Car Navigation Through Computer Vision Methods With Rudimentary Implementation Under Android

Roy Schestowitz

June 30, 2012

Abstract

The increasingly common problem of navigation past obstacles, e.g. on roads and motorways, is typically addressed by a top-down approach (or an overview/bird's eye paradigm) that relies on GPS – or equivalent – and remotely-stored maps, persistently accessed via a network. This is not always sufficient and contingencies can be utilised that are based on real-time data. The situation is further exacerbated when there is no human operator, e.g. driver, involved in this process in order to practice complex judgment. Factors such as traffic in motion are unaccounted for, so automated steering based on obstacles detection cannot be done reliably. This project undertakes the task of tackling car navigation by observing objects through in-car camera (or cameras), utilising mobile device (or devices) mounted upon a dashboard (or another part of a vehicle). The end goal is to produce a semi-autonomous (or computer-assisted) driving experience which exploits off-the-shelf hardware such as Android-based mobile computers (smartphones and tablets). Android is initially chosen as the target platform because, according to various US-centric market surveys, it now holds a majority market share. Results suggest that average framerate in the existing framework permits information of real use to a driver to be collected and delivered in audio/visual form; however, due to lack of time, I am unable to explore refinement of the framework and subsequent adaptation for real-world applications.

1 Introduction

The ubiquity and ever-decreasing cost of mobile/portable devices has gradually increased interest in their usage inside cars. In order to guide cars' preferred routes on the road, programmers are able to harness and truly exploit computer vision methods. It is not clear, however, which ones work best and are also practical to run on mobile hardware with decreased performance capabilities in mind (relative to desktops). This project explores the question by implementing a system which alerts the driver about obstacles on the road, primarily other vehicles[1]. These ideas and accompanying code are extensible in the sense that detecting more types of nearby objects is a task largely hinged on additional training of multiple classifiers, necessitating more crude manual work. Figure 1 shows the program running under a physical Android device.

[1] While most of the scene is immutable and can be understood based on GPS (provided satellite signal), neighbouring cars are unpredictable and cannot be studied off-line. Human behaviour is hard to simulate.

