Vision-based Road Detection

Mestrado Integrado em Engenharia Informática e Computação
Visão por Computador
EIC0104-1S

Jorge Miguel Guerra Santos - 201106922 (ei11057@fe.up.pt)
Jose Rui Neto Faria - 201104362 (ei11046@fe.up.pt)
Ricardo Daniel Soares da Silva - 201108043 (ei11079@fe.up.pt)

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

30 de Outubro de 2015

Conteúdo

1 Introdução

Este projeto foi realizado no âmbito da unidade curricular de Visão de Computador do Mestrado Integrado em Engenharia Informática e Computação. O principal objectivo do trabalho era desenvolver um programa de detecção de estradas em imagens e vídeos, usando vários tipos de segmentação e detecção de características, utilizando a linguagem de programação C++ e a biblioteca OpenCV.

2 Algoritmo Proposto

2.1 Tratamento Inicial da Imagem

Inicialmente ocorre a eliminação da secção superior da imagem, uma vez que nessa parte encontra-se, normalmente, o céu e as nuvens, que dão origem a erros, para além da diminuição da performance do programa ao analisar arestas pouco relevantes para a detecção da estrada.

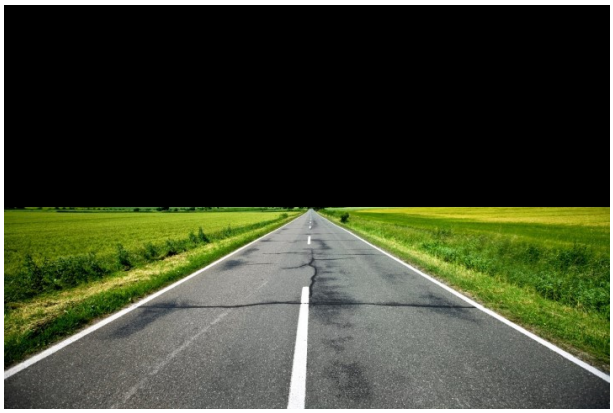

Figura 1: Secção inferior da imagem

2.2 Segmentação por cor

De seguida é aplicado uma segmentação por cor, usando o sistemas de cores HSV, e transformações morfológicas, tais como Opening e Closing, de modo a eliminar a imagem de fundo o melhor possível e facilitar cálculos ao longo do algoritmo proposto.

Figura 2: Segmentação da estrada

2.3 Deteção da Estrada

2.3.1 Tratamento da Imagem

Seguidamente, de forma a detetar os limites da estrada na imagem, é aplicado Gaussian Blur, de forma a reduzir o ruído da imagem, seguido de Canny Edge Detection, para a detecção inicial de arestas, e, por fim, é aplicado o Probabilistic Hough Transform, de forma a complementar o resultado do método anterior, resultando num vector de linhas.

2.3.2 Deteção das linhas

De forma a encontrar as duas linhas que correspondem aos separador direito e esquerdo da estrada é feito uma análise aos dois pontos de cada linha, observando os seus pixeis vizinhos, à procura de pixeis semelhantes à cor normalmente encontrada nas faixas da estrada. De momento são analisados 36 pixeis vizinhos, divididos em 4 "quadrantes" de 9 pixeis, sendo escolhido o quadrante com maior percentagem de pixeis brancos. Será selecionado a linha com a maior percentagem de pixeis brancos.

Caso não sejam encontrados pixeis brancos em nenhum dos pontos das linhas analisadas, os parâmetros de pesquisa são alterados,passando-se a calcular a melhor linha através do seu ângulo.

Se isso falhar, o programa usa as últimas linhas que passaram na análise.

Por fim, o cálculo da interseção das linhas escolhidas obtem a estimação do vanishing point e sinaliza-o na imagem.

