#include <iostream>

#include <fstream>

#include <string>

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

//#include <opencv2/imgproc/imgproc.hpp>

//#include "opencv2/imgcodecs.hpp"

#include "opencv2/imgproc.hpp"

#include "opencv2/imgcodecs.hpp"

#include "opencv2/highgui.hpp"

#include <communication/multi\_socket.h>

#include <models/tronis/ImageFrame.h>

#include <grabber/opencv\_tools.hpp>

#include <models/tronis/BoxData.h>

#define \_USE\_MATH\_DEFINES

#include <math.h>

using namespace std;

using namespace cv;

class LaneAssistant

{

// insert your custom functions and algorithms here

public:

LaneAssistant()

{

}

bool processData( tronis::CircularMultiQueuedSocket& socket )

{

// do stuff with data

socket.send( tronis::SocketData( "Ego Fahrzeug Geschwindigkeit >>> " +

to\_string( ego\_velocity\_ ) ) );

// send steering value via socket

getSteeringInput( socket );

// send throttle value via socket

getThrottleInput( socket );

return true;

}

protected:

std::string image\_name\_;

cv::Mat image\_;

tronis::LocationSub ego\_location\_;

tronis::OrientationSub ego\_orientation\_;

double ego\_velocity\_;

// parameters in Aufgabe2

Point ego\_leftS, ego\_leftE; // ego left lane start and end point

Point ego\_rightS, ego\_rightE; // ego right lane start and end point

Point directionS, directionE; // car driving direction

Point mittle\_of\_lane;

Point mittle\_of\_image;

double rows = 512, cols = 720; // original Picture size (720, 512).

// parameters in Aufgabe 3

double steering\_input; // Send the steeering input value to Tronis Socket

double steering\_pc =

0.0001; // PID Controller: partitial when the car direction is

// vert2ical to the lane, the maximum steering input value

double steering\_dc = 0.04; // PID Controller: differential

double steering\_ic = 0.0001;

double Err\_steering;

double dErr\_steering;

double iErr\_steering;

double lastErr\_steering = 0;

// parameters in Aufgabe 4

tronis::ObjectVector Objects\_BBox;

double throttle\_input = 1; // Send the throttle input value to Tronis Socket

double throttle\_pc = 0.5;

double throttle\_dc = -0.002;

double throttle\_ic = -0.02; // PID Controller: intergral

double Err\_velocity;

double lastErr\_velocity = 0;

double dErr\_velocity;

double sumErr\_velocity;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Aufgabe 2: lane detection

vector<Vec4d> setLanes()

{

cv::Mat blur\_img; // Remove noise by blurring with a Gaussian filter ( kernel size = 3 )

GaussianBlur( image\_, blur\_img, Size( 3, 3 ), 0, 0, BORDER\_DEFAULT );

cv::Mat gray\_img; // Convert the image to grayscale

cvtColor( blur\_img, gray\_img, cv::COLOR\_BGR2GRAY );

cv::Mat binary\_img; // transfer the image to binary one

cv::threshold( gray\_img, binary\_img, 120, 255, cv::THRESH\_BINARY );

cv::Mat edge\_img; // Edge detection

Canny( binary\_img, edge\_img, 100, 200 );

// Another way to get the edge image: Sobel Derivitive()

// Mat grad\_x, grad\_y;

// Mat abs\_grad\_x, abs\_grad\_y;

// Sobel( binary\_img, grad\_x, CV\_16S, 1, 0, 1, 1, 0, BORDER\_DEFAULT );

// Sobel( binary\_img, grad\_y, CV\_16S, 0, 1, 1, 1, 0, BORDER\_DEFAULT );

// convertScaleAbs( grad\_x, abs\_grad\_x ); // converting back to CV\_8U

// convertScaleAbs( grad\_y, abs\_grad\_y );

// addWeighted( abs\_grad\_x, 0.5, abs\_grad\_y, 0.5, 0, edge\_img );

// set a polygon mask to only keep thed region of interest

cv::Mat mask = Mat::zeros( image\_.size(), edge\_img.type() );

const int num = 6;

Point points[1][num] = {Point( 0, rows ),

Point( 0, rows \* 0.7 ),

Point( cols \* 0.33, rows \* 0.6 ),

Point( cols \* 0.66, rows \* 0.6 ),

Point( cols, rows \* 0.7 ),

Point( cols, rows )};

const Point\* polygon = points[0];

fillConvexPoly( mask, polygon, num, Scalar( 255 ) );

cv::Mat roi\_img;

cv::bitwise\_and( edge\_img, mask, roi\_img );

imshow( "Canny output: Region of Interest", edge\_img );

vector<Vec4d> raw\_lanes; // will hold all the results of the detection

HoughLinesP( roi\_img, raw\_lanes, 1, CV\_PI / 180, 50, 50,

10 ); // Probabilistic Line Transform

return raw\_lanes;

