INSTITUTE OF COGNITIVE SCIENCE

Bachelor's Thesis

DESIGN AND IMPLEMENTATION OF A REMOTE CONTROL INTERFACE FOR THE PUPIL EYE-TRACKER

Pablo Prietz

June 3, 2016

First supervisor: Prof. Dr. Frank Jäkel
Second supervisor: M.Sc. Steffen Waterkamp

Design and Implementation of a Remote Control Interface for the Pupil Eye-Tracker

Eye tracking is a popular field of study in Psychology. Its main research tool are eye trackers which record the subject's eye movements and use them to calculate his or her gaze. Pupil Interface, the software I developed in the context of this thesis, is intended to work with the Pupil eye-tracking platform by Pupil Labs (Kassner, Patera, & Bulling, 2016). The Pupil hardware consists of an head-mounted eye tracker which allows the subject to move freely. The open-source software provided by Pupil Labs (Pupil Labs, 2016d) features a flexible and extendable plugin structure as well as a set of communication interfaces. With Pupil Interface, I integrated the existing interfaces in a single Python module that provides a straightforward and high-level remote control functionality.

Acknowledgements

I would like to thank Prof. Dr. Jäkel who supervised this thesis and provided valuable guidance throughout all stages of the project.

Particularly I give thanks to Moritz Kassner, William Patera and the Pupil community for providing technical support and insight into the Pupil software.

Additionally many thanks to Lisa Goerke, Patrick Faion, Babette Winter, Katharina Stein, Greta Häberle, and Nicole Knodel for proofreading this thesis. Thanks to Sebastian Höffner for providing his LaTeX thesis template (Höffner, 2016).

Contents

Chapter 1

Introduction

The investigation of a person's gaze is a popular field of study, e.g. the perception of design (Höning et al., 2008). In order to allow as many experiments as possible to happen, one wants to have a measuring instrument that is affordable and non-invasive. Popular tools to track gazes which fulfill these requirements are video-based eye tracking devices. They use cameras to capture the eyes and track their movement with features like e.g. pupil positions and reflections on the cornea (Purkinje images). The cameras are either be positioned in front of the subject or attached to a head-mounted device. Capturing the eye movements alone does not reveal where the subject fixates. The eye positions have to be correlated with an image of the subject's field of view. A typical solution is either to use a computer screen or a forward facing camera (e.g. attached to a head mount). To calculate the unknown correlation between the eyes' positions and the field of view, the subject is often asked to fixate specific points which are presented to him or her. Using this information one can estimate the subject's gaze. This process is called calibration. Estimation inaccuracies depend on the eye tracker in use and technical details of the calibration's implementation.

Most eye trackers come with their own software. Often the software is proprietary and only provides a restricted access to the eye tracker's data and functionality. In some cases the software is open source and therefore allows full access and the possibility of extending the functionality. The amount of control one wants over the software depends on the usecase. For example, one could only be interested in the data produced by the eye tracker. In this case it is enough for the eye tracking software to use a well documented file format. Another usecase might be the processing of live data, e.g. controlling a mouse pointer using gaze. For this to work efficiently, one needs direct access to the eye tracking software. Having a lot of customizable access is good for use cases with very specific requirements. This might require a long learning period depending on the user's knowledge and experience. Therefore a good software interface needs both, simple high-level functions as well as low-level functions providing customizable access.

This thesis presents my implementation of a high-level remote control interface, called Pupil Interface, for the Pupil eye-tracking platform. Pupil is a mobile head-mounted, video-based eye tracker developed by Pupil Labs (Kassner et al., 2016). It consists of a headset with one or two eye cameras and a front camera. It is light and allows free head movements during usage. This

makes it advantageous for experiments with high mobility requirements. Its software is Open Source, has a modular design and is easily extendable. A detailed introduction into Pupil's design concepts and built-in plugins is given in chapter 2.

The main goal of this thesis was to create a high-level interface which is easy accessible for users without deeper knowledge about Pupil itself. Another goal was to make it easily integratable into PsychoPy experiments. PsychoPy is a software framework which provides tools to build a variety of psychological experiments without the need of advanced programming skills.

The resulting software, called Pupil Interface, is presented in chapter 3. Additionally, I programmed utilities to complete a seamless integration of Pupil into PsychoPy, which are described in chapter 4. These utilities replicate some of Pupil's visual interfaces using PsychoPy mechanisms (e.g. visual stimuli used for the calibration procedure). In combination, Pupil Interface and the utilities create a high-level interface that provides a lot of control and data access without having to deal with Pupil's internal implementation.

Chapter 2

Pupil - The Eye-Tracking Platform

2.1 Overview

Pupil is a mobile eye tracking platform developed by Pupil Labs. It is characterized by a modular design which can be recognized in its hardware and software. The hardware consists of an head mount which supports a variety of exchangable cameras. Section 2.2 provides details on the hardware's composition possibilities. The software consists of two applications—Capture for recording data (section 2.3) and Player for analyzing the data (section 2.4)—whose functionality is built on a simple plugin structure. Each feature is encapsulated in a plugin, which can be loaded and unloaded dynamicly. This means that the user is able to customize the process depending on the task without having to run unnecessary features. This structure increases performance and provides easy extensibility. Section 2.5 provides detailed information on how to implement an own plugin. An overiew over built-in plugins can be found in section 2.6.

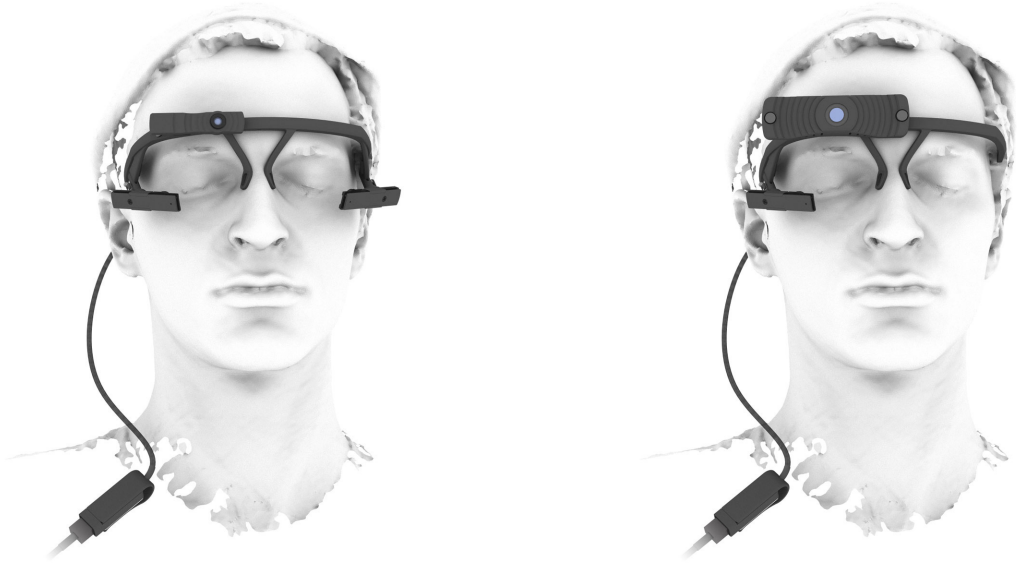
2.2 Hardware

The hardware consists of a glasses-like headset to which all cameras are attached. Eye cameras capture the eye movement and a front camera records the carrier's view. The headset itself comes in different variations depending on the number of eye cameras and the type of front camera. The options for the latter are either a high resolution camera (see figure 2.1b) or a high-speed camera (see figure 2.1a). Binocular eye cameras are only available for the high-speed camera.

The hardware is either available as a finished product (Pupil Labs, 2016g) or as Do-It-Yourself version (Pupil Labs, 2016e) for noncommercial purposes exclusively.

2.3 Capture

Capture is one of two Pupil applications—the second being Player (see section 2.4). It is responsible for processing and storing data during recordings. Processing eye camera footage is a demanding task. For this reason, Capture separates pupil detection and data processing



(a) Binocular eye cameras with high-speed front camera.

(b) Monocular eye camera with high resolution front camera.

Figure 2.1: Pupil Headset variations

into different subprocesses (see section 2.3.1). Processing functionality is separated into plugins. Each plugin takes over a specific task. It is recommended to write a custom plugin to add functionality to Capture instead of modifying existing ones. The following sections describe the general application structure and plugin design concept. These topics are important in case one wants to create a custom plugin.

2.3.1 Process Structure

The Capture application has to process a number of different inputs (e.g. eye camera video). These inputs are independent of each other. Therefore Capture also processes them independently in a concurrent fashion. This means that each input is handled by a different subprocess. To be precise, the *World* process runs all plugins, and for each eye camera exists one *Eye* process, whose task is to detect pupil positions.

The World process depends on the results of the pupil detection. To convey the results, Capture uses a queue which is indepentently accessible for each process. This means that the Eye processes can submit their results to the queue, no matter which state the World process is in. Likewise the World process is able to process queued pupil positions.

The world process also handles the front camera's video stream and the execution of the plugins. Therefore it is the central process for data processing. It is best practice for plugins with high work load to detach their functionality into further subprocesses. This avoids performance issues.

2.3.2 Lifecycle

Event loops in general Programs are lists of instructions which are worked off on execution. After the last instruction is done, the program is terminated. The runtime of programs with a graphical user interface often depend on the user's input (e.g. click on a button). To be able to respond to such events these programs implement a so called event loop. Most of the time it consists of an endless loop which draws the programs content to the screen and processes events on occurrence.

World event loop As mentioned above, Capture consists of different subprocesses. Each subprocess has to manage its own event loop. The event loop of the World process plays an important role for the development of plugins since it manages the plugins.

Plugins are actually objects with a specific list of functions and attributes (see section 2.5). Knowledge over the event loop structure supports the understanding of the plugin object design. Figures 2.2 and 2.3 provide a general overview over the sequence of the World event loop.

