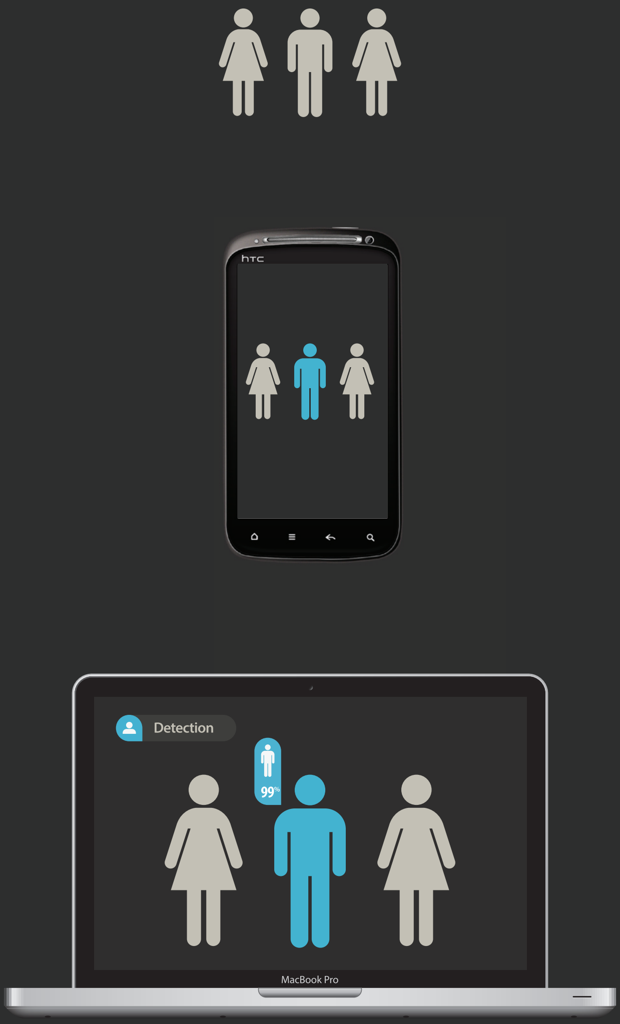
March 2012

Andrej Pancik

Lightweight Robot Vision Architecture

Image Processing Pipeline Based on Commodity Hardware



0. Contents

1 What is it?

Lightweight robot vision architecture is a collection of code snippets and APIs oriented on image processing and computer vision that work in a compatible way. In this project, I have designed and started to develop this architecture that has already become an ultimate tool in the rapid prototyping of my ideas.

I tried to create a full data and image processing pipeline including data acquisition, multiple processing steps and visualisation. The goal was to base it solely on commodity hardware that is widely available and the costs of which are reduced by mass production.

1.1 Phone as a Platform

One of the key decisions was to use an Android phone as a data acquisition device. Built-in cameras were used to get a video stream and Wi-Fi to transfer it to a personal computer. Of course, the result is more than just an expensive Wi-Fi camera. There is much more potential in it. Each Android phone is a universal device. Camera, GPS, gyroscope, acceleration sensors, and compass are just a few of wide range built-in sensors. Moreover, today's smartphones provide a powerful processors to perform pre-processing over the data.

Another advantage is that for development on such a platform, one does not need any specialised hardware or expensive development kits. A personal computer with a USB cable is enough.

1.2 Importance of Integration

Of course, an important consideration was integration with other relevant open source projects such as Robot Operating System, OpenCV, Robot Vision Toolbox and Arduino. All of these projects must coexist in synergy.

Development was conducted while keeping the best practices of software development in mind. Thanks to the focus on a flexible design, many code reductions and optimisation were performed during the development time, resulting in clean and lightweight API.

The output is a simple platform, which facilitates a rapid robot algorithms development across multiple platforms.

2 Brain in the Air

I believe that Android powered phones (or any other mobile devices) could be used as processing units of autonomous robots. There has already been some work done in this area and projects such as Cellbots[1] have appeared. Cellbots, however, target hobby robotics and low demand applications.