}

void getLanes( vector<Vec4d> raw\_lanes )

{

vector<Vec4d> left\_lanes, right\_lanes;

Vec4f left\_lane\_function, right\_lane\_function;

vector<Point> left\_points, right\_points;

ego\_leftS.y = 300;

ego\_rightS.y = 300;

ego\_leftE.y = 500;

ego\_rightE.y = 500;

double left\_k, right\_k; // gradient

Point left\_b, right\_b;

for( auto lane : raw\_lanes ) // divide the line set into left and right part based on the

// line center point

{

double lane\_center = ( lane[0] + lane[2] ) / 2;

if( lane\_center < cols / 2 )

{

left\_lanes.push\_back( lane );

}

else

{

right\_lanes.push\_back( lane );

}

}

// get the left lines

for( auto left\_lane : left\_lanes ) // add all the points into a vector

{

left\_points.push\_back( Point( left\_lane[0], left\_lane[1] ) );

left\_points.push\_back( Point( left\_lane[2], left\_lane[3] ) );

}

if( left\_points.size() > 0 ) // fit a line with the method of least square

{

// fitLine(input vector, output line, distance type, distance parameter, radial

// parameter, angle parameter) output (vx, vy, x, y)

cv::fitLine( left\_points, left\_lane\_function, cv::DIST\_L2, 0, 0.01, 0.01 );

left\_k = left\_lane\_function[1] / left\_lane\_function[0];

left\_b = Point( left\_lane\_function[2], left\_lane\_function[3] );

ego\_leftS.x = ( ego\_leftS.y - left\_b.y ) / left\_k + left\_b.x;

ego\_leftE.x = ( ego\_leftE.y - left\_b.y ) / left\_k + left\_b.x;

}

// get the right lines

for( auto right\_lane : right\_lanes )

{

right\_points.push\_back( Point( right\_lane[0], right\_lane[1] ) );

right\_points.push\_back( Point( right\_lane[2], right\_lane[3] ) );

}

if( right\_points.size() > 0 )

{

cv::fitLine( right\_points, right\_lane\_function, cv::DIST\_L2, 0, 0.01, 0.01 );

right\_k = right\_lane\_function[1] / right\_lane\_function[0];

right\_b = Point( right\_lane\_function[2], right\_lane\_function[3] );

ego\_rightS.x = ( ego\_rightS.y - right\_b.y ) / right\_k + right\_b.x;

ego\_rightE.x = ( ego\_rightE.y - right\_b.y ) / right\_k + right\_b.x;

}

// Aufgabe3

directionS = ( ego\_leftS + ego\_rightS ) / 2;

directionE = ( ego\_leftE + ego\_rightE ) / 2;

mittle\_of\_lane.y = directionE.y;

mittle\_of\_lane.x = directionE.x;

mittle\_of\_image.x = cols \* 0.5;

mittle\_of\_image.y = mittle\_of\_lane.y;

// cv::Vec4d direction( directionS.x, directionS.y, directionE.x, directionE.y );

}

void detectLanes( ) // Function to detect lanes based on camera image

// Insert your algorithm here

{

vector<Vec4d> raw\_lanes = setLanes();

// vector<Vec4d> warning\_lanes = setWarnings();

getLanes( raw\_lanes );

// Draw the lane lines and show results

line( image\_, ego\_leftS, ego\_leftE, Scalar( 0, 0, 255 ), 3, LINE\_AA );

line( image\_, ego\_rightS, ego\_rightE, Scalar( 0, 0, 255 ), 3, LINE\_AA );

// Aufagbe3: Draw the driving direction lines and show results

line( image\_, Point( directionS.x, directionS.y ), Point( directionE.x, directionE.y ),

Scalar( 0, 255, 0 ), 3, LINE\_AA );

int radius = 5; // The radius of the point, you can change this to your liking.

Scalar color1( 0, 0, 255 ); // red color.

