



EÖTVÖS LORÁND UNIVERSITY  
FACULTY OF INFORMATICS  
DEPARTMENT OF SOFTWARE  
TECHNOLOGY AND METHODOLOGY

---

# CURSOR MOVEMENT VIA HEAD POSITION ESTIMATION ON WINDOWS SYSTEMS

Supervisor:

**Tőser, Zoltán**

assistant research fellow

Department of Software Technology  
and Methodology

Written by:

**Fazekas, Bálint**

Bachelor's of Computer Science

Specilaization of Software Application

Budapest, 2017

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>User documentation</b>	<b>4</b>
<b>3</b>	<b>Technical documentation</b>	<b>21</b>
3.1	Algorithmic background . . . . .	21
3.1.1	Face recognition . . . . .	21
3.1.2	Coordinate systems . . . . .	22
3.1.3	Algorithm . . . . .	25
3.2	Architecture . . . . .	32
3.2.1	Static architecture . . . . .	32
3.2.2	Dynamic workflow . . . . .	38
3.2.3	Implementation details . . . . .	39
3.2.4	Testing . . . . .	65
	<b>Bibliography</b>	<b>73</b>

# Chapter 1

## Introduction

For many decades, countless of researches and attempts were made in the technological field of science to produce tangible virtual reality – the human *in vivo* experience in a digitally or virtually created environment. One of the earliest of these attempts was the creation of Morton Heilig’s *Sensorama* (built in 1962[1]), which he described as “The future of cinema”[2].

Virtual reality is defined as “an artificial environment which is experienced through sensory stimuli (such as sights and sounds) provided by a computer and in which one’s actions partially determine what happens in the environment”[3]. There are several approaches for creating an environment which can be defined as *virtual reality*, out of which many lie on a scale of how close the hardware should be to the user. With the quickly developing technological world and the ever increasing performance of computers, virtual reality has developed to a level of mass demand. Devices, such as the Wii<sup>(TM)</sup> and the PlayStation4<sup>(TM)</sup> both include motion sensing hardware, while Microsoft’s Kinect<sup>(TM)</sup> includes body gesture tracking. In the first case, the user is required to equip or wear certain hardware, which captures and translates the motions of the user to numerical data. The data is then passed on to the computing device that analyzes the data appropriately to simulate the user’s movements. In the latter case, the user is only required to stand in the field of view of one or several cameras. The camera detects the user on its input image, and analyzes the gestures of the user. The gestures are then interpreted by the computing device, that uses the data appropriately for the actually running application.

For convenient usage, both cases require real-time motion or gesture tracking.

Exploiting the fact that recently most portable electronic devices include a built-in camera, this thesis focuses on a possibility of using them for gesture tracking. More precisely: an attempt of creating a software, which uses a single camera to detect a human face and allows the user to control the position of the mouse cursor of their device, based on head position.

The problem of head position estimation is essentially the problem of object detection, which is a topic of *computer vision*. Detecting a three-dimensional object has several methods. For example, a common technique in the digital entertainment media industry is to use several cameras to capture an object from different angles. Using trigonometric functions on the different views, the position and the size of the object can be determined.

Another approach is the Perspective-N-Points (PNP) algorithm. The purpose of the PNP algorithm “is to determine the position and orientation of the camera with respect to a scene object from  $n$  correspondent points”[4, p. 2]. The number of points used by the algorithm is denoted by  $n$ .

The number of corresponding points in this thesis is chosen to be 3 – ( $n = 3$ ) – for the PNP problem. By using experimentally obtained data, this thesis further simplifies the PNP problem to achieve three-dimensional head pose estimation, in order to bypass many calculations in the PNP algorithm. The planning and the implementation of the project was inspired by the articles of **iplimage**[5] and **Nghia Ho**[6].

# Chapter 2

## User documentation

QuteCursor is an easy-to-use, portable application, which enables the user to control the mouse cursor of their Windows 10 (or above) system.

### **System Requirements:**

- Windows 10 or greater
- At least 200Mb of RAM
- At least 300Mb of Disk space
- A built-in and/or attached webcam, that Windows recognizes as a webcam
- At least 645 by 460 screen resolution

The QuteCursor application is compressed in a QuteCursor.zip folder. This folder includes every piece of software that is necessary to run the QuteCursor application.

### **Installation:**

As mentioned above, QuteCursor is a portable application, meaning that by itself it does not need to be installed, only ran. However, it is required for the 'Visual C++ Redistributable for Visual Studio 2015' program to be installed. This program ensures that QuteCursor runs without any problems. The installation packages for the Visual C++ Redistributable program can be found inside the QuteCursor.zip compressed folder, or it can be downloaded from <https://www.microsoft.com/en-us/download/details.aspx?id=48145>.

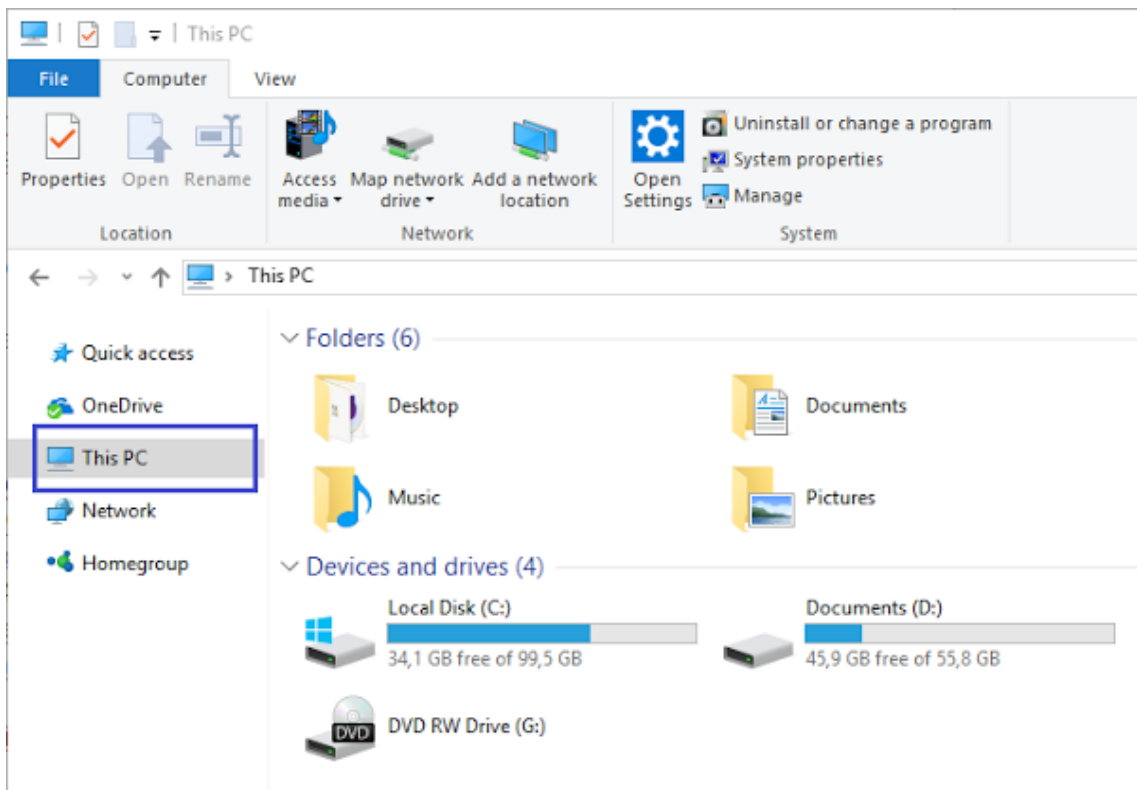
Both x86 and x64 Visual C++ Redistributable installers are included in the .zip folder. (Note: If You have a 32 bit system, use the vc\_redist.x86.exe installer. If You have a 64 bit system, use the vc\_redist.x64.exe installer.)

### **How can I check if my computer is eligible for running the QuteCursor application?**

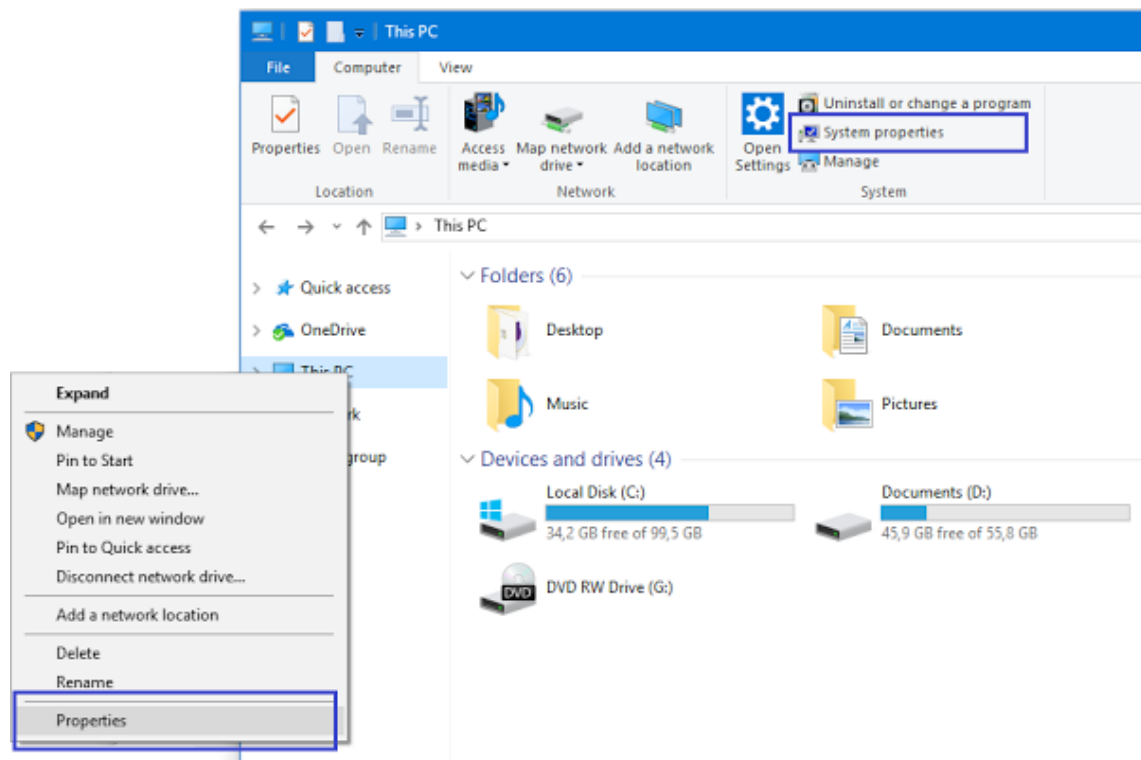
To check Your system version and amount of RAM installed in Your system: Open File Explorer.



