Università degli Studi di Padova
Facoltà di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di laurea magistrale

iCruise: development of a smartphone app for lane detection

Application intended to support CruiseControl devices

Candidato:
Alvise Rigo
Matricola 1013970

Relatore:
Prof. Luca Schenato

Anno Accademico 2012/2013

Nella nostra galassia ci sono quattrocento miliardi di stelle, e nell'universo ci sono più di cento miliardi di galassie.
Pensare di essere unici è molto improbabile.

Margherita Hack

✻

Everyday life is like programming, I guess. If you love something you can put beauty into it.

Donald Knuth

CONTENTS

LIST OF FIGURES

SOMMARIO

La sempre maggiore diffusione di dispositivi mobili, quali smartphone, e la sempre maggiore potenza computazionale in essi equipaggiata, rendono possibile l'utilizzo di questi dispositivi in ambiti sempre nuovi e interessanti. Il sistema operativo Android inoltre, che al momento della stesura conta più di 900 milioni di attivazioni totali, da la possibilità di esplorare questi nuovi ambiti realizzando applicativi complessi che possono interfacciarsi con una moltitudine di sensori, tra i quali la fotocamera. In questa tesi verrà presentata e descritta un'applicazione Android atta ad assistere un dispositivo di *CruiseControl* attraverso la fotocamera dello smartphone. L'applicazione curerà esclusivamente la parte di *lane detection*, ossia il riconoscimento dei margini della corsia che il veicolo equipaggiato dello smartphone sta percorrendo. Fondamentale per l'applicazione sarà quindi, grazie agli algoritmi presentati, l'essere conscia dell'ambiente circostante: compiere di fatto *computer vision* attraverso l'elaborazione delle immagini. La sfida del progetto, che nella sua categoria non rappresenta un novità assoluta ma una discreta novità se si considera la piattaforma per cui viene realizzata, è quella di realizzare un'applicazione il più snella possibile, che non necessiti di un quantitativo di memoria eccessivo e che possa offrire un'esperienza gradevole in fatto di fluidità e di rappresentazione visiva del grado di coscienza che il programma ha dell'ambiente circostante.

ABSTRACT

The continuously growing diffusion of mobile devices, as smartphones, and the always greater computational power equipped allow the employ of these for new and very interesting purposes. Moreover, the Android operating system, which at the moment counts more than 900 millions of activations all over the world, allows the developer to explore these new fields creating complex applications that can interact with many different sensors, first of all the embedded camera. Aim of this thesis is to introduce a new Android application for the assistance of a *CruiseControl* module through the camera equipped by the smartphone: in particular the part of *lane detection* will be covered. The application has to be able to place, in the most realistic manner, the car in which it is running inside the roadway boundaries using algorithms of *computer vision* created ad-hoc for the purpose. The true challenge of the project, that in its own area of interest is not a really new finding but a quite interesting discovery if we consider the platform for which it was developed, is to realize a very slim application, that has not to require a huge amount of memory and that can offer a direct user experience in terms of smoothness and visual representation of the awareness of the surrounding environment. As an introduction of the outlined application, this thesis will be focused on all the aspects regarding the development of the application and the algorithm of lane detection and vanishing point evaluation.

RINGRAZIAMENTI

Considero questa tesi il capitolo conclusivo di un lungo cammino di crescita e formazione che dal punto di vista professionale mi ha notevolmente cambiato, migliorandomi. Voglio ringraziare pertanto un po' di persone che sono state importanti in questi ultimi anni di magistrale, senza le quali probabilmente non sarei qui ora a scrivere queste righe.

In primis voglio ringraziare i *miei genitori*, sia per aver creduto nella mia scelta di studi "alternativa" mantenendomi per molti anni, sia per avermi consigliato e supportato.

Andrea, per la sua costante presenza interessata e curiosa della mia passione informatica.

Carlotta, per la sua costruttiva, onnipresente partecipazione a ogni passo della mia carriera universitaria.

Gloria, capace sempre di allietarmi i momenti difficili; saggia consigliera, senza di lei tutto sarebbe stato più tortuoso e meno divertente. Un grazie sentito veramente speciale per tutto, comprese le varie riletture di questo scritto.

Grazie ai miei *amici "di casa"*: *Mattia*, per le innumerevoli chiacchierate illuminanti scientifiche, *Monica ed Elisa*, per essere sempre vere amiche, *Margherita e Ciro*, per la vostra originalità e intrapprendenza.

Grazie ai miei *amici dell'Università*, che da buoni compagni si sono sempre mostrati disponibili ad aiutarmi, ma si sono fidati anche del mio aiuto e dei miei consigli che ho sempre cercato di condividere senza trattenermi: sono convinto che se li meritino tutti. Sperando di non dimenticarmene, ringrazio *Alberto, Alessandro, Angela, Daniele, Federica, Federico, Gianluca, Giulio, Ilaria, Lucia, Marco, Mattia, Rossella e Sebastian.* Grazie per tutto, vi auguro il meglio.

Ringrazio infine il *Prof. Schenato*, che mi ha dato l'opportunità di realizzare questo lavoro di tesi, concedendomi la giusta libertà di scelta.

Padova, luglio 2013 a. r.

INTRODUCTION

The purpose of this thesis is to introduce a software for Android smartphones capable of detecting the lanes inside the roadway and estimating the vanishing point in the scene that is nothing more than a picture coming from the camera of the device. For this work was used a HTC One X, equipped with Android operating system and a Tegra 3 processor.

This application is part of a bigger project called *iCruise*. The purpose of iCruise is to build a features-rich application which can offer assistance to a CruiseControl device installed in a vehicle. Essentially, the main lack of the commonly used CruiseControl devices is that they are not aware of what is happening outside the vehicle and more importantly, they do not know the distance between them and the following vehicle. This is why we devoted our efforts in the development of this application that will try to fix the mentioned hole. Actually, there are already available in the market solutions that offer a complete experience in fact of CruiseControl, however these implementations make use of sophisticated (and so expensive) radars that continuously monitor the vehicles in the road. These ones are indeed working solutions but too expensive and hardly portable. A smartphone application can, on the other hand, be very cheap and bring a rich CruiseControl service to everyone. Moreover, this application can take advantage from being open source, in such a way to let everyone to improve the code, making it safer and faster.

These kind of devices are called `adaptive CruiseControls`

RELEVANT CONTRIBUTION

As previously anticipated, the target device of the work we are going to describe is a smarthphone, with very limited computational capabilities if compared to a desktop PC. Firstly, an attempt to use the traditional methods has been made which however has led to not completely satisfying results. In particular an ad-hoc Hough transform has been implemented, but, although all the appropriate optimizations applied, the speed of the algorithm was too slow and to much gullible from the noise. Probably, adjusting the Hough transform properly, is possible to obtain a reliable algorithm for this particular employment, but still too slow. These are the motivations that led to the development of a new algorithm for line (lane) detection which is presented in the chapter 5.

The open source nature of OpenCV allows to freely read and modify all the methods code

In addition to that, a new work flow is presented for lane detection that is based on the reiteration of the same algorithm of line detection, enriched with some expedients.

THE CHALLENGE

Basically, this thesis is all about computer vision, a new field that, starting from the image analysis, tries to produce useful information from the picture. The computer vision is a complicated subject for the computer science that requires smart algorithms to emulate the human mind (as far as the human vision); sadly it turns out that the human mind is naturally inclined to perform some basic operations like edge detection, shape recognition an so on, all useful activities but difficult to realize through a computer. Moreover, even if we came to a brilliant algorithm for edge detection (for instance), we would soon face a sad truth when we will become aware of the huge amount of power required by the newly found algorithm. The nature of the target device makes the things harder, since it has not all the computational power available in a desktop PC, platform for which state of the art algorithms are already available. The application has to analyse all the frames that come from the camera, thus it has to work at about 20 frame per second which gives only 50ms to process the actual frame.

Even if in this work we will not talk about the final usage of this application, it is easy to understand the importance of getting robust and correct information about the road boundaries. However, there are a lot of difficulties to face that can affect the reliability of the program or that can degrade the performance of the lane detection. Some of those could be:

- `overall illumination of the scene:` obviously working at night is different from working during a sunny day: some of the useful details in the scene can be omitted by the camera. We have to keep in mind that the camera of a smartphone is far from being perfect: even in a shaded scene there could be a lot of noise in terms of colour fidelity. Also the sun position can lead to several problems; in fact the shade position and the brightness of the picture can vary greatly.
- `unexpected objects and occlusion:` one of the most relevant knots of computer vision is the appearance of unexpected pixels that can be simply related to noise but also to the unlikely presence of objects. For instance, a particular intrusive shade of a tree or a bag in the roadway, but also the legitimate presence of raindrops on the front windows of the car or the crossing of the windshield wiper. These are all examples that have not to compromise the output of the software; but its robustness has to prevent it. Occlusion, on the other hand, could also be problematic when, for instance, a car in front of the vehicle in which the software is running hide the part of the road needed to localize the lane.
- `noise:` it is unpredictable and can generate pixels that do not match with the neighborhood, bringing possible misunderstanding of part of the picture.

Usually, the colour variation brought by sensor noise is minimal in a sunny day, but get worse when the light decreases

This work will show, step by step, all the expedients found to achieve this result: some of these regard only technical choices (normal matter when developing), others on the other hand could be tricks inside an algorithm.

BRIEF DESCRIPTION OF THE FOLLOWING CHAPTERS

- `Chapter 1`
  In this chapter we will run through the state of the art in the field of lane detection and vanishing point estimation. The literature available is considerably wide so only the most significance contributions or those contributions that have been proved to be meaningful in the development of the project will be proposed. After that, will be presented the major innovations brought by this work.
- `Chapter 2`
  The main subject of this chapter is Android. Since it is not yet an operating system widely used for this type of employ, it will be presented very carefully,

starting from the operating system itself and than coming to its positive aspects and drawbacks. We will focus a little bit on a really critical aspect of this work that is the use of native code inside the Android applications.

- `Chapter 3`
  Now that the Android O.S. and the possibility to run native code have been introduced, in this chapter we will run through the Android SDK, the main tool to build Android apps, and the NDK, a development kit which can compile native code for Android devices. As we are dealing with computer vision, the OpenCV libraries will be introduced: these are the decisive libraries used in the project which also are, de facto, the standard in this context. The chapter ends with and overview of the OpenGL libraries and the description of the device where the application was tested.

- `Chapter 4`
  This application is made of various phases, that accomplish various sub tasks. In this chapter we will introduce the general work flow followed by the application and will also talk about the features expected by the program, because reliability imposes some constraints to the application. In the last part, the inner architecture will be presented in order to describe how the logical tasks are divided into running threads.

- `Chapter 5`
  In this chapter we come to the application, describing all its features. Here we will talk about problems and solutions faced during the developing process.

- `Chapter 6`
  A marginal chapter, which will not introduce new concepts but instead it will show some clarifying examples on how the knowledge of the vanishing point can make the whole work flow easier.

- `Chapter 7`
  The name says it all, here you find a review of all the work done together with some final thoughts.

1 RELATED WORK

This chapter wants to introduce other relevant works made to accomplish the job of lane detection. From now on we will mainly refer to this topic leaving the rest of the implementation of iCruise as future work.

There are various papers that show working solutions aimed to do lane detection for Autonomous Guided Vehicles by mean of a camera. It should be emphasized that most of this works do not deal with smartphones camera installed on unstable supports (like in the case of this work), but with steady cameras that can deliver better stability. This chapter wants to give an idea of the state of the art in this particular field, describing how our colleagues have achieved the same result. One of the most interesting method relies on the birds-eye view of the road, which is a sort of top-view image. For the reasons explained in later chapters, the images acquired by a camera are affected by aberration and prospective. This side effects can be nullified through a birds-eye view. A birds-eye view can make easier the detection of the lane, but imply more work for the processor, an issue that has to be handled carefully.

