Visualization and Control of an EDVS Vision Sensor Using Android Mobile Devices

Lab Report by Lukas Murmann und Victor Orlinskiy

Neuroscientific System Theory Group (NST)
Technische Universität München

Prof. Dr. sc. nat. Jörg Conradt

Supervisor:
Cristian Axenie, M.Sc.

Beginn:	xx.xx.200x
Zwischenbericht:	xx.xx.200x
Abgabe:	xx.xx.200x

Abstract

In this report we present the development of an Android application for visualization and processing the data from a wireless embedded dynamic vision sensor. This sensor responds not to dynamic changes in the brightness and allows faster reaction with less computational time than common frame-based image sensors.

The dynamic vision sensor transmits data to a WLAN module. A tablet computer or a smartphone can be used to create a hot-spot and connect to the sensor. The Android application allows configurable visualization of received data as well as implementation of various use-cases, which can be easily attached to the basic structure.

Every use-case implements an extension to a basic java class Algorithm and contains methods for handling data processing, visualization and user interface callback. Our application is based on Android fragments so its interface can be extended for every new Algorithm. It can either use a predefined layout for its interface or create it at run-time.

In the beginning we discuss technical background of dynamic vision sensors and Android platform. In the following chapters we describe the architecture of our application. At the end we implement line-tracking and an optic-flow detection algorithms to evaluate the performance of our application and provide examples of algorithm integration.

Contents

Chapter 1

Introduction

Human brain processes information using tenths of billions of neurons, that communicate by sending impulses through axons and receiving them through synapses. Each neuron can be connected to up to hundred thousand of others. Although the interactions between single neurons are quite simple, through complex networking and parallel processing of impulses they can handle many complex tasks in an efficient and robust way.

By studying processes in neuronal systems it is possible to imitate their behavior in hardware and software . Electronic devices, which are inspired by living creatures are often called neuromorphic due to their biological motivation. Many of them are preferably used in real-time applications for example in robotics [**?**].

In this lab course we work with a biologically inspired vision sensor. This sensor reacts to its environment in a similar way as human retina does: not by sending sequences of frames, but by sending an impulse, whenever one pixel changes its brightness.

We create an application for Android based mobile phones and tablet PCs, which can be used to receive and process data from the dynamic vision sensor. The main goal is to provide a platform for demonstrating the abilities of the sensor as well as the performance of the algorithms.

Chapter 2

Technical Background

We now introduce the technical background to the event-based vision sensor, as well as the Android operating system.