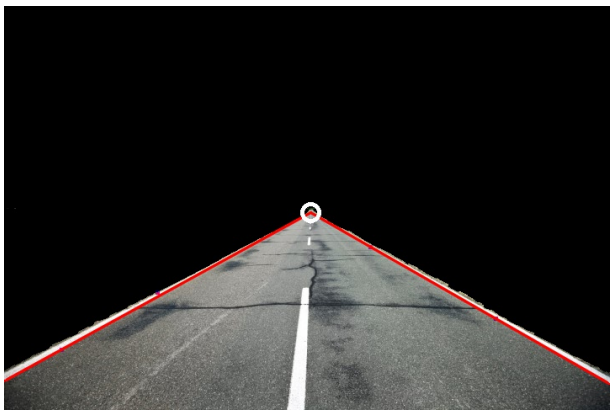

Figura 3: Deteção de Linhas e do Vanishing Point

3 Estado do Projecto

Apesar do projecto desenvolvido obter bons resultados, ainda há desafios que não foram ultrapassados, nomeadamente:

1. Estradas com curvas ou deformações(ex: cruzamentos);
2. Estradas com diferentes níveis de iluminação ou cores invulgares;

O algoritmo desenvolvido neste projecto é eficiente a nível de processamento da imagem e é capaz de detectar ruas estruturadas (excepto com curvas), ruas estruturadas com carros e algumas ruas não estruturadas.