Other works have taken a different path believing that it is worth to describe the progress of the road with high precision. These works, like Jung and Kelber, 2012, make a differentiation from the near field (the portion of the lane which is near to the vehicle) and the far field. In picture 1 there is an example of such a division of the road. While the lane boundaries of near field can be always be considered like straight lines, the lane boundaries of the far field on the contrary could be described as parabolic lines when a curve is approaching. The cited work presents a very precise way to do that which relies on choosing the right coefficients for both the lines (that approximate the line boundaries) in the near field and the parabolic curve in the far field. Obviously, the straight line and the parabolic line have to be consecutive. For this reason, in the conjunction point, the continuity condition and the differentiability condition must be satisfied. The whole problem, at the end, turns in minimizing the error carried by the near and far field lines as approximation of the lane with the conditions of continuity and differentiability satisfied.

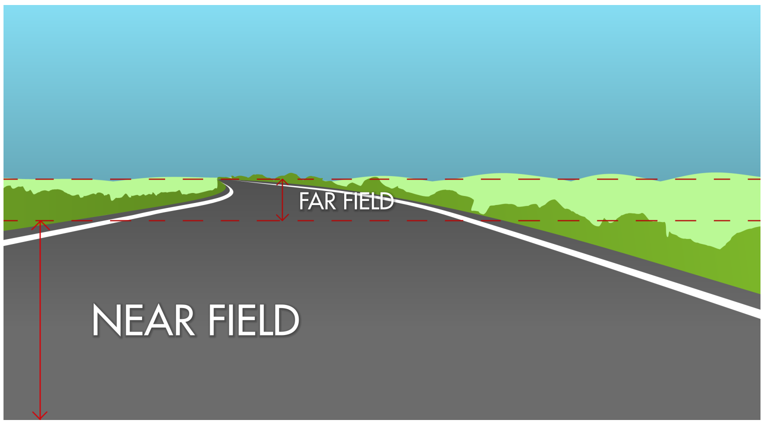

Figure 1: Near and far field distinction

The work proposed in King Hann Lim and Ang, 2012 presents another interesting work flow for lane detection that can be summed with these steps:

- `horizon localization:` to eliminate the disturbances from the sky segment the horizon localization is performed to partition the scene in two sectors; it is likely that the sky segment occupies a great part of the image, so it is worth to remove it and to benefit of a smaller image. To do so, a vertical meaning distribution is computed by averaging the gray values in every row on a blurred image. Plotting the resulting graph, it is possible to evaluate the exact point in which the switch to the sky colour happens. This is a really common approach that does not deserve a deep analysis.

- `lane region analysis:` in this step the road region is estimated performing some ad-hoc evaluations on the color of the image. The main purpose of this step is to eliminate from the scene all those pixels that belong to the road, in such a way to empathize the white pixels of the lane markings.

More on this in the mentioned paper

- `adaptive line segment:` exploiting the mathematical properties of the resulting image, it is possible to have the precise position of the lane markings. Initially, the gradient magnitude (∇D_{map}) and the orientation (θ) are computed starting from the horizontal edge map (D_x) and the vertical edge map (D_y). The relationship between this quantities are:

$$\begin{aligned} |\nabla D_{map}| &\approx |D_x| + |D_Y|, \\ \theta &= \tan^{-1}\left(\frac{D_x}{D_y}\right) \end{aligned} \tag{1}$$

  and are useful to define an *edge distribution function*, also known as EDF. The authors decided to divide the road in n different rows, where the first $n-1$ from the bottom represent the near field, the nth instead the far field. After obtaining θ and ∇D_{map} for each row, looking at the maximum ∇D_{map} in the negative and positive angles, it is possible to estimate the position of the right and left markings of the lane. In the picture 2 we clearly see the two tallest peaks, that refer to the two lines edge. The more that pixels are closer to the horizon, the bigger is the amount of space that they represent, analogously, the closer we are to the horizon, the smaller the lane will be. For this reason the noise, which is uniformly distributed in the whole image, has a bigger impact in those pixels close to the horizon. This a natural drawback which affects not only a camera, but also all human eyes; the authors of the cited paper decided to deal with the problem adopting the already seen distinction between far field and near field. In the near field the EDF technique is actually a working solution, but in the far field they developed another one called *"the river flow edge detection"*.

- `river flow edge detection:` in the far field, the edges are not clearly predictable and they are deeply affected by the noise. The idea is simple and is really close to what the water of a river does: it follows the bank of the river. This is what the river flow algorithm actually does: it follows the lane marking choosing as “bank pixels” those with higher intensity.

These presented works show indeed interesting approaches, but are too complex for our purpose and, somehow, too much elegant. In fact, the purpose of this project does not require to distinguish when a curve is coming and, moreover, does not require to show to the user that the curve is actually coming. What on the contrary is really needed is the right positioning of the lane where the vehicle is running and the right outlining of the lane area in which it is worth to check for

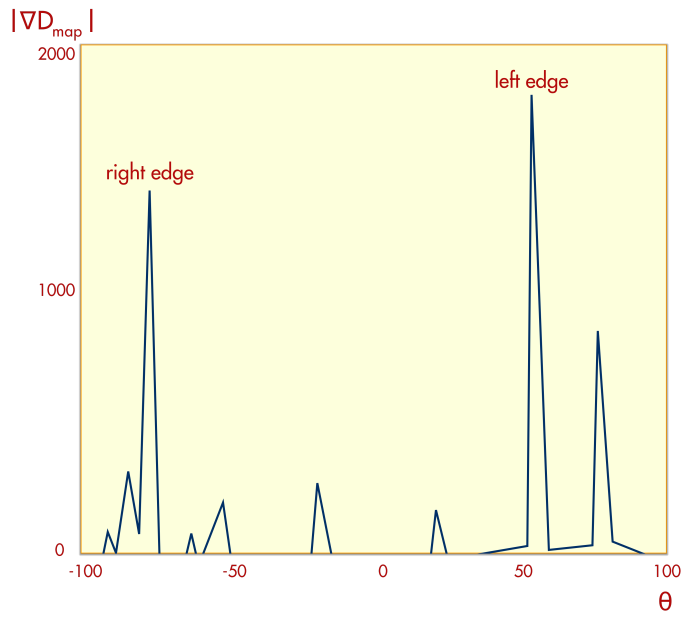

Figure 2: Example of EDF for edge recognition

some possible excessively near objects (vehicles), as, for instance, a vehicle that has slowed down suddenly. In order to meet these requirements, it is not needed to have a detailed description of the far field and we can approximate merging the near field with the far field using a linear description of the lane boundaries. Clearly, when this approximate model has to manage a narrow curve, the outcome will not be precise, but this is an unlikely case, negligible if we consider the real use cases of a cruise control application.

THE LINE DETECTION PROBLEM In any of these possible approaches there is a common feature that is the line detection. The lanes in the roadway are delimited with white lines that are easily distinguishable thanks to the darker colour of the rest of the road. This almost always present characteristic allow us to exploit it using one of the available algorithms for line detection. Actually, there are not single and “all inclusive” algorithms for line detection, but a set of common approaches that allow to distinguish lines inside an image. In fact, an algorithm such the Hough transform (which is one of the most used for line detection) is not decisive when applied to a noisy image without taking care of distinguishing the relevant borders of the picture through, for instance, a Kenny edge detector. All the efforts present in the literature show slightly different techniques to distinguish the lines of the lane and, of course, this work will present its own method for line detection.

A general work flow is now necessary to accomplish the final goal: a robust algorithm for line detection represents only the first step to accomplish the output. In fact, many analogous works proceed as follow: in the first frame the algorithm of line detection is applied in order to know the precise position of the lane (or the lanes); then in the other frames, an algorithm of lane following is continuously run to adjust the position of the lanes previously found. The clear advantage is that, once known the approximate position of a lane edge, in the future frames it will be possible to reduce the region of interest of the picture into a smaller one allowing a drastic reduction of computational time. More on this on chapter 4.

1.1 STEREO IMAGING

Stereo imaging is used by human brain to compute distances.

Even if it is not used by this work, it is worth to spend some words to present how an alternative work flow based upon the stereo imaging can make the things easier. The stereo imaging expects that the scene is acquired by mean of two cameras, one next to the other; in this way it is possible to have a picture of the scene taken exact at the same time but from two positions that differ for few centimetres. In essence, the steps to cover to obtain useful information from the stereo imaging are:

1. Radial and tangential lens distortion removal. The output of this step is an undistorted image.

2. Rectification: this is a process that adjusts the angles and distances between cameras.

3. Find the same feature in the right and left camera view. The output of this stage is a `disparity map` which tells, in term of pixels, the distance that characterizes the correspondence.

4. At the end, knowing the relative position of the two cameras, we can position each feature inside a 3-dimensional scene through triangulation. This step is called reprojection.

The third step is the key point of the stereo imaging, suppose that the same feature is acquired in the first camera at the coordinate (x_1, y_1) and in the second camera at the coordinate (x_2, y_2). Assuming that previously we have successfully fulfil steps 1 and 2, then we have an undistorted and rectified image and we can reasonably argue that $y_1 = y_2$. We can't state the same for the x value which has to be different because of the mutual position of the two cameras. For this reason $x_1 \neq x_2$ and, more precisely, $|x_1 - x_2|$ gives a quantification of the distance of the feature from the cameras. Sadly, although this approach gives very precise results, it has two drawbacks:

- it requires a lot of computational power, due to the number of steps needed;

- it requires two cameras, an impressive requirement for a smartphone (actually, such a device exists and probably within few years a smartphone with two cameras will be very common; but at the moment we have to be satisfied with only one camera).

2 ANDROID OS

Android is an open source software stack that includes, above all, the operating system, a middleware for the communications and a set of core applications, along with a set of API libraries for writing applications that are meant to be really integrated with the operating system and that respect its own style and patterns. Android is, with some exceptions, an open source software, making it very attractive for developers and in general for those curious people who likes skimming the code for fun. Android initially was developed upon both the Linux kernel 2.6 and the typical C libraries used inside every GNU/Linux operating system, making it an OS open source and with a strong base. Although the early versions haven't been really exciting (even more so if compared to the competition), the latest versions offer to the developers a flexible developing platform along with an outstanding user experience.

ANDROID, A GOAL OF THE OPEN SOURCE SOFTWARE Android was originated by group of companies known as the Open Handset Alliance, led by Google. All this companies believed that an open source platform is really necessary and so have addressed their efforts to this cause in order to bring new devices to the market equipped with this operating system. Android is intentionally open source, and not *free* "as in freedom" Free Software Foundation, 2007a: it is a shared effort carried by a group of organizations with shared needs. There are a lot of companies that are born with the purpose of building Android applications or devices. The possibility to have the source code available made the developing process quite agile and, in few years, it was possible to see great devices entering the market of smartphones. However, what probably has stricken the most are all those communities of developers and enthusiasts arose around, basically, the released source code. Now, after five years, there are entire ecosystems based on branches of the Android source code like, for example, the CyanogenMod community. Nowadays Android is probably the most concrete contribution of the last years in terms of innovation for the open source community.

Android is a major step towards an ethical, user-controlled, free-software portable phone, but there is a long way to go.
—Richard Stallman

WHY NOT A COPYLEFT LICENSE? Good question. First of all it is worth to introduce some concepts. There are basically two type of free license:

- *free, copy left licenses*: an example could be the GPL license [Free Software Foundation, 2007a] or its newer, less strict, version LGPL [Free Software Foundation, 2007b]. This kind of licenses allows everyone to use the code with no limitation. Every developer or company who wants to use it to build an application, can do that without any limitations, but, once he decides to publish it, he has to use the very same license that initially protected the code. As a consequence, anyone can, in turn, use that code to build an improved version of the application and release it. It is said that the source code been copied *infects* the software developed upon it.

