**Listing {1}** *main.cpp* ( kalibracja)

#include "StereoCalibration.h"

#include "FilterCalibration.h"

#include "opencv2\opencv.hpp"

//sprawdzenie czy liczba zawiera sie w przedziale

bool checkRange(int number, int min, int max)

{

if (number >= min && number <= max)

return true;

else

{

cout << "Liczba musi sie zawierac w przedziale: <" << min << ";" << max << ">\n";

return false;

}

}

//pobiera ciąg znaków z konsoli, sprawdza czy jest liczba i zawiera sie w przedziale

int inputNumber(int minNumber, int maxNumber)

{

int number;

do

{

cin >> number;

while (cin.fail())

{

cin.clear();

cin.ignore(numeric\_limits<std::streamsize>::max(), '\n');

cout << "Zla liczba, wprowadz ponownie: ";

cin >> number;

}

} while (!checkRange(number, minNumber, maxNumber));

return number;

}

int main()

{

int choice = 0;

cout << "PROGRAM KALIBRUJACY SYSTEM WIZYJNY\n 1) Kalibracja kamer\n 2) Ustawienie parametrow filtrowania\n";

choice = inputNumber(1,2);

if (choice == 1)

{

cv::Size chessBoardSize;

int squareSize, minSamples, leftCamID, rightCamID;

cout << "Podaj szerokosc szachownicy: ";

chessBoardSize.width = inputNumber(1, 100);

cout << "Podaj wysokosc szachownicy: ";

chessBoardSize.height = inputNumber(1, 100);

cout << "Podaj dlugosc boku pojedynczego pola szachownicy [mm]: ";

squareSize = inputNumber(1, 100);

cout << "Podaj wymagana liczbe probek: ";

minSamples = inputNumber(5, 50);

CStereoCalibration stereoCalib(chessBoardSize, squareSize, minSamples);

do

{

cout << "Podaj ID lewej kamery: ";

leftCamID = inputNumber(0, 10);

cout << "Podaj ID prawej kamery: ";

rightCamID = inputNumber(0, 10);

} while (stereoCalib.openCameras(leftCamID, rightCamID) == 0);

cout << "Uruchamiana jest procedura kalibracji\n";

stereoCalib.runStereoCalibration();

cout << "Zapisac parametry do pliku? ( 1-TAK, 2-NIE)";

choice = inputNumber(1, 2);

if (choice == 1)

stereoCalib.saveSettings("calibrationParameters.xml");

stereoCalib.closeCameras();

}

else if(choice == 2)

{

CFilterCalibration filterCalib;

int method, camID;

do

{

cout << "Podaj ID kamery: ";

camID = inputNumber(0, 10);

} while (filterCalib.openCamera(camID) == 0);

cout << "Podaj metode filtrowania:\n 1) RGB\n 2) HSV";

method = inputNumber(1, 2);

filterCalib.runFilterCalibration(method);

cout << "Zapisac parametry do pliku? ( 1-TAK, 2-NIE)";

choice = inputNumber(1, 2);

if (choice == 1)

filterCalib.saveFilterParameters("filterParameters.xml", method);

filterCalib.closeCamera();

}

return 1;

}

**Listing {2}** *StereoCalibration.h*

#pragma once

#include "opencv2\opencv.hpp"

#include <time.h>

#include <vector>

#include <iostream>

using namespace std;

class CStereoCalibration

{

public:

CStereoCalibration();

CStereoCalibration(cv::Size chessboardSize, int squareSize, int samplesRequired);

~CStereoCalibration();

vector<vector<cv::Point3f>> calcObjectPoints(int imagesNumber);

int getCalibImagePoints(vector<cv::Mat>& frames, int delay);

int openCameras(int leftCamID, int rightCamID);

int closeCameras();

void saveSettings(char\* path);

void showImage(cv::Mat image, bool waitForKey);

void showImage(char\* windowName, cv::Mat image, bool waitForKey);

int runStereoCalibration();

inline void timerStart() { timer = (double)cv::getTickCount(); };

inline double timerElapsed() { return ((double)cv::getTickCount() - timer) / cv::getTickFrequency(); };

bool camsOpened;

int samplesRequired;

int samples;

double timer;

double error\_rms;

cv::VideoCapture leftCam, rightCam;

vector<vector<cv::Point2f>> leftImagePoints, rightImagePoints;

cv::Mat leftCameraMat, leftCameraDistorsion, rightCameraMat, rightCameraDistorsion;

cv::Mat rotationMat, translationMat, essentialMat, fundamentalMat,

leftRectificationMat, leftProjectionMat,

rightRectificationMat, rightProjectionMat;

cv::Mat disparityToDepthMat;

cv::Mat leftFrame, rightFrame;

cv::Rect leftValidPixROI, rightValidPixROI;

cv::Size imageSize;

cv::Size chessboardSize;

int squareSize;

};

**Listing {3}** *StereoCalibration.cpp*

#include "StereoCalibration.h"

using namespace cv;