1. Import modules and built-in plugins.
2. Initialize the `g_pool` variable—a global container for important variables like Pupil's user directory, list of available plugins or the currently selected calibration plugin.
3. Import custom plugins from the user directory.
4. Define UI event callbacks.
5. Initialize user interface.
6. Start up event loop (see figure 2.3 on p.17).

Figure 2.2: World event loop initialization.

2.4 Player

The Player application processes recordings done with Capture. Its plugins are e.g. able to visualize data like gaze positions on top of the video recording. Navigation through the recording session is done in a frame-by-frame manner. This means that the user can either play back the video in an automatic fashion or one frame at a time. Player also provides the ability to correct, combine, and export data. Its event loop is similar to Capture's World event loop but differs in the way how data is given to the plugins. This means that one needs to design a custom plugin differently depending on if it should run in Capture or Player. More information about this topic can be found in section 2.5.4.

2.5 `Plugin` Class Interface

Both Pupil applications are built on the idea of encapsulating functionality into plugins. Plugins are objects with a common set of functions and attributes. This allows the application to control the plugins independently of their exact functionality. It is necessary to know the technical details of this interface to be able to create a custom plugin.

Every plugin object inherits from an abstract base class called `Plugin`, which defines the smallest common set of functions and attributes each plugin has. All active plugins are managed by a `Plugin_List` object. It manages uniqueness, execution order and clean up of plugins. Those properties are described in more detail below.

2.5.1 Attributes

`g_pool` The global variable `g_pool` is created by the respective main process and serves as container consisting of references to a list of useful variables as e.g. different UI elements. It is passed to each plugin on initialization.

`uniqueness` This attribute describes how many instances of the plugin are allowed to run concurrently. The default value is `by_class`. This means that only one instance of this plugin can be present at a time. If there already is an active instance it will not be replaced. The `not_unique` value allows the plugin to be loaded as often as necessary.

`by_base_class` uniqueness allows only one plugin with a specific base class. This is used amongst others in calibration plugins (see section 2.6.2). In this case there is an abstract `Calibration_Plugin` class which all calibration plugins inherit from. Therefore their base class is `Calibration_Plugin` and only one plugin with this specific base class can be active at a time. If there already is an active plugin with the same base class, `Plugin_List` will unload the active one and replace it with the new instance.

`order` A value between 0 and 1 with 0.5 as default. `Plugin_List` orders all plugins in an ascending manner using this attribute. Consequently, this influences execution order of methods like `update()` (see below). Table 2.1 shows an excerpt of Pupil's built-in plugins and their order.

A plugin should have an `order` of less than 0.5 if it adds or modifies "information that will be used by other plugins and rely on untouched data"(Pupil Labs, 2016c, l. 38), e.g. gaze mapping plugins with an `order` of 0.1. This type of plugins should not edit the `frame` object in the `update()` method.

An `order` higher than 0.5 is recommended for plugins "that depend on other plugins work like display, saving and streaming"(Pupil Labs, 2016c, l. 41).

2.5.2 Override Methods

The following methods are designed to be overridden by custom plugins.

Table 2.1: Partial list of shared plugins and their order.

Shared Plugins	Order
Gaze_Mapping_Plugin	0.1
Marker_Detector	0.2
Fixation_Detector	0.5
Display_Recent_Gaze	0.8
Pupil_Server	0.9

__init__(g_pool) The abstract instantiation operation of the Plugin class. It can be extended with keyed arguments (Python 2.7 Glossery, 2016) which are important if the plugin should implement a session persistent behavior (see get_init_dict() below).

init_gui() This method is only called once on load of the plugin. A plugin should initialize its user interface elements here. These elements might include e.g. menu entries in the sidebar or action buttons. The Pupil applications use *pyglui* (Pupil Labs, 2016l), a Cython powered OpenGL graphical user interface (GUI) framework. pyglui elements are able to draw themselves if necessary. The g_pool attribute (see above) allows access to certain areas of the user interface. Those include the following variables:

- gui: Yields the graphical context used by pyglui.
- image_tex: Gives access to the texture used to draw the camera image.
- main_window: Refers to the OpenGL window instance.
- quickbar: Includes fast accessible buttons on the left (e.g. start/stop recording/calibration).
- sidebar: The menu on the right holding submenus created by plugins.

Manual drawing should be done in the gl_display() method described below.

update(frame,events) This method is called once every frame. It provides the plugin with all relevant informations which are needed for the current frame.

The frame object provides image information like size (width, height), time of capture (timestamp) and the actual pixel data (img, three dimensional RGB array).

The events dictionary includes gaze and pupil positions as well as other data which might be added by other plugins with a lower order (see above). The dt value reports the amount of time passed since the last event loop iteration. If the front camera's frame rate is set to n frames per second then dt should be approximately $\frac{1}{n}$ seconds. This can be used as a simple performance measure, i.e. if the value is way higher than $\frac{1}{n}$ seconds then there are performance issues. gaze_positions and pupil_positions are keys for arrays containing Python dictionaries with the most recent events. See section 2.6.2 for more information on these.

`gl_display()` Like the `update()` method from above this method is called once every frame. This is the place to draw using the OpenGL context provided by *GLFW* (see *GLFW, a multi-platform library for OpenGL, window and input*, 2016). OpenGL provides the possibility to draw with high performance with the initial disadvantage of being very low-level.

`on_click(pos,button,action)` This callback is triggered if the user clicks somewhere in the window. It is called in context of GLFW polling OpenGL events. `pos` is a tuple containing the coordinates of the click in frame image pixels. `button` is an identifier to determine which mouse button was used (see *GLFW, mouse button documentation*, 2016). `action` is used to identify if the button was pressed or released.

`on_window_resize(window,w,h)` GLFW detects window resizes by the user during polling OpenGL events (similar to `on_click()`). If this happens, `on_window_resize()` is called with an reference to the resized window and the new width and height values.

`on_notify(notification)` The primary approach for communication between plugins is the usage of notifications. Plugins receive notifications through the `on_notify()` method. Notifications are simple Python dictionaries with an mandatory `subject` key. Its value is the name of the notification and it is used to identify notifications of interest. It might contain more custom fields added by the source plugin.

`get_init_dict()` This method can be overridden if the plugin should be session persistent. This means that the plugin will be loaded in the same state on startup as it was on termination during the last session. Such a plugin session starts on load of the plugin and ends with its termination.

In order for the session persistance to work this method has to return a Python dictionary with keys matching the `__init__()`'s keyed arguments (Python 2.7 Glossery, 2016). The returned dictionary will be stored as part of the user settings in the user directory of the running programm. On start of the next session it is passed to the `__init__()` method.

The `g_pool` attribute should not be included in the returned dictionary.

`cleanup()` This is called as part of the plugin termination process, i.e. this can be called voluntarily (through a specific user action) or forced (programm termination). If the plugin created GUI elements or custom GLFW windows (see `init_gui()` above) then they should be removed or destroyed when this method is called.

2.5.3 Non-Override Methods and Attributes

The following methods and attributes should not be overriden by custom plugins.

Listing 2.1: Example for a Python class with multiple inheritance.

```
1 class My_Subclass(Superclass_A, Superclass_B):
2     pass
```

notify_all(notification) Call this method with a notification dictionary to notify all other plugins. They will receive the notification through the on_notify() method described above. The passed dictionary has to include a field called subject which is used to identify the type of the notification (e.g. "calibration_successful"). Adding 'record':True will make the recorder save the notification during recording. Adding 'network_propagate':True will send the event to other pupil sync nodes (see section 2.6.4) in the same group. Recording and network propagation require the notification to be pickalable and able to be recreated though repr() and eval() (*repr function*, 2016; *eval function*, 2016). More fields can be added upon necessity. (Pupil Labs, 2016c, l. 114).

notify_all_delayed(notification,delay) This function delays the notification delivary by a certain amount of time. This time is given by delay whose default value is 3 seconds. See notify_all() for notification format specifications.

alive Boolean attribute which indicates whether this plugin should be destroyed during the cleanup step of the next event loop iteration. Setting it to false schedules it for destruction.

Utility Attributes The Plugin interface defines a set of attributes to access its class, base class and their respective class names. This set includes this_class, class_name, base_class, base_class_name and pretty_class_name. Note that base_
class returns the rightmost base class to be consistent in cases of multiple inherited classes. The example 2.1, base_class yields Superclass_B. This is relevant in case that the subclassing plugin uses a by_base_class uniqueness.

pretty_class_name simply replaces underscore characters with spaces. This is used if plugins are listed in the UI, e.g. in the calibration plugin selection dropdown.

2.5.4 Data access

In Capture the update() method is the primary approach how plugins access data. It is always supplied with the newest captured video frame. Therefore plugins are provided with fresh data on each call. This allows online data processing.

In Player the update() method is supplied with the currently displayed frame. A result of this is that plugins could be given the same frame over and over again. For that reason Player plugins usually do not use the update() method to access data. It is best practice to preprocess the whole recording on load of the plugin or on change of data. The update() method is used afterwards to access relevant preprocessed data for the current frame. If the preprocessing takes a high amount of time, it is common to delegate this task to a subprocess. This way the user interface and the other plugins do not get blocked.

As described in section 2.3.1 Capture detects pupil positions separated from the World process. Therefore the pupil positions need to be correlated with video recording of the front camera. Player does this by default. It provides correlated pupil and gaze positions, as well as fixation points, aligned by the video frame's timestamps. This data can be accessed using the `pupil_positions_by_frame`, `gaze_positions_by_frame` and `fixations_by_frame` attributes of the global `g_pool` variable.

2.6 Built-in Plugins

This section provides an overview over existing plugin groups and a selection of specific plugins which are important for the usage of *Pupil Inteface* in chapter 3. A plugin group is defined by an abstract Plugin subclass with a `by_base_class` uniqueness.