- *open source, not copy left licenses*: as not copy left license we can mention the ALv2 [The Apache Software Foundation, 2004]. The software covered by this license, as open source, has to have its code freely available together with its binary file but it doesn't have any particular restriction for any future redistribution. This means that any company can actually use the code in its own manner, releasing it with any license. Since all the Android framework and userspace applications are released under ALv2 license, all the device manufacturers can greatly take advantage from it, who can build a firmware that fits perfectly with the device. While this opportunity encourages the proprietary distribution, it discourages the opportunity to bring more openness to the mobile software scene, allowing these companies to close their enhancements and to perpetuate their aggressive commercial strategies.

Google has his own motivations to believe that GPLv3 license would have dramatically reduced the manufacturers interested in developing under Android; considering that is up to Google to choose the commercial politic, we have to live with his choices. Luckily, the Linux kernel (which is a fundamental component of the Android operating system) is benefiting from all this success as it is covered by the GPL license; when porting a software to new devices or architectures, some adjustments are always necessary that for the reasons mentioned before, have to be really free.

2.1 DESCRIPTION OF THE OS

Android is made up of several necessary and dependent parts; we can sum up the most important with the following list:

- The Linux kernel that provides the low level interface with the hardware, memory management, scheduling of processes and all the basic activities of an operating system.
- Open source libraries for software development, including SQLite, Webkit and OpenGL, necessary to build great applications starting from a solid, already working, base.
- A software layer used to run Android applications, including the Dalvik Virtual Machine and some other libraries that provides Android-specific functionalities. The Dalvik Virtual Machine is a modified version of the Java Virtual Machine (JVM[1]), greatly optimized for mobile devices.
- A set of pre installed applications; most of them are used to control Google services, like Gmail or the PlayStore.
- The user interface framework, to control the applications.

The Dalvik VM is a modified version of the original JVM, the name comes from the birth place of the main developer: Dalvik is an Iceland's village

The engineering team inside Google has decided to build this "hybrid" operating system for one fundamental reason: the *code portability*. As we will later see, this particular Android architecture (and its license) allows to have many different devices running the same operating system with the same applications, no matter which hardware architecture they are running on. The key of all this flexibility is the choice to build all the top level applications using the Java programming language, and of course, the adoption of a virtual machine. Any program written in Java could not be executed directly on the architecture of the machine, but on top of a particular machine, called virtual machine, which will execute the source code translated in the so called *bytecode*. There is no hardware architecture that execute entirely this code, but a virtual one that in Android is called Dalvik Virtual Machine.

1 Java Virtual Machine

2.1.1 Briefly analysis of Android components

In this section we will go through the analysis of the main components of the Android OS. As an open application stack, it is possible to find in the Internet a lot of resources that describe more precisely the OS. For instance, the site Android Developers Site is a good starting point.

2.1.2 The Linux kernel

This is the bottom layer upon which it is build all the Android ecosystem. The Linux kernel provides all the basic functionalities of the operating system, allowing memory virtualization and caching, processes scheduling, I/O management and takes care of all the needed communications with the peripherals using their drivers. If we would find the component of the Android architecture that is really platform-dependent this is the kernel and all its drivers; they have to be compiled for the hardware in which they'll run. As mentioned before, while a Java process has not to be compiled for the architecture of the device but for the Java Virtual Machine, this does not hold for the kernel, which has to strictly interact with the hardware.

2.1.3 Libraries

Some computer libraries provide really useful functions, which are often used by the processes to fulfill basic services like the encryption of a password or the rendering of a 3-dimensional object. All these libraries are not included in the high level application framework of the Android architecture, but instead are placed right at the top of the kernel. Sometime the choice to keep the existing and working code instead of writing new one has to be made and in this case Google's engineers decided to keep the available code to fulfill these basic services. If some of those processes could not have been implemented upon the Android framework (like, for instance, the libc lilbrary), some others could actually have been, but building them using efficient C code leads to the most achievable performances. The direct drawback is, again, that these services have to be compiled properly for the processor architecture making them harder to deploy in a variety of different devices. Significant example of native libraries are:

C code can take advantage of the most optimized compilers

- `Surface Manager`: provides the compositing functionalities to the OS
- `SQLite`: this is the main database provided by the Android framework. If an application needs to store some data that has to be consistent and permanent in all its execution, a light database like SQLite can fulfill this requirement. The Android framework that we will discuss in later chapters, offers a lot of useful methods that makes the interfacing process agile.
- `OpenGL`: nowadays the smartphone is a device for almost every purpose, gaming included. Mostly the high-end devices are equipped with powerful graphic cards integrated right inside the main chip (in fact, almost every device has all the functionalities integrated in one chip; in fact we speak about SoC[2]. In order to use all the available power of the GPU[3] (the components that provides the graphic capabilities), the OpenGL libraries have to be used, giving to the developers some primitive method to draw 3-dimensional object in the screen. In the chapter 4 we will see how these libraries have been used for the project.

2 System on a chip
3 Graphic Processing Unit

2.1.4 The Dalvik Virtual Machine

This is one of the main components of Android. The Dalvik Virtual Machine is derived directly from the Java Virtual Machine, but it is optimized to run in multiple copies on top of mobile devices. This is clearly an advantage if we think at the laying operating system that sees many similar processes and doesn't have to worry about some technical difficulties like the dynamic memory allocation or the simultaneous coexistence of Java thread. Unlike the Java Virtual Machine, the Dalvik Virtual Machine doesn't run the .class files but .dex files; the conversion between .class to .dex happens in compilation times and provides a reduction in terms of memory weight.

UNLIKELY ALTERNATIVE TO AVOID THE DALVIK VM Does it mean that an Android application is meant to run only inside the Dalvik VM[4]? Actually no, we can always build native application if we desire, but only in some particular cases, where the performances are crucial for the application. For example, most of the games are build not as processes of the Dalvik VM but as native processes that are executed directly by the processor. As we will see, the project uses this unlikely alternative to obtain maximum performances.

2.1.5 Application Libraries

Together with the VM, Android provides some useful libraries to accomplish more application-specific functions, like the ones that we can find in the standard Java library. The whole library is available easily on-line at the web address *Android Reference Library*. Note that all these libraries are written in Java because are meant to be used for Android applications inside the Dalvik VM.

2.1.6 Applications and Application Framework

At the top of the Android stack we have applications and the framework with which they interact. An application running inside the Android OS can take advantage of some services useful to create rich applications; these services can offer the possibility to fix the actual position by means of the GPS or can let the application to receive incoming messages. All these services build the so called Android Framework that, in simple words, is the soil where the application will live and the inspector which will control their lifecycle and their mutual interactions.

4 Virtual Machine

3 DEVELOPMENT ENVIRONMENT

In this chapter we will analyze the development environment used to build the software. In addition to a brief description of all the components that the environment includes, we will focus on the possibility to use native code inside an Android application. Actually, this is not a likely case, but when the performances are a fundamental requirement (like in this project), then also the most difficult development alternatives are welcome. At the end of this chapter the device in which the code has been tested is introduced together with its most relevant characteristics.

3.1 SDK

The Android SDK[1] provides all the tools needed to build, run and debug Android applications. The most convenient way to develop applications for Android is using Eclipse, a very popular IDE which embeds a lot of useful functionalities to simply the development process. The key component of the integration of the Android SDK and Eclipse is the ADT plugin, which somehow links all the SDK features right into the IDE, providing:

- `Graphical UI builder`, a tool that simply the drawing process of the user interface. The UI is meant to be build through an .xml file, which defines all the elements of the interface and how they are positioned in the screen. This tool automatically compiles this file leaving to the developer only few adjustments.

- `Various debugging outputs`, like memory analysis, OpenGL tracking and other useful information.

- `Virtual devices`, in which we can easily deploy the application and automatically run it. Even if this option is really helpful, it hasn't been used, for obvious complications regarding camera availability and performances.

The development process can also take advantage of some facilities of the target device that can display real time information about CPU usage, layout bounds and graphical activity of the OpenGL stack.

3.2 NDK

As the SDK, the NDK[2] is also a development kit, but is intended for slightly different purposes. The fundamental requirement of speed for this application leads to the critical choice of using native code to write the algorithms involved in the detection of lanes, a very heavy work load.

1 Software Development Kit
2 Native Development Kit

NATIVE CODE When we talk about the Android architecture we made distinction between the Java Android application framework and the rest of the system: bottom libraries and Linux kernel. While the first requires Java code compiled for the Dalvik Virtual Machine, the second runs directly in the architecture of the device. Obviously, passing through a virtual machine leads to additional overhead because the code has to be interpreted by the VM and then translated on-the-fly for the processor. This passage can sometimes be exploited to gain a boost in the overall performances, however it is an unlikely case which is not inherent to our discussion. The NDK allows to build C and C++ code for the CPU, but, although this possibility can improve the performances of the software, it brings also some drawbacks. The development will in fact get complicated since the developer has to use a whole new development kit with its rules and that hardly integrates with the normal Java code. Moreover, if we think that the last ARMv7 architecture can fetch, decode and execute some of the Java bitecode, probably we will dismiss the possibility to use native code.

SO, WHY USE THE NDK? Computer vision is a new branch of the computer science and for this reason there aren't many libraries that provide the implementation of basic filters or famous algorithms; but, for sure, we can say that the most mature and widespread library for computer vision is OpenCV. OpenCV libraries, particularly devoted to performances, are written in C (and starting from the version 2.0 also in C++), and not in Java code. If from a certain point of view this is clearly a limitation, from another is a great advantage. The obligation to write and compile code in C/C++ (the latter in our case) allows the developer to test all the algorithms on a desktop pc, solution faster and convenient with respect to the one offered by a mobile device.

All the tests were made offline on a laptop

The main mechanism which allows to run native code inside an Android application is called JNI[3].

3.2.1 JNI calls

The JNI is a vendor-neutral interface that permits the Java code to interact with the laying native code. It also has support for loading code from dynamic shared libraries. All this translates to particular Java methods that, in turn, call the native code.

DEVELOPMENT WORK FLOW The really simple work flow used for the code development can be described as follow (the setup of the development environment is excluded):

1. The computer vision algorithms have been written on a desktop PC equipped with the OpenCV libraries installed. After a successful compiling, the code has been tested right on the PC, taking advantage of providing the algorithms with some videos recorded off-line. Those videos had the purpose to simulate with high fidelity a real time acquisition of the road by the camera.

2. Once that the algorithms satisfies the requirements that will be mentioned in the chapter 4, it is possible to proceed to compile the code using the NDK for the specific architecture of the device.

3. After assembling the application (this time using the Android SDK), it is time to test it on the road and, in case, restart from the starting point.

3 Java Native Interface

3.3 OPENCV LIBRARY

OpenCV is a C++ library which is, de-facto, standard when dealing with computer vision. This library offers many useful, state-of-the-art, methods for image processing, image visualization and other specific functions commonly used in computer vision. In the most recent versions they also added the compatibility with the Android operating system, inserting some java bindings that facilitate the work of integrating OpenCV code inside Android applications. As we already pointed out, we didn't take advantage from this possibility choosing a quicker approach that consists on using native code.