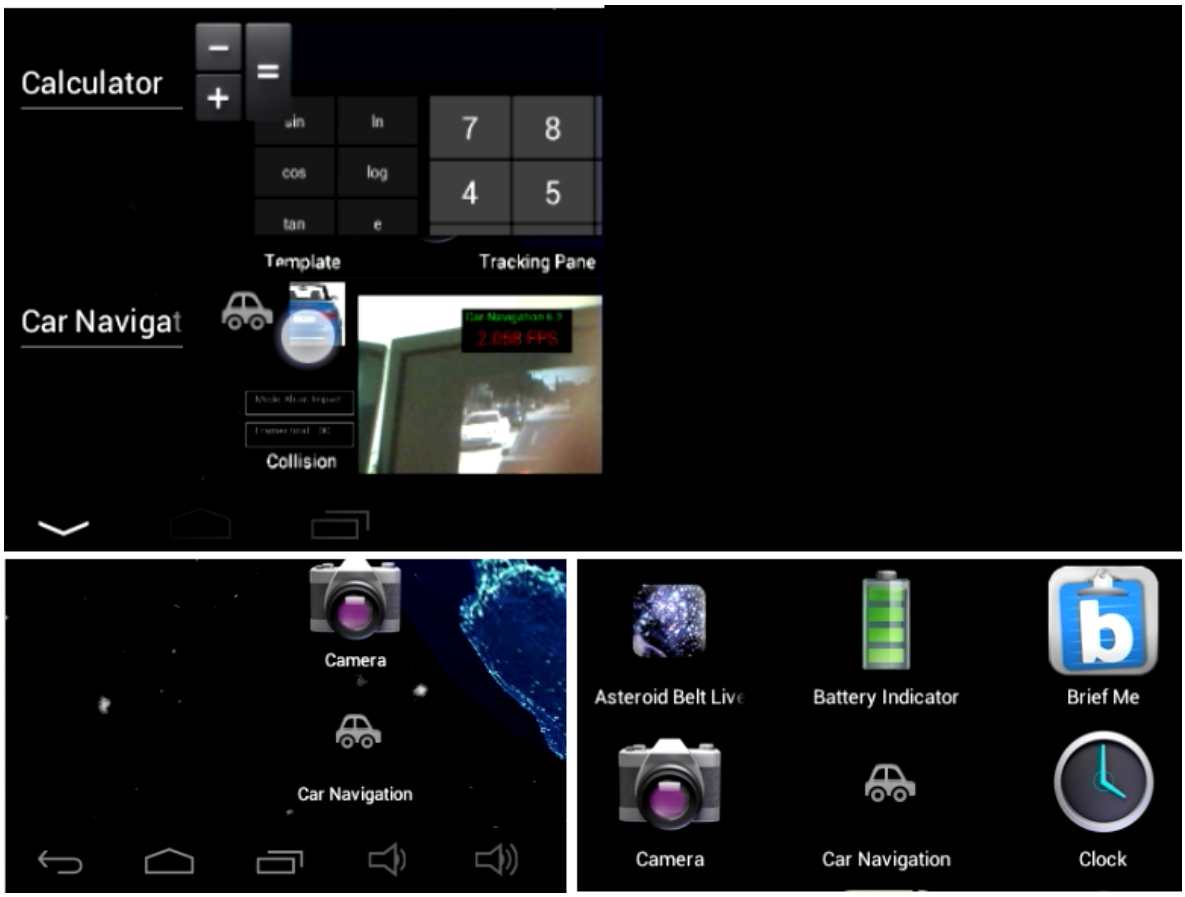

Figure 1: Top: the Car Navigation program running in idle/inactive mode (as one of several programs, with static preview); Bottom left: the program as positioned in the main Android workspace; Bottom right: the program in the Apps list.

In this document I outline implementation aspects of this project. A comparison of multiple methods should be possible, but it remains beyond the scope of this work due to time constraints. I will begin by presenting the development framework, the scientific methods in brief, and the hardware used in subsequent experiments.

1.1 OpenCV

Exploring the problem of car navigation and in particular implementation thereof depends on tools that already exist. Many commonly used algorithms are properly implemented and are well documented. Having surveyed what is available for mobile platforms, I found about 3-4 distinct packages/libraries, most of them proprietary (non-libre) and non-gratis. Advice from veterans in the area (computer vision for Android and iOS) led me to the BSD-licensed OpenCV, which – although poorly documented for particular platforms – remains one of the popular choices, as evidenced by the thriving development community rallying around it. OpenCV boasts many classes of relevance to my application, enabling rapid development and rich visualisation. A company known as Willow Garage [2] has brought OpenCV [1] to Android, a Linux- and Dalvik-based operating system maintained and rapidly developed almost exclusively by Google (through a consortium called Open Handset Alliance, or OHA for short).

1.2 Hardware and Architecture

The original OpenCV framework, primarily led and developed by Intel, has adapted and been tailored to ARM-based devices that are versatile, mobile, and highly power consumption-aware. Performance-wise, as one shall see later, ARM and other non-x86 architectures have their limitations too. It is a matter of market dynamics and inertia, not an inherent flaw. If one pursues a high framerate, e.g. for the purpose of tracking, then hardware acceleration for video becomes quite imperative. It is no coincidence that startups specialising in computer vision on devices end up selling hardware with particular strengths, or otherwise provide hardware addons that make up for desktop-versus-mobile performance anomalies[2]. Multi-core devices and rendering-centric chips yield a lot more frames per second and offload instructions from the processing pipeline.

It is worth emphasising that depending on the hardware at hand – and in particular acceleration of software at hardware level (e.g. video playback) – one can swiftly move from infeasibility to feasibility. Some code was furthermore optimised for particular chips at compilation time. For pattern recognition tasks, for example, it is possible to parallelise particular instructions or join together more data to run operations on. OpenCV takes this approach, but with a Dalvik virtual machine the performance gain might be eroded and acceleration impeded. In the future, it may be worth exploring how overall performance varies depending on hardware used; at present, only emulation mode on dual-core 64-bit and 32-bit x86/AMD Athlon hardware is assessed alongside non-emulated binaries (APK) on a single-core ARM-class chipset. Section 4 will touch on that.

1.3 Pattern Recognition

Unlike methods for navigation which rely on robotics, maps, and path-finding (e.g. [3, 4]), my planned method is to concentrate on geometry as seen from the ground, not from above. The plan is to detect and classify objects on the road, then make a decision based on inter-frame changes. Section 2, starting below, will delve into the details.

2 Methods and Development

The framework I work with is mostly complete for the setting up of video streams, drawing upon the canvas that contains them, and invoking commonly used computer-driven filters. These are incorporated as examples into the program and they include posterise, Canny edge detection, Sobel edge detection, histograms, and so on. In practice, any bit of filtering considerably slows down the process and thus becomes detrimental to the whole pipeline (more on that later); so after much shallow experimentation I ended up dealing with just the raw frames as they come; edge detection can be used to shrink the image by annulling pixels associated with the sky, based on horizon detection (it has expected edge-based characteristics). The implemented features were exploratory at first and more practical later, as getting familiar with the classes and methods can take experience and practice. Compiling and installing the modified programs on a device can take a lot of time too; as such, development cycles are slowed down by complexity associated with the fact that programming is done away from the target device, then passed over a USB 2.0 interface.

