# BioTracker- an extensible computer vision framework

Bachelor Thesis

Julian Alexander Tanke
Matrikelnummer: 4664328


Gutachter: Dr. Tim Landgraf
Zweitgutachter: Prof. Raúl Rojas

January 2016

# Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben. Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Datum Unterschrift

**Abstract**

Video tracking aids users in various domains. In medicine, it plays an increasingly important role in assisting clinicians in diagnosis[1] while in arts image processing helps Special Effects technicians in applying visual effects to movies[2]. Another area of application is surveillance where real-time video tracking enables security personnel to monitor a large set of video collectors[3]. The BIOROBOTICS LAB profits greatly from computer vision[4][5] and realized that many parts of vision applications can be reused for future projects to reduce programming overhead. Functions, that are useful, regardless of the computer vision task, include loading and decoding videos, the interaction with image and video data, like pausing, panning and zooming, and means to serialize trajectory data. This motivated the BIOTRACKER [6], a modular open source C++ framework for computer vision applications. However, the application still experiences a number of issues and inconveniences. For example, it cannot be embedded into other programs or used in automated environments like servers and super computers. Furthermore, extending the application with new computer vision algorithms is unintuitive and error-prone. This thesis restructures the BIOTRACKER to tackle the aforementioned problems and furthermore introduces a novel technique to bring scriptability to the application using ØMQ, a high performance messaging queue, to greatly simplify the creation of new tracking modules. The new BIOTRACKER framework enables users to apply general purpose computer vision and image processing techniques to any type of domain, not limited to Biology.

# Contents

# Chapter 1

# Introduction

Today, computer vision has many applications including traffic control[7], entertainment[2][8] or medicine[1] and an increasing number of scientists apply this technologies to advance their research. For example, in Biology, animal behavior is recorded on video to later apply statistical analysis on individuals trajectories[9]. Automating this process is beneficial as it discards labor intensive and subjective work. The BIOROBOTICS LAB[1] makes extensive use of computer vision[5][4][10] and it became evident that many parts of computer vision applications can be reused in future projects. This includes functions that load and decode video files, functions that interact with the images and videos, like pausing, panning and zooming, and means to serialize trajectory data. This motivated the BIOTRACKER [6], a modular open source C++ framework for computer vision applications. It attempts to bring together users, who want a tool to aid their research, and developers, who want to implement their own computer vision and tracking algorithms. Users can apply arbitrary computer vision algorithms to any video or image. When developing the BIOTRACKER, strong focus was put on usability as many free video tracking tools, like CTRAX [9] and IDTRACKER [11], suffer in this regards. Furthermore, the application can be used for free distinguishing it from commercial tools like ETHOVISION [12]. Developers can easily implement their own vision algorithms as the inversion of control[13] simplifies the way they interact with the framework, when compared to well-known computer vision libraries like OPENCV [14]. Chapter 2 will detail this in more depth.

However, several issues can be identified in the current version of the BIOTRACKER. For example, the BIOTRACKER is not, as claimed, a framework that can be used by other applications but rather a coherent desktop application that can be extended to some degree. Writing new algorithms is centralized and unnecessarily complicated as they must be compiled into the application, forcing developers to add their code to the GIT source tree of the BIOTRACKER. This bloats the development process and encourages developers to bypass the framework as they can directly access other parts of the application. The centralization makes it impossible to individually develop new vision algorithms and share them with other users as they are

[1] http://biorobotics.mi.fu-berlin.de/wordpress/

always combined with the complete BioTracker application.

This thesis diversifies the BioTracker making it a true framework. The application is split into a Core framework and a GUI application enabling the BioTracker to be embedded into other applications and to be used in automated environments like servers and super computers. Furthermore, the workflow for developing new computer vision algorithms is improved as they can be build independently as shared libraries which can be loaded by the Core framework. As an addition, vision algorithms can now be written in script languages like python to encourage rapid prototyping of new image processing techniques. Our long-term goal is to provide a full ecosystem for image processing and computer vision.

# Chapter 2

# State of the art

In Biology and Ethology several computer vision tools are used to help researchers. EthoVision [12], a commercial tool used in many laboratories, is specialized on tracking rodents in well-defined environments. Extensions and plugins enable it to track fishes, larvae and insects, the latter both in field and under laboratory conditions. Further extensions let users track any set of animals from arbitrary video feeds. Tools for data selection, visualization and analysis help researchers to detect patterns and enable them to analyze the output. However, the basic version costs $5850.00 and is difficult to afford for many research laboratories. Furthermore, EthoVision can only be extended with plugins from Noldus Information Technology, the the company behind the application. While this assures a certain level of quality it makes users depend on a company's decision whether a certain research area is profitable or not - which could hold back unorthodox research that simply does not fit into any predefined framework. Ctrax [9] is an open-source, background-subtraction-based computer vision program for estimating the positions and orientations of many walking flies and other insects that has been used in various research [9] [15] [16] and is written in python. However, the application is rather unintuitive and slow, supports only few video formats and uses outdated software components like motmot.wxvideo[1]. Furthermore, Ctrax is geared towards a special set of experiments and animals and can be neither extended nor scripted. zootracer [17] was developed at Microsoft Research as a collaborative work between a computer vision researcher and an ecologist. It is capable of accurately tracking multiple, unmarked, interacting individuals in arbitrary video footage. However, the application only works on Microsoft Windows and the installation process is clumsy as users have to manually copy outdated external libraries. According to **Lucas Joppa**[2], one of the people who envisioned zootracer, the application was abandoned when the team split up. idTracker [11] is a tracking software that manages to keep track of the correct identity of each individual in a video using probabilistic estimations. However, idTracker can only be applied to videos as it needs to run through the whole video before tracking. This prevents life tracking with video cameras. Furthermore, as

[1] https://github.com/motmot/wxvideo (last commits are from 2012)
[2] http://research.microsoft.com/en-us/people/lujoppa/

the developers only wanted to showcase their algorithm, they did not spent much effort into the graphical user interface which is not intuitive and which, like the whole application, yields poor performance. WINAnalyze is an automatic motion analysis software that can track arbitrary objects without markers. It is used worldwide, and even in space[3], for motion analysis applications. However, it is commercial and provides only a set of functions that cannot be extended by the user. Furthermore, it only works on Microsoft Windows. OpenCV [14] is an open-source programming library that provides a set of functions for computer vision and machine learning. Its goal is to provide a common framework for computer vision applications. OpenCV implements a comprehensive set of classical as well as state-of-the-art algorithms and provides basic GUI elements. Inherit to its nature as a programming library, Though it can be scripted, for example using python, OpenCV requires good understanding of software development and software engineering. The BioTracker [6], a modular open source c++ framework for computer vision applications that runs on all major platforms, was developed at the Biorobotics Lab. As mentioned in Chapter 1, it was motivated by the observation, that many computer vision applications share the same basic functions, a graphical user interface, means to load and store data and the capability to interact with the video or image. The BioTracker allows researchers to implement their own tracking algorithms but also to utilize various algorithms that are already implemented. The BioTracker allows users to load images, videos or access video cameras so that arbitrary computer vision algorithms can be applied. The application aids the user with serializing the tracking output, restoring old tracking states and provides easy means to visualize data. Computer vision algorithms are instantiated in the form of **Trackers**. Trackers define how they interact with images and how their results should be visualized. Furthermore, trackers let users directly interact with them by extending the graphical user interface with their own widgets. Figure 2.1 explains the basic functionality of a **Tracker** in more detail. The BioTracker still faces some issues which are discussed in Section 3.1.