Figure 3: OpenCV Logo

3.3.1 Tegra

Since nowadays OpenCV is often used to build smartphone applications, this library recently included a new interesting feature which allows to make faster applications. Tegra is a new brand created by Nvidia to indicate all the new SoCs produced by the company; it is like a particular architecture, ARM based, with marked capabilities of OpenGL rendering. The Tegra SDK offers in fact a convenient way to build fast applications that makes intense use of 3D operations. Tegra, moreover, allows to use the computational power coming from the GPU to accomplish task not necessarily of 3D computation. This is why OpenCV exhibits a closed source implementation of certain methods which probably comes from NVIDIA rather from the OpenCV community which are actually faster than the original version. Since these particular implementations are already compiled and integrated inside the OpenCV library package, an application can automatically use the optimized version of a method when running on a Tegra device. Our target device is Tegra capable, but since we mainly use custom code, we exploited the Tegra enhancements only for basic operation of filtering.

3.4 TARGET DEVICE

There is not much to say about this device except that it is equipped with a Tegra processor. Here we have the precise specifications of the device:

O.S.	Android Jelly Bean 4.1.2 with HTC Sense 4
SoC	Nvidia Tegra 3
CPU	1.5GHz quad-core ARM Cortex A9
GPU	Nvidia Geforce ULP
RAM	1GB
Storage	32GB

For more information refer to Wikipedia - HTC One X, 2013.

Figure 4: HTC One X, the testing device

3.5 OPENGL ES LIBRARY

The way we chose to display the frames of the camera and the highlighted lane makes use of OpenGL ES library. This is the fastest solution because all the workload is demanded to the GPU, reducing the CPU utilization to a minimum level. The CPU time in fact is really precious in this context and if there is a way to subtract some of the work from the CPU, this way is welcome. Nowadays the GPU is fundamental even in the 2D operations: starting from Android 4.0, the UI is almost entirely drawn by the GPU, making all the animations very smooth and fluid. Going back to our application, the screen of the smartphone is considered like a face of a parallelepiped and the picture of the road is a simple texture applied to one face. Looking at the screen we will have the impression to look to a simple 2D image, but this is not our case: we are looking instead to a static 3D object (with a variable texture) through a static "window".

When dealing with pixels, the GPU can be decisive in terms of speed

Figure 5: OpenGL ES logo

4 TASKS AND WORK FLOW

In this chapter we describe the main ideas underlying the process that, starting from a simple image of the road comprehensive of lanes boundaries (lanes markers), returns the exact position of the boundaries of the lane in which the vehicle is running.

Purpose here is not to illustrate in details how the code actually works, but simply to draw the basic operations that the code implements.

4.1 EMPHASIZING MARGINS

First of all it is necessary to point out all the lanes boundaries. Luckily, the lane marker is always painted in white, with rare exceptions where the application is not needed. The application takes advantage of this characteristic and emphasizes it thorough an algorithm (an image filter) that can find the contours inside an image. The literature commonly refers to these type of algorithms as *edges detectors*, as the very famous *Sobel Filter*. All these algorithms use various techniques that can require the application of multiple filter in order to come at the final result; in this project a custom chain of filters has been used and will be described in the chapter 5.

4.2 LANE DETECTION

4.2.1 Searching of straight lines

After that the contours are clearly visible, it is possible to go further with the localization of the lines inside the scene. We do so because the segment of lane marker closer to the vehicle (which will be in the lower part of the image) can be always approximated by a straight line. So, if we find a straight line in the lower part of the image which is characterized by a certain tilt angle, then we are almost sure that a lane boundary has been found. As in the previous case, there are various algorithms that suite for this purpose: first of all, the *Hough Transform*. The Hough Transform is an algorithm commonly used for line detection and it can be found implemented in the OpenCV library; it is available in two different implementation: in addition to the classic one, there's also a probabilistic implementation which should offer more performances. However, as we will see, none of the two has shown itself particularly suitable; this is why we opted for a custom algorithm.

LANE DETECTION Once that we have found all the relevant lines inside the scene, we are ready to calculate and position the lanes inside the roadway. This is the final task of the application, and, if the previous task was completed successfully, it comes quite easily: it's all about right estimations starting from the available information.

4.3 THE VANISHING POINT ESTIMATION

4.3.1 outline on perspective

The perspective can be easily nullified by mean of a bird-eye view; this generates additional work

When we acquire an image of the real world with any type of camera (or simply with our eyes), we have to tacitly agree with the underlying physic optical rules and so with the perspective. In the perspective model every set of parallel lines in the real world are translated into a pair of lines converging at an infinite distance from the camera to the vanishing point. This is exactly our scenario, where all the straight lines that define the road are parallel and, in our simplified model, can be considered infinite long and so converting to the vanishing point. While this could be a complication (because search for parallel lines inside an image is much easier than find incidental ones), it can be also an advantage if we consider where are positioned the lines we are searching.

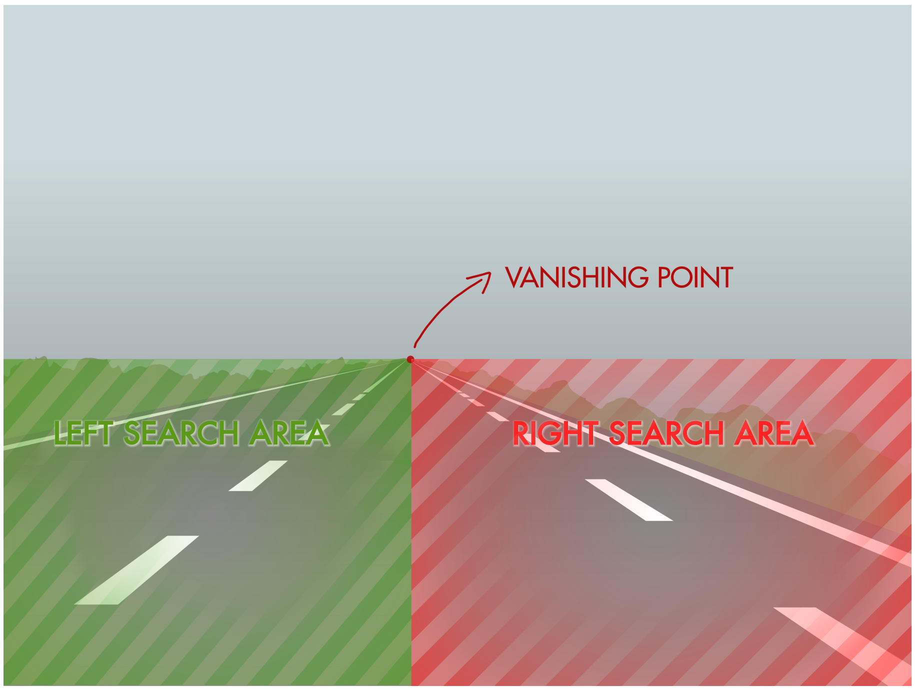

Figure 6: Example of left and right search region

4.3.2 benefits derived from the knowledge of the vanishing point

If we look at the image 6 we can clearly see how can we take advantage of knowing the position of the vanishing point: the vanishing point can be considered as the top right angle of a rectangle with the bottom left angle overlapped to the bottom left angle of the acquired image; this rectangle will be the region where lines with tilt angle $\in \left(0, \frac{\pi}{2}\right)$ lay. We call such region *left search area*. Similarly, the *right search area* is the region with all the lines (of our interest) which are tilted with an angle $\in \left(-\frac{\pi}{2}, 0\right)$.

In a mobile context, a trick like the one just explained is very appreciated because can significantly reduce the amount of work load for the CPU; but we can do more using also the position of the vanishing point to validate all the lines we are going to find. If a line, at the infinitum, does not converge to the vanishing point, it will be considered as a false positive and so will be discarded. In the chapter 6 will be described the process of vanishing point estimation, while, in the following chapter we will see how we took advantage from it.

4.4 GENERAL WORK FLOW

In this section is outlined the general process that, de facto, originates the whole application. The process is explained in the following points (more details on these will follow).

1. `acquisition of the picture`
   The camera continuously acquire picture of the scene. A thread, inside an infinite cycle, grabs the image from the buffer of the camera and copies it in a class variable in order to share it with the rest of the application. This particular task is independent and has maximum priority because the image acquired has to be the more updated as possible.

2. `lane detection`
   Once that we are provided with the most updated picture of the real scene, we are ready to run our algorithms. But firstly we need to prepare the picture to be easily readable.

   a) *image filtering* is a crucial task because it allows to get rid of all the unwanted information that are present in the picture. It can also enhance it removing most of the noise produced by the camera. The key purpose of this phase is also to return a binary image where every dot with value greater then zero could actually be considered part of a line.

   b) now the *line detection* algorithm comes to the scene to find all the correct lines that can be associated with a lane margin. As previously anticipated, this algorithm is not present in the literature and can be considered as a valid alternative to the famous concurrent methods.

   c) With *lanes estimation* we intend the phase in which, starting from the exact location of all the possible lines inside the image, we have to understand where the lanes are located.

3. `vanishing point estimation`
   This task is meant to find a good approximation of the vanishing point position and is the last part of the work flow with logic intents. The reason why we should find the position of the vanishing point and the benefits deriving from this were presented earlier and will be remarked in the proper chapter.

4. `output of the result`
   This is a not critical task, but it deserves however attention. The possibility to see directly from the display the output of the previous phases is really useful when it comes to test the application, and maybe debug it. There's no more direct way to verify the correctness of an algorithm than seeing its output "graphically". An interesting aspect of this phase is the way used to display the picture. For the purpose of relieving the processor of an additional burden, we decided to let the graphic card manage all the needed to display the image in the screen of the smartphone. In order to do so, the OpenGL graphic library was used; more on this in the proper section 3.5.

4.5 THE PROBLEM OF LENS DISTORTION

The scene acquired by the camera is far to be exactly the same as in the reality; it is in fact heavily distorted because taken using no perfect lens. The optic of any kind of camera is one of the main factor that determines the quality of the coming picture. Also in the best reflex in the market equipped with the best lenses we have lens distortion; obviously, in a smartphone that usually has very cheaper lenses we have distortion too, a lot more. The reason of such distortions is mainly for manufacturing: it is much easier to make spherical lens than to make a more ideal lens. This limitation in the manufacturing process leads to imperfect lenses that suffer mainly of two distortion: the *radial distortion* and the *tangential distortion*.

- `radial distortion`: it affects the acquired image in the pixels near the edge. It's the cause of the usually called "fish-eye effect". In the context of our application, this type of distortion did not lead to severe problems. Our fear of acquiring parabolic lines instead of straight ones hopefully did not happen.

- `tangential distortion`: it comes when the sensor of the camera is perfectly parallel to the optical lenses. This type of distortion leads to pixels that are displaced with respect to the position they should have in an ideal image. As for the other distortions, we did not have to correct it because its effect were negligible.

4.6 FEATURES EXPECTED FROM THE SOFTWARE

In this section we will illustrate some of the features that the lane detection algorithm has to satisfy. The main problem when executing is an unexpected situation, where the software has to deal with unlikely states of the processed image. The three key elements that negatively impact in the reliability of the application are:

- `light`
- `shadows`
- `unlikely objects`

The practical effect that these elements lead to is unexpected straight lines inside the roadway. We will see that is by mean of (straight) line detection that the application computes the position of the lane: having accidental lines inside the road that don't belong to the lane boundaries could lead to a wrong behaviour of the lane detection process.

4.7 APPLICATION ARCHITECTURE

The work flow can result, as it was presented, linear where each phase waits the previous one to execute. However the application was designed a little bit more complex in order to be more flexible and configurable. There are two threads that run inside the application: to the first is demanded the acquisition of the images from the buffer of the camera; it is an easy task, which has only to copy arrays of pixel from a location to another. The second thread, on the contrary, is a more complex and is the responsible for all the tasks of lane detection. These threads run independently from one another, but at a certain time they must access to the same location in memory; to avoid any bad misbehaviour, a semaphore controls the interaction of the two thread with the shared variables. In picture 7 is clearly visible the architecture, with specified the two types of operations that are performed by the threads.

