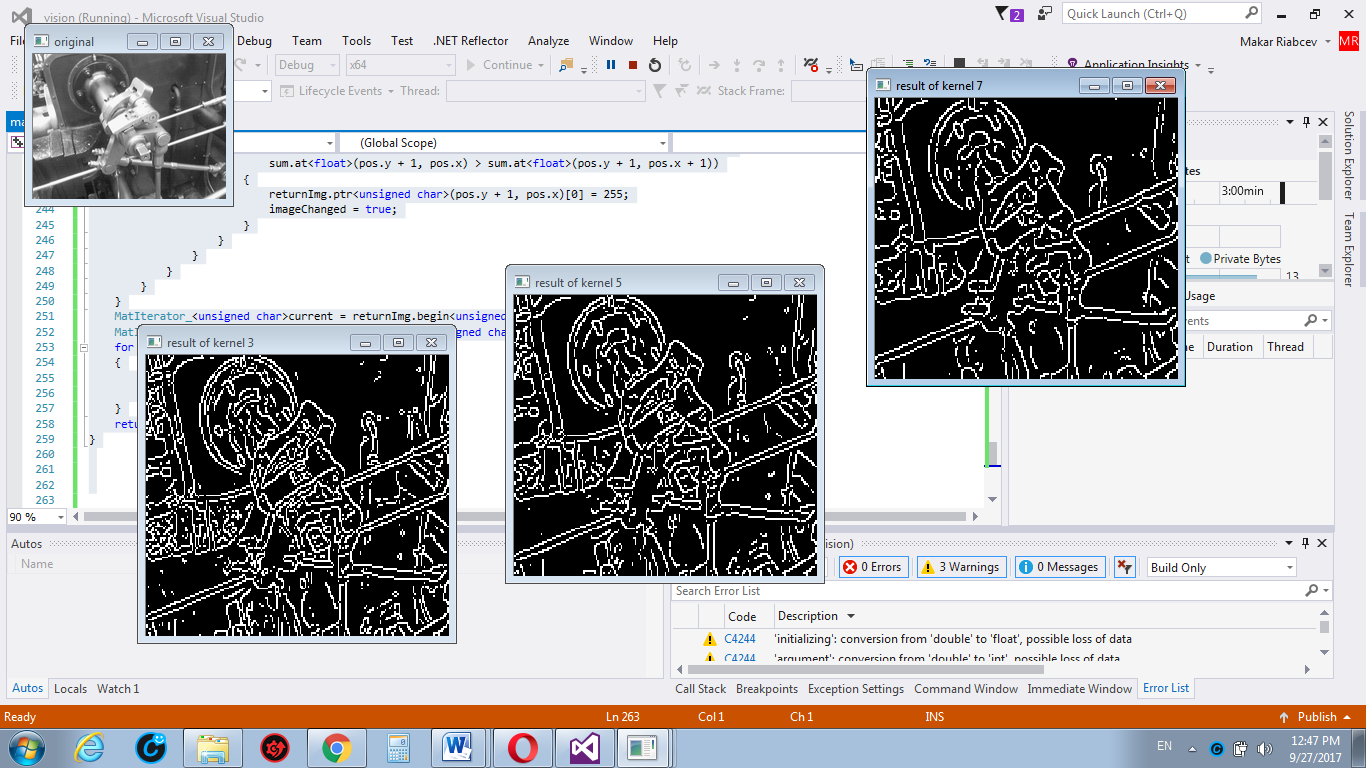
\*current = 255;

}

return returnImg;

}

**Result:**



**Comments:**

For this assignment I decided to use 3 different kernel sizes to see the difference and as the result the difference is well visible. The higher kernel size the more picture smooth and with less noise.

To implement MyCanny filter I had to do following steps:



1. Convert the original image to grayscale
2. Apply *Gaussian Blur filter* to reduce some noises
3. Determine the intensity gradients. It can be determined by using *Sobel filter.* After calculate the magnitude and angle of the directional gradients.
4. Apply non maximum suppression to thin out the edges. It works by finding the pixel with the maximum value in an edge. By certain conditions we keep this pixel, by another we make it black pixel.
5. Double thresholding must be applied to remove some noises in the image and to remove some edges which actually can not be edges. Low and High thresholdings help us to solve this problem.
6. Edge tracking by hysteresis. Weak edges that are connected to strong edges will be actual/real edges. Weak edges that are not connected to strong edges will be removed.