[3]http://www.mikromak.com/download/WINanalyze_ISS_Use.pdf

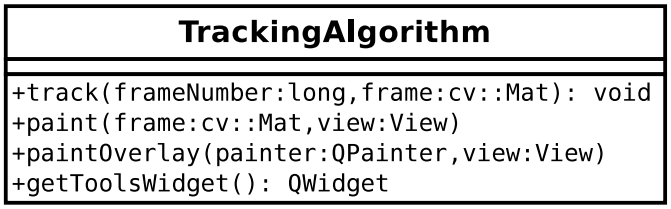


Figure 2.1: Basic Tracker functions: **Track** must implement the tracking algorithm. Its parameters are the current frame number as well as an OpenCV matrix that represents the picture. **Paint** defines the way the OpenCV matrix is rendered on-screen: if this function does not change the matrix, the current frame is drawn. However, the user is able to transform the matrix and is even allowed to transform the dimensions (1 and 3 channels) to achieve a different visualization. Furthermore, the tracker can define a set of **View**'s to allow several distinct ways of rendering. **PaintOverlay** enables the user to draw any kind of overlay onto the image using a QPainter. Similar to **paint**, a view may alter the way the tracker represents the data. **GetToolsWidget** allows the user to add any combination of Qt-elements enabling a wide range of input and output options.


Figure 2.2: BioTracker: The BioTracker enables users to load videos or images and apply computer vision algorithms to them. Users can either choose algorithms from a selected list or create their own implementation. The file menu (**1**) enables users to load different resources into the BioTracker application. This resources can either be data to work on, like pictures, videos or access to the computer's camera or data from previous executions of the program that has to be loaded into the current tracker. Users may choose from various computer vision algorithms (**2**) to analyze the given pictures or videos which are shown in the application (**3**). Each tracker can provide arbitrary GUI elements (**4**) to interact with the user or send output information to the console (**5**). The BioTracker provides functionality that aids handling videos or sets of images, like an interactive progress bar (**6**), means to pause, play, stop, or step through the data (**7**), the ability to pan and zoom the current image (**8** ), the option to modify the replay speed (**9**) as well as an actual calculation of the current replaying speed (**10**). Last but not least, trackers are enabled to provide different views (**7**) to give additional insight into a video or algorithm.

# Chapter 3

# Implementation

## 3.1 Problem Statement

The current BioTracker successfully combines usability for users with extensibility for developers. However, some issues can be identified:

- Problem 1 **: Lack of usability in automated environments**: the current BioTracker requires a human to operate the program. However, some use case might need to track video data in automated environments, for example when executed on a server or when integrated into a pipeline process that tracks huge numbers of videos at once.
- Problem 2 **: Implementing new tracking algorithms is still unnecessarily complicated** : though the BioTracker lets users write their own tracking algorithm it is still rather involving to actually integrate said algorithm into the application as the user must incorporate his tracker into the BioTracker source code and then recompile the whole project. This actually defeats the purpose to easily share trackers.
- Problem 3 **: No standardized infrastructure**: Though the Biorobotics Lab provides some infrastructure to host and manage the development of the BioTracker it is not publicly available yet. This makes it difficult for new developers and for programmers who are not affiliated to the Biorobotics Lab to contribute. Furthermore, setting up the development environment is not trivial and poses an entry barrier for software development beginners.

## 3.2 Requirements

To solve the aforementioned problems this thesis proposes the following solutions:

- Proposal 1: In order to tackle Problem 1 the BioTracker is split into two separate projects, a Core framework, which provides a clean API that handles all tracking

functionality, tracker loading and resources and a GUI application that only manages the graphical interaction with users. The Core will be made available as a shared library that is used by the GUI, the actual entry point for the BioTracker application. This way, other applications can use the Core functionality without relying on a graphical interface.

- Proposal 2: To solve Problem 2 users are enabled to write their own trackers as shared libraries. The Core loads those trackers and applies them to input data. Furthermore, the new BioTracker can be scripted to enable fast prototyping of algorithms and to further ease the use of the tracker. To enable maximum flexibility, a ØMQ [18] interface is offered with which any programming language that supports ØMQ can be used to interface with the BioTracker. ØMQ (zeroMQ) is a high-performance asynchronous message queue that does not rely on an underlying dedicated message broker.

- Proposal 3: To make BioTracker development more accessible the git repository was open sourced and moved from the Biorobotics Lab server to Github. Additionally, scripts and Docker-images are provided to ease the setup of development environments. This is explained in more detail in the future work of Michael Wittig[19].

## 3.3 Architecture

### 3.3.1 Core-GUI-split

The first task was to separate the classes of the old BioTracker into elements for the GUI application and for the Core library. This was non-trivial as the code grew several years (the project started in 2012) and elements that were chosen to be part of the Core were strongly intertwined with elements that had clear implications on GUI side. On top of that, the old implementation used inheritance to pass information from Core parts to GUI elements. Instead of using inheritance, composition is now used to wire together Core and GUI (composition over inheritance[20]). To hide and encapsulate the complexity of the Core framework, like multithreading, a facade class was created that provides basic interfacing into the library. In Figure 3.1 an overview of the new class layout is shown.

To simplify graphics development and to remove several bugs related to it, all native OpenGL calls where replaced with Qt functions as they make use of OpenGL but provide a much simpler API. Our tests suggest that this does not harm the performance of the application while the simplification of the source code makes the BioTracker much easier to maintain.

As CMake is used to manage the build system, CPM (c++ Package Manager) is used to simplify the embedding of our libraries. CPM is based on CMake and git and, as a package manager, automates the downloading and updating of c++ modules.