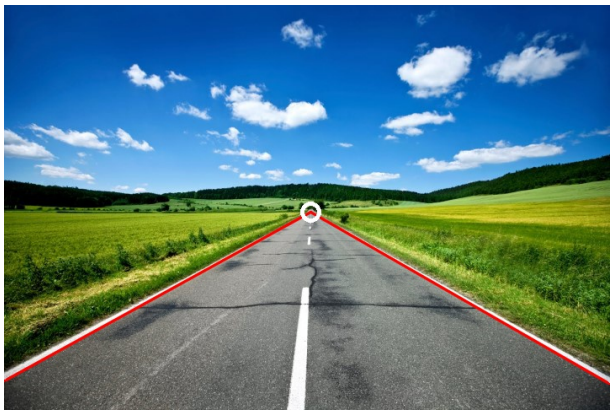

Figura 4: Resultado Final

Anexos

A Exemplos

Figura 5: Deteção de Linhas e do Vanishing Point

Figura 6: Deteção de Linhas e do Vanishing Point

Figura 7: Deteção de Linhas e do Vanishing Point

B Código

```
#include <iostream>
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"

using namespace cv;
using namespace std;

Point RIGHT_LINE[2], LEFT_LINE[2], INTERSECT = Point(0, 0);
int MENU_OPTION;
string FILENAME;
const bool RIGHT = true, LEFT = false;

double cross(Point v1, Point v2) {
   return v1.x*v2.y - v1.y*v2.x;
}

bool getIntersectionPoint(Point a1, Point a2, Point b1, Point b2,
    Point & intPnt) {
   Point p = a1;
   Point q = b1;
   Point r(a2 - a1);
   Point s(b2 - b1);

   if (cross(r, s) == 0) { return false; }

   double t = cross(q - p, s) / cross(r, s);

   intPnt = p + t*r;

   return true;
}

double Slope(int x0, int y0, int x1, int y1) {
   return (double)(y1 - y0) / (x1 - x0);
}

// Get a full line from two points
void fullLine(Mat *img, Point a, Point b, Point new_point[]) {
   double slope = Slope(a.x, a.y, b.x, b.y);
   int line_thickness = 2;

   new_point[0] = Point(0, 0);
   new_point[1] = Point(img->cols, img->rows);
```

```
   new_point[0].y = -(a.x - new_point[0].x) * slope + a.y;
   new_point[1].y = -(b.x - new_point[1].x) * slope + b.y;
}

void drawLine(Mat *img, Point line[], Point intersection) {

   if (line[0].y > line[1].y) {
      cv::line(*img, line[0], intersection, Scalar(0, 0, 255), 2,
         CV_AA);
   }
   else {
      cv::line(*img, line[1], intersection, Scalar(0, 0, 255), 2,
         CV_AA);
   }


}

//Check if neighbourhood pixels are white
double checkNeighbourhoodPixels(Mat src, Mat src_hsv, double
    numNeighbourhoodPixel, Vec4i line, bool orientation) {

   double numWhitePixelsBottomRight = 0, numWhitePixelsBottomLeft =
      0, numWhitePixelsTopRight = 0, numWhitePixelsTopLeft = 0;

   for (int i = 0; i < sqrt(numNeighbourhoodPixel / 4); i++) {
      for (int j = 0; j < sqrt(numNeighbourhoodPixel / 4); j++) {
         if ((line[0] - i > 0 && line[0] + i < src.cols) &&
             (line[1] - j > 0 && line[1] + j < src.rows)) {
            Vec3b neighbourPixelBottomRight =
                src_hsv.at<Vec3b>(Point(line[0] + i, line[1] + j));
            neighbourPixelBottomRight.val[0] += 90;
            if (neighbourPixelBottomRight.val[2] > 170 &&
                (neighbourPixelBottomRight.val[1] >= 0 &&
                neighbourPixelBottomRight.val[1] < 40))
               numWhitePixelsBottomRight++;

            Vec3b neighbourPixelBottomLeft =
                src_hsv.at<Vec3b>(Point(line[0] - i, line[1] - j));
            neighbourPixelBottomLeft.val[0] += 90;
            if (neighbourPixelBottomLeft.val[2] > 170 &&
                (neighbourPixelBottomRight.val[1] >= 0 &&
                neighbourPixelBottomRight.val[1] < 40))
               numWhitePixelsBottomLeft++;
```

```
            Vec3b neighbourPixelTopRight =
                src_hsv.at<Vec3b>(Point(line[0] + i, line[1] - j));
            neighbourPixelTopRight.val[0] += 90;
            if (neighbourPixelTopRight.val[2] > 170 &&
                (neighbourPixelBottomRight.val[1] >= 0 &&
                neighbourPixelBottomRight.val[1] < 40))
               numWhitePixelsTopRight++;

            Vec3b neighbourPixelTopLeft =
                src_hsv.at<Vec3b>(Point(line[0] - i, line[1] + j));
            neighbourPixelTopLeft.val[0] += 90;
            if (neighbourPixelTopLeft.val[2] > 170 &&
                (neighbourPixelBottomRight.val[1] >= 0 &&
                neighbourPixelBottomRight.val[1] < 40))
               numWhitePixelsTopLeft++;

            src.at<Vec3b>(Point(line[0] + i, line[1] + j)) =
                neighbourPixelBottomRight;
            src.at<Vec3b>(Point(line[0] - i, line[1] - j)) =
                neighbourPixelBottomLeft;
            src.at<Vec3b>(Point(line[0] + i, line[1] - j)) =
                neighbourPixelTopRight;