Scalar color2( 255, 0, 0 );

circle( image\_, mittle\_of\_lane, radius, color1, -1 ); // "-1" indicates to fill the circle.

circle( image\_, mittle\_of\_image, radius, color2, -1 );

// LaneAssistant::pipeline2( image\_ );

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Aufgabe 3: Steering control

void setSteeringInput()

{

/\*if( directionS.x == directionE.x ) // when the car drives straight

{

steering\_input = 0;

}

else

{\*/

/\*double slope = -( directionS.y - directionE.y ) /

( directionS.x - directionE.x ); // positive: up right to down left

double steering\_winkel =

M\_PI\_2 - abs( atan( slope ) ); // 0: vertical, pi/2: horizontal\*/

Err\_steering = mittle\_of\_lane.x - cols \* 0.5;

dErr\_steering = Err\_steering - lastErr\_steering;

iErr\_steering = Err\_steering + lastErr\_steering;

lastErr\_steering = Err\_steering;

steering\_input =

steering\_pc \* Err\_steering + steering\_dc \* dErr\_steering + steering\_ic \* iErr\_steering;

/\*if( slope > 0 ) // driving to the right is positive

{

steering\_input = -( steering\_input );

}\*/

//};

if( steering\_input > 0.09 )

steering\_input = 0.3;

if( steering\_input < -0.03 )

steering\_input = -0.03;

};

void getSteeringInput( tronis::CircularMultiQueuedSocket& socket )

{

setSteeringInput();

string prefix\_steering = "Steering value ";

socket.send( tronis::SocketData( prefix\_steering + to\_string( steering\_input ) ) );

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Aufgabe 4: Throttle control

bool processPoseVelocity( tronis::PoseVelocitySub\* msg )

{

ego\_location\_ = msg->Location;

ego\_orientation\_ = msg->Orientation;

ego\_velocity\_ = msg->Velocity \* 3.6 \* 1e-2; // from cm/s to km/h

return true;

}

bool processObject( tronis::BoxDataSub\* sensorData )

{

// do stuff

Objects\_BBox = sensorData->Objects;

return true;

}

double processDistance( tronis::LocationSub location )

{

float pos\_x = location.X / 100;

float pos\_y = location.Y / 100;

double dist = sqrt( pow( pos\_x, 2 ) + pow( pos\_y, 2 ) );

// float angle = atan( pos\_y / pos\_x ); could be useful for the opposite direction

// car detection

return dist;

}

void setThrottleInput( double dist )

{

Err\_velocity = 60 - ego\_velocity\_; // set the max speed to 60km/h

sumErr\_velocity = lastErr\_velocity + Err\_velocity;

dErr\_velocity = lastErr\_velocity - Err\_velocity;

lastErr\_velocity = Err\_velocity;

double min\_distance = 20;

if( dist < min\_distance ) // in the simulation the speed could reach 58km/h

{

if( abs( ego\_velocity\_ ) < 1 ) // make it absolutely stop

{

throttle\_input = 0;

}

else // urgent stop

{

throttle\_input = -0.5;

}

}

else if( dist < min\_distance + 5 && ego\_velocity\_ > 40 ||

dist < min\_distance + 10 && ego\_velocity\_ > 45 ) // keep the distance

{

throttle\_input = -0.5; // the environment resistance throttle would be -0.5

}

else

{

if( abs( ego\_velocity\_ ) > 20 ) // to keep the cvelocity stable

{

throttle\_input = throttle\_pc \* Err\_velocity + throttle\_dc \* dErr\_velocity +

throttle\_ic \* sumErr\_velocity;

}

else // make it reaccelerate faster after deaccelerating

{

throttle\_input = 1;

}

}

}

void getThrottleInput( tronis::CircularMultiQueuedSocket& socket )

{

string prefix\_throttle = "Throttle value ";

if( Objects\_BBox.size() )

{

for( size\_t i = 0; i < Objects\_BBox.size(); i++ )

{

tronis::ObjectSub& object = Objects\_BBox[i];

if( object.ActorName.Value() == "AutoCar" )

{

double dist = processDistance( object.Pose.Location );

setThrottleInput( dist );

socket.send(

tronis::SocketData( prefix\_throttle + to\_string( throttle\_input ) ) );

cout << "Autocar is in the front !" << endl;

}

else

{

setThrottleInput( 100.0 );

socket.send(

tronis::SocketData( prefix\_throttle + to\_string( throttle\_input ) ) );

cout << "lane or track got detected, please ignore !" << endl;

}

}

}

else

{

// set the initial throttle input in case there are no objects in the front

setThrottleInput( 100.0 );

socket.send( tronis::SocketData( prefix\_throttle + to\_string( throttle\_input ) ) );

cout << "no objects in the front !" << endl;

}

}

// Helper functions, no changes needed

public:

// Function to process received tronis data

bool getData( tronis::ModelDataWrapper data\_model )

{

if( data\_model->GetModelType() == tronis::ModelType::Tronis )

{

std::cout << "\n"

<< "Id: " << data\_model->GetTypeId() << ", Name: " << data\_model->GetName()

<< ", Time: " << data\_model->GetTime() << std::endl;

// if data is sensor output, process data

switch( static\_cast<tronis::TronisDataType>( data\_model->GetDataTypeId() ) )