My idea is to use mobile platforms for high performance computation and image preprocessing/processing. Current generation hardware might not have the latency that is good enough for this purpose, but I believe that looking into this area is beneficial. An important factor is that all the hardware of mobile phones has been commoditized and is comparable or cheaper than custom solutions.

A price of a usually cheap Arduino board extended with a battery, Wi-Fi, GPS and other modules could easily sky-rocket to several hundreds pounds. However, a cheap phone connected with a bare Arduino via an Amarino Toolkit[2] can replace those modules or at least provide a development platform for testing and evaluation.

In the scope of implementation, appropriate technologies have to be picked and available hardware scanned.

An ultimate goal might be getting a "brain in the air" – a phone controlled quadcopter. Of course, this might be unrealistic with the current state of art. This project is about the first steps in this area.

1 http://www.cellbots.com/
2 http://www.amarino-toolkit.net/

3 Architecture

3.1 General Architecture

General Architecture for the processing stack created in this project is almost trivial. A video from a video source is streamed by a video streamer and to a video host. The host pushes the frames to a video processor, which then takes care of frame processing and optionally visualisation.

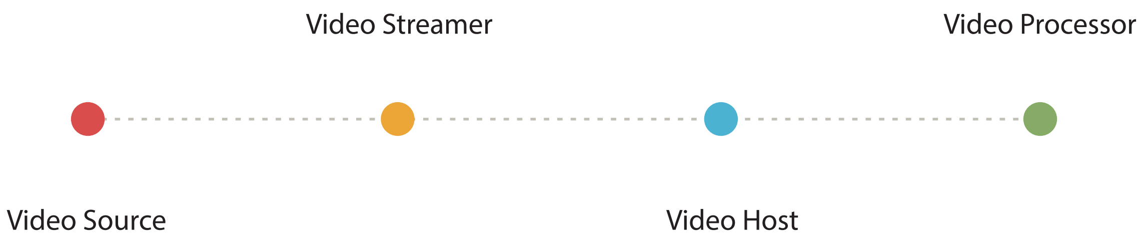


Figure 1. Architecture visualisation

Implementation-wise, the solution employs client-server architecture. The video host oversees the activity of the video streamers and manages the processing of delivered frames by the video processor.

3.2 Video Host

The video host takes a multitude of roles in the application. It instantiates the processor, stores the stream for later playback and displays the preview. For my test purposes I have implemented two sample video hosts.

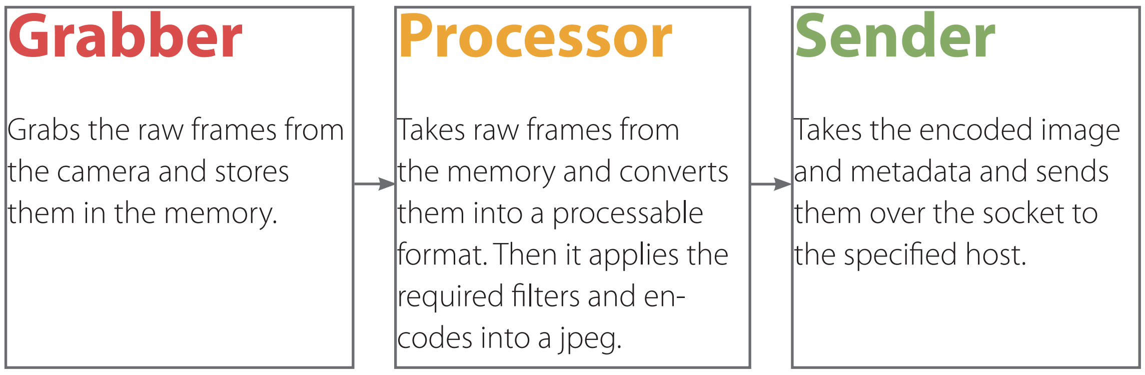
One is processing received data with the option to display or capture on disk. The second is integrated into Matlab ready to utilize a powerful collection of image processing and computer vision algorithms, which are either present in the Image Processing Toolbox or the mature Machine Vision Toolbox[1]. The implementation of the Matlab video host uses Matlab MEX and does not require the expensive Image Acquisition Toolbox from Mathworks. Therefore, it can operate within student licence of Matlab.