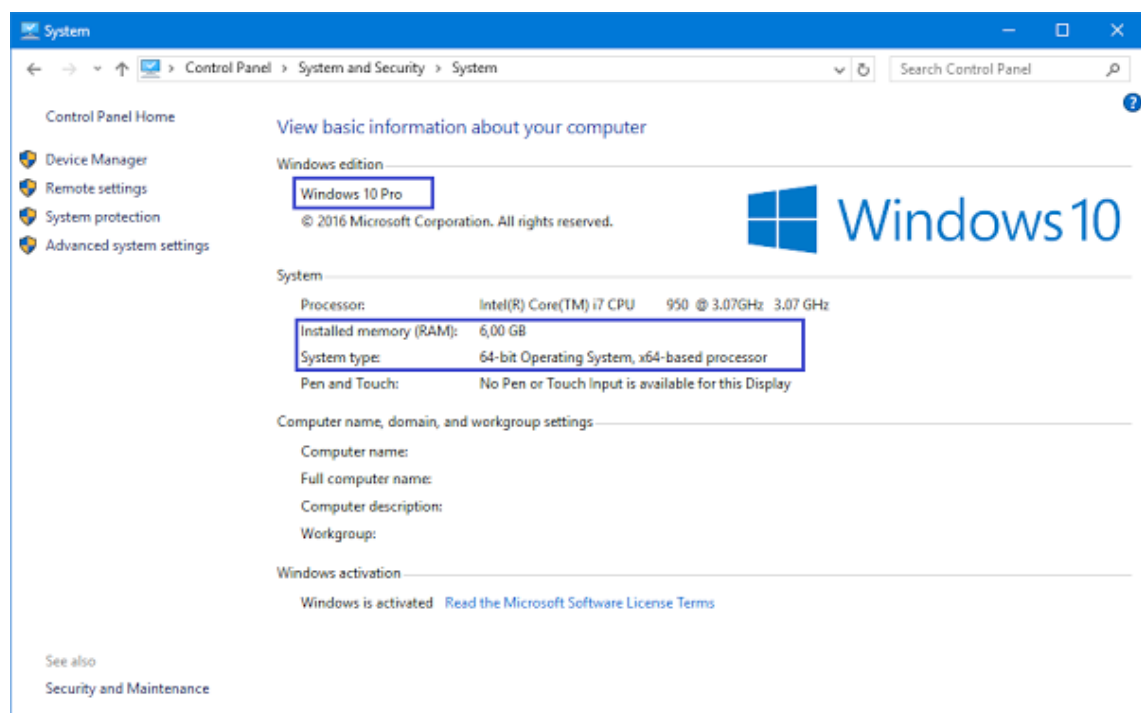
You will be able to see how much space You have on Your hard drive(s). (1Gb is about 1000 Mb.) Right-click on 'This PC'.



Click on 'Properties' or select 'System Properties'.



You should be able to see Your computer's properties.

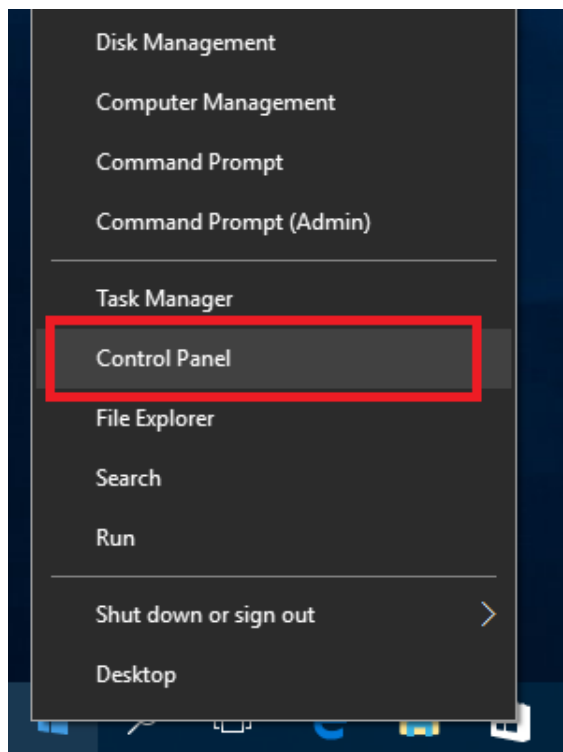


Checking if You have 'Visual Studio C++ Redistributable for Visual Studio 2015' installed

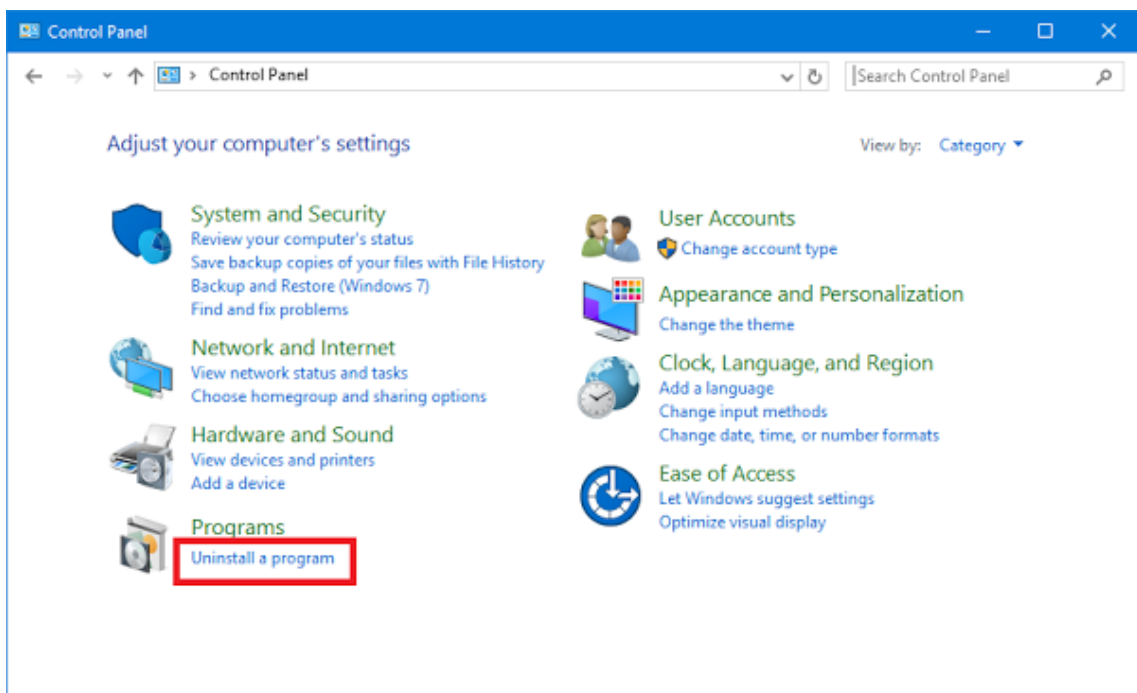
Right-click on the Windows icon, also known as the 'start button'.



Select 'Control Panel'.

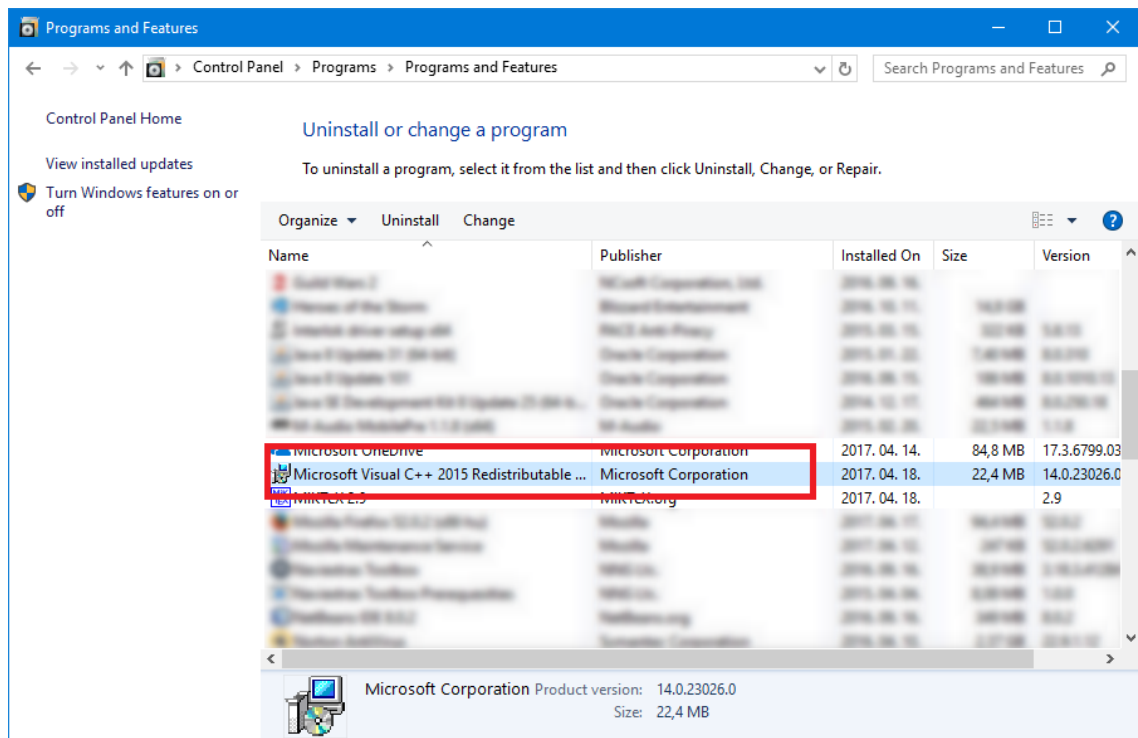


Go to 'Programs' → 'Uninstall a program'



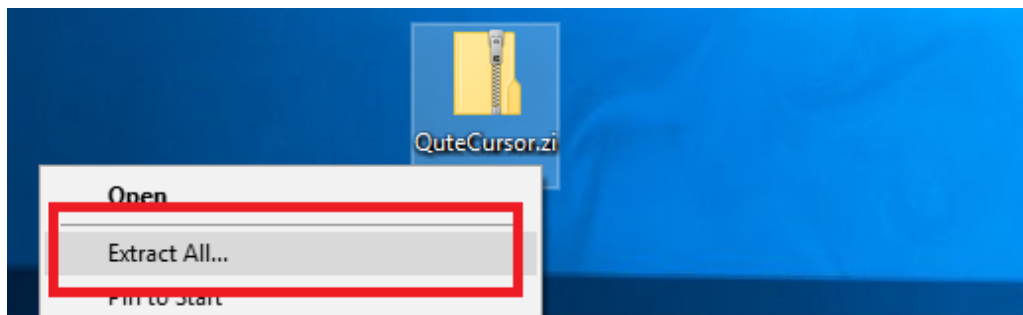
You can see the list of software installed on Your computer. If You cannot find 'Microsoft Visual C++ 2015 Redistributable...' in this list, then You do not have it installed on Your system, and You need to install it.



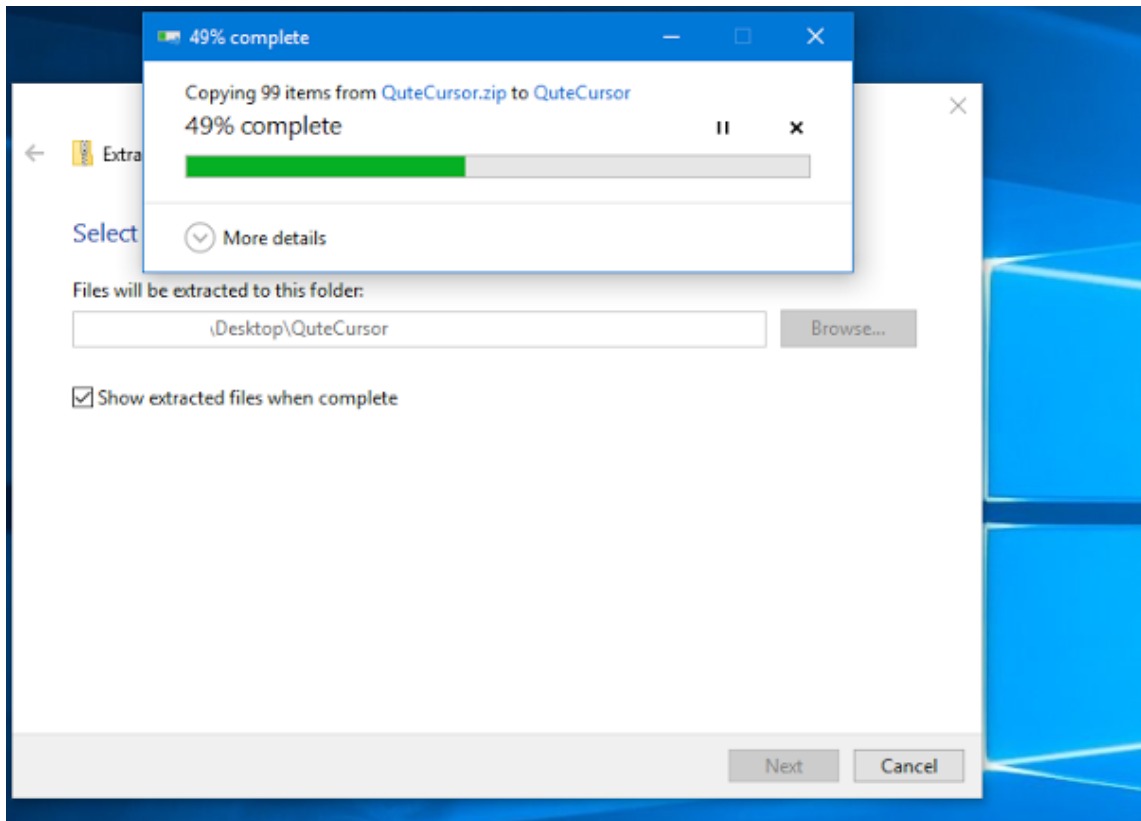


**Installing ‘Visual Studio C++ Redistributable for Visual Studio 2015’**  
(skip these instructions, if You already have it installed):

Unzip QuteCursor.zip somewhere on Your device. To do this, right-click on the zip folder, and select ‘Extract all...’



