Technical Documentation

Documentation for developers

Co-funded by the
Erasmus+ Programme
of the European Union

COLLABORATING PARTNERS

CDV Centre pour le développement des compétences relatives à la vue	Centre pour le développement des compétences relatives à la vue 17a, route de Longwy L-8080 Bertrange Luxembourg	http://www.cc-cdv.lu
lega del filo d'oro	Lega del filo d'oro Sede centrale di Osimo Via Montecerno, 1 60027 Osimo (AN)	http://www.legadelfilodoro.it
aspaym castilla y león	Aspaym Castilla y Léon C/ Treviño 74 47008 - Valladolid	http://aspaymcyl.org
Politechnika Warszawska	Polytechnika Waszawska Plac Politechniki 1 00-661 Warszawa	http://www.pw.edu.pl
Center IRIS	Center Iris Langusova 8 1000 Ljubljana	http://www.center-iris.si

This project has been funded with support from the European Commission.

This publication reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Table of contents

Co-funded by the
Erasmus+ Programme
of the European Union

INTRODUCTION

During the last decades, technology has become one of the most important tools as a window to the world. Unfortunately, this window is not open for everyone and presents some accessibility barriers. Those barriers prevent people with different capacities from accessing the Internet and other digital contents. In order to overcome those limitations, there are some proposals, such as Eye Trackers (ET), but they are not specifically oriented towards visually impaired people (VIP), a collective with special needs which can take very big advantages from the audiovisual media.

In our attempt to facilitate the access to these contents for VIP, we've created a software library, in collaboration with experts from Luxembourg, Poland, Italy, Slovenia and Spain from different professional profiles, all of them related to the visual impairment, special education and enabling technologies. The requisites for the ET system were strongly committed to a low-cost approach, by using a webcam on the desktop which almost every user could afford to buy.

A software for gaze tracking has been developed. This software is based on video processing and deep learning. The software itself is meant to be used as an external library but, as it is open source, it can be included in every project and be used as a framework incase that suits better the needs of the development team. To test the library's capabilities, a magnifying application (eZoom) has been developed aside. This application uses the library to make an estimate on where the user is looking at, magnifying the contents which are being shown in that area. The magnified area is configurable in many ways, for instance, the dimensions, the colour palette and zooming level among others.

Following, a more specific guide for better understanding the development and how to use it in your own projects.

HARDWARE AND SOFTWARE DEPENDENCIES

Since the calculations are very demanding a higher end PC is needed to train the network. After this, running the program works fine on a laptop - as long as it has some kind of dedicated GPU.

Windows 10 is required. Furthermore a GPU which supports CUDA and a webcam with a resolution of 1080p is strongly advised.

ABSTRACT ARCHITECTURE OF THE PROJECT

The structure of the project folder is as follows. You can read up on them in the marked chapters.

- tools (See Advanced configuration – Tools.exe)
- magnifier (See Developer Guide - Showcase application)
- tracker (See Developer guide – Library)

Co-funded by the Erasmus+ Programme of the European Union

BUILDING OPEYE

In this section you will learn how to build OPEYE on Linux and Windows.

Note: The download links are provided at the end of this chapter.

For Linux

Note: If you're running Ubuntu make sure to grab the .deb packages. This will make the process easier.

To get started, you will first need to download dlib and link it to the opeye/dlib folder:

```
ln -s /path/to/dlib-19.13/ dlib
```

Then you'll need to download and install CUDA using the following commands:

```
sudo dpkg -i cuda-repo-ubuntu1710-9-2-local_9.2.88-1_amd64.deb
sudo apt-key add /var/cuda-repo-<version>/7fa2af80.pub
sudo apt-get update
sudo apt-get install cuda
```

Make sure the right NVIDIA driver is installed. You will notice if it's not since it will complain about unmet dependencies.

Afterwards download and install the libcudnn (CUDNN) runtime & development version. For this you will need a Nvidia account.

Finally, when all is set, you can build opeye using these commands:

```
mkdir build
cd build
cmake ..
```

For Windows

In case you'd like to build on Windows, you'll firstly need to download the Windows Development Env image. In some cases it might be sufficient to install the "Visual Studio Build Tools" on your computer.

