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 №6

Инвариантность дескрипторов к изменениям масштаба

«Интеллектуальные технологии обработки изображений»

ИЗ 231000.3.000 О

Студент группы 8ПИ-41 А.Ю. Смирнова

(И.О. Фамилия)

Преподаватель старший преподаватель М.Г. Казаков

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

Барнаул 2015

**Цель работы**

Познакомиться с возможностью достижения относительной инвариантности дескрипторов к изменению масштаба изображений.

**Формулировка задачи**

* Реализовать подобную методу SIFT схему выбора масштаба изображения, на котором будет рассчитываться дескриптор окрестности интересной точки. Для этого включить построение гауссовой пирамиды с разделением на октавы в фазу подготовки изображения.
* Реализовать алгоритм поиска экстремумов в наборе DoG. Внести соответствующие изменения в алгоритм поиска интересных точек.
* Сравнить полученные результаты, сравнить с полученными в пятой работе.

**Исходый код:**

pyramid.cpp

shared\_ptr<Pyramid> Pyramid::build(const Image &image, int octaveNum, int levelNum, float sigma0, float sigmaInit)

{

if(octaveNum > 0 && levelNum > 0)

{

shared\_ptr<Pyramid> result = make\_shared<Pyramid>(octaveNum, levelNum);

shared\_ptr<Image> next, top1, top2;

float k = pow(2, 1./levelNum);

sigma0 = sigma0 / k;

float curSigma = sigma0;

float bottomSigma = sigma0;

auto octaveBottom = FilterManager::SeparatedFilter(image, \*MaskFactory::GaussSeparated(sqrt(sigma0\*sigma0 - sigmaInit\*sigmaInit)), false, EdgeMode::MIRROR);

curSigma \*= k;

float innerSigma;

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

{

innerSigma = 1;

result->images.push\_back(make\_shared<PyramidLevel>(octaveBottom, i, -1, bottomSigma, innerSigma));

next = octaveBottom;

curSigma = bottomSigma;

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

{

next = FilterManager::SeparatedFilter(\*next, \*MaskFactory::GaussSeparated(curSigma\*sqrt(k\*k-1)), false, EdgeMode::MIRROR);

curSigma \*= k;

result->images.push\_back(make\_shared<PyramidLevel>(next, i, j, curSigma, innerSigma));

innerSigma \*= k;

}

octaveBottom = next->compress(2);

bottomSigma = curSigma;

top1 = FilterManager::SeparatedFilter(\*next, \*MaskFactory::GaussSeparated(curSigma\*sqrt(k\*k-1)), false, EdgeMode::MIRROR);

curSigma \*= k;

result->images.push\_back(make\_shared<PyramidLevel>(top1, i, levelNum, curSigma, innerSigma));

innerSigma \*= k;

top2 = FilterManager::SeparatedFilter(\*top1, \*MaskFactory::GaussSeparated(curSigma\*sqrt(k\*k-1)), false, EdgeMode::MIRROR);

curSigma \*= k;

result->images.push\_back(make\_shared<PyramidLevel>(top2, i, levelNum + 1, curSigma, innerSigma));

innerSigma \*= k;

}

shared\_ptr<Image> DoG;

shared\_ptr<Image> top, bottom;

for(int i=0; i<result->images.size()-1; i++)

{

if(result->images[i]->getOctave() == result->images[i+1]->getOctave())

{

top = result->images[i+1]->getImage();

bottom = result->images[i]->getImage();

DoG = make\_shared<Image>(top->getHeight(), top->getWidth());

for(int y=0; y<DoG->getHeight(); y++)

{

for(int x = 0; x<DoG->getWidth(); x++)

{

DoG->setPixel(y,x,top->getPixel(y,x) - bottom->getPixel(y,x));

}

}

result->DoG.push\_back(make\_shared<PyramidLevel>(DoG, result->images[i]->getOctave(), result->images[i]->getLevel(), result->images[i]->getSigma(), result->images[i]->getInnerSigma()));

}

}

return result;

}

else

{

qFatal("Wrong level or octave number");

return nullptr;

}

}

vector<Point> Pyramid::findLocalMaximaAndMinima(int halfWindow) const

{

vector<Point> result;

float curValue, value;

bool isLocalMaxMin;

int more, less;

int border;

float trace, det;

shared\_ptr<Image> dxx, dyy, dxy;

for(int dog = 1; dog < DoG.size() - 1; dog++)

