Oculus Rift Retina Display
Towards Psychophysical Studies

Project Report

Felix Bernhardt, Richard Kurle and Nicolas Scheiner

10/2014 – 01/2015

Table of Contents

1 Introduction 3
2 Experiment 4
3 Technical Realization 9
3.1 Retina Core Application 9
3.2 Graphics User Interface 11
3.3 Server-Client Communication 16
4 Results 17
5 Future Work 19

1 Introduction

The aim of the Retina Rift Project is to conduct a psychophysical experiment gaining insight on how much data our brain needs to perceive depth. One reason to do this is the upcoming use of retinal implants. As these implants need to be powered, the consumption and heat emission shall be as low as possible. An approach to break these limitations is to reduce the data an implant records and processes. In order to shed light on the question how to do this, a psychophysics experiment will be designed, which will artificially diminish visual data for the human subjects thereby trying to find a threshold at which depth perception becomes no longer possible. The detailed current test setup for this will be discussed in Section 2.

To better understand, which steps and actions were necessary to realize a working test setup, a short description of the previous test setup and the program that was given to us will be rendered. The hardware setup consists of two eDVS sensors which are mounted on fixed positions on a plastic plate. This plate can be mounted on the Oculus Rift DK1, but has to be held manually while in use. The program provided a window which showed a specifically for the oculus rift adjusted video. There were three basic modes given, mode 0 simply shows the live stream of the two eDVS sensors, mode 1 additionally records this data and mode 2 could play a file recorded in mode 1. If parameters had to be changed (for instance a filename), a recompiling was needed. Modes were not switchable. Once the mode is chosen in the eclipse console, no further inputs could be made excluding the zoom in/out and the adjustment of the eye distances. All other parameters were hardcoded. Several features like the rendering loop or save/load settings (program crash) were dysfunctional and had to be fixed. A great deal of the program was written in the main function and even some code blocks were redundant.

In order to achieve the project goal, there were three main technical and one experimental tasks to be considered and worked out. The overall improvement of the retina core application, i.e. add and improve functions, restructure and greatly increase the handling (responsiveness at runtime) of the application. Thirdly, the development of a graphics user interface, i.e. control of the core application, the test execution and documentation via GUI. The third task was the development of an appropriate experiment, the recording of data for test and the test itself with healthy human subjects as well as the evaluation of the experiment results.

2 Experiment

Ehrenstein and Ehrenstein[1] provided a basis for standard procedures in psychophysical experiments, which was taken into account for the experiment design. For recording the test sequences there was a mounting plate („eDVS shield“) available on which both eDVS sensors can be attached in a fixed distance, so that all recordings could be made with the same eye distance. The so created data is being stored simultaneously in two different video files for the left and the right eye.

The experiments shall be easy reproducible, moreover a couple of parameters such as the distance to the test objects shall be evaluated, therefore the aim was to design a system, which could easily but precisely record different test setups.

A basic problem for the test sequences was the choice of objects to be filmed with the sensors. Those objects should have a very similar shape in order to minimize the effects of extra parameters, which could possibly help identifying depth in the sequences. Furthermore the influence of the absolute size should be eliminated or at least made measureable, thus objects that exist in varying sizes were favored. For this purpose three sets of Matryoshka dolls were used (see below in Figure 1), so that it was possible to make recordings with up to three dolls of the same size.

Figure 1: Matryoshkas

For just depth recognition, the actual project goal, static 3D pictures would usually have been enough, but with the eDVS sensors only showing the changes between two scenes video

[1] Walter H. Ehrenstein and Addie Ehrenstein, 1999: Psychophysical Methods, In: U. Windhorst and H. Johansson: Modern Techniques in Neuroscience Research, Springer, Berlin (1999), pp. 1211-1241

sequences have to be recorded, which must result from a relative movement between sensor and recorded object. The original solution was to mount the sensor shield on an Oculus Rift device (see above) and create a quasi-statical picture with small head motions by repeatedly viewing the same setup. The connection to the Oculus Rift was improved with some laser cut plastic pieces, however first test showed that it was very hard to create reproducible test scenarios this way. That is why for the next recording session a Lego Technic shaker was built, which can actuate the sensor shield at a constant pace. The speed of the shaker is continuously adjustable with a variable ratio transformer, which was set to a constant value.

