Stereoscopic real-time interface to camera-equipped robot

PRACTICAL COURSE

submitted by
Stefan Urban

NEUROSCIENTIFIC SYSTEM THEORY
Technische Universität München

Supervisor:	Dr.rer.nat. Christoph Richter
	Dipl.-Inf. Nicolai Waniek
Final Submission:	07.07.2015

Abstract

An OmniRob is packed with 7 eDVS cameras covering every viewable direction. In this project, software was developed to allow a user with a head mounted display to watch the eDVS images from the robots perspective. This was achieved by writing a client-server application that transmits the events from the OmniRob to a users PC with a connected Oculus Rift. The problem that the cameras overlap an the borders was not addressed.

Contents

1 Introduction

1.1 Task description

An already operational OmniRob is equipped with an ARM based computer unit and 7 embedded dynamics vision sensor (eDVS[1]) cameras which cover a 360 degree viewing angle. The goal is to allow a user wearing a head mounted display (Oculus Rift[2]) to view the events from the robots perspective.

For this purpose the camera's signals have to be efficiently transported to a PC. From there on, the Oculus SDK[3] has to be used to visualize the data.

Since the cameras do have overlapping areas at their borders, a mechanism should be implemented which quantifies this overlap and joins two neighboring images seamlessly. (Image Stitching)

As the robot has a set of wheels, a user should be able to drive around using a connected joystick or gamepad.

1.2 Task Areas

The project can be split up in three different major parts:

- **Server**: Software on the robot's ARM computing unit, which manages the eDVS cameras, movement and connections to clients
- **Client**: Software on a user PC, which connects to the server, receiving the eDVS events, handling the visualization via an Oculus Rift and dealing with a plugged joystick or gamepad
- **Image Stitching**: Mathematical and computer vision background research regarding the optimal combination of two overlapping (event) images

2 Hardware - Spezifikation

This section describes the already available equipment with which the project was completed.

2.1 OmniRob and ARM-Board

The OmniRob is a development platform used at the NST. This version is equipped with a computer based on an ARM processor running Ubuntu. Connectivity is

[1] `https://www.nst.ei.tum.de/en/projects/edvs/`
[2] `http://www.oculus.com`
[3] software development kit

provided by an integrated WLAN module which automatically connects to the NST robot network. It can be accessed with the IP `10.162.177.202`.

Additionally 7 eDVS cameras are mounted on top of the robot. The orientation of the individual cameras can be seen in Figure 2. Six are arranged equally distributed horizontally and one is on the top. So all possible viewing angles, which are not into the floor, are captured.

All devices (cameras and robot control) are connected as usb serial devices and accessible through `/dev/ttyUSB0` to `/dev/ttyUSB7`.

- OmniRob: `/dev/ttyUSB0`
- Top camera: `/dev/ttyUSB1`
- Other cameras: `/dev/ttyUSB2` to `/dev/ttyUSB7`