            src.at<Vec3b>(Point(line[0] - i, line[1] + j)) =
                neighbourPixelTopLeft;

         }

      }
   }

double numWhitePixelsQuadrants[4] = { numWhitePixelsBottomRight,
    numWhitePixelsBottomLeft, numWhitePixelsTopRight,
    numWhitePixelsTopLeft };
double numWhitePixels = numWhitePixelsQuadrants[0];

for (int i = 0; i < sizeof(numWhitePixelsQuadrants) /
    sizeof(numWhitePixelsQuadrants[0]) - 1; i++) {
   if (numWhitePixelsQuadrants[i + 1] >
       numWhitePixelsQuadrants[i])
      numWhitePixels = numWhitePixelsQuadrants[i + 1];
}

double percentageNeighbourhoodWhitePixels = (numWhitePixels /
    (numNeighbourhoodPixel / 4)) * 100;

return percentageNeighbourhoodWhitePixels;
```

```
}

void detectLines(Mat original, Mat src, bool retry = false) {
   int edgeThresh = 1;
   int lowThreshold = 50;
   int const maxThreshold = 400;
   int kernel_size = 3;

   Mat src_gray, src_hsv, dst, color_dst, copyOriginal;

   original.copyTo(copyOriginal);

   //Gaussian Blur
   cvtColor(src, src_gray, CV_RGB2GRAY);
   cvtColor(src, src_hsv, CV_BGR2HSV);

   GaussianBlur(src_gray, dst, Size(kernel_size, kernel_size), 0,
      0);

   //Canny
   Canny(dst, dst, lowThreshold, maxThreshold, kernel_size);

   src.copyTo(color_dst);

   vector<Vec4i> lines;
   // detect lines
   if (retry) {
      HoughLinesP(dst, lines, 1, CV_PI / 180, 50, 50, 10);
   }
   else {
      HoughLinesP(dst, lines, 1, CV_PI / 180, 50, 100, 1);
   }

   cvtColor(dst, dst, CV_GRAY2BGR);

   // draw lines
   int right_angle = 9999, left_angle = 0;
   double percentageNeighbourhoodWhitePixels = 0;

   for (size_t i = 0; i < lines.size(); i++)
   {
      Vec4i l = lines[i];
      double angle = atan2(l[3] - l[1], l[2] - l[0]) * 180.0 /
         CV_PI;
      angle = angle < 0 ? angle + 360 : angle;
```

```
    angle = angle > 180 ? angle - 180 : angle;

    double numNeighbourhoodPixel = 36;

    if (angle > 10 && angle <= 45) {
       if (angle > left_angle) {
          percentageNeighbourhoodWhitePixels =
              checkNeighbourhoodPixels(src, src_hsv,
              numNeighbourhoodPixel, l, LEFT);

          if (percentageNeighbourhoodWhitePixels >= 1) {
             fullLine(&copyOriginal, Point(l[0], l[1]),
                 Point(l[2], l[3]), LEFT_LINE);
             left_angle = angle;
          }
       }
    }
    else if (angle < 170 && angle >= 135) {
       if (angle < right_angle) {
          percentageNeighbourhoodWhitePixels =
              checkNeighbourhoodPixels(src, src_hsv,
              numNeighbourhoodPixel, l, RIGHT);

          if (percentageNeighbourhoodWhitePixels >= 1) {
             fullLine(&copyOriginal, Point(l[0], l[1]),
                 Point(l[2], l[3]), RIGHT_LINE);
             right_angle = angle;
          }
       }
    }
}

if (percentageNeighbourhoodWhitePixels < 1) {
    for (size_t i = 0; i < lines.size(); i++)
    {
       Vec4i l = lines[i];
       double angle = atan2(l[3] - l[1], l[2] - l[0]) * 180.0 /
           CV_PI;
       angle = angle < 0 ? angle + 360 : angle;
       angle = angle > 180 ? angle - 180 : angle;

       double numNeighbourhoodPixel = 36;

       if (angle > 10 && angle <= 45) {
          if (angle > left_angle) {
```

```
                fullLine(&copyOriginal, Point(l[0], l[1]),
                    Point(l[2], l[3]), LEFT_LINE);
                left_angle = angle;
            }
        }
        else if (angle < 170 && angle >= 135) {
            if (angle < right_angle) {
                fullLine(&copyOriginal, Point(l[0], l[1]),
                    Point(l[2], l[3]), RIGHT_LINE);
                right_angle = angle;
            }
        }
    }
}

getIntersectionPoint(LEFT_LINE[0], LEFT_LINE[1], RIGHT_LINE[0],
    RIGHT_LINE[1], INTERSECT);

if (INTERSECT == Point(0, 0) && MENU_OPTION == 1 && !retry) {
    detectLines(original, src, true);
    return;
}

if (INTERSECT != Point(0, 0)) {
    drawLine(&copyOriginal, LEFT_LINE, INTERSECT);
    drawLine(&copyOriginal, RIGHT_LINE, INTERSECT);