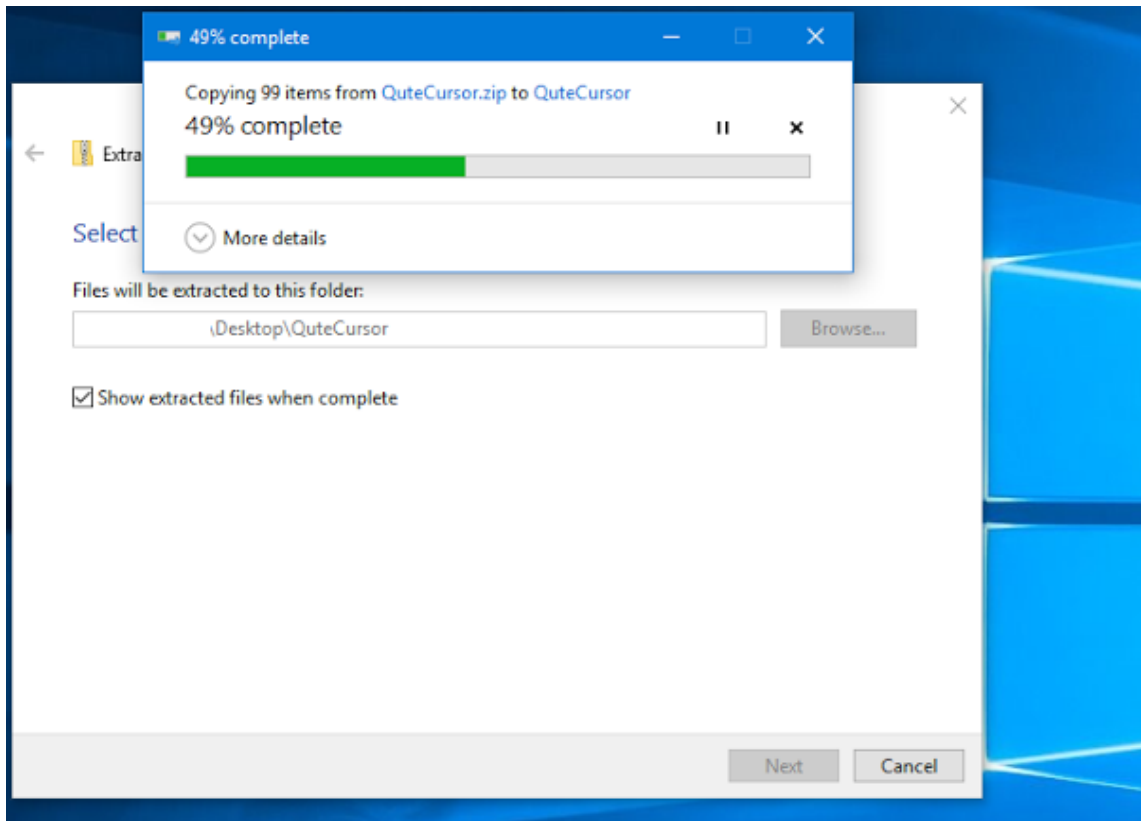
Once it is done extracting, double-click on either vc\_redist.x64.exe or vc\_redist.x86.exe, depending on the type of system You have. Note: If You do not trust these executable files, You could download them officially from <https://www.microsoft.com/en-us/download/details.aspx?id=48145>. Click on ‘Install’ to install the software.



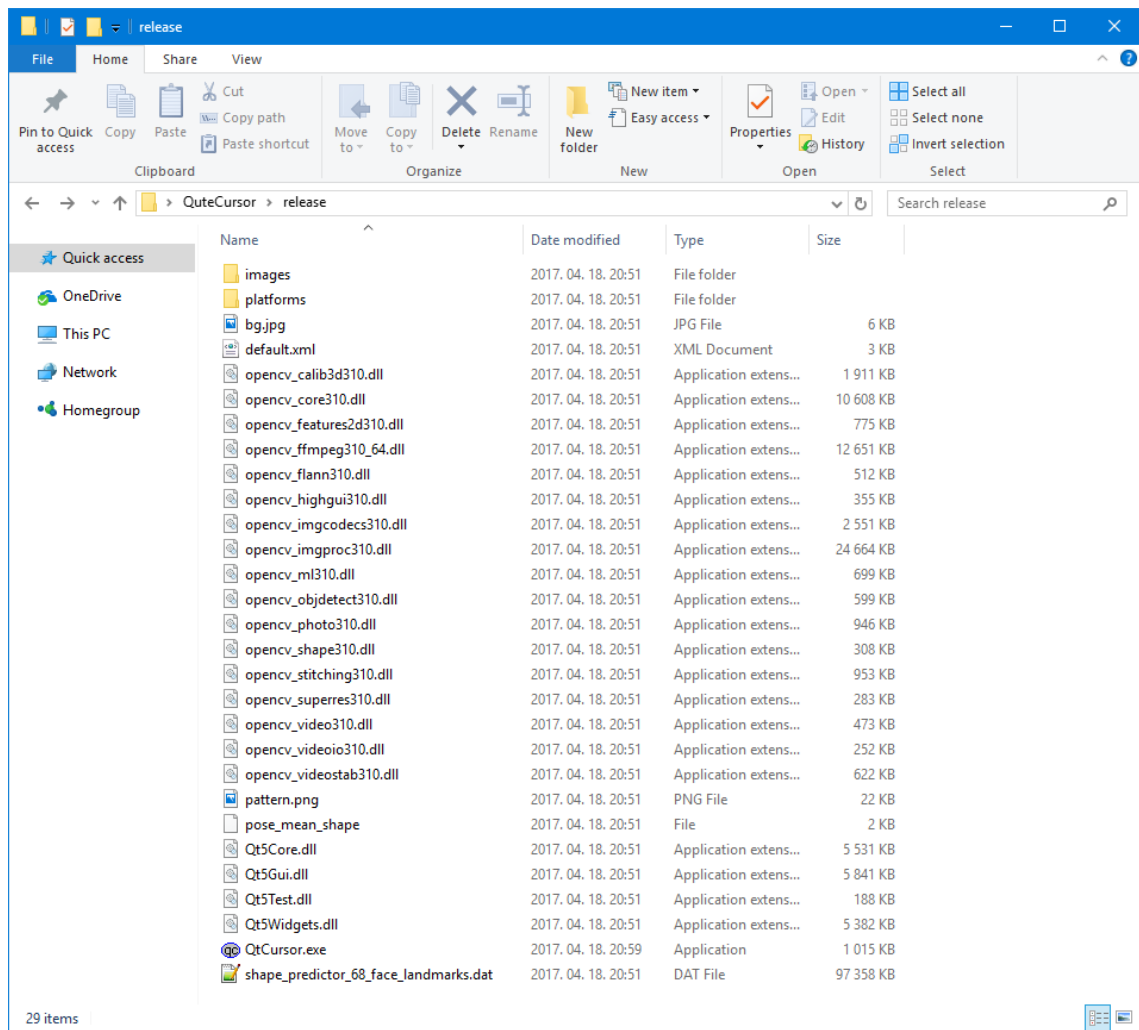
### **Opening QuteCursor for the first time:**

Before You unzip QuteCursor.zip, please note that any Anti-Virus programs will scan the QuteCursor folder contents, and will most probably mark them as 'virus' or 'dangerous malware', and therefore remove them upon extraction. If You wish to use QuteCursor, exclude QuteCursor.zip to be scanned by Your anti-virus program, or turn off Your anti-virus program for a short amount of time. (Since QuteCursor was developed as a personal project, Windows is right to be suspicious of this software. However, QuteCursor only sees the contents of its directory, and it does not affect the rest of Your system.)

Unzip QuteCursor.zip anywhere on Your device.



After it is done extracting, open the extracted QuteCursor folder. You should see two executable files (for installing Visual C++ Redistributable thoroughly detailed in the previous section), and a folder called 'release'. Double-click on 'release'. You should be able to see a great number of contents.



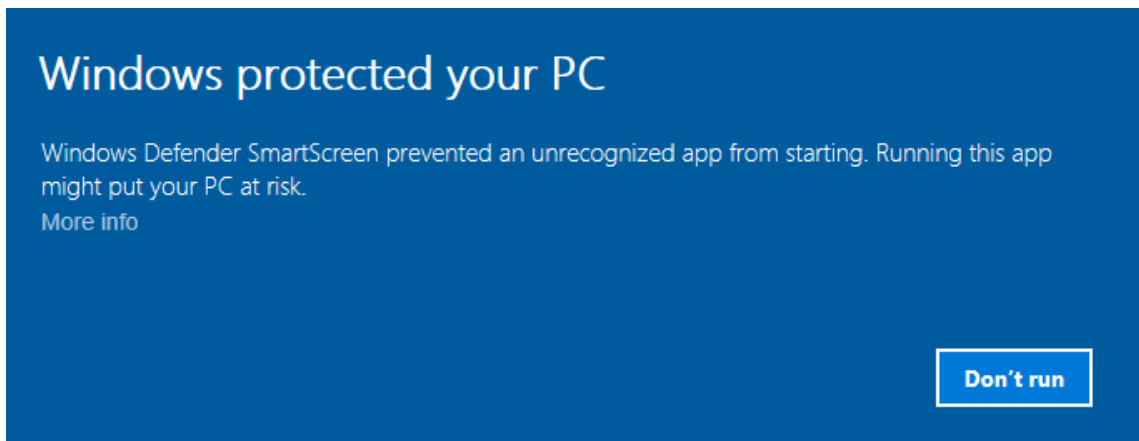
These are the minimum required files for You to be able to run QuteCursor, so make sure You do NOT rename, delete, and / or change the files presented in this folder. The files are listed below:

- ☐ QuteCursor
  - ☐ vc\_redist.x86.exe
  - ☐ vc\_redist.x64.exe
  - ☐ release
    - ☐ bg.jpg
    - ☐ default.xml
    - ☐ opencv\_calib3d310.dll
    - ☐ opencv\_core310.dll
    - ☐ opencv\_features2d310.dll