2.1 Introduction to EDVS

2.1.1 Biologically inspired vision sensors

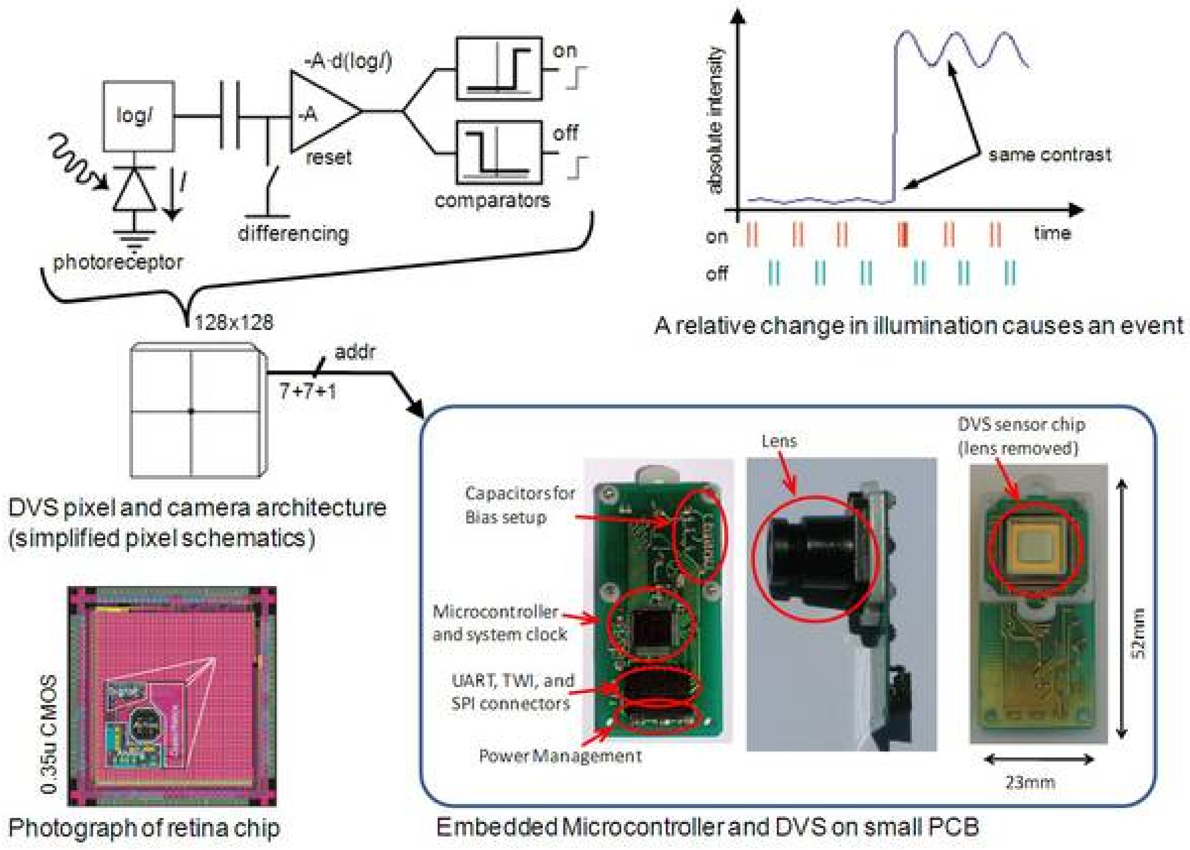
A lot of applications in control and automation rely on vision sensors. Most of these sensors produce frame-based data therefore limit the reaction to synchronous processing at a given frame-rate. The data received this way is highly redundant, because even unchanged pixels have to be transferred in any frame.

The retina in human brain consists of neurons, which produce impulses whenever they detect sufficient brightness changes. This impulses are than processed by further layers of neurons, so that we can recognize objects and movements. Embedded dynamic vision sensor (EDVS) use the same principles and implement them as circuits on silicon chips. A single pixel of such sensor contains a circuit which is capable of providing an impulse, whenever a change in brightness is detected.

2.1.2 Functional principles of EDVS

In a simplified way the circuit in EDVS pixel works as follows: a voltage in a capacitor, connected to a constant voltage source through a photo diode, is compared to a sampled voltage in another capacitor. If the difference between these voltages exceeds certain level, than an event is generated and the voltage in the measure capacitor is sampled to the sampler capacitor. This principles are shown in the image **??**

The sensor communicates with a microcontroller using address event representation (AER) protocol. The impulses are send immediately as messages on an



log/
-A·d(log/)
-A
reset
on
off
comparators
differencing
photoreceptor
128x128
addr
7+7+1
DVS pixel and camera architecture (simplified pixel schematics)
absolute intensity
same contrast
time
on
off
A relative change in illumination causes an event
0.35u CMOS
Photograph of retina chip
Capacitors for Bias setup
Microcontroller and system clock
UART, TWI, and SPI connectors
Power Management
Lens
DVS sensor chip (lens removed)
52mm
23mm
Embedded Microcontroller and DVS on small PCB

Figure 2.1: Functional principles of the embedded dynamic vision sensor [?].

asynchronous bus. As far as the fire rate of the impulses from a single neuron is sufficiently lower than the bus speed, time division multiplexing can be used.