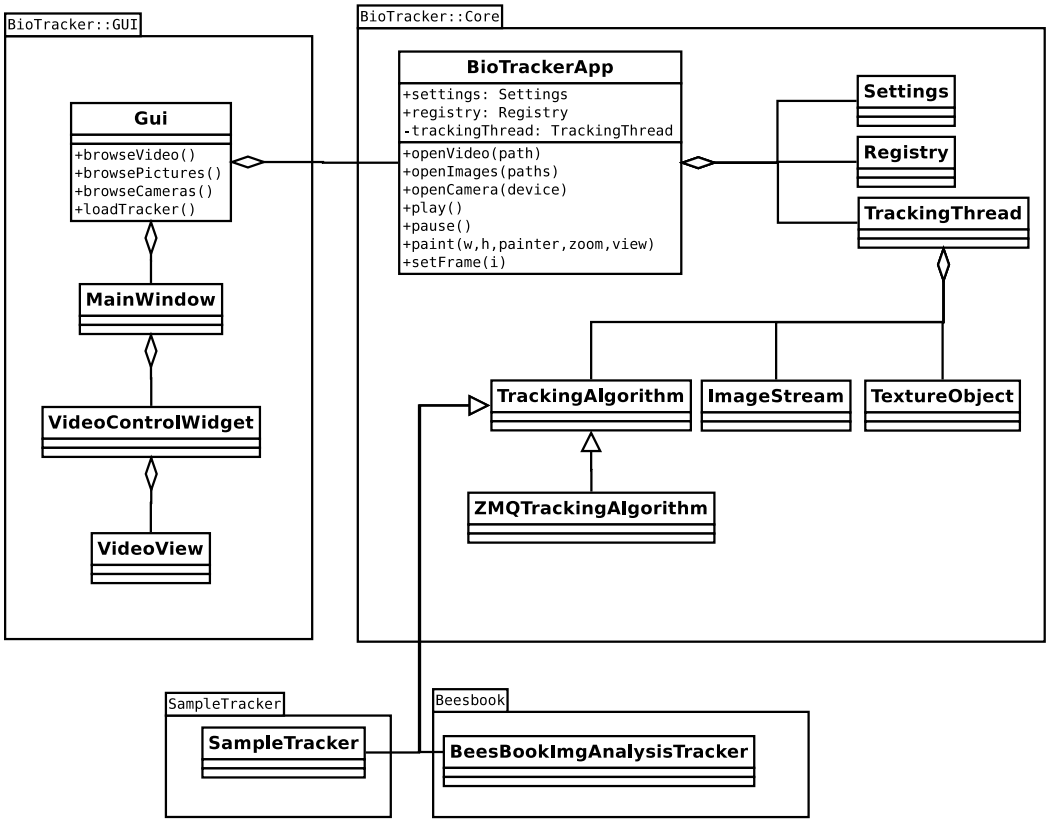


Figure 3.1: Simplified class diagram of the new BioTracker implementation: instead of being one giant module the BioTracker now consists of the two modules Core and GUI. Furthermore, trackers are implemented as their own module which use the Core as a dependency. The Core API is defined in the **BioTrackerApp** facade class which is the entry point for users of the library. From there one can load videos and images, access other elements of the Core like **Settings** and load and start new trackers. The **TrackingThread** manages the actual tracking and is for internal use only. The **TrackingAlgorithm** is an abstract class that represents a tracker. Users who want to implement their own tracking algorithm must inherit from this class. The **ZmqTrackingAlgorithm** handles the communication between BioTracker and ØMQ tracker applications and is for interal use only. All graphical user elements, like file selection or video viewing, are handled in the GUI module that has the Core as a dependency.

### 3.3.2 Ømq

Message queues are well-known software-engineering components that enable processes to communicate over well-defined protocols. Control and content is passed in form of messages while a queue ensures correct temporal order. ØMQ (zeromq) is a high-performance asynchronous message queue that does not rely on an underlying dedicated message broker that is typically present in comparable use-cases. It enables two or more processes to communicate with atomic messages, even over network borders. ØMQ can carry messages over a broad set of network protocols, including TIPC, inproc and TCP. Trackers run on top of TCP as it is the most portable socket protocol across various operation system. ØMQ ensures that messages arrive in one piece and in order. It supports various message-queuing patterns like pubsub, push-pull or PAIR. The **PAIR** pattern that is described in figure 3.2 is used as it is easy to setup and

allows bidirectional communication. Only one peer can connect to a server, who listens on a certain port.

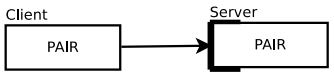


Figure 3.2: ØMQ PAIR pattern: Paired ØMQ sockets are similar to regular sockets and their setup is easy: a server simply binds a port to which a client may then connect. After initializing the connection, both server and client may send messages to each other. PAIR'ed sockets have a 1 : 1 connection which means that no other client can connect to the server as long as it is connected to some client.

The client executes an extremely simple state machine that is depicted in Figure 3.3.

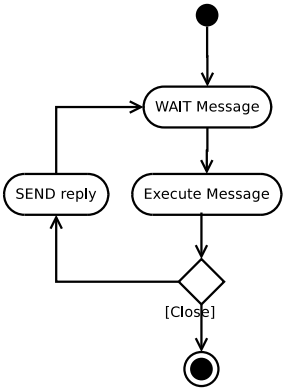


Figure 3.3: ØMQ state machine: the ØMQ client process starts and immediately waits for incoming messages, sent by the server. Upon message arrival, the client executes the task and then, depending if it is instructed do so, either shuts down or sends back a reply and proceeds waiting for further messages.

The communication between client and server is modeled as a master/slave relation: only one side, the server, is allowed to initiate communication, the other side, the client, can only send messages when it explicitly is asked to do so. The advantage of this construct is its simplicity that allows a simple implementation like the one depicted in Figure 3.3.

BioTracker trackers can emit asynchronous events to force a redraw, jump to particular frames or to provide feedback using arbitrary text. This cannot be directly handled with the master/slave ØMQ setup mentioned above as it would require the server to constantly listen for incoming events and thus block the whole application. Instead, a simple heuristic is used that events will most likely be thrown when the client is executing protocols that are initiated by the server (like paint, paintOverlay, track, mouseClick, etc.). This is shown in detail in Figure 3.4.

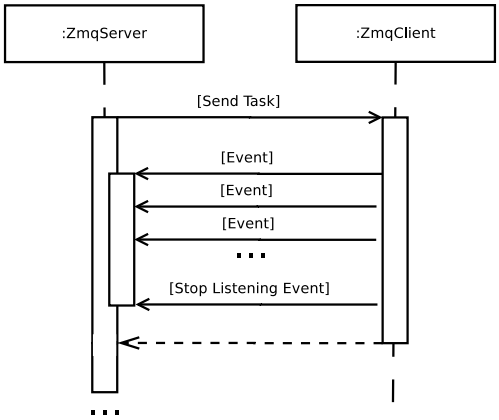


Figure 3.4: ØMQ event listening: The server initiates communication and tasks the client to execute a given protocol. The server then starts a loop waiting for any event message that the client sends. Upon arrival of said event message the server executes the actual event. When the client is done with its task it sends a **Stop Listening Event** that tells the server to stop listening for events and to continue executing the program.

