

**FACULTY OF INFORMATION TECHNOLOGIES**

**DEPARTMENT OF INFORMATION SYSTEMS MANAGEMENT**

**MACHINE VISION**

*Report*

**Laboratory work #8**

|  |  |
| --- | --- |
|  | **Done by: Sailaubekova Zhansaya** |
|  | **Checked by: Auezova Anel** |

**Almaty 2015**

**SURF feature**

#include<iostream>

#include <opencv2/core/core.hpp>

#include "opencv2/imgproc/imgproc.hpp"

#include "opencv2/calib3d/calib3d.hpp"

#include "opencv2/imgproc/types\_c.h"

#include <opencv2/highgui/highgui.hpp>

#include "opencv2/features2d/features2d.hpp"

//#include "opencv2/xfeatures2d/nonfree.hpp"

//#include "opencv2/nonfree.hpp"

using namespace cv;

using namespace std;

int main()

{

Mat image1;

Mat image = imread("Users/Zhansaya/Decktop/fruits.jpg");

cvtColor(image, image, COLOR\_RGB2GRAY);

image1 = image;

//imshow("Original image", image);

Mat featureImage;

// vector of keypoints

vector<KeyPoint> keypoints;

// Construct the SURF feature detector object

SurfFeatureDetector surf(2500); // threshold

// Detect the SURF features

surf.detect(image,keypoints);

// Draw the keypoints with scale and orientation information

drawKeypoints(image, // original image

keypoints, // vector of keypoints

featureImage, // the resulting image

Scalar(255,255,255), // color of the points

DrawMatchesFlags::DRAW\_RICH\_KEYPOINTS); //flag

imshow("SURF feature scale 1", featureImage);

image = image1;

keypoints.clear();

surf.detect(image,keypoints);

drawKeypoints(image, // original image

keypoints, // vector of keypoints

featureImage, // the resulting image

Scalar(255,255,255), // color of the points

DrawMatchesFlags::DRAW\_RICH\_KEYPOINTS); //flag

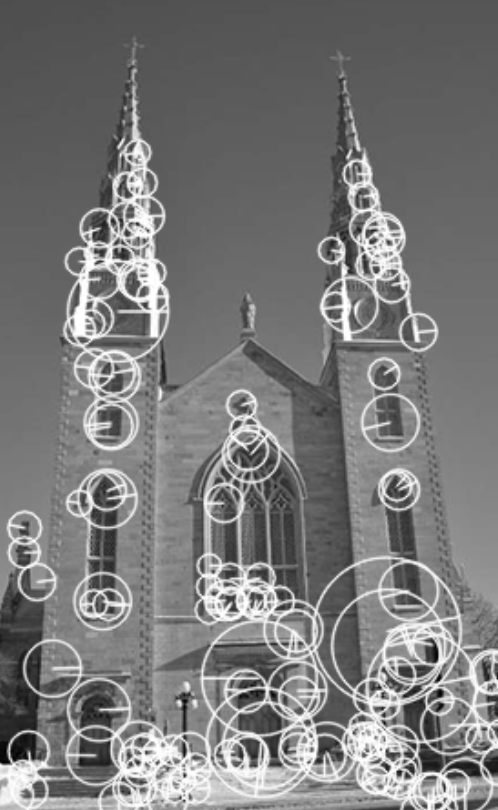
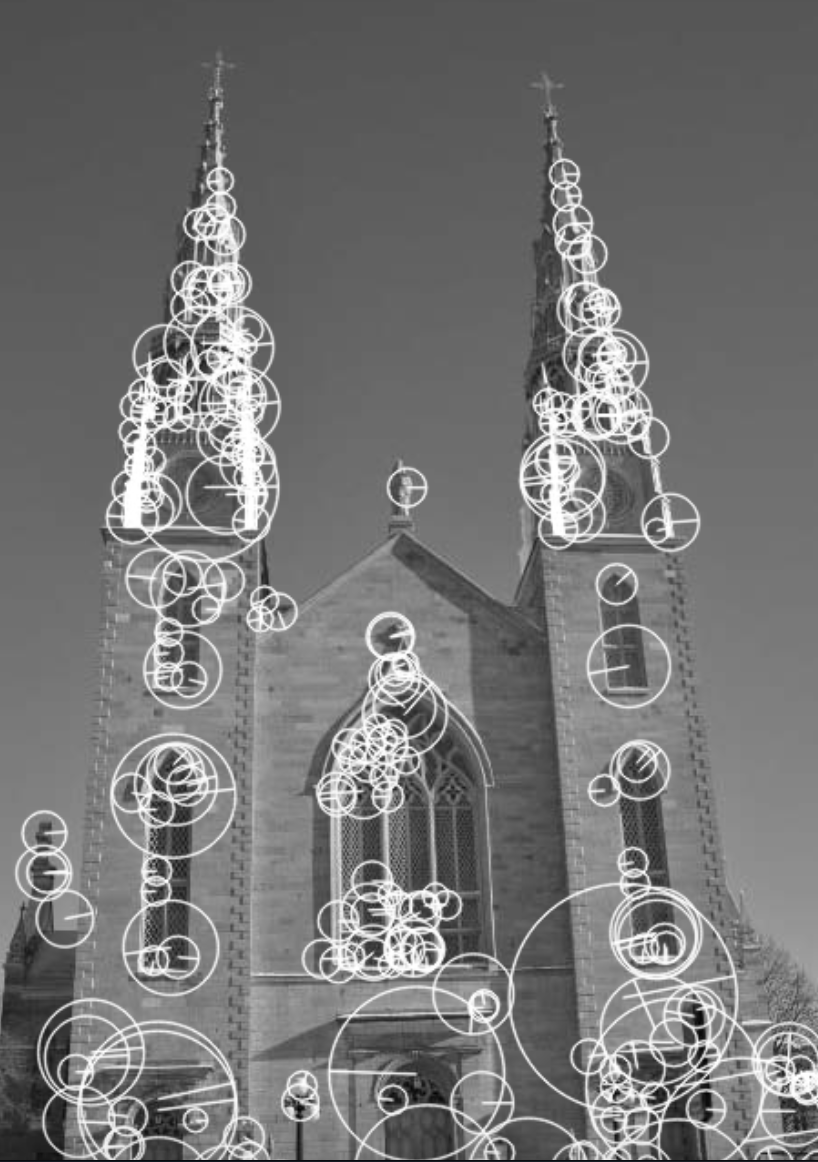
imshow("SURF feature scale 2", featureImage);