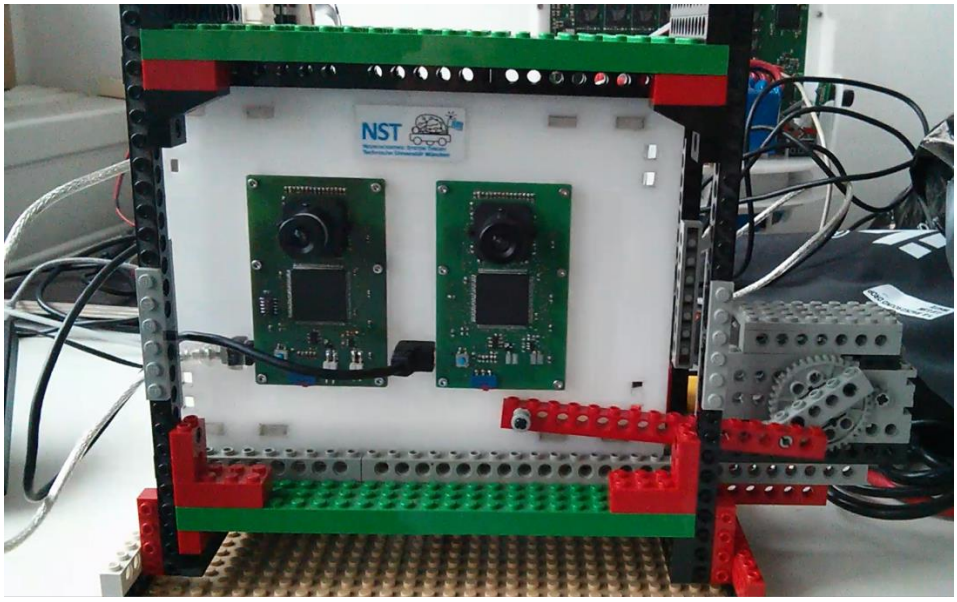


Figure 2: eDVS shield

Another big problem when recording objects was, that the bottom edge of those objects often gave a very strong cue about the spatial depth of an object. Therefore these objects had to be uncoupled from the ground and should also be made adjustable in their heights. An easy way to accomplish both goals was to use small transparent plastic containers, which cannot be detected by the eDVS sensors. Figure 3 shows the outer dimensions of one of those boxes and a stack of them respectively.

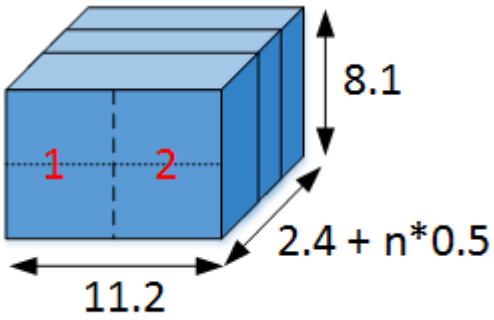


Figure 3: Stacked box dimensions (in cm)

In order to gain a multitude of results with only few recordings, a number of three dolls per video sequence was chosen to be able to get more differentiated results than „in the front/back“ as it would be the result of only two objects.

The entity of all adjustable parameters and scenarios for left, middle and right dolls are:

- Two different distances between eDVS shield and upfront object row
- Six levels of spatial depth referring to the first object row
- Two-stage elevation and resulting angle of the sensor shield
- Three different heights for the test objects
- Three different doll sizes

The exact meaning of these parameters is visualized in Figure 4 and Figure 5.

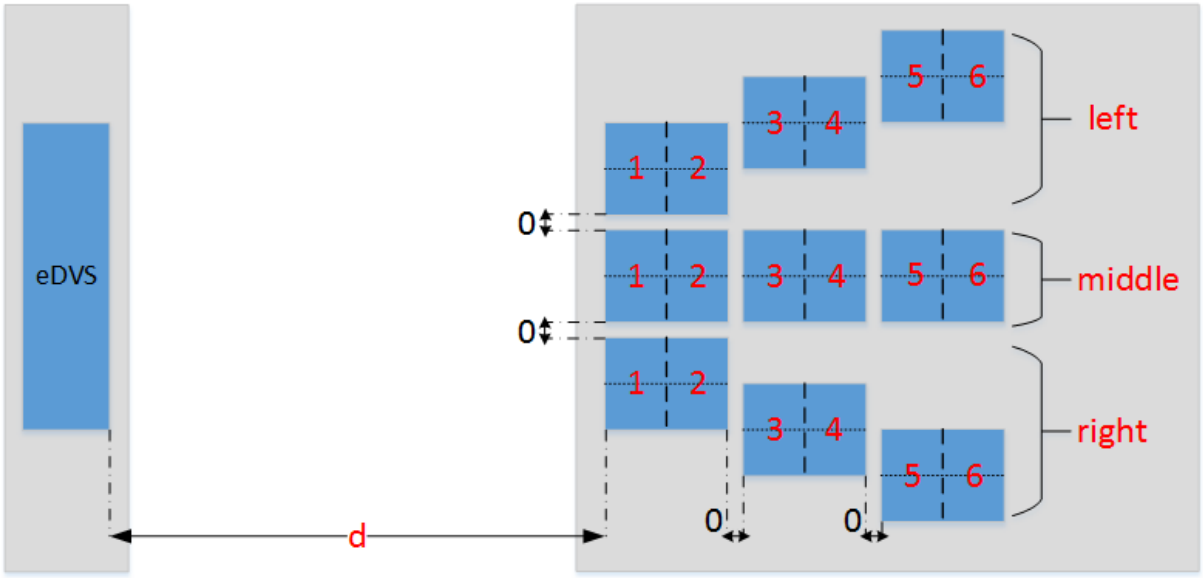


Figure 4: Experiment schematic (view: above)

As can be seen in Figure 4, the left, middle and right position rows have to diverge when increasing the distance to the sensors. That makes it impossible for the upfront dolls to hide parts of the others behind them and so indicating their spatial distribution.