[2] Mobile GPUs are still in the stage of gradual acceptance in the market, but they are consistently expensive.

2.1 Haar and Local Binary Patterns

Based on previous work and some further reading, the program might be fine with the following implementation which takes Haar filters or their newer extension into account. Box diagram for the implementation in place can be seen in Figure 2.

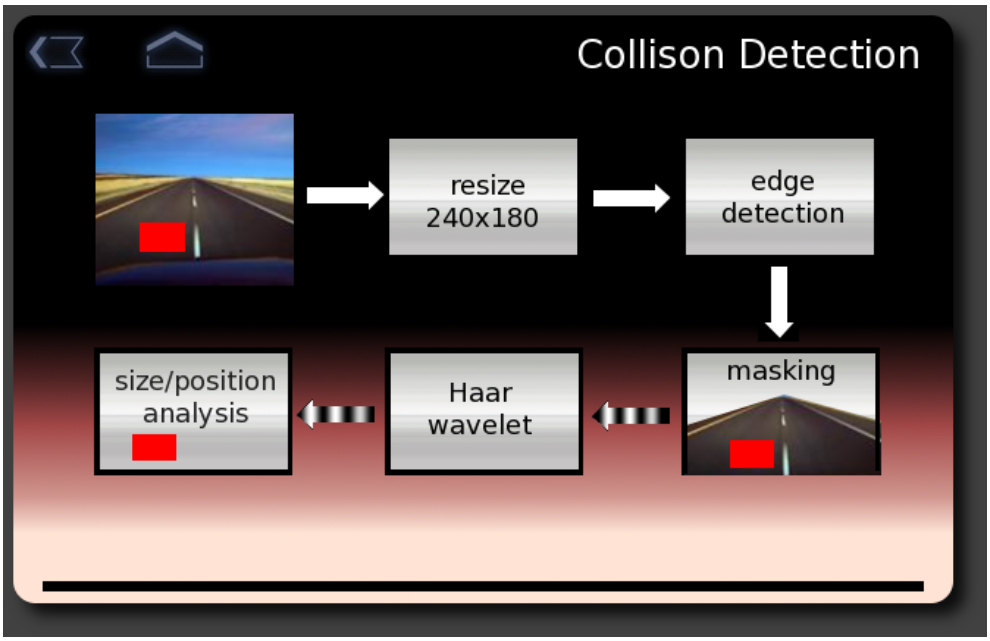

Figure 2: Flow of the foreseen implementation for collision detection/alerting. Resizing if not done, however, unless a very high framerate is sought after (HD footage would be painfully slow).

Based on the paper titled "Effective Traffic Lights Recognition Method for Real Time Driving Assistance System in the Daytime," Kim *et al.* use similar tools. The paper is quite recent and it too uses Haar filters. Before looking into this approach I also attempted to use template-based search; it is infeasible, however, because cars come in many shapes and sizes, not to mention colour and illumination, context, etc. The example template image (see Figure 3) enables me to deal with something that in a mockup-type experiment[3] would be easy to test, e.g. by putting the phone/tablet in front of a computer screen with the exact same static image on it.

[3]The equivalent of synthetic data, or the applications run on known data with simple characteristics.

Figure 3: The template image used as a reference representing a car

2.2 Planned Implementation

Originally and principally, my goal is to detect the back of cars. There may be multiple objects in the scenes that need detecting as such, or none. The baseline object from which to calculate distances to nearby rears of cars is marked in green hues.

Rather than taking a lot of movies with a phone and running experiments solely on those, I used a few static images and several videos from YouTube (dashboard-mounted cameras in an urban setting). I trigger for collision warnings typically when the size of the object in front of the car seems too great relative to prior frames (there are variables to keep track of distance changes). Sound gets used if that distance is beneath a particular (predefined) threshold. There are several different sounds based on perceived alert severity.

At present, the program is not features-rich and it lacks accuracy because the classifier was trained on a relatively small-sized annotated set. Training and targets will be explained in the section about experiments, but it is worth noting that the program now latches onto features using local binary patterns rather than the sliding window, template-based approach, which slowed things down a great deal. Framerate is steady at around 5 frames per second, for now, depending on the hardware at hand.

2.3 Local Binary Patterns

I am training a classifier based on Local Binary Patterns and leaving Haar methods aside for the time being, simply because they are slower and older, particularly the methods and their implementation in OpenCV. In vain I tried finding out if there are any large database of car images in an open lab; in particular, I needed a photographic database containing images with annotation marking the back of cars (a rectangular selection around all the cars in the images), just for the purpose of training a Local Binary Patterns classifier. The next section will explain what I ended up using.