1 http://petercorke.com/Machine_Vision_Toolbox.html

3.3 Video Streamer

The video streamer takes any video source and publishes it. In my tests I used multiple video streamers such as an application that grabs the video from the webcam or grabs the video from the video file.

Another video streamer implemented is the Android application that sends sensory information along the image data. To fully utilize the dual-core architecture of my development phone, I created a multithreaded program with three concurrent threads. Each single thread operates as a background worker.



Grabs the raw frames from the camera and stores them in the memory.

Takes raw frames from the memory and converts them into a processable format. Then it applies the required filters and encodes into a jpeg.

Takes the encoded image and metadata and sends them over the socket to the specified host.

Grabbing video on an Android turned out to be the most challenging part of the project. Even after spending a lot of time on this, it is still not fully resolved. I have been experiementing with two ways of grabbing the video on an Android: an Android grabber and OpenCV native grabber.

An Android grabber supports grabbing frames in a YUV colour space. API exports Android grabber to Java code, allowing the processing or colour conversion to be done either in Java itself or with OpenCV, but with a potential performance decrease due to extensive memory copying.

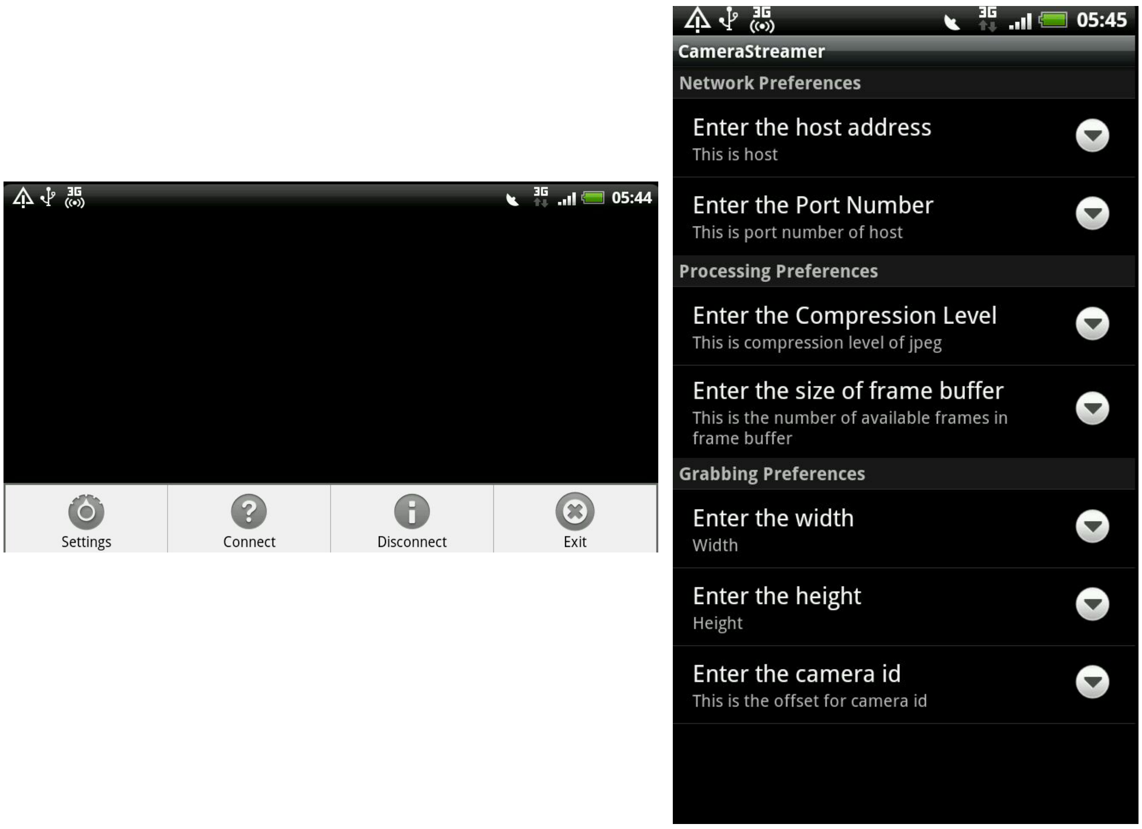
An OpenCV native grabber accesses the camera directly and then exports the API via JNI in Java or directly to C++ code. It is slightly slower in comparison to an Android grabber. However, only one memory copy is performed and therefore a native grabber is very suitable for cases where processing in C++ or with OpenCV will be performed. In the current version it is buggy, but powerful.