The AER protocol in EDVS is based provides reaction whenever a pixel changes its brightness using coordinates of the pixel as the address. There are also two kinds of impulses, which indicate positive and negative brightness changes respectively.

The amount of data from a EDVS chip is an order of magnitude smaller, than that from a comparable frame-based sensor. Due to low redundancy and asynchronous communication the algorithms for processing AER data are an order of magnitude faster, compared to frame-based image processing. [?].

2.2 The Android Platform

2.2.1 General Information

In production environments, EDVS events will be processed on specialized platforms suitable for robotic applications. However, we still investigate the capabilities and potential use cases of EDVS. Therefore, a flexible, affordable, and easy to use demonstration platform is needed.

When we started our project, the only way to visualize EDVS data was using a PC running Java software. This demonstration environment was not flexible enough for example for live demos at conferences. A more mobile solution was needed.
The Android operating system runs on a large number of mobile devices. The same code can be executed on small smartphones, as well as on large, 11.5” tablet devices.

Android’s source code is available at `http://source.android.com`. This means that students and researchers can examin the code, or might even compile a custom version of the system.

Most programs on Android are written in Java. A language that’s in widespread use in academia. For performance-critical applications, parts of an Android app can be written in C++ as well.
As Android gains more and more support from developers, documentation is widely available. Researchers extending our EDVS application are unlikely to encounter problems that are not yet discussed somewhere on the web.
The official documentation at `http://developer.android.com` covers most needs. Tutorial-style text books are available from various publishers.

Of course, Android is not the only platform for mobile application development. So why is it particularly suited for the task of EDVS visualization, compared to for example iOS and Windows Phone?
The most important factor is the wide availability of devices with different form factors and in various price categories. Entry-level Android phones sell for a third of the cheapest devices running iOS. Anroid tablets are available with 7” and 11.5” screens; some devices have a USB port that can provide power for the vision sensor.

Android is programmed using Java, a language that is widely used in academia. Deploying software to a device is very simple. Code can be deployed to an unlimited number of phones and tablets. No registration for a developer program is necessary.