2.6.1 Pupil Detection

Pupil Capture has two methods of pupil detection to its disposal. The 2D "algorithm does not depend on the corneal reflection, and works with users who wear contact lenses and eyeglasses" (Kassner, Patera, & Bulling, 2014) and is explained in more detail in Kassner et al. (2014, subsection "Pupil Detection Algorithm", page 4). In summary, it uses Canny edge detection (Canny, 1986), composes the found edges to contours by connected components (Suzuki et al., 1985) and tries to fit ellipses (Fitzgibbon, Fisher, et al., 1996) onto them. The candidate ellipises are evaluated and filtered by their "confidence" (see Kassner et al. (2014)).

The second method is available since v0.7 of the Pupil software. It is based on Swirski and Dodgson work "A fully-automatic, temporal approach to single camera, glint-free 3D eye model fitting". This procedure models "the eyeball as a sphere and the pupil as a disk on that sphere." (Pupil Labs, 2016i). The model yields a three-dimensional position of the eyeball in the camera space, the rotation of the pupil around the eyeball's center and the pupil size (assuming an "average human eyeball diameter of 24mm" (Pupil Labs, 2016j)). It follows that this model takes movement of the eye camera into account. Furthermore the front camera is modeled as a pinhole camera with distortion (see Moeslund, Hilton, Krüger, & Sigal, 2011, chapter 2.3.1 and Kannala & Brandt, 2006). This does not only counteract the fish-eye distortion by the front camera's lens but also provides a three-dimensional space in which the gaze vector can be projected into. For this projection to work, it is necessary "to know the rotation translation of world and eye camera" (Pupil Labs, 2016j). Here the rotation translation references to the relative positions of the eye cameras to the front camera. The estimation of such a rigid transformation can be reduced to a bundle adjustment problem (Triggs, McLauchlan, Hartley, & Fitzgibbon, 1999). The combination of the three-dimensional eye model and the pinhole-camera model eliminates the problem of eye tracker slippage and means that it only needs to be calibrated once per subject (since the eye cameras might need adjustments).

2.6.2 Calibration and Gaze Mapping

An eye-tracker's task is to predict what the subject, who is wearing it, is looking at. It does so by identifying the subject's pupil positions and mapping them onto the front camera image (the

gaze). Since this mapping is not fixed, the eye-tracker has to find a function which describes the mapping best. There are different factors influencing it depending on the eye-tracker in use. A usual problem is the relative position between eye-tracker and eyes. Learning such relations is done by estimating parameters of a possible mapping function and is called calibration. It is done by asking the subject to focus a specific point (often a visual marker) in its field of view. This point should be known to the eye-tracking software (e.g. by detecting the marker in the front camera image). Once the calibration procedure yields the parameters one can use them to calculate the gaze from new pupil positions. This is called gaze mapping.

Calibration Plugins

On the Pupil platform the calibration procedure is split into two steps. The first step collects pupil positions and tries to find the respective positions in the front camera image (reference points). The `Calibration_Plugin` class has three subclasses which differ in the technique how the reference points are collected. Their exact differences are explained below.

Reference Points The term "calibration marker" refers to a specific visual pattern which the subject should focus, and can be detected automatically using software. A "stop marker" will end the calibration procedure on detection. Pupil provides different versions of stop and calibration markers. The two most recent versions are ν0.2 and ν0.3 (see figure A.2, appendix A, page X). The ν0.3 markers were recently added in the ν0.7.6 release of the Pupil software (Pupil Labs, 2016k). Their detection works by finding edges in the front camera image which are created by strong contrast and assemble a specific amount of concentric circles (Repository, 2016). ν0.2 distinguishes between stop and calibration marker by the number of concentric circles found. ν0.3 expects the same number of concentric circles but a different change of contrast. This allows ν0.3 to be smaller without losing detection performance. The release notes state that "a diameter of 40mm can be used at distances of 2m" (Pupil Labs, 2016k).

Manual Marker Calibration This plugin will simply try to find calibration markers, which do not move much over consecutive frames. When a marker is found, a "calibration marker found" notification is dispatched. If it moves too quickly after detection, a "calibration marker moved too quickly" notification will be sent and the plugin will look for a new marker. When enough samples were found, the plugin broadcasts a "calibration marker sample completed" notification. The procedure will stop as soon as a stop marker is detected. The important difference to the Screen Marker Calibration method below is that this plugin expects an external source to present the markers. This can be done by either printing or drawing the markers using custom software (see section 4.2).

Screen Marker Calibration The Screen Marker Calibration plugin works in the same way as the Manual Marker Calibration with the difference that it draws the calibration markers on its own. There is no need to e.g. print the calibration markers. It shows different amounts of markers, depending on which mapping mode (2D or 3D, see below for more information) is selected. It uses 5 points for the 3D option and 10 for the 2D one.

Natural Features Calibration Instead of using calibration markers the subject is asked to focus on visual features in its field of view. The person executing the experiment has to click on this feature in the front camera view of Pupil Capture. The software will try to track this natural feature and create reference points this way. Because this procedure needs a lot of coordination between the subject and the experimenter it is not recommended to use this calibration method. Additionally, it costs a lot of CPU load to track the natural feature through consecutive frames.

Calibration The second step does the actual estimation using the collected pupil positions and reference points. This is implemented in the `finish_calibration()` method which all of the above calibration plugins call after collecting the data. Pupil implements two ways of learning the relation between pupil positions and the reference points. As described in section 2.3.1 each eye is recorded separately. Therefore one does not always have a perfectly synced pair of pupil positions. For this reason the closest matches to reference points have to be calculated. This is done by looking for the pupil positions whose timestamps are closest to the one of each reference point.

The 2D option fits a two-dimensional polynomial of varying degree using the least-squares-error criterion. Afterwards, outliers are removed and the polynomial is fitted again without the outliers. Outliers are defined by having residuals higher than a given threshold. If the outlier filtering removes all data points, the calibration will fail with a "calibration_failed" notification. The data used for the fitting process are the normalized pupil coordinates and their matching reference locations. The calculated parameters are later used to initilize a gaze mapper of class `Simple_Gaze_Mapper` (or `Binocular_Gaze_Mapper` if binocular).

The 3D option calculates the rotation translation between the front and eye camera but keeps the polynomial mapping as fallback option. The necessary optimizations are run by the Ceres Solver (Agarwal, Mierle, & Others, n.d.), which solves the underlying bundle adjustment problem (Triggs et al., 1999). Once the positional relation between the cameras is known it can be used to initialize a `Vector_Gaze_Mapper` (or `Binocular_Vector_Gaze_Mapper` if binocular) instance.

Camera Intrinsics Estimation This calibration plugin does not perform an actual gaze calibration. It estimates the front camera's intrinsics by capturing a 11x8 asymmetrical circle grid (see figure A.3) from different angles. They consist mainly of the radial and tangential distortion coefficients of the lens. The actual calculation is done by the OpenCV (Bradski, n.d.) function `calibrateCamera()` which is based on work by Zhang (2000) and J.Y.Bouguet, n.d..

Class Reference `Calibration_Plugin` defines two simple abstract functions to start (`start()`) and stop (`stop()`) the calibration routine. Additionally it allows access to the currently chosen calibration plugin by setting the `active_calibration_plugin` attribute of `g_pool`.

Gaze Mapping Plugin

A gaze mapper's task is to project pupil positions into the image space of the front camera. There are different classes of gaze mappers depending on the pupil detection algorithm and calibration procedure used. There are two classes—`Simple_Gaze_Mapper` and `Binocular_Gaze_Mapper`—for two-dimensional gaze mapping which use the fitted polynomial (see above) to map the normalized pupil positions to normalized world view coordinates. The mapping by `Vector_Gaze_Mapper` and `Binocular_Vector_Gaze_Mapper` is a geometrical projection of the pupil's gaze vector into the front image. The `projectPoints()` of the OpenCV libary is used for this purpose which "[p]rojects 3D points to an image plane." (*OpenCV projectPoints() Documentation*, 2016). All gaze mapping plugins use the `update()` method to map the newest pupil positions. They extend the `events` object with the mapped gaze points (Key: `gaze_positions`). All gaze plugins have an inherent `order` value of 0.1 such that they are executed first and all the other plugins have access to the gaze points as well.

2.6.3 Surface Tracker

Pupil maps the subject's gaze onto the image captured by the front camera. In some situations it is important to know the gaze in relation to an object or surface in the image. A typical example are psychological experiments where subjects are presented stimuli on a computer screen and one wants to know the gaze in relation to the screen. A common approach by other eye trackers to this problem is to calibrate the gaze directly to the screen, i.e. using screen coordinates as reference points. Although, this presents other challenges, e.g. tracking of head and device movement. Pupil uses a different approach. It is able to define plane surfaces in the world image with the help of markers. It tracks those, calculates the surfaces' position in relation to the camera, and maps the gaze accordingly to the surface.

The implementation of the Surface Tracker plugin (previously known as Marker Detector) is "greatly inspired by the ArUco marker tracking library (Garrido-Jurado, Muñoz-Salinas, Madrid-Cuevas, & Marín-Jiménez, 2014)" (Pupil Labs, 2016n). It uses 5x5 square markers (see figure A.1, appendix A, page IX) which can be generated by the `make_square_markers.py` script (Pupil Labs, 2016a). Surfaces are registered and named using the plugin's user interface. They are also stored and loaded automaticly over the span of different sessions. A surface is defined by at least one marker. The same marker should not appear twice in a single frame. "Surfaces defined with more than 2 markers are detected even if some markers go outside the field of vision or are obscured." (Pupil Labs, 2016n).

The plugin uses homographic transformations (Berger, 2009, chapter 4) to map gaze positions to normalized locations within the surface. Homography uses geometric projection to map points from one plane to another. The necessary calculations are done by OpenCV functions for geometric image transformations (Bradski, n.d.).

During the `update()` (see section 2.5.2) call the plugin adds all detected surfaces to the `events` object. They are accessible under the key `surfaces`. Each surface is represented as a dictionary containing the following entries:

- `name`: The surface's name.