3.4 Networking

An important part of the project, due to heavy data traffic between hosts, was the networking layer. Not only was it necesarry to transfer the encoded image data, but also to transfer any arbitrary data such as computed image features. In addition one of the major focus points was rapid development, so I was on the lookout for a very flexible and multiplatform solution.

For this reason, I have picked two suitable frameworks for data interchange : Google Protocol Buffers[2] and Apache Thrift[3]. After initial tests of both frameworks, I have decided to go for Google Protocol Buffers.

Boost ASIO lightweigh networking provides a neat abstraction above low level network operation. Moreover, it also supports asynchronous functioning (not implemented in the released version of this project) which might bring a performance boost in case of multiple video streamers.



CameraStreamer

Network Preferences

Enter the host address
This is host

Enter the Port Number
This is port number of host

Processing Preferences

Enter the Compression Level
This is compression level of jpeg

Enter the size of frame buffer
This is the number of available frames in frame buffer

Grabbing Preferences

Enter the width
Width

Enter the height
Height

Enter the camera id
This is the offset for camera id

Figure 2. Screenshots from Android Video Streamer

2 http://code.google.com/apis/protocolbuffers/
3 http://thrift.apache.org/

4 Examples of Use

The advantage of a flexible image processing pipeline is that we can move and distribute processing steps along the pipeline. Processing steps, when it makes sense, can stay on the smartphone and save the limited Wi-Fi bandwidth and costly image encoding.

Let's take another simple example. Let's say we want to get detected edges from the camera of smartphone to Matlab environment. In technical terms, we want to create a simple pipeline where one HTC Sensation S is connected to MacBook Pro 2.4 GHz Intel Core 2 Due with 4GB RAM running Mac OS X running Matlab R2012a and streaming grayscale video stream of dimensions 640 x 480 while finding edges with canny edge detection.

Base grabbing speed in this setup was at around 17 frames per second (FPS), limited by a native grabber used on the Android phone. My lightweight architecture took only a small performance tax, and so the unprocessed video stream was coming to Matlab for processing in the speed of 16.93 FPS. Processing the image in the video host (with the use of OpenCV in C++) led to an insignificant drop to 16.82 FPS. However, moving the processing step to Matlab caused a substantial frame rate drop to 4.84 FPS. In contrast,