    circle(copyOriginal, INTERSECT, 10, Scalar(255, 255, 255), 3,
        8);
}

imshow("Road Detection", copyOriginal);
}

void roadDetection(Mat src) {

int iLowH = 0, iHighH = 179;
int iLowS = 0, iHighS = 80;
int iLowV = 0, iHighV = 255;

Mat mask, imgOriginal, imgHSV, imgThresholded, whiteImg, road;

//Crop image - uses the bottom half of the image for line
    detection
mask = Mat::zeros(src.size(), CV_8UC3);
rectangle(mask, Point(0, 1 * (mask.rows / 2)), Point(mask.cols,
    mask.rows), Scalar(255, 255, 255), CV_FILLED);
```

```
    cvtColor(mask, mask, CV_BGR2GRAY);

    src.copyTo(imgOriginal, mask);

    //Color segmentation
    cvtColor(imgOriginal, imgHSV, COLOR_BGR2HSV);
    inRange(imgHSV, Scalar(iLowH, iLowS, iLowV), Scalar(iHighH,
        iHighS, iHighV), imgThresholded);

    //morphological opening (removes small objects from the
        foreground)
    erode(imgThresholded, imgThresholded,
        getStructuringElement(MORPH_ELLIPSE, Size(5, 5)));
    dilate(imgThresholded, imgThresholded,
        getStructuringElement(MORPH_ELLIPSE, Size(5, 5)));

    //morphological closing (removes small holes from the foreground)
    dilate(imgThresholded, imgThresholded,
        getStructuringElement(MORPH_ELLIPSE, Size(5, 5)));
    erode(imgThresholded, imgThresholded,
        getStructuringElement(MORPH_ELLIPSE, Size(5, 5)));

    whiteImg = Mat::ones(imgThresholded.size(),
        imgThresholded.type()) * 255;

    imgOriginal.copyTo(road, imgThresholded);

    detectLines(src, road);
}

//Function for video processing
bool videoProcessing() {
    Mat src;
    Mat imgOriginal;

    VideoCapture video(FILENAME);
    Mat thresh, final_mask, image_final, mask;

    if (!video.isOpened())
    {
        cout << "Could not open or find the video" << endl;
        return false;
    }

    while (video.isOpened())
    {
```

```
      bool success = video.read(src);

      if (success) {
         roadDetection(src);
      }

      if (waitKey(30) == 27) //wait for 'esc' key press for 30ms.
          If 'esc' key is pressed, break loop
      {
         cout << "esc key is pressed by user" << endl;
         break;
      }
   }

   return true;
}

//Function for image processing
bool imageProcessing() {
   string imageName(FILENAME);

   Mat src;

   src = imread(imageName.c_str(), IMREAD_COLOR);

   if (src.empty())
   {
      cout << "Could not open or find the image" << endl;
      return false;
   }

   roadDetection(src);

   waitKey(0);

   return true;
}

void menu() {

   cout << " ____                _   ____      _           _   _
                " << endl;
   cout << " | _ \\ ___  __ _  __| | | _ \\ ___| |_ ___ ___| |_(_)
      ___ _ __ " << endl;
   cout << " | |_) / _ \\ / _` |/ _` | | | | |/ _ \\ __/ _ \\/ __|
      __| |/ _ \\| '_ \\ " << endl;
```

```
    cout << " | _ < (_) | (_| | (_| | | |_| | __/ || __/ (__| |_| |
        (_) | | | |" << endl;
    cout << " |_| \\_\\___/ \\__,_|\\__,_| |____/
        \\___|\\__\\___|\\___|\\__|_|\\___/|_| |_|" << endl;
    cout << endl;
    cout << " _____ ___  ___  ___  ___  ___  ___ " << endl;
    cout << " |___ / / _ \\ / _ \\ / _ \\ / _ \\ / _ \\ / _ \\ " <<
        endl;
    cout << "  |_ \\| | | | | | | | | | | | | | | | | | |" << endl;
    cout << " ___) | |_| | |_| | |_| | |_| | |_| | |_| |" << endl;
    cout << " |____/ \\___/ \\___/ \\___/ \\___/ \\___/ \\___/ " <<
        endl;
    cout << endl;

    string option;

    cout << "Choose an option:" << endl;
    cout << "1 - Image" << endl;
    cout << "2 - Video" << endl;
    cout << "0 - Exit" << endl;
    cin >> option;

    MENU_OPTION = atoi(option.c_str());

    if (MENU_OPTION == 0) {
        return;
    }

    cout << endl << "Filename:" << endl;
    cin >> FILENAME;

}

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

    menu();

    switch (MENU_OPTION)
    {
    case 1:
        imageProcessing();
        break;
    case 2:
        videoProcessing();
        break;
```

```
   default:
      return 0;
      break;
   }

   return 0;
}
```