Lane detection

Piotr Libera

Main task

The main goal of this program is to provide real-time coordinates of the edges of the road lane that the vehicle is currently travelling on. The program takes vehicle's camera feed as the input and calculates the road lane location. This information can be further used to precisely locate the vehicle on the road to provide proper steering adjustments.

To emulate the camera feed the program currently uses a prerecorded video. Certain assumptions were also made, e.g. stable lighting conditions as well as permanent camera location and orientation on the vehicle. This program also might require setting a proper region of interest if bright obstacles are present near the road.

Detailed working principle

The program uses OpenCV library to process images and videos. OpenCV's VideoCapture class is used to open and access frames of the input video. Every frame is processed in the same way. The current version of the program uses a configuration file, in which paths to videos and calibration data are stored.

After reading, each frame (a) is transformed using the data from the file. This transformation is supposed to transform the perspective of the image as if the camera was directly above the road, so to provide a bird's view of the current situation (b). The transformed image is then binarized using a constant predefined threshold (c). These steps are performed by BirdsEyeView class.

a) The original input frame

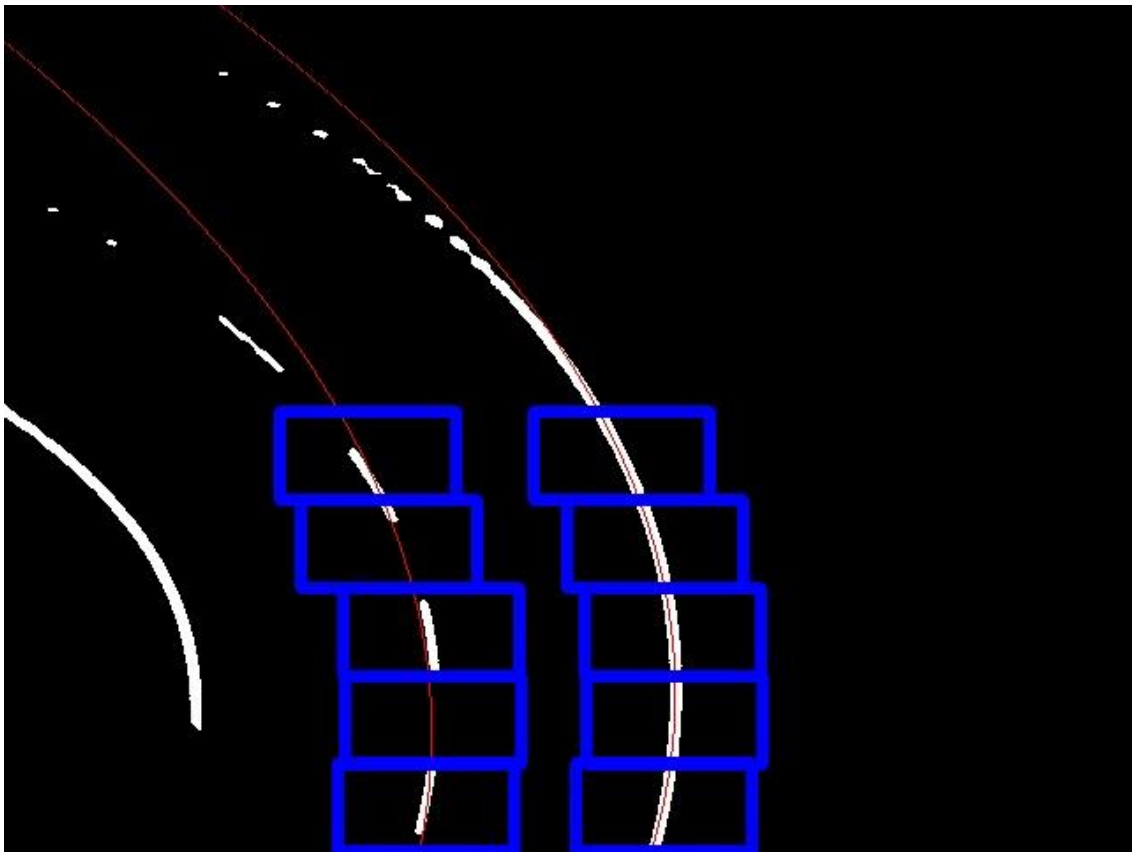
b) Transformed frame

c) Binarized transformed frame

The next step is to find the lines on the binarized image. If the threshold is set correctly, white pixels on the binarized image correspond to brighter pictures of the road lines on the original image. Next the sliding window method is used – the program chooses windows on the binarized image, which are rectangular regions of interest on the image, and calculates how many white pixels there are in each column of the window. The initialization and update methods work based on this method.