CStereoCalibration::CStereoCalibration()

{

chessboardSize.width = 9;

chessboardSize.height = 6;

squareSize = 25;

timer = 0;

samplesRequired = 20;

samples = 0;

}

CStereoCalibration::CStereoCalibration(Size ChessboardSize, int SquareSize, int SamplesRequired)

{

chessboardSize = ChessboardSize;

squareSize = SquareSize;

timer = 0;

samplesRequired = SamplesRequired;

samples = 0;

}

CStereoCalibration::~CStereoCalibration()

{

if (camsOpened)

closeCameras();

}

vector<vector<Point3f>> CStereoCalibration::calcObjectPoints(int imagesNumber)

{

vector<vector<Point3f>> objectPoints;

objectPoints.resize(imagesNumber);

// zalozenie: wszystkie pola w osi Z = 0;

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

{

for (int j = 0; j < chessboardSize.height; j++)

{

for (int k = 0; k < chessboardSize.width; k++)

objectPoints[i].push\_back(Point3f(float(k\*squareSize), float(j\*squareSize), 0));

}

}

return objectPoints;

}

int CStereoCalibration::getCalibImagePoints(vector<Mat>& frames, int delay = 4)

{

vector<Point2f>leftImagePointsBuffer, rightImagePointsBuffer;

bool leftFound = false, rightFound = false;

leftImagePointsBuffer.resize(chessboardSize.height\*chessboardSize.width);

rightImagePointsBuffer.resize(chessboardSize.height\*chessboardSize.width);

//do petli for mozna wprowadzic vector z wieksza iloscia klatek, np. wczytane obrazy z dysku

//naprzemian lewy i prawy obraz

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

{

leftFound = findChessboardCorners(frames[i], chessboardSize, leftImagePointsBuffer,

CV\_CALIB\_CB\_ADAPTIVE\_THRESH | CV\_CALIB\_CB\_NORMALIZE\_IMAGE | CV\_CALIB\_CB\_FAST\_CHECK); //CV\_CALIB\_CB\_FILTER\_QUADS

drawChessboardCorners(frames[i], chessboardSize, leftImagePointsBuffer, leftFound);

showImage("leftCam", frames[i], false);

rightFound = findChessboardCorners(frames[++i], chessboardSize, rightImagePointsBuffer,

CV\_CALIB\_CB\_ADAPTIVE\_THRESH | CV\_CALIB\_CB\_NORMALIZE\_IMAGE | CV\_CALIB\_CB\_FAST\_CHECK);

drawChessboardCorners(frames[i], chessboardSize, rightImagePointsBuffer, rightFound);

showImage("rightCam", frames[i], false);

if (frames[i - 1].size() != frames[i].size())

{

cout << "Rozne rozmiary obrazow!" << endl;

return 0;

}

if (rightFound && leftFound)

{

//odstep czasowy pomiedzy pobraniem probek

if (timerElapsed() >= delay || timer == 0)

{

samples++;

leftImagePoints.push\_back(leftImagePointsBuffer);

rightImagePoints.push\_back(rightImagePointsBuffer);

std::cout << "PROBKI: " << samples << endl;

timerStart();

}

}

leftImagePointsBuffer.clear();

rightImagePointsBuffer.clear();

}

return 1;

}

int CStereoCalibration::openCameras(int leftCamID, int rightCamID)

{

leftCam.open(leftCamID);

if (!leftCam.isOpened())

{

cout << "Nie mozna uruchomic kamery ID:" << leftCamID << endl;

return 0;

}

rightCam.open(rightCamID);

if (!rightCam.isOpened())

{

cout << "Nie mozna uruchomic kamery ID:" << rightCamID << endl;

return 0;

}

camsOpened = true;

return 1;

}

int CStereoCalibration::closeCameras()

{

if (leftCam.isOpened())

leftCam.release();

if (rightCam.isOpened())

rightCam.release();

camsOpened = false;

return 1;

}

void CStereoCalibration::saveSettings(char\* path)

{

FileStorage fileStream;

time\_t actualTime;

fileStream.open(path, FileStorage::WRITE);

time(&actualTime);

fileStream << "calibrationDate" << asctime(localtime(&actualTime));

fileStream << "leftCameraMat" << leftCameraMat;

fileStream << "leftCameraDistorsion" << leftCameraDistorsion;

fileStream << "rightCameraMat" << rightCameraMat;

fileStream << "rightCameraDistorsion" << rightCameraDistorsion;

fileStream << "rotationMat" << rotationMat;

fileStream << "translationMat" << translationMat;

fileStream << "leftRectificationMat" << leftRectificationMat;

fileStream << "leftProjectionMat" << leftProjectionMat;

fileStream << "rightRectificationMat" << rightRectificationMat;

fileStream << "rightProjectionMat" << rightProjectionMat;

fileStream << "imageSize" << imageSize;

fileStream << "errorRMS" << error\_rms;

fileStream.release();

