**计算机与网络学院**

《机器视觉》课程设计报告

基于OpenCV的车道线与道路标示牌的实时检测

|  |  |  |  |
| --- | --- | --- | --- |
| 姓名1： | 肖肇斌 | 姓名2： | 陈佳婷 |
| 学号1： | 201641402213 | 学号2： | 201641402444 |

|  |
| --- |
| 指导老师评语:  评定成绩: 签名: 日期: |

目录

**[一、概述](#_Toc532238799)** [1](#_Toc532238799)

[1.1 课程设计的目的 1](#_Toc532238800)

[1.2 课程设计任务及要求 1](#_Toc532238801)

[1.3 开发该系统软件环境及使用的技术说明 1](#_Toc532238802)

**[二、系统设计的基本概念与原理](#_Toc532238803)** [1](#_Toc532238803)

[2.1 系统基本工作流程 1](#_Toc532238804)

[2.2 车道线检测算法设计 1](#_Toc532238805)

[2.3 道路标示牌检测算法设计 1](#_Toc532238806)

[2.4 用户接口设计 1](#_Toc532238807)

**[三、系统实施详细说明及结果](#_Toc532238808)** [1](#_Toc532238808)

[3.1道路检测算法的实现 1](#_Toc532238809)

[3.2 道路标示牌检测算法实现 1](#_Toc532238810)

[3.3系统运行结果及算法性能 1](#_Toc532238811)

**[四、课程设计总结](#_Toc532238812)** [1](#_Toc532238812)

**[参考文献](#_Toc532238813)** [2](#_Toc532238813)

**一、概述**

1.1 课程设计的目的

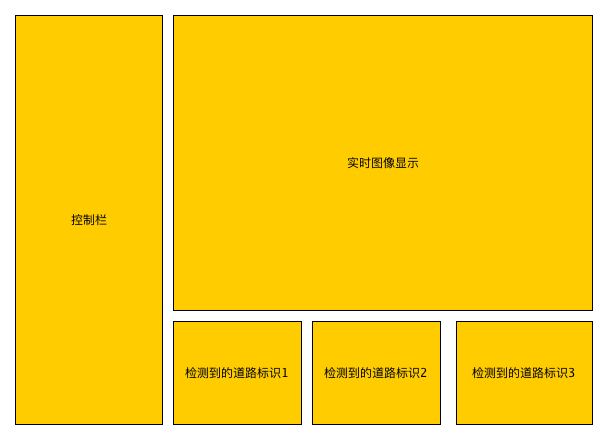
据调查，道路交通事故中有 1/3 是由车辆变道或车辆偏离其正常行驶车道区域所导致的。美国联邦公路局的研究表明：如果可以获得车辆与车道之间的相对位置信息，则可以防止 53%左右的车道偏离事故，因此针对路面标线检测的研究是实现车道偏离警告系统的关键技术，其对于实现车辆的完全自主驾驶具有深远的意义。

1.2 课程设计任务及要求

**利用OpenCV库和其他开源工具，实现车道线和道路标示牌的实时检测**

系统的基本功能要求如下：

1. 利用C/C++代码实现，用git进行版本管理，代码提交应该包含有开发日志（git提交日志）。
2. 测试和验证视频样本由老师提供，也可自行采集，自行采集的应该包括至少两类目标：明显的道路线和一定数量的道路标识牌。
3. 提供比较友好的用户接口，可以由用户自行加载不同的视频。应该包含合适的输出界面，将结果呈现给用户。
4. 检测流程应该包括“道路预处理🡪车道线特征提取🡪车道线检测”
5. 检测后的结果应该能实时输出，例如：检测到的车道线实时与视频显示在同一窗口。
6. 实时检测出视频中的道路标识，从视频中把道路标识分割出来，并显示在窗口中。
7. 界面实现可以参考下图，也可以按照自己的方式实现。