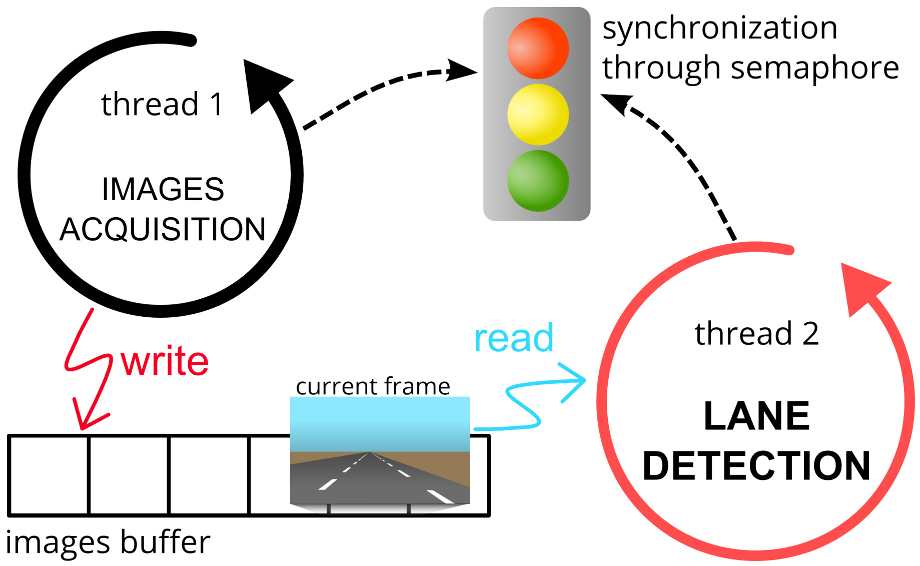

Figure 7: Internal architecture of the application

5 LANE DETECTION

In this chapter the problem of lane detection will be deeply analysed with particular attention on the implementation of the developed software. For the sake of completeness a wrong attempt will be also introduced, in order to don't let somebody else to make the same mistakes. It is important to remark that all the code, working or not, was developed using the OpenCV library in its C++ implementation. It is really possible that, following different leading paths or using a different version of the OpenCV library, could lead to better results and maybe turn what we called "wrong attempt" into a "successful attempt".

5.0.1 a wrong attempt

With all the preliminary remarks mentioned before, we are going to illustrate this first attempt which was declared as not completely working. In this first attempt, we tried to follow the already working examples that the literature has to offer (we mentioned at them in chapter 1). In particular, we tried to used the Hough transform already available and implemented in the OpenCV library as `HoughLinesP`, defined in the header file `opencv2/imgproc/imgproc.hpp`.

HoughLinesP makes use of fixed point operations to improve performances

Prior the phase of line detection we prepared the image using some filters that we also used in the working solution illustrated in section 5.1. As the exact same filters have been used, we are not going to illustrate them now, but we relegate them to the mentioned section.

5.0.2 the reason why it doesn't work for lane detection

As every algorithm meant to be used in a general context and for general purposes, this is an example of a really well implemented algorithm ready to work in many situations. For example, it does not offer a way to fix the angle which will characterize the found straight lines; obviously, using this algorithm in its original behaviour and then discarding the useless lines (the ones with an improper angle) is not a feasible solution because computationally excessive. Actually, an attempt has been made in order to adapt the algorithm for the purpose of line detection (and then lane detection): the main idea was to provide the algorithm with additional parameters to confine the research only in the desired angles. The source code of the function `HoughLinesP` was deeply revisited and at the end reached a working stage. But, although it was working well, it was too slow.

PROVIDE THE HOUGHLINESP WITH THE RIGHT ANGLES The main lack of the implemented Hough transform method was the impossibility to specify the range of angles in which the lines we are looking for have to belong. A naive approach can, at the end of the searching activity, discard the unwanted lines in favour of only those that rely on the angles range; obviously this is something that we can't afford. Convinced that the Hough transform was the right way, we tried to provide the method with other parameters through which specify the angle range. Sadly this try was unsuccessful for the main reason that the code, born to be used in a slightly different manner, was not suitable for so much changes. Discarded the Hough transform, a new algorithm for line detection was written.

5.1 THE FINAL SOLUTION

Now that we introduced all the necessary subjects, we are ready to jump into the central part of the software, showing for each step its output.

5.1.1 image filtering

As previously mentioned, the image considered as it comes out from the camera buffer, is rich in details and colours, all information that are not always meaningful for our purposes. Consider for example all the information carried by the pixels representing the sky, the are all meaningless, except for the fact that they are bluish and that surely don't contains the road lanes.

Another element which is insignificant is the colour of the pixel: we don't care which is the exact colour of a roadway pixel: it could be lighter or darker than the next one, without any particular reason. As we already said, the chip which constitutes the camera is not perfect and for this reason it is very likely that objects in the real scene with the exact same colour will be acquired with different tints.

Even if these slightly differences of colour can be overcome easily, we decided to address this problem not considering it at all. The only derivatives along the two axis x and y have been enough to fulfil our needs.

applying linear filters

When we talk about digital image processing we are actually referring to linear filtering and so to image filters. A linear filter, in the digital era, can be easily applied performing the convolution of the image with the kernel which expresses the linear filter. The kernel is a simple matrix, which usually is squared and characterized by an edge with odd number of coefficients. It also has an *anchor point* which is usually located at the center of the matrix. In order to apply a filter to an image, a process called *filtering convolution* has to be done. The aim of this procedure is to update all the pixels of the image with a new value (which has not to be different) according to the application of the kernel to that pixel.

Usually, bigger kernel means better results and slower performances

Suppose that we want to apply the kernel

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & \mathbf{0} & 0 \\ -1 & 0 & 1 \end{bmatrix} \tag{2}$$

to the image with the following pixels values:

$$\begin{bmatrix} 50 & 200 & 189 & 12 \\ 40 & 223 & 200 & 10 \\ 60 & 205 & 212 & 24 \\ 53 & 210 & 201 & 11 \end{bmatrix} \tag{3}$$

The image 8 is a simple example of vertical straight line in a grey scale image where the second and third column of pixels represent the lines; we can see it in the following picture.

The process of convolution is really simple and is dictated by this equation:

$$H(x,y) = \sum_{i=0}^{M_x-1} \sum_{j=0}^{M_y-1} I(x+i-a_x, y+j-a_y)K(i,j) \tag{4}$$

where:

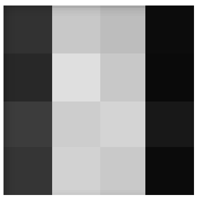

Figure 8: Image of a vertical line

- I is the source image which we want apply the filter to.
- H is the new image which is going to be created; it's the output of the filter.
- K is the kernel matrix.
- i and j are respectively an index over the columns and rows of the filtering kernel. In our case $M_x = M_y$.
- (a_i, a_j) is the coordinate of the anchor point inside the kernel matrix. Usually the anchor pixel is the central one; look at the kernel matrix 2: the anchor point is bold.

Refer to OpenCV 2.4.5.0 Online Documentation for additional details of linear filtering with OpenCV.

The application of the mentioned kernel to the picture of the line through the process of filtering convolution leads to the following result:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 13 & 7 & 0 \\ 0 & 0 & 14 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{5}$$

Once again, it is an image with the same dimension of the original one, but the pixels values are drastically changed. The result image shows that, where the original picture contains high-valued pixels near low-valued ones, then the same pixels in the filtered image (result image) will have a value that is barely visible, but the surrounding low-valued pixels instead will have a zero value.

5.1.2 margin emphasis

Essentially, what we are calculating is an approximated derivative of the image which will emphasize the margins of the picture. Probably the previous example is not really relevant, but the following example better conveys the idea.

Fortunately, although we are dealing with a really simple kernel, it works really well when used for lane detection: the characteristic white of the lane boundaries over the dark grey of the road help the effectiveness of this kernel. Note that the images were firstly converted to a grey scale; considering the colours present in the road, starting from an image with 3 channels would have brought to an almost 3 times slower execution of the filtering process without any additional benefit.

IMPLEMENTATION DETAILS The kernel used to implement the derivative filter allowed us to execute the convolution of the image in a faster way than the normal convolution. In fact, it turns out that:

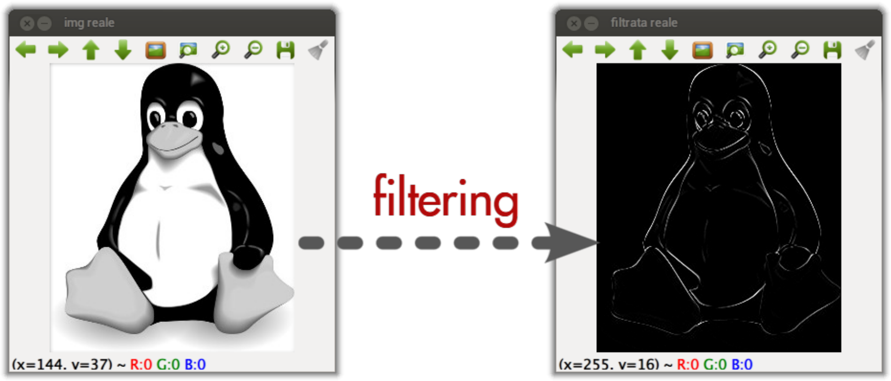

Figure 9: *before* and *after* the application of the kernel

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} \tag{6}$$

basically, it means that we can apply the filter independently to each row and each column to obtain exactly the same result. OpenCV offers an easy way to apply 2D filters to an image, even in the case they were separable like in our case. The function responsible for this task is called `sepFilter2D`. It takes as input the source image that we called I, the destination image H and the kernel for rows convolution K_x and the kernel for columns convolution K_y.

5.1.3 noise reduction

The task of noise reduction plays an important role in the whole project. Even if the previous filter gets rid of a huge amount of noise in the scene, there are still a lot of "bad pixels" that can compromise the phase of line detection. The three devices we use in order to address the problem of noise reduction have been:

The most used averaging filter is probably the gaussian blur, however, for our purposes, a simpler version is enough

1. *blur filter*, this is a very common approach to noise reduction. A blur filter can be implemented in many ways; in this case we opted for a very simple solution: the filter in fact averages the value of the pixel at the position (x, y) with the values of surrounding pixels. From a logical point of view, the operation principle here is the same we saw for the derivative filter, only the kernel is changing. The kernel K_{smooth} used for smoothing the image is:

$$K_{smooth} = \frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{7}$$

It basically takes the pixel value under the anchor point and averages it with the nine (eight if we exclude the current pixel) surrounding values. The aim of such a filter is to *smooth* the aliasing of all the borders in the image, making them more uniform for the following operations. In addition to that, it can also weaken all those pixels that are isolated (in groups of few pixels) because not part of a relevant margin. When the threshold filter will be explained, the smooth filter will be adequately justified.

2. *erosion* is the second filter used to address the problem of noisy pixels. Its principle is again similar to the convolution of the image with a kernel, but this time the equation which determines the output pixels is different:

$$H(x,y) = \min_{(x',y')|K_{erode}(x',y')\neq 0} I(x+x', y+y') \tag{8}$$

The minimum value among the pixels in the neighborhood of the computed pixel (x, y) is taken as new (x,y) value. The neighborhood is defined by the not-null values of the eroding kernel. This process gets rid of all the isolated pixel, in group of 3 members maximum. For our purposes, the kernel which suited the best has been the following:

$$K_{erode} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{9}$$

3. *threshold filter*, the most simple filter: it removes all those pixels that have a value under a predefined threshold (15 for this project) and set the remaining ones to a fixed value. This basic filter permits to construct a *binary* image, where all pixels not blank have the same value.