}

void CStereoCalibration::showImage(Mat image, bool waitForKey = false)

{

namedWindow("window");

imshow("window", image);

if (waitForKey)

waitKey();

destroyWindow("window");

}

void CStereoCalibration::showImage(char\* windowName, Mat image, bool waitForKey = false)

{

imshow(windowName, image);

if (waitForKey)

waitKey();

}

int CStereoCalibration::runStereoCalibration()

{

if (!camsOpened)

return 0;

namedWindow("leftCam");

namedWindow("rightCam");

vector<Mat> frames(2);

while (samples < samplesRequired)

{

waitKey(1); // inaczej nie wyswietla podgladu

leftCam >> frames[0];

rightCam >> frames[1];

getCalibImagePoints(frames, 5); // 5s pomiedzy probkami

}

imageSize = frames[0].size();

vector<vector<Point3f>> objectPoints = calcObjectPoints(samples);

leftCameraMat = initCameraMatrix2D(objectPoints, leftImagePoints, imageSize, 0);

rightCameraMat = initCameraMatrix2D(objectPoints, rightImagePoints, imageSize, 0);

double timerCalibrate = (double)getTickCount();

error\_rms = stereoCalibrate(objectPoints, leftImagePoints, rightImagePoints,

leftCameraMat, leftCameraDistorsion, rightCameraMat, rightCameraDistorsion,

imageSize, rotationMat, translationMat, essentialMat, fundamentalMat,

CALIB\_ZERO\_TANGENT\_DIST +

CALIB\_FIX\_FOCAL\_LENGTH +

CALIB\_FIX\_ASPECT\_RATIO +

CALIB\_SAME\_FOCAL\_LENGTH,

TermCriteria(TermCriteria::COUNT + TermCriteria::EPS, 100, 1e-5));

stereoRectify(leftCameraMat, leftCameraDistorsion, rightCameraMat, rightCameraDistorsion,

imageSize, rotationMat, translationMat,

leftRectificationMat, rightRectificationMat,

leftProjectionMat, rightProjectionMat, disparityToDepthMat, 0, -1, imageSize, &leftValidPixROI, &rightValidPixROI);

std::cout << "ZAKONCZONO KALIBRACJE!\nBLAD RMS = " << error\_rms << endl;

std::cout << "CZAS KALIBRACJI DLA " << samplesRequired << " PROBEK WYNIOSL: " <<

((double)cv::getTickCount() - timerCalibrate) / cv::getTickFrequency() << endl << endl;

return 1;

}

**Listing {4}** *FilterCalibration.h*

#pragma once

#include "opencv2\opencv.hpp"

#include <time.h>

#include <iostream>

#define RGB 1

#define HSV 2

using namespace std;

class CFilterCalibration

{

public:

CFilterCalibration();

~CFilterCalibration();

int openCamera(int camID);

int closeCamera();

void saveFilterParameters(char\* path, int method);

int runFilterCalibration(int method);

bool camOpened;

int min[3];

int max[3];

cv::VideoCapture cam;

};

**Listing {5}** *FilterCalibration.cpp*

#include "FilterCalibration.h"

using namespace cv;

CFilterCalibration::CFilterCalibration()

{

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

{

min[i] = 0;

max[i] = 255;

}

}

CFilterCalibration::~CFilterCalibration()

{

}

int CFilterCalibration::openCamera(int camID)

{

cam.open(camID);

if (!cam.isOpened())

{

cout << "Nie mozna uruchomic kamery ID:" << camID << endl;

return 0;

}

camOpened = true;

return 1;

}

int CFilterCalibration::closeCamera()

{

if (cam.isOpened())

cam.release();

camOpened = false;

return 1;

}

void CFilterCalibration::saveFilterParameters(char\* path, int method)

{

FileStorage fileStream;

time\_t actualTime;

string minsString = "min";

string maxsString = "max";

fileStream.open(path, FileStorage::WRITE);

time(&actualTime);

fileStream << "Date" << asctime(localtime(&actualTime));

if (method == RGB)

fileStream << "method" << RGB;

else if (method == HSV)

fileStream << "method" << HSV;

else

fileStream << "method" << "unknown";

fileStream << "min1" << min[0];

fileStream << "min2" << min[1];

fileStream << "min3" << min[2];

fileStream << "max1" << max[0];

fileStream << "max2" << max[1];

fileStream << "max3" << max[2];

}

int CFilterCalibration::runFilterCalibration(int method)

{

if (!camOpened)

return -1;

Mat frame, filteredFrame;

namedWindow("cam");

namedWindow("filtered");

String trackbarNames[6];

if (method == RGB)

{

trackbarNames[0] = "Bmin";

trackbarNames[1] = "Gmin";

trackbarNames[2] = "Rmin";

trackbarNames[3] = "Bmax";

trackbarNames[4] = "Gmax";

trackbarNames[5] = "Rmax";

}

else if (method == HSV)