1.3 开发该系统软件环境及使用的技术说明

开发环境: ubuntu,Qt,Clion,github

所用技术: Qt,opencv,c++

**二、系统设计的基本概念与原理**

2.1 系统基本工作流程

系统整个工作流程图

文字描述：

1. 创建窗体
2. 每个窗体点击之后将choice赋值为对应的值
3. 视频检测函数在另一个线程(设置detach),循环判断choice（-2时退出）的值
4. 视频检测函数检测到为对应值时并获取到执行锁时执行相应操作：

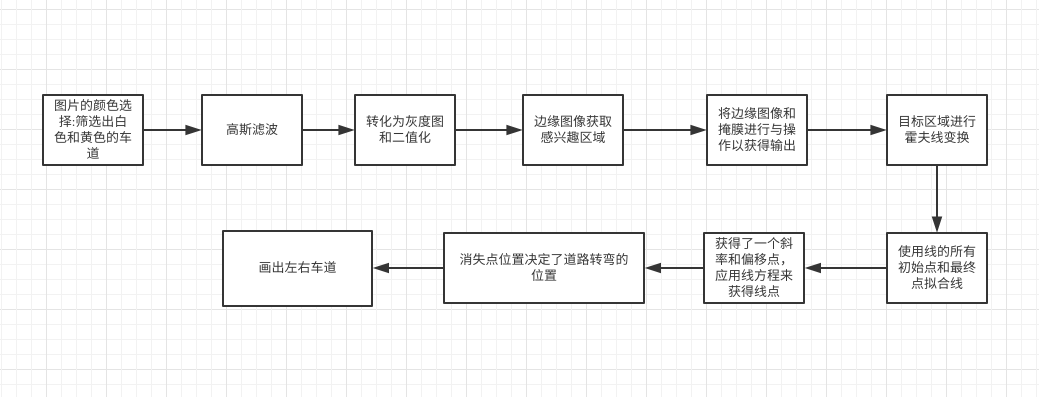
1:道路检测 2:路标检测 3:路标和道路一起检测 4:暂停/开始 其它:choice=-1,退出

执行锁是防止choice为 -1(未选择视频)和 0(未选择检测模式) 时增加cpu负载

1. 窗体退出之后，主线程将choice = -2 （使子线程退出）

2.2 车道线检测算法设计

算法流程图:



文字描述:

1. 选取颜色区域
2. 高斯滤波
3. 边缘检测
4. 获得ROI(感兴趣区域）＋　滤波
5. 在处理后的图像中获得霍夫线
6. 检测车道是否转向
7. 应用回归在车道的每一侧获得唯一的线
8. 通过确定线的消失点来预测转弯

2.3 道路标示牌检测算法设计

算法流程图：



文字描述：

1. 对图片进行前期处理（便于后续操作）：模糊化，滤波处理，RGB转为HSV
2. 根据指定颜色的阈值对图像进行分割
3. 对分割后的图像进行二值化(使图像对比效果更好并去部分噪点)
4. 对图像进行闭运算(使得轮廓封闭，此处调节参数使得路标内部仍存在内轮廓)
5. 对于部分视频可以自行设置不感兴趣区域（指定区域不检测）
6. 对现在的图像进行轮廓查找
7. 对每个轮廓都进行初步筛选
8. 对筛选过后的轮廓做外接矩形
9. 根据角度初步筛选矩形
10. 根据长宽比例对矩形进行筛选
11. 根据矩形将原图对应位置裁剪并显示

2.4 用户接口设计

接口设计思路和文字描述

1. 设计窗体的框架



1. 在Qt的design模式下将窗体初步设置为上面的布局
2. 对每个小窗口分别设置好对应于父窗体的比例和窗体扩展模式
3. 对每个小窗体设置好容易辨识的对象名称和工具提示