Listing 2.2: How to access the calculated gaze on a surface.

```
def update(self,frame,events):
  name = 'surface name'
  key  = 'realtime gaze on ' + name
  data = events[key]
```

- `uid`: Unique ID of the surface.
- `m_to_screen`: Projection matrix to transform points from surface space to image space.
- `m_from_screen`: Projection matrix to transform points from image space to surface space.
- `timestamp`: The `frame`'s timestamp.

Additionally, the plugin extends entries of `gaze_positions` (see section 2.6.2) with the projected gaze to each located surface. They can be accessed using a concatenation of "realtime gaze on" and the respective surface's name as key (see example 2.2).

2.6.4 Network

Pupil works as a stand-alone application. When running on a computer, it might use a lot of resources so that other timing-dependent software on the same system (e.g. a running experiment displaying stimuli) are disturbed. A simple solution to this situation is to run the interfering programms on different systems. This leads to the necessasity of synchronizing systems in case that the data of the programs has to be correlated.

Pupil provides a communication interface which uses the local network. This interface is seperated into two different plugins which serve two different purposes. One, Pupil Sync, synchronizes time and exchanges notifications. The other, Pupil Server, broadcasts events. Both interfaces are built on top of the ØMQ network libary (iMatix Corporation, 2016).

Pupil Sync

Distributing a program's work load to multiple systems requires a mean of communication between these systems. This communication should follow a well defined protocol. The term *Pupil Sync* refers to such a protocol which follows a *many-to-many* communication model. Systems implementing this protocol are refered to as Pupil Sync nodes or instances.

The protocol states that Pupil Sync instances should detect each other in the local network using the ØMQ Realtime Exchange Protocol (iMatix Corporation, 2009). This means that Pupil Sync nodes are able to communicate with each other without configuration effort by the user. The protocol also defines how the instances' clocks should be synchronized and how the notifications should be exchanged.

Time synchronization For clock synchronization each instance submits a worthiness for their clock (from 0 (unworthy) to 1 (destined)). The instance with the highest worthiness is chosen as clock master. All other instances (the clock followers) change their clocks according to the clock master. Every instance has a universally unique identifier. These are generated randomly. In case of a tie in worthiness the node with the higher UUID wins the tie-breaker.

The time syncronization follows a simple scheme once the clock master is set. A clock follower sends a time request to the clock master at time t0. Upon reception of the request the master replies with its own time t1. The clock follower records the time t2 simultaneous to receiving the clock master's time. Using these three measurments one can calculate the $latency (= t2 - t1)$ of the connection and the target time ($t1 + \frac{latency}{2}$). "[Therefore] [a]ccuracy is bounded by network latency and clock jitter." (Pupil Labs, 2016b, l. 22) Clock jitter refers to the clock's deviation from its true time. This means that a steady latency as well as a low clock jitter will increase the accuracy of the time follower's clock.

Technical note According to the documentation of the example Pupil Sync node implementation by Pupil Labs (Pupil Labs, 2016h, ll. 40-45) the system clock jitter depends on the operating system in use. In this case they measured the jitter of the standard Python time function `time.time()`. They claim OS X has a low jitter (less than 1 millisecond). In comparison Linux has a higher jitter (less than 3 milliseconds) so that it is recommended to use the `get_time_monotonic()` function of their *pyuvc* framework (Pupil Labs, 2016m, file "uvc.pyx", l. 706). Unfortunately, an implementation of this function for Windows is not provided.

Pupil Server

The second network interface provided by the Pupil applications is Pupil Server. It uses a *one-to-many* model with unidirectional communication which is based on the ØMQ Publisher-Subscriber scheme (iMatix Corporation, 2014). This means that a dedicated instance (publisher) is able to send data to the various other instances (subscribers) in parallel. This scheme has the disadvantage that the publisher does not know if subscribers are successfully connected or if they are able to keep up with the speed of arriving data. On the other side, the unidirectional data flow allows high performance despite of a high volume of traffic.

The Pupil Server plugin behaves as publisher for application data, e.g. gaze positions in Capture. Other applications can subscribe to the data stream und process it. One should be aware that this communication model does not provide 100% reliable and correct transfer of the sent data. Therefore it is recommended for situations in which the loss of single transmissions is not important (e.g. live applications which discard past events).

Pupil Remote

The Pupil Remote plugin provided a simple network interface to remote control the Pupil applications. It was the predecessor of Pupil Sync and was superseded in the *v*0.6 release (Pupil Labs, 2015d) of the Pupil software.

Pupil Sync can be used as remote control as well. The user needs to send specific notifications using Pupil Sync to simulate this behaviour. Some notifications are currently not documented

well and require further investigations by the user. At the start of the thesis project there was no interface available which allowed simple control over the Pupil applications without detailed knowledge of just these. This led to the creation of Pupil Interface (see chapter 3), a remote control module built on top of the Pupil Sync protocol. Other community members asked for a simple remote control feature as well (Pupil Labs, 2015a). Since Pupil Interface is not fully documented publicly yet, Pupil Labs decided to bring back the Pupil Remote plugin (Pupil Labs, 2015b) with release v0.7.6 (Pupil Labs, 2016k).

Technical note Pupil Remote uses the ØMQ Request-Reply (REQ-REP) Pattern (iMatix Corporation, 2013) to send simple string messages between the remote and the controlling system.

In the REQ-REP pattern the requesting system sends a message to the target system. The receiver will respond with a single reply message. Until reception of the reply the requesting system will silently discard all incoming messages by any instances other than the target system. The usage of the REQ-REP pattern is appropriate since Pupil Remote's protocol is kept rather simple. It reduces the complexity of simple clients, too. This protocol is not designed to receive continuous updates (e.g. for time synchronization or the progress of the calibration procedure). If one is dependent on these one has to use Pupil Sync.

1. Get the current `frame` object from the front camera. It includes image information as well as the timestamp of the capture.
2. Get recent pupil positions from the global queue and add them to the `event` object which is passed to the plugins later on.
3. Move due notifications from the delayed notifications pool to the pool of current notifications.
4. Work off the pool of current notifications by calling the `on_notify()` method of each plugin. It is best practice to use the notification system for inter-plugin communication (see section 2.5.2).
5. Call the `update()` method of each loaded plugin and pass the `frame` and `event` objects. Plugins are able to add their own data to `event`. This additional information can be used by other plugins if they have a higher `order`.
6. Unload plugins if necessary.
7. Draw the `frame` image into OpenGL buffer.
8. Call `gl_display()` method of each plugin. This enables plugins to draw on top of the `frame` image.
9. Draw GUI elements like menus and options.
10. Actually display the drawn image by swapping the OpenGL buffer.
11. Poll for OpenGL events. This calls earlier registered callbacks if necessary.

Figure 2.3: Overview over the event loop of the World process. The event loop is conditioned on the `WindowShouldClose` event. This means, that the process will terminate on closing the window.

Chapter 3

Pupil Interface

3.1 Overview

The previous chapter described the Pupil platform and its interaction possibilities. The flexible plugin structure allows the extension of application functionality and direct interaction with other plugins (see section 2.5). Pupil also provides interfaces to exchange messages between plugins (through Pupil Sync, section 2.6.4) and access data (through Pupil Server, section 2.6.4) over the network. It is also remote controllable through Pupil Remote (see section 2.6.4). Although Pupil Remote does not require knowledge about implementation details, it is restricted to a simple communication pattern by design. This pattern does not provide continuous time synchronization and application updates (e.g. calibration progress).

This chapter presents a novel interaction method to remote control Pupil applications. It is called *Pupil Interface* and uses Pupil's existing built-in communication interfaces. It provides full Pupil Sync functionality although it does not assume knowledge about implementation details of the Pupil applications. Additionally it allows to start and stop recording sessions, and receives continues updates of ongoing calibration procedures.

Pupil Interface is implemented as Python module. It acts as a Pupil Sync node (section 2.6.4) and is able to subscribe to Pupil Servers (section 2.6.4). This allows the user to control the calibration and recording functionality of Pupil Capture through a set of simple Python functions. These functions can be used in a synchronous manner (by blocking the running script until it receives an answer from Pupil Cpature) as well as in a aynscronous manner (by defining callback functions which are called in case of a response). Under the hood it reuses code provided by the Pupil Helpers repository (Pupil Labs, 2016h).

PsychoPy It is best practice to use already present tools to implement psychological experiments for the use on computer screens. The tool of choice for experiments programmend in Python is *PsychoPy*. PsychoPy is an extensive framework providing visuals for presenting stimuli, storing data and other utility functions. Pupil Interface is designed in such a way that it is uncomplicated to integrate it into existing and future PsychoPy experiments.

Section 3.3 gives details on the network implementation which follows from the design concepts presented in section 3.2. Section 3.4 describes Pupil Interface's event management and how to utilize it for own applications. Sections 3.5 and 3.6 provide a full reference over the modules functionality.

3.2 Design Concepts

The goal was to implement the remote control functionality in such a way that one would only have to import the module without having to know anything about the actual communication details between the module and the Pupil applications. Another important assumption was also that the script running Pupil Interface might maintain its own event loop and user interface. Therefore the script has to run concurrently to the Pupil Interface. There should also be the possibility for the script to calibrate the eye tracker without having to exit a potential fullscreen view.

In total the module requires as little as possible foreknowledge about the Pupil applications and allows as much interaction as possible (in the sense of communication between the module and the Pupil applications) without interference (in the sense of reciprocal performance restrains).