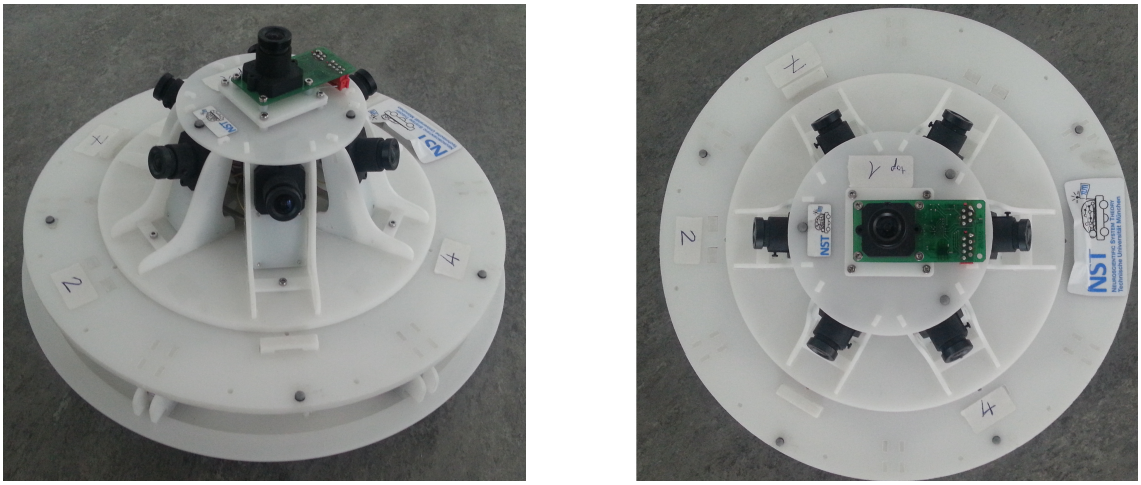

Figure 1: 7 eDVS cameras mounted on top of the OmniRob

2.2 eDVS Camera

The embedded dynamic vision sensor is a special camera that does not capture image frames but detect illumination changes asynchronously. It can create an event every 15 μs up to 1 $Mevents/s$.

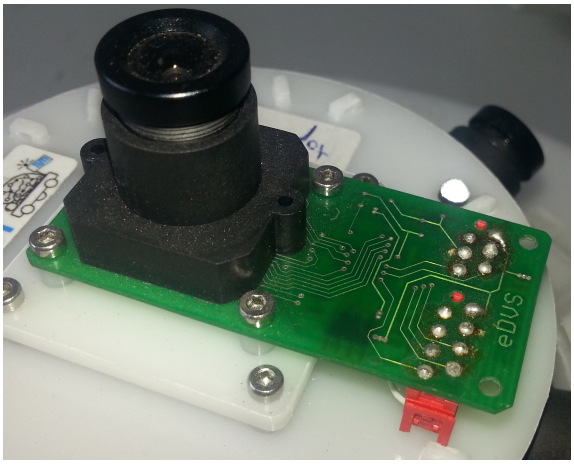

Figure 2: eDVS camera

2.3 Oculus Rift

The Oculus Rift is a head mounted display. With the help of a high resolution display and special lenses a virtual reality should be created.

Due to the technical boundaries of the hardware solution, many problems like the chromatic distortion are solved using software. Therefore the creators of the Oculus Rift provide a SDK which covers these problems almost automatically. The client application only has to connect to a daemon software called `ovrd` that provides all necessary information to use the head mounted display.

As for now, the Linux support for the Oculus Rift is postponed[4], so the already released version 0.5.0.1-beta will be used for this project. It is working well but has one problem in particular which prohibits usage on a PC with Intel Graphics Adapters.

Figure 3: Oculus Rift - Development Kit 2 (DK2)

[4] `https://www.oculus.com/en-us/blog/powering-the-rift/`

3 Basic Software Implementation

The software is separated into an application on the robot (server) and an application on a PC with a connected Oculus Rift (client). This section covers some general design aspects regarding the software structure and transmission protocols which are used in both programs.

3.1 Applications

Everything inside the software is organized in threads, called applications. They can run individually and interact but do not depend on each other. For debugging purposes individual applications can be excluded from execution by simply not starting them.

3.2 Dispatcher

To simplify the applications, they normally are completely isolated from each other. So they do not share the same variables and so on. But of course they need to communicate at certain points. This job is carried out by the dispatcher.

The dispatcher accepts so called events from all applications and decides to which subscriber (also an application) this event will be redirected. All there has to be done is to define by hand which event will be redirected to which subscriber.