To properly find the road lines a general location of these lines must be known. It is found in the initialization method – the bottom half of the binarized image becomes a window, in which the number of white pixels in each column is calculated. The starting points of each new road line are chosen in areas of the highest density of white pixels, where the road lines should be found. Next steps use smaller windows of predefined size. Each starting point has a small starting window assigned. The window updates its centre to match the centre of the

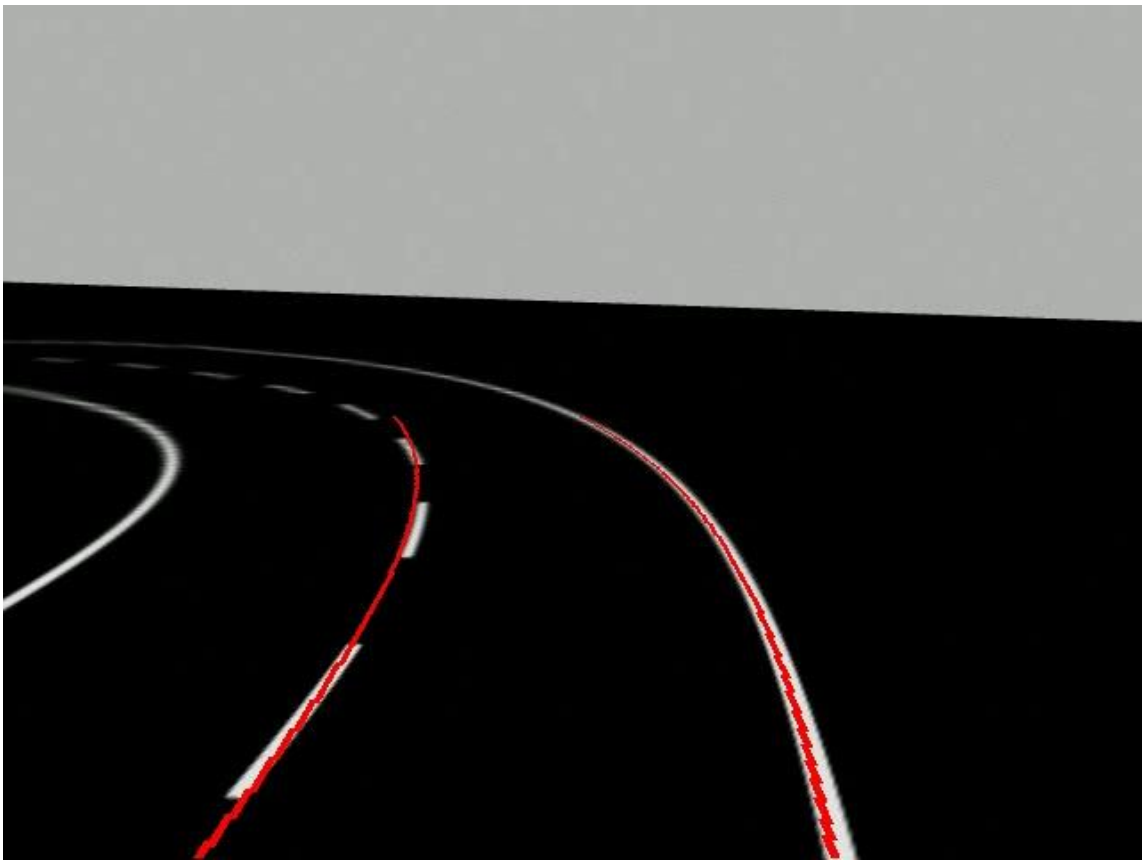
layout of white pixels in this window. After that a new window, further in the y axis, is created and the same operation is performed. After creating a predefined number of windows, each set represents an approximation of the road line, as shown in the picture.



An approximation of the road lines using sliding windows on the binarized image

The second method is responsible for updating previously created windows. These windows are roughly in correct positions and often require only minor adjustments, as this operation is performed for each frame of the input video. For each window the same operation is performed – number of white pixels in each column is calculated and the centre of the window is moved to match the centre of their layout.

At the end of each loop positions of the windows are approximated with a polynomial within every set to provide a mathematical description of the edges of the followed road lane.



Calculated road lane approximation with a second-degree polynomial on the original frame

Detailed description of classes

The computation in this project is a linear one, with almost every class and method using results taken from previous method, following the computation process described above. Hence there is only one case of inheritance and there are many classes providing independent functionality. In the following pages every class and its member functions are described in detail.

BirdsEyeView

The task of this class is to provide perspective transformations of the input image based on given region of interest.

Private attributes

cv::Mat src	input image
cv::Mat transformed	transformed input image
cv::Mat result	binarized transformed image
cv::Mat unwarped	input image transformed in the opposite way
bool inputSet	
FourPoints roi	region of interest used in the transformation

Private member functions

void performTransform()	Performs perspective transform from a rectangle on the input image to the given roi, binarizes the transformed image using the threshold function, using the defined THRESHOLD value

Public member functions

BirdsEyeView()	
BirdsEyeView(cv::Mat input)	Calls setInput(input)
~BirdsEyeView()	
void setInput(cv::Mat input)	Converts input image to grayscale and calls performTransform
cv::Mat getTransformed()	
cv::Mat getResult()	
cv::Mat getUnwarped()	
void setRoi(std::vector<cv::Point> roiCorners)	Sets new roi, roiCorners must be a vector of 4 points

FourPoints

This class stores sets of points of size 4. It uses a group representation for repeated sets, as the chosen points are often special and can be stored multiple times. It also controls if every point stays within defined boundaries, which in case of this project are the size of an image.