Then download and install git. Afterwards install cmake. On the download page chose the msi version and make sure the check the option "*add cmake to the system path for all/the current users*" during the installation.

At this point you can prepare your project folder.

```
git clone <link to repo>
cd opeye
```



Co-funded by the
Erasmus+ Programme
of the European Union

Then you'll need to load the json lib:

```
cd lib
mv json json.old
git clone https://github.com/nlohmann/json.git json
```

Make sure that there is a src folder otherwise rename the include folder to src.

In case you have a GPU that supports CUDA, download and install CUDA. Running this project without a GPU that supports CUDA is advised against, since training the network will take considerably longer.

Next, download and extract dlib. DLib will be built together with the project. Incase you want to test DLib, you can build it using these instructions: http://dlib.net/compile.html. Link the latest dlib version to the src folder of the project or simply extract it to that location.

Afterwards download the latest opencv source release. You will need to build it yourself:

Extract it to *C:\opencv*, then build it:

```
mkdir build && cd build && cmake .. && cmake --build . --config Release
```

Download and install freeglut: (https://www.transmissionzero.co.uk/software/freeglut-devel/) extract it to *C:*. Then download and install qt. Use the online installer of the latest prebuild for *msvc 2017*.

Finally, now that everything is set in place you can run the following commands to build the project:

```
mkdir build
cd build
cmake ../src/ -DCMAKE_PREFIX_PATH="C;\\opencv\\build;C:\\freeglut;C:\Qt\5.11.0;C:\\Qt\\5.11.0\\msvc2017_x64 -A x64"
cmake --build .  --config Release
```

Download links

- Dlib: http://dlib.net/files/dlib-19.16.zip
- OpenCV: https://www.opencv.org/releases.html
- NVIDIA CUDA: https://developer.nvidia.com/cuda-downloads
- CUDNN: https://developer.nvidia.com/rdp/cudnn-download
- Window Development Env image: https://developer.microsoft.com/en-us/windows/downloads/virtual-machines
- Visual Studio Build Tools: https://visualstudio.microsoft.com/
- Git: https://git-scm.com/downloads
- Cmake (.msi): https://cmake.org/download/
- Freeglut: https://www.transmissionzero.co.uk/software/freeglut-devel/
- Qt: https://www.qt.io

Co-funded by the Erasmus+ Programme of the European Union

ADVANCED CONFIGURATION – TOOLS.EXE

In this chapter the different launch options for tools.exe will be explained. Tools consists of a main task (*main.cpp*) that captures arguments and parameters from the command line and executes these commands using the various tasks found in /tasks.

For this it relies on the tracker library detailed in the chapter Developer guide – Tracker Library.

BASICS

The usage of tools.exe requires DLIB (http://dlib.net/). A GPU (graphics processing unit) that supports CUDA (https://www.geforce.com/hardware/technology/cuda) is strongly advised since it speeds up the process immensely.

Launching tools.exe without any command options, or by adding *-h/--help* the following output will be displaying in the console:

Opeye Gaze estimation.

OPTIONS:

-h, --help	**Display this help menu**
-r	**Record samples**
-t	**Training the network**
-v	**Test the network**
-m	**Magnify with the network**
-p	**Pupil center**
-s	**Simplify**
-b	**Evaluate a network**
-I [image path]	**Path to the training images**
-n [network path]	**Path to the networks**
-l [levels]	**Depth of the neural network to work with**
-e[epochs]	**Number of epochs to train the neural network**
-x[resolution]	**Resolution of the window (i.e. 1080 or 2160)**

RECORDING SAMPLES

To record samples, launch *tools.exe -r*. This will ask the subject to look at various points on the monitor. The subject must look at the given square and press the spacebar to confirm. Tools.exe will capture the output of the webcam and add metadata such as the position of the marking.

Co-funded by the Erasmus+ Programme of the European Union

TRAINING THE NEURAL NETWORK

The option *-t* will train the network using the recorded data. The training of the neural network can be adjusted with two parameters, namely the options *-l* (Depth of the NN) and *-e* (Number of epochs to train the NN).