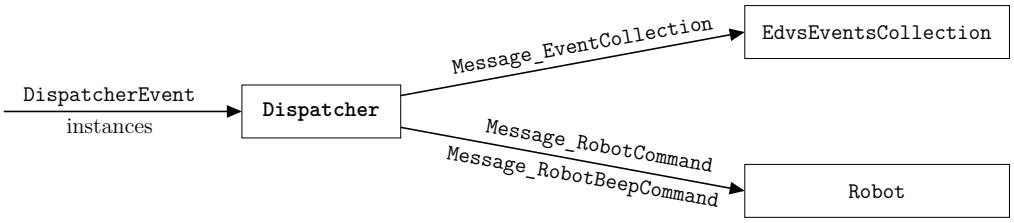

Figure 4: Schematic of dispatcher function in server software

In this project, this concept is mostly used to determine the recipient of TCP messages. Dependent on their content, they are only of importance for some applications, so only they get notified about a message retrieval.

A practical example taken from the server process can be seen in Figure 4. Incoming TCP messages are packed as generic `DispatcherEvents` with a certain type id. Messages containing eDVS event information are passed to the event handling. Those regarding robot movement and beeping are redirected to the robot application.

3.3 Transmission Protocol

Every transmission is carried out over a permanent TCP connection. To simplify development, the boost library[5] is used. This is a collection of a variety of predefined software functions and algorithms that tries to give well written solutions to common problems. The only thing one has to do, is to handle the incoming and outgoing data, everything else will be taken care of.

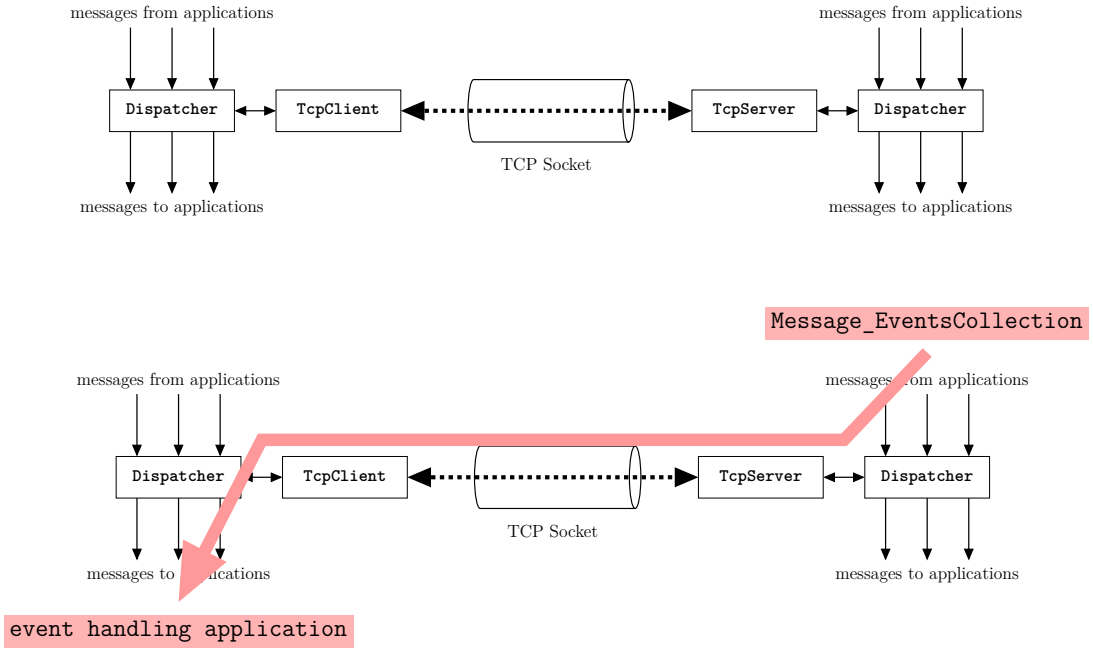

Figure 5: Top: high level representation of transmission path - Bottom: Example flow of a event from the server side (right) to a client (left)

Figure 5 shows a typical message flow from the server to a client. The application receiving the eDVS events from the cameras packs those into an object and hands it over to the dispatcher. Because the `TcpServer` is enlisted as listener for network messages, it will receive the object and send it over the TCP socket. On the client side the `TcpClient` will receive this message and hand it over to its own dispatcher. All applications that did subscribe incoming `Message_EventCollection` objects will be called.