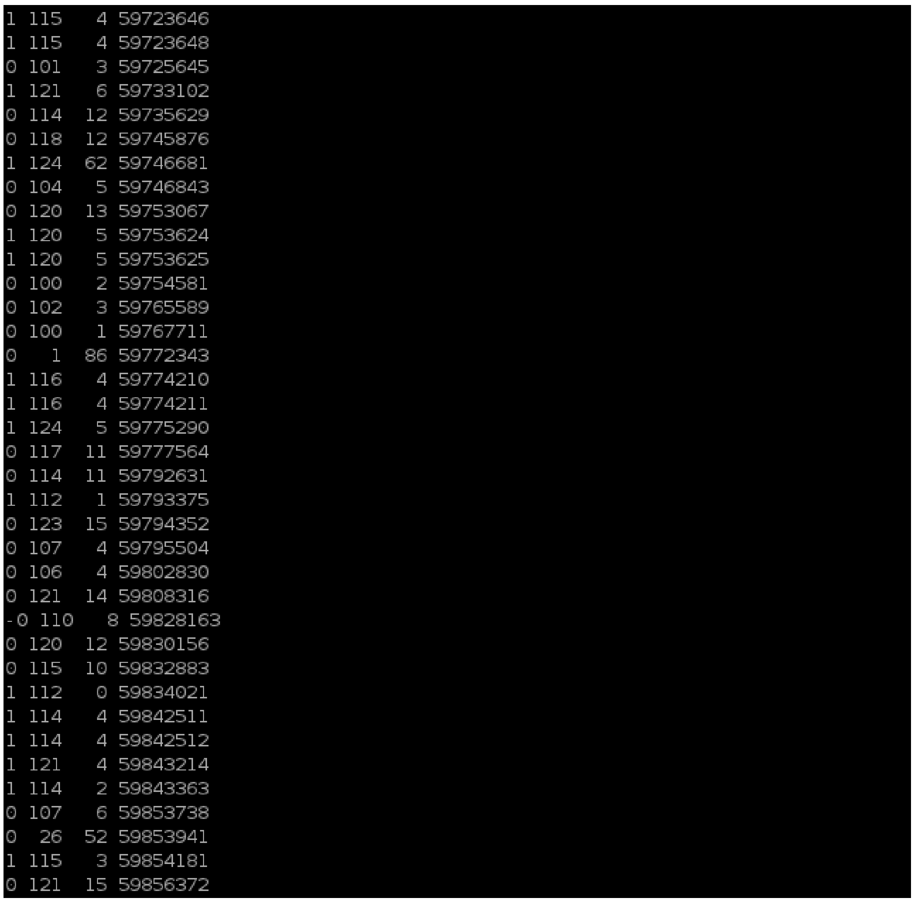
2.2.2 Support for EDVS needs

Some of Android's features make it particularly well-suited for the use together with EDVS Sensor for mobile demonstration purposes.

Android devices can act as a WLAN access point The EDVS sensor can connect to WLAN access points. An android device can create such a hotspot. For demonstrations, we are therefore not dependent on foreign WiFi infrastructure. Nor do we have to set up additional WiFi hardware.

Full support for java.io and java.nio APIs The EDVS module act as either a TCP (port 56000) or UDP (port 56002) server. Once connected to the WiFi access point, applications running on the Android device can subsequently initiate a socket connection to the sensor.
All Android devices support network IO trough the standard java.io and java.nio. IO code written for one device will thus run on all other available Android devices. Furthermore, the standard java code could even be shared between Android and desktop clients.

High-Performance graphics With several thousand events per second, the visualization of EDVS events can become computationally expensive. Android supports various graphics APIs. Among them flexible and portable solutions using software rendering (Canvas API), as well as standard-compliant implementations of the hardware-accelerated OpenGL ES and OpenGL ES 2.0 graphics APIs `http://www.khronos.org/opengles/`.



```
1 115   4 59723646
1 115   4 59723648
0 101   3 59725645
1 121   6 59733102
0 114  12 59735629
0 118  12 59745876
1 124  62 59746681
0 104   5 59746843
0 120  13 59753067
1 120   5 59753624
1 120   5 59753625
0 100   2 59754581
0 102   3 59765589
0 100   1 59767711
0   1  86 59772343
1 116   4 59774210
1 116   4 59774211
1 124   5 59775290
0 117  11 59777564
0 114  11 59792631
1 112   1 59793375
0 123  15 59794352
0 107   4 59795504
0 106   4 59802830
0 121  14 59808316
-0 110   8 59828163
0 120  12 59830156
0 115  10 59832883
1 112   0 59834021
1 114   4 59842511
1 114   4 59842512
1 121   4 59843214
1 114   2 59843363
0 107   6 59853738
0  26  52 59853941
1 115   3 59854181
0 121  15 59856372
```

Figure 2.2: Example of text-formatted EDVS events

Chapter 3

Application Architecture

The EDVS application is composed of several building blocks. This chapter devotes a section to each of them. Details on algorithms, the most versatile kind of building block can be found in the next chapter that serves as a manual on how to develop additional algorithms for the application.
Events are either received from a sensor or read from a file. The two methods are abstracted by the EventSource interface. After the algorithms processed events, they can draw their output (for example detected lines) on an EdvsView. There is usually only one view and one source active at the same time. However, multiple Algorithms will usually be activated and will draw their visualizations on top of each other. The final building block is the controlling activity. It is in the middle of all other mentioned components and dispatches sensor and drawing events.

The EDVS application is written for devices running v2.3 (Gingerbread) or higher. We omited all features that are exclusive to the newer Android versions in order to ensure the app runs on most of the devices that are in use today.
Some of the features introduced in versions 3 and 4 of Android have been backported however and can thus be used on earlier versions as well by including the android-support-v4 library in the project. Using this library, we can make use of newer Android features such as fragments. The use of those features will make it easier to port the application to newer Android versions, especially when creating a customized version for tablet devices.