For example, if you plan to train **16+4+1 neural networks** for **30000 epochs:**

```
tools.exe -t -l 3 -e 30000
```

The complete network will be able to distinguish between **64 screen regions**.

Use **tools.exe -v -l x** to test the neural network. Where *x* corresponds to the level that should be evaluated at most. You can check the results using the option *-v*. Examples:

tools.exe -v -l 1 **->** 4 regions will be distinguished

tools.exe -v -l 2 **->** 16 regions will be distinguished

tools.exe -v -l 3 **->** 64 regions will be distinguished

More samples will lead to better results.

Launching *tools.exe -m -l x* will visualize smoothened results with calibration. Where *x* corresponds to the level that should be evaluated at most (e.g. 3).

To adjust the paths of the stored training images and the network use the options *-l*.

MAGNIFYING TOOL

Launching the magnifying sample application can be done using tools.exe *-m*. In case you are using a higher than 1080p monitor, adjust the resolution using *-x* – i.e.:

```
tools.exe -m -x 2160
```

To calibrate, glance at a displayed point and press the right mouse button. After having done this four times, these points will then be used to calibrate the glaze detection.

Note: The calibration only works if many levels have been evaluated.

Co-funded by the
Erasmus+ Programme
of the European Union

DEVELOPER GUIDE – TRACKER LIBRARY

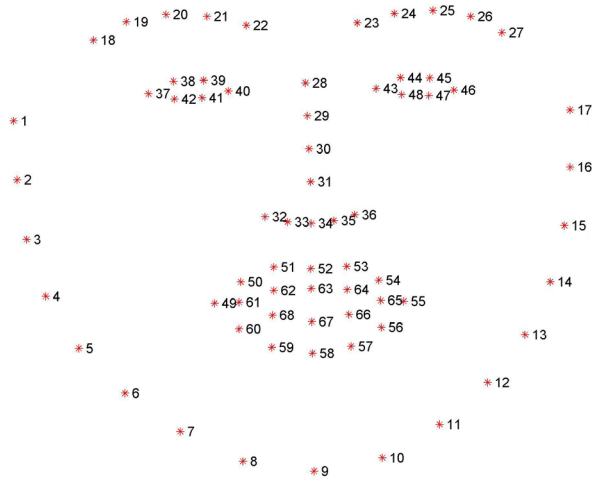
Opeye aims to use a person's gaze as media input source. A screen coordinate is computed based on a user's gaze. In a showcase application, which is part of this source code, the screen contents are magnified with respect to a user's gaze in order to help a person with viewing deficiencies.

Opeye's gaze estimation extracts the user's head & eyes portions of a webcam picture. Then feeds the eye part of the picture through a pyramid of neural networks to determine the precise position that is gazed at.

In detail, this takes advantage of OpenCV’s Haar cascade facial landmark detection algorithm. This allows to map a picture of a face to precise landmarks, including the eyes, as shown in the figure of the facial points.

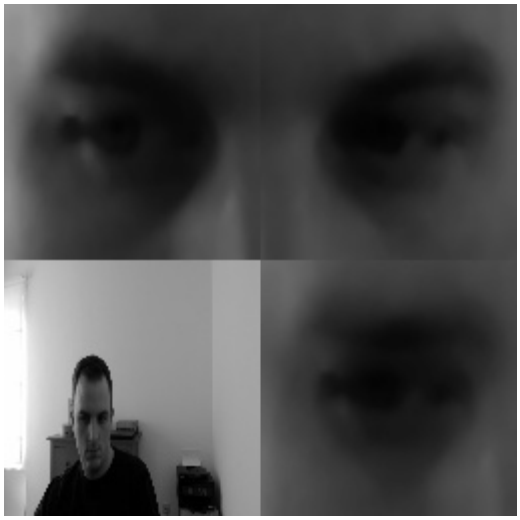
The image of both eyes is then extracted and composed to together to a smaller image that in a first step desaturated and then normalized. It serves serves as the input to the first neural network.