Message format

Every message consists of three basic elements:

- body length in byte (4 bytes)
- message type (1 byte)
- actual data (variable length)

[5] `http://www.boost.org`

The body format is specified by different message classes. They have to implement an abstract `Message` class to be transmittable. Figure 6 shows a list of available messages which will be described below.

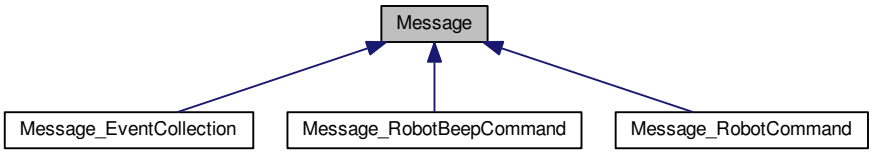

Figure 6: Inheritance diagram for class Message

3.3.1 `Message_EventsCollection`

This message is used by the server to let the clients know about recently occurred eDVS camera events. It is sent periodically and can contain multiple events at once. To reduce overhead only the timestamp (64 bit variable!) of the first event is stored completely, the following contain only a duration between the first and their own timestamp (16 bit variable). This reduces the needed bandwidth for time information nearly by 75 %.

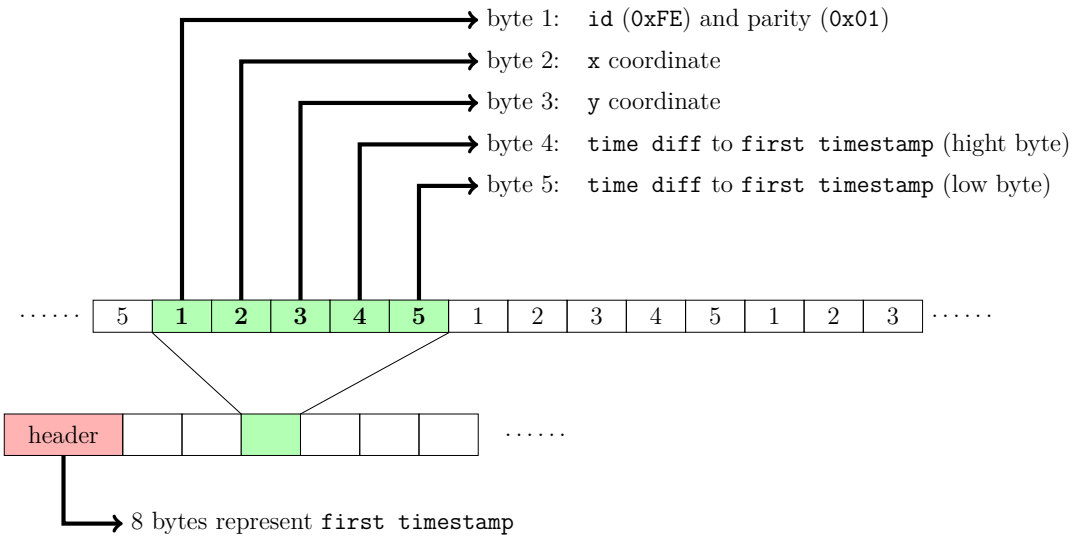

Figure 7: Format of `Message_EventsCollection`

Because it was too time intensive, the decoding and reconstruction of eDVS events was a problem. This was solved by using packed structs.[6] With these, the encoding and decoding could be efficiently implemented with simple memory copy functions.

[6] `https://gcc.gnu.org/onlinedocs/gcc/Common-Type-Attributes.html#Common-Type-Attributes`

3.3.2 `Message_RobotCommand`

This command signals the server process on the robot to initiate a movement. The serial OmniRob interface requires speed in x and y direction and rotation speed.

3.3.3 `Message_RobotBeepCommand`

For testing the connection without e.g. actually actuating any movement, the server can execute a robot beep command.