**三、系统实施详细说明及结果**

## 3.1 道路检测算法的实现

## LaneDetector lanedetector; // 创建车道检测类

## cv::Mat selected\_img;

## cv::Mat img\_denoise;

## cv::Mat img\_edges;

## cv::Mat img\_mask;

## cv::Mat img\_wrape;

## cv::Mat M;

## cv::Mat img\_lines;

## std::vector<cv::Vec4i> lines;

## std::vector<std::vector<cv::Vec4i> > left\_right\_lines;

## std::vector<cv::Point> lane;

## std::string turn;

## int flag\_plot = -1;

## int i = 0;

## //选取颜色区域

## selected\_img=lanedetector.selectColor(frame);

## // 高斯滤波

## img\_denoise = lanedetector.deNoise(selected\_img);

## // 边缘检测

## img\_edges = lanedetector.edgeDetector(img\_denoise);

## // 获得ROI(感兴趣区域）＋　滤波

## img\_mask = lanedetector.mask(img\_edges);

## img\_mask=lanedetector.deNoise(img\_mask);

## cv::Mat temp=img\_mask.clone();

## // 在处理后的图像中获得霍夫线

## lines = lanedetector.houghLines(img\_mask);

## if (!lines.empty())

## {

## // 检测车道是转向

## left\_right\_lines = lanedetector.lineSeparation(lines, img\_edges);

## // 应用回归在车道的每一侧获得唯一的线

## lane = lanedetector.regression(left\_right\_lines, frame);

## if(!lane.size()){

## return;

## }

## // 通过确定线的消失点来预测转弯

## turn = lanedetector.predictTurn();

## // 在视频中显示方向

## flag\_plot = lanedetector.plotLane(frame, lane, turn);

## i += 1;

## }

## else {

## flag\_plot = -1;

## }

头文件：

#include <string>

#include <vector>

#include <opencv2/opencv.hpp>

#include "LaneDetector.h"

//图片的颜色选择

cv::Mat LaneDetector::selectColor(cv::Mat inputImage){

cv::Mat white\_mask;

cv::Mat yellow\_mask;

cv::Mat mask\_wy;

cv::Mat masked;

cv::inRange(inputImage,cv::Scalar(0,200,0),cv::Scalar(255,255,255),white\_mask);

cv::inRange(inputImage,cv::Scalar(10,0,100),cv::Scalar(40,255,255),yellow\_mask);

// cv::namedWindow("white", 0);

// cv::imshow("white",white\_mask);

// cv::namedWindow("yellow", 0);

// cv::imshow("yellow",yellow\_mask);

cv::bitwise\_or(white\_mask,yellow\_mask,mask\_wy);

// cv::namedWindow("mask\_wy", 0);

// cv::imshow("mask\_wy",mask\_wy);

cv::bitwise\_and(inputImage,inputImage,masked,mask\_wy);

// cv::namedWindow("masked", 0);

// cv::resizeWindow("masked",800,380);

// cv::imshow("masked", masked);

return masked;

}

//高斯滤波

cv::Mat LaneDetector::deNoise(cv::Mat inputImage) {

cv::Mat output;

cv::GaussianBlur(inputImage, output, cv::Size(3, 3), 0, 0);

return output;

}

// 边缘检测

cv::Mat LaneDetector::edgeDetector(cv::Mat img\_noise) {

cv::Mat output;

cv::Mat kernel;

cv::Point anchor;

// cv::resize(img\_noise,img\_noise,cv::Size(),0.75,0.75);

// 灰度图

cv::cvtColor(img\_noise, output, cv::COLOR\_RGB2GRAY);

// 二值化

cv::threshold(output, output, 140, 255, cv::THRESH\_BINARY);

//创建内核[-1 0 1]

//这个内核基于在Mathworks的车道偏离警告系统

anchor = cv::Point(-1, -1);

kernel = cv::Mat(1, 3, CV\_32F);

kernel.at<float>(0, 0) = -1;

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

kernel.at<float>(0, 2) = 1;