Pupil Interface Pupil Interface actually implements a Pupil Sync node but is also able to connect to a Pupil Server. Therefore it synchronizes its clock with other Pupil Sync nodes, is able to exchange notifications, and subscribes to broadcasted events, e.g. gaze positions. Since Pupil Sync works over the local network, it is possible to run the Pupil software on one system and the custom script running Pupil Interface on a different one. This minimizes potential interference (in the sense of performance issues) between both processes while still allowing interaction. The simple functionality of a remote control is already implemented in the Pupil helper example code (Pupil Labs, 2016h) by sending the necessary notifications over Pupil Sync. But the implementation only allows one-way communication. Pupil Interface is able to provide feedback about the progress of specific procedures (e.g. manual calibration) or the network state (see the section 3.5 below about available events). The user is able to wait for specific events or to be notified about recent events using callbacks (see section 3.4 below).

3.3 Network Backend

Section 2.6.4 describes how Pupil defines its communication interfaces. It uses the network libary ØMQ to implement these interfaces. One of these is Pupil Sync. It uses a *many-to-many* scheme where nodes are able to detect each other in the local network. Pupil makes use of the Zyre framework (Zyre, 2016) which provides the node detection innately. Pupil Interface reuses the mentioned frameworks for code compatibility and reusability.

Network connections are represented by a pair of sockets, each one representing and managing one end of the connection (see Hall, 2016, chapter 2). They need to be created and setup correctly by the user. Futhermore, they need to be checked for new content regularly to make

sure that all data sent by the other side is able to be processed. Pupil Interface moved the network functionality to two separate threads to avoid this complexity for the user. Threads are small code sequences which can be executed in parallel (see Ullenboom, 2004, chapter 12). This prevents the network activity of affecting the responsiveness of the user interface.

The example implementation of a Pupil Sync node (Pupil Labs, 2016h, will be refered to as `Pupil_Sync_Node`) already splits the network communication handling into a separate thread. Pupil Interface does the same thing additionally for the subscription to the Pupil Server. Its background event loop follows a similar structure to the one in the example. See section 2.3.2 for a recapitulation of event loops. Both make use of a ØMQ class called `Poller`. It provides a simple interaction scheme if one needs to work with multiple sockets. First all sockets which shall be observed are registered. Calling `poll()` will block until at least one of the registered sockets generates new activity (e.g. new received data). The function will return the respective socket and if the activity was incoming or outgoing. In this case the incoming traffic has to be processed. `Pupil_Sync_Node` observes the following three sockets:

- The Zyre network socket to monitor messages by other Pupil Sync nodes.
- The communication pipe which is used to receive commands by the main thread (e.g. termination of the thread).
- A custom pipe to initiate a time announcement (in case the node is the clock master) or to check if the current clock master timed out (in case of a clock follower).

There are also three sockets being observed in the thread responsible on subscribing to the Pupil Server:

- The subscription network socket which receives broadcast messages by the Pupil Server.
- A pipe to receive commands by the main thread.
- The network monitor socket provided ØMQ. It receives connection events including e.g. ceonnecting and disconnecting to the Pupil Server.

In summary `Pupil_Sync_Node` and Pupil Interface's extension use background threads and ØMQ's capability to monitor mulitple sockets to process information from multiple sources without impeding the main thread.

3.4 Event Management and Callbacks

As described above the goal is to create a communication interface with reduced complexity for the user. This is achieved by hiding complexity in the background but requires an uncomplicated method for interaction between user and background. Section 3.6 describes how the user can send commands to the background. The user has to handle a set of events emitted by the background to receive feedback (see section 3.5). The user has two possibilities to handle events. Either one can be notified about new events (using callbacks, see example 3.2) or one can wait explicitly for one or more specific events to happen (see example 3.1).

Figure 3.1: Calibration wait example

```
# import modules
from pupil_interface import Communicator
from pupil_interface.const import CAL_SUC, CAL_FAIL
# CAL_SUC:  Calibration was successfull
# CAL_FAIL: Calibration failed

# initialize Pupil Interface node and start calibration
node = Communicator()
node.startCalibration()

# Wait for the calibration to finish
data = node.waitAnyEvent([CAL_SUC,CAL_FAIL])
print data
node.close()
```

The events are created in the background threads of the network backend since they process the incoming network traffic (see 3.3). This means that it is necessary to make them available to the main thread. An easy solution would be to invoke the callbacks directly from the background thread. This would lead to the situation that the functions would be called in a different context than the main thread. This can cause problems, if the callback accesses shared memory or includes functions which are supposed to be executed in the main thread. Examples for this type of functions are those that interact with the user interface or the graphical context of the application.

Since Pupil Interface does not assume awareness of this problem, it is solved differently by Pupil Interface. It uses a thread-safe `Queue` object as transportion method. The background enqueues all created events and the main thread polls the queue for new events. Afterwards the main thread only has to identify which type of callback to execute. Python modules are not able to force the main thread to execute a function unless they are also executed from the main thread. This means that Pupil Interface is not able to check for new events on its own. The reason for this is that there is no event loop provided by Python itself. To avoid this problem one should call the `checkEvents()` method (see below) at least once each iteration of their event loop. This makes sure that the event queue is emptied regularly and that all related callbacks are called. This method should be used if you want to send commands to the Pupil software but its answer is not necessary for the process to continue (e.g. waiting for gaze events).

The second possibility for the user to handle events is to wait for them explicitly. Calling `waitAnyEvent()` or `waitAllEvents()` (see API reference below) will call `checkEvents()` until it encounters one or all of the events the user is looking for. A possible solution would be to call the function in a busy loop, i.e. a loop that continuesly loops and utilizes the whole CPU. This is considered bad practice. Instead Pupil Interface uses a `threading.Event` object to get notified about new events. `Event` objects are thead-safe and can either be set or not. Calling `wait()` on it in one thread will suspend the thread until it is set somewhere else. This means that Pupil Interface will suspend the main thread after calling the `checkEvents()` method and not finding the events it was looking for. The moment the network backend encounters a new event it will

Figure 3.2: Calibration callback example

```
# import modules
from time import sleep
from pupil_interface import Communicator

# define callback
def callback_function(event, data):
    print event, data

# initialize Pupil Interface node and start calibration
node = Communicator()
node.startCalibration(callback=callback_function)

# start event loop
while True:
    # run custom code...
    # check for events, will call 'callback_function'
    node.checkEvents()
node.close()
```

queue it, set the `Event` object and therefore unsuspend the main thread. It will continue to check the newly enqueued events and wait again if necessary. In total this method should be used if the event one is waiting for is crucial for the program to continue (e.g. a script should not continue until the recording is started).

3.5 Available Events

Every constant has a respective short version.

- `EVENT_TIMEOUT` (`TIME_OUT`) Event that is local to the particular system. It occures on calling `waitAnyEvent()` or `waitAllEvents()` and them timing out.

Calibration Events These are possible events during a running calibration procedure.

- `EVENT_CALIBRATION_MARKER_MOVED_TOO_QUICKLY` (`CAL_MMTQ`)
  A marker was found but either the subject moved her or his head too fast or the marker itself moved too quickly. Pupil will look for a new steady marker.

- `EVENT_CALIBRATION_STEADY_MARKER_FOUND` (`CAL_SMF`)
  Pupil found a steady marker. It will try to calculate its position averaged over a number of frames.

- `EVENT_CALIBRATION_SAMPLE_COMPLETED` (`CAL_SC`)
  Pupil found enough frames to calculate the position of the marker and will look for a new steady marker.

- `EVENT_CALIBRATION_SUCCESSFULL` (`CAL_SUC`)
  Pupil finished its calibration procedure successfully.
- `EVENT_CALIBRATION_FAILED` (`CAL_FAIL`)
  Current calibration procedure stopped but was not successfull.

Recording Events

- `EVENT_RECORDING_STARTED` (`REC_STA`) Pupil started a new recording session.
- `EVENT_RECORDING_STOPPED` (`REC_STO`) Pupil stopped the current recording session.

Network Events The following constants are related to Pupil Sync as well as to the Pupil event broadcast server.

- `EVENT_NET_NODE_JOINED_GROUP` (`NET_JOIN`) Pupil Interface joined a Pupil Sync group and is able to receive and send messages to other nodes in this group.
- `EVENT_NET_NODE_EXITED_GROUP` (`NET_EXIT`) Pupil Interface left the current Pupil Sync group.
- `EVENT_NET_CONNECTED` (`NET_CONN`) Pupil Interface connected to a Pupil Server and will receive its broadcasted messages.
- `EVENT_NET_DISCONNECTED` (`NET_DISC`) Pupil Interface disconnected from the Pupil server.
- `EVENT_RECEIVED_GAZE_POSITIONS` (`RCV_GAZE`) Indicates newly received gaze positions from the Pupil server.

3.6 Full API Reference

3.6.1 `Pupil_Sync_Node`

This class is part of the Pupil Helpers collection (Pupil Labs, 2016f). It talks to other Pupil Sync nodes and synchronizes the clocks between them. This is done by the usage of a background thread which takes care of the network connection and the clock synchronization process.

`__init__([name, group, time_grandmaster])` As described in section 2.6.4, Pupil Sync nodes connect to each other by joining a group with an specific name (`group`, by default "default group"). They are internally identified by an unique id but have also an visible name. It can be set using the `name` parameter. Its default value is "unnamed Pupil". `time_grandmaster` is a boolean which if set `True` will lead to the node trying to become the clock master. If successfull this node's clock will be taken as reference for time synchronization between nodes.

`on_notify()` Gets called when a notification is received. This should be overwritten by subclasses for message handling. Prints the passed notification by default.

notify_all(notification) Forwards notification to all connected Pupil Sync nodes if its network_propagate value is set to True.

get_time() Returns the synchronized timestamp.

get_unadjusted_time() Returns unmodified timestamp.

clock_master_worthiness() Returns a value between 0 (unworthy) and 1 (destined) as a measure for the node's clock master worthiness. The clock with the highest clock master worthiness will be the reference for time synchronization. If there is a tie between nodes the choice will be made on the node whose unique id has a higher integer value.