3.1 Controlling Activity

The main activity first inflates the main layout. It consists of a quadratic EDVSView on top and a ViewPager underneath. The ViewPager is then filled with layouts from individual algorithms. This way, the user can horizontally page through the UI controls and two special pages: The source selection and the color configuration.

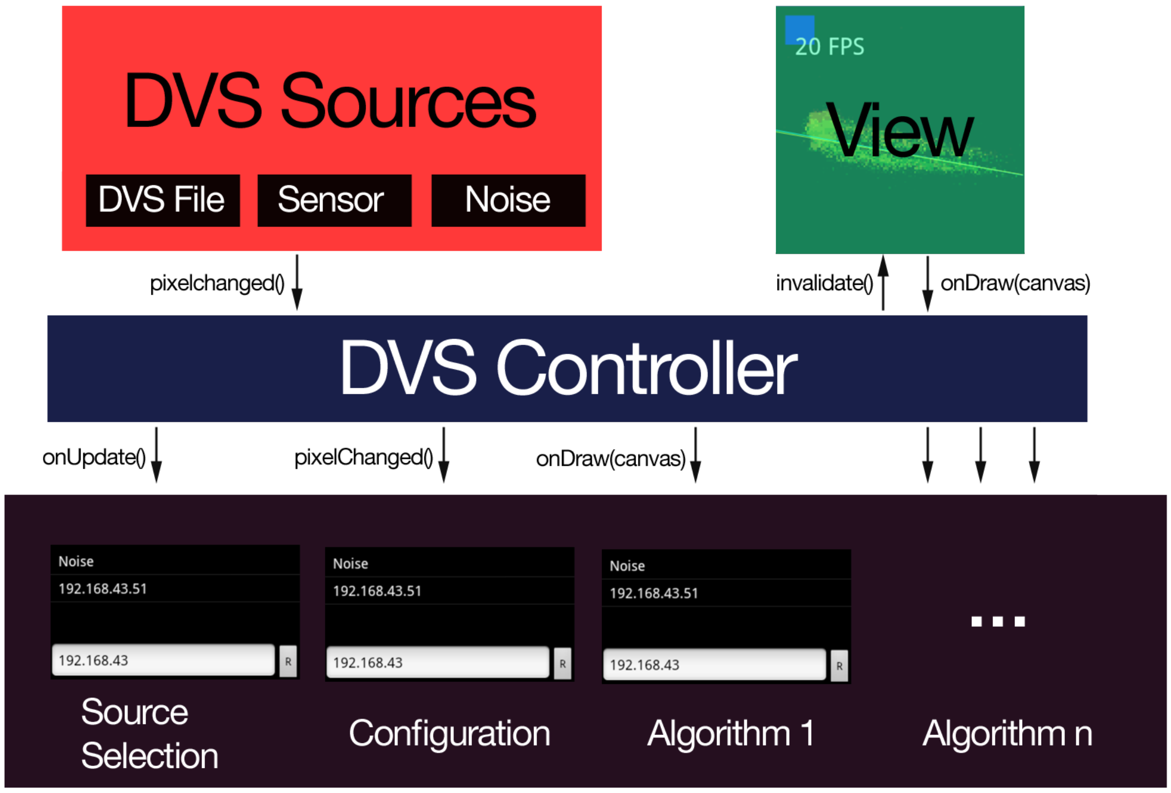


Figure 3.1: Architecture of the EDVS Anroid Application

The main activity registers itself as a listener for EDVS events coming from the currently active event source (the sensor by default). It then dispatches all event notifications to the currently active algorithms.