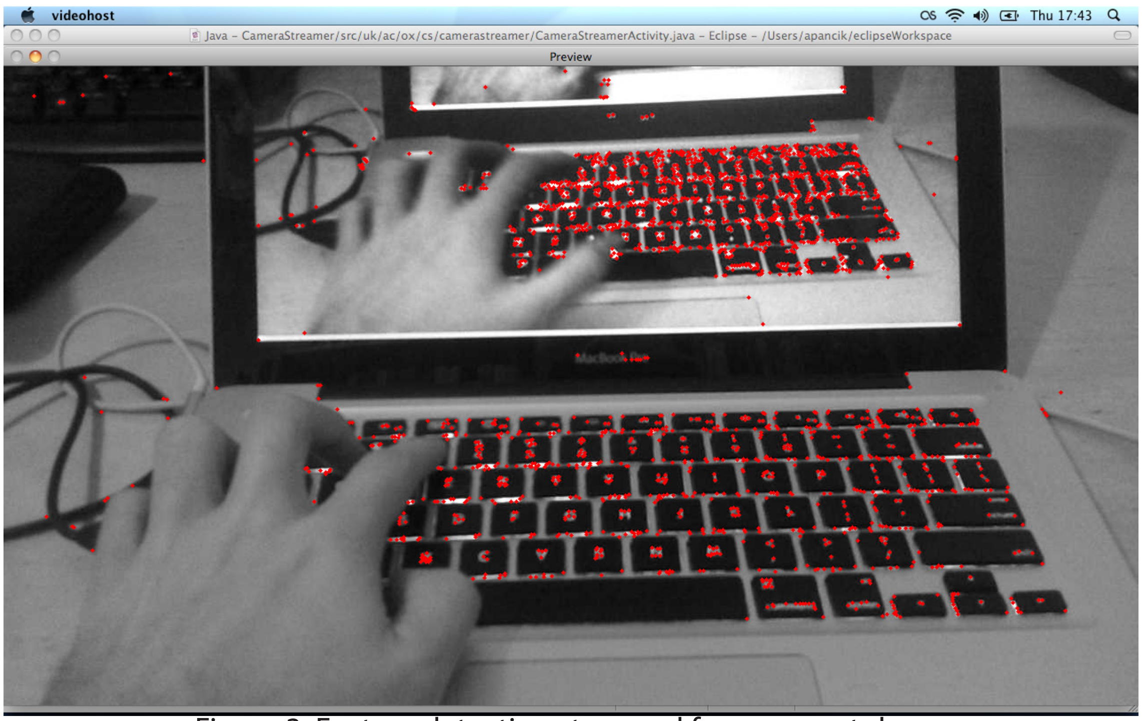
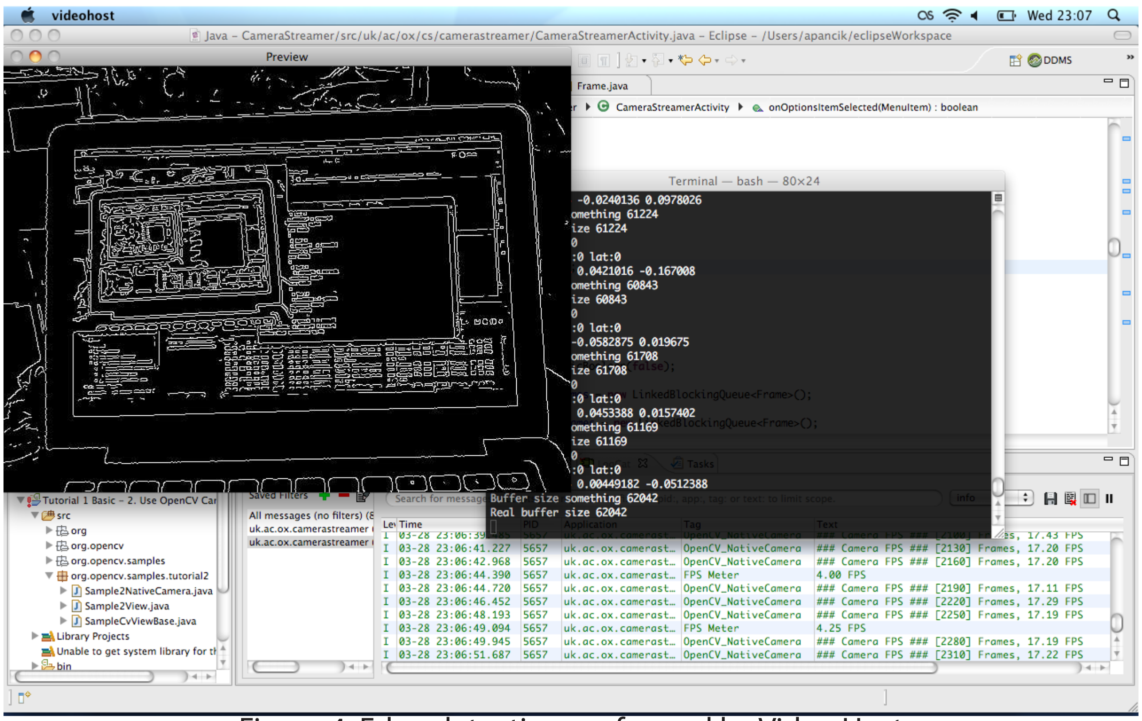

videohost
Thu 17:43
Java – CameraStreamer/src/uk/ac/ox/cs/camerastreamer/CameraStreamerActivity.java – Eclipse – /Users/apancik/eclipseWorkspace
Preview

