**Липецкий государственный технический университет**  
Факультет автоматизации и информатики  
Кафедра автоматизированных систем управления

Лабораторная работа №2

по дисциплине «Организация графических систем и систем мультимедиа»

Реализация алгоритма Canny для определения контуров изображения

Пархомов А.С.

Студент

Группа М-АС-21

Кургасов В.В.

Руководитель

Доцент

Липецк 2022 г.

Цель работы

Изучить метод Canny для определения контуров изображения с помощью GPU.

Задание кафедры

Реализовать алгоритм выделения границ и контуров Canny с обработкой на GPU и графическим интерфейсом пользователя.

Содержание

[1 Теоретические сведения 5](#_Toc103537135)

[2 Ход работы 6](#_Toc103537136)

[2.1 Использование приложения 6](#_Toc103537137)

[2.2 Создание установщика 8](#_Toc103537138)

[Вывод 10](#_Toc103537139)

[Приложение А 11](#_Toc103537140)

1. Теоретические сведения

В качестве языка программирования для реализации проекта была выбрана платформа .NET Framework и язык программирования C#. Графический интерфейс пользователя реализован с помощью фреймворка WPF. Данные инструменты были выбраны для достижения высокой производительности с использованием современных технологий. При разработке приложения были использованы только компоненты платформы без дополнительных библиотек и зависимостей. Взаимодействие с GPU обеспечивает платформа .NET Framework через метод параллелизации Parallel.For. Разработанный установщик позволяет использовать приложения на операционных системах семейства Windows начиная с версии Windows XP. Исходный код основной части программы представлен в приложении А.

Алгоритм Canny состоит из пяти шагов:

1. Сглаживание. Размытие изображения для удаления шума.
2. Поиск градиентов. Границы отмечаются там, где градиент изображения приобретает максимальное значение.
3. Подавление не-максимумов. Только локальные максимумы отмечаются как границы.
4. Двойная пороговая фильтрация. Потенциальные границы определяются порогами.
5. Трассировка области неоднозначности. Итоговые границы определяются путём подавления всех краёв, несвязанных с определенными (сильными) границами.

Перед применением детектора, изображение преобразуется в оттенки серого, чтобы уменьшить вычислительные затраты. Этот этап характерен для многих методов обработки изображений.

Для подавления шума, используют размытие изображения фильтром Гаусса.

Оператор Собеля часто применяют в алгоритмах выделения границ. По сути, это дискретный дифференциальный оператор, вычисляющий приближенное значение градиента яркости изображения. Результатом применения оператора Собеля в каждой точке изображения является либо вектор градиента яркости в этой точке, либо его норма. Оператор Собеля основан на свёртке изображения небольшими целочисленными фильтрами в вертикальном и горизонтальном направлениях, поэтому его относительно легко вычислять. С другой стороны, используемая им аппроксимация градиента достаточно грубая, особенно это сказывается на высокочастотных колебаниях изображения.

1. Ход работы

Разработанное приложение позволяет пользователю применить к загруженному изображению алгоритм Canny для выделения границ: пользователь может регулировать параметры алгоритма для достижения наилучшего результата. Кроме того, приложение позволяет сохранить обработанное изображение в разных форматах: png, jpeg и bmp.

* 1. Использование приложения

На рисунке 1 показана работа программы с загруженным изображением. На рисунке 2 отображён процесс сохранения обработанного изображения в требуемый формат.

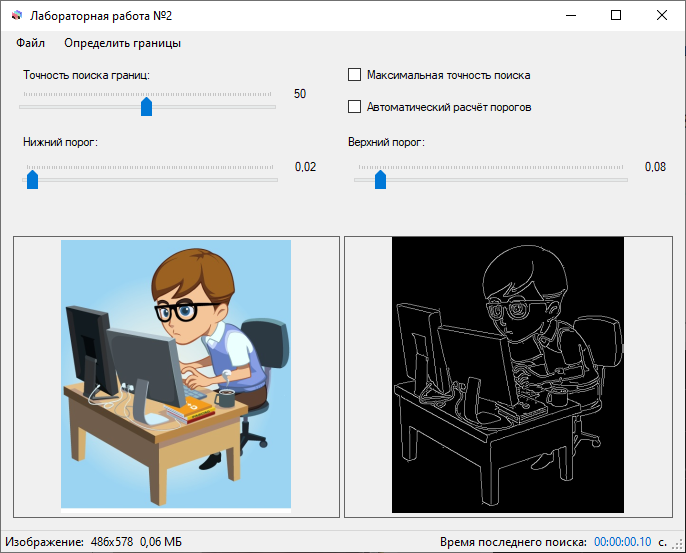


Рисунок 1 – Результат работы программы

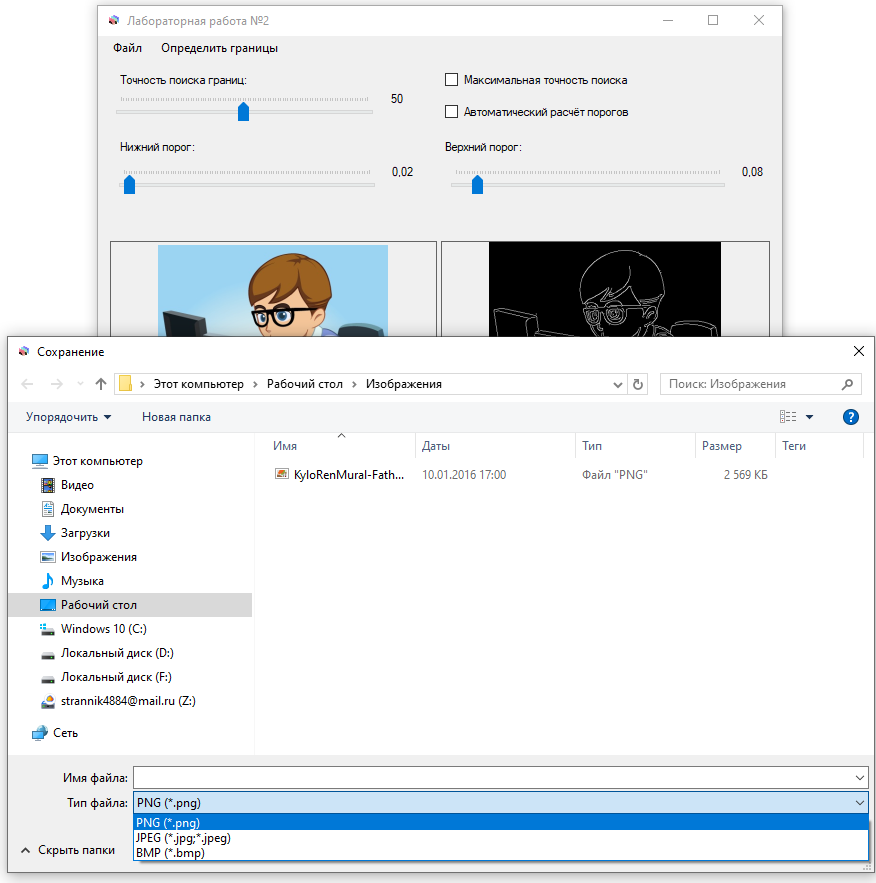


Рисунок 2 – Сохранение обработанного изображения

* 1. Создание установщика

Для удобного распространения разработанного приложения был создан установщик с помощью средств Visual Studio (рисунок 3). Данный установщик позволяет распространять как саму программу (рисунок 4), так и платформу .NET Framework требуемой версии (рисунок 5).

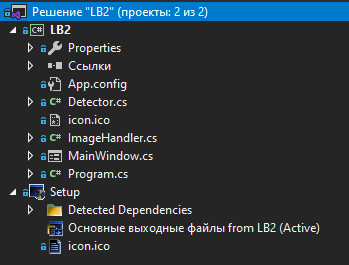


Рисунок 3 – Структура проекта с установщиком

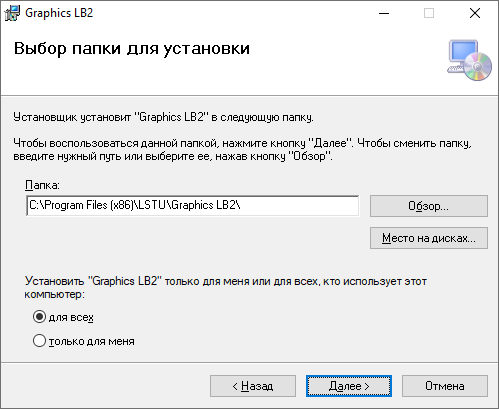


Рисунок 4 – Установка приложения

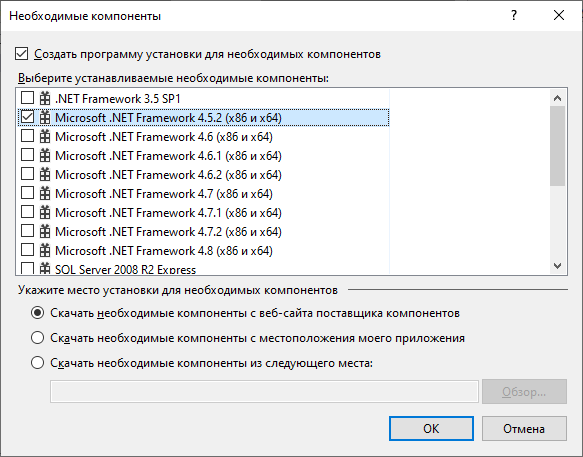


Рисунок 5 – Платформа .NET Framework, распространяемая вместе с установщиком

Вывод

В ходе выполнения данной лабораторной работы был изучен метод Canny для определения контуров изображения с помощью GPU и разработано приложение, которое позволяет пользователю применить к загруженному изображению алгоритм Canny для выделения границ: пользователь может регулировать параметры алгоритма для достижения наилучшего результата. Кроме того, приложение позволяет сохранить обработанное изображение в разных форматах: png, jpeg и bmp.

Также был создан установщик для удобного распространения приложения вместе с платформой .NET Framework.

Приложение А

Исходный код основных классов приложения

Исходный код класса ImageHandler.cs:

using System;

using System.Drawing.Imaging;

using System.Drawing;

namespace LB2

{

public class ImageHandler

{

private string bitmapPath;

private Bitmap currentBitmap;

private Bitmap originalBitmap;

private bool isGrayscale = false;

private byte bitsPerPixel;

// флаг применения фильтра оттенков серого

public bool IsGrayscale

{

get

{

return isGrayscale;

}

private set

{

isGrayscale = value;

}

}

// Bitmap оригинального изображения

public Bitmap OriginalBitmap

{

set

{

originalBitmap = value;

}

get

{

return originalBitmap;

}

}

// Bitmap текущего изображения

public Bitmap CurrentBitmap

{

get

{

return currentBitmap;

}

set

{

currentBitmap = value;

isGrayscale = false;

}

}

// путь до изображения

public string BitmapPath

{

get

{

return bitmapPath;

}

set

{

bitmapPath = value;

}

}

// функция получения количества бит на один пиксель

public byte GetBitsPerPixel(PixelFormat pixelFormat)

{

byte BitsPerPixel;

switch (pixelFormat)

{

case PixelFormat.Format8bppIndexed:

BitsPerPixel = 8;

break;

case PixelFormat.Format24bppRgb:

BitsPerPixel = 24;

break;

case PixelFormat.Format32bppArgb:

case PixelFormat.Format32bppPArgb:

BitsPerPixel = 32;

break;

default:

BitsPerPixel = 0;

break;

}

return BitsPerPixel;

}

// применение фильтра оттенков серого на текущее изображение

public unsafe void SetGrayscale()

{

// если текущий Bitmap пуст или фильтр уже был применён - выходим из функции

if (CurrentBitmap == null || isGrayscale)

{

return;

}

// получаем данные текущего Bitmap'а и блокируем его

BitmapData bData = currentBitmap.LockBits(new Rectangle(0, 0, currentBitmap.Width, currentBitmap.Height), ImageLockMode.ReadWrite, currentBitmap.PixelFormat);

// получаем количество битов на один пиксель

bitsPerPixel = GetBitsPerPixel(bData.PixelFormat);

// получаем указатель на данные Bitmap'а

byte\* scan0 = (byte\*)bData.Scan0.ToPointer();

// применяем фильтр

byte\* data;

// проходимся по всем пикселям

for (int i = 0; i < bData.Height; ++i)

{

for (int j = 0; j < bData.Width; ++j)

{

// вычисляем позицию начала текущего пикселя

data = scan0 + i \* bData.Stride + j \* bitsPerPixel / 8;

// если цветовая палитра больше или равна 24 битам на пиксель

if (bitsPerPixel >= 24)

{

// вычисляем значение фильтра

var gray = (byte)(.299 \* data[2] + .587 \* data[1] + .114 \* data[0]);

// применяем значение к цветовой палитре RGB

data[0] = gray;

data[1] = gray;

data[2] = gray;

}

}

}

// разблокируем Bitmap текущего изображения

currentBitmap.UnlockBits(bData);

isGrayscale = true;

}

// нормализуем Bitmap текущего изображения

public unsafe double[,] GetNormalizedMatrix()

{

// если текущий Bitmap пуст - выходим из функции

if (originalBitmap == null)

{

return null;

}

// получаем данные текущего Bitmap'а и блокируем его

BitmapData bData = originalBitmap.LockBits(new Rectangle(0, 0, originalBitmap.Width, originalBitmap.Height), ImageLockMode.ReadWrite, originalBitmap.PixelFormat);

// получаем количество битов на один пиксель

bitsPerPixel = GetBitsPerPixel(bData.PixelFormat);

// получаем указатель на данные Bitmap'а

byte\* scan0 = (byte\*)bData.Scan0.ToPointer();

// выделяем массив под нормализованный Bitmap

var normalizedMatrix = new double[originalBitmap.Width, originalBitmap.Height];

// нормализуем Bitmap

byte\* data;

// проходимся по всем пикселям

for (int i = 0; i < bData.Height; ++i)

{

for (int j = 0; j < bData.Width; ++j)

{

// вычисляем позицию начала текущего пикселя

data = scan0 + i \* bData.Stride + j \* bitsPerPixel / 8;

// нормализуем

normalizedMatrix[j, i] = data[0] / 255d;

}

}

// разблокируем Bitmap текущего изображения

originalBitmap.UnlockBits(bData);

return normalizedMatrix;

}

// денормализация Bitmap'а текущего изображения

public unsafe void DenormalizeCurrent(double[,] norm)

{

// если массив пуст - выходим из функции

if (norm == null)

{

return;

}

// размеры массива

int n = norm.GetLength(0);

int m = norm.GetLength(1);

// проверяем размеры массива с размерами текущего Bitmap'а

if (m != currentBitmap.Height || n != currentBitmap.Width)

{

throw new Exception("Sizes don't match.");

}

// получаем данные текущего Bitmap'а и блокируем его

BitmapData bData = currentBitmap.LockBits(new Rectangle(0, 0, currentBitmap.Width, currentBitmap.Height), ImageLockMode.ReadWrite, currentBitmap.PixelFormat);

// получаем количество битов на один пиксель

bitsPerPixel = GetBitsPerPixel(bData.PixelFormat);

// получаем указатель на данные Bitmap'а

byte\* scan0 = (byte\*)bData.Scan0.ToPointer();

// денормализуем Bitmap

byte\* data;

// проходимся по всем пикселям

for (int i = 0; i < bData.Height; ++i)

{

for (int j = 0; j < bData.Width; ++j)

{

// вычисляем позицию начала текущего пикселя

data = scan0 + i \* bData.Stride + j \* bitsPerPixel / 8;

// рассчитываем новое значение для цвета с учётом переполнения

byte newCol = norm[j, i] == 0 ? (byte)0 : (byte)255;

// если цветовая палитра больше или равна 24 битам на пиксель

if (bitsPerPixel >= 24)

{

// применяем значение к цветовой палитре RGB

data[0] = newCol;

data[1] = newCol;

data[2] = newCol;

}

else

{

data[0] = newCol;

}

}

}

// разблокируем Bitmap текущего изображения

currentBitmap.UnlockBits(bData);

}

// очистка всех данных

public void CleanUp()

{

currentBitmap = null;

originalBitmap = null;

isGrayscale = false;

bitsPerPixel = 0;

bitmapPath = "";

GC.Collect();

}

}

}

Исходный код класса Detector.cs:

using System;

using System.Threading.Tasks;

namespace LB2

{

public class Detector

{

private bool isMaxPrecision;

private double lowerTreshold;

private double upperTreshold;

private double[,] extractedX;

private double[,] extractedY;

private double[,] gradientValue;

private double[,] directionGradient;

private readonly double[,] xMatrix = { { 1, 0, -1 }, { 2, 0, -2 }, { 1, 0, -1 } };

private readonly double[,] yMatrix = { { -1, -2, -1 }, { 0, 0, 0 }, { 1, 2, 1 } };

private readonly double[,] gaussMatrix = {

{0.0121, 0.0261, 0.0337, 0.0261, 0.0121},

{0.0261, 0.0561, 0.0724, 0.0561, 0.0261},

{0.0337, 0.0724, 0.0935, 0.0724, 0.0337},

{0.0261, 0.0561, 0.0724, 0.0561, 0.0261},

{0.0121, 0.0261, 0.0337, 0.0261, 0.0121}

};

// флаг использования максимальной точности

public bool IsMaxPrecision

{

get

{

return isMaxPrecision;

}

set

{

isMaxPrecision = value;

}

}

// нижний порог срабатывания

public double LowerTreshold

{

get

{

return lowerTreshold;

}

set

{

lowerTreshold = value;

}

}

// верхний порог срабатывания

public double UpperTreshold

{

get

{

return upperTreshold;

}

set

{

upperTreshold = value;

}

}

// свёртка

private double[,] Convolution(double[,] imageInput, double[,] kernel, int radius)

{

if (imageInput == null) return null;

int yPosition = imageInput.GetLength(1);

int xPosition = imageInput.GetLength(0);

double[,] imageOutput = new double[xPosition, yPosition];

Parallel.For(0, xPosition, i => {

for (int j = 0; j < yPosition; j++)

{

double newValue = 0;

for (int innerI = i - radius; innerI < i + radius + 1; innerI++)

for (int innerJ = j - radius; innerJ < j + radius + 1; innerJ++)

{

int idxX = (innerI + xPosition) % xPosition;

int idxY = (innerJ + yPosition) % yPosition;

int kernelX = innerI - (i - radius);

int kernelY = innerJ - (j - radius);

newValue += imageInput[idxX, idxY] \* kernel[kernelX, kernelY];

}

imageOutput[i, j] = newValue;

}

});

return imageOutput;

}

// расширение матрицы

public double[,] ExpandMatrix(double[,] matrix, int expansion)

{

if (matrix == null) return null;

int x = matrix.GetLength(0), y = matrix.GetLength(1);

double[,] expandedMatrix = new double[x + 2 \* expansion, y + 2 \* expansion];

for (int i = -expansion; i < x + expansion - 1; i++)

for (int j = -expansion; j < y + expansion - 1; j++)

{

var ii = (i + x) % x;

int jj = (j + y) % y;

expandedMatrix[i + 2, j + 2] = matrix[ii, jj];

}

return expandedMatrix;

}

// определение границ

public double[,] Detection(double[,] normalImage, int precision)

{

if (normalImage == null) return null;

int blurX, blurY;

double[,] bluredMatrix;

try

{

bluredMatrix = Convolution(normalImage, gaussMatrix, 2);

blurX = bluredMatrix.GetLength(0); blurY = bluredMatrix.GetLength(1);

extractedX = Convolution(bluredMatrix, xMatrix, 1);

extractedY = Convolution(bluredMatrix, yMatrix, 1);

}

catch (OutOfMemoryException)

{

throw;

}

int xExcerptX = extractedX.GetLength(0), yExcerptY = extractedX.GetLength(1);

gradientValue = new double[xExcerptX, yExcerptY];

directionGradient = new double[xExcerptX, yExcerptY];

for (int x = 0; x < blurX; x++)

{

for (int y = 0; y < blurY; y++)

{

gradientValue[x, y] = Math.Sqrt(extractedX[x, y] \* extractedX[x, y] + extractedY[x, y] \* extractedY[x, y]);

double pom = Math.Atan2(extractedX[x, y], extractedY[x, y]);

if ((pom >= -Math.PI / 8 && pom < Math.PI / 8) || (pom <= -7 \* Math.PI / 8 && pom > 7 \* Math.PI / 8))

directionGradient[x, y] = 0;

else if ((pom >= Math.PI / 8 && pom < 3 \* Math.PI / 8) || (pom <= -5 \* Math.PI / 8 && pom > -7 \* Math.PI / 8))

directionGradient[x, y] = Math.PI / 4;

else if ((pom >= 3 \* Math.PI / 8 && pom <= 5 \* Math.PI / 8) || (-3 \* Math.PI / 8 >= pom && pom > -5 \* Math.PI / 8))

directionGradient[x, y] = Math.PI / 2;

else if ((pom < -Math.PI / 8 && pom >= -3 \* Math.PI / 8) || (pom > 5 \* Math.PI / 8 && pom <= 7 \* Math.PI / 8))

directionGradient[x, y] = -Math.PI / 4;

}

}

var max = GetMatrixMaxValue(gradientValue);

for (int i = 0; i < xExcerptX; i++)

{

for (int j = 0; j < yExcerptY; j++)

{

gradientValue[i, j] /= max;

}

}

if (upperTreshold == 0 && lowerTreshold == 0) GetThresholds(blurX, blurY);

for (int i = 0; i < xExcerptX; i++)

{

for (int j = 0; j < yExcerptY; j++)

{

gradientValue[i, j] = gradientValue[i, j] < lowerTreshold ? 0 : gradientValue[i, j];

}

}

for (var x = 1; x < blurX - 1; x++)

{

for (var y = 1; y < blurY - 1; y++)

{

if (directionGradient[x, y] == 0 && (gradientValue[x, y] <= gradientValue[x - 1, y] || gradientValue[x, y] <= gradientValue[x + 1, y]))

gradientValue[x, y] = 0;

else if (directionGradient[x, y] == Math.PI / 2 && (gradientValue[x, y] <= gradientValue[x, y - 1] || gradientValue[x, y + 1] >= gradientValue[x, y]))

gradientValue[x, y] = 0;

else if (directionGradient[x, y] == Math.PI / 4 && (gradientValue[x, y] <= gradientValue[x - 1, y + 1] || gradientValue[x, y] <= gradientValue[x + 1, y - 1]))

gradientValue[x, y] = 0;

else if (directionGradient[x, y] == -Math.PI / 4 && (gradientValue[x, y] <= gradientValue[x - 1, y - 1] || gradientValue[x, y] <= gradientValue[x + 1, y + 1]))

gradientValue[x, y] = 0;

}

}

for (var x = 2; x < blurX - 2; x++)

{

for (var y = 2; y < blurY - 2; y++)

{

if (directionGradient[x, y] == 0)

if (gradientValue[x - 2, y] > gradientValue[x, y] || gradientValue[x + 2, y] > gradientValue[x, y])

gradientValue[x, y] = 0;

if (directionGradient[x, y] == Math.PI / 2)

if (gradientValue[x, y - 2] > gradientValue[x, y] || gradientValue[x, y + 2] > gradientValue[x, y])

gradientValue[x, y] = 0;

if (directionGradient[x, y] == Math.PI / 4)

if (gradientValue[x - 2, y + 2] > gradientValue[x, y] || gradientValue[x + 2, y - 2] > gradientValue[x, y])

gradientValue[x, y] = 0;

if (directionGradient[x, y] == -Math.PI / 4)

if (gradientValue[x + 2, y + 2] > gradientValue[x, y] || gradientValue[x - 2, y - 2] > gradientValue[x, y])

gradientValue[x, y] = 0;

}

}

for (var x = 0; x < blurX; x++)

{

for (var y = 0; y < blurY; y++)

{

if (gradientValue[x, y] > upperTreshold)

gradientValue[x, y] = 1;

}

}

// начало расчёта гистерезиса

var pomH = 0;

var pomStaro = -1;

var passage = 0;

var isContinue = true;

while (isContinue)

{

passage++;

pomStaro = pomH;

for (int x = 1; x < xExcerptX - 1; x++)

{

for (int y = 1; y < yExcerptY - 1; y++)

{

if (gradientValue[x, y] <= upperTreshold && gradientValue[x, y] >= lowerTreshold)

{

double pom1 = gradientValue[x - 1, y - 1];

double pom2 = gradientValue[x, y - 1];

double pom3 = gradientValue[x + 1, y - 1];

double pom4 = gradientValue[x - 1, y];

double pom5 = gradientValue[x + 1, y];

double pom6 = gradientValue[x - 1, y + 1];

double pom7 = gradientValue[x, y + 1];

double pom8 = gradientValue[x + 1, y + 1];

if (pom1 == 1 || pom2 == 1 || pom3 == 1 || pom4 == 1 || pom5 == 1 || pom6 == 1 || pom7 == 1 || pom8 == 1)

{

gradientValue[x, y] = 1;

pomH++;

}

}

}

}

if (isMaxPrecision)

{

isContinue = pomH != pomStaro;

}

else

{

isContinue = passage <= precision;

}

}

for (int i = 0; i < xExcerptX; i++)

{

for (int j = 0; j < yExcerptY; j++)

{

if (gradientValue[i, j] <= upperTreshold)

gradientValue[i, j] = 0;

}

}

// конец расчёта гистерезиса

return gradientValue;

}

// автоматическое определение значений порогов

private void GetThresholds(int dimX, int dimY)

{

double sum = 0;

double number = 0;

for (var x = 1; x < dimX - 1; x++)

for (var y = 1; y < dimY - 1; y++)

{

if (gradientValue[x, y] != 0)

{

sum += gradientValue[x, y];

number++;

}

}

upperTreshold = sum / number;

lowerTreshold = 0.4 \* upperTreshold;

}

// поиск максимального значения в массиве

private double GetMatrixMaxValue(double[,] mat)

{

double m = -1;

foreach (var el in mat)

{

m = el > m ? el : m;

}

return m;

}

// очистка всех данных

public void CleanUp()

{

extractedX = null;

extractedY = null;

gradientValue = null;

directionGradient = null;

GC.Collect();

}

}

}