In order to support algorithms that must periodically update their view. For example for animations, the main activity creates a seperate thread. This loop thread sleeps for a specific interval and the calls the update() method on all activy algorithms. After that, it invalidates() the EDVS View and thus causes the View to redraw itself.

onDraw() events of the EDVSView are forwarded to the main activity. The activity then dispatches the onDraw() calls to all active algorithms so they can overlay their custom visualizations on top of the sensor output.

3.2 Event Sources - Sensor and Files

All visualization and calculations depend on an event source that issues the EDVS events. An event is defined by a coordinate pair (x and y) and a polarity flag that specifies if the observed point just got brighter (true) or darker (false).

We implemented two sources: The first one connects to an EDVS sensor on a given IP address and TCP port. It receives sensor events in a binary format and translates the incoming byte stream into method calls on the main activity.
The second implemented event source reads recorded event streams from a file. We defined a binary file format that stores a timestamp, coorinates and polarity for each event. The player reads an number of events into a playback buffer and calls a method on the main activity at the right point in time. Once the playback buffer runs empty, it reads another batch from the stored file until the EOF is reached.

3.3 Visualization with the Canvas API

The aforementioned EDVSView doesn't perform any drawing itself. It merely lays itself out quadratically, and forwards all calls to its onDraw() method to the main activity() where they get forwarded to the algorithms.

All algorithms have the full power of the Android Canvas API at their disposal. They can call methods to draw rectangles, lines, or arbitrary polygons on top of the EDVS sensor output.
The canvas further supports the drawing of text, bitmaps, and arbitrary transformation and blending effects.

It may take some time to read the canvas documentation found at `http://developer.android.com/guide/topics/graphics/2d-graphics.html`, however, the use of the basic android canvas for visualization poses the least restrictions on developers of algorithms and let's them create almost any 2D graphics effect that is supported on the Android platform.

3.4 Algorithm Fragments

Algorithms get notified about all interesting events happening in the system. The main activity calls the for every incoming event from the sensor, calls an update method in regular intervals, and requests the algorithm to redraw it's visualization whenever the EDVSView is redrawn.

Algorithm Developers can furthermore specify a GUI layout that is displayed in the ViewPager underneath the quadratic sensor visualization.
The next chapter contatains detailled information on how to implement additional algorithm fragments.

Chapter 4

Integration of Additional Algorithms

Additional Plug-in algorithms must have some kind of managed lifecycle. A method that tells them to acquire resources and, another that tells them when to release them. For example, an Algorithm that logs to a file must know when to open and when to close this file.

Furthermore, Algorithms must be able to extend the UI of the application with their own UI elements. We want all algorithm UIs to be part of a single, coherent Android applicatation. At the same time, we want to be able to add and remove those additional UI elements in a modular way, so all algorithms must define their UI as a self-contained, independent building block.

With the release or Android 3.0 early in 2011, Google introduced just what we needed: Fragments. Reusable bundles of application logic and user interface. All Algorithms that work on EDVS data subclass the abstract Algorithm base class which in turn inherits from the Fragment class.
In this chapter, we first give a brief overview on the behaviour of fragments. After that, we describe how the Algorithm class extends Fragment in order to provide all infrastructure that is needed by custom EDVS algorithms.

4.1 The Lifecycle of an Algorithm

This section describes the individual lifecycle methods that algorithm developers might want to implement.

onCreateView() This method is called once the UI of the fragment must be displayed. E.g. when the UI is about to shup up in the ViewPager below the EDVS view.

For very simple UIs, it is posible to programatically create a layout and add a view to it here. The more common approach however is to define all required UI controls in an XML file. onCreateView() has a LayoutInflater parameter that can be used to inflate the layout that has been defined in XML.

onCreateView() is where the UI should be initially set up. ClickListeners should be defined here. Once the UI is set up, the layout that was defined in XML is returned to the calling function.

onDestroyView() Here all allocated resources must be released. Any changes of the user, e.g. changed algorithms parameters should be saved, for example using the Android preferences class.