A peculiar feature of this message is that it does not have a body. Since there is not value to transmit it is not necessary.

4 Server-Software

The server software consists of the following applications:

- 7 **serial ports** to communicate with the eDVS sensors
- **TCP server** which handles client connection attempts
- Periodic **event transmission**
- **Robot movement** handling

4.1 Serial ports

For the connection to eDVS cameras there exists a library by David Weikersdorfer called **edvstools**.[7] But due to some unexpected noise issues an own implementation is used.

Basic function either way is like this:

1. open connection
2. send "E+" to start event transmission
3. read 2 bytes
4. is highest bit of first byte high? if no, read 1 byte and go back to 3.
5. create new event object and populate with data
6. queue event for transmission
7. go back to 3.

Message format for these two bytes is: `1xxxxxxx pyyyyyyy` with p (1=brighter, 0=darker) being parity and the many "x"s and "y"s the coordinates.

[7] `https://github.com/Danvil/edvstools`

4.2 TCP Server

The Boost.Asio library[8] offers an easy way to realize client-server-communications. For each single client a `TcpSession` is created and will handle the message flow. One server can have more than one client. They will receive all messages simultaneously.

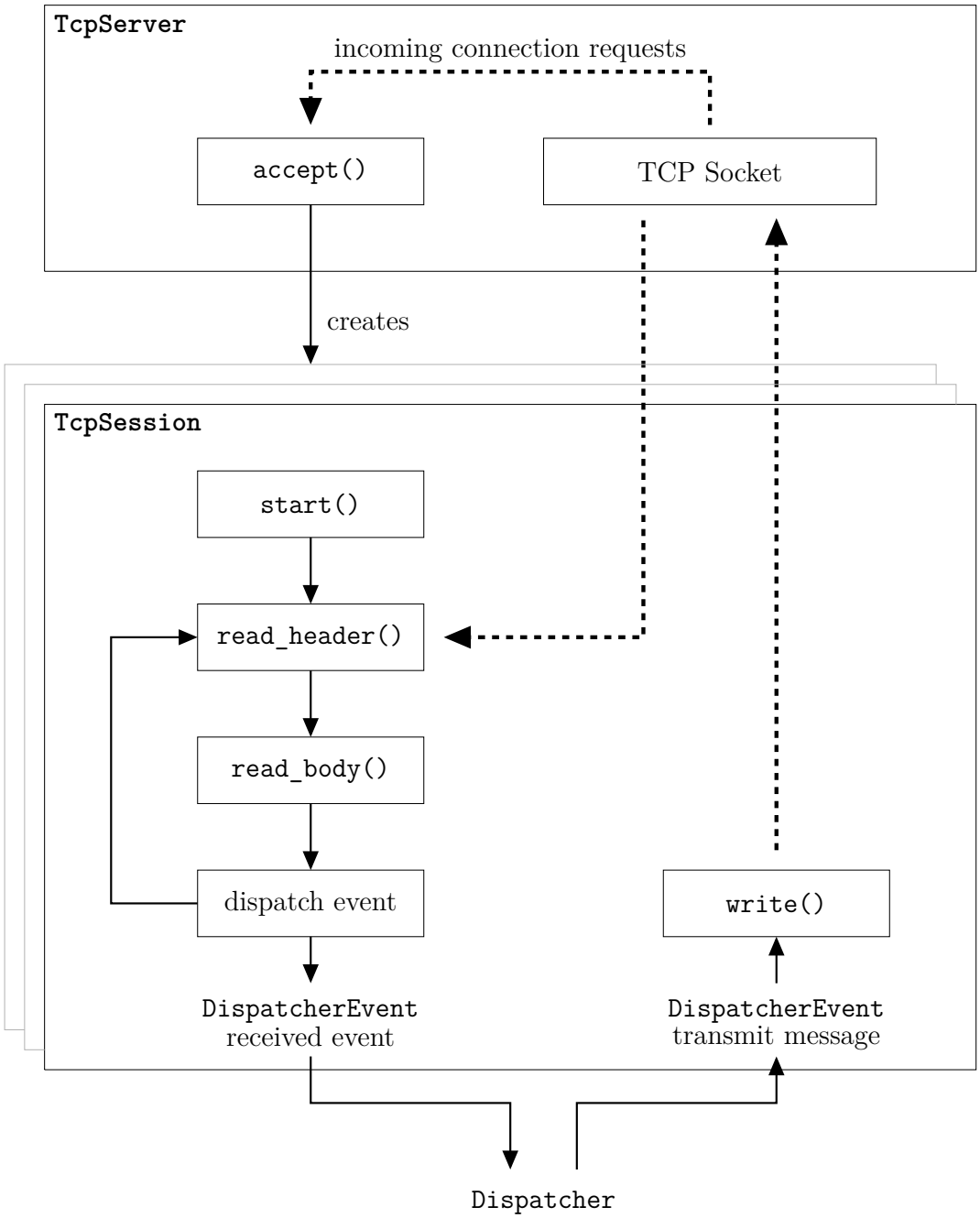

Figure 8: Flow chart showing a single TCP connection on the server

Figure 8 shows how the `TcpServer` creates a `TcpSession` each time a new client

[8] `http://www.boost.org/doc/libs/1_58_0/doc/html/boost_asio.html`

requests a connection. After that, all communication with other applications will be done by the dispatcher. After instantiation the session will start reading the incoming bytes from the TCP socket. The first bytes to arrive will be the body length (see section 3.3 for reference). After that the body is read and handed over to the `Dispatcher` as `DispatcherEvent`. Finally the `TcpSession` goes back to waiting for the next message.

4.3 Event transmission

The serial port applications (section 4.1) will fill a existing buffer that will be periodically packed into an `Message_EventCollection` object and sent to the clients.

Main problem is the number of events can get to high. The bottleneck is not the sending part but as you can see later on the receiving part. Therefore at this point a limitation for events per cycle is introduced. Many eDVS events will be dropped, but this has no big influence on visualization.