By default this method will return 1 if time_grandmaster was set to True during the initialization (see above) else it will return 0.

sync_status_info() Returns a string with the current status. In the current implementation there are only three possible stati:

1. Waiting for sync.
2. Node is clock sync master.
3. Node is clock sync follower.

set_name(new_name) Sets the name of the node to new_name. This will restart the internal synchronization process (the background thread).

set_group(new_name) Changes the node's group to new_name. This will restart the internal synchronization process (the background thread).

3.6.2 Communicator

This class inherits from Pupil_Sync_Node and extends it by implementing the actual methods used for controlling the Pupil software.

__init__([sub_addr,sub_port]) Since this is a subclass all optional arguments from aboves __init__() method can be passed, too. The default value for name is overwritten to "Pupil Interface Node". Additional parameters are sub_addr and sub_port which are used to subscribe to a Pupil Server accessible under given address. The default address is tcp://127.0.0.1:5000.

startRecording([session_name, callback]) Starts a new recoding session by sending "should_start_recording" notification. session_name is a string which is used to name the folder containing the recording. Its default value is "Unnamed session". The callback triggers on receiving the REC_STA event. See example 3.3.

`stopRecording([callback])` Stops an ongoing recoding session by sending "should_stop_recording" notification. The callback triggers on receiving the `EVENT_RECORDING_STOPPED` event.

`startCalibration([callback])` Starts the calibration procedure of the currently selected calibration plugin by sending "should_start_calibration" notification. The callback triggers on receiving one of the calibration events.

`stopCalibration([callback])` Stops an ongoing calibration procedure by sending "should_stop_calibration" notification. The callback triggers on receiving one of the calibration events.

`checkEvents()` Events produced by other threads are buffered in a queue. Calling `checkEvents()` checks this queue, calls appropriate callbacks and empties the queue afterwards. Ideally this is called from the main thread and at least once in each iteration of the event queue. This ensures that the callbacks are executed in the main thread as well. Use `waitAnyEvent()` or `waitAllEvents()` (see below) instead of a loop calling `checkEvents()` if you want to pause your program until a specific event happens.

Returns a Python list of tuples. The first component is the respective constant of the encountered event. The second component is a context-specific object providing information about the event.

`waitAnyEvent(events[,timeout])` Blocks the current thread until one or more events in `events` happen. `events` can be either an event constant or a Python list of event constants (see section 3.5 for possible values). `timeout` specifies, if present and not `None`, a timeout in seconds after which the waiting is aborted and the current thread continued.

Returns a Python dictionary containing the encountered events. The keys are the event constants and are paired with their respective event objects (see `checkEvents()`). Only the most recent context object is paired with its respective constant since it can only appear once as a key. If the operation times out the returned dictionary will include the `TIME_OUT` constant as key.

`waitAllEvents(events[,timeout])` Works in the same way as `waitAnyEvent()` with the difference that it blocks the current thread until all events in `events` were encountered at least once. Again, waiting longer than `timeout` seconds will interrupt the procedure.

Figure 3.3: How to start and stop a recording session

```
from pupil_interface import Communicator
from pupil_interface.const import REC_STA
node = Communicator()

# start recording
node.startRecording(session_name="Example Session")

# Wait until the recording is running
node.waitAllEvents(REC_STA)

# ... execute other code ...

node.stopRecording()
node.close()
```

Chapter 4

PsychoPy Pupil Utilities

4.1 Overview

When conducting computer-aided psychology experiments, one does normally not rely on programming everything oneself. An extensive framework for creating those kind of experiments is PsychoPy (Peirce, 2007). It is written in Python and provides, among otherst classes for creating and displaying visual stimuli. Often these experiments are run fullscreen to prevent distraction of the subject. This is one of the reasons why Pupil provides the possibility to run their calibration procedure in fullscreen. A typical sequence would be to run a setup view in PsychoPy, switch to Pupil, run the calibration, switch back and continue with the experiment. Depending on the operating system in use and how the fullscreen mode is implemented, such a switch from one application to the other could create undesirable visual effects. Additionally, one has no control over the Pupil procedure from the PsychoPy script. For this reason, PsychoPy Pupil Utilities provide classes to visualize the necessary markers. One would draw the respective markers using PsychoPy avoiding the switch of applications and retaining control over the image presented to the user.

Another use case of the PsychoPy Pupil Utilities is the ability to draw surface markers with PsychoPy. Drawing them on the screen edges removes the necessity of affix them physically on the screen.

4.2 Calibration Marker

As described in section 2.6.2 Pupil Capture uses sepcific patterns as calibration and stop markers. The `psychopy_pupil_utils` module includes two `psychopy.visual` classes to draw the corresponding markers in a PsychoPy context. Unfortunately only version *v*0.2 of the calibration markers is currently implemented in the classes below. This presents no problem since the target release *v*0.7.6 of the Pupil platform is backwards compatible to the old markers.

Version *v*0.2 The calibration and stop markers differ in the number of concentric circles drawn. Both marker classes inherit from the `visual.Circle` class provided by PsychoPy. They consist of a number of subcircles that are positioned in a way that they produce the desired pattern

Figure 4.1: Calibration marker initilization example

```
# import modules
from psychopy import visual
import psychopy_pupil_utils as utils

# create window with size 'width' x 'height'
win = visual.Window([width,height], units='pix')

# create markers
calibration_marker = utils.PupilCalibrationMarker(win,radius=75)
stop_marker        = utils.PupilStopMarker(win,radius=75)
```

Figure 4.2: Marker drawing example

```
import psychopy_pupil_utils as utils
# initialize window and markers

# calibration positions
cal_pos = utils.ninePointCalibrationPositions()
offset = calibration_marker.radius
for pos in cal_pos:
    calibration_marker.drawAtCalibrationPosition(pos, offset)
    win.flip() # actually draws the marker in the window
    # wait for e.g. user interaction

# stop calibration
stop_marker.draw()
win.flip()
```

of concentric circles. The calibration marker is actually a stop marker with an additional inner circle. Its implementation follows this idea in the same way that it inherits the subcircles of the stop marker and adds another `visual.Circle` instance on its own. It follows that these classes' task is to just manage their subcircles. This includes change propagation of following attributes: `pos`, `size`, `ori`, `opacity`, `interpolate`, and `autoDraw`. Calling `draw()` will propagate the call to all subcircles.

Initilization The calibration and stop markers `__init__()` functions take both a `visual.Window` instance as required argument. Additionally, all keyword arguments from the `Circle` class are allowed. The keywords `lineWidth`, `fillColor` and `edges` are an exception. These are overwritten to ensure the correct depiction of the markers.

Utility functions Additionally, they implement a set of utility functions which simplify the positioning of the markers. It is important to cover as much as possible of the subject's field of view during calibration. That means that the positions of the calibration markers should not be close to another. When the calibration procedure is done using a screen, one positions the markers typically at the corners and borders of the screen to maximize the field of view used.

The two following methods return a list of 9 tuples representing corner and edge points one would typically use. Their two components represent relative `x` and `y` coordinates and have a range of $[0, 1]$.

- `ninePointCalibrationPositions()` returns 9 fixed calibration positions.
- `randomizedNinePointCalibrationPositions()` returns the 9 calibration positions mentioned above in a shuffled list.

The other positioning function is `drawAtCalibrationPosition()`. It takes a mandatory argument and an optional one called `offset`. The mandatory argument is expected to be a tuple in the same format as the tuples returned by the two functions above. `offset` is a scalar value which is `None` by default. Its value is the amount of pixels, that the marker will be moved in direction of the screen center, in case it would be clipped by the screen. Positioning the markers will set their center instead of the boundaries' corners. This will lead to clipping if one uses one of the provided calibration positions (with exception of `(0.5,0.5)`) since the markers' centers will be positioned on the screen edges. Therefore it is recommended to use at least the marker's radius as offset such that the markers boundary will not be clipped. In case one also uses the surface markers (see below) it is recommended to use an offet of $radius + max(width, height)$ where $radius$ is the calibration marker's radius and $width$ and $height$ the size of the surface marker. Listings 4.1 and 4.2 provide examples on how to initialize and draw calibration markers.

4.3 Surface Marker

As described in section 2.6.3 the Pupil platform is able to track plane surfaces in the capture. It uses specific markers to identify them. The `SurfaceMarkers` class encapsulates the functionality of loading a range of pre-rendered surface markers and positioning them on the screen. The single markers are pre-rendered as Portable Network Graphics using the *make_square_markers.py* script (Pupil Labs, 2016a) which is part of the Pupil Helper repository. Those are displayed using multiple `visual.ImageStim` objects which are managed by the `SurfaceMarkers` class. To ensure the visibility of the markers a white border is added around each marker. The size of the border follows the recommendation of being 1.2 times as big as the grid size. The grid size is determined by dividing the size of the marker by 5 (since it is a 5x5 grid).

Initilization Again, as a `visual` class, `SurfaceMarkers` requires a `Window` instance during initialization. Additionally to the inherited keywords from `BaseVisualStim` and `ContainerMixin`, there are following optional arguments:

- `markerIDs`: An array with 8 indexes in the range of $0 - 63$. Decides which marker images to use.
- `screen_pos`: An array of 8 tuples defining the location of each marker. The locations are relative and have values in range of $[0, 1]$ (see section 4.2).
- `size`: scalar (applies to both dimensions) indicating the marker's size in pixels (without the border).

`SurfaceMarkers` allows access to its markers' sizes through a property named `size`. It returns (on getting) and takes (on setting) a scalar value which represents the markers size without its border. To get a marker's size including its borders one may access the read-only `total_size` attribute. Calling `draw()` will propagate the call to all markers. Listing 4.3 provides an example on how to use surface markers.