The disadvantage of this heuristic is that it makes every call to the client blocking. Users therefore must ensure that any other function than **track** must return as soon as possible to not freeze the GUI. Tracking is executed in another thread and thus the GUI stays responsive. The whole life cycle of the ØMQ environment can be seen in Figure 3.5.

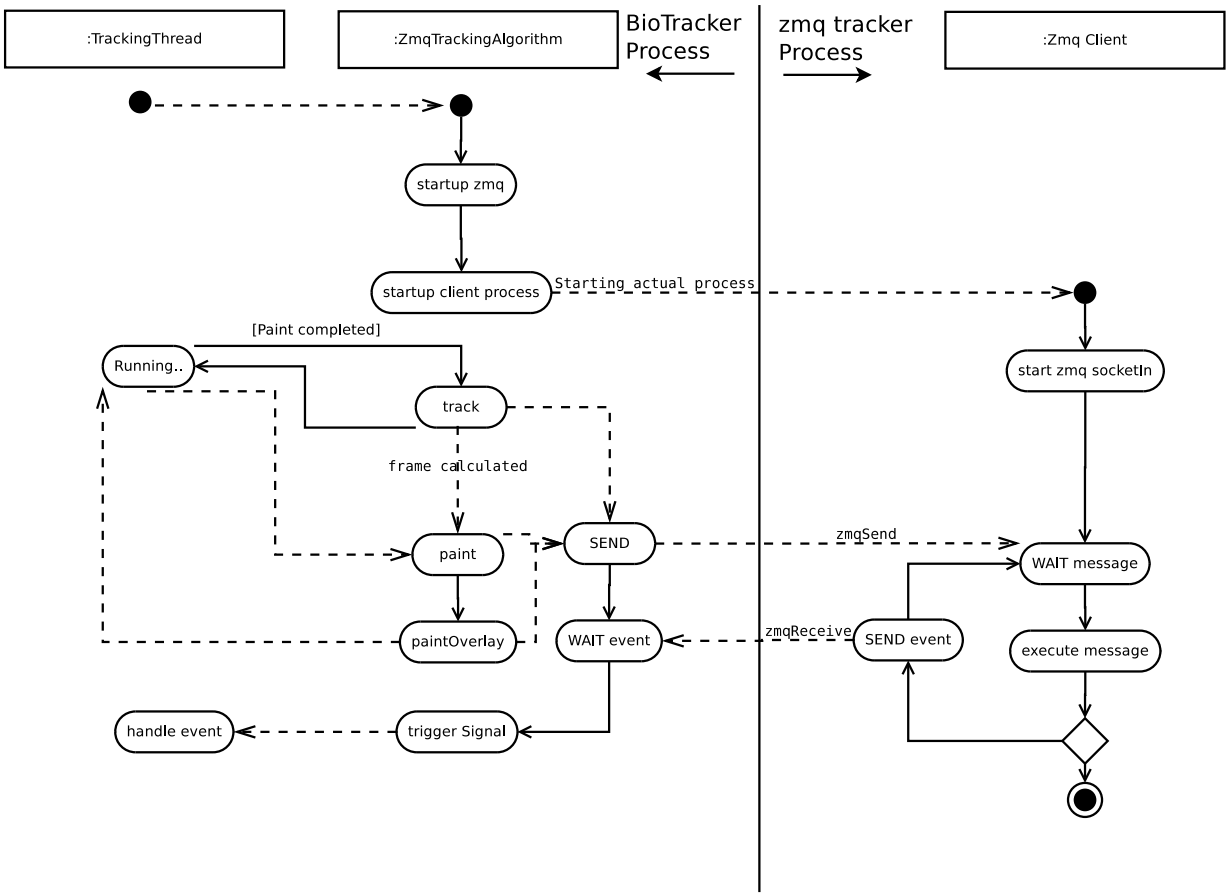


Figure 3.5: ØMQ life cycle: given a running BioTracker application, a **Tracking Thread** is started. This thread loops forever to trigger track-calls into the respective tracking algorithm. When a ØMQ tracker is selected it binds a ØMQ socket and then starts the client process. The client, upon startup, initiates the network connection with the server over ØMQ and then executes the protocol that was described in Figure 3.3. From the tracking thread, the BioTracker then initiates a track-request. When this is done, the **ZmqTrackingAlgorithm** triggers the event **frame calculated** which initiates the rendering from the main thread. When the full rendering cycle is finished the BioTracker releases a lock that blocks the **Tracking Thread**'s endless loop. During the whole process, whenever the server initiates communication with the client, it listens for events to trigger them.