4.4 Robot movement

The robot movement handling need three parameters to function:

- Speed in X direction
- Speed in y direction
- Rotation speed

After receiving a command from a client, it is translated into a serial request and sent to the OmniRob.

To prevent damage to the robot due to malfunctioning transmission paths, these commands have to be updated periodically. If none is picked up for some time, a security stop kicks in and sets the movement to zero.

5 Client-Software

The client software consists of the following applications:

- **TCP Client** to connect to server
- **Visualization** of eDVS events
- **Joystick** connection and interpretation

5.1 TCP Client

The structure of the TCP Client is pretty similar to the one presented in the server software description (section 4.2). Only difference is the `TcpSession` is not created by a connection attempt but simply on startup.

5.2 eDVS Event Visualization

All events should be displayed as little squares. Because the client in running on Linux, everything will come down to using OpenGL. But this is not done directly.

5.2.1 Oculus Rift Framework

First of all, the characteristics of the head mounted display have to be considered. Many of these have to be implemented in software, like the right lense distortions. To save time and frustration, a already quite finished framework is used. It was created as part of an upcoming book about the Oculus Rift.[9]

An example project of this framework was extracted and implemented as application into the client program. The part with the actual scene rendering was heavily modified to satisfy the specific demands.

The framework itself is implemented also for use in Windows, but all self written code is not prepared for that.

5.2.2 Graphics Library

All code regarding OpenGL was written with the help of OGLplus[10] which is a C++ wrapper. This brings various conveniences object orientation offers with it.

For the implementation the following aspects are important:

1. Definition of vertices and their attributes
2. Vertex Shader: Code that is executed for each vertex
3. Fragment Shader: Code that is executed for each pixel

Vertices and their attributes
A vertex is a 3D point in the scene. Besides its position it can have different attributes like in our case: camera id and color (parity).