4.2 Receiving Events from Sensor and Recordings

pixelChangedEvent() notifies the algorithm about any events that were generated by the event source (sensor or recorded file).
As onPixelChanged is called from the thread that handles networking or file access, no long-lasting calculations should be performed in implementations of this method.
Algorithms that must perform complex calculations should save pixelChangedEvents to a temporary buffer that will then be read by the implementation of the algorithm's update() method.
As pixelChangedEvend() is called from the network thread, Android will crash if the algorithm tries to access the UI from implementations of this method.

4.3 When to Perform Calculations

Algorithms must implement the abstract update() method. It is called in fixed intervals and - unless the framerate is running low - will always be followed by an invocation of onDraw().

Algorithms should perform all calculations in the update() method. Due to being called periodically, it is a good place to perform animations. The visualization of the raw sensor data for example uses the update() method for the fade effect on previously lit pixels.
paragraph As update is called from a seperate thread, it is illegal to directly access UI elements from the Update method.

4.4 Drawing the Algorithm Output

onDraw() hands the algorithm a canvas on which it can visualize the results of the performed calculations. In order to not negatively affect the framerate or UI responseiveness, onDraw should only have to look up results of the algorithm that have previously been calculated in update(). onDraw() is callced from the main thread (UI thread) of the application.

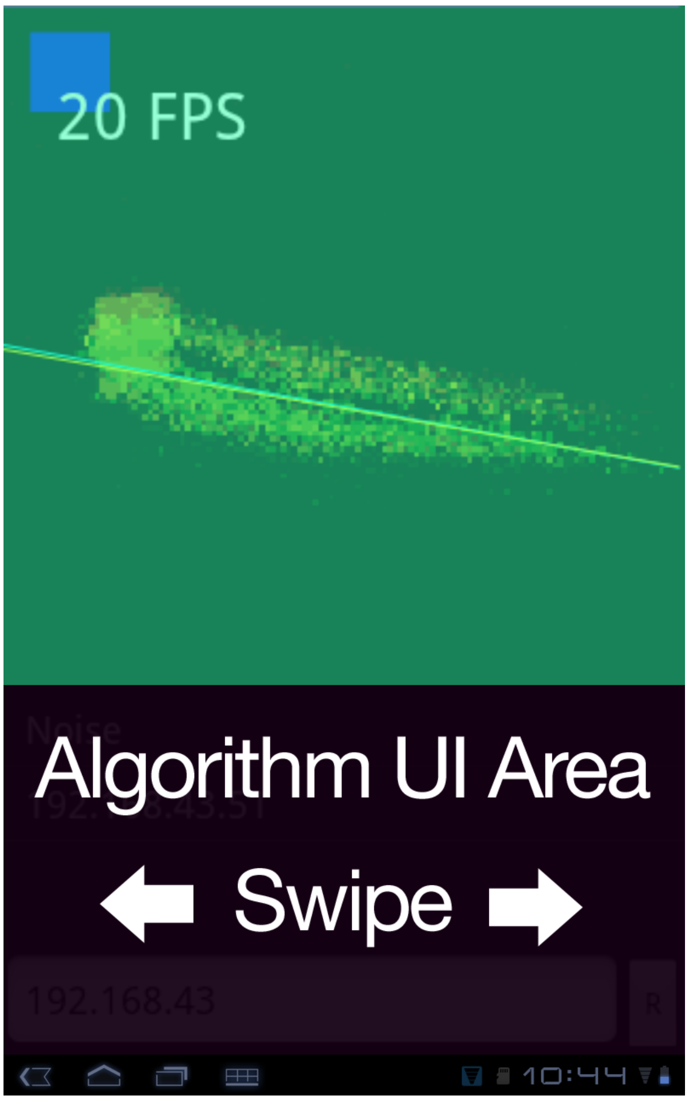


Figure 4.1: The EDVS Events and algorithm visualizations are shown in the quadratic view on top. The UI pane below displays configuration menus and the UI elements of each algorithm.