Co-funded by the Erasmus+ Programme of the European Union



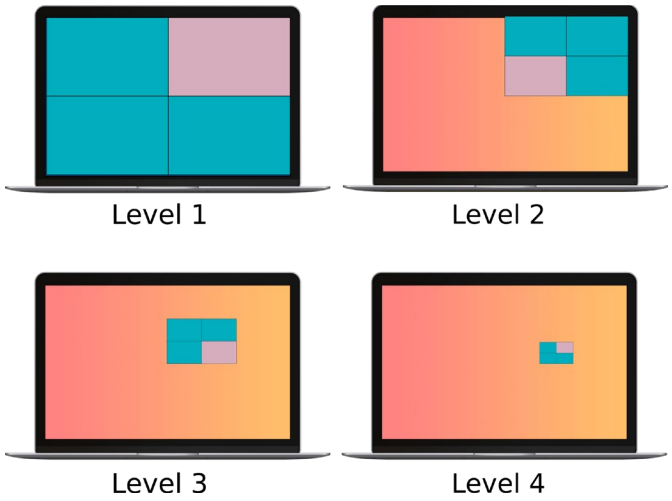
Note this image also has a small version of the whole input to provide the neural network with a position of the user.

The neural networks do form a pyramid. At the first level the whole screen is partitioned into 4



parts. Depending on the user's gaze, at this level, a single neural network determines the screen quadrant. The selected quadrant represents a next neural network that is specialized on that region and can be used to choose a sub quadrant again. This process can be repeated to achieve the desired precision. In the figure below, the rose square show the quadrant looked at for each level.

Co-funded by the
Erasmus+ Programme
of the European Union



The above example shows the simplicity and elegance of this system. Each of these neural networks is a small specialist, trained for a small portion of the screen. With four levels, a theoretical precision of (display dimension) / 2^4 can be achieved along either axis.

The tracker is the heart of the recognition part of the implementation. In this chapter the various components of the API are summarized. The library is divided into the following classes:

DATA

Calibration Class allowing a simple calibration. Feed it samples with “should” vs. “is” positions and it can apply a perspective distortion trying to eliminate unprecissions on various factors.

Library The library managing various samples with labels. The library also assembles the tree to teach the neural networks.

Network The core network library. It holds the different layers of neural networks.

Tracker The tracker computes the gazed at position based on a given frame.

It is the main library that can be used to teach and apply the neural networks as well as to serialize them. The tracker automatically loads the whole network tree and knows how to navigate through the different layers.

Types A sample is a simple picture with the position the subject gazed at during recording.

VIDEO

Frame Class that defines the “frame”

Co-funded by the Erasmus+ Programme of the European Union

Frameprocess Class responsible for processing the various frames. It makes use of dlib to detect a face in the frame.

Screen Screen is a helper class that enables text or shapes to be displayed on top of the visual output

Webcam Handles the video stream from the webcam

UTILS

/utils/.. reacts to keyboard and mouse inputs. Both contain public methods for update calls.

DEFINES.H

In this header-file various buttons, keys (mouse & keyboard) and structs are defined.

Additionally the dimension of the processor (i.e. input image size) are set to **256**. Larger image sizes yield better results but are very computationally intensive. One should try to stick with powers of two.

Furthermore, it defines the the structure of the neural network. The input is sampled with 2 convolutions of 5x5 with at a stride of 2x2. It is then fed to a fully connected 128 nodes layer.

```
dlib::relu<dlib::fc<128,
                    dlib::con<2, 5, 5, 2, 2,
                              SUBNET>>>;
```

ISSUES

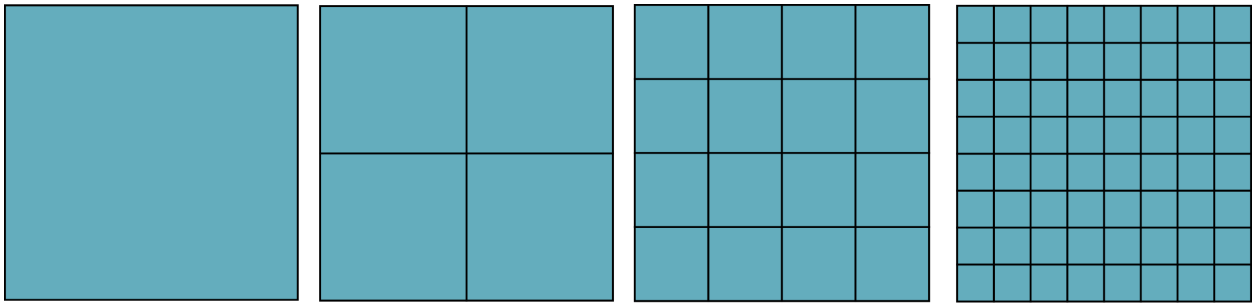
Due to the complexity of the project many issues were faced and some do remain.