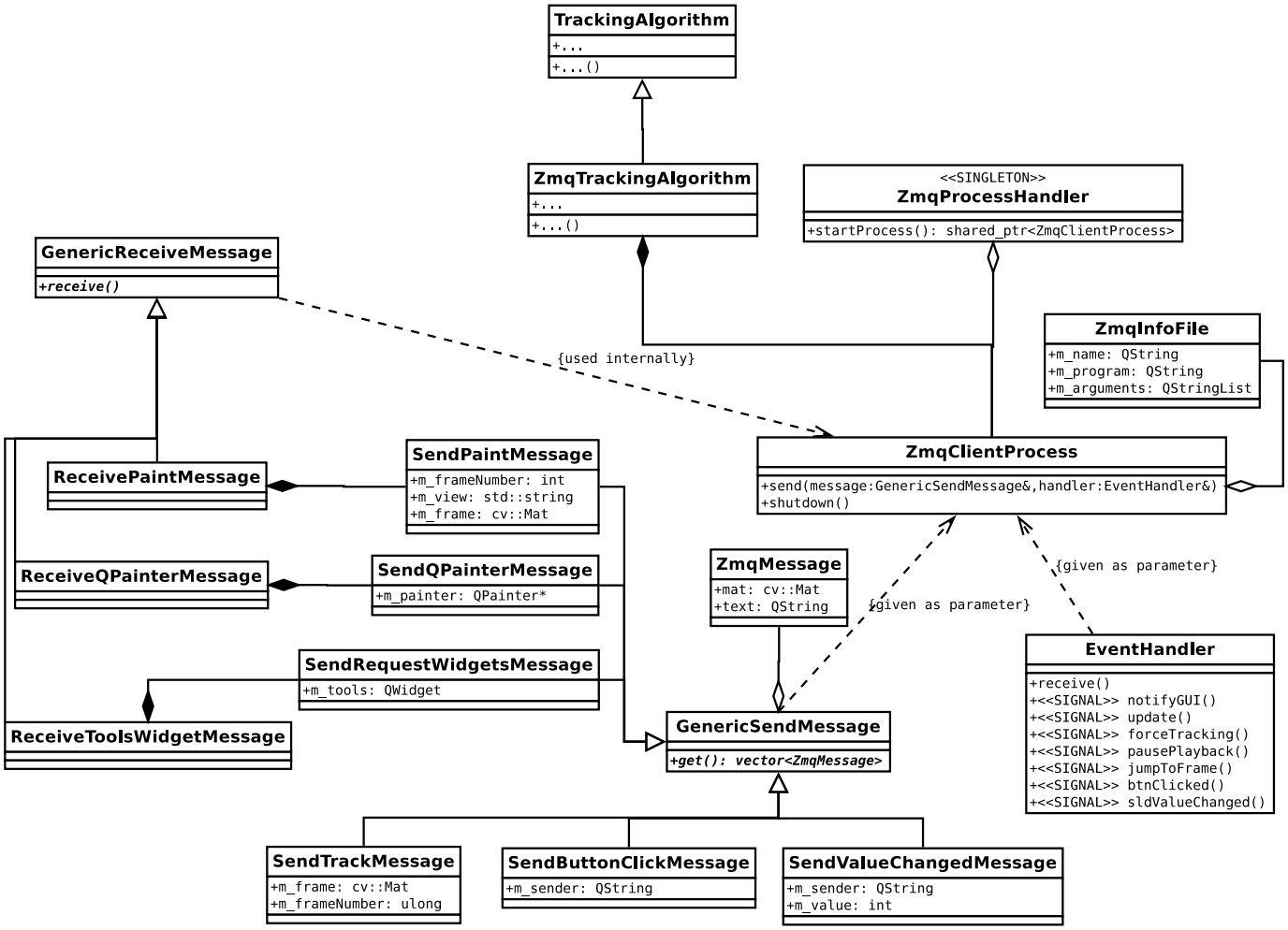


Figure 3.6: ØMQ class diagram: the **ZmqTrackingAlgorithm** inherits from the **TrackingAlgorithm** described in Figure 2.1. This implementation can then be used for tracking in the BioTracker application. The **ZmqProcessHandler** is a singleton that generates **ZmqClientProcess**es, an abstraction for the actual client process that provides simple methods for sending and receiving messages. **GenericReceiveMessage** and **GenericSendMessage** are abstractions for the actual messages that are sent or received. They encapsulate the definition of how ØMQ messages are parsed which makes it very easy to change the ØMQ message structure in the future. **ZmqInfoFile** represents the ØMQ client description file explained in Listing 3.1 and is needed to generate the **ZmqClientProcess**. The **EventHandler** handles the asynchronous events sent by the client like notifications and user input.

As any program that can execute ØMQ can pose as a tracker, a simple text file is provided, depicted in Listing 3.1 to inform the BioTracker of how to start the respective ØMQ client process.

Listing 3.1: file.tracker.zmq

```
1 # Comment
2 python3
3 sobel.py
```

Listing 3.2: command that is generated from Listing 3.1

```
1 python3 sobel.py
```

Any line that starts with a # is ignored as a comment. The first non-comment line will be interpreted as the program that executes the script file while the second non-comment line will be interpreted as the actual script that will be executed. Listing 3.2 yields the resulting command generated from the file 3.1. As mentioned above, this enables the BioTracker to use any tracker written as ØMQ client, regardless of the underlying technology. The class structure to manage the ØMQ tracker on the server side is described in detail in Figure 3.6.

ØMQ provides no explicit message structure. To keep the interface as simple and as far-reaching as possible, a primitive message system was implemented on top of ØMQ (Figure 3.7). **ZMQ_SNDMORE** is a flag provided by ØMQ to summarize multiple messages into one multi-part message. ØMQ ensures that a multi-part message is either completely received or not received at all and furthermore ensures that the order stays intact. By convention, the first part of the multi-part message contains a single string that represents the type of task that the message transports. The ØMQ client, as well as the server, must exactly know how to handle each message type. Strings are chosen over binary representation as many languages provide extensive tools to work with this data type which reduces the complexity to support new languages.

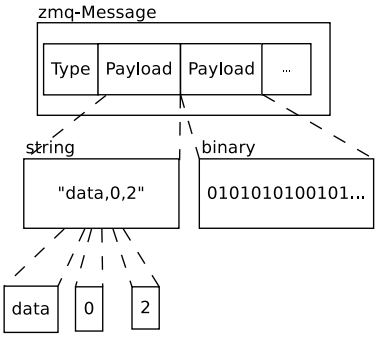


Figure 3.7: BioTracker ØMQ message layout: the top layer represents a ØMQ multi-part message where each inner box is its own ØMQ message. The BioTracker standard defines that the first message must represent the type of of the message, encoded as a string. Any message following the initial type-message contains payload which can either be in binary form, which is used to transport the content of images, or a comma-separated string that gives additional information regarding the type of message. The different types are described in Table 3.1.

| id | Function | # of payload messages | payload |
|---|---|---|---|
| 0 | Track | 2 | 1.) image shape, 2.) image binary data |
| 1 | Paint | 1 | 1.) paint parameters |
| 2 | Shutdown | 0 | |
| 3 | PaintOverlay | 0 | |
| 4 | requestTools | 0 | |
| 5 | toolsUpdate | 1 | 1.) widget id + parameter |

Table 3.1: ØMQ server messages: the messages defined in this table are closely related to the functions explained in Figure 2.1. The **id** column contains the type definition of each message which will be sent as first message for any message type. This is necessary so that the receiver of the message can identify what type of action to execute. As a BioTracker message can contain multiple payload messages we also describe how many payload messages each message type provides and what purpose they serve. **Track** tells the client to do tracking. The payload consists of the image shape, encoded as comma-separated string, and the image itself in row-major order, encoded as binary blob. **Paint** requests the client to send a modified image, if applicable. If the client wants its modified image to be drawn, it must utilize the same encoding used for **track** to send the image back to the server where it will be drawn. **Shutdown** tells the client to end the process. **PaintOverlay** allows the client to draw overlay objects onto the image. The objects that should be drawn must be string-encoded. At the moment, the BioTracker supports only basic shapes like rectangles and circles. **RequestTools** requests a set of GUI widgets that allow the user to directly interact with the tracker. At the moment only text-, button- and slider-widgets are supported. Last but not least, **toolsUpdate** is used to update the widgets that were sent with **RequestTools**.

To showcase the power and flexibility achieved by using a ØMQ tracker a python library was developed that handles all complications of communicating with the BioTracker application. Images send by the server are converted to NumPy arrays. In Listing 3.3 we provide an example tracker implementation for applying the Sobel operator[21] to an image.