Chapter 5

Conclusion and Outlook

At the final stage of our project we took two algorithms, specially designed to process the data from EDVS and implemented them in the Android application. Both algorithms are intended to be used in real-time applications e.g. pencil balancing system and swarm of autonomous helicopters.

Line-tracking algorithm approximates a line position according to EDVS data and optic flow algorithm recognizes movement and estimates the velocity. Both algorithms were already implemented on a PC version of the Application.

Transferring algorithms to mobile application required only minor changes in their code, since both applications are based on java. In PC application both algorithms process new events directly after they occur. This approach is not suitable for mobile application, because it results in a slow-motion effect in visualization when too many events occur. We implemented a buffer class for storing the events, so that they can be processed during update() method. Apart from this aspect no changes to algorithm were required.

Both algorithms worked well in the Android application and no instability or frame rate decrease were observed. This proves that Android based devices are good platforms for demonstrating EDVS and algorithms for event-based image processing. The visual output of both algorithms are presented in the images **??** and **??**.

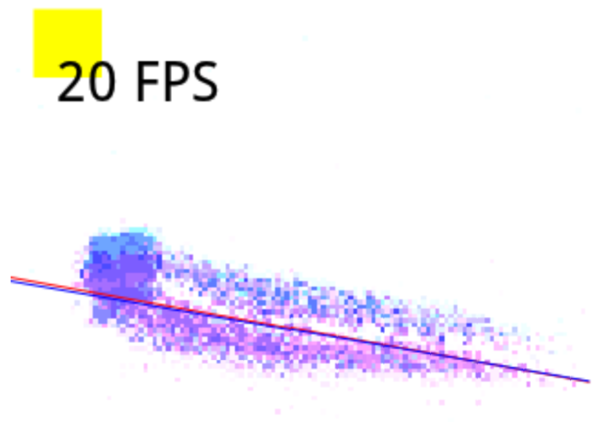


Figure 5.1: Visualization of EDVS data and line tracking algorithm in the Android application (colors inverted).

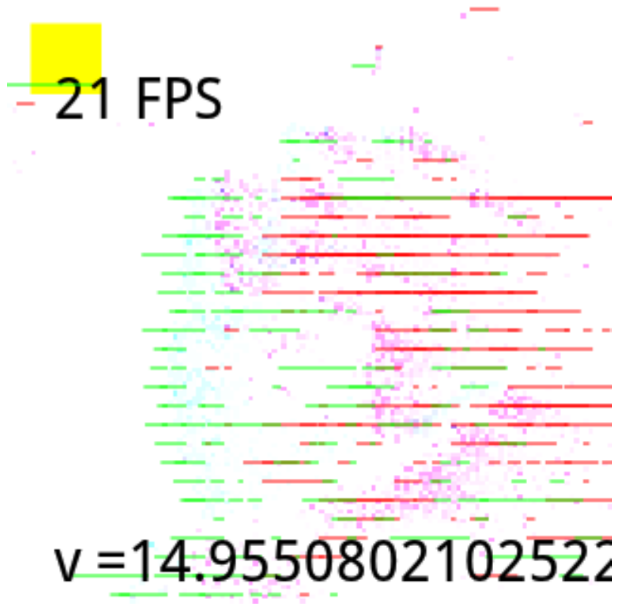


Figure 5.2: Visualization of EDVS data and optic flow algorithm in the Android application (colors inverted).

List of Figures

Bibliography

[Hei06] Daniel Heiss. Neuromorphic chips in avlsi und das adress event representation protokoll. 2006.

[LPD08] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruc. A 128 128 120 db 15 us latency asynchronous temporal contrast vision sensor. 2008.

[Sen] An event based dynamic vision system. `http://www.lsr.ei.tum.de/research/research-areas/systemic-neuroscience-and-biomedical-engineering/an-event-based-dynamic-vision-system/`.