{

case tronis::TronisDataType::Image:

{

processImage( data\_model->GetName(),

data\_model.get\_typed<tronis::ImageSub>()->Image );

break;

}

case tronis::TronisDataType::ImageFrame:

{

const tronis::ImageFrame& frames(

data\_model.get\_typed<tronis::ImageFrameSub>()->Images );

for( size\_t i = 0; i != frames.numImages(); ++i )

{

std::ostringstream os;

os << data\_model->GetName() << "\_" << i + 1;

processImage( os.str(), frames.image( i ) );

}

break;

}

case tronis::TronisDataType::ImageFramePose:

{

const tronis::ImageFrame& frames(

data\_model.get\_typed<tronis::ImageFramePoseSub>()->Images );

for( size\_t i = 0; i != frames.numImages(); ++i )

{

std::ostringstream os;

os << data\_model->GetName() << "\_" << i + 1;

processImage( os.str(), frames.image( i ) );

}

break;

}

case tronis::TronisDataType::PoseVelocity:

{

processPoseVelocity( data\_model.get\_typed<tronis::PoseVelocitySub>() );

break;

}

case tronis::TronisDataType::BoxData:

{

processObject( data\_model.get\_typed<tronis::BoxDataSub>() );

// std::cout << data\_model.get\_typed<tronis::BoxDataSub>()->ToString() <<

// std::endl;

break;

}

default:

{

std::cout << data\_model->ToString() << std::endl;

break;

}

}

return true;

}

else

{

std::cout << data\_model->ToString() << std::endl;

return false;

}

}

protected:

// Function to show an openCV image in a separate window

void showImage( std::string image\_name, cv::Mat image )

{

cv::Mat out = image;

if( image.type() == CV\_32F || image.type() == CV\_64F )

{

cv::normalize( image, out, 0.0, 1.0, cv::NORM\_MINMAX, image.type() );

}

cv::namedWindow( image\_name.c\_str(), cv::WINDOW\_NORMAL );

cv::imshow( image\_name.c\_str(), out );

}

// Function to convert tronis image to openCV image

bool processImage( const std::string& base\_name, const tronis::Image& image )

{

std::cout << "processImage" << std::endl;

if( image.empty() )

{

std::cout << "empty image" << std::endl;

return false;

}

image\_name\_ = base\_name;

image\_ = tronis::image2Mat( image );

detectLanes();

showImage( image\_name\_, image\_ );

return true;

}

};

// main loop opens socket and listens for incoming data

int main( int argc, char\*\* argv )

{

std::cout << "Welcome to lane assistant" << std::endl;

// specify socket parameters

std::string socket\_type = "TcpSocket";

std::string socket\_ip = "127.0.0.1";

std::string socket\_port = "7778";

std::ostringstream socket\_params;

socket\_params << "{Socket:\"" << socket\_type << "\", IpBind:\"" << socket\_ip

<< "\", PortBind:" << socket\_port << "}";

int key\_press = 0; // close app on key press 'q'

tronis::CircularMultiQueuedSocket msg\_grabber;

uint32\_t timeout\_ms = 500; // close grabber, if last received msg is older than this param

LaneAssistant lane\_assistant;

while( key\_press != 'q' )

{

std::cout << "Wait for connection..." << std::endl;

msg\_grabber.open\_str( socket\_params.str() );

if( !msg\_grabber.isOpen() )

{

printf( "Failed to open grabber, retry...!\n" );

continue;

}

std::cout << "Start grabbing" << std::endl;

tronis::SocketData received\_data;

uint32\_t time\_ms = 0;

while( key\_press != 'q' )

{

// wait for data, close after timeout\_ms without new data

if( msg\_grabber.tryPop( received\_data, true ) )

{

// data received! reset timer

time\_ms = 0;

// convert socket data to tronis model data

tronis::SocketDataStream data\_stream( received\_data );

tronis::ModelDataWrapper data\_model(

tronis::Models::Create( data\_stream, tronis::MessageFormat::raw ) );

if( !data\_model.is\_valid() )

{

std::cout << "received invalid data, continue..." << std::endl;

continue;

}

// identify data type

lane\_assistant.getData( data\_model );

lane\_assistant.processData( msg\_grabber );

}

else

{

// no data received, update timer

++time\_ms;

if( time\_ms > timeout\_ms )

{

std::cout << "Timeout, no data" << std::endl;

msg\_grabber.close();

break;

}

else

{

std::this\_thread::sleep\_for( std::chrono::milliseconds( 10 ) );

key\_press = cv::waitKey( 1 );

}

}

}

msg\_grabber.close();

}

return 0;

}