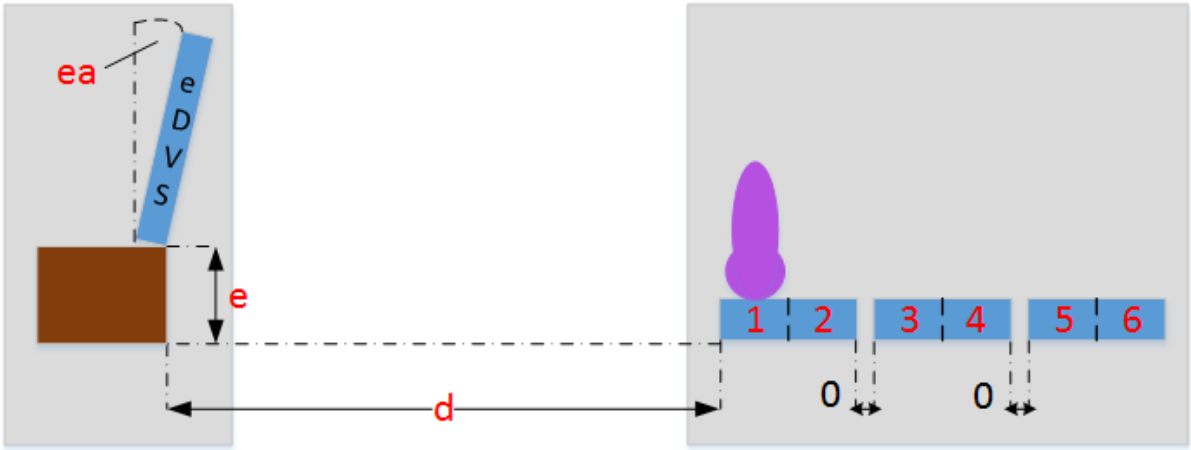


Figure 5: Experiment schematic (view: side)

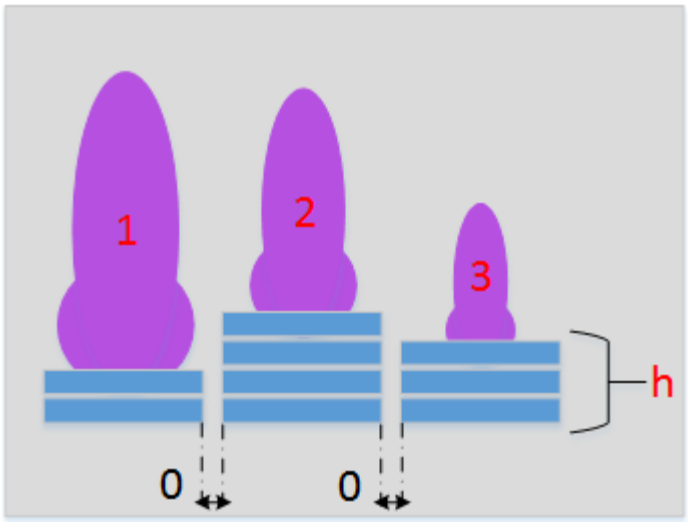


Figure 6: Experiment schematic (view: front)

From the many possible setups, 48 suitable settings have been chosen. When selecting those sequences there was a focus on not having too similar scenarios as they would be i.e. for permutations of “left“/“middle“/“right“ for otherwise equal setups. Moreover an aim was to avoid most redundant information as it would occur i.e. for setups 111 and 555 (related to the object depth). Nevertheless the chosen settings are still similar enough to finally make conclusions about critical parameters.

The recorded files have their setup parameters directly encoded in their name. For example „e1d30l1m2r3h211f123“ indicates „elevation 1, camera distance 30, left pos. 1, middle pos. 2 …“. A definition of all these parameters can be found in Table 1. A list of all used record files can be found on the CD.

Table 1: Filename coding

e - Sensor Elevation	
e1	eDVS shaker not elevated, angle ea = 0°
e2	eDVS shaker elevated 28.2cm, angle ea = 15°
d - Sensor Distance	
d30	45cm sensor distance to position 1
d60	60cm sensor distance to position 1
l / m / r - Object Distance	
l / m / r	Indicates the position of the object for left, middle and right column according to Figure 4
h – Object Height	
1 / 2 / 3 / 4	From left to right: 1: 2 boxes = 2.9cm 2: 14 boxes = 7cm 3: 26 boxes = 11.3cm 4: 18 boxes = 7.9cm
F – Object Size	
1 / 2 / 3	From left to right: 1: second largest puppet of the set 2: third largest puppet of the set 3: fourth largest puppet of the set

As it is the actual goal to find a critical level of possible data reduction the 48 tracks were reprocessed with a MATLAB script that randomly drops a specified amount of data points. This process was executed for five different levels of data reduction resulting in 240 recordings to be presented to the test subjects. With a small first test group it was postulated that usually five seconds should be sufficient time to make substantiated statements about the depths. Adding another five second safety buffer, recordings of ten seconds each were made. The 240 videos were then separated again in three different groups in order to have one test subject not work with the Oculus Rift for more than 20 minutes, because it might happen that people get sick for longer tests.