Protected attributes

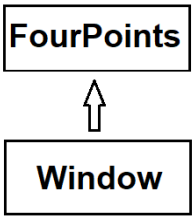
struct Points	Struct used for group representation
- cv::Point* point	Pointer to a set of points
- int count	Number of sets stored in this representation
- Points()	
- ~Points()	
std::vector<Points*> points	Pointers to stored sets of points
cv::Point boundary	It is assumed that coordinates of every point must be grater or equal than 0 and less than the corresponding boundary coordinate

Public member functions

FourPoints()	
FourPoints(cv::Point bounds)	Sets new boundaries
FourPoints(std::vector<cv::Point> input, cv::Point bounds)	Sets new boundaries and first entry
FourPoints(cv::Point a, cv::Point b, cv::Point c, cv::Point d, cv::Point bounds)	Sets new boundaries and first entry
FourPoints(const FourPoints& ck);	
FourPoints& operator=(const FourPoints& ck)	
~FourPoints()	
friend std::ostream &operator<< (std::ostream &output, const FourPoints &p)	
void push_back(std::vector<cv::Point> input)	
void push_back(cv::Point a, cv::Point b, cv::Point c, cv::Point d)	
std::vector<cv::Point> getPoints(int index)	
void pop_back()	
void clear()	
int size()	

void setBoundaries(int x, int y)	
void setBoundaries(cv::Point newBoundary)	
cv::Point getBoundaries()	
bool withinBoundaries(cv::Point a)	Checks if a is within currently set boundaries
virtual cv::Point polygonCentre(int index)	Returns the average of coordinates

Window



This class is responsible for storing coordinates for windows, as well as create histograms of binarized input images. Its methods require an image to perform operations on, hence every object must be constructed with one.

Private attributes

cv::Mat src	Input image, must be binarized
cv::Mat hist	Histogram of the white pixels layout
std::vector< std::vector<int> > histograms	Vector storing histograms for every window

Private member functions

void createHistogram(int index)	Creates a histogram for the window of this index

Public member functions

Window(cv::Mat image)	
Window(cv::Mat image, std::vector<cv::Point> input)	
Window(cv::Mat image, cv::Point a, cv::Point b, cv::Point c, cv::Point d)	
~Window()	
virtual cv::Point polygonCentre(int index)	Returns the centre of the rectangular window
void createHistograms()	Creates histograms for every window
std::vector<int> getHistogram(int index);	
std::vector<std::vector<int> > getHistograms();	
cv::Mat histToImg(int index = 0);	Creates an image representation of the chosen histogram and saves it to hist

void setWindowCentre(int index, cv::Point centre);	Moves the chosen window to match the new centre while staying inside of boundaries
void setWindowCentres(std::vector<cv::Point> centres);	Sets new centres for every window using setWindowCentre()
std::vector<cv::Point> getWindowCentres();	
cv::Point getWindowCentre(int index);	
void setInput(cv::Mat image);	

Polynomial

A template to store polynomials with various types of coefficients (denoted as class Type).

Private attributes

int deg	Degree of the polynomial
Type* coeffs	Coefficients of the polynomial

Public member functions

Polynomial(int d = DEGREE)	
Polynomial(const Polynomial<Type>& ck)	
Polynomial<Type>& operator=(const Polynomial<Type>& ck)	
~Polynomial()	
Type & operator[](int i)	
int degree()	
Type value(int x)	Calculates the value of the polynomial for given x

PolyFit

The task of this class is to approximate points given as input with a polynomial of a defined degree. It uses the Polynomial template to store results in.

Private attributes

std::vector<cv::Point> input	
Polynomial <double> output	

Public member functions

PolyFit()	
PolyFit(std::vector<cv::Point> in)	
~PolyFit()	
void setInput(std::vector<cv::Point> in)	
Polynomial<double> solve()	This function solves the optimization problem, saves it to output and returns it. It uses least squares method, formulates the normal matrix B, on which it then performs Gaussian Elimination and acquires the results from.

LaneDetect

This class performs and coordinates main lane detection. Its two main methods use Window objects to detect and follow road lines on the binarized input image.

Private attributes

cv::Mat input	Input binarized image
bool inputSet	
std::vector<Window> roadLines	Vector containing sets of Windows approximating each road line
std::vector<Polynomial<double> > lineFits	Polynomial approximations for every road line

Public member functions

LaneDetect()	
LaneDetect(cv::Mat in)	
void setInput(cv::Mat in)	
std::vector<Polynomial<double> > getLineFits()	
void initLines()	Clears both roadLines and lineFits and initializes following of every line. At the beginning crates a histogram of the bottom half of the input image using a window of that size. It analyzes the histogram and initializes first windows in areas with biggest density of white pixels. After that it creates smaller windows in these chosen areas and updates the centres of these windows to match the centre of the layout of white pixels. After updating a window it moves up in the y axis of the image and repeats the same operation until the full set of windows is created.
void updateLines()	It performs the same operation for each window as in initLines(), without destroying or creating any windows. It updates the centres of these windows to match the current centre of the layout of white pixels.