{

trackbarNames[0] = "Hmin";

trackbarNames[1] = "Smin";

trackbarNames[2] = "Vmin";

trackbarNames[3] = "Hmax";

trackbarNames[4] = "Smax";

trackbarNames[5] = "Vmax";

}

else

return 0;

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

{

createTrackbar(trackbarNames[i], "filtered", &min[i], 255);

createTrackbar(trackbarNames[i + 3], "filtered", &max[i], 255);

}

while (waitKey(5) == -1)

{

cam >> frame;

imshow("cam", frame);

if (method == HSV)

cvtColor(frame, frame, CV\_BGR2HSV);

inRange(frame, Scalar(min[0], min[1], min[2]), Scalar(max[0], max[1], max[2]), filteredFrame);

imshow("filtered", filteredFrame);

}

return 1;

}

**Listing {6}** *main.cpp* ( wykonawczy)

#define NOMINMAX

#include <iostream>

#include <fstream>

#include <string>

#include <opencv2\opencv.hpp>

#include "StereoVision.h"

#include "TCPConnection.h"

#define KAWASAKI\_ADDRESS "11.12.1.30" // przydatne dla stalego adresu IP robota

#define KAWASAKI\_PORT "9001" // przydatne dla stalego portu nasluchiwania robota

#define MIN\_POINTS 3 // min. liczba probek do sredniej

#define FIRST\_POINTS\_IGNORE 2 // liczba pierwszych punktow do ignorowania

using namespace cv;

using namespace std;

float getPixelValue(Mat& img, int x, int y)

{

float\* ptr = img.ptr<float>(x-1);

return ptr[y-1];

}

void saveToFile(ofstream& file, Point3f& point)

{

if (file.is\_open())

{

file << point.x << ";"

<< point.y << ";"

<< point.z << "\n";

}

}

int loadCordTransformation(char\* path, Point3f &trans, Point3f &rot)

{

FileStorage fileStream;

fileStream.open(path, FileStorage::READ);

if (!fileStream.isOpened())

{

std::cout << "Nie udalo sie otworzyc pliku z transformacja ukladu wspolrzednych" << std::endl;

return 0;

}

fileStream["translationX"] >> trans.x;

fileStream["translationY"] >> trans.y;

fileStream["translationZ"] >> trans.z;

fileStream["rotationX"] >> rot.x;

fileStream["rotationY"] >> rot.y;

fileStream["rotationZ"] >> rot.z;

fileStream.release();

return 1;

}

Point3f averagePoints(vector<Point3f>& pointsVec)

{

Point3f average;

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

{

average += pointsVec[i];

}

average.x /= pointsVec.size();

average.y /= pointsVec.size();

average.z /= pointsVec.size();

return average;

}

bool checkRange(int number, int min, int max)

{

if (number >= min && number <= max)

return true;

else

{

cout << "Liczba musi sie zawierac w przedziale: <" << min << ";" << max << ">\n";

return false;

}

}

int inputNumber(int minNumber, int maxNumber)

{

int number;

do

{

cin >> number;

while (cin.fail())

{

cin.clear();

cin.ignore(numeric\_limits<std::streamsize>::max(), '\n');

cout << "Zla liczba, wprowadz ponownie: ";

cin >> number;

}

} while (!checkRange(number, minNumber, maxNumber));

return number;

}

int main()

{

/\* //obiekt obslugujacy zapis do pliku - przydatne przy diagnozowaniu/testowaniu

ofstream plik;

plik.open("nazwa\_pliku.txt", std::ios::out);

\*/

CStereoVision stereoVision;

Mat detectedPoint4D;

Point3f detectedPoint3D, coordsTrans, coordsRot;

CTCPConnection robotConnection;

vector<Point3f> points;

int firstPointsToIgnore = 0;

int leftID, rightID;

string robotAddress, robotPort;

namedWindow("leftCam");

namedWindow("rightCam");

do

{

cout << "Podaj ID lewej kamery: ";

leftID = inputNumber(0, 10);

cout << "Podaj ID prawej kamery: ";

rightID = inputNumber(0, 10);

} while (stereoVision.initStereoVision("calibrationParameters.xml", "filterParameters.xml", leftID, rightID) != 1);

if (!loadCordTransformation("coordinateTransformation.xml", coordsTrans, coordsRot))

return 0;

cout << "Wczytano dane z plikow\n";

//opcja podawania za kazdym razem adresu IP i portu robota

do

{

cout << "Podaj adres IP robota: ";

cin >> robotAddress;

cout << "Podaj port robota: ";

cin >> robotPort;

} while (!robotConnection.setupConnection(robotAddress.c\_str(), robotPort.c\_str()));

/\* //opcja do stalego adresu IP i portu robota

if (!robotConnection.setupConnection(KAWASAKI\_ADDRESS, KAWASAKI\_PORT))

return 0;

\*/

cout << "Polaczenie z robotem ustabilizowane\n";

//sprawdzenie jakosci rektyfikacji

while ((waitKey(5) == -1))