The association of the test sequences with the three user groups was made in such a way that the single recording parameters were equally distributed over all three groups, so that even with only few test subjects all parameters can be covered. The order of the video sequences was randomly determined by a Python script. That script ensures that two sequences that only differ in their level of data reduction cannot follow after each other, thus preventing undesired biasing effects.

3 Technical Realization

The operating system was Linux Ubuntu (64 Bit) 14.04. LTS. A dedicated graphics card (Nvidia GTX 650 was used) was needed, due to the lack of the OpenGL support by Intel Integrated Graphics. The retina core application runs in Eclipse Luna SR1. The GLFW Framework was used in Version 3.0.4, the Oculus Rift Framework (OVR) in Version 0.3.2. Further dependencies and instructions can be found in the install readme.

As a consequence of the above mentioned state of the previous program, the core part, the render loop and control flow, was entirely redone, additionally structured and then extended. The program runs stable. Debug messages and more detailed documentation were added to the code.

3.1 Retina Core Application

One of the goals of this project was to create a flexible core application, which can be used by a GUI, keyboard inputs or any other interface. This application should grant the possibility to flexibly decide during runtime whether to record data with eDVS sensors or show already recorded data stored in files. Also methods for controlling the state of the video (Pause/Play/Stop) or for changing the file to be displayed and others should be available and work stable. As described in section 2, the code of the predecessors of this project was already able to record from eDVS or play from a text file. This code consisted of the class eDVSGL and a main function. Inside the main function, the whole initialization of the oculus rift, openGL framework, the eDVSGL object and objects of other libraries, as well as the rendering process was implemented. As a consequence, this code was very static in functionality and difficult to modify. Given that the experiments required the program to be flexible, it was necessary to restructure the code into separate function blocks that can be used independently. These functions should provide the possibility to react to user inputs (e.g. changing the mode from recording to displaying, changing the displayed file, switching the monitor to the oculus rift, etc). Since the code of the project's predecessors was not ordered by corresponding libraries and it was not clear whether this seemingly arbitrary ordering was useful or required, the best strategy was to restructure the code step by step and test it each time afterwards. Thus, the code of the retina core application had to be reworked completely in order to handle various commands coming from the GUI application.The code of the retina core application has been adjusted following the subsequent steps:

- Delete unnecessary or wrong parts of code: The provided code had some flaws such as function declarations inside functions which could be resolved quickly. On the other hand there were some redundant pieces of code which were more difficult to identify as such
- Split the main function into an initialization function and a render loop: This step helped to understand the provided code by revealing dependencies and showing which of the various globally defined variables were actually needed in the render loop

- Correct the render loop: The provided code's render loop was not able to run at 60Hz. In fact, the frame rate was not even constant. Thus the loop had to be re-implemented to work at a constant frame rate. Furthermore, an fps counter has been added to verify the constant frame rate of 60Hz by displaying it in the window
- Rearrange the code to have coherent code next to each other: This step helped to analyze the code furthermore and find out which pieces of code actually belong to the same origin such as the same library or class. It had to be taken care that no potential dependencies of variables or objects were violated
- Encapsulate coherent code into functions
- Port the code into a class and switch to an object orientated programming paradigm: A class that manages all the variables and objects needed permanently by the program has been implemented. Also the initialization and render functions have been added to this class. As a consequence, this class contains most of the code that has been rearranged and implemented until this step.
- Add another class to store and handle modifiable parameters concerning the display: The class "ParameterManager" has been added to manage certain parameters flexibly by calling the corresponding method instead of modifying the values of these parameters by a hard-coded factors at different positions in the code. Some of the parameters just mentioned are given subsequently:
    - translate-back offset: a zoom factor
    - viewport offset: determines the eye distance of the oculus rift which is different for every person
    - decay factor: a factor which determines the color decay of an eDVS-event from one frame to the next frame
- Add various functions needed to control the program during runtime:
    - set mode: Change the mode from and to any of the possible modes (Play from a file, show live data from eDVS-Sensors, show and record live data from eDVS-Sensors)
    - set control: switch the video state to Play, Pause or Stop
    - set file: Change the file to be displayed or the filename of the new recordings
    - Use Oculus: Toggle using Oculus Rift as monitor for the displayed data or the "real" computer monitor
    - Set Color: Toggle the event colors between black/white and red/green
- Add a class to handle multithreading of the server and the core application: The server and the core application run in two different threads. Adding multiple mutexes seemed to be inappropriate in the case of this application, because many and especially large pieces of code would have to be mutually excluded. It was more sufficient to implement another class called RetinaServerInterface, doing nothing more than storing the server messages and setting flags for pending messages which then could be called from the thread containing the RetinaManager object

During all of the preceding steps, it had to be taken care of invalid program states which (theoretically) should never be reached and invalid function calls (e.g. commands from the

GUI) which would lead to a program crash. Also debug messages which can be turned on and off have been added to help searching for bugs.