Figure 4.3: Surface marker initialization and drawing example

```
1 # import modules
2 from psychopy import visual
3 import psychopy_pupil_utils as utils
4
5 win = visual.Window([width,height], units='pix')
6 markers = SurfaceMarkers(win,size=100,markerIDs=range(42,50))
7
8 markers.draw()
9 win.flip()
```

Chapter 5

Conclusion

With Pupil, Pupil Labs has managed to create an eye-tracking platform which is highly customizable in its software. The headset's design (see chapter 2.2) allows the user to move freely. The front camera captures the subject's field of view, while the eye cameras record his or her eyes. The open-source software provided by Pupil Labs uses a 3D model to recognize the subject's pupil positions and projects the eyes' gaze into the front camera's image (see chapters 2.6.1 and 2.6.2). With the help of special markers, one can also define surfaces which the Pupil software will map the subject's gaze onto (see chapter 2.6.3). By placing these markers around a screen, this feature can be used for screen-based experiments. One should keep in mind that the markers can act as an unintended source of distraction.

To streamline the usage of the Pupil eye tracker in such experiments, I developed a Python remote control module called Pupil Interface (see chapter 3) and a set of utility functions (see chapter 4). Pupil Interface provides a bidirectional communication method to e.g. start and stop a Pupil Capture recording session. The utility functions allow the user to display Pupil surface and calibration markers in their applications.

Surface tracking stability I conducted a small test to evaluate the stability of the surface tracking. For this purpose, a test program would create a test surface which should be tracked by the Pupil software. The Pupil Capture recording of this test is available on the CD attached to this thesis. Unfortunately, the surface tracking stability was not high during the test. This can be retraced by opening the mentioned recording in Pupil Player and using the *Offline Surface Tracking* plugin. Possible reasons for the instability are the distortion of the front camera and the detection of false-positive markers. The distortion only presents a problem during the recording. The Offline Surface Tracking plugin uses the camera intrinsics (see chapter 2.6.2) to compensate the distortion. The amount of false-positives can be lowered by using bigger surface markers and an higher *min marker perimeter* value. This is a parameter for the minimal size of a surface marker which is tracked by the Pupil applications.

Future work Pupil Labs is steadily working on the development of its software, e.g. the introduction of 3D gaze mapping. This often includes changes and improvements to the communication interfaces used by the Pupil applications. This means that Pupil Interface

will need consequent maintenance to insure compatibility to the newest Pupil versions. An upcoming feature is e.g. the unified communication backend (Pupil Labs, 2015c). This change results in the restructuring of Pupil's event and communication handling. It allows, amongst others, direct data access for external applications. A possible overhaul of Pupil Interface, such that it uses the unified backend, might reduce its complexity and increase its performance.

Appendix A

Visual Markers

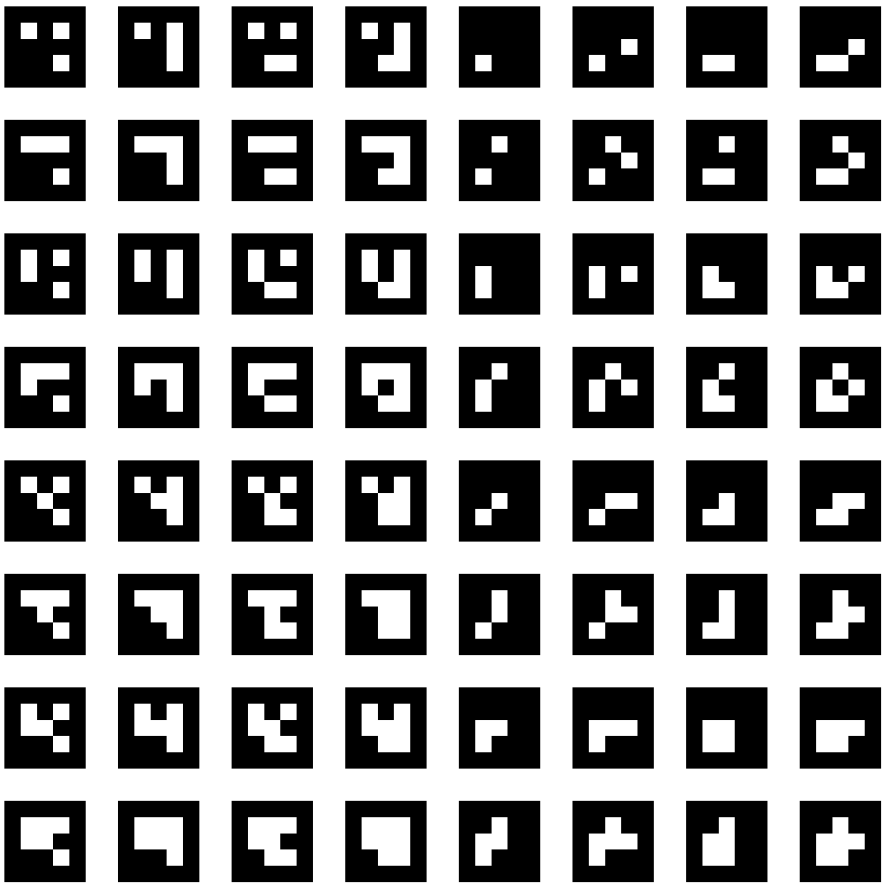
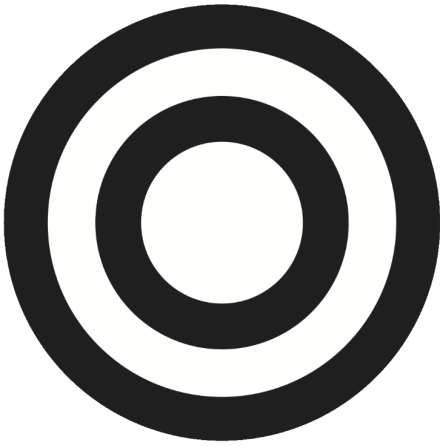


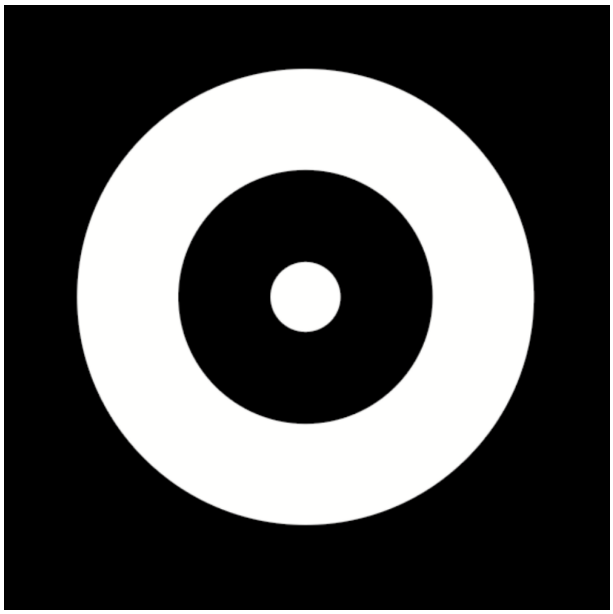
Figure A.1: Surface markers

(a) Calibration Marker v0.2



(b) Stop Marker v0.2

(c) Calibration Marker v0.3



(d) Stop Marker v0.3

Figure A.2: Calibration and Stop Marker

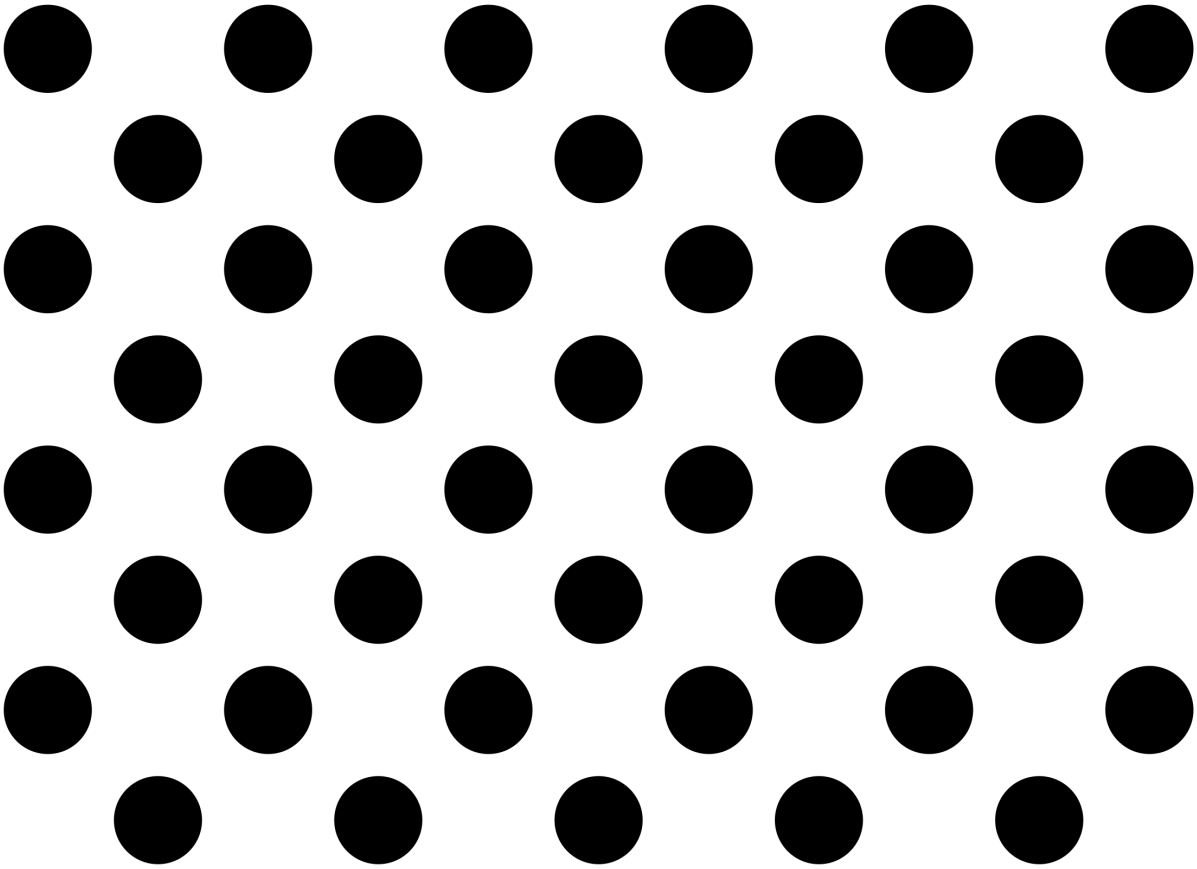


Figure A.3: 11x8 asymmetrical circle grid used to estimate the camera intrinsics.