{

stereoVision.grabFrames();

stereoVision.undistortRectifyFrames(stereoVision.leftFrame, stereoVision.rightFrame);

stereoVision.drawParallelLines(stereoVision.leftTransformedFrame);

stereoVision.drawParallelLines(stereoVision.rightTransformedFrame);

imshow("leftCam", stereoVision.leftTransformedFrame);

imshow("rightCam", stereoVision.rightTransformedFrame);

}

while ((waitKey(5) == -1))

{

stereoVision.grabFrames();

stereoVision.undistortRectifyFrames(stereoVision.leftFrame, stereoVision.rightFrame);

stereoVision.filterFrames(stereoVision.leftTransformedFrame, stereoVision.rightTransformedFrame, stereoVision.filterMethod);

imshow("leftCam", stereoVision.leftFilteredFrame);

imshow("rightCam", stereoVision.rightFilteredFrame);

detectedPoint3D = stereoVision.triangulate(stereoVision.leftFilteredFrame, stereoVision.rightFilteredFrame);

// jesli nie wykryto punktu

if (detectedPoint3D == Point3f(0, 0, 0))

{

points.clear(); // wyczysz bufor do usrednienia probek

firstPointsToIgnore = 0; // ponownie ignoruj pierwsze punkty

continue;

}

// saveToFile(plik, detectedPoint3D); //zapis punktu polozenia do pliku

//pierwsze wykryte punkty sa ignorowane

if (firstPointsToIgnore < FIRST\_POINTS\_IGNORE)

{

firstPointsToIgnore++;

continue;

}

points.push\_back(detectedPoint3D); // dodaj punkt do bufora punktow

// jesli jest wystarczajaca liczba probek i jest polaczenie...

if (points.size() >= MIN\_POINTS && robotConnection.isConnected())

{

Point3f pointToSend = averagePoints(points); // srednia z bufora probek

cout << "KAMERY: " << pointToSend << endl; // wydrukuj w konsoli polozenie punktu wzgl. kamer

pointToSend = stereoVision.coordinateTransform(pointToSend, coordsTrans, coordsRot); // punkt w odniesieniu do ukl. robota

cout << "ROBOT: " << pointToSend << endl; // wydrukuj w konsoli polozenie punktu wzgl. robota

points.clear(); // wyczysc bufor

std::string dataToSend = std::to\_string(pointToSend.x) + ";" +

std::to\_string(pointToSend.y) + ";" +

std::to\_string(pointToSend.z) + ";"; // mozna wprowadzic staly offset, np. dla osi Z

// wyslij string dataToSend zamieniony na const char

if (robotConnection.sendData(dataToSend.c\_str()) != 0)

{

cout << "WYSLANO:\n" << dataToSend.c\_str() << endl;

}

}

}

//plik.close(); // zamkniecie pliku

return 1;

}

**Listing {7}** *StereoVision.h*

#pragma once

#include "opencv2\opencv.hpp"

#include <math.h>

#define PI 3.14159265358979323846

#define RGB 1

#define HSV 2

class CStereoVision

{

public:

CStereoVision();

~CStereoVision();

int initStereoVision(char\* path, char\* filterParamsPath, int leftCamID, int rightCamID);

int loadSettings(char\* path);

int loadFilter(char\* path);

int openCameras(int leftCamID, int rightCamID);

int closeCameras();

int grabFrames();

void filterFrames(cv::Mat& left, cv::Mat& right, int method);

int undistortRectifyFrames(cv::Mat &leftFrame, cv::Mat &rightFrame);

void showImage(cv::Mat image, bool waitForKey);

void showImage(char\* windowName, cv::Mat image, bool waitForKey);

void drawParallelLines(cv::Mat &image);

cv::Point2f findPoint(cv::Mat& img);

float getPixelValue(cv::Mat& img, int x, int y);

cv::Point3f calcPoint3D(cv::Mat& point4D);

cv::Point3f triangulate(cv::Mat& leftImg, cv::Mat& rightImg);

cv::Point3f coordinateTransform(cv::Point3f point, cv::Point3f trans, cv::Point3f rot);

bool settingsLoaded;

bool camsOpened;

int filterMethod;

int filterMins[3];

int filterMaxs[3];

cv::VideoCapture leftCam, rightCam;

cv::Mat leftCameraMat, leftCameraDistorsion, rightCameraMat, rightCameraDistorsion;

cv::Mat leftRectificationMat, leftProjectionMat,

rightRectificationMat, rightProjectionMat;

cv::Mat leftFrame, rightFrame;

cv::Mat leftFilteredFrame, rightFilteredFrame;

cv::Mat leftTransformedFrame, rightTransformedFrame;

cv::Size imageSize;

cv::Point3f coordnateTrans;

cv::Point3f coordnateRot;

};

**Listing {8}** *StereoVision.cpp*

#include "StereoVision.h"

using namespace cv;

CStereoVision::CStereoVision()

{

camsOpened = false;

settingsLoaded = false;