Some examples of known bugs which have been solved are given in the following:

- Calling functions of the RetinaManager class directly from the server which runs in a different thread must be avoided: added the RetinaServerInterface class to solve this issue
- Sending the control command “play”, given a wrong filename or a file which is already at end of file: If a file cannot be opened correctly, the program will not change into the state “play”, instead a debug message will be displayed. When a file is at end of file, the program will switch into the state “stop”. Whenever the control command “play” is received while being in state “stop”, the file to be played will be re-initialized
- Trying to record data, when eDVS sensors are not connected: Changing into one of the modes which uses the sensors is not possible, if the sensors cannot be detected. Instead, a debug message will be displayed and the mode will change to an “error mode”.

3.2 Graphics User Interface

One of the main project goals was to make the orginal C++ program accessible via a graphical user interface (GUI), which should also handle all the new tasks, which arise due to experiment execution.

The implementation was made with the open cross-platform application framework Qt, which provides an easy way to create GUIs, comes with a lot of online support and potent object classes, which promised to make the whole design rather simple and accessible from different operating systems. For a database textfiles are being used, because they are easy to create and evaluate not only with C++ but also Matlab and Pyhton scripts for data evaluation. An attempt to merge the GUI files with those of the main program in one big project failed, therefore communication between those two programs works with a client-server application as described in the previous chapter.

While programming the GUI particular attention was paid to make operating the system easy, intuitive and robust against bad input. The two main tasks of the GUI are first controlling the Oculus Rift Ecplise program via the client described in chapter 3.3 and second to manage the experiment execution, thus user administration, parameter adjustment and data recording and storing.

Table 2 gives a brief overview of the most important classes and functions of the GUI:

Table 2: Most important GUI classes and functions

guistartup	Startup window of GUI
testWindow	Main window for experiment execution
usermanagement	Window for user administration
testexecution	Window for test block handling
applicationhandler	Interface between Qt classes and client
alreadyrecorded / closewindow / saveclosewindow	Warning windows to confirm user input
recordNames	Linker arrays for test sequence assignments to the right files
client	Client communication

When starting the program the first window being displayed is an instance of the class „guistartup“. From this window it is possible to control the most basic functions of the main program, such as track recording or rendering or direct rendering of the current eDVS view without saving the data. The last option from here is to start the test environment, which will close the startup window and lead the user to the test area, where all human subject research is executed from.

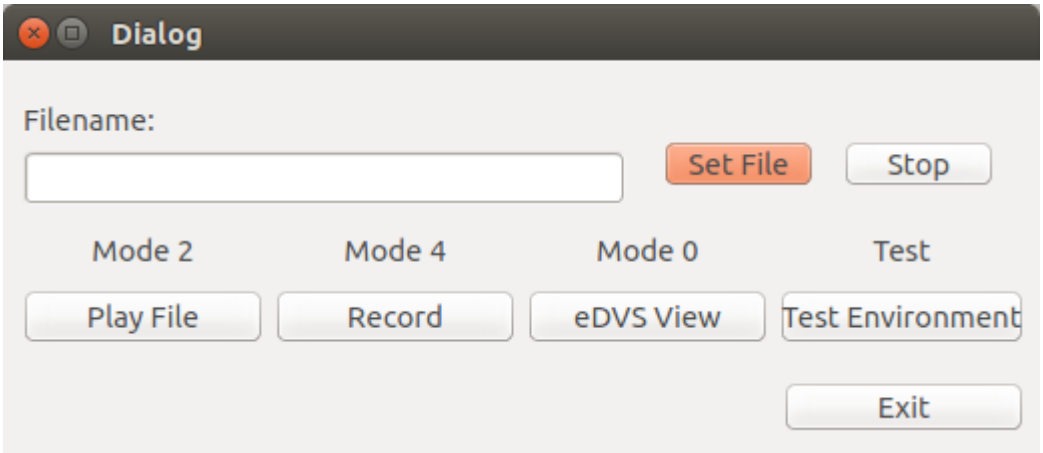


Figure 7: GUI startup window

The main window of the the test environment shows the active user and his or her individual parameters. To create a new user or alter the values of some existing user parameters one clicks „Manage User Profiles“ and a new window with that same name will open (see Figure 9).