// 过滤二进制图像以获得边缘

cv::filter2D(output, output, -1, kernel, anchor, 0, cv::BORDER\_DEFAULT);

// cv::namedWindow("output", 0);

// cv::resizeWindow("output",800,380);

// cv::imshow("output", output);

return output;

}

// 边缘图像获取感兴趣区域

cv::Mat LaneDetector::mask(cv::Mat img\_edges) {

cv::Mat output;

int width,height;

width=img\_edges.cols;

height=img\_edges.rows;

cv::Mat mask = cv::Mat::zeros(img\_edges.size(), img\_edges.type());

cv::Point pts[4] = {

cv::Point(0, height),//左下角开始顺时针

cv::Point(width\*0.1, height\*0.95),

cv::Point(width\*0.4, height\*0.6),

cv::Point(width\*0.9, height\*0.95)

};//original的参数

// cv::Point pts[4] = {

// cv::Point(0, height\*0.95),//左下角开始顺时针

// cv::Point(width\*0.1, height\*0.75),

// cv::Point(width\*0.4, height\*0.75),

// cv::Point(width\*0.5, height)

// };//second01.02的参数

// cv::Point pts[4] = {

// cv::Point(0, 720),

// cv::Point(210, 450),

// cv::Point(350, 450),

// cv::Point(550, 720)

// };

// 创建二进制多边形，填充多边形获取车道内部

cv::fillConvexPoly(mask, pts, 4, cv::Scalar(255, 0, 0));

// cv::namedWindow("mask", 0);

// cv::resizeWindow("mask",800,380);

// cv::imshow("mask",mask);

// 将边缘图像和掩膜进行与操作以获得输出

cv::bitwise\_and(img\_edges, mask, output);

// cv::namedWindow("ROI", 0);

// cv::resizeWindow("ROI",800,380);

// cv::imshow("ROI",output);

// //查找多边形

// cv::approxPolyDP(output,output,)

return output;

}

// 霍夫线

std::vector<cv::Vec4i> LaneDetector::houghLines(cv::Mat img\_mask) {

std::vector<cv::Vec4i> line;

// 使用概率Hough变换在二进制图像中查找线段

HoughLinesP(img\_mask, line, 1, CV\_PI / 180, 20, 20, 30);

return line;

}

// 筛选左右车道

std::vector<std::vector<cv::Vec4i> > LaneDetector::lineSeparation(std::vector<cv::Vec4i> lines, cv::Mat img\_edges) {

std::vector<std::vector<cv::Vec4i> > output(2);

size\_t j = 0;

cv::Point ini;

cv::Point fini;

double slope\_thresh = 0.3;

std::vector<double> slopes;

std::vector<cv::Vec4i> selected\_lines;

std::vector<cv::Vec4i> right\_lines, left\_lines;

// 计算所有检测到的线的斜率

for (auto i : lines) {

ini = cv::Point(i[0], i[1]);

fini = cv::Point(i[2], i[3]);

// slope = (y1 - y0)/(x1 - x0)－－斜率

double slope = (static\_cast<double>(fini.y) - static\_cast<double>(ini.y)) / (static\_cast<double>(fini.x) - static\_cast<double>(ini.x) + 0.00001);

// 如果斜率接近０，则丢弃该线

// 否则，分别保存斜率

if (std::abs(slope) > slope\_thresh) {

slopes.push\_back(slope);

selected\_lines.push\_back(i);

}

}

// 将线条分成右线和左线

img\_center = static\_cast<double>((img\_edges.cols / 2));

while (j < selected\_lines.size()) {

ini = cv::Point(selected\_lines[j][0], selected\_lines[j][1]);

fini = cv::Point(selected\_lines[j][2], selected\_lines[j][3]);

// 将线分类为左侧或右侧的条件

if (slopes[j] > 0 && fini.x > img\_center && ini.x > img\_center) {

right\_lines.push\_back(selected\_lines[j]);

right\_flag = true;