}

CStereoVision::~CStereoVision()

{

if (camsOpened)

closeCameras();

}

int CStereoVision::initStereoVision(char\* settingsPath, char\* filterParamsPath, int leftCamID, int rightCamID)

{

if (loadSettings(settingsPath) != 1)

return 0;

if (loadFilter(filterParamsPath) != 1)

return 0;

if (openCameras(leftCamID, rightCamID) != 1)

return 0;

else

camsOpened = true;

settingsLoaded = true;

return 1;

}

int CStereoVision::loadSettings(char\* path)

{

FileStorage fileStream;

fileStream.open(path, FileStorage::READ);

if (!fileStream.isOpened())

{

std::cout << "Nie udalo sie otworzyc pliku z parametrami kamer" << std::endl;

return 0;

}

fileStream["leftCameraMat"] >> leftCameraMat;

fileStream["leftCameraDistorsion"] >> leftCameraDistorsion;

fileStream["rightCameraMat"] >> rightCameraMat;

fileStream["rightCameraDistorsion"] >> rightCameraDistorsion;

fileStream["leftRectificationMat"] >> leftRectificationMat;

fileStream["leftProjectionMat"] >> leftProjectionMat;

fileStream["rightRectificationMat"] >> rightRectificationMat;

fileStream["rightProjectionMat"] >> rightProjectionMat;

fileStream["imageSize"] >> imageSize;

fileStream.release();

return 1;

}

int CStereoVision::loadFilter(char\* path)

{

FileStorage fileStream;

fileStream.open(path, FileStorage::READ);

if (!fileStream.isOpened())

{

std::cout << "Nie udalo sie otworzyc pliku z parametrami filtrowania" << std::endl;

return 0;

}

fileStream["method"] >> filterMethod;

fileStream["min1"] >> filterMins[0];

fileStream["min2"] >> filterMins[1];

fileStream["min3"] >> filterMins[2];

fileStream["max1"] >> filterMaxs[0];

fileStream["max2"] >> filterMaxs[1];

fileStream["max3"] >> filterMaxs[2];

fileStream.release();

return 1;

}

int CStereoVision::openCameras(int leftCamID, int rightCamID)

{

leftCam.open(leftCamID);

if (!leftCam.isOpened())

{

std::cout << "Nie udalo sie uruchomic kamery: " << leftCamID << std::endl;

return 0;

}

rightCam.open(rightCamID);

if (!rightCam.isOpened())

{

std::cout << "Nie udalo sie uruchomic kamery: " << rightCamID << std::endl;

return 0;

}

return 1;

}

int CStereoVision::closeCameras()

{

if (leftCam.isOpened())

leftCam.release();

if (rightCam.isOpened())

rightCam.release();

camsOpened = false;

return 1;

}

int CStereoVision::grabFrames()

{

if (!camsOpened)

{

std::cout << "Kamery nie sa uruchomione" << std::endl;

return 0;

}

leftCam >> leftFrame;

rightCam >> rightFrame;

return 1;

}

void CStereoVision::filterFrames(cv::Mat& left, cv::Mat& right, int method)

{

if (method == HSV)

{

cvtColor(leftFrame, leftFrame, CV\_BGR2HSV);

cvtColor(rightFrame, rightFrame, CV\_BGR2HSV);

}

else if (method == RGB);

else return;

inRange(leftFrame, Scalar(filterMins[0], filterMins[1], filterMins[2]), Scalar(filterMaxs[0], filterMaxs[1], filterMaxs[2]), leftFilteredFrame);

inRange(rightFrame, Scalar(filterMins[0], filterMins[1], filterMins[2]), Scalar(filterMaxs[0], filterMaxs[1], filterMaxs[2]), rightFilteredFrame);

}

int CStereoVision::undistortRectifyFrames(Mat &leftImage, Mat &rightImage)

{

Mat leftMapX, leftMapY, rightMapX, rightMapY;

initUndistortRectifyMap(leftCameraMat, leftCameraDistorsion, leftRectificationMat, leftProjectionMat, imageSize, CV\_32F, leftMapX, leftMapY);

initUndistortRectifyMap(rightCameraMat, rightCameraDistorsion, rightRectificationMat, rightProjectionMat, imageSize, CV\_32F, rightMapX, rightMapY);

remap(leftImage, leftTransformedFrame, leftMapX, leftMapY, INTER\_LINEAR);

remap(rightImage, rightTransformedFrame, rightMapX, rightMapY, INTER\_LINEAR);

return 1;

}

void CStereoVision::showImage(Mat image, bool waitForKey)

{

namedWindow("window");

imshow("window", image);

if (waitForKey)

waitKey();

destroyWindow("window");

}

void CStereoVision::showImage(char\* windowName, Mat image, bool waitForKey = 0)

{

imshow(windowName, image);

if (waitForKey)

waitKey();

}

void CStereoVision::drawParallelLines(Mat & image)

{

Size imageSize = image.size();

for (int i = 0; i < imageSize.height; i+=32)