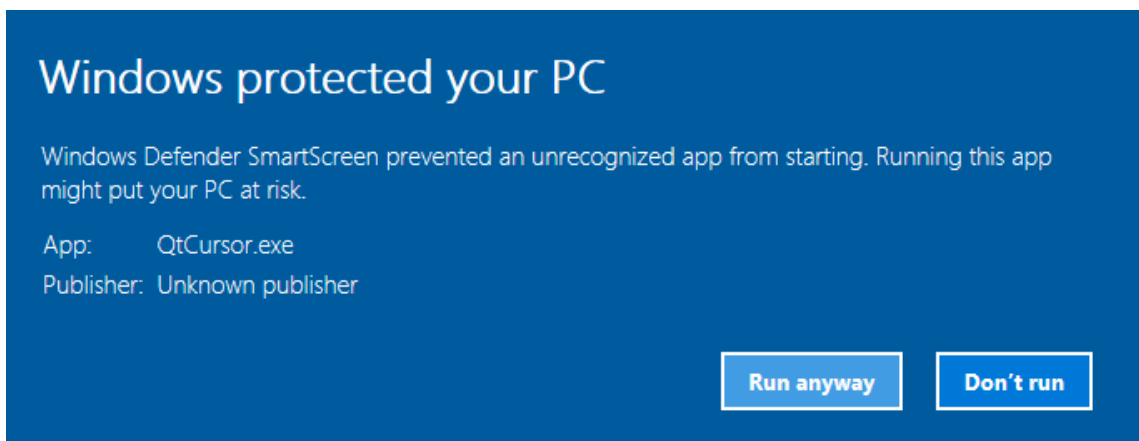
- ☐ opencv\_ffmpeg310\_64.dll
- ☐ opencv\_flann310.dll
- ☐ opencv\_highgui310.dll
- ☐ opencv\_imgcodecs310.dll
- ☐ opencv\_imgproc310.dll
- ☐ opencv\_ml310.dll
- ☐ opencv\_objdetect310.dll
- ☐ opencv\_photo310.dll
- ☐ opencv\_shape310.dll
- ☐ opencv\_stitching310.dll
- ☐ opencv\_superres310.dll
- ☐ opencv\_video310.dll
- ☐ opencv\_videoio310.dll
- ☐ pattern.png
- ☐ opencv\_videostab310.dll
- ☐ pose\_mean\_shape
- ☐ Qt5Core.dll
- ☐ Qt5Gui.dll
- ☐ Qt5Test.dll
- ☐ Qt5Widgets.dll
- ☐ QtCursor.exe
- ☐ shape\_predictor\_68\_face\_landmarks.dat
- ☐ images
  - ☐ (this folder contains a few example images for the camera calibration)
- ☐ platforms
  - ☐ qdirect2d.dll
  - ☐ qdirect2dd.dll
  - ☐ qdirect2dd.pdb
  - ☐ qminimal.dll

- ☐ qminimald.dll
- ☐ qminimald.pdb
- ☐ qoffscreen.dll
- ☐ qoffscreenend.dll
- ☐ qoffscreenend.pdb
- ☐ qwindows.dll
- ☐ qwindowasd.dll
- ☐ qwindowasd.pdb

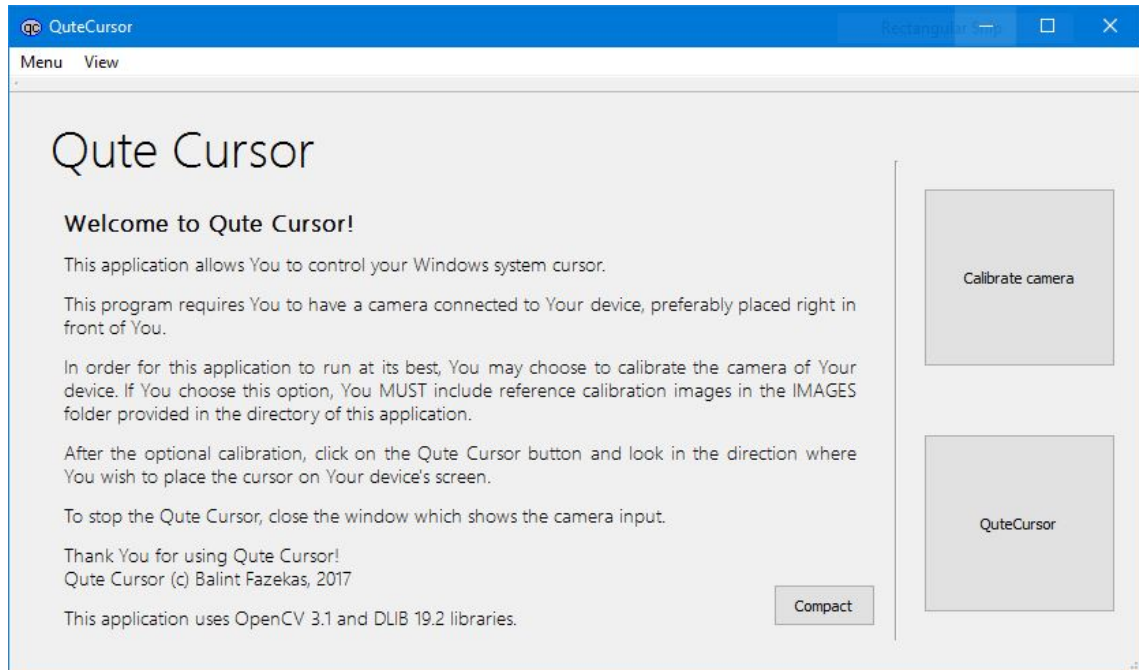
To run QuteCursor, double-click on 'QtCursor.exe' in this folder. If You have Windows Defender enabled on Your system, You may come across the following warning message:



(Since QuteCursor was developed as a personal project, Windows is right to be suspicious of this software. However, QuteCursor only sees the contents of its directory, and it does not affect the rest of Your system.) Click on 'More info' and 'Run anyway' to run QuteCursor.



If You have installed Visual C++ Redistributable for Visual Studio 2015 properly, then You should be able to see the following window:



### Using QuteCursor:

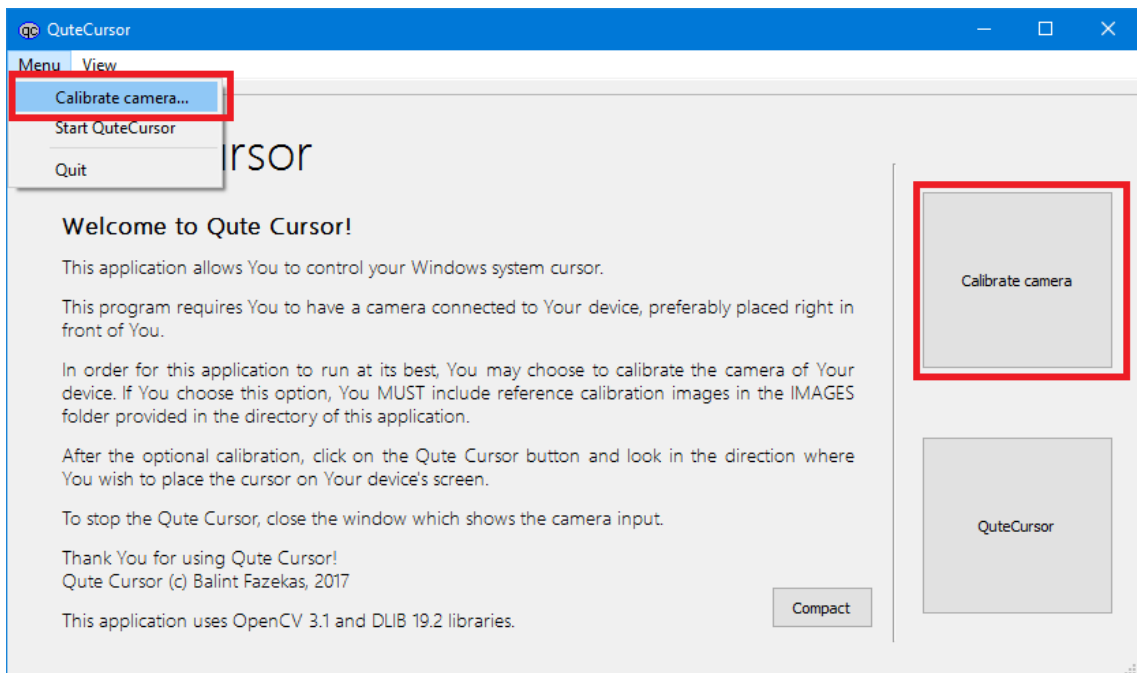
Once You have successfully opened QuteCursor, You will be able to choose between two options.

- ☐ Calibrate camera
- ☐ QuteCursor

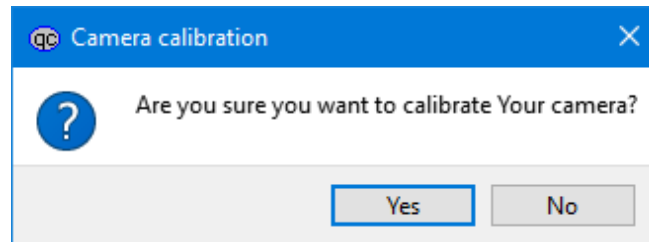
### Camera Calibration:

Before You start using QuteCursor, it is advised that You calibrate the camera of Your device. In order to do so, You need a physical print of the 'pattern.png' image included in the folder. The image should be printed as-is on an A4 paper. After You have a physical copy, attach it onto a flat surface, and take several photos of You holding it up in different positions. Example photos can be found in the *images* folder. It is important that You take the pictures *with the device You wish to use QuteCursor on!* After You are done taking the photos, You can delete all the example images from the images folder, and replace them with Your photos. The more photos, the better, but the *calibration works with the minimum of two valid photos added in the images folder.*

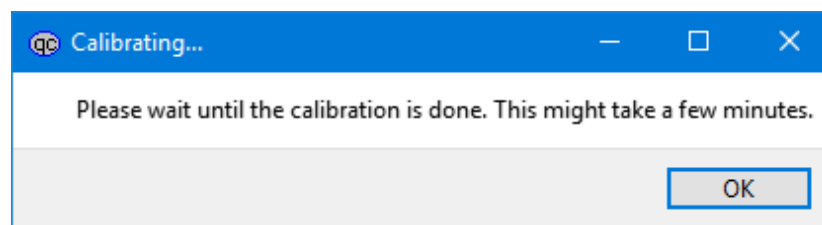
Once You have Your photos included in the proper folder, You can click on the *Calibrate camera* button, or choose *Calibrate camera...* from the Menu.



QuteCursor will ask for Your confirmation to calibrate Your camera. Choose 'Yes' to proceed, or click 'No' to cancel the calibration procedure.

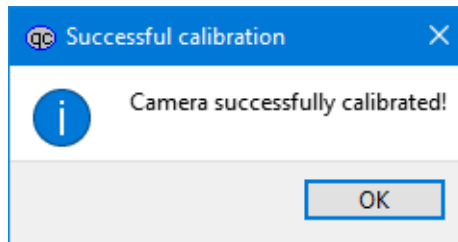


Note that You will not be able to interact with the main window of the application while the confirmation window is still open, and it will remain in the foreground of the application. During calibration, the following message will appear:



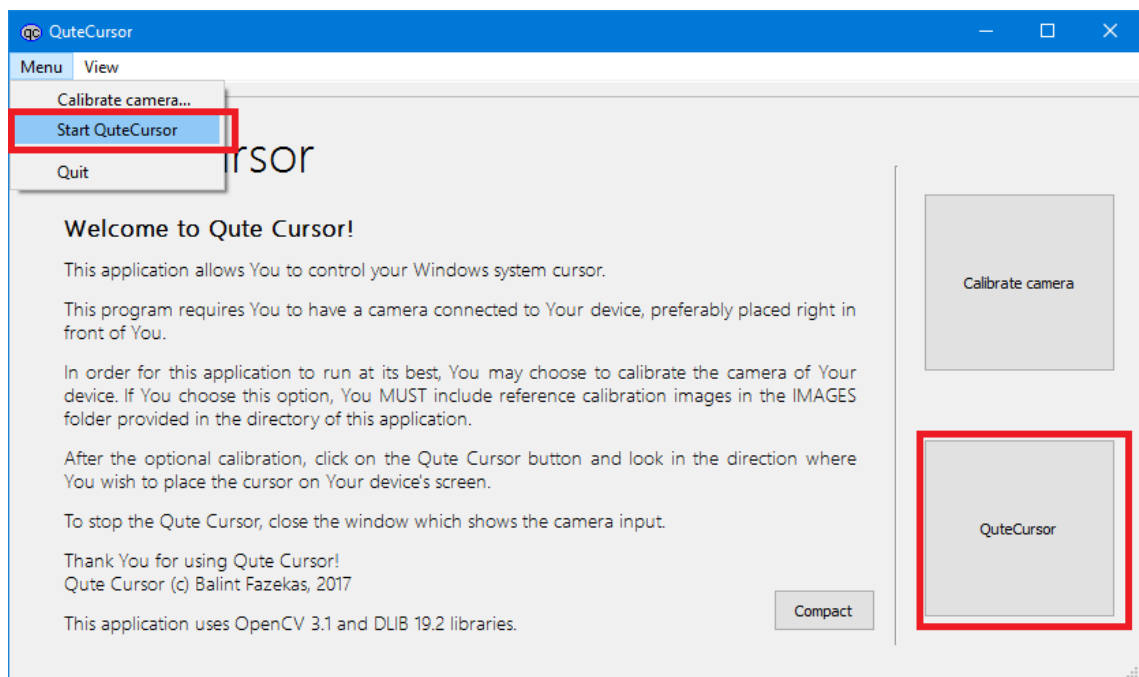
The calibration procedure will scan through the images in the *images* folder, and based on the photos it finds inside, it will return customized calibration values to use for Your camera. You will be notified with the following message, after QuteCursor is done calibrating:



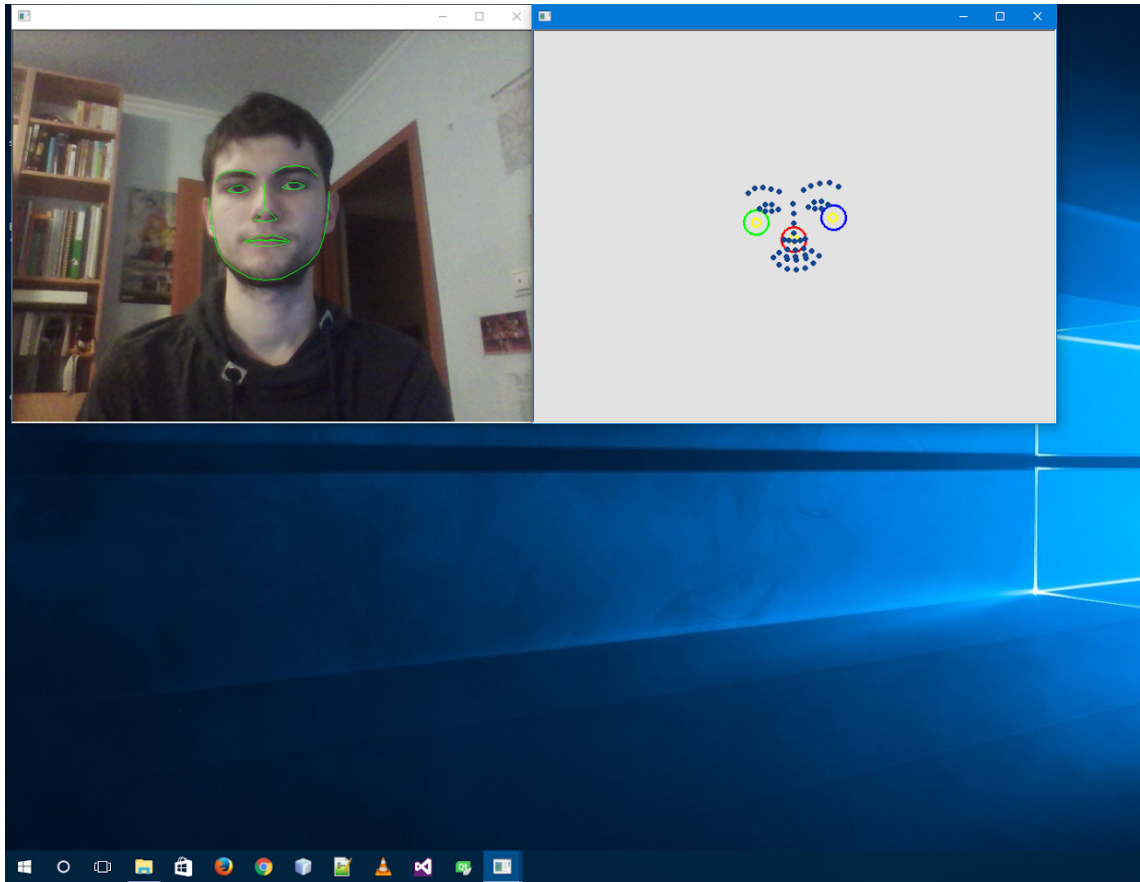


### QuteCursor:

To start controlling the cursor with Your head, click on 'QuteCursor' or choose 'Start QuteCursor' from the Menu.



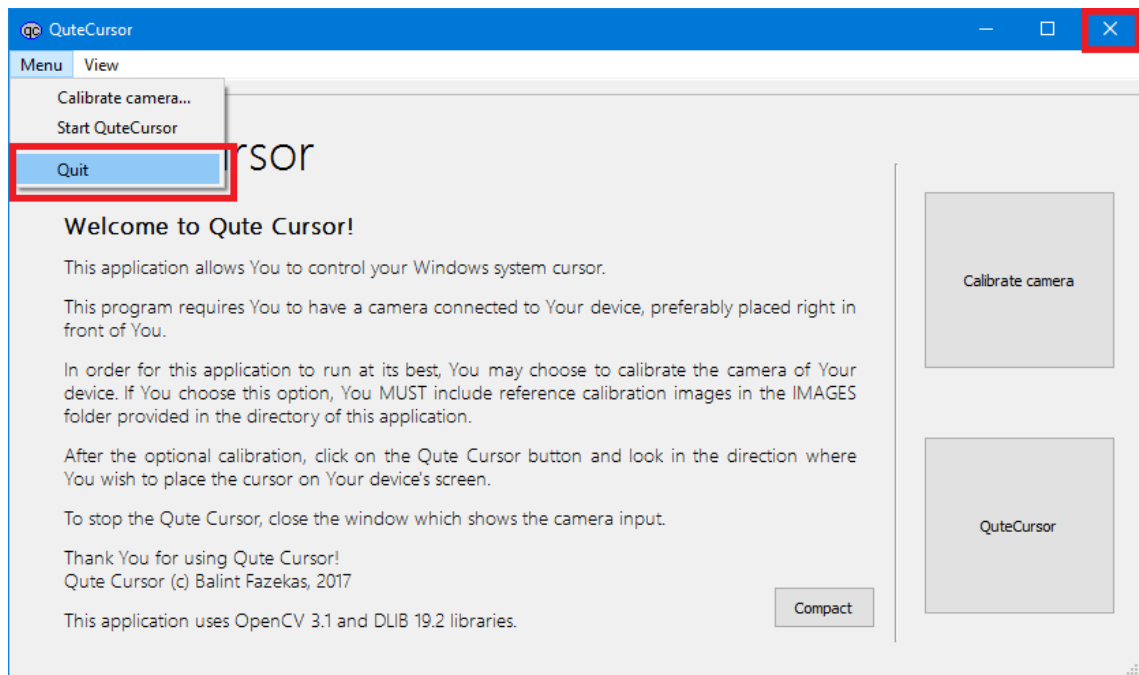
You will see two windows opening. The left window shows the camera input with the outlined recognized face. The right window shows the direction which the recognized face looks at – represented as a point mesh, that resembles a human face.



While the QuteCursor control is active, You will not be able to interact with, or see the main window of the application. In order to stop the QuteCursor control, close the window with the camera input. (Note: It is quite hard to control the cursor with the a mouse if QuteCursor is able to detect a face with the camera. If You wish to close the window with a mouse, it is the easiest to cover the camera so it does not detect a face.) With QuteCursor, You are able to hold down the left mouse button by slightly tilting Your head either left or right. To release the left mouse button, simply put Your head back in a straight position.

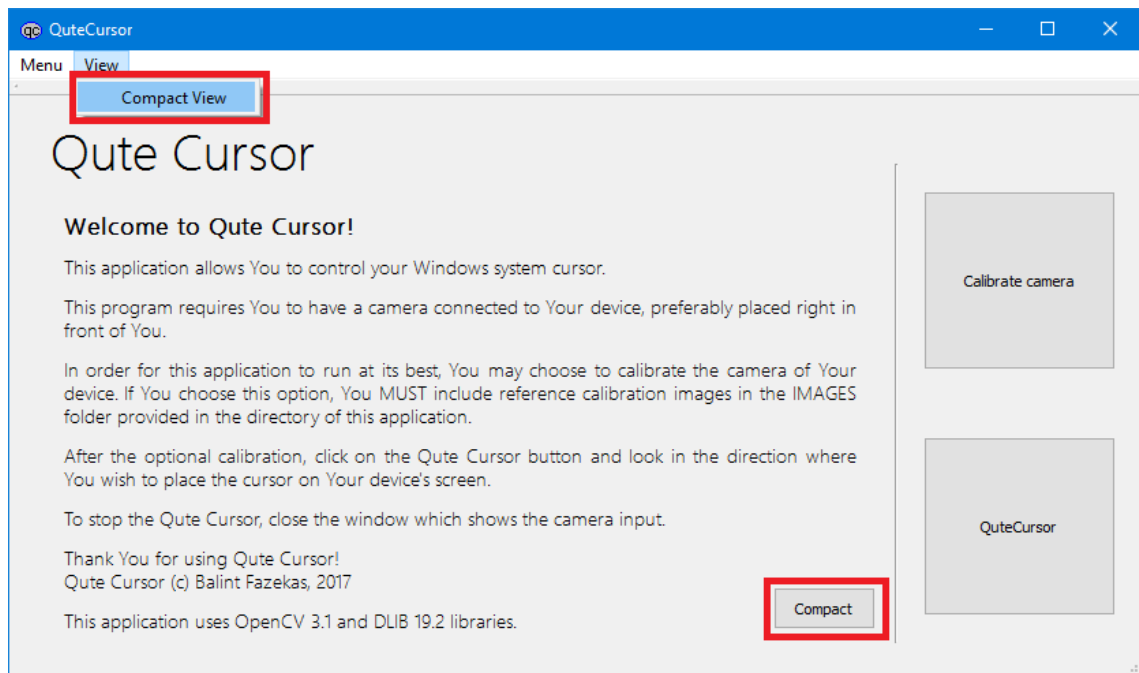
### **Closing QuteCursor:**

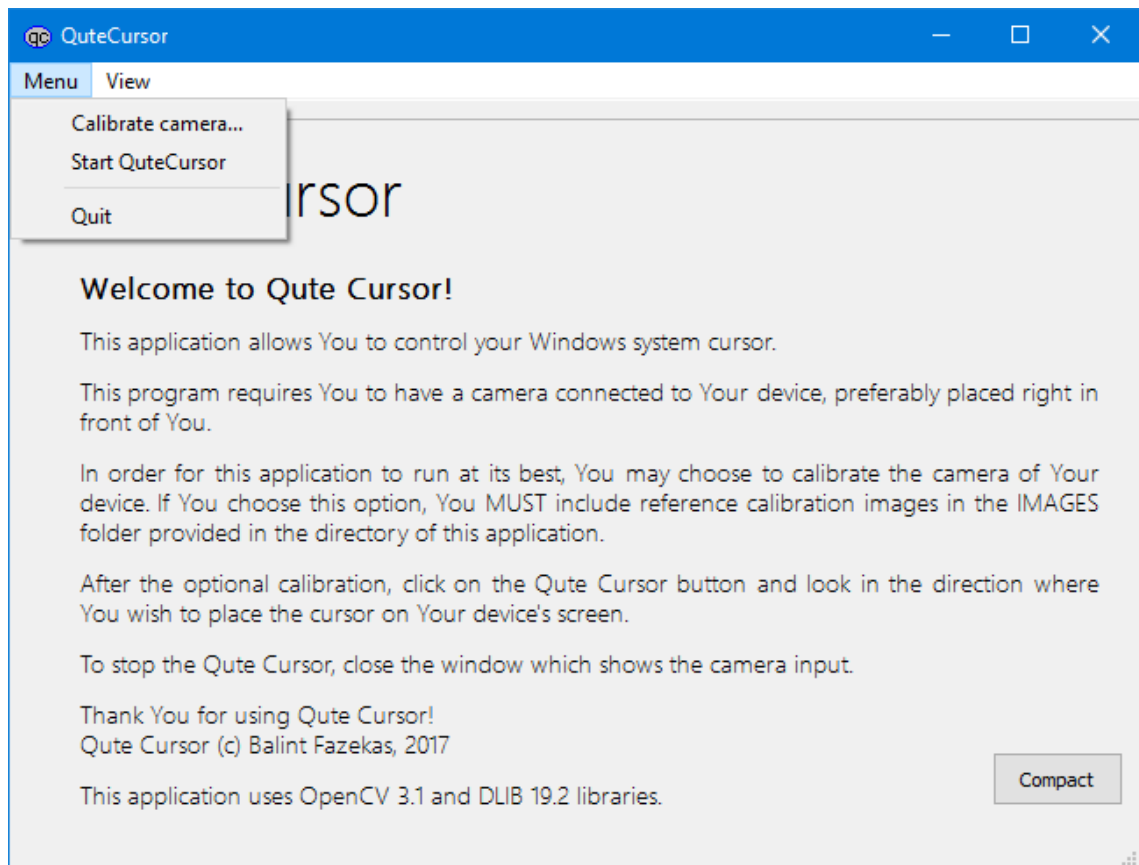
To quit the application, click the red '×' button on the top right corner of the main window, or choose 'Quit' from the Menu.