}

else if (slopes[j] < 0 && fini.x < img\_center && ini.x < img\_center) {

left\_lines.push\_back(selected\_lines[j]);

left\_flag = true;

}

j++;

}

output[0] = right\_lines;

output[1] = left\_lines;

return output;

}

// 回归

std::vector<cv::Point> LaneDetector::regression(std::vector<std::vector<cv::Vec4i> > left\_right\_lines, cv::Mat inputImage) {

if(!right\_flag||!left\_flag){

vector<cv::Point> n;

return n;

}

std::vector<cv::Point> output(4);

cv::Point ini;

cv::Point fini;

cv::Point ini2;

cv::Point fini2;

cv::Vec4d right\_line;

cv::Vec4d left\_line;

std::vector<cv::Point> right\_pts;

std::vector<cv::Point> left\_pts;

// 如果检测到右线，则使用线的所有初始点和最终点拟合线

if (right\_flag == true) {

for (auto i : left\_right\_lines[0]) {

ini = cv::Point(i[0], i[1]);

fini = cv::Point(i[2], i[3]);

right\_pts.push\_back(ini);

right\_pts.push\_back(fini);

}

if (right\_pts.size() > 0) {

// 合成右线

cv::fitLine(right\_pts, right\_line, CV\_DIST\_L2, 0, 0.01, 0.01);

right\_m = right\_line[1] / right\_line[0];

right\_b = cv::Point(right\_line[2], right\_line[3]);

}

}

// 如果检测到左线，则使用线的所有初始点和最终点拟合线

if (left\_flag == true) {

for (auto j : left\_right\_lines[1]) {

ini2 = cv::Point(j[0], j[1]);

fini2 = cv::Point(j[2], j[3]);

left\_pts.push\_back(ini2);

left\_pts.push\_back(fini2);

}

if (left\_pts.size() > 0) {

// 合成左线

cv::fitLine(left\_pts, left\_line, CV\_DIST\_L2, 0, 0.01, 0.01);

left\_m = left\_line[1] / left\_line[0];

left\_b = cv::Point(left\_line[2], left\_line[3]);

}

}

// 获得了一个斜率和偏移点，应用线方程来获得线点

int ini\_y = inputImage.rows;

int fin\_y = 470;

double right\_ini\_x = ((ini\_y - right\_b.y) / right\_m) + right\_b.x;

double right\_fin\_x = ((fin\_y - right\_b.y) / right\_m) + right\_b.x;

double left\_ini\_x = ((ini\_y - left\_b.y) / left\_m) + left\_b.x;

double left\_fin\_x = ((fin\_y - left\_b.y) / left\_m) + left\_b.x;

output[0] = cv::Point(right\_ini\_x, ini\_y);

output[1] = cv::Point(right\_fin\_x, fin\_y);

output[2] = cv::Point(left\_ini\_x, ini\_y);

output[3] = cv::Point(left\_fin\_x, fin\_y);

return output;

}

// 转向

std::string LaneDetector::predictTurn() {

std::string output;

double vanish\_x;

double thr\_vp = 10;

// 消失点

vanish\_x = static\_cast<double>(((right\_m\*right\_b.x) - (left\_m\*left\_b.x) - right\_b.y + left\_b.y) / (right\_m - left\_m));

// 消失点位置决定了道路转弯的位置

if (vanish\_x < (img\_center - thr\_vp))

output = "Left Turn";

else if (vanish\_x >(img\_center + thr\_vp))

output = "Right Turn";

else if (vanish\_x >= (img\_center - thr\_vp) && vanish\_x <= (img\_center + thr\_vp))

output = "Straight";

return output;

}

// 画出结果

int LaneDetector::plotLane(cv::Mat inputImage, std::vector<cv::Point> lane, std::string turn) {

std::vector<cv::Point> poly\_points;

cv::Mat output;