2.4 Development Environment

As a baseline example and starting point, Figure 4 shows a prototype of a program that captures about 5 frames per second from an Android-enabled camera.

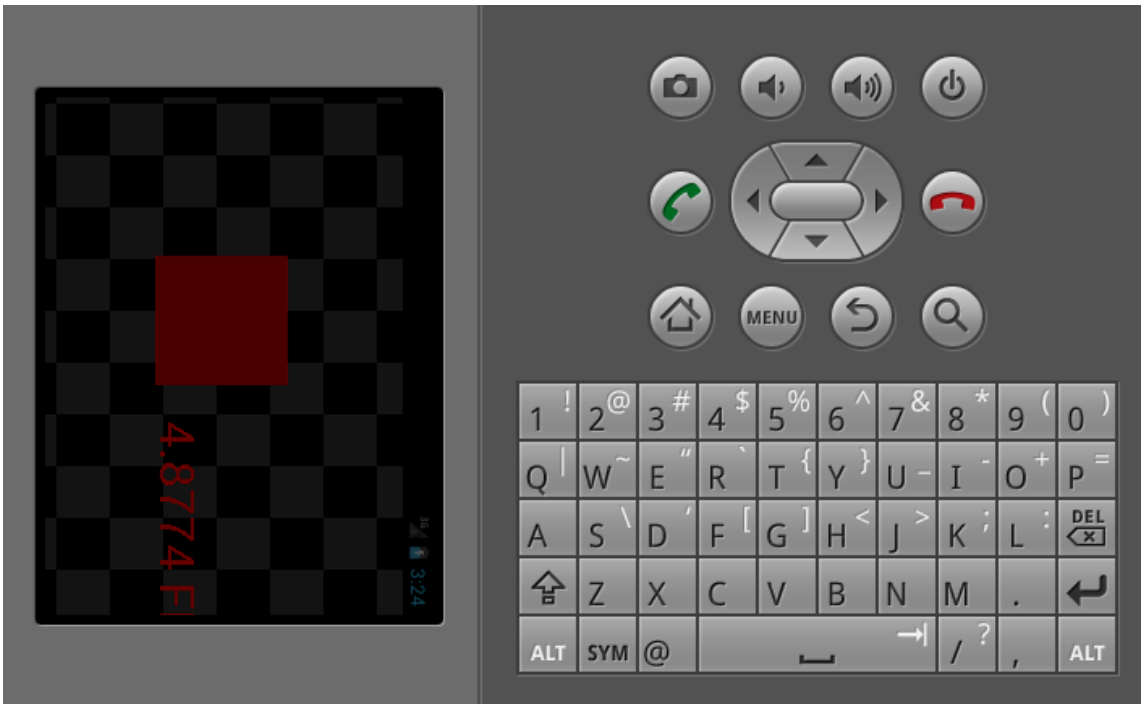

Figure 4: Example of an OpenCV-enabled application run on a low-resolution device with a keyboard (emulation). The number on the left represents the number of frames per second and the checkerboard pattern indicates that no camera is currently operating.

The interface is quite different for a high-resolution smartphone or a tablet, as shown in emulation mode in figures 5 and 6.

Figure 5: Emulation of Android on a tablet-type device with OpenCV

Figure 6: Applications installed on Android on a tablet-type device with OpenCV

Figure 7 shows the image obtained by applying Canny edge detector [5] to the synthetic scene shown before.