Appendix B

Resources

The following resources are provided as hard copy on a CD attached to the printed version of this thesis:

- Digital copy of this thesis
- Pupil source code repository, https://github.com/pupil-labs/pupil/tree/v0.7.6
- Pupil documentation, https://github.com/pupil-labs/pupil/wiki
- Pupil Interface and PsychoPy Pupil Utilities source code repositories, https://github.com/papr/Pupil-Research-Plugin-Suite
- List of software dependencies for Pupil Interface
- Recording of the surface stability test

List of Figures

References

Agarwal, A., Mierle, K., & Others. (n.d.). *Ceres solver.* Retrieved 2016-05-16, from http://ceres-solver.org/

Berger, M. (2009). *Geometry i*. Springer Science & Business Media.

Bradski, G. (n.d.).
Dr. Dobb's Journal of Software Tools.

Canny, J. (1986). A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*(6), 679–698.

eval function. (2016). Retrieved 2016-05-30, from https://docs.python.org/2/library/functions.html#eval

Fitzgibbon, A. W., Fisher, R. B., et al. (1996). A buyer's guide to conic fitting. *DAI Research paper*.

Garrido-Jurado, S., Muñoz-Salinas, R., Madrid-Cuevas, F., & Marín-Jiménez, M. (2014). Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, *47*(6), 2280 - 2292. Retrieved from http://www.sciencedirect.com/science/article/pii/S0031320314000235 doi: http://dx.doi.org/10.1016/j.patcog.2014.01.005

GLFW, a multi-platform library for opengl, window and input [Repository]. (2016). Retrieved 2016-05-30, from https://github.com/glfw/glfw

GLFW, mouse button documentation. (2016). Retrieved 2016-05-30, from http://www.glfw.org/docs/latest/group__buttons.html

Hall, B. (2016). *Beej's guide to network programming* [Guide]. Retrieved 2016-05-31, from http://beej.us/guide/bgnet/output/print/bgnet_A4.pdf

Höning, N., König, P., Stelzer, F., Steger, J., Wilming, N., Betz, T., … Schmitz, M. (2008). Goodgaze: Eine technologie aus der hirnforschung analysiert webseiten auf ihre aufmerksamkeitswirkung. *Tagungsband UP08*.

Höffner, S. (2016). *Thesis template uos* [Repository]. Retrieved 2016-05-16, from https://github.com/Faedrivin/ThesisTemplateUOS

iMatix Corporation. (2009). *Ømq realtime exchange protocol* (RFC No. 36). Retrieved 2016-05-16, from http://rfc.zeromq.org/spec:36/ZRE

iMatix Corporation. (2013). *Ømq request-reply pattern* (RFC No. 28). Retrieved 2016-05-16, from http://rfc.zeromq.org/spec:28

iMatix Corporation. (2014). *Ømq publish-subscribe pattern* (RFC No. 29). Retrieved 2016-05-16, from http://rfc.zeromq.org/spec:29/

iMatix Corporation. (2016). *Ømq network libary.* Retrieved 2016-05-30, from http://zeromq.org/

J.Y.Bouguet. (n.d.). *Camera calibration toolbox for matlab.* Retrieved 2016-05-16, from http://www.vision.caltech.edu/bouguetj/calib_doc/

Kannala, J., & Brandt, S. S. (2006). A generic camera model and calibration method for conventional, wide-angle, and fish-eye lenses. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, *28*(8), 1335–1340.

Kassner, M., Patera, W., & Bulling, A. (2014, April). Pupil: An open source platform for pervasive eye tracking and mobile gaze-based interaction. Retrieved from http://arxiv.org/abs/1405.0006

Kassner, M., Patera, W., & Bulling, A. (2016). *Pupil Labs.* Retrieved 2016-05-16, from https://pupil-labs.com/

Moeslund, T. B., Hilton, A., Krüger, V., & Sigal, L. (2011). *Visual analysis of humans*. Springer.

Opencv projectpoints() documentation. (2016). Retrieved 2016-05-30, from http://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#projectpoints

Peirce, J. W. (2007). Psychopy—psychophysics software in python. *Journal of neuroscience methods*, *162*(1), 8–13.

Pupil Labs. (2015a). *Issue #352 start and stop recording with zeromq requests* [Github Issue]. Retrieved 2016-05-16, from https://github.com/pupil-labs/pupil/issues/352

Pupil Labs. (2015b). *Pull Request #370 pupil remote* [Github Pull Request]. Retrieved 2016-05-16, from https://github.com/pupil-labs/pupil/pull/370

Pupil Labs. (2015c). *Pull Request #385 zmq ipc* [Github Pull Request]. Retrieved 2016-05-16, from https://github.com/pupil-labs/pupil/pull/385

Pupil Labs. (2015d, Sep). *Pupil version release v0.6* [Release Notes]. Retrieved 2016-05-16, from https://github.com/pupil-labs/pupil/releases/tag/v0.6

Pupil Labs. (2016a). *make_square_markers.py commit 8ce5e6f* [Source Code]. Retrieved 2016-05-16, from https://github.com/pupil-labs/pupil-helpers/blob/8ce5e6f320be3fc11f3fcee71a5541dd2b69e384/make_square_markers.py

Pupil Labs. (2016b). *network_time_sync.py commit 8ce5e6f* [Source Code]. Retrieved 2016-05-31, from https://github.com/pupil-labs/pupil-helpers/blob/8ce5e6f320be3fc11f3fcee71a5541dd2b69e384/pupil_sync/network_time_sync.py

Pupil Labs. (2016c). *plugin.py commit 9debff7f43* [Source Code]. Retrieved 2016-05-16, from https://github.com/pupil-labs/pupil/blob/9debff7f4333f66693d81de069b4a955797e6db0/pupil_src/shared_modules/plugin.py

Pupil Labs. (2016d). *Pupil* [Repository]. Retrieved 2016-05-16, from https://github.com/pupil-labs/pupil/

Pupil Labs. (2016e). *Pupil hardware do-it-yourself guide.* Retrieved 2016-05-30, from https://github.com/pupil-labs/pupil/wiki/Getting-Pupil-Hardware#diy

Pupil Labs. (2016f). *Pupil helpers* [Repository]. Retrieved 2016-05-30, from https://github.com/pupil-labs/pupil-helpers

Pupil Labs. (2016g). *Pupil labs store.* Retrieved 2016-05-30, from https://pupil-labs.com/store/

Pupil Labs. (2016h). *pupil_sync_complete.py commit 8ce5e6f* [Source Code]. Retrieved 2016-05-31, from https://github.com/pupil-labs/pupil-helpers/blob/8ce5e6f320be3fc11f3fcee71a5541dd2b69e384/pupil_sync/pupil_sync_complete.py

Pupil Labs. (2016i, Jan). *Pupil version release v0.7.1* [Release Notes]. Retrieved 2016-05-16, from https://github.com/pupil-labs/pupil/releases/tag/v0.7.1

Pupil Labs. (2016j, Mar). *Pupil version release v0.7.4* [Release Notes]. Retrieved 2016-05-16, from https://github.com/pupil-labs/pupil/releases/tag/v0.7.4

Pupil Labs. (2016k, May). *Pupil version release v0.7.6* [Release Notes]. Retrieved 2016-05-16, from https://github.com/pupil-labs/pupil/releases/tag/v0.7.6

Pupil Labs. (2016l). *pyglui, cython powered opengl gui* [Repository]. Retrieved 2016-05-30, from https://github.com/pupil-labs/pyglui

Pupil Labs. (2016m). *pyuvc, python bindings for libuvc* [Repository]. Retrieved 2016-05-31, from https://github.com/pupil-labs/pyuvc

Pupil Labs. (2016n). *Surface tracking documentation.* Retrieved 2016-05-16, from https://github.com/pupil-labs/pupil/wiki/Pupil-capture#surface-tracking

Python 2.7 Glossery. (2016). *Keyword argument* [Glossery]. Retrieved 2016-05-30, from https://docs.python.org/2.7/glossary.html#term-argument

Repository, P. (2016). *circle_detector.py commit 9debff7f43* [Source Code]. Retrieved 2016-05-30, from https://github.com/pupil-labs/pupil/blob/9debff7f4333f66693d81de069b4a955797e6db0/pupil_src/shared_modules/circle_detector.py

repr function. (2016). Retrieved 2016-05-30, from https://docs.python.org/2/library/functions.html#func-repr

Suzuki, S., et al. (1985). Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, *30*(1), 32–46.

Swirski, L., & Dodgson, N. (2013). A fully-automatic, temporal approach to single camera, glint-free 3d eye model fitting. *Proc. PETMEI*.

Triggs, B., McLauchlan, P. F., Hartley, R. I., & Fitzgibbon, A. W. (1999). Bundle adjustment—a modern synthesis. In *Vision algorithms: theory and practice* (pp. 298–372). Springer.

Ullenboom, C. (2004). *Java ist auch eine insel* (Vol. 1475). Galileo Press.

Zhang, Z. (2000). A flexible new technique for camera calibration. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, *22*(11), 1330–1334.

Zyre. (2016). *Zyre, an open-source framework for proximity-based peer-to-peer applications* [Repository]. Retrieved 2016-05-30, from https://github.com/zeromq/zyre

Declaration of Authorship

I hereby certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other university.

Osnabrück, June 3, 2016