**Part B**

**Code:**

#include <iostream>

#include <vector>

#include <opencv2/highgui.hpp>

#include <opencv2/imgproc.hpp>

#include <opencv2/opencv.hpp>

#include <opencv2/core.hpp>

#include <opencv2/video/background\_segm.hpp>

#include <math.h>

using namespace cv;

using namespace std;

void changeColor(Mat temp, Mat grad\_x, Mat grad\_y, Mat &result, bool table = false);

const double M\_PI = 3.14;

//values for BGR colors

const Vec3b red = (Vec3b(0, 0, 255));

const Vec3b yel = (Vec3b(0, 255, 255));

const Vec3b blu = (Vec3b(255, 0, 0));

const Vec3b grn = (Vec3b(0, 255, 0));

const Vec3b blk = (Vec3b(0, 0, 0));

void colorGradient(Mat temp, Mat grad\_x, Mat grad\_y, Mat &result, bool table = false)

{

for (int y = 0; y < temp.rows; y++)

{

for (int x = 0; x < temp.cols; x++)

{

//skip black pixels

if (temp.at<uchar>(y, x) < 127)

{

continue;

}

//scale results to 0.0 to 1.0

short grad\_x\_px = grad\_x.at<short>(y, x);

double val\_x\_px = ((double)grad\_x\_px) / 32767.0;

short grad\_y\_px = grad\_y.at<short>(y, x);

double val\_y\_px = ((double)grad\_y\_px) / 32767.0;

//skip no-gradient areas

if (val\_y\_px == 0.0 && val\_x\_px == 0.0)

{

continue;

}

//tan(a) = dy/dx

double gradient = atan2(val\_y\_px, val\_x\_px);

// convert radian to degrees

double degs = gradient \* 180.0 / M\_PI;

degs = 180.0 - degs;

if (!table)

{

Mat toHSV(1, 1, CV\_8UC3, Scalar((degs \* 255) / 720, 255, 255));

Mat temp;

cvtColor(toHSV, temp, CV\_HSV2BGR);

result.at<Vec3b>(y, x) = temp.at<Vec3b>(0, 0);

}

// 4 types of angle ranges

else

{

if (degs < 45.0) {

result.at<Vec3b>(y, x) = red;

}

else if (degs >= 45.0 && degs < 135.0) {

result.at<Vec3b>(y, x) = yel;

}

else if (degs >= 135.0 && degs < 225.0) {

result.at<Vec3b>(y, x) = blu;

}

else if (degs >= 225.0 && degs < 315.0) {

result.at<Vec3b>(y, x) = grn;

}

else if (degs >= 315.0 && degs < 360.0) {

result.at<Vec3b>(y, x) = red;

}

}

}

}

}

int main(int argc, char \*\* argv) {

bool useTable = false;

Mat src, res, grad\_x, grad\_y;

src = imread("C:\\SobelGradient.png");

// convert to a grayscale

cvtColor(src, res, CV\_BGR2GRAY);

// make black and white

for (int y = 0; y < res.rows; y++) {

for (int x = 0; x < res.cols; x++)

{

if (res.at<uchar>(y, x) < 127)

{

res.at<uchar>(y, x) = 0;

}

else

{

res.at<uchar>(y, x) = 255;

}

}

}

// fill the circle so there's just one edge

floodFill(res, Point(res.cols / 2, res.rows / 2), Scalar(255));

// blur image

GaussianBlur(res, res, Size(15, 15), 0, 0, BORDER\_DEFAULT);

// sobel function generates the gradients

Sobel(res, grad\_x, CV\_16S, 1, 0, 3, 1, 0, BORDER\_DEFAULT);

Sobel(res, grad\_y, CV\_16S, 0, 1, 3, 1, 0, BORDER\_DEFAULT);

Mat result(src.rows, src.cols, CV\_8UC3, Scalar(0, 0, 0));

// color the image

colorGradient(res, grad\_x, grad\_y, result, useTable);