Now that we introduced all the used filters (that are applied to every image coming from the camera sensor), we can appreciate them in actions with the following image.

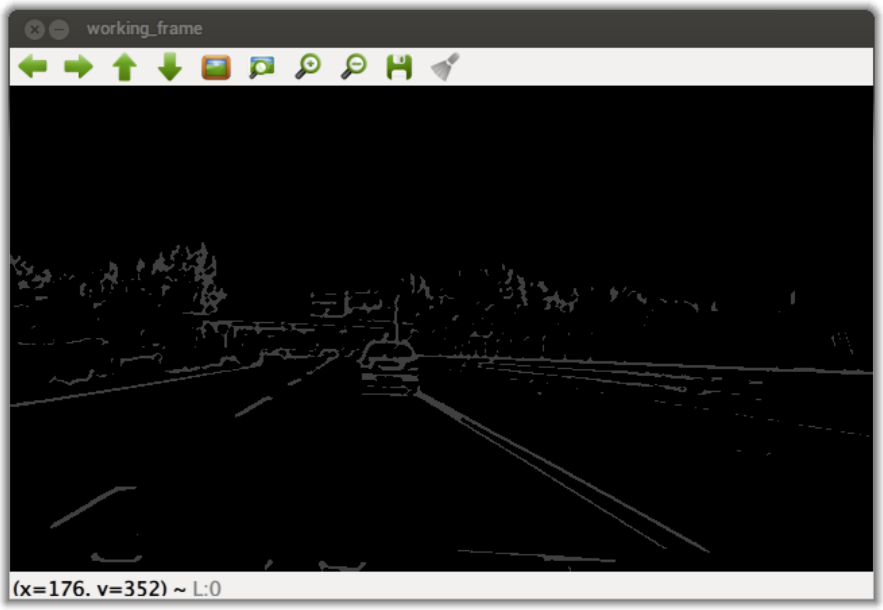

Figure 10: Example of filtered image

The most important thing to notice is the homogeneity of the road: it has no pixels that are not black; as a binary image it actually doesn't give any information about the roadway, except for the position of the lane boundaries. This fact is really helpful for the next phase which has the purpose of detecting the lanes position.

5.1.4 line detection

The next phase takes advantage of the algorithm of line detection described in section 5.2. This method does not alter the image which analyses because it creates all the "buffer" images that are needed and all the required data structures. The output is nothing more than an array, which tells the exact location of the found lines. The algorithm is not smart enough to find only those line that are part of a lane mark and, for this reason, an additional phase is needed that can distinguishes from this set of lines which ones are actually relevant.

5.1.5 lane estimation

Once that the straight lines are found, the estimation of the lane position is an easy job

Now consider the image 11: the lines that have been found are red and every line has the starting point (green point) marked and the end point (blue point). As we can see, there are both false negative as, for instance, the discontinued lane marker, and false positive, as the two lines found over the car. One key point of the software is that it is robust to such false results; it knows that, if there is no line detected at a certain point, it could likely be that the line hasn't been detected in the current frame for accidental reasons. For the same reason, it is smart enough to quickly update the position of the line margin when a new line is found in a more proper position. This case is represented by the figure 12, where all the lines found by the line detection algorithm (we can see them in figure 11) don't suggest the presence of a lane boundaries in the middle of the roadway.

As soon as a line in the middle of the roadway is found, the software updates the lane margins to reflect the new information. See figure 18 and 20 in the next chapter to see the updated position.

The logic that lies below the estimation of lanes position is quite simple and doesn't deserve more attention. The most relevant thing to notice is that all the information about the lane position are not built from scratch at every frame, but are progressively updated with the analysis of the upcoming frames.

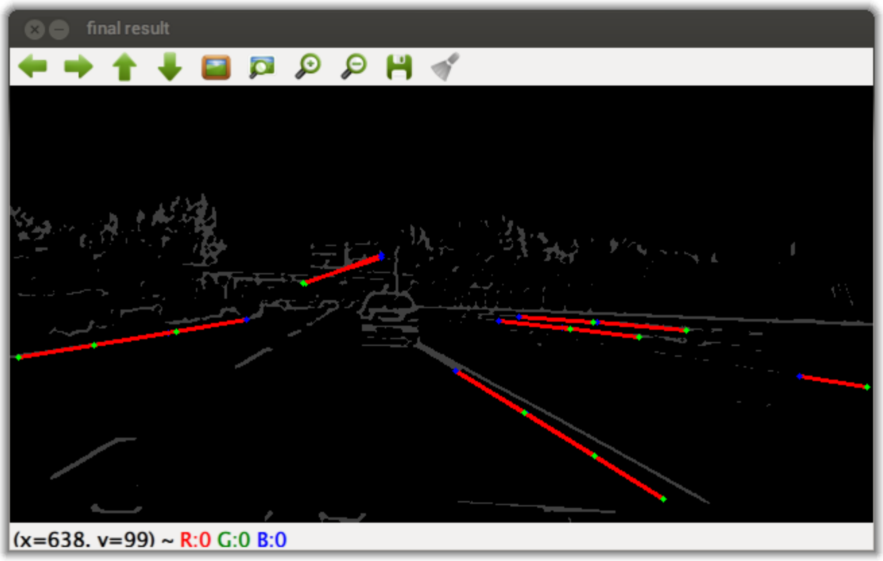

Figure 11: Output of the detection line algorithm *before* the vanishing point estimation

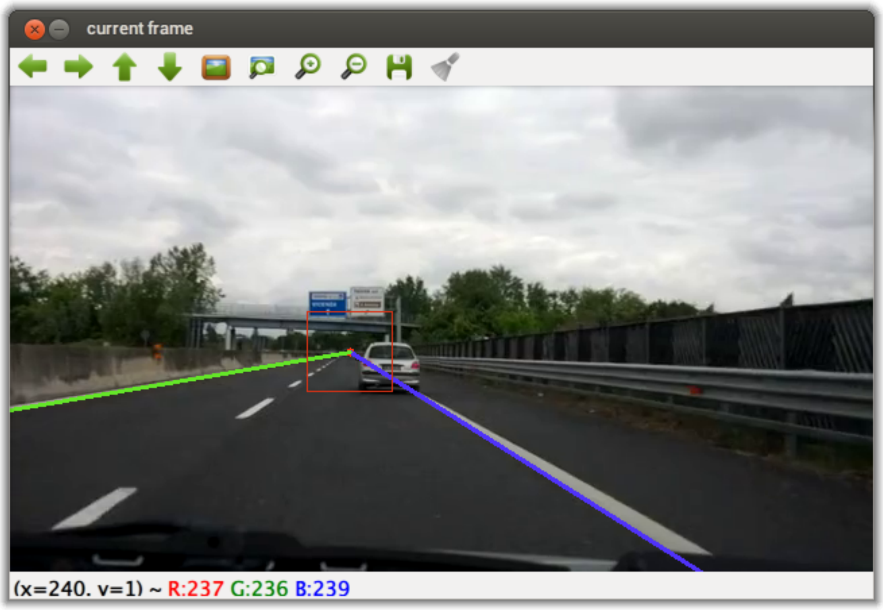

Figure 12: Final output of the work flow *before* the vanishing point estimation

5.2 A CLOSER LOOK TO THE LINE DETECTION ALGORITHM

This section is entirely dedicated to the algorithm used for line detection. As stated in the introducing part of this chapter, we also tried other alternatives before creating a new one, but they were all too complex (and so slow) to execute in a smartphone.

Now we will run through the most important steps of the algorithm.

5.2.1 line creation

Every pixel of the image is crossed starting from the center of the last (the bottom one) row in a bottom-up fashion. We divided the search of the lines in two different and independent regions as shown in figure 6; consider that at the beginning the location of the vanishing point is not known and the middle of the image is used as limit of the two regions. We will illustrate only a left search because the right counterpart is identical. The figure 13 conveys the idea of “left search”.

This step has the main purpose of detecting lines that are likely to belong to a lane mark. The algorithm tries to attribute all the pixels he finds to the line which it most likely belongs to. If there’s no probable line then it creates a new line. So, every time that the algorithm has finished to analyse a row of pixel, it will have either updated the previously found lines with a new pixel or it will have created new ones. The picture 14 tries to draw this process.

This algorithm is basically a cycle over all the rows of the image. At the end of every row, it checks the correctness of the lines that have been found and removes those who not satisfy some requirements. This is why for each line a counter is kept that can tell which is the last row after which it has had no more updates.

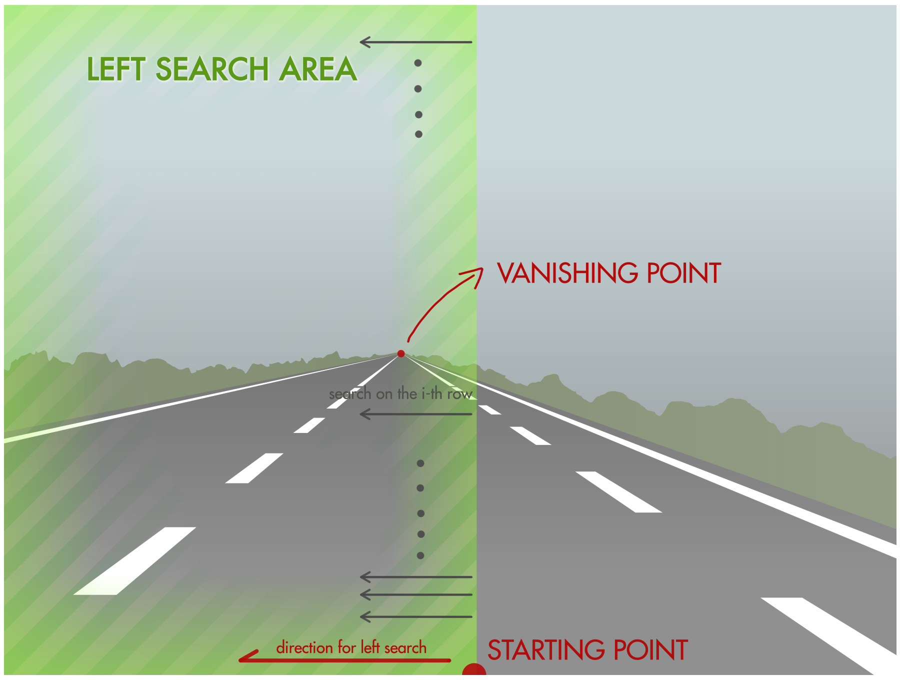

Figure 13: example of left search

5.2.2 line removal

The previous step, if executed as is, without any particular expedient, would be too slow and would not suite its purpose for the huge amount of detected lines. This is the reason why at the end of every analysed row, the algorithm checks whether there are some lines that are not meant to be found, for various reasons:

- The line is not enough long and, even if there will be some pixels that can belong to this line (because aligned with it), the gap formed of blank pixels would be too big. This check is performed looking at the counter previously mentioned.
- The line has no way to satisfy the length requirement because it starts close to the last available row and has too many blank holes.

The image 15 sums up the procedure of line removal.