{

if(DoG[dog-1]->getOctave() != DoG[dog]->getOctave() || DoG[dog+1]->getOctave() != DoG[dog]->getOctave())

continue;

dxx = FilterManager::SeparatedFilter(\*DoG[dog]->getImage(), \*MaskFactory::SobelSeparated(Asix::X));

dyy = FilterManager::SeparatedFilter(\*DoG[dog]->getImage(), \*MaskFactory::SobelSeparated(Asix::Y));

dxy = FilterManager::SeparatedFilter(\*dxx, \*MaskFactory::SobelSeparated(Asix::Y));

dyy = FilterManager::SeparatedFilter(\*dyy, \*MaskFactory::SobelSeparated(Asix::Y));

dxx = FilterManager::SeparatedFilter(\*dxx, \*MaskFactory::SobelSeparated(Asix::X));

border = halfWindow\*DoG[dog]->getInnerSigma()\*sqrt(2);

for(int x = border; x < DoG[dog]->getImage()->getWidth()-border; x++)

{

for(int y = border; y < DoG[dog]->getImage()->getHeight()-border; y++)

{

curValue = DoG[dog]->getImage()->getPixel(y, x);

if(fabs(curValue) < 8) continue;

isLocalMaxMin = true;

more = 0;

less = 0;

for(int dNum = -1; dNum <= 1; dNum++)

{

for(int dx = -1; dx <= 1; dx++)

{

for(int dy = -1; dy <= 1; dy++)

{

if(dNum == 0 && dx == 0 && dy == 0) continue;

value = DoG[dog + dNum]->getImage()->getPixel(y + dy, x + dx);

if(curValue > value)

{

more++;

if(less > 0)

{

isLocalMaxMin = false;

goto exit;

}

}

else if(curValue < value)

{

less++;

if(more > 0)

{

isLocalMaxMin = false;

goto exit;

}

}

}

}

}

exit: if(isLocalMaxMin)

{

//удаление точек на краях

trace = dxx->getPixel(y,x) + dyy->getPixel(y,x);

det = dxx->getPixel(y,x) \* dyy->getPixel(y,x) - pow(dxy->getPixel(y,x),2);

if(pow(trace,2) / det <= pow(11,2)/10)

{

Point p(x, y, curValue,DoG[dog]->getSigma(), DoG[dog]->getInnerSigma());

auto angles = getLevel(DoG[dog]->getSigma())->getGenerator()->getDescriptor(p, 16, 1, 36)->getMaxAngle();

p.angle = angles.first;

p.x \*= pow(2, DoG[dog]->getOctave());

p.y \*= pow(2, DoG[dog]->getOctave());

result.push\_back(p);

if(angles.second > numeric\_limits<float>::min())

{

Point p1(x, y, curValue,DoG[dog]->getSigma(), DoG[dog]->getInnerSigma());

p.angle = angles.second;

p.x \*= pow(2, DoG[dog]->getOctave());

p.y \*= pow(2, DoG[dog]->getOctave());

result.push\_back(p1);

}

}

}

}

}

}

return result;

}

Pyramid::~Pyramid()

{

}

shared\_ptr<PyramidLevel> Pyramid::getLevel(float sigma) const

{

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

{

if(sigma == images[i]->getSigma() && images[i]->getLevel() > -1 && images[i]->getLevel() < levelsNum)

{

return images[i];

}

}

return nullptr;

}

shared\_ptr<PyramidLevel> Pyramid::getLevel(int octave, int level) const

{

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

{

if(octave == images[i]->getOctave() && level == images[i]->getLevel())

{

return images[i];

}

}

return nullptr;

}

shared\_ptr<Descriptor> Pyramid::getDescriptor(Point p, int surSize, int gistNum, int basketNum) const

{

auto level = getLevel(p.scale);

//find point angle

//auto gist = level->getGenerator()->getDescriptor(p1, surSize, 1, 36);

p.x = p.x / pow(2, level->getOctave());

p.y = p.y / pow(2, level->getOctave());

auto desk = level->getGenerator()->getDescriptor(p, surSize, gistNum, basketNum);

p.x = p.x \* pow(2, level->getOctave());

p.y = p.y \* pow(2, level->getOctave());

desk->point = p;

return desk;

}

detectors.cpp

vector<Point> Detectors::ScaleInvariant(const Image &image, const Pyramid& pyramid, int halfWindow)

{

vector<Point> result;

result = pyramid.findLocalMaximaAndMinima(halfWindow);

return result;

}

**Результаты работы программы:**

****