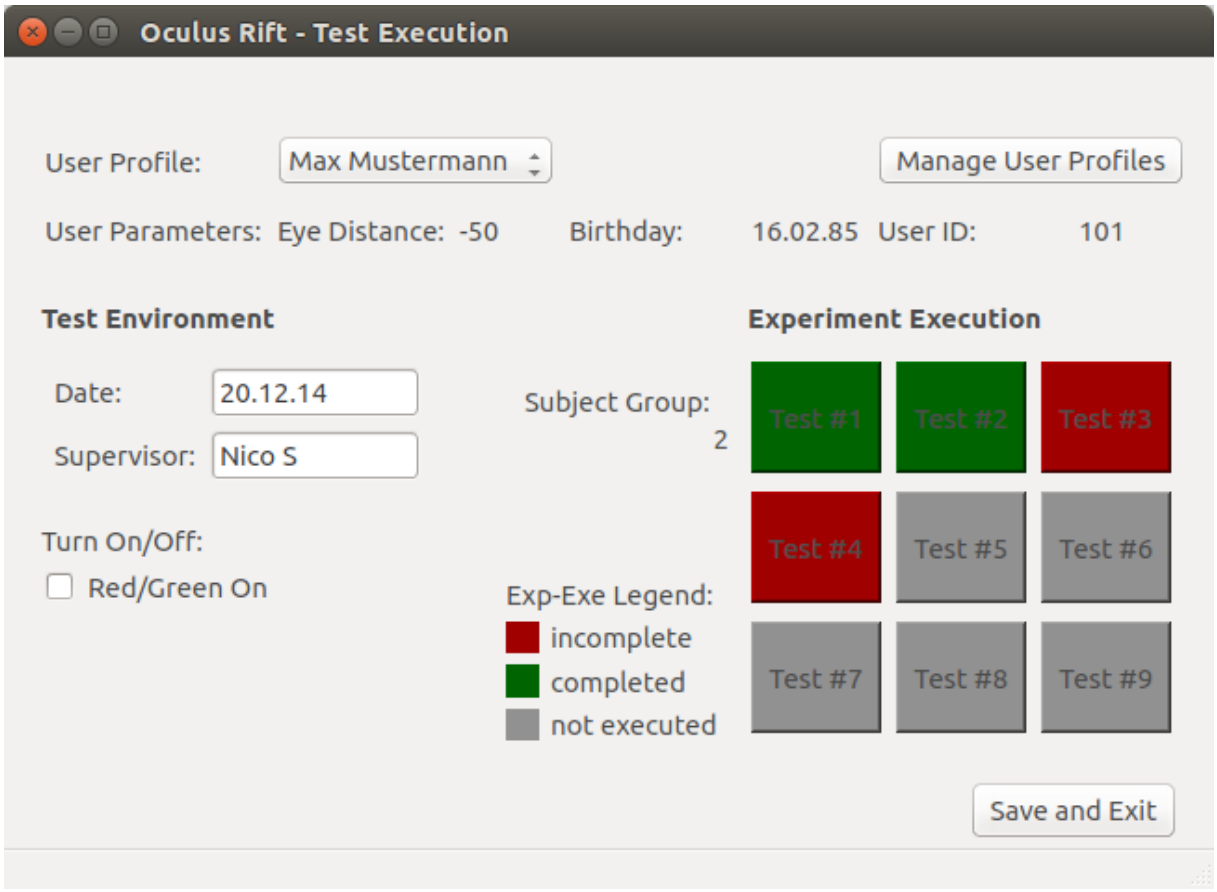


Figure 8: Test overview

In the „Manage User Profiles“ window it is possible to either change the user parameters or create a new user, who will be added to a textfile based database, which stores all the users and their parameters. The important parameters are:

- Name and birthday → for easy user recognition
- Eye distance → for optimal Oculus Rift adjustments
- Subject group → indicates which test videos the test subject will evaluate
- User ID → automatically created by the programm for easy subject references

To measure the optimal value for the eye distance parameter, the subject will see a test sequence that can be adjusted by the supervisor by changing this parameter until no further sharpness improvement can be noticed.

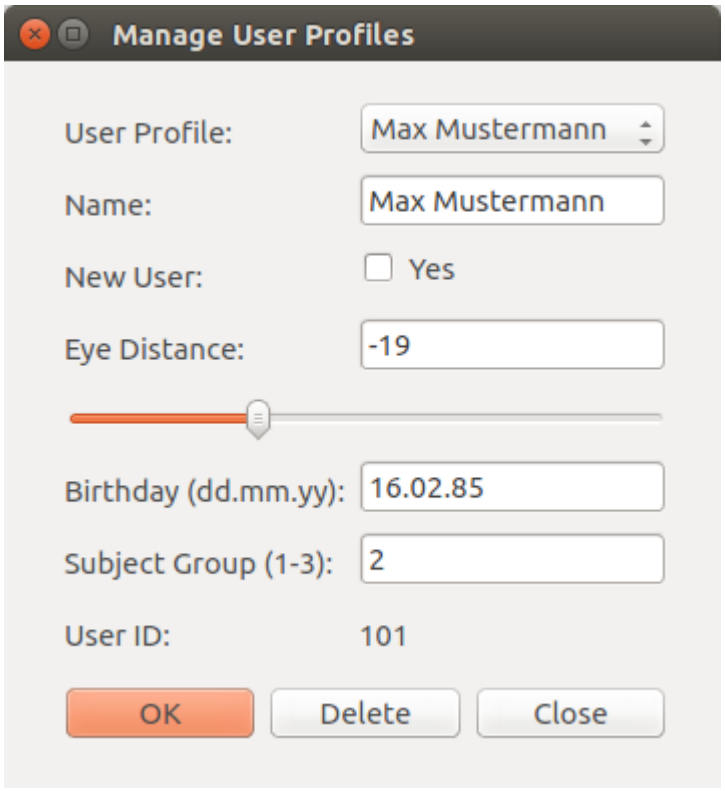


Figure 9: GUI Userprofile

Once a user profile has been created, it will be already be selected in the experiment main window. The supervisor only should fill out the two lineedit boxes on the left side of the window concerning the date and the supervisor name, then the testing can begin. The default Oculus Rift displaying is in different gray shades. By selecting „Red/Green on“ the sequences can also be tested with colors.