## Changing the view:

The application allows to resize the main window by clicking of the 'Compact' button, or selecting 'Compact View' from the 'View' menu. This will disable the two big buttons, and the functions of the application will only be available for use from the 'Menu'.



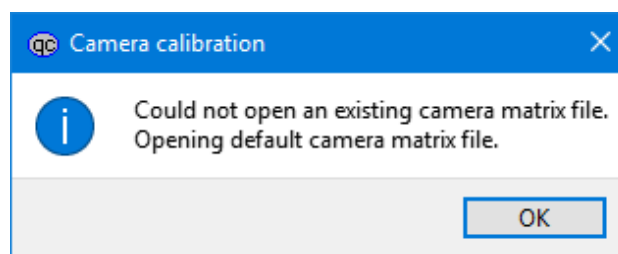


### Possible error messages:

Note, that only *one* application should have access to the camera at a time on the given device, otherwise the application will not recognize the camera.

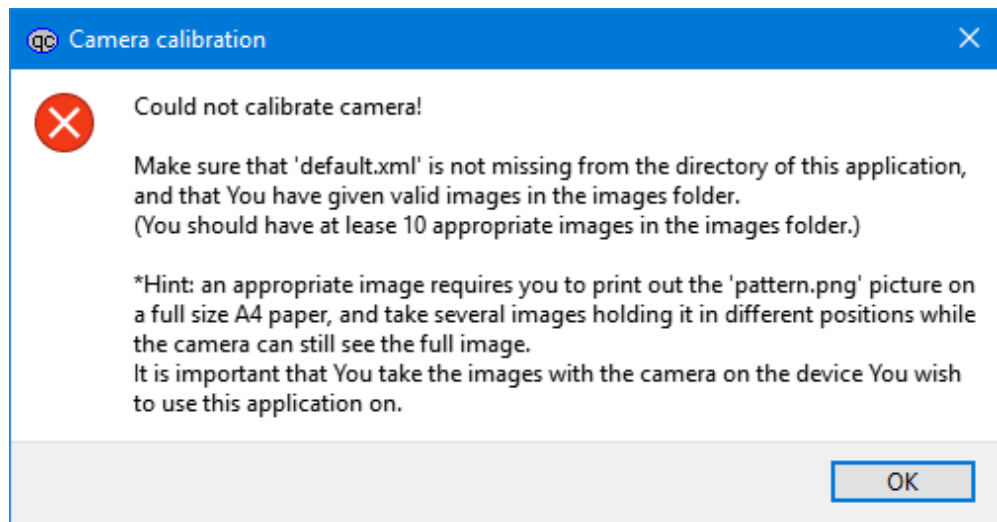
#### 1. *Camera not calibrated before*

Even though this is technically not an error message, this window is shown when the Your camera is not calibrated, and QuteCursor will use a default calibration setting.



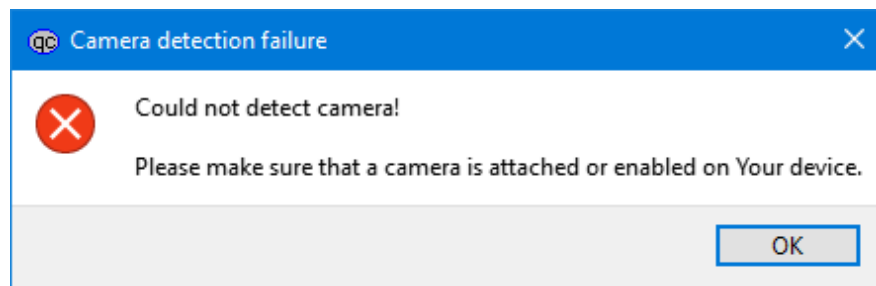
#### 2. *Missing file, or not enough pictures for calibration*

This message is shown, if You are missing a crucial file, or do not have enough valid pictures in the *images* folder.



### 3. *Cannot detect camera*

This message is shown, if QuteCursor is unable to detect a camera on Your device, and therefore cannot start the cursor control procedure.



# Chapter 3

## Technical documentation

### 3.1 Algorithmic background

#### 3.1.1 Face recognition

The main aim of the algorithm is to support relatively fast real-time facial recognition in order to allow the user to move their system's cursor with a relatively quick response time, with reasonable accuracy. In order for this happen, there are several simplifications in the acquired data.

The first step of the algorithm, is to get an input that resembles the face of a person. The full facial recognition cannot be simplified, and therefore the full face must be identified by every given input frame from the camera. This step is crucial for the algorithm to give an as precise calculation as possible. The facial recognition returns a set of points which can be analyzed for the purpose of the algorithm.

The most important points that are used in the process are what are called 'rigid-points'. The characteristic of these rigid-points is that no matter in what direction the person is looking, or what kind of expression one makes, these points do not seem to change position relative to the face. Such points are the inner and the outer corner of the eyes, the tip of the nose, and the points along the nose bridge. Since the main aim is to move the cursor by analyzing the set of points of the recognized face, it makes the most sense to choose points that are further apart from each other, and do not lie on the same line – evidently, this algorithm uses the outer points of the eyes and the tip of the nose. These three points create a

triangle, which can easily define a plane. These rigid-points can be later compared to a given set of three dimensional points, that resemble a face. This given set of points is referred to as the face mesh reference points.



Figure 3.1: A recognized face, and a set of possible rigid points

However, the set of points itself is not enough for determining any three-dimensional rotation of the recognised face, as from the camera input they all lie on the same two-dimensional screen coordinate system.

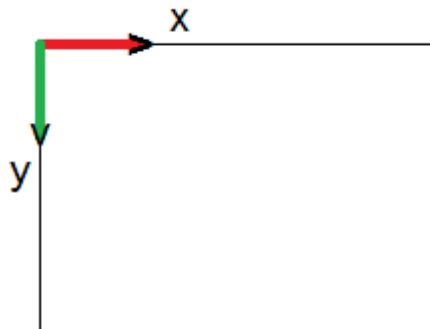


Figure 3.2: Screen coordinate system

### 3.1.2 Coordinate systems

In order to make a three dimensional mesh out of the three used points, the points are transformed onto a unit sphere coordinate system. First, the three points are

translated around the origin by their centroid – or their center of gravity. The translation also includes giving these points a third dimensional property.

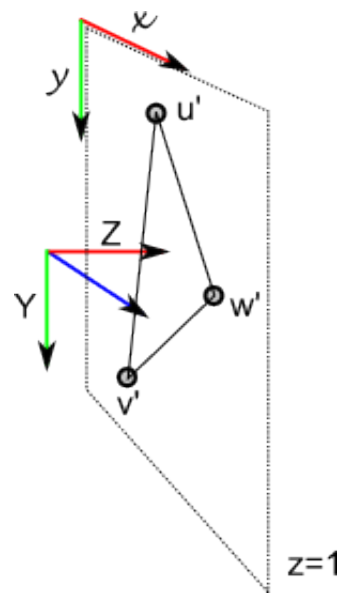
$$\begin{cases} u'_x = \frac{u_x - c_x}{f_x} \\ u'_y = \frac{u_y - c_y}{f_y} \\ u'_z = 1 \end{cases}$$


Figure 3.3: Translating the image from the image coordinate system [5]

For simplicity, the given  $z$ -coordinate of the three points is going to be 1. After their translation, they are projected onto a unit sphere, that has a center at the origin. Let this unit sphere have its own coordinate system, that will be referred to *sphere coordinate system* from now on in this thesis. To project the points into the sphere coordinate system, we divide their the three dimensional coordinates by their relative length that is acquired by the camera. In other words, we use the second normal for the three-dimensional point to be divided with.

The projected points are now placed in a three dimensional space, where the points never lie on the same plane. The acquired points in the sphere coordinate system can now be used for comparing them to the points of the face mesh.

As the algorithm uses information from the screen and the camera, there are several coordinate systems that need to be considered.

The first coordinate system, is the screen coordinate system. The screen coordinate system defines a two dimensional coordinate system, where – unlike what most



$$\begin{cases} N_u = \sqrt{(u'_x)^2 + (u'_y)^2 + (u'_z)^2} \\ u''_x = \frac{u'_x}{N_u} \\ u''_y = \frac{u'_y}{N_u} \\ u''_z = \frac{u'_z}{N_u} \end{cases}$$

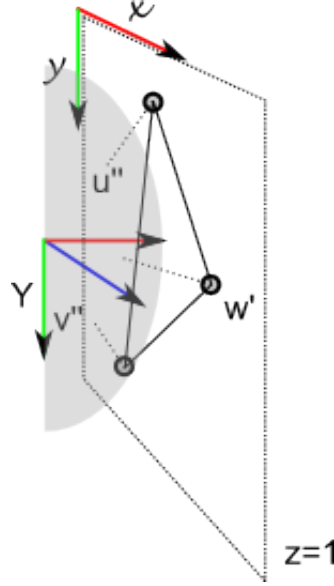


Figure 3.4: Projecting the points to the unit sphere coordinate system [5]

people are already used to – the y–coordinate increases downwards, the x–coordinate increases from left to right.

There are several importance of taking the screen coordinate system in consideration. The first is that the algorithm uses a set of three dimensional points for the default face mesh. The origin of the face mesh is unknown, and it is most popular to use a coordinate system where the y-axis increases from bottom to top. However, the used facial points will have a rather opposite coordinates regarding their positions on the y-axis. There are two possible solutions for making sure that the recognized coordinates can be matched up to the already known face mesh.

The first solution is to give such a mesh, where the face 'seems to be upside down', meaning that the upper the point is on the face, the less of a y–coordinate value it has. (For example, the tip of the nose would have a greater y–coordinate than the that of the corners of the eyes.)

The second solution is a bit more complicated. As the recognized points have values which cannot be predetermined, the algorithm must work with the originally

recognized points, and cannot alter them before they are acquired. After they are gained and pulled around the origin by their centroid, their values could be negated. This way, the greater y-coordinate means a higher point on the face. However, after every single recognition, these points would have to be negated one by one, which is an unnecessary step that the first solutions easily and successfully bypasses.

The sphere and the mesh's coordinate system are essentially both defined in the world coordinate system, and therefore the two set of points can be compared with each other.