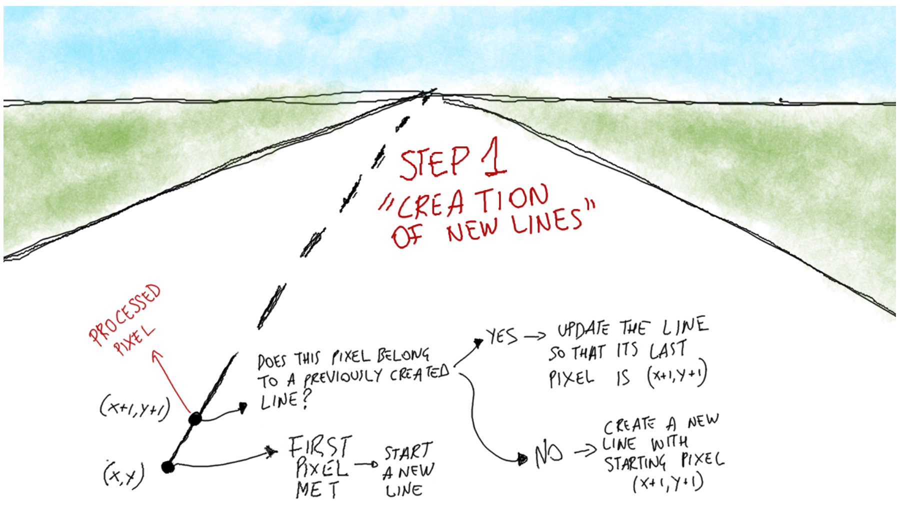
STEP 1
"CREATION OF NEW LINES"
PROCESSED PIXEL
(x+1,y+1)
(x,y)
DOES THIS PIXEL BELONG TO A PREVIOUSLY CREATED LINE?
YES → UPDATE THE LINE SO THAT ITS LAST PIXEL IS (x+1,y+1)
NO → CREATE A NEW LINE WITH STARTING PIXEL (x+1,y+1)
FIRST PIXEL MET → START A NEW LINE

Figure 14: First step: line creation

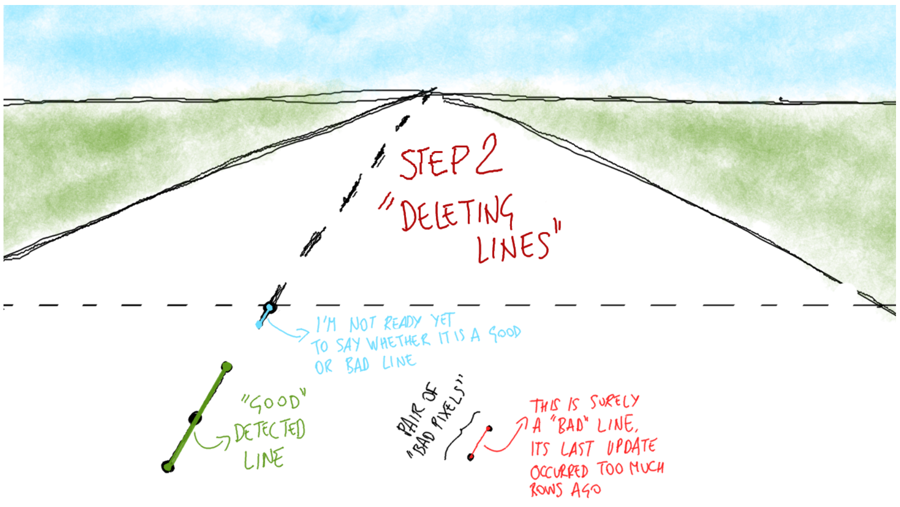
STEP 2
"DELETING LINES"
I'M NOT READY YET TO SAY WHETHER IT IS A GOOD OR BAD LINE
"GOOD" DETECTED LINE
PAIR OF "BAD PIXELS"
THIS IS SURELY A "BAD" LINE, ITS LAST UPDATE OCCURRED TOO MUCH ROWS AGO

Figure 15: Second step: line removal

THE DIAGONAL CHECK One of the most annoying problem in the first stage of the algorithm is illustrated in the figure 16. Basically, in order to add flexibility to the algorithm various tricks and threshold have been used. The threshold `max_y_allowed_gap` for instance sets the maximum x-distance (in pixels) between two consecutive pixels. While this threshold works in most cases, in some else can lead to an unwanted behaviour depicted in the mentioned figure. We can clearly see how the first five pixels can actually be attributed to a straight line, but, the more we get far from these pixels, the more the detected line changes deeply its tilt angle leading to an erroneous line. To address the misbehaviour, when the line that is going to be detected is enough long, the *diagonal check* is performed. This check goes through the last pixel of the line toward the first and counts all the non blank pixels: if the counter stops to a low value, then is most likely that the line is not properly straight and so has to be discarded.

The diagonal check is only one of the checks made

MAX_Y_GAP As for the x axis, there is also a max allowed gap that regards the y direction. The lines detected are not required to be continuous along the y axis (there could be some row of the image that hasn't got any pixel of the detected line), again, this is a matter of flexibility: as we already said the noise plays an

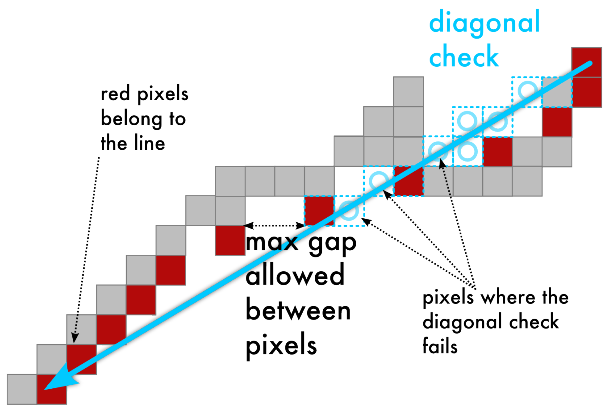
diagonal check
red pixels belong to the line
max gap allowed between pixels
pixels where the diagonal check fails

Figure 16: Practical example of diagonal check

important role here and can seriously frustrate the successful detection of a line. But there should be a limit, in fact the threshold `max_y_gap` defines the maximum size of this "hole"; if this limit is exceeded, the line is deleted.

6 VANISHING POINT ESTIMATION

Once that we calculated a first estimation of the lane, calculating the position of the vanishing point is not that big deal. In fact, the vanishing point is actually the point to which the two markers of the detected lane converge. Its position can be very helpful to increase the performances of the whole process and can also enhance the phase of line detection.

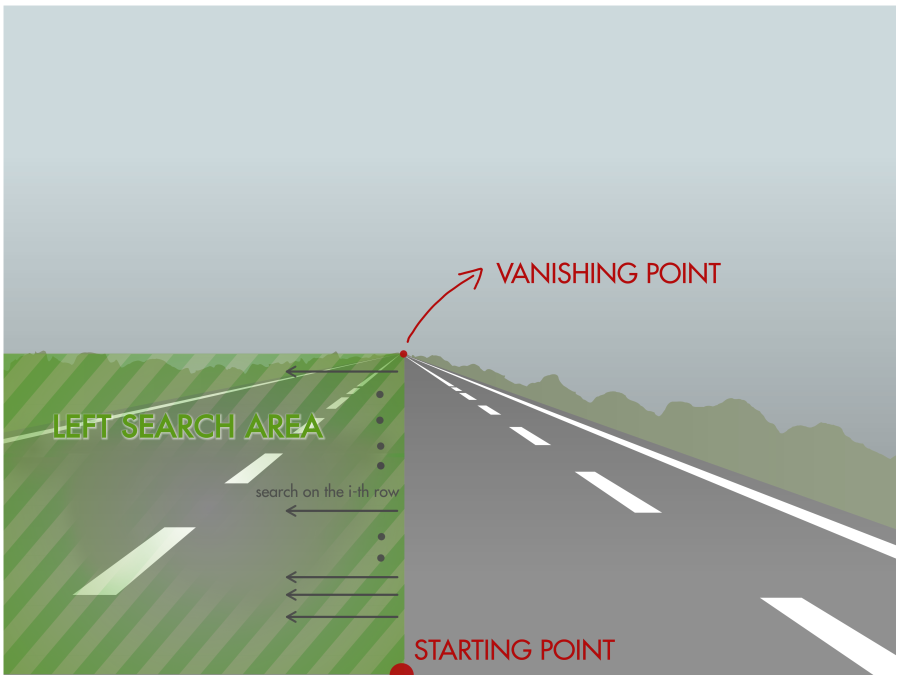

Figure 17: Left search phase *after* the vanishing point estimation

The filtering process can really take advantage from the location of the vanishing point. It is not necessary, for instance, to filter the part of the image above the vanishing point, because no relevant line can lay in that region. For this reason, the output of the filter chain is a bit different as conveyed by the picture 18.

The vanishing point is not entirely updated on every frame, but a mean of the last twenty computed values is kept in order to make it stable

Regarding the line detection, knowing the location of the vanishing point leads to two main advantages:

- the search area is drastically reduced (for instance all the sky area is untouched),
- all the resulting lines are much more accurate because it is possible to know the precise tilt angle of each line.

In fact, as can see from picture 17, the search areas are defined in that convenient way where the left search is the search of all those lines that converge to the vanishing point from the left and, in the other way around, the right search is meant to find all the lines converging to the vanishing point from the right.

As for before, it doesn't worth to search straight lines in the sky but it does worth to stop the search right below the vanishing point.

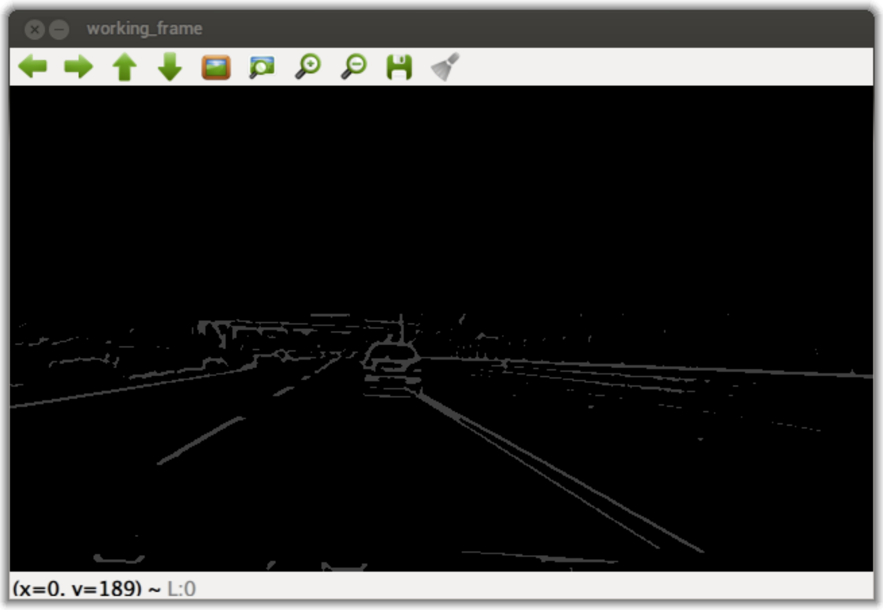

Figure 18: Filtered image *after* the vanishing point estimation

The above considerations clarify why there are no lines detected over the vanishing point and why the image is so narrow in the picture 20. Its size proves that the rest of the image was actually discarded.

Not surprisingly the final output shows the entire image because all the computations of the lane margins are finished and the software draws the boundaries in a complete image in order to be more user friendly as possible.