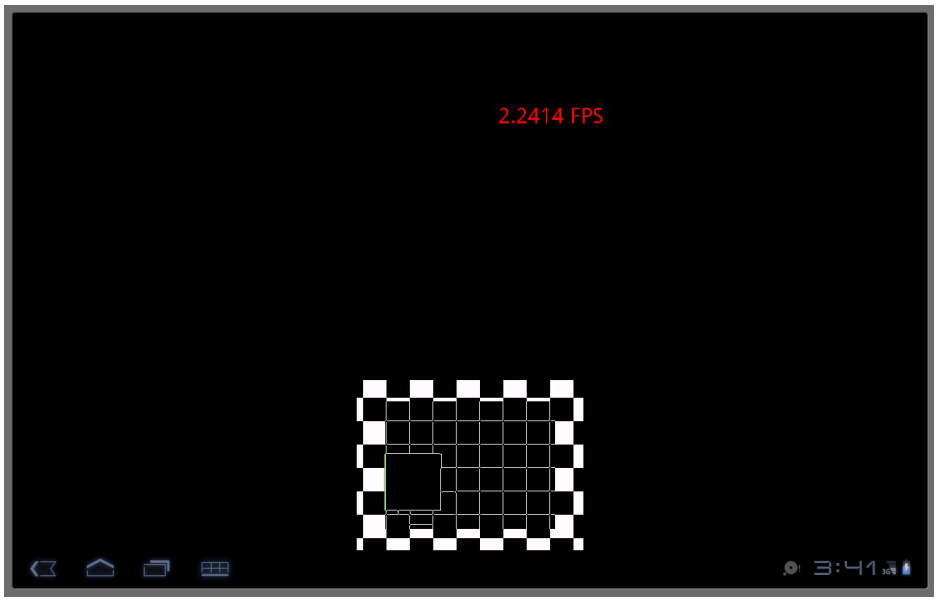

Figure 7: A simple program that applies the Canny edge detector to the video input (the default checkerboard pattern in this case) and the accompanying frame rate estimator on a dual-core AMD workstation (emulation mode). This may correspond to a high-resolution smartphone such as the Galaxy series or even a comparable tablet.

The SDK in use is Eclipse, which is further enhanced with Google addons, as shown in Figure 8. The program requires the Android SDK, NDK, typically Eclipse, and of course OpenCV (I used version 2.4.x).

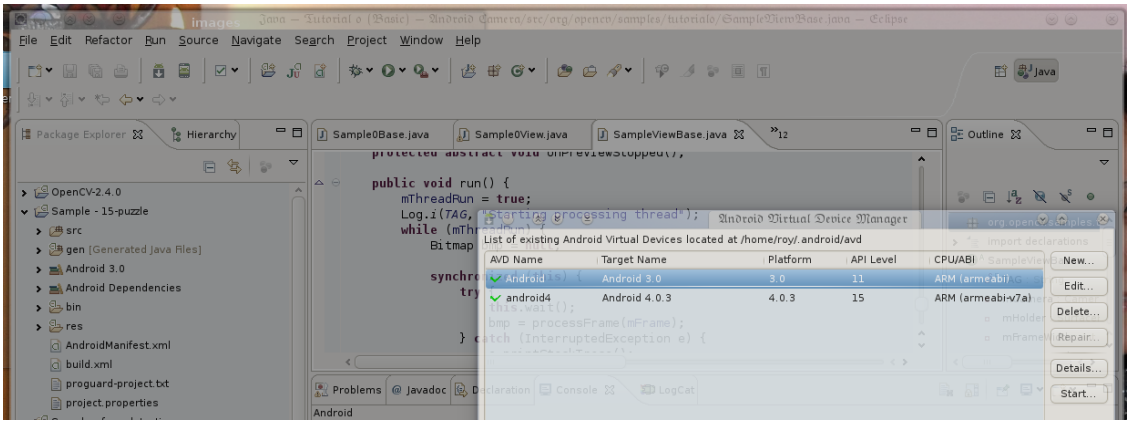

Figure 8: Eclipse SDK and the two device types being tested with the simple program. Notice how the Eclipse toolbar is augmented by Google addons.

One would have to photograph the tablet to properly show this running with real data. In emulation I tested several device types, but the camera is emulated (the example in Figure 9 is WVGA), unlike when it is run on the device.