The initial idea, after settling for a neural network based approach, was to train a huge neural network with many layers within, doing big convolutions, with many samples to result in a set of x,y coordinates. This idea appealed by its simplicity. Yet, while sometimes the results were impressive, it was impossible to achieve stable results if the circumstances changed (Position of the head, lighting, etc.). The reason being that the neural network was not able to generalize well. Migrating to an approach that has small very specialized networks with only 4 different output states allowed to train the first levels with much more data over many more epochs, yielding very good results that also generalized better to changing environmental parameters.

As shown in the figure of the neural networks below, one can see that there is a huge amount of neural networks necessary to be trained for this approach. 4 levels require the training of 85 networks. Each of these consume, at the configuration described above, approximately 16MB, summing up to about 1,5GB of memory required. This is certainly a remaining drawback of any neural network approach. Keep in mind that only a small portion of these neural networks (one per level) is actually used in the processed throughout the interpretation of a precise gaze; this

Co-funded by the Erasmus+ Programme of the European Union

tremendously saves computational power over a more generic “big single neuronal network” approach; as only a limited amount of memory has to be evaluated at processing.



Finally, besides the exploding memory requirements emerging from an addition of new levels, the precision, that can really be achieved is likely limited to the amount of label you can afford to collect and computation power you can invest. A single training of the 21 neural networks for 3 levels with approximately 10000 samples over 1000 epochs (iterations) using a modern graphics board (GTX 1080) takes 2 days. Changing a single parameter in the computation frequently requires the complete recomputation of said network.

While 10000 labeled samples appear to be many, this means that for the higher levels only a subset of the samples can be used. I.e. at level 3, only about 500 samples are available. These networks thus have less chance to achieve the same precision and genericity as the levels that were trained with more samples.

This leads to the conclusion that more computational resources, more development time and much more samples would yield better results.

Co-funded by the Erasmus+ Programme of the European Union

DEVELOPER GUIDE - SHOWCASE APPLICATION

In this chapter the showcase application “eZoom” is documented. This explanation serves as a practical example for a use case of the OPEYE library. The source code for the showcase application can be found in the folder *src/magnifier*. It is written in C++. For the Graphical User Interface Qt was used.

INPUT

The magnified area can either be moved by the mouse or using the gaze tracker. Using the interface *InputInterface/InputInterface.h* additional input like joystick could be programmed.

For the interface the following functions need to be implemented:

```
virtual QPoint getInputPosition() = 0;
virtual AdditionalInput getAdditionalSignal(QKeyEvent *event) const = 0;
virtual void stop(){};
```

Basic example

getAdditionalSignal() allows the triggering of the actions Close and ScreenShot.

```
    if (event->key() == Qt::Key_Escape)
    {
        return MagnifyingGlass::AdditionalInput::Close;
    }
    else if (event->key() == Qt::Key_P)
    {
        return MagnifyingGlass::AdditionalInput::ScreenShot;
    }
    return MagnifyingGlass::AdditionalInput::None;
```

In both implementations pressing *P* on the keyboard triggers a screenshot and *Esc* closes the window.

getInputPosition() requests the position of where the magnified area should be displayed.

This Input-Impementation gives an easy to follow example for the implementation of this code:

```
QPoint MagnifyingGlass::StandardInput::getInputPosition()
{
    QScreen *screen = QGuiApplication::primaryScreen();
    return QCursor::pos(screen);
}
```

It simply returns the position of the mouse on the primary monitor.