waitKey(0);

return 0;

}

**Result:**

**Camera calibration**

**#include "class.h"**

**#include <vector>**

**#include <iostream>**

**#include <opencv2/core/core.hpp>**

**#include "opencv2/imgproc/imgproc.hpp"**

**#include "opencv2/calib3d/calib3d.hpp"**

**#include "opencv2/imgproc/types\_c.h"**

**#include <opencv2/highgui/highgui.hpp>**

**using namespace cv;**

**using namespace std;**

**int main()**

**{**

**Mat image = imread("Users/Zhansaya/Decktop/fruits.jpg",**

**CV\_LOAD\_IMAGE\_GRAYSCALE);**

**imshow("Original image", image);**

**Size boardSize(6,4);**

**vector<string> files;**

**files.push\_back("Users/Zhansaya/Decktop/fruits.jpg");**

**CameraCalibrator cam;**

**if (cam.addChessboardPoints(files, boardSize)) {**

**cam.calibrate(image.size());**

**Mat res = cam.remap(image);**

**imshow("Colibrate", res);**

**}**

**else printf("FAIL");**

**waitKey(0);**

**return 0;**

**}**

**class.h**

#include <vector>

#include <iostream>

#include <opencv2/core/core.hpp>

#include "opencv2/imgproc/imgproc.hpp"

#include "opencv2/calib3d/calib3d.hpp"

#include <opencv2/highgui/highgui.hpp>

class CameraCalibrator {

private:

// input points:

// the points in world coordinates

std::vector<std::vector<cv::Point3f>> objectPoints;

// the point positions in pixels

std::vector<std::vector<cv::Point2f>> imagePoints;

// output Matrices

cv::Mat cameraMatrix;

cv::Mat distCoeffs;

// flag to specify how calibration is done

int flag;

// used in image undistortion

cv::Mat map1,map2;

bool mustInitUndistort;

public:

//~CameraCalibrator(void);

CameraCalibrator() : flag(0), mustInitUndistort(true) {};

// Open chessboard images and extract corner points

int addChessboardPoints(const std::vector<std::string>& filelist, cv::Size & boardSize) {

// the points on the chessboard

std::vector<cv::Point2f> imageCorners;

std::vector<cv::Point3f> objectCorners;

// 3D Scene Points:

// Initialize the chessboard corners

// in the chessboard reference frame

// The corners are at 3D location (X,Y,Z)= (i,j,0)

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

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

objectCorners.push\_back(cv::Point3f(i, j, 0.0f));

}

}

// 2D Image points:

cv::Mat image; // to contain chessboard image

int successes = 0;

// for all viewpoints

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

// Open the image

image = cv::imread(filelist[i],0);

// Get the chessboard corners

bool found = cv::findChessboardCorners(image, boardSize, imageCorners);

// Get subpixel accuracy on the corners

cv::cornerSubPix(image, imageCorners,cv::Size(5,5),cv::Size(-1,-1),cv::TermCriteria(cv::TermCriteria::MAX\_ITER +cv::TermCriteria::EPS,

30, // max number of iterations

0.1)); // min accuracy

//If we have a good board, add it to our data

if (imageCorners.size() == boardSize.area()) {

// Add image and scene points from one view

addPoints(imageCorners, objectCorners);

successes++;

}

}

return successes;

}

// Add scene points and corresponding image points

void addPoints(const std::vector<cv::Point2f>&

imageCorners, const std::vector<cv::Point3f>& objectCorners) {

// 2D image points from one view

imagePoints.push\_back(imageCorners);

// corresponding 3D scene points

objectPoints.push\_back(objectCorners);

}

// Calibrate the camera

// returns the re-projection error

double calibrate(cv::Size &imageSize)

{

// undistorter must be reinitialized

mustInitUndistort= true;

//Output rotations and translations

std::vector<cv::Mat> rvecs, tvecs;

// start calibration

return calibrateCamera(objectPoints, // the 3D points

imagePoints, // the image points

imageSize, // image size

cameraMatrix, // output camera matrix

distCoeffs, // output distortion matrix

rvecs, tvecs, // Rs, Ts

flag); // set options

}

// remove distortion in an image (after calibration)

cv::Mat remap(const cv::Mat &image) {

cv::Mat undistorted;

if (mustInitUndistort) { // called once per calibration

cv::initUndistortRectifyMap(cameraMatrix, // computed camera matrix

distCoeffs, // computed distortion matrix

cv::Mat(), // optional rectification (none)

cv::Mat(), // camera matrix to generate undistorted

image.size(), // size of undistorted

CV\_32FC1, // type of output map

map1, map2); // the x and y mapping functions

mustInitUndistort= false;

}

// Apply mapping functions

cv::remap(image, undistorted, map1, map2,

cv::INTER\_LINEAR); // interpolation type

return undistorted;

}

};

**Result:**