Nine test buttons on the right hand side start different test blocks, each one containing ten individual video tracks. While the GUI also supports pausing during the test, this multi-block option was chosen, so that it is easier for the supervisor to keep track of the experiment progress and to provide good points for a break or repeating particular sequences.

By clicking on a test button a „Test Execution“ window opens, which guides test subject and supervisor through the test block. After hitting „Play“ the first test sequence in block starts running for ten seconds. During that time the subject has to make and tell his or her decision about how the three figures are distributed over the test area. After those ten seconds the next sequence will start. If the subject does not need that much time, the sequences can be forwarded, if something goes wrong the test can also be interrupted. During the test the supervisor will see both the current test sequence highlighted as well as the remaining time for each sequence. The test results entered by the buttons next to the current sequence display will be written into a textfile after closing the window with „Ok“. If the test is aborted prematurely a warning window will ensure that this input was intentional. If the whole sequence was completed successfully, the according test button will be highlighted green or red if not all but some test results are available.

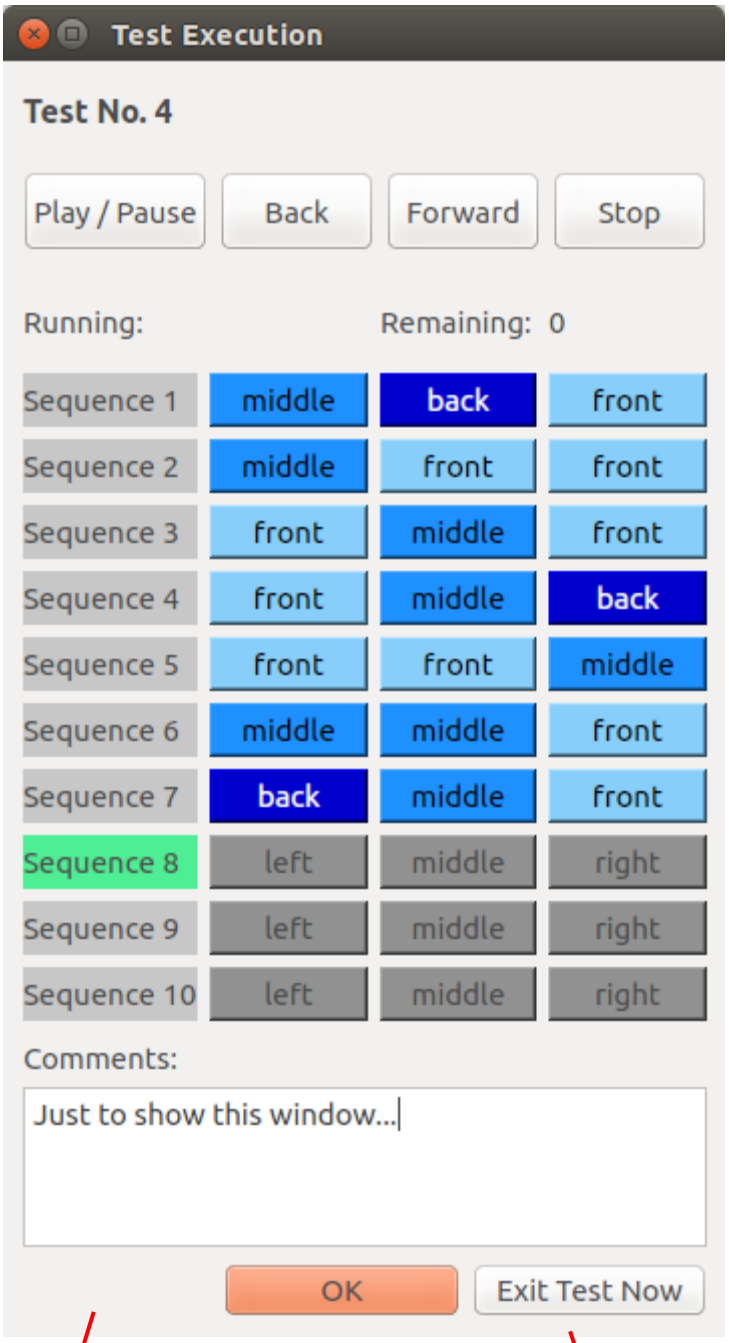
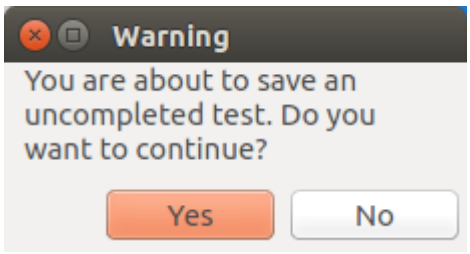
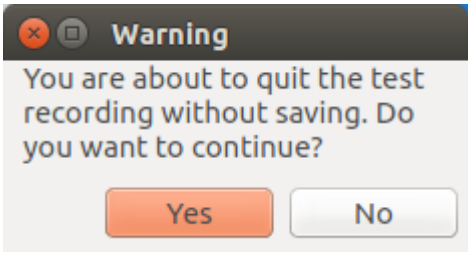


Figure 10: Test execution





3.3 Server-Client Communication

To interconnect the GUI and the retina core application an interfacing function was needed. The reason why the decision fell on the server-client model will be discussed in the following.