Figure 3. Feature detection streamed from a smartphone



```
 -0.0240136 0.0978026
omething 61224
ize 61224
0
:0 lat:0
 0.0421016 -0.167008
omething 60843
ize 60843
0
:0 lat:0
-0.0582875 0.019675
omething 61708
ize 61708
0
:0 lat:0
 0.0453388 0.0157402
omething 61169
ize 61169
0
:0 lat:0
 0.00449182 -0.0512388
Buffer size something 62042
Real buffer size 62042
```

```
I  03-28 23:06:41.227  5657  uk.ac.ox.camerast…  OpenCV_NativeCamera  ### Camera FPS ### [2130] Frames, 17.20 FPS
I  03-28 23:06:42.968  5657  uk.ac.ox.camerast…  OpenCV_NativeCamera  ### Camera FPS ### [2160] Frames, 17.20 FPS
I  03-28 23:06:44.390  5657  uk.ac.ox.camerast…  FPS Meter            4.00 FPS
I  03-28 23:06:44.720  5657  uk.ac.ox.camerast…  OpenCV_NativeCamera  ### Camera FPS ### [2190] Frames, 17.11 FPS
I  03-28 23:06:46.452  5657  uk.ac.ox.camerast…  OpenCV_NativeCamera  ### Camera FPS ### [2220] Frames, 17.29 FPS
I  03-28 23:06:48.193  5657  uk.ac.ox.camerast…  OpenCV_NativeCamera  ### Camera FPS ### [2250] Frames, 17.19 FPS
I  03-28 23:06:49.094  5657  uk.ac.ox.camerast…  FPS Meter            4.25 FPS
I  03-28 23:06:49.945  5657  uk.ac.ox.camerast…  OpenCV_NativeCamera  ### Camera FPS ### [2280] Frames, 17.19 FPS
I  03-28 23:06:51.687  5657  uk.ac.ox.camerast…  OpenCV_NativeCamera  ### Camera FPS ### [2310] Frames, 17.22 FPS
```

Figure 4. Edge detection performed by Video Host

when I moved the processing step to the phone and let 'canny processed' images flow via the air, Matlab got an optimistic frame rate of 5.25.

One might ask why we would compute anything on the smartphone when using OpenCV in C++ host achieved the best results? The following example might provide a reasoning to this motivation. Let's say we want to detect FAST (Features from Accelerated Segment Test) from an image streamed from the smartphone to the computer. As far as hardware is concerned, it is the same setup as in the previous example.

However, we also have an impractical – but for the reason of this demonstration suitable – requirement of the input image having dimensions 1280 x 720 (Full HD). The computation of features on the phone led to an average frame rate of around 2.85 FPS on the test scene, while sending the image over the Wi-Fi and performing the detection on the computer side ended with an average frame rate of 1.42 FPS. That means that for the images of these dimensions, the processing on the phone is exactly 2 times faster than on the computer.

Of course, these examples are not practical, but they show a performance benchmark of lightweight robot vision architecture.

5 Conclusion

Whilst working on the project, I started to implement a minimalistic vision-processing stack for the rapid prototyping of robot applications. Combining a wide range of technologies into one interoperating unit is always a challenge. C++/Boost, Java, Google Protocol Buffers, OpenCV are all technologies curated and tested to perform well in this use case. Almost every component of this project had to be completely rewritten several times in order to improve performance or flexibility.

My original intention to use Matlab as the processing language did not turn out to be a good idea, as Matlab was slow and unreliable in comparison to C++ with Boost and OpenCV. Because of this low performance, Matlab is generally not suitable for real-time applications.

My efforts are now directed to the creation of an accessible robotic framework using an Android phone as the computing centre of the autonomous robot. While this idea might not yet be realistic, the use of an Android phone as a platform for quick tests of robotic vision applications is already here.

6 Resources

6.1 Source Code

The source code can be downloaded from http://dl.dropbox.com/u/39605091/Light-weight.Robot.Vision.Architecture.2012.03.26a.zip

6.2 Licence

Most of the framework is and will be released under GNU GPL 2 licence and therefore usable by anybody without warranty, charge, and on an ”as is” basis.

Some parts such as mexopencv by Kota Yamaguchi which is included in the package are licensed under different but compatible licences.