//show results

namedWindow("source", WINDOW\_NORMAL);

imshow("source", src);

namedWindow("result", WINDOW\_NORMAL);

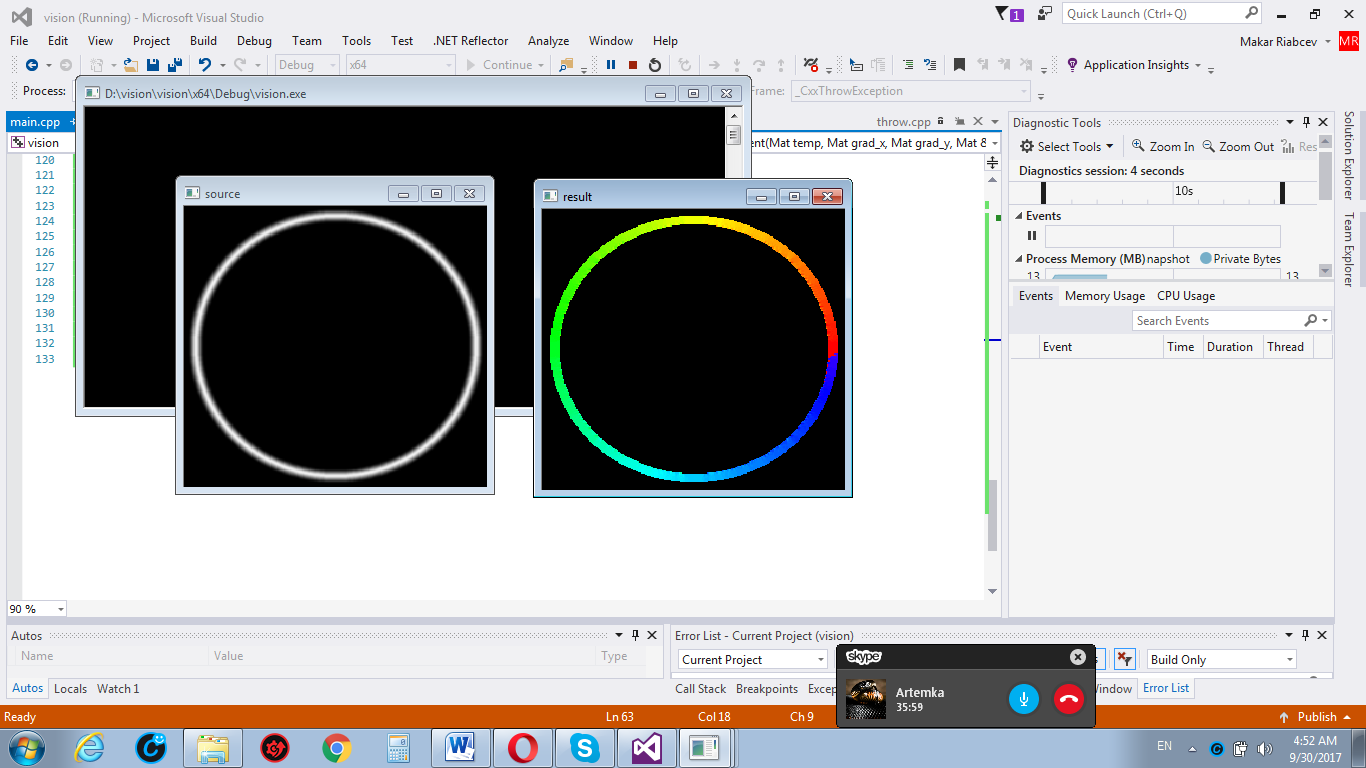
imshow("result", result);

waitKey(0);

return 0;

}

**Result:**



**Comments:**

The pixels of the circle were colored by the provided requirements, as it’s possible to see it on the result picture.

**Links:**

<https://portal.fhict.nl/Studentenplein/LMC/1718nj/Technology/minor-es/vision/VIS-Edges/INFO%20Implementing%20Canny%20from%20Scratch.pdf>

<http://justin-liang.com/tutorials/canny/>

<https://portal.fhict.nl/Studentenplein/LMC/1718nj/Technology/minor-es/vision/VIS-Edges/INFO%20Canny%20Edge%20Detection.pdf>