Listing 3.3: Sobel PYTHON implementation

```
from biotracker import (Signals, Helper, Button, run_client)
from scipy import signal

Mat = None
Kx = np.array([[-1,0,1],[-2,0,2],[-1,0,1]])  # sobel operator

def track(frame, M):  # M is a n*m*3 rgb matrix
    global Mat, Kx
    M = Helper.rgb2gray(M)
    Signals.notify_gui("send hello")  # show a message on the GUI
    Mat = signal.convolve2d(M, Kx, boundary='symm', mode='same')

def paint(frame):
    global Mat
    return Mat  # when a matrix is returned here, it will be drawn

def paint_overlay(qpainter):
    pass

def button_click():
    pass

def request_widgets():  # Generate a button that lets users interact
    return [Button("Click me", button_click)] # with the tracker

def shutdown():
    pass

run_client(  # start the tracking client using the functions
    track,   # that are implemented above
    paint,
    paint_overlay,
    shutdown,
    request_widgets=request_widgets)
```

# Chapter 4

# Experiments

The ØMQ implementation comes with additional costs as the OpenCV matrix must be parsed, copied, sent at the server side and received and restored at the client side for each frame. In the worst case, the client sends back the whole, possibly modified, frame for the BioTracker to paint. To evaluate the performance of the ØMQ implementation, two trackers are compared, one implemented in python using NumPy and ØMQ and the other implemented in c++ using OpenCV. The trackers should convolve the Sobel operator $G_x$ with the given image $I$ to produce the derivative $I_x$. As $I$ has 3 channels, BGR, it must first be converted into a grayscale picture $I'$ using $f$. The tracker then returns $I_x$ to the BioTracker for painting.

$$f : \mathbb{Z}^3_{<255} \to \mathbb{R}_{+,<255}, f(\vec{x}) = \vec{x} \cdot \begin{pmatrix} 0.114 & 0.587 & 0.299 \end{pmatrix}$$

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$I' = f(I)$$

$$I_x = I' * G_x$$

Figure 4.1 shows the communication workflow that is needed for the ØMQ implementation. The process depicted there cannot be run in parallel but must be executed in order as the ØMQ client first needs to get a copy of the image so that it can modify and return it later.

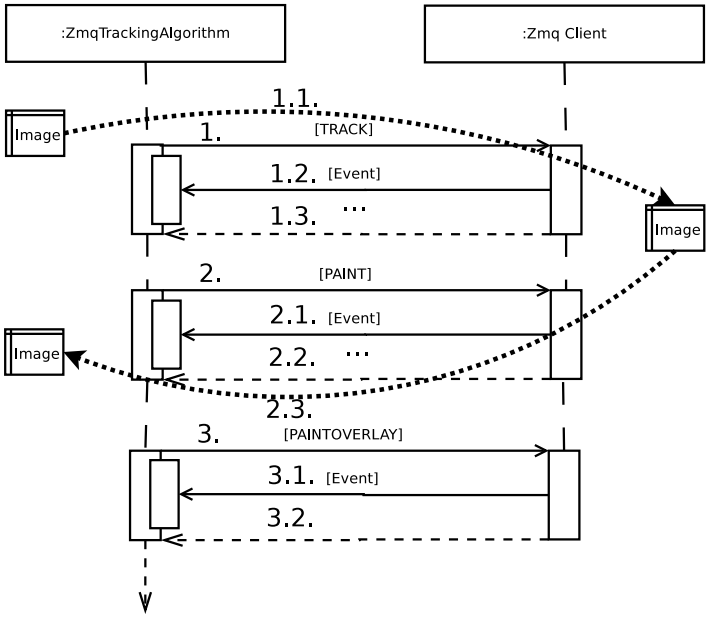


Figure 4.1: Tracking and rendering cycle for the performance experiment: at **1** the ØMQ server initiates the tracking by sending the track-message. Integrated into the message is a serialized copy (**1.1**) of the current frame as server and client do not share memory. In **1.2** the server waits for possible messages sent by the client and in **1.3** the client notifies the server that the tracking is over. Now the server can initiate the paint-process **2**. As in **1.2**, events are handled in **2.1** until the ØMQ client finishes **2.2** and sends back the modified image(**2.3**). In **3** the ØMQ server goes on by initiating the paintoverlay process, again, waiting for arbitrary events in **3.1** and finally receiving the QPainter-events in **3.2**

The calls executed in **1.1** and **2.3** are the bottlenecks of the whole procedure as the potentially large image must be serialized, copied, sent over a network socket, received, and restored. In the experiment, this is needed twice as the BioTracker has to paint the image modified by the tracker.

## 4.1 Performance Test

**Frame rate** (fps) is an easy-to-calculate yet descriptive metric for measuring overall performance. The performance experiment evaluates how the two Sobel-implementations, one in C++ and the other in python-ØMQ, degenerate with increasing image size. To calculate the average frame rate, the experiments were run with videos that contain 200 frames where each $n \times n$ frame has 3 channels, each 8-bit deep. The experiment starts with a video with frame size $50 \times 50$ and then doubles the size until $6400 \times 6400$ is reached. The BioTracker attempts to meet $60 fps$ to be in sync with the monitors refresh rate. All benchmarks were conducted in following environment:

- Ubuntu 15.10, Kernel 4.2.0-25-generic
- Intel(R) Core(TM) i7-4500U CPU @ 1.80GHz

- 7876MiB memory

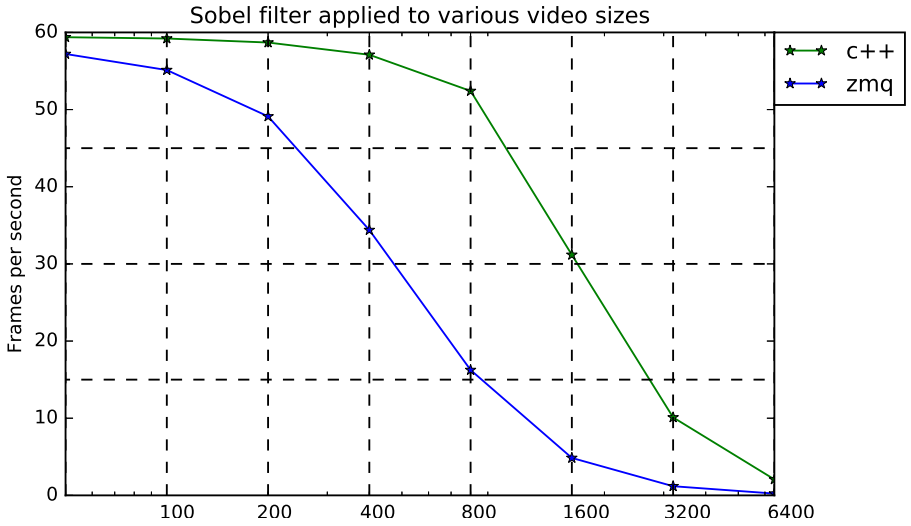


Figure 4.2: Frame rate: the x-axis represents the size $n$ of the $n \times n \times 3$ frame while the y-axis depicts the frames per second, averaged over 200 frames. The green line represents the C++ tracker while the blue line represents the ØMQ tracker. The graph suggests that the performance of the ØMQ tracker degenerates faster than the one of the C++ tracker, which was expected.