{

line(image, Point(0, i), Point(imageSize.width, i), Scalar(0, 255, 0),1);

}

}

Point2f CStereoVision::findPoint(Mat& img)

{

float xMin = img.cols, xMax = 0, yMin = img.rows, yMax = 0;

int counter = 0;

uchar\* pointer;

for (int i = 0; i < img.rows; i++)

{

pointer = img.ptr(i);

for (int j = 0; j < img.cols; j++)

{

if (pointer[j] == 255)

{

counter++;

if (j < xMin)

xMin = j;

if (j > xMax)

xMax = j;

if (i < yMin)

yMin = i;

if (i > yMax)

yMax = i;

}

}

}

if (counter == 1) // jeden punkt odnaleziony

return Point2f(xMax, yMax);

else if (counter == 0) // 0 punktow

return Point2f(0,0);

else // wiele punktow - blop

return Point2f(xMin + (xMax - xMin) / 2, yMin + (yMax - yMin) / 2);

}

Point3f CStereoVision::triangulate(Mat& leftImg, Mat& rightImg)

{

std::vector<Point2f> leftPoint, rightPoint;

Point2f left, right;

Point3f point3D;

Mat point4D = Mat(4, 1, CV\_32F);

left = findPoint(leftImg);

right = findPoint(rightImg);

if (left == Point2f(0, 0) || right == Point2f(0, 0)) // czyli brak punktow / (0,0)

point3D = Point3f(0, 0, 0);

else

{

leftPoint.push\_back(left);

rightPoint.push\_back(right);

triangulatePoints(leftProjectionMat, rightProjectionMat,

leftPoint, rightPoint, point4D);

point3D = calcPoint3D(point4D);

}

return point3D;

}

float CStereoVision::getPixelValue(Mat& img, int x, int y)

{

float\* ptr = img.ptr<float>(x - 1);

return ptr[y - 1];

}

Point3f CStereoVision::calcPoint3D(Mat& point4D)

{

Point3f point3D;

if (getPixelValue(point4D, 3, 1) == 0)

point3D = Point3f(0, 0, 0);

else

{

float w = getPixelValue(point4D, 4, 1);

point3D.x = getPixelValue(point4D, 1, 1) / w;

point3D.y = getPixelValue(point4D, 2, 1) / w;

point3D.z = getPixelValue(point4D, 3, 1) / w;

}

return point3D;

}

Point3f CStereoVision::coordinateTransform(Point3f point, Point3f trans, Point3f rot)

{

Mat rotXMat = Mat::eye(4, 4, CV\_32F);

Mat rotYMat = Mat::eye(4, 4, CV\_32F);

Mat rotZMat = Mat::eye(4, 4, CV\_32F);

Mat transMat = Mat::eye(4, 4, CV\_32F);

Mat invRotXMat, invRotYMat, invRotZMat, invTransMat;

Mat cameraPoint = Mat(point);

cameraPoint.resize(4);

cameraPoint.at<float>(3, 0) = 1;

transMat.at<float>(0, 3) = trans.x;

transMat.at<float>(1, 3) = trans.y;

transMat.at<float>(2, 3) = trans.z;

rotXMat.at<float>(1, 1) = (float)cos(rot.x \* PI / 180);

rotXMat.at<float>(1, 2) = (float)-sin(rot.x \* PI / 180);

rotXMat.at<float>(2, 1) = (float)sin(rot.x \* PI / 180);

rotXMat.at<float>(2, 2) = (float)cos(rot.x \* PI / 180);

rotYMat.at<float>(0, 0) = (float)cos(rot.y \* PI / 180);

rotYMat.at<float>(0, 2) = (float)sin(rot.y \* PI / 180);

rotYMat.at<float>(2, 0) = (float)-sin(rot.y \* PI / 180);

rotYMat.at<float>(2, 2) = (float)cos(rot.y \* PI / 180);

rotZMat.at<float>(0, 0) = (float)cos(rot.z \* PI / 180);

rotZMat.at<float>(0, 1) = (float)-sin(rot.z \* PI / 180);

rotZMat.at<float>(1, 0) = (float)sin(rot.z \* PI / 180);

rotZMat.at<float>(1, 1) = (float)cos(rot.z \* PI / 180);

Mat result = transMat \* rotXMat \* rotYMat \* rotZMat \* cameraPoint;

result.resize(3);

return Point3f(result);

}

**Listing {9}** *TCPConnection.h*

#define WIN32\_LEAN\_AND\_MEAN

#pragma once

#include <windows.h>

#include <winsock2.h>

#include <ws2tcpip.h>

#include <stdlib.h>

#include <iostream>

#pragma comment (lib, "Ws2\_32.lib")

#pragma comment (lib, "Mswsock.lib")

#pragma comment (lib, "AdvApi32.lib")

#pragma once

class CTCPConnection

{

public:

CTCPConnection();

~CTCPConnection();

WSADATA wsaData;