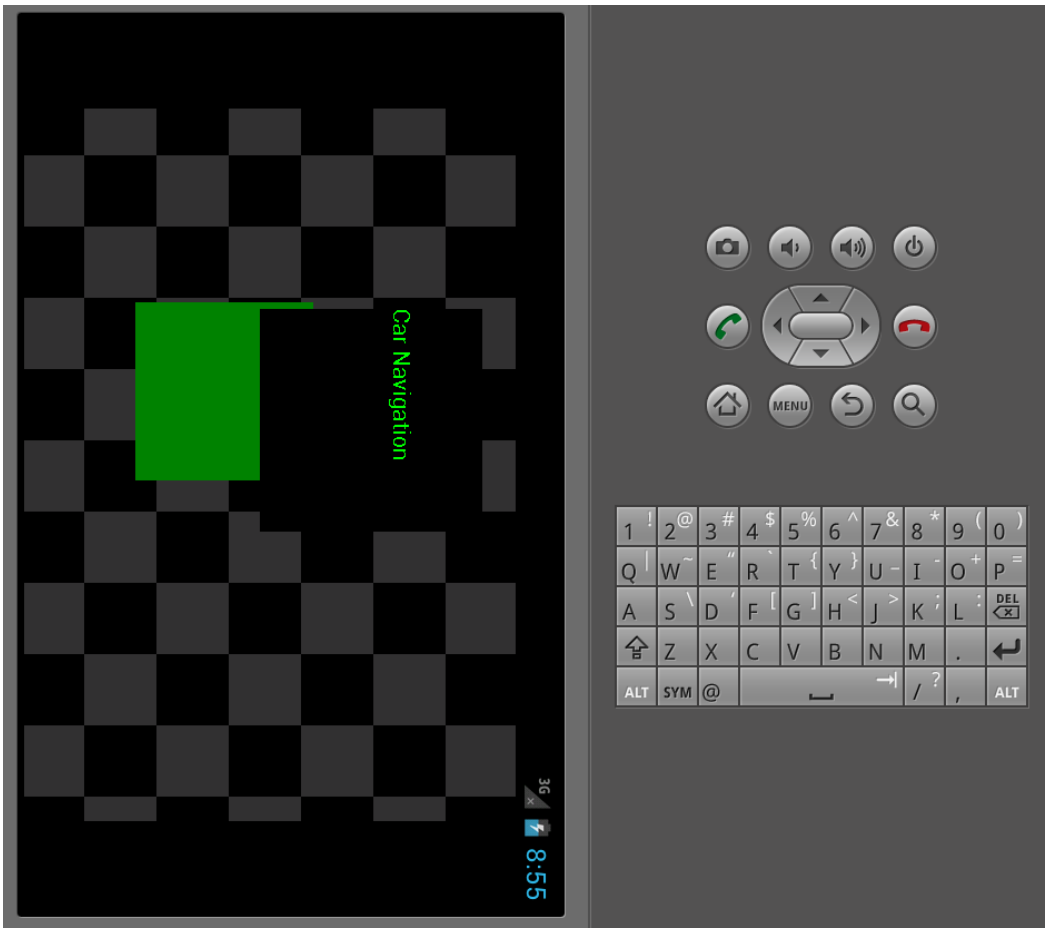

Figure 9: The program in WVGA, emulated

For demonstration purposes I gathered some screenshots of the complete program running on a real device, a high-resolution tablet in this case. Figure 10 shows some of those screenshots. These show the latest version of the program in "About Impact" mode - a menu option that enables tracking nearby obstacles. There are nearly 10 such menu options (the total varies depending on classification of distinction), but some are simpler filters.

Figure 10: Top left: car detection from a distance; Top right: detection from a shorter distance; Bottom left: car too close for detection; Bottom right: detection from afar. The program also comes with a menu and is not complicated to work with.

2.5 Profiling

Prior to use of cascade classifiers I attempted to exploit sliding window-based template matching, where a score for match is calculated in one of 6 possible ways (provided by and implemented in OpenCV). This idea was quickly dropped because of performance issues that had been identified by debugging and profiling executed code. Matching templates proved to be the most problematic part based on this profiling. Matching templates takes about 5 seconds, or an order of magnitude longer than all the other stages *combined*[4]. Here is an example sequence of operations as echoed in `System.out`:

```
05-28 23:42:41.558:  I/System.out(10194):  Fetching template from the Internet
05-28 23:42:41.628:  I/System.out(10194):  Resizing template
05-28 23:42:41.638:  I/System.out(10194):  Height:  20
05-28 23:42:41.638:  I/System.out(10194):  Width:  20
05-28 23:42:56.058:  I/System.out(10194):  Starting Canny edge detection
05-28 23:42:56.248:  I/System.out(10194):  Starting Gaussian Blur
05-28 23:42:56.338:  I/System.out(10194):  Encoding conversions
05-28 23:42:56.338:  I/System.out(10194):  Matching template
05-28 23:43:02.598:  I/System.out(10194):  Frame completed
```

[4]The speed of digestion of frames is at around 12 per second (maximum) for the hardware and conditions at hand (depending on light, available RAM, etc.), but when some more processing comes into play it slows down the actual display rate (not capturing rate) down to about 0.1, depending on what the program is trying to achieve. I have put the tablet in debugging mode so that I can view system statistics (load average and other kernel-level output) on the tablet's screen, along with real-time shell output (warnings for example) in Eclipse running Ant.

```
05-28 23:43:02.728:  I/System.out(10194):  Starting Canny edge detection
05-28 23:43:02.888:  I/System.out(10194):  Starting Gaussian Blur
05-28 23:43:02.968:  I/System.out(10194):  Encoding conversions
05-28 23:43:02.978:  I/System.out(10194):  Matching template
05-28 23:43:09.108:  I/System.out(10194):  Frame completed
05-28 23:43:09.228:  I/System.out(10194):  Starting Canny edge detection
05-28 23:43:09.408:  I/System.out(10194):  Starting Gaussian Blur
05-28 23:43:09.498:  I/System.out(10194):  Encoding conversions
05-28 23:43:09.498:  I/System.out(10194):  Matching template
05-28 23:43:15.718:  I/System.out(10194):  Frame completed
05-28 23:43:15.818:  I/System.out(10194):  Starting Canny edge detection
05-28 23:43:15.988:  I/System.out(10194):  Starting Gaussian Blur
05-28 23:43:16.078:  I/System.out(10194):  Encoding conversions
05-28 23:43:16.078:  I/System.out(10194):  Matching template
05-28 23:43:22.018:  I/System.out(10194):  Frame completed
05-28 23:43:22.198:  I/System.out(10194):  Starting Canny edge detection
05-28 23:43:22.368:  I/System.out(10194):  Starting Gaussian Blur
05-28 23:43:22.448:  I/System.out(10194):  Encoding conversions
05-28 23:43:22.458:  I/System.out(10194):  Matching template
05-28 23:43:29.078:  I/System.out(10194):  Frame completed
05-28 23:43:29.458:  I/System.out(10194):  Starting Canny edge detection
05-28 23:43:30.028:  I/System.out(10194):  Starting Gaussian Blur
05-28 23:43:30.248:  I/System.out(10194):  Encoding conversions
05-28 23:43:30.258:  I/System.out(10194):  Matching template
05-28 23:43:40.278:  I/System.out(10194):  Frame completed
```