Vertex Shader
In the VS, the event position (x, y) is translated into model coordinates (X, Y, Z)

[9] `http://www.manning.com/bdavis/`
[10] `http://oglplus.org/`

in the 3D world. Depending on the camera id, the events are shifted by a multiple of 60 degree in horizontal direction.

Fragment Shader
The FS simply assigns events with parity 1 a red color and the ones with 0 green.

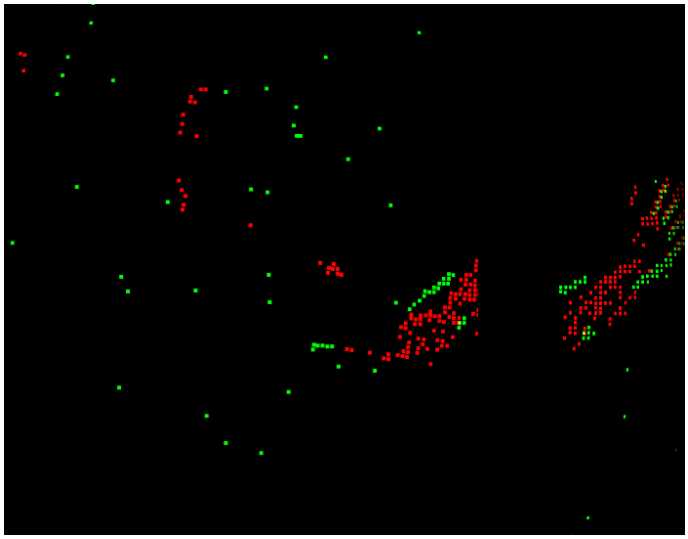

Figure 9: Screenshot from the Oculus Rift displaying a moving hand at the border between two cameras

5.2.3 Transformation of the event stream

There are different possibilities to alter the eDVS event stream:

At event receipt
New events are handled by `EdvsEventHandler::event()`. Inside this function, event based transformations are possible.

Before rendering
Before the beginning of each frame rendering the `EdvsRiftApp::update()` method is called. It copies all currently available events at once. Transformations regarding a whole set of events are possible in here.

After rendering
The method `EdvsRiftApp::renderScene()` renders all available events. To apply further transformations, some framework methods can be used. One rotation and scaling operation are already implemented in there.

Inside the vertex/fragment shader
Per vertex or per pixel transformations can be performed in the VS/FS. Data can be provided using uniforms.

5.2.4 What did not work

Till the final solution there were different attempts which all failed, not because they are theoretically wrong, but they implement a very inefficient way to achieve a certain goal. These will be listed here for not working examples:

Camera image as texture
This is not performant because textures are not meant to change over time. As a result, the frame rate dropped to under 1 FPS.

Render a plane 128 x 128 times
Basically this creates as many OGLplus programs as there are pixel. Each of them will load its own shaders, resulting also in big frame drops.

Render a spherical calotte and add color with FS
Create the vertices that represent the eDVS image on a sphere. The color information gets loaded inside a "uniform". Besides the fact, a uniform can not hold this amount of data, the result was unsatisfactorily with less than 500 events per frame possible.

5.3 Joystick Control

For using a joystick or gamepad the Linux Joystick API[11] is used. A simple C++ wrapper class provides access to the joystick events. It does not matter if the user plugs in a joystick or gamepad, as long as there is a device file like `/dev/input/js0` available.

The joystick application handles the connection and reconnection attempts so the user can plug and unplug the device at any time. If the setup is all right, the controllers axis position is obtained and converted to a movement intention. That is transmitted as `Message_RobotCommand` (section 3.3.2) to the server who will then try to perform the requested actions.

5.4 Software Limitations

For various reasons, some features could not be implemented or bug not fixed.