Figure 4.2 depicts the results of the frame-rate experiment and yields results that were to be expected: the frame rate of both trackers degenerates, the more the frame size grows. The ØMQ tracker degenerates faster than the C++ tracker which can be explained with the additional overhead that is added by copying frames, sending them, and restoring them again.

## 4.2 Memory consumption

Another important metric is the **memory footprint**, which measures the amount of main memory that a program uses while running. To measure the used memory in C++, VALGRIND [22] is used which provides MASSIF, a tool that measures how much heap memory a program occupies. To determine the consumed memory for the PYTHON process, the LINUX-specific operation system call **getrusage** is used reading the field **ru_maxrss** from the result which represents the resident set size in kilobytes used by the process. The resident set size represents the amount of main memory that is occupied by the process.

This experiment is threefold: it measures the consumed memory for three separate runs where the same video is replayed:

1. At first, the video is used with the same C++-Sobel tracker used in the frames-experiment in Section 4.1

2. Then, no tracker is used at all to compare the memory usage

3. And, in the end, the ØMQ-Sobel tracker from the frames-experiment is used as well. In this experiment both, the memory consumed by the BioTracker program as well as by the ØMQ client in PYTHON, is measured.

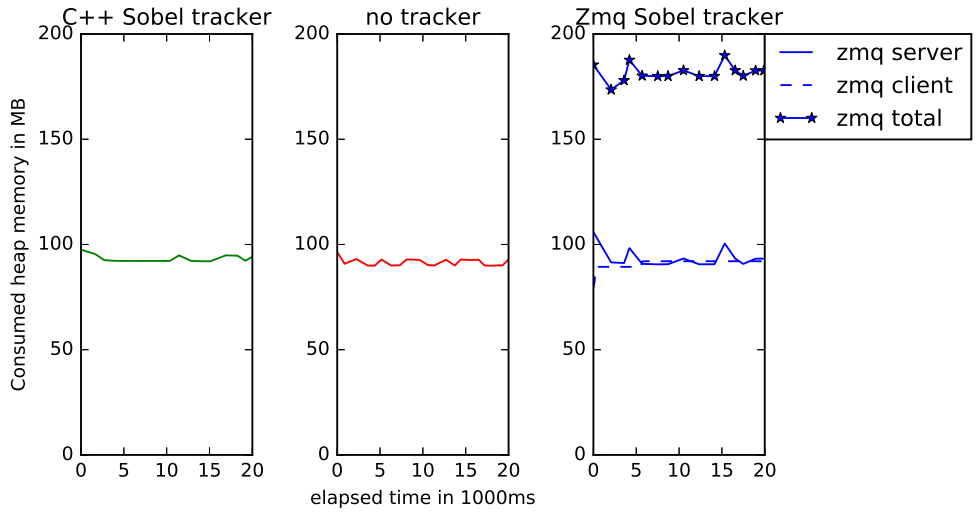


Figure 4.3: Memory consumption: The x-axis depicts the elapsed time in seconds while the y-axis represents the consumed memory in megabyte. The left graph shows the memory consumed by the BioTracker when using the C++ Sobel tracker while replaying a video. The middle graph shows the memory consumption of the BioTracker when only replaying a video. The last graph shows the memory consumption of the BioTracker when using the ØMQ tracker to replay a video. The strike-though line represents the ØMQ server side, the actual BioTracker application, while the dashed line represents the memory consumption of the PYTHON process that that runs the ØMQ client. The line on the top of the graph represents the sum of both, BioTracker and PYTHON process to show the total memory occupation.

Figure 4.3 pictures the memory consumption of the BioTracker. Again, the results are not surprising: the BioTracker does not occupy much more memory when applying an algorithm in C++. This can be explained by the simple fact that the algorithm used is not using any additional local variables that would be big enough to factor much into the total memory consumption. The ØMQ tracking BioTracker however consumes much more memory as it must utilize both the BioTracker process as well as the ØMQ client process, which, in the context of this experiment, is the PYTHON process.

# Chapter 5

# Conclusions

## 5.1 Contributions

The contribution of this thesis is the separation of the monolithic BioTracker application into a Core framework and a GUI application, as Proposal 1 suggested. Proposal 2 is tackled as well as the BioTracker now is able to dynamically load C++ tracker modules. Another major contribution that also touches Proposal 2 is the ØMQ implementation for trackers that enables arbitrary tracking by any programming language that supports ØMQ. Furthermore, several issues regarding Proposal 3 where solved while working on this thesis, like moving the git repository to Github and providing a unified development environment via Docker images.

## 5.2 Discussion

Though there were major contributions made during this thesis there are still some open issues that need to be resolved in the future. As the API for the trackers changed to remove some technical debt acquired during earlier iterations of the BioTracker, many of the old trackers are not compatible with the new version of the application yet.

Another problem is that some features of the old BioTracker are not implemented in the framework yet, for example, data is not restored when trackers are switched out and in again, closing the application does not save the current state anymore and the application can only be built on Linux. The reason for this delay is that the redevelopment of large parts of the code was greatly underestimated. For example, the replacement of the OpenGL code took much more time than initially anticipated. Another factor that delayed progress was the python experimenting phase where, for several weeks, the python interpreter was embedded into the BioTracker. However, this was replaced by embedding ØMQ instead, as ØMQ promised to be more flexible when targeting multiple platforms.

Another bone of contention is the way of distributing the application as it does not consist

of a single monolithic file anymore but rather of a set of libraries that are used by a small GUI application. The ideas range from providing packages for LINUX distributions to installers to zipped containers that carry all libraries with them. related to this problem is the question of how to enable users to script the BIOTRACKER. As the current supported scripting environment is PYTHON it is imaginable to deliver the PYTHON libraries needed for a minimum of functionality. Another option would be to require users to maintain their scripting installation themselves which could be perceived as more complicated by some. A third option would be to provide the needed scripting environment in a potential installer where the user can select the environment appropriate for the current situation.

As it is not possible to serialize QT classes so they can be sent over ØMQ sockets, mirroring classes were developed that can be used on the ØMQ client side. However, this mirror classes are extremely incomplete and lack many functions that need to be implemented by hand. Furthermore, elements that require more user interaction require the BIOTRACKER-ØMQ-message protocol to be extended which involves both client as well as server side to be adapted. Another missing feature for ØMQ trackers is the serialization of trajectory data. The problem is that computer vision algorithms define their serializable data through inheritance which cannot be carried over ØMQ. An alternative solution could be to provide a general string serialization class so that the ØMQ clients could use data-formats like JSON. However, this could have significant influence on the performance of the tracker, depending on the number of tracked objects and would need further investigation.