For real-time applications one needs to carefully think how to thin down the pipeline. I am using only one CPU core, so it is similar enough to ARM-based smartphones hardware (more on the hardware in Section 4).

For visual augmentation of the above profiling information I wrote a shell script which I have made freely available[5]. Its goal is to turn this text into a graph, eventually displayed using `gnuplot` and other command-like GNU tools. The bumps help show just how considerable the toll of template matching really was at the time (see Figure 11).

[5]The script can be handy later on when assessing performance elsewhere in the algorithm.

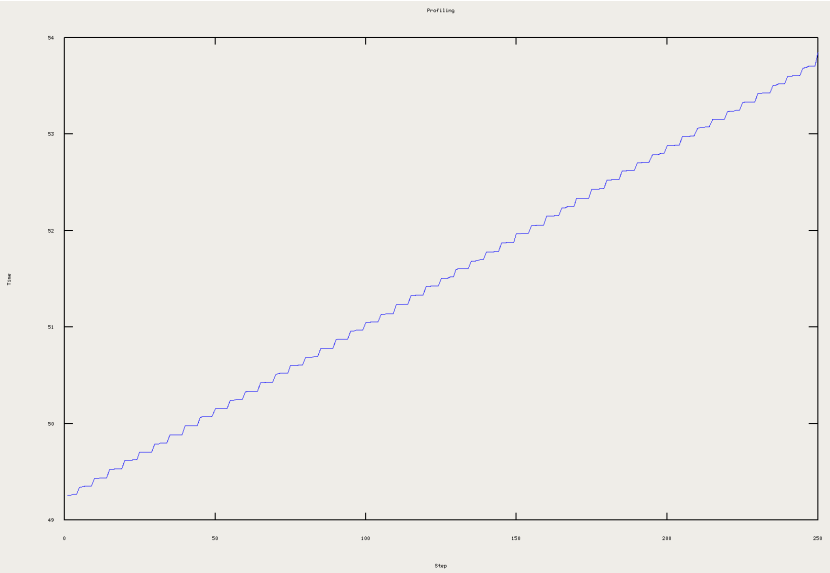

Figure 11: The steps in the pipeline (multiple frames) versus time, showing quite clearly the toll of template matching (steep vertical climbs in the curve)

3 Data

All experiments are using a Caltech-produced database of the rear of cars. There may be more options out there, but not many. Training is done on those images and testing is done on unseen videos or images similar to the training set (but not overlapping or identical). I have begun tracing some cars which were not included in the training set and many examples are shown in video form (auxiliary); it works reasonably well despite training on just 15-50 positives and 1000 negatives (commonly called "background"). The pace of processing is at around 5 frames per second when it is done properly. This can be sped up easily if the will is there, especially because the current code is wasteful (debugging bits are the main culprits).

Figure 12: Examples of background images (negatives)

Figure 13: Examples of positives

Figure 14: Examples of positives with multiple scales (many rears of many cars)

To get more positives I will need to do manual work or just write a good script. It would not be so trivial as it requires a scanner/parser to pick up and collate pertinent tokens, then do some maths because formats vary semantically, not just structurally (PASCAL being inherently different from OpenCV input is an impediment). Figures 12-14 show example images from the data set used for the following experiments.

4 Experiments

The experiments were not quite systematic or comparative, as such experiments would require the code to mature and they would also consume a lot of time. Over time I created (i.e. trained) about 30 different classifiers, then tested them by observation and subsequently tweaked classifiers in according with all prior observations. Improvements were rapid but gradual. I took a dozen videos for comparative analysis. The aim has been to find the right sensitivity levels (minimum hit rate, maximum false alarm rate, etc.), sample sizes (24x24 pixels would work well as an abstraction of patterns seen in car rears), and sizes for both sets (it would require a lot of manual work to expand, or in other words a time investment).