[11] `https://www.kernel.org/doc/Documentation/input/joystick-api.txt`

Configuration
Many configuration flags and parameters can only be found in the source code. A possible solution would be a command line based menu at startup which lets the user adjust them or a simple configuration file using libconfig.[12]

No support for Intel Graphics Adapters
The Oculus SDK has a litte bug in combination with Mesa 3D which prevents the usage of the Oculus Rift with an Intel Graphics Adapter under Linux. Due to the fact that the creators of the VR device postponed Linux development for an indefinite period a fix will not be available soon. Only possibility is to port the software to Microsoft Windows on which the bug does not occur.

No event timing implemented
An eDVS event information always consists of the coordinates, the parity and the time. Because the time has a relatively high resolution in microseconds, it needs much memory (8 bytes). The implementation is fairly simple but was not done due to lack of time. The advantages will be not great because the events are transmitted every 30 milliseconds in a guaranteed order (TCP). On various parts of the software this implementation is already prepared but commented out.

6 Image Stitching

The reason we need some sort of image overlap is simple: Two neighboring cameras capture the same events at their common border. Image Stitching is the operation to join these two images into one.

The first task is to quantify the overlap between two cameras. This can be very difficult due to uncalibrated cameras and other distortions. A trivial approach can be cross correlation. First test brought some promising results:

[12] `http://www.hyperrealm.com/libconfig/`

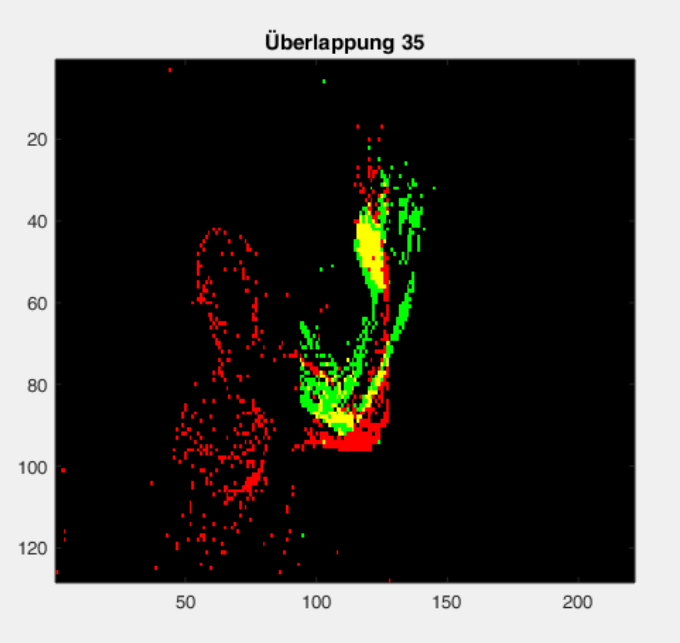

Figure 10: Correlation test of two cameras with an overlap of 35 pixels (red = left camera, green = right camera)

Due to the lack of time, further investigations on this topic were not possible.

7 Conclusion

At the end of this project a solid base was created on which image stitching tryouts can easily be implemented and tested.

The next step would be to including the geometry into the image stitching algorithm. The euclidean transformation between the cameras has to be determined. With that and a simple correlation, the overlapping pixel area can be roughly quantified.

The not further dis task of stereoscopic vision is very difficult to achieve be-cause a 3D reconstruction of single events is needed. But this can not be done for the parts of the images that do not overlap. One possible solution is to double the amount of cameras facing in one direction, increasing the total amount to 14.

Bibliography

[1] Conradt, J., Simon, P., Pescatore, M., and Verschure, PFMJ. (2002). Saliency Maps Operating on Stereo Images Detect Landmarks and their Distance, Int. Conference on Artificial Neural Networks (ICANN2002), p. 795-800, Madrid, Spain.

[2] Axenie, C., Conradt, J. (2014) Cortically inspired sensor fusion network for mobile robot egomotion estimation. Robotics and Autonomous Systems, Special Issue on "Emerging Spatial Competences: From Machine Perception to Sensorimotor Intelligence", Volume 71, pages 69-82.