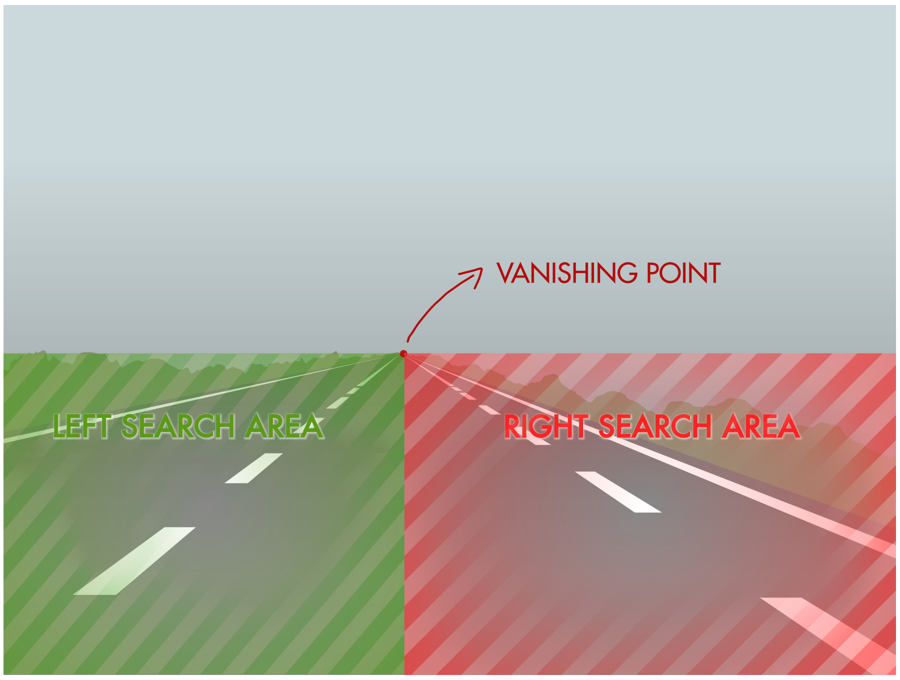

Figure 19: Two search region distinction

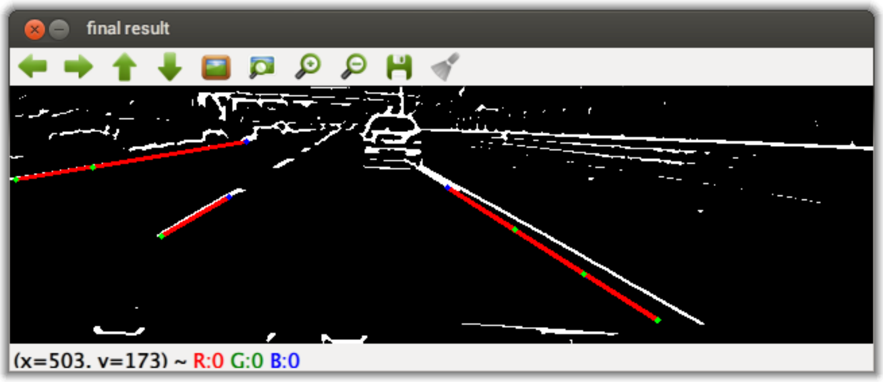

Figure 20: Output of the detection line algorithm *after* the vanishing point estimation

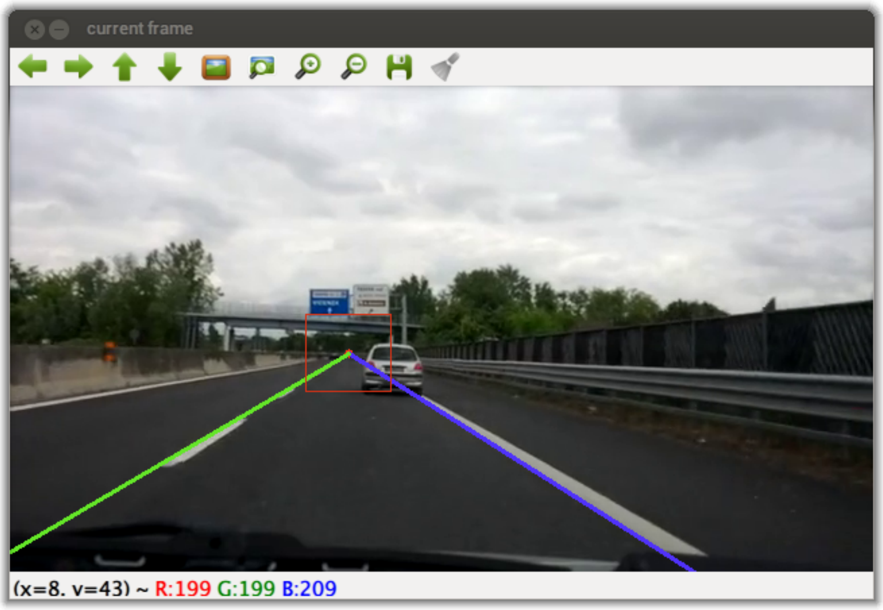

Figure 21: Final output of the work flow *after* the vanishing point estimation

7 | CONCLUSION

The purpose of this thesis was to present an Android application for lane detection. This work is actually part of a wider project called *iCruise*, which wants to provide an all-inclusive service for the assistance of cruise control devices. There are already devices (usually integrated in new, high end cars) which provide a rich cruise control experience; they, in addition to maintain the speed of the vehicle, deactivate the automated control when the following car is too close to the vehicle. However all these features have a cost which can be imputed to the high cost of a radar that is the mean by which the device "sees" (or better, is aware of) the road.

The real challenge of this work can be summed with the following question: *can we make an alternative to these expensive devices, which can be cheaper, portable and that can of course fulfil all the needs of a features-rich cruise control device?* More roughly we asked: *could we use the camera of a smartphone as eye of the cruise control device?* This was the starting point of the whole project, and the first part of the answer was to use a smartphone as target platform. The further choice to use specifically an Android device was dictated mainly by the tools chosen for the implementation of the logic part of the software. In fact, the OpenCV library are becoming particularly suitable for developing Android application; they can, for example, take care of the acquisition of the output of the camera through a proper class which interfaces with the low-level library. OpenCV are clearly far to be perfect, but the work that the community is doing is amazing and the help you can get from the developers around *OpenCV questions* is amazing too. Since the Android support of the OpenCV library is something new, we can't really expect a better library and the fact that it is completely open source makes it really interesting. The availability of the source code can, in addition to broaden the programmer's horizon with some really well written algorithms, provide you with an initial code to build your own: this was in fact what was done with the Hough transform, which was therefore discarded as technique of line detection. Going back to iCruise, the first step of the project was to decide all the needed features of the software; this is how we came to the *lane detection*, which is the task of this thesis. With the lane detection we want to make the software aware of the position of the vehicle in the context of road transit; since the roadway is logically divided into lanes, then we want to be able to position the lane in which the vehicle is running. The biggest advantage which a working lane detection brings is a drastic reduction of the search area for possible vehicles that can be dangerous for the vehicle carrying the cruise control device. As we already mentioned, if the following vehicle comes too close, we need to deactivate the automated service.

The OpenCV class responsible of managing the interaction with the camera is still a work in progress. For example, prior a certain update, it wasn't working in the target device

Pondered various alternatives, we decided to use the following tools:

- The Android SDK was used to build the Android application, comprehensive of all its callbacks methods to fulfil the Android application lifecycle.
- The Android NDK allowed us to embed the application with C++ code with the main purpose of making the computational load faster. The additional requirement of integrating native libraries (like OpenCV indeed) made this choice a forced one.
- OpenCV libraries to implement all the algorithms.

- OpenGL libraries to display the frames and to highlight the lane boundaries.

As for the implementation, we configured the whole process in the following points:

- The frames are continuously taken from the camera internal buffer and copied in an array. This task is accomplished by an independent thread.
- A second thread proceed, in cycle, as follow:
  - firstly it filters the image in order to prepare it for the following phases; if the vanishing point is already computed, it takes advantage from it to limit the image,
  - then it applies the ad-hoc written algorithm of line detection; again, if it knows the vanishing point, the searching regions are smartly defined,
  - it estimates the position of the lane,
  - finally it estimates the location of the vanishing point for the first two step of this cycle.
- The last phase is simple: the software has only to display the result into the screen.

If we consider the logical part of the software, including all the algorithms created and the inner architecture of the application, we can say that it is already working, but of course it is not perfect. The software is stable in terms of noise, light variations and shadows which can somehow mislead the process of lane recognition. It hasn't yet been tested in any particular situations, like during a snowy day, or under a heavy rain. However it shows quite good potentialities in terms of reliability. Probably, the main value is the algorithm of line detection which, supported by a reliable estimation of the vanishing point, can really be able to locate only those lines that are of interest in the process of lane detection.

From a performance point of view probably the software shows its main limitation. If we limit the performance analysis to the status quo, probably we would say that software provides the right amount of speed in order to accomplish all the task in a reasonably short time. However, we have to remember that the final purpose of the software is not to provide the only lane detection but a whole service to a cruise control device, capable also of vehicle detection. In this wider context the software is probably taking too much to successfully finish all its tasks, but some deeper analysis have to be made in this direction: for the time being we can't say the exact amount of time that a full lane detection could take since we don't know the time requirements of the following parts of the project. The fact that the power equipped inside the modern smartphones is fated to increase very quickly allows us to stay positive and to be confident that the computational burden in the scope of this application, soon or later, can be overcame. The actual implementation, that has an HTC One X as target device, is capable to run at 15 fps. This means that the lane boundaries are updated around 15 times per second, a rate which probably suites with the purpose of the project. As we already stated, the lane detection is independent from the camera preview that can run at the speed of more than 20 frame per second.

FINAL THOUGHTS At the beginning of this final chapter (and of the whole project) we ask ourself if it were possible to build an Android application capable of assisting a CruiseControl device. Surely the answer to this question is not covered by the scope of this project, but we can prove that the task of lane detection is feasible using a smartphone, without expensive radars.

ACRONYMS

VM	Virtual Machine	8
JVM	Java Virtual Machine	6
SOC	System on a chip	7
GPU	Graphic Processing Unit	7
SDK	Software Development Kit	9
NDK	Native Development Kit	9
JNI	Java Native Interface	10

BIBLIOGRAPHY

Android Developers Site

2013a *Android Reference Library*, http://developer.android.com/reference/packages.html. (Cited on p. 8.)

2013b *Application Fundamentals*, http://developer.android.com/guide/components/fundamentals.html. (Cited on p. 7.)

Free Software Foundation

2007a *GNU General Public License*, http://www.gnu.org/licenses/gpl.html. (Cited on p. 5.)

2007b *GNU Lesser General Public License*, http://www.gnu.org/licenses/lgpl.html. (Cited on p. 5.)

Jung, Claudio Rosito and Christian Roberto Kelber

2012 "A Robust Linear-Parabolic Model for Lane Following", *Universidade do Vale do Rio dos Sinos Ciencias Exatas e Tecnologicas.* (Cited on p. 1.)

King Hann Lim, Kah Phooi Seng and Li-Minn Ang

2012 "River Flow Lane Detection and Kalman Filtering-Based B-Spline Lane Tracking", *International Journal of Vehicular Technology.* (Cited on p. 2.)

Meier, Reto

2012 *Professional Android 4 Application Development*, John Wiley & Sons, Inc., 10475 Crosspoint Boulevard, Indianapolis.

OpenCV

2013 *OpenCV main site*, http://www.opencv.org. (Cited on p. 11.)

OpenCV 2.4.5.0 Online Documentation

2013 *Making you own linear filter!*, http://docs.opencv.org/doc/turorials/imgproc/imgtrans/filter_2d/filter_2d.html. (Cited on p. 21.)

OpenCV Questions

2013 *OpenCV questions*, http://answers.opencv.org. (Cited on p. 33.)

The Apache Software Foundation

2004 *Apache License, Version 2.0*, http://www.apache.org/licenses/LINCENSE-2.0. (Cited on p. 5.)

Wikipedia - HTC One X

2013 *HTC One X*, http://it.wikipedia.org/wiki/HTC_One_X. (Cited on p. 12.)