Training with large data sets is an essential stage because without good detection of moving obstructions (mostly cars) the alerting would be spurious and therefore counter-productive. The framerate attained is still above 5 frames per second, which is actually very decent for this hardware . Idle, standalone video capture remains is only at about 10 FPS on this inexpensive MIPS/RISC board. It's mediocre equipment on a relative scale.

4.1 Hardware and Performance

The aforementioned screenshots and explanations sometimes rely on desktop emulation; in practice, I wish to investigate performance on real hardware in real-world situations. After many hours that involved studying a lot of options (both device and desktop end), then editing files etc. I managed to successfully install my program on an Android 4.0 tablet. For the basic edge detection and analysis I can get about 4 frames per second and the hardware specifications include a 10" screen (VGA), 1GHz CPU, and 1GB of on-board RAM [6]. Testing over USB is trivial and fast once particular workarounds are mastered, reducing lag. Regarding the hardware I currently test and work with, it is important to note that it is supposed to take account and align with what is cheaply available on the market right now. Cutting edge would be unrealistic and overly optimistic.

When I proceeded to doing profiling for feasibility I found some caveats and bottlenecks. For a window size of 402px by 512px (an inner window, leaving margins out) the performance is a little better than emulation

[6] VGA being processed on single-core Rockchip 2918 CPU running at 1 GHz with 1024 MB of RAM, Linux kernel version 3.0.8, and Android ICS 4.0.3 to be more specific

on x86 with half a gigabyte of RAM. The hardware here is not something advanced like Tegra (those are expensive) and unlike some Sony-made ICS tablets that come with high-resolution cameras at the front and the back, this one uses a resolution which can be managed by real-time image processors. If I can only deal with a dozen frames per second, this may still be enough to get prediction of objects' motion on the road.

At the earlier staged I applied some edge detectors and then tried to identify a silhouetted road part, which could then be stripped apart to avoid performing computation on irrelevant parts like the sky. At earlier stages I also tried to apply template matching (cars) using one of 6 methods that I have, but it slows down the pace of processing to sub-1 (FPS) levels, so I needed to decrease the resolution for this part (coarse or coarse-to-fine). At that moment the program was leaking some memory, so I was still debugging it.

I are sure one can do better than 5 frames per second, but while developing and debugging the problem I do not worry about performance too much, at least not yet. For standard video sequences on this ARM-based device I can sometimes get about 8 frames per second (raw and idle). Maybe it is a hardware limitation, related perhaps to throughput or lack of hardware acceleration (some x86 chipsets have MPEG acceleration on-board). Tagra tablets (they are all rather expensive) have very smooth video in comparison. They are HD as well. Trying to run the same program on them would be an interesting exercise.

5 Summary

I have presented an outline and developed a user-friendly application/framework for crudely identifying and tracking objects on the road. Considering time limitations, it is safe to say that the full potential of this project was not fulfilled. The code is made free/open source software, however, encouraging others to carry on from the point reached. Utilisation of the framework depends on one's ability to comprehend the underlying concepts, primarily those associated with pattern recognition and real-time tracking. Nothing particularly novel has yet been attempted.

This document detailed and showed compartments of the framework, then demonstrated how it adapts to various form factors, depending for the most part on screen resolution. This was tested in emulation mode for the most part. It then delved deeper into the methods explored and the resultant performance. Data for training classifiers was explained, then some tests of the classifiers were shown. Many demos were produced in video form and uploaded to the World Wide Web. In them, evidence of gradual improvement over time – depending on the size of the training set and the evolving implementation – exists. Further details about the projects, including videos, source code, binary executables, and further explanations, can be found online at http://schestowitz.com/Research/Cars/Car_Navigation/.

Acknowledgements: the project was funded by the European Research Council.

References

[1] G. Bradski amd A. Kaehler, "Learning OpenCV: Computer Vision with the OpenCV Library," O'Reilly Media, 2008. 2

[2] K. T. Benoit, "Willow Garage," International Book Marketing Service Limited, 2012. 2

[3] K. Nagatani, Y. Iwai, and Y. Tanaka, "Sensor based navigation for car-like mobile robots using generalized Voronoi graph," International Conference on Intelligent Robots and Systems, vol. 2, pp. 1017–1022, 2001. 3

[4] H. A. Vasseura, F. G. Pina, and J. R. Taylor, "Navigation of car-like mobile robots in obstructed environments using convex polygonal cells,"Robotics and Autonomous Systems, vol. 10, pp. 133–-146, 1992. 3

[5] L. M. Surhone, M. T. Tennoe, and S.F. Henssonow, "Canny Edge Detector," VDM Verlag, 2010.

7