At first, there were several potential solutions for the realization of the GUI: displaying the GUI in the window that the core application creates with GLFW, integrating the Qt framework into the retina core application or use inter process communication protocols, separating the GUI from the core application entirely.

Wanting to realize control and monitoring functions with optical feedback (e.g. which test runs at the moment) for the test supervisor, it was not possible to choose the GLFW framework for it has poor high level capabilities creating GUIs. The option to integrate Qt in the core application was discarded for two reasons. Firstly, GLFW and Qt are not - out of the box - compatible to each other, they compete for shared resources. Secondly, the supervisor would have been bound to the workstation used by the subject. For greater flexibility in applications the third choice was taken and expanded to not only support inter process but inter system communication. This was realized by using a TCP server and client.

Both server and client have their own threads. The server additionally dispatches another handler thread for the GUI. The GUI application, which is the client, connects to the hosted server of the retina core application. For a more robust operation only a single connection is allowed by the server. Any further connection tries will be rejected by the host. When the client is connected, the user can use the provided functions in the GUI. The client sends the server the appropriate commands and parameters. As C does not provide a java-like feature for creating and using interfaces, this was done manually. An exemplary message sent to the server would be “-control play”, which starts the render loop of the retina core application. The message structure is basically identical to the command line interfaces used by linux applications. The first parameter preceded by the dash is defined as the command, the second is the parameter of the command. The handler only expects these two parameters. Additional information in the packet will be discarded.

4 Results

To recapitulate the goals described in section 1, the technical goal of this can be summarized in implementing a GUI, reworking the core application and implementing a server/client interface for the communication between those two applications. This part, which made up most of the work could be completed successfully. The GUI application is able to send data via the server/client interface to core application, which reliably reacts to these messages. In more detail, the GUI application is able to change the core application's mode, change the file which is being displayed or change the filename of a file to be recorded, control the state of the video (Play/Pause/Stop) or send a command to toggle between displaying the video on the Oculus Rift or the "real" monitor, connected to the computer. The GUI application also stores the user data and experiment results successfully. This can all be done during runtime, i.e. there is no need to edit and recompile code. Furthermore, the core application runs stable and catches invalid commands from the GUI or the keyboard inputs and handles invalid program states.

The goal for the expeiment, which consists of creating, conducting and evaluating a psychophysical experiment could not be achieved entirely. As described in section 3, a promising experiment has been designed. Also a device for moving the eDVS sensors was built. Data which should be presented to the subjects has been recorded two times. Unfortunately, both times the data was unsatisfactory. At the first attempt, the videos were partly laterally cropped, because the field of view of the eDVS sensors is too small for the width of the scene at a distance of 30cm between the sensors and the objects. This could not be seen wearing the Oculus Rift, which we used as reference when recording the sequences. At the second attempt, the data collected from one eDVS device was horizontally elevated about a few pixels compared to the other device. The cause for this problem has not been found, but the suspicion is that one of the utilized devices is unreliable (e.g. bad lenses or mounted wrong). Due to the corrupted data, the first subjects performed much worse than expected and consequently the experiments were aborted. Because of these drawbacks, there was not enough time to conduct the experiments with new, flawless data.

Some first tendencies we could draw from the partially conducted experiments is that the assumption that the objects are positioned in plane influences the results most. Because the single-colored ground is not visible in the video data, humans apparently assume that the objects must be positioned in plane. Thus when regarding the scene with a slight angle from above, objects that are positioned higher on the screen/retina and which are smaller, appear farther away. During one of the recording session, these effects have been tested and it turned out that even though we know the real scene, by first watching it without the Oculus Rift, we then get fooled by optical illusions. If for example two same objects are used, one is positioned on a higher ground (which is not visible) and one a lower ground, the higher positioned object appears to be positioned farther in the scene and bigger also, because it has the same size on the retina/screen while we think it's positioned farther. It should be mentioned that this has

not been scientifically evaluated yet. In order to make assumptions based on statistic, these video sequences have to be shown to many subjects and evaluated.

Summarizing the results of this project, the software part works properly and a promising experiment has been designed. However new video material has to be recorded, experiments have to be conducted and the results have to be evaluated.

5 Future Work

Much effort has been put into implementing the software which is necessary to conduct the experiments automatically. Hence, it is desirable that this project's result will be used and finally result in a meaningful psychophysical experiment. The most important thing that has to be accomplished is to record new data for the experiment. Then an experiment can be conducted with several subjects and the obtained results can be evaluated afterwards. The experiment itself could be improved as well. It turned out that the objects of the scene (matryoshka) which were assumed to be sufficiently similar (same shape, but different color pattern) might be not similar enough. In fact, the eDVS sensors generate different amounts of events for the different matryoshkas, depending on their color pattern. Thus different objects might suite better in this experiment. Furthermore, the object to object distances could be improved, i.e. all valid positions could have a constant relative distance. Also the program code can be improved. Firstly, the code could be adapted to C11 standard. Secondly the class RetinaManager could be split further, to make it easier to read and understand.