Co-funded by the Erasmus+ Programme of the European Union

Eye Tracker

The implementation of the eye tracker is understandably more complex. The values are requested by the Opeye-Tracker-Library (*tracker.h* and *webcam.h*).

```
QPoint MagnifyingGlass::EyeTrackerInput::getInputPosition()
{
    auto point = this->update();
    QScreen *screen = QGuiApplication::primaryScreen();
    return  QPoint(point.x()  *  screen->size().width(),  point.y()  *  screen-
>size().height());
}
```

The point received by *this->update()* is updated automatically. It contains two values (x, y) between 0 and 1. Before returning the result, the values need to be adjusted to the screen size.

Example: If the subject looks at the exact middle of the screen point would be (0.5, 0.5). The output (on a 1080p screen) would then be (960, 540).

this->update() calls *assertRunning()*, which in turn creates a thread that updates the point value continuously using values provided by the tracker library. Additionally, a LOCK is set to prevent racing conditions.

```
QPointF MagnifyingGlass::EyeTrackerInput::update()
{
    assertRunning();

    LOCK(updateLock);
    return lastPoint;
}
```

On the first cycle of the thread created in *this->update()* the tracker object is created.

```
tracker = TrackerPtr(new Tracker(levels, "network", "net_0"));
```

If the Tracker fails to be created with the given levels amount (levels = 4) the amount of levels is decreased, and the creation will then be re-tried.

After the tracker is ready data on the current tracking position is requested.

```
Rect r = tracker->track(lastFrame);
lastPoints.push_back(
      Point({r.min.x + r.dimension.w / 2, r.min.y + r.dimension.h / 2})
);
```

To prevent oscillation-effects, the values received from the tracker are damped:

Co-funded by the Erasmus+ Programme of the European Union

```
while (lastPoints.size() > DAMPING_SPAN)    // DAMPING_SPAN = 20
{
      lastPoints.pop_front();
}

double cnt = 1.0f;
double tot = 1.0f;
double x = 0.0f;
double y = 0.0f;
for (const Point &p : lastPoints)
{
      x += p.x * cnt;
      y += p.y * cnt;
      tot += cnt;
      cnt++;
}

x /= tot;
y /= tot;
```

After the point has been processed, lastPoint is updated and the location of the magnification window on the screen is adjusted.

MAGNIFICATION

The implementation of the magnification can be found in src/magnifier. Qt5 was used for the GUI. The actual magnification and color correction are realized using OpenGL and the following shaders:

- **colorConvert.frag** fragment shader responsible for changing colors of magnified area
- **magnifier.frag and .vert** fragment and vertex shader which enlarges an area
- **passthrough.vert** vertex shader which enables passthrough visuals

Co-funded by the Erasmus+ Programme of the European Union

SETTINGS AND CONFIGURATION

As the names imply, *Settings.json* and *SettingsOperator* handle the various options that can be configured. *SettingsOperator* contains the methods for reading and writing the settings from/to the given configuration path.

```
private:
    const std::string settingsPath;
public:
    SettingsOperator(const std::string &settingsPath);

    void readSettings();
    void writeSettings();
```

The following values are being tracked:

```
public:
    int widthPercent;
    int heightPercent;
    float zoomLevel;
    int hueRotation;
    float sharpenAmount;
    QString screenShotDirectory;
    QMap<QString, QVariant> inputMethods;
};
```

The default values can be adjusted in Settings.json:

```
{
    "widthPercent": 20,
    "heightPercent": 20,
    "zoomLevel": 1.0,
    "screenShotDirectory": "screenshots",
    "hueRotation": 0,
    "sharpenAmount": 0,
    "inputMethods": {
        "EyeTracker input": "EyeTrackerInput",
        "Standard input [mouse and keyboard]": "StandardInput"
    }
}
```

NOTES

We have proven that we can interpret a person's gaze and compute real time screen coordinates based on a webcam image.


Co-funded by the
Erasmus+ Programme
of the European Union

The resulting success is limited by the computing power / resources available. In theory more memory, more GPU power and more labeled samples should yield better results.

Opeye was realized within the context of an Erasmus+ project. http://opeye.eu

Co-funded by the Erasmus+ Programme of the European Union