SOCKET ConnectSocket;

private:

int actionResult; //dodac obsluge wyjatkow

bool connected;

public:

inline int isConnected() {return connected;};

int setupConnection(const char\* address, const char\* port);

int sendData(const char\* data);

int closeConnection();

};

**Listing {10}** *TCPConnection.cpp*

#include "TCPConnection.h"

CTCPConnection::CTCPConnection()

{

connected = false;

actionResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

}

CTCPConnection::~CTCPConnection()

{

if (connected)

closeConnection();

}

int CTCPConnection::setupConnection(const char\* address, const char\* port)

{

struct addrinfo \*result = NULL,

\*ptr = NULL,

hints;

ZeroMemory(&hints, sizeof(hints));

hints.ai\_family = AF\_UNSPEC;

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_protocol = IPPROTO\_TCP;

actionResult = getaddrinfo(address, port, &hints, &result);

if (actionResult != 0)

{

std::cout << "Wystapil blad podczas uzyskiwania parametrow polaczenia: " << actionResult << std::endl;

WSACleanup();

return 0;

}

for (ptr = result; ptr != NULL; ptr = ptr->ai\_next)

{

ConnectSocket = socket(ptr->ai\_family, ptr->ai\_socktype,

ptr->ai\_protocol);

if (ConnectSocket == INVALID\_SOCKET)

{

std::cout << "Wystapil blad socket'a: " << WSAGetLastError() << std::endl;

WSACleanup();

return 0;

}

actionResult = connect(ConnectSocket, ptr->ai\_addr, (int)ptr->ai\_addrlen);

if (actionResult == SOCKET\_ERROR)

{

closesocket(ConnectSocket);

ConnectSocket = INVALID\_SOCKET;

continue;

}

break;

}

freeaddrinfo(result);

if (ConnectSocket == INVALID\_SOCKET)

{

std::cout << "Nie udalo sie polaczyc - zly socket\n";

WSACleanup();

return 0;

}

connected = true;

return 1;

}

int CTCPConnection::sendData(const char\* data)

{

if (!connected)

{

std::cout << "Najpierw nalezy ustanowic polaczenie!" << std::endl;

return 0;

}

actionResult = send(ConnectSocket, data, (int)strlen(data), 0);

if (actionResult == SOCKET\_ERROR)

{

std::cout << "Wystapil blad przesylania: " << WSAGetLastError() << std::endl;

closesocket(ConnectSocket);

WSACleanup();

return 0;

}

return 1;

}

int CTCPConnection::closeConnection()

{

if (!connected)

{

std::cout << "Najpierw nalezy ustanowic polaczenie!" << std::endl;

return 0;

}

actionResult = shutdown(ConnectSocket, SD\_SEND);

if (actionResult == SOCKET\_ERROR)

{

std::cout << "Wystapil blad zamykania polaczenia: " << WSAGetLastError() << std::endl;

closesocket(ConnectSocket);

WSACleanup();

return 0;

}

connected = false;

return 1;

}

**Listing {11}** Program robota

.PROGRAM d\_tcp()

WHILE TRUE DO

PRINT "Otwieram port TCP"

CALL d\_openport

WHILE NOT SIG(2003) DO

CALL d\_recvdata

END

PRINT "Zamykam port TCP"

CALL d\_closeport

END

.END

.PROGRAM d\_openport()

num = 1

timeout = 20

ret = 0

port = 9001

sock\_id = 0

DO

TCP\_LISTEN ret,port

UNTIL ret==0

DO

TCP\_ACCEPT sock\_id,port,timeout

UNTIL sock\_id>0

PRINT "PORT OTWARTY"

RETURN

.END

.PROGRAM d\_closeport()

TCP\_CLOSE ret,sock\_id

TCP\_END\_LISTEN ret,port

PRINT "PORT ZAMKNIETY"

.END

.PROGRAM d\_recvdata()

IF NOT SIG(2002) THEN

TCP\_RECV ret,sock\_id,$recv[0],num,timeout,255

IF ret<>0 THEN

PRINT "Error: ",ret

ELSE

CALL d\_transc

END

END

RETURN

.END

.PROGRAM d\_transc()

SIGNAL 2002

i = 0

DO

$tmp = $DECODE($recv[0],";",0)

cords[i] = VAL($tmp)

$tmp = $DECODE($recv[0],";",1)

i = i+1

UNTIL $recv[0]==""

PRINT 2: "X: ",cords[0]

PRINT 2: "Y: ",cords[1]

PRINT 2: "Z: ",cords[2]

cords[3] = 37

cords[4] = 178

cords[5] = 66

JMOVE TRANS(cords[0],cords[1],cords[2],cords[3],cords[4],cords[5])

SIGNAL -2002

PRINT 2: "WYKONANO"

RETURN

.END

.PROGRAM d\_repairport()

port = 9001

ret = 0

sock\_id = 0

TCP\_END\_LISTEN ret,port

PRINT "PORT ZAMKNIETY"

.END