### 3.1.3 Algorithm

Now that the problem is outlined, and the used data explained, an algorithm can be precisely defined for solving the head position estimation and cursor movement. When the possibility of dealing with different types of cameras and lenses arises, it is important to keep in mind that not all of the cameras are the same, and evidently, they could have different focal lengths. Even though this thesis does not go into much detail about how cameras can be calibrated, it considers the usage of different focal lengths. The importance of calibration is to gain as precise information as possible – hence making the algorithm as precise as it can be. The calibration function is separated from the main algorithm, and it is left as an optional choice for the user. However, if the calibration is skipped, then default calibration values will be used in the main algorithm. These default values are gained by experimentation on the device which the calibration was tested on.

After the optional calibration process, there are two ways, which the main algorithm could start with. The first option is to read the predefined face mesh reference points, which are used later on to compare them with the points that are processed after face recognition. The second way is to detect if there is any camera attached to the device at all. Even though these two options can be separated, threading these options would be unnecessary, since the algorithm could not continue if the camera input is missing. Therefore, the algorithm first checks, if it could detect a camera and then reads a predefined set of points. Taking memory management in consideration, this second option also ensures that the algorithm only stores information that it needs for each step, and not more.

Once the camera input is accessed, the focal lengths are stored. These values are read from a file which is either created during calibration, or by default, if no calibration file was found.

In order for the user to get feedback on how the algorithm processes the camera input, two windows are opened and placed beside each other. The first window shows the camera input, outlining the recognized face; while the second window shows the rotated face mesh reference points, where to rotational values are gained later in the algorithm. Since both windows show data which are processed during the algorithm, they will be updated and drawn on at the very end.

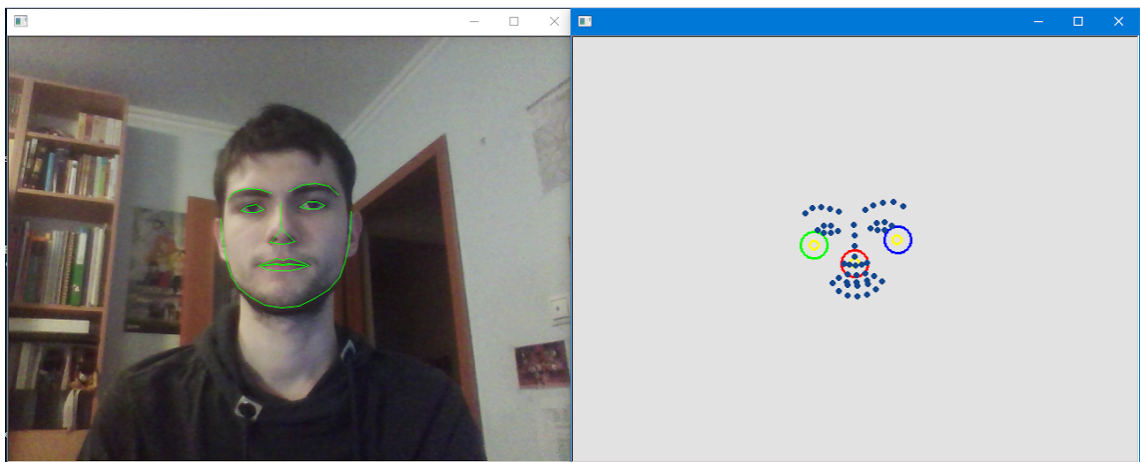


Figure 3.5: A recognized face, with a response window showing its current estimated position

After the windows are created, the algorithm reads the face mesh points from a file into an array, and separately stores the three corresponding points that we wish to use for the detected face landmark points. The last thing the algorithm needs before it can start processing data, is a tool for facial recognition.

With the tools and variables defined above, a loop can be created that will analyze the input data frame-by-frame, until the camera input is closed. First, the algorithm reads the input of the opened camera of the device, and stores its information. The face detector then analyzes the stored image, and returns with a set of points, each corresponding with a defined characteristic point on the face. These points can be used to extract single point data, that helps simplifying the problem of head pose estimation. The extracted set of points lie on the *screen coordinate system*, and they must be translated in order for these points to have a

meaningful projection on the *sphere coordinate system*.

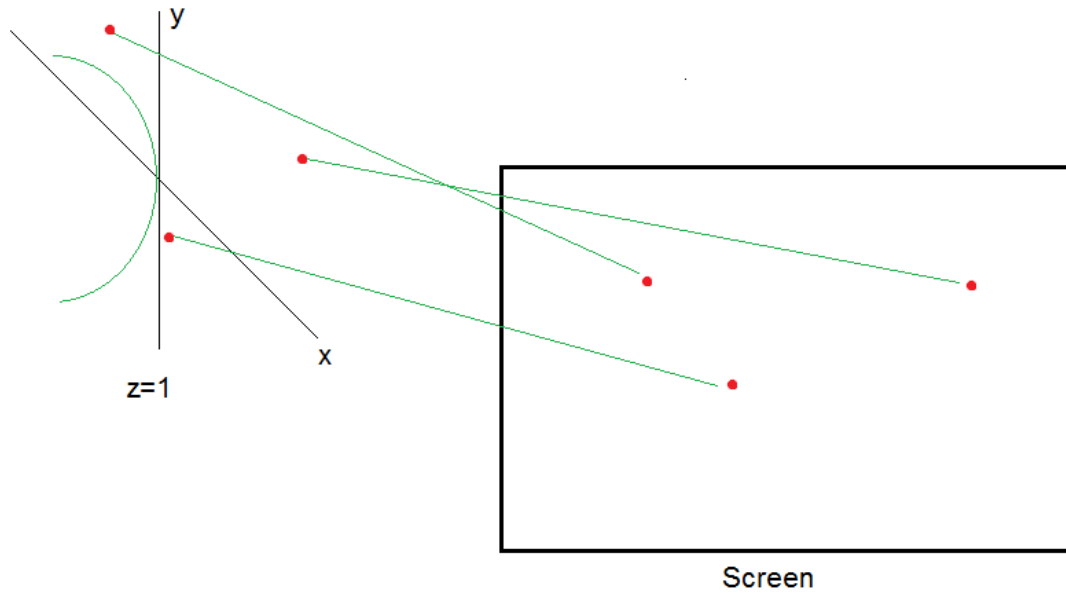


Figure 3.6: Transforming the points from the screen coordinate system to the unit sphere coordinate system

The first step of the translation is calculating the centroid of the collected points, which, in other words, is the points' center of gravity. The centroid helps in translating the points such that they are scattered around the origin. The second step is to give these points a third dimensional attribute, so that they lie in a three dimensional space. At this state, the algorithm also removes the pixel units, which is done by dividing the x-and y-coordinates of the points by the focal lengths (of the camera, along the x-and y-axis) respectively (Figure 3.3). The points are now ready to be projected onto a unit sphere.

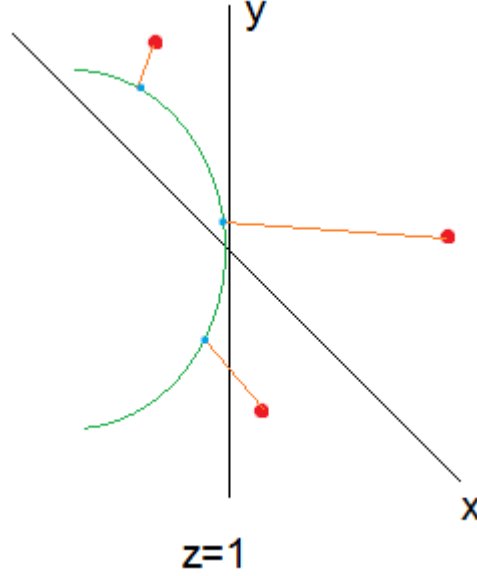


Figure 3.7: Projecting the points onto a unit sphere

Let the origin of a three dimensional space be the center of the unit sphere, and let the plane of the translated points placed ‘in front of’ the unit sphere, moved along the  $z$ -axis. The algorithm calculates the three dimensional distances of the points, and use these values as dividing coefficients for their corresponding points. The projection is done by dividing each point by their corresponding coefficient. It is easy to see, that the projected points are now set in three dimensions, where none of the sets of three points lie on the same two-dimensional plane (Figure 3.4).

In order for the points on the unit sphere to be closely comparable with the face mesh reference points, the calculated points must be enlarged relative to the order of magnitude of the reference points. After the geometric inflation of the points on the unit sphere, the algorithm can begin calculating the transformation of the face mesh reference points that is needed for them to coincide with the calculated points.

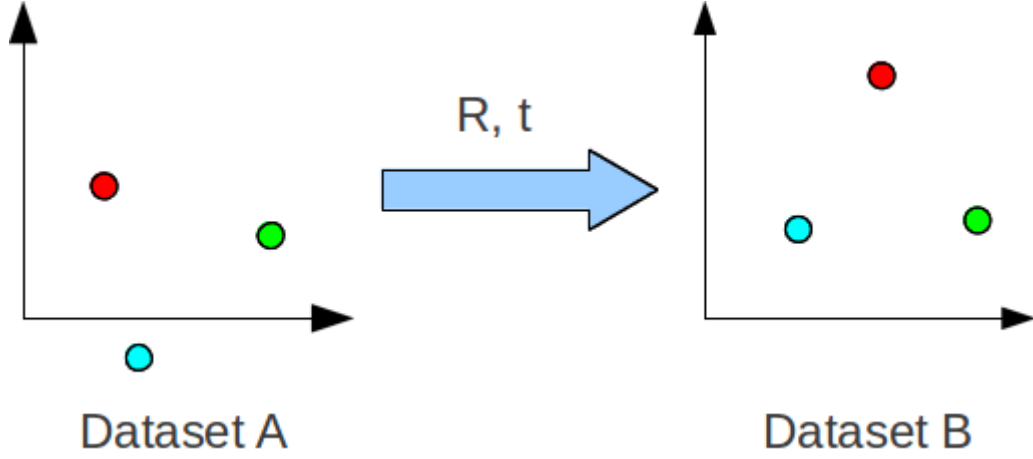


Figure 3.8: Rotating a set of points onto another corresponding set of points.[6]

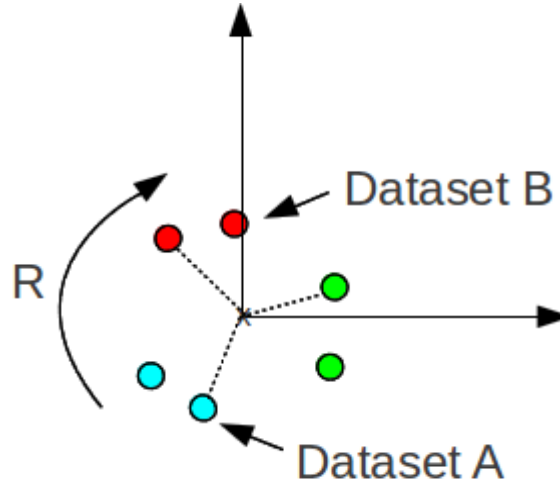


Figure 3.9: Defining the same centroid for both set of points, so only the matrix of rotation is needed to be calculated.[6]

The transformation detailed previously, can be defined by a certain matrix of rotation. The matrix of rotation is constructed by multiplying vectors. The vectors essentially define both calculated and face mesh reference points. Multiplying a point of the face mesh with the transponent of its inflated corresponding point (both in vector representation), returns a three-by-three matrix.

$$H = \sum_{i=1}^N (P_A^i - centroid_A) \cdot (P_B^i - centroid_B)$$

Figure 3.10: Equation for calculating the matrix used by the SVD [6]

$$\Rightarrow H = \vec{a} \cdot \vec{b} = \sum_{i=1}^N \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}_i \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}_i^\top$$

$$\Rightarrow H = \sum_{i=1}^N \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}_i \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}_i = \sum_{i=1}^N \begin{bmatrix} a_1 b_1 & a_1 b_2 & a_1 b_3 \\ a_2 b_1 & a_2 b_2 & a_2 b_3 \\ a_3 b_1 & a_3 b_2 & a_3 b_3 \end{bmatrix}_i$$

The matrix the algorithm uses, is composed by the sum of all multiplied pair of points. The Single Value Decomposition (SVD) of the calculated matrix can be used to get the matrix of rotation between the two sets of points. This thesis does not go into detail of how the SVD calculation works, however, it "factorises a matrix [...] into 3 other matrices, such that:  $[U, S, V] = SVD(E)$  and  $E = USV^\top$ "[6].