## 5.3 Future Work

TORCH is a scientific computing framework for machine learning algorithms built around the scripting language LUA. A next step would be to provide a ØMQ interface for LUA so that users can choose between PYTHON and LUA for their tracker prototyping. However, with an increasing numbers of BIOTRACKER-ØMQ implementations, means to standardize those implementations are needed to ensure they do not diverge as well as to provide people, who want to build their own ØMQ implementations, with an easy tool to determine whether their implementation is correct or not. A tool can be imagined that starts any potential ØMQ client, executes a predefined set of ØMQ-BIOTRACKER functions and expects certain results to check if a certain implementation is valid or not[1].

Another interesting contribution would be a GUI integration for scripting so that users can modify, debug and execute their scripting trackers directly in the BIOTRACKER application. However, several questions would be needed to be answered as it must be very well defined how debugging and code highlighting should work. For debugging one could imagine an extension to the BIOTRACKER-ØMQ messages that carry additional debug information.

A server implementation for the BIOTRACKER would be a major step forward to enable

[1]This is inspired by the **autobahn** test suite: http://autobahn.ws/

automated image recognition. A tracker could be prototyped and fine-tuned with the BioTracker GUI application and then be deployed unsupervised onto a super computer. Related to this idea one can envision a webserver implemented around the BioTracker Core that serves users over a webbrowser. This could allow users to collaborate easier and would also eliminate the problems that revolve around installing the BioTracker at a users computer.

Last but not least, letting users easily combine different trackers would be a major contribution. This could be thought of as a pipeline where different trackers are stacked together and an image is passing through a tracker that generates output which will be used as input for the following tracker. This would greatly increase productivity as no program must be written nor must there be a deeper understanding of the underlying technologies.

# Bibliography


[1] Y. Wang, B. Georgescu, T. Chen, W. Wu, P. Wang, X. Lu, R. Ionasec, Y. Zheng, and D. Comaniciu, "Learning-based detection and tracking in medical imaging: a probabilistic approach," in *Deformation Models*. Springer, 2013, pp. 209–235.

[2] R. J. Radke, *Computer vision for visual effects*. Cambridge University Press, 2012.

[3] R. Vezzani, "Computer vision for people video surveillance," Ph.D. dissertation, Ph. D. Thesis, 2006.

[4] F. Wario, B. Wild, M. J. Couvillon, R. Rojas, and T. Landgraf, "Automatic methods for long-term tracking and the detection and decoding of communication dances in honeybees," *Frontiers in Ecology and Evolution*, vol. 3, p. 103, 2015.

[5] T. Landgraf and R. Rojas, "Tracking honey bee dances from sparse optical flow fields," 2007.

[6] T. von Falkenhausen, "Biotracker, a modular c++ framework for computer vision applications," 2015.

[7] D. Beymer, P. McLauchlan, B. Coifman, and J. Malik, "A real-time computer vision system for measuring traffic parameters," in *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*. IEEE, 1997, pp. 495–501.

[8] G. R. Bradski, "Computer vision face tracking for use in a perceptual user interface," 1998.

[9] K. Branson, A. A. Robie, J. Bender, P. Perona, and M. H. Dickinson, "High-throughput ethomics in large groups of drosophila," *Nature methods*, vol. 6, no. 6, pp. 451–457, 2009.

[10] T. Landgraf, D. Bierbach, H. Nguyen, N. Muggelberg, P. Romanczuk, and J. Krause, "Robofish: increased acceptance of interactive robotic fish with realistic eyes and natural motion patterns by live trinidadian guppies," *Bioinspiration & biomimetics*, vol. 11, no. 1, p. 015001, 2016.

[11] A. Prez-Escudero and G. G. de Polavieja, "Collective animal behavior from bayesian estimation and probability matching," *CoRR*, vol. abs/1105.1117, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1105.html#abs-1105-1117

[12] L. P. Noldus, A. J. Spink, and R. A. Tegelenbosch, "Ethovision: a versatile video tracking system for automation of behavioral experiments," *Behavior Research Methods, Instruments, & Computers*, vol. 33, no. 3, pp. 398–414, 2001.

[13] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of object-oriented programming*, vol. 1, no. 2, pp. 22–35, 1988.

[14] G. Bradski, *Dr. Dobb's Journal of Software Tools*.

[15] J. C. Simon and M. H. Dickinson, "A new chamber for studying the behavior of drosophila," *PLoS One*, vol. 5, no. 1, p. e8793, 2010.

[16] J. Schneider, M. H. Dickinson, and J. D. Levine, "Social structures depend on innate determinants and chemosensory processing in drosophila," *Proceedings of the National Academy of Sciences*, vol. 109, no. Supplement 2, pp. 17 174–17 179, 2012.

[17] *Interactive Feature Tracking using K-D Trees and Dynamic Programming*, vol. 1, 2006. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1640813

[18] F. Akgul, *ZeroMQ*. Packt Publishing, 2013.

[19] M. Wittig, "Biotracker architecture," 2016.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[21] I. Sobel, R. Duda, P. Hart, and J. Wiley, "Sobel-feldman operator."

[22] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1273442.1250746

[23] Ø. D. Trier, A. K. Jain, and T. Taxt, "Feature extraction methods for character recognition-a survey," *Pattern recognition*, vol. 29, no. 4, pp. 641–662, 1996.

[24] J. D. Evans, C. Saegerman, C. Mullin, E. Haubruge, B. K. Nguyen, M. Frazier, J. Frazier, D. Cox-Foster, Y. Chen, R. Underwood *et al.*, "Colony collapse disorder: a descriptive study," *PloS one*, vol. 4, no. 8, p. e6481, 2009.

[25] M. Stanghellini, J. Ambrose, and J. R. Schultheis, "The effects of honey bee and bumble bee pollination on fruit set and abortion of cucumber and watermelon," *American Bee Journal*, vol. 137, no. 5, pp. 386–391, 1997.

[26] D. Pham, A. Ghanbarzadeh, E. Koc, S. Otri, S. Rahim, and M. Zaidi, "The bees algorithm–a novel tool for complex optimisation," in *Intelligent Production Machines and Systems-2nd I* PROMS Virtual International Conference 3-14 July 2006*. Elsevier, 2011, p. 454.

[27] M. Dorigo and T. Stützle, *Ant Colony Optimization*. Scituate, MA, USA: Bradford Company, 2004.

[28] C. A. Schneider, W. S. Rasband, K. W. Eliceiri *et al.*, "Nih image to imagej: 25 years of image analysis," *Nat methods*, vol. 9, no. 7, pp. 671–675, 2012.