// 创建清晰的多边形以更好地显示通道

inputImage.copyTo(output);

poly\_points.push\_back(lane[2]);

poly\_points.push\_back(lane[0]);

poly\_points.push\_back(lane[1]);

poly\_points.push\_back(lane[3]);

cv::fillConvexPoly(output, poly\_points, cv::Scalar(0, 0, 255), CV\_AA, 0);

cv::addWeighted(output, 0.3, inputImage, 1.0 - 0.3, 0, inputImage);

// 绘制车道边界的两条线

cv::line(inputImage, lane[0], lane[1], cv::Scalar(0, 255, 255), 5, CV\_AA);

cv::line(inputImage, lane[2], lane[3], cv::Scalar(0, 255, 255), 5, CV\_AA);

// 显示转向

cv::putText(inputImage, turn, cv::Point(50, 90), cv::FONT\_HERSHEY\_COMPLEX\_SMALL, 3, cvScalar(0, 255, 0), 1, CV\_AA);

// // 显示最终图片

// cv::namedWindow("Lane", 0);

// cv::resizeWindow("Lane",800,380);

// cv::imshow("Lane", inputImage);

return 0;

}

## 3.2 道路标示牌检测算法实现

关键代码及文字说明

//合并的路标检测函数  
**void** RoadSign::checkRoadSign(Mat inputImage,Mat outputImage,**double** colorLimit[],**int** binaryLimit[],**int** erodeNum[],**int** dilateNum[],**int** closeNum[],**double** angleLimit[],**double** wid\_hei[],**int** sizeOfArea,**int** someArea,**int** someOperate){  
 Mat hsv,operateMat;;  
 Mat operateImage = inputImage.clone();  
  
 //滤波  
 **if**(someOperate) {  
 blur(operateImage, operateImage, Size(3, 3));  
 GaussianBlur(operateImage, operateImage, Size(3, 3), 0, 0);  
 medianBlur(operateImage, operateImage, 5);  
 }  
  
 //RGB转为HSV图像显示  
 cv::cvtColor(operateImage, hsv, cv::*COLOR\_BGR2HSV*);  
  
 //阈值分割  
 //cv::inRange(hsv, cv::Scalar(214-BLUE,61-BLUE, 62-BLUE), cv::Scalar(214+BLUE, 61+BLUE,62+BLUE), blue\_mask);  
 cv::inRange(hsv, cv::Scalar(colorLimit[0], colorLimit[1],colorLimit[2]), cv::Scalar(colorLimit[3],colorLimit[4],colorLimit[5]),  
 operateMat);  
  
 //二值化  
 threshold(operateMat, operateMat, binaryLimit[0],binaryLimit[1], *CV\_THRESH\_BINARY*);  
  
 //腐蚀膨胀  
 erode(operateMat, operateMat, getStructuringElement(*MORPH\_RECT*, Size(erodeNum[0],erodeNum[1])));  
  
 dilate(operateMat, operateMat, getStructuringElement(*MORPH\_RECT*, Size(dilateNum[0],dilateNum[1])));  
  
 //闭运算  
 Mat kernel = getStructuringElement(*MORPH\_RECT*, Size(closeNum[0],closeNum[1]));  
 morphologyEx(operateMat, operateMat, *MORPH\_CLOSE*, kernel);  
  
 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*添加感兴趣区域\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 **if**(someArea) {  
 **int** width, height;  
 width = operateMat.cols;  
 height = operateMat.rows;  
  
 //original的参数  
 cv::Point pts[4] = {  
 Point(width / 5, height / 4),//左下角开始顺时针  
 Point(width / 5, 0),  
 Point(width \* 4 / 5, 0),  
 Point(width \* 4 / 5, height / 4)  
 };  
  
 // 创建二进制多边形，填充多边形获取感兴趣区域  
 cv::fillConvexPoly(operateMat, pts, 4, cv::Scalar(0, 0, 0));  
 //imshow("blue", operateMat);  
 }  
 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
  
 //查找轮廓并显示  
 vector<vector<Point>> contours;  
 vector<Vec4i> hierarchy;  
 findContours(operateMat, contours, hierarchy, CV\_RETR\_TREE, CV\_CHAIN\_APPROX\_SIMPLE);  
 vector<vector<Point>> approxContours(contours.size());  
 **unsigned int** index = 0;  
 **while** (index < contours.size()){  
 approxPolyDP(contours[index],approxContours[index],20,**true**);  
 **if**(approxContours.at(index).size()>15){  
 //drawContours(outputImage,approxContours,index,Scalar(0,0,255),2);  
 index++;  
 **continue**;  
 }  
  
 //轮廓筛选  
 **if** (contours.at(index).size() >= 4 /\*轮廓角点初步筛选\*/  
 && hierarchy.at(index).val[2] >= 0 && hierarchy.at(index).val[3] < 0 /\*找具有内轮廓的最外层轮廓\*/) {  
 RotatedRect cnt = minAreaRect(contours.at(index));//找出外接矩阵  
 **if** (cnt.angle > angleLimit[0]&& cnt.angle <=angleLimit[1]) {//矩阵外界角度筛选  
 **int** middle = cnt.size.width / cnt.size.height;  
 **if**(middle >= wid\_hei[0]&&middle <= wid\_hei[1]&&cnt.size.area() > 100 && cnt.size.area() < sizeOfArea) {//矩阵长宽比和矩阵大小筛选  
 Point2f box[4];  
 cnt.points(box);  
 //画出外接矩阵  
 //rectangle(outputImage,box[0],box[2],Scalar(0,255,255),2);

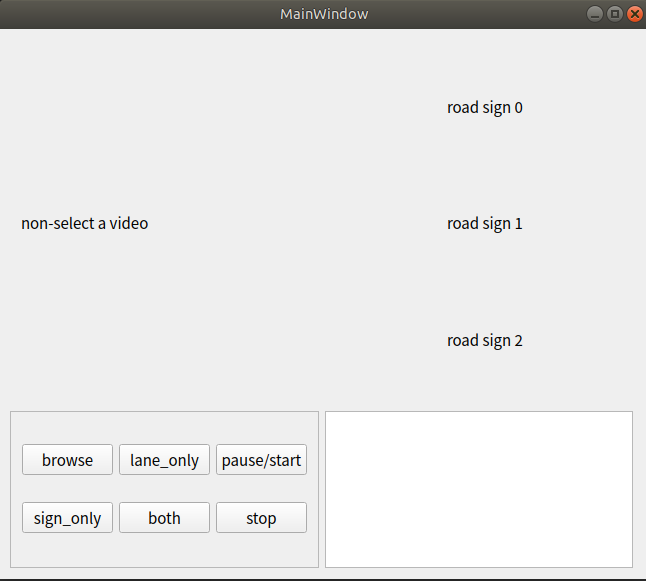
//裁剪出筛选后  
 Rect rect=boundingRect(contours.at(index));  
 Mat ROI=outputImage(rect).clone();  
 cvtColor(ROI,ROI,CV\_BGR2RGB);  
 QImage image = QImage( (**const unsigned char**\*)(ROI.data), ROI.cols, ROI.rows, QImage::Format\_RGB888 );  
 MainWindow \*p=MainWindow::ptr;  
 **int** wid=ROI.cols>p->getUI()->roadSign0->width()? p->getUI()->roadSign0->width():ROI.cols;  
 **int** hei=ROI.rows>p->getUI()->roadSign0->height()?p->getUI()->roadSign0->height():ROI.rows;

//将裁剪的图片显示到窗体指定位置

**if**(count==0){  
 p->getUI()->roadSign0->setPixmap( QPixmap::fromImage(image.scaled(wid,hei)));  
 //p->ui->outputImage->resize(100,100);  
 p->getUI()->roadSign0->show();  
 }**else if**(count==1){  
 p->getUI()->roadSign1->setPixmap( QPixmap::fromImage(image.scaled(wid,hei)));  
 //p->ui->outputImage->resize(100,100);  
 p->getUI()->roadSign1->show();  
 }**else**{  
 p->getUI()->roadSign2->setPixmap( QPixmap::fromImage(image.scaled(wid,hei)));  
 //p->ui->outputImage->resize(100,100);  
 p->getUI()->roadSign2->show();  
 }  
 count++;  
 count%=3;  
 }  
 }  
 }  
 index++;  
 }  
}  
  
//获取不感兴趣区域  
Mat RoadSign::mask(cv::Mat img) {  
 cv::Mat output;  
 **int** width, height;  
 width = img.cols;  
 height = img.rows;  
  
 cv::Mat mask = cv::Mat::zeros(img.size(), img.type());  
  
 cv::Point pts[4] = {  
 Point(width \* 2 / 5, height / 5),//左下角开始顺时针  
 Point(width \* 2 /5 , 0),  
 Point(width \* 4 / 5, 0),  
 Point(width \* 4 / 5, height / 5)  
 };  
  
 // 创建二进制多边形，填充多边形获取感兴趣区域  
 cv::fillConvexPoly(mask, pts, 4, cv::Scalar(255, 255, 255));  
  
 // 将边缘图像和掩膜进行与操作以获得输出  
 cv::bitwise\_and(img, mask, output);  
  
 cv::namedWindow("ROI", 0);  
 cv::resizeWindow("ROI", 800, 380);  
 cv::imshow("ROI", output);  
  
 **return** output;  
}

## 3.3系统运行结果及算法性能

1. 运行结果:
2. 前端ui:



1. 路标检测:



1. 道路检测：



1. 算法效率：

路标检测: 138.186ms (取十次时间求平均)

道路检测: 66.7734ms (取十次时间求平均)

**四、课程设计总结**

1. 路标检测：
2. 路标阈值的阈值无法确定(无法做到普遍性，对于不同视频采用的阈值一般不同)

解决方式：

多次调试，尽量使阈值可以用于不同视频

1. 图像处理的许多函数不是很清楚

解决方式：

查看opencv官方文档

百度/Google部分demo

1. ui：
2. 对于窗体布局的不了解

解决方式：

依靠Qt官方文档去了解

百度/Google有关demo

1. Qt的UI设计中Thread函数貌似失去“作用（使用和C++官方文档中的说明不相符合”

解决方式：

百度/Google有关Qt窗体中的线程知识

Qt官方文档查找有关线程的说明使用

C++官方文档查看有关thread类的说明使用

Qt的Ui中使用QTtread等函数，thread类在主函数中使用

1. 程序运行结果不知道如何显示到UI

解决方式：

百度/Google有关Qt显示视频和图片的demo

查看Qt官方文档

1. UI显示方面windows和ubuntu在Qt中的实现有所不同

解决方式：

UI及显示方面的API等均采用两个系统共有

百度/Google有关两个系统的Qt库

# **参考文献**

**Qt官方文档 :http://doc.qt.io/**

**C++官方文档 :http://www.cplusplus.com/doc/ 和 https://devdocs.io/cpp/**

**opencv官方文档 :https://docs.opencv.org/master/examples.html**

**CSDN博客: <https://blog.csdn.net/qq_30909117/article/details/79164055>**

**<https://blog.csdn.net/pockyym/article/details/12203709>**

**<https://blog.csdn.net/sinat_36420785/article/details/61432531>**

**https://blog.csdn.net/u012637471/article/details/48623657**

**<https://blog.csdn.net/u012637471/article/details/48623657>**

**其他: <https://www.cnblogs.com/k1412/p/6884484.html>**