Since the SVD decomposes the given matrix into three matrices (U, S, and V matrices), they can be used to find out the matrix of rotation. The following equation returns the exact matrix of rotation[6] :

$$R = VU^\top$$

The gained matrix of rotation can be decomposed and analyzed to calculate the rotations along the x, y, and z axes (otherwise known as yaw, pitch, and roll axes). It is important to decompose the matrix of rotation, since the mouse cursor is controlled by the interpretation of these values.

There are several methods of extracting the rotation along the major axes in a three dimensional coordinate system, that can be found in many books and websites online. The following method is acquired from a book called *Planning Algorithms* written by Steven M. LaValle[7, pp. 99–100].

Suppose that  $R^{3 \times 3}$  matrix of rotation is given.

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

The following three equations return the values of rotations around the main axes

in radians:

$$\alpha = R_x = \tan^{-1}(r_{21}/r_{11})$$

$$\beta = R_y = \tan^{-1}(-r_{31}/\sqrt{(r_{32}^2 + r_{33}^2)})$$

$$\gamma = R_z = \tan^{-1}(r_{32}/r_{33})$$

*Planning Algorithms* can be officially downloaded from <http://planning.cs.uiuc.edu/>, and the referred page numbers are cited as they are present in the third chapter of the online version of the book.

Along with rotations around the different axes, the matrix of rotation is also used to rotate the face mesh reference points to give a feedback to the user – showing the outcome of the analyzed data.

The final step of the algorithm is to move the cursor by using the gained rotational axes. The following figures shows the model created for calculating the mouse cursor positions.

Based on experimentation, this thesis considers the angle of rotation around the x-axis (roll axis) to be 6 degrees above the plane of horizon ( $\omega_{x_1}$ ), and 30 degrees below the plane of horizon ( $\omega_{x_2}$ ). The angle of rotation around the y-axis (pitch axis) is considered to be 5 degrees in both left and right directions ( $\omega_y$ ). Let these angles be called the angles of deviation, and mean the angles which the head is rotated along the x-and-y-axes when looking at the top, bottom, left side, or right side of the screen, as opposed to it looking at the center of the screen. The cursor position is calculated with trigonometric functions based on the presumptions of the previously described angles of deviation.

Let ' $P_H$ ' denote the position of the head, ' $T_S$ ' the top of the screen, ' $B_S$ ' the bottom of the screen, ' $L_S$ ' the left side of the screen, and ' $R_S$ ' the right side of the screen. Using these notations, the following diagrams show the model created for calculating the cursor position.



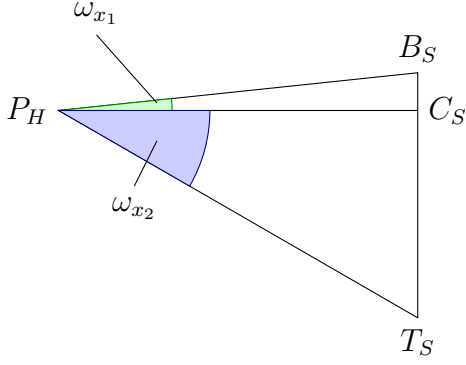


Figure 3.11: Angles of deviation along the x-axis

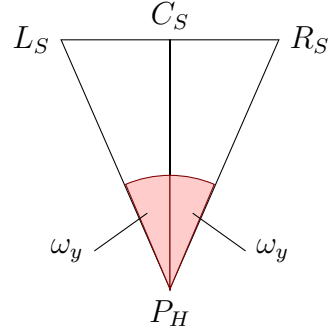


Figure 3.12: Angles of deviation along the y-axis

## 3.2 Architecture

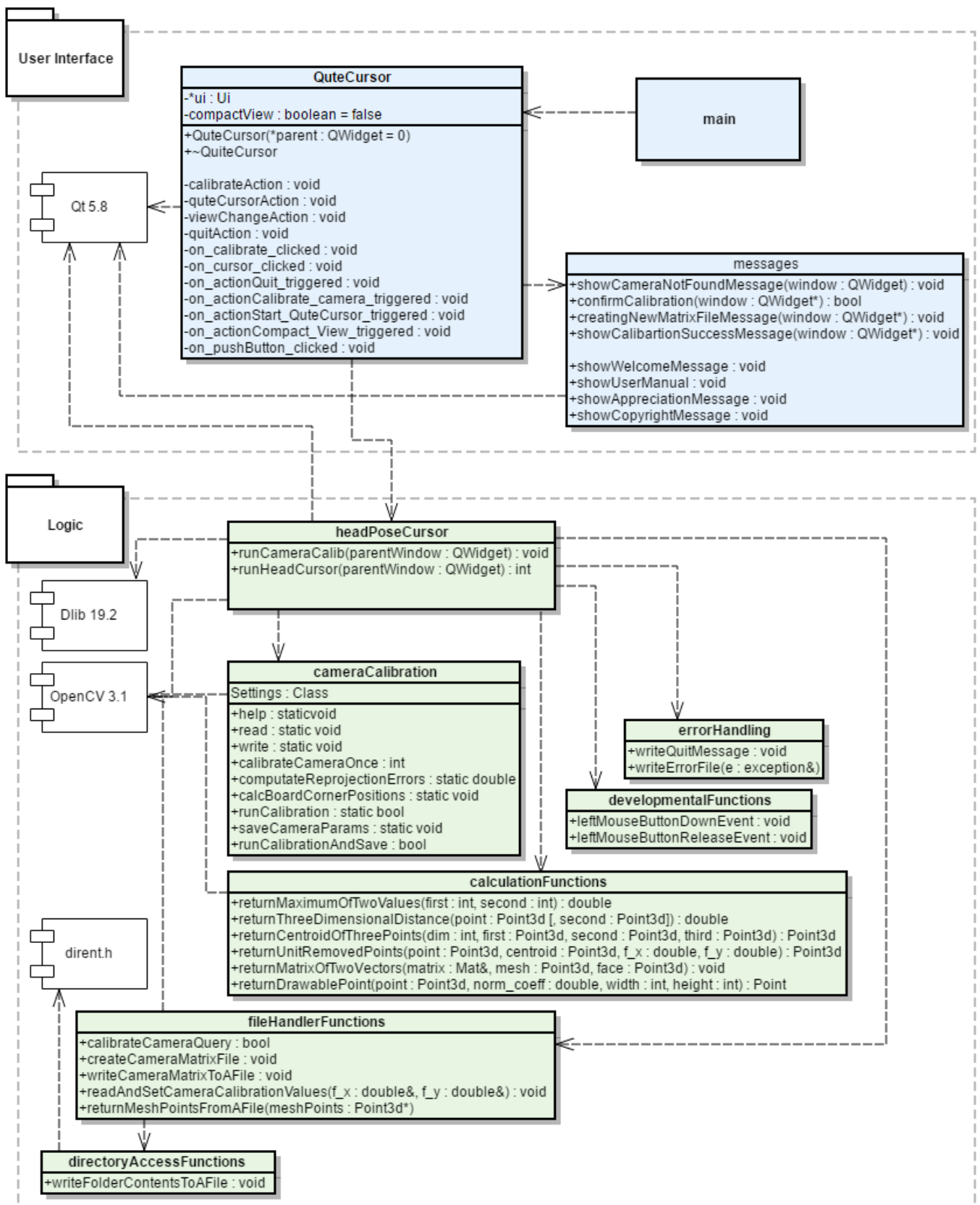
### 3.2.1 Static architecture

The QuteCursor project is designed to solve the cursor movement by head position estimation, which is essentially one complex problem. However, the complexity of this problem only reaches to the extent of returning newer positions of the mouse cursor. Therefore, the creating a separate class for the main algorithm is not justified.

Rather than creating the algorithm its own class, the functions it uses are separated into smaller sets, each set having its own source file. From now on, this chapter will refer to “classes” and “classification” as the way of logic which the subsets were defined.

The main program is separated into two general classes – the logic and the user interface classes. Everything, which is classified and defined under the logic section is responsible for the calibration, calculations, directory checking, and file and error handling. In order to avoid complicated parameterization between the user interface and the logic sections, the windows – that return the response to the user – are also defined in the logic layer. Everything that is responsible for convenient user experience, such as the main window and its contents, and the responsive output messages are defined in the user interface class.

The connection of each set of functions is represented by the dependency diagram below.



### **Source files and functions:**

This section of the paper will go through the separated source files, thoroughly explaining the purpose of each function they contain. As this is only the architectural definition of the project, the exact implementation will be detailed in the Technical specifications chapter. This part gives a general outline of how the functions were named, what return values, and what parameters they have, regardless of the used programming language.

To avoid confusion about the direction of dependencies, the structure is explained from the functions which need the least dependencies to the ones which need the most.

### **Error handling:**

In case of any errors or exceptions, it is the most convenient to write out the error message in a separate file, and halt the program with a quit message. This allows the developer to check back on the latest error message that the program encountered, if a developer framework would not support the output of error messages upon running the program. Therefore, the error handling file contains two functions.

1. *writeErrorFile* is a function which gets the error as a parameter, and writes its contents out in a file specified within the function. The file that this function writes in is not returned, but created and closed in the function.
2. *writeQuitMessage* is a function which outputs a quit message to the user. The purpose of this function is to halt the program in case it would run further with possible errors. There are no parameters or return values.

### **Directory access functions:**

The camera calibration requires the program to check on the contents of a specified directory, which includes the images that are used for the calibration. The names of the contents of the file are then written in an xml document, which is needed for the calibration process. The program uses the generated xml document which contains the paths of each image (inside the *images* directory) to calibrate the user's camera.

1. *writeFolderContentsToAFile* is a function which opens such directory, then checks its contents, and writes their names inside the xml document. There is no return value, the generated file is created and modified within the function.

### **File handler functions:**

The camera calibration process requires the program to check for previously saved calibration values, find and save calibration values both to and from a file, and it needs to be able to read a set of predefined three dimensional points. The following functions are used in the file handler source.

1. *calibrateCameraQuery* asks for the user's confirmation whether they want to start the calibration process or not. There are no input parameters of this function, and it returns a boolean value that represents the user's answer of confirmation or denial.
2. *createCameraMatrixFile* is ran only if there were no previously saved calibration values, and a file must be created in order for the head pose estimation to work. The head pose estimation uses these values for its calculations, and therefore a calibration file must be present. This function creates a file which stores values based on experimentation on the device this program was created and tested on. There are no return values.
3. *writeCameraMatrixToAFile* is used during the calibration process, and its purpose is to store the acquired camera parameters in a file. This function needs the acquired data of the calibration as an input parameter (which is a matrix). The created file contains the focal lengths of the camera. The focal lengths are needed for the head pose estimation process. There are no return values.
4. *readAndSetCameraCalibrationValues* is a function that reads out the focal lengths of the camera from a file. It is presumed that the properties of the camera were already stored in a file, which is done by the previously outlined methods. This function needs two floating number input parameters by reference, which are given values read from a specified file. The parameters are the focal lengths about the x and y axes. There is no return value.

5. *returnMeshPointsFromAFile* reads a specified file, and stores its values in an array, which contains three dimensional points. The array is given as a reference input parameter. It is important that the file this functions reads contains only numbers (integer and / or floating point). Each line should have three numbers written beside each other – the leftmost being the x, the middle being the y, and the rightmost being the z coordinate of the three dimensional point. The numbers should be separated by a whitespace. Each line represents a three dimensional point, hence a carriage return must be placed after each line, except from the last