Министерство образования и науки Российской федерации

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

«Алтайский государственный технический университет им. И. И.Ползунова»

Факультет информационных технологий

Кафедра прикладной математики

Отчет защищен с оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

«\_\_\_\_\_\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2015 г.

Отчет по лабораторной работе №1

Свертка

«Компьютерное зрение»

ИЗ 231000.3.000 О

Студент группы 8ПИ-41 А.Ю. Смирнова

(И.О. Фамилия)

Преподаватель М.Г. Казаков

(должность, ученое звание) (И.О. Фамилия)

Барнаул 2015

Задание:

* Написать основу для представления изображений и их обработки свертками. Дополнительно реализовать:
  + раздельную обработку по осям для сепарабельных фильтров
  + различные варианты обработки краевых эффектов при свертке
* Реализовать вычисление частных производных и оператора Собеля
* Реализовать отображение полученных результатов

Решение:

В ходе решения поставленной задачи была разработана следующая система классов:

1. Image – исходное изображение. Включает функции получения и обработки изображения.
2. Mask – маска для фильтра
3. SeparatedMask – маска для сепарабельных фильтров
4. MaskFactory – фабрика для получения масок
5. FilterManager – класс, реализующий обработку изображения заданным фильтром.

Исходный код:

image.h

#ifndef IMAGE\_H

#define IMAGE\_H

#include <memory>

#include <limits>

#include <QImage>

#include <QtDebug>

#include <math.h>

using namespace std;

enum EdgeMode

{

ZEROS = 1,

MIRROR = 2,

COPY = 3

};

class Image

{

public:

Image(int h, int w);

static shared\_ptr<Image> fromFile(const QString &fileName);

static shared\_ptr<Image> fromQImage(QImage picture);

QImage toQImage() const;

bool toFile(const QString &fileName)const;

shared\_ptr<Image> compress(int scale) const;

void normalize(float min, float max, float bottom = numeric\_limits<float>::max(), float top = numeric\_limits<float>::min());

float getPixel(int i, int j, EdgeMode mode=EdgeMode::ZEROS) const;

float setPixel(int i, int j, float value);

int getHeight() const;

int getWidth() const;

private:

unique\_ptr<float[]> image;

int height;

int width;

};

#endif

image.cpp

#include "image.h"

Image::Image(int h, int w)

{

image = make\_unique<float[]>(h\*w);

height = h;

width = w;

}

shared\_ptr<Image> Image::fromFile(const QString& fileName)

{

QImage picture;

bool isLoaded = picture.load(fileName);

if(isLoaded)

{

qDebug() << "Load image from file: " << fileName;

return fromQImage(picture);

}

else

{

qWarning() << "Failed to load image from file";

return nullptr;

}

}

shared\_ptr<Image> Image::fromQImage(QImage picture)

{

shared\_ptr<Image> result = make\_shared<Image>(picture.height(), picture.width());

QRgb original;

float color;

for(int i=0; i<picture.height(); i++)

{

for(int j=0; j<picture.width(); j++)

{

original = picture.pixel(j,i);

color = 0.299 \* qRed(original) +

0.587 \* qGreen(original) +

0.114 \* qBlue(original);

result->setPixel(i,j,color);

}

}

return result;

}

QImage Image::toQImage() const

{

QImage result = QImage(width, height, QImage::Format\_RGB32);

int color;

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

{

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

{

color = qRound(getPixel(i, j));

result.setPixel(j, i, qRgb(color,color,color));

}

}

return result;

}

bool Image::toFile(const QString &fileName) const

{

QImage image = toQImage();

return image.save(fileName, "JPEG");

}

shared\_ptr<Image> Image::compress(int scale) const

{

shared\_ptr<Image> result = make\_shared<Image>(height/scale, width/scale);

for(int i=0; i<result->getHeight(); i++)

{

for(int j=0; j<result->getWidth(); j++)

{

result->setPixel(i,j,getPixel(i\*scale, j\*scale));

}

}

return result;

}

float Image::getPixel(int i, int j, EdgeMode mode) const

{

if(i<height && j<width && i>=0 && j>=0)

{

return image[i\*width + j];

}

else

{

switch(mode)

{

case EdgeMode::ZEROS: return 0;

case EdgeMode::COPY:

return image[min(max(i,0), height-1)\*width + min(max(j,0), width-1)];

case EdgeMode::MIRROR:

if(i<0) i = i\*(-1) - 1;

if(j<0) j = j\*(-1) - 1;

if(i>=height) i = height - (i%height) - 1;

if(j>=width) j = width - (j%width) - 1;

return image[i\*width + j];

}

}

return 0;

}

float Image::setPixel(int i, int j, float value)

{

if(i<height && j<width && i>=0 && j>=0)

{

image[i\*width + j] = value;

return value;

}

else

{

qFatal("setPixel: index out of the range");

return -1;

}

}

void Image::normalize(float min, float max, float bottom, float top)

{

if(bottom == numeric\_limits<float>::max() || top == numeric\_limits<float>::min() )

{

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

{

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

{

if(image[i\*width+j] < bottom)

{

bottom = image[i\*width+j];

}

else if(image[i\*width+j] > top)

{

top = image[i\*width+j];

}

}

}

}

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

{

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

{

image[i\*width+j] = min + (max - min)\*(image[i\*width+j] - bottom)/(top - bottom);

}

}

}

int Image::getHeight() const

{

return height;

}

int Image::getWidth() const

{

return width;

}

mask.h

#ifndef MASK\_H

#define MASK\_H

#include <memory>

#include <QtGlobal>

using namespace std;

class Mask

{

public:

Mask(int h, int w);

float getPixel(int i, int j) const;

float setPixel(int i, int j, float value);

int getHeight() const;

int getWidth() const;

~Mask();

private:

unique\_ptr<float[]> core;

int height;

int width;

};

#endif

mask.cpp

#include "mask.h"

Mask::Mask(int h, int w)

{

this->height = h;

this->width = w;

core = make\_unique<float[]>(h\*w);

}

float Mask::getPixel(int i, int j) const

{

if(i<height && j<width && i>=0 && j>=0)

{

return core[i\*width + j];

}

else

{

qFatal("getPixel: index out of the range");

return -1;

}

}

float Mask::setPixel(int i, int j, float value)

{

if(i<height && j<width && i>=0 && j>=0)

{

core[i\*width + j] = value;

return value;

}

else

{

qFatal("setPixel: index out of the range");

return -1;

}

}

int Mask::getHeight() const

{

return height;

}

int Mask::getWidth() const

{

return width;

}

Mask::~Mask()

{

}

separatedmask.h

#ifndef SEPARATEDMASK\_H

#define SEPARATEDMASK\_H

#include <memory>

#include <QDebug>

#include "mask.h"

using namespace std;

class SeparatedMask

{

public:

shared\_ptr<Mask> setRow(shared\_ptr<Mask> r);

shared\_ptr<Mask> setColumn(shared\_ptr<Mask> c);

shared\_ptr<Mask> getRow() const;

shared\_ptr<Mask> getColumn() const;

~SeparatedMask();

private:

shared\_ptr<Mask> row;

shared\_ptr<Mask> column;

};

#endif

separatedmask.cpp

#include "separatedmask.h"

shared\_ptr<Mask> SeparatedMask::setRow(shared\_ptr<Mask> r)

{

if(r->getHeight() == 1)

{

row = r;

return row;

}

else

{

qFatal("setRow: wrong row height");

return nullptr;

}

}

shared\_ptr<Mask> SeparatedMask::setColumn(shared\_ptr<Mask> c)

{

if(c->getWidth() == 1)

{

column = c;

return column;

}

else

{

qFatal("setColum: wrong column width");

return nullptr;

}

}

shared\_ptr<Mask> SeparatedMask::getRow() const

{

return row;

}

shared\_ptr<Mask> SeparatedMask::getColumn() const

{

return column;

}

SeparatedMask::~SeparatedMask()

{

}

maskfactory.h

#ifndef MASKFACTORY\_H

#define MASKFACTORY\_H

#include "mask.h"

#include "separatedmask.h"

#define M\_PI 3.14159265358979323846

#define M\_E 2.71828182845905

enum Direction{

LEFT = 0,

RIGHT = 1,

UP = 2,

DOWN = 3

};

enum Asix{

X = 0,

Y = 1

};

class MaskFactory

{

public:

MaskFactory();

static shared\_ptr<Mask> Shift(int pixelNum, Direction direction);

static shared\_ptr<Mask> Blur(int k, int blur);

static shared\_ptr<Mask> Sobel(Asix asix);

static shared\_ptr<Mask> Pruit(Asix asix);

static shared\_ptr<Mask> Shar(Asix asix);

static shared\_ptr<Mask> Gauss(float sigma);

static shared\_ptr<SeparatedMask> GaussSeparated(float sigma);

static shared\_ptr<SeparatedMask> SobelSeparated(Asix asix);

~MaskFactory();

};

#endif // MASKFACTORY\_H

maskfactory.cpp

#include "maskfactory.h"

MaskFactory::MaskFactory()

{

}

shared\_ptr<Mask> MaskFactory::Shift(int pixelNum, Direction direction)

{

int size = pixelNum\*2+1;

shared\_ptr<Mask> mask = make\_shared<Mask>(size,size);

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

{

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

{

mask->setPixel(i,j,0);

}

}

switch(direction)

{

case Direction::UP:

mask->setPixel(0, pixelNum, 1);

break;

case Direction::RIGHT:

mask->setPixel(pixelNum, size - 1, 1);

break;

case Direction::DOWN:

mask->setPixel(size - 1, pixelNum, 1);

break;

case Direction::LEFT:

mask->setPixel(pixelNum, 0, 1);

break;

}

return mask;

}

shared\_ptr<Mask> MaskFactory::Blur(int k, int blur)

{

int size = k\*2+1;

shared\_ptr<Mask> mask = make\_shared<Mask>(size,size);

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

{

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

{

mask->setPixel(i,j, 1./blur);

}

}

return mask;

}

shared\_ptr<Mask> MaskFactory::Sobel(Asix asix)

{

shared\_ptr<Mask> result = make\_shared<Mask>(3,3);

float \*mask;

switch(asix)

{

case Asix::X:

static float mask1[] = {-1,0,1,-2,0,2,-1,0,1};

mask = mask1;

break;

case Asix::Y:

static float mask2[] = {-1,-2,-1,0,0,0,1,2,1};

mask = mask2;

break;

}

for(int i=0; i<3; i++)

{

for(int j=0; j<3; j++)

{

result->setPixel(i,j,mask[i\*3+j]);

}

}

return result;

}

shared\_ptr<Mask> MaskFactory::Pruit(Asix asix)

{

shared\_ptr<Mask> result = make\_shared<Mask>(3,3);

float \*mask;

switch(asix)

{

case Asix::X:

static float mask1[] = {-1,0,1,-1,0,1,-1,0,1};

mask = mask1;

break;

case Asix::Y:

static float mask2[] = {-1,-1,-1,0,0,0,1,1,1};

mask = mask2;

}

for(int i=0; i<3; i++)

{

for(int j=0; j<3; j++)

{

result->setPixel(i,j,mask[i\*3+j]);

}

}

return result;

}

shared\_ptr<Mask> MaskFactory::Shar(Asix asix)

{

shared\_ptr<Mask> result = make\_shared<Mask>(3,3);

float \*mask;

switch(asix)

{

case Asix::X:

static float mask1[] = {-3,0,3,-10,0,10,-3,0,3};

mask = mask1;

break;

case Asix::Y:

static float mask2[] = {-3,-10,-3,0,0,0,3,10,3};

mask = mask2;

}

for(int i=0; i<3; i++)

{

for(int j=0; j<3; j++)

{

result->setPixel(i,j,mask[i\*3+j]);

}

}

return result;

}

shared\_ptr<Mask> MaskFactory::Gauss(float sigma)

{

int k = 3\*sigma;

int size = 2\*k+1;

shared\_ptr<Mask> result = make\_shared<Mask>(size,size);

int x, y;

float value;

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

{

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

{

x = i - k;

y = j - k;

value = (pow(M\_E,-(x\*x + y\*y)/(2\*sigma\*sigma)))/(2\*M\_PI\*sigma\*sigma);

result->setPixel(i,j,value);

}

}

return result;

}

shared\_ptr<SeparatedMask> MaskFactory::SobelSeparated(Asix asix)

{

shared\_ptr<SeparatedMask> result = make\_shared<SeparatedMask>();

float \*row;

float \*column;

shared\_ptr<Mask> rowMask = make\_shared<Mask>(1,3);

shared\_ptr<Mask> columnMask = make\_shared<Mask>(3,1);

switch(asix)

{

case Asix::X:

static float row1[] = {-1,0,-1};

static float column1 [] = {1,2,1};

row = row1;

column = column1;

break;

case Asix::Y:

static float row2[] = {1,2,1};

static float column2 [] = {1,0,-1};

row = row2;

column = column2;

}

for(int i=0; i<3; i++)

{

rowMask->setPixel(0, i, row[i]);

columnMask->setPixel(i, 0, column[i]);

}

result->setColumn(columnMask);

result->setRow(rowMask);

return result;

}

shared\_ptr<SeparatedMask> MaskFactory::GaussSeparated(float sigma)

{

int k = 3\*sigma;

int size = 2\*k+1;

shared\_ptr<SeparatedMask> result = make\_shared<SeparatedMask>();

shared\_ptr<Mask> rowMask = make\_shared<Mask>(1,size);

shared\_ptr<Mask> columnMask = make\_shared<Mask>(size,1);

int x;

float value;

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

{

x = i - k;

value = (pow(M\_E,-(x\*x)/(2\*sigma\*sigma)))/(sqrt(2\*M\_PI)\*sigma);

rowMask->setPixel(0, i, value);

columnMask->setPixel(i, 0, value);

}

result->setColumn(columnMask);

result->setRow(rowMask);

return result;

}

MaskFactory::~MaskFactory()

{

}

filtermanager.h

#ifndef FILTERMANAGER\_H

#define FILTERMANAGER\_H

#include"image.h"

#include"mask.h"

#include"maskfactory.h"

#include"separatedmask.h"

class FilterManager

{

public:

static shared\_ptr<Image> Filter(const Image &source, const Mask& mask, bool norm = true, EdgeMode edgeMode = EdgeMode::ZEROS);

static shared\_ptr<Image> SeparatedFilter(const Image& source, const SeparatedMask &mask, bool norm = true, EdgeMode edgeMode = EdgeMode::ZEROS);

static shared\_ptr<Image> SobelOperator(const Image &source, EdgeMode edgeMode = EdgeMode::ZEROS);

static shared\_ptr<Image> SobelOperatorSeparated(const Image &source, EdgeMode edgeMode = EdgeMode::ZEROS);

};

#endif

filtermanager.cpp

#include "filtermanager.h"

shared\_ptr<Image> FilterManager::Filter(const Image &source, const Mask &mask, bool norm, EdgeMode edgeMode)

{

shared\_ptr<Image> result = make\_shared<Image>(source.getHeight(), source.getWidth());

float pixel;

int heighHalf = (mask.getHeight() - 1)/2;

int widthHalf = (mask.getWidth() - 1)/2;

float max = numeric\_limits<float>::min(), min = numeric\_limits<float>::max() ;

for(int i=0; i<result->getHeight(); i++)

{

for(int j=0; j<result->getWidth(); j++)

{

pixel = 0;

for(int u = 0; u<mask.getHeight(); u++)

{

for(int v = 0; v<mask.getWidth(); v++)

{

pixel = pixel + mask.getPixel(u,v) \* source.getPixel(i-(u-heighHalf), j-(v-widthHalf), edgeMode);

}

}

result->setPixel(i, j, pixel);

if(pixel > max)

{

max = pixel;

}

if(pixel < min)

{

min = pixel;

}

}

}

if(norm) result->normalize(0,255,min,max);

return result;

}

shared\_ptr<Image> FilterManager::SobelOperator(const Image &source, EdgeMode edgeMode)

{

shared\_ptr<Image> gradX = FilterManager::Filter(source, \*(MaskFactory::Sobel(Asix::X)), false, edgeMode);

shared\_ptr<Image> gradY = FilterManager::Filter(source, \*(MaskFactory::Sobel(Asix::Y)), false, edgeMode);

shared\_ptr<Image> result = make\_shared<Image>(source.getHeight(), source.getWidth());

float gX, gY;

float max = numeric\_limits<float>::min(), min = numeric\_limits<float>::max();

float pixel;

for(int i=0; i<result->getHeight(); i++)

{

for(int j=0; j<result->getWidth(); j++)

{

gX = gradX->getPixel(i,j);

gY = gradY->getPixel(i,j);

pixel = sqrt(gX\*gX + gY\*gY);

result->setPixel(i,j, pixel);

if(pixel > max)

{

max = pixel;

}

if(pixel < min)

{

min = pixel;

}

}

}

result->normalize(0, 255, min, max);

return result;

}

shared\_ptr<Image> FilterManager::SeparatedFilter(const Image &source, const SeparatedMask &mask, bool norm, EdgeMode edgeMode)

{

shared\_ptr<Image> rResult = FilterManager::Filter(source, \*(mask.getRow()), false, edgeMode);

shared\_ptr<Image> cResult = FilterManager::Filter(\*rResult, \*(mask.getColumn()), false, edgeMode);

if(norm) cResult->normalize(0,255);

return cResult;

}

shared\_ptr<Image> FilterManager::SobelOperatorSeparated(const Image &source, EdgeMode edgeMode)

{

shared\_ptr<Image> gradX = FilterManager::SeparatedFilter(source, \*(MaskFactory::SobelSeparated(Asix::X)), false, edgeMode);

shared\_ptr<Image> gradY = FilterManager::SeparatedFilter(source, \*(MaskFactory::SobelSeparated(Asix::Y)), false, edgeMode);

shared\_ptr<Image> result = make\_shared<Image>(source.getHeight(), source.getWidth());

float gX, gY;

float max = numeric\_limits<float>::min(), min = numeric\_limits<float>::max();

float pixel;

for(int i=0; i<result->getHeight(); i++)

{

for(int j=0; j<result->getWidth(); j++)

{

gX = gradX->getPixel(i,j);

gY = gradY->getPixel(i,j);

pixel = sqrt(gX\*gX + gY\*gY);

result->setPixel(i,j, pixel);

if(pixel > max)

{

max = pixel;

}

if(pixel < min)

{

min = pixel;

}

}

}

result->normalize(0, 255, min, max);

return result;

}

Пример использования:



Рисунок 1. Исходное изображение.

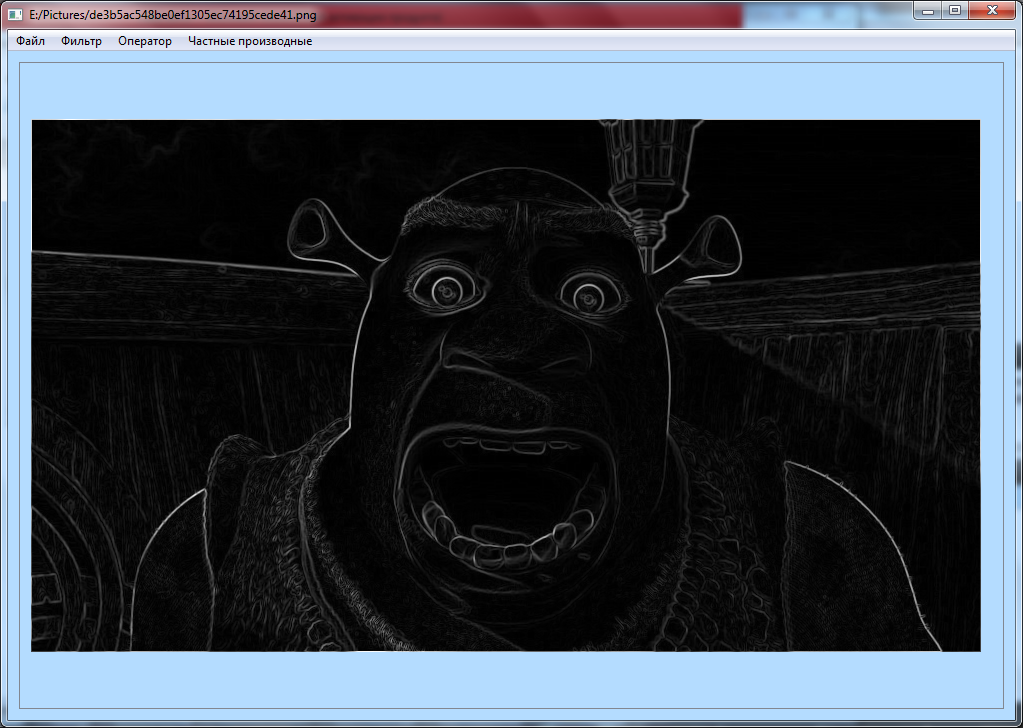


Рисунок 2. Изображение после применения оператора Собеля.



Рисунок 3. Изображение, обработанное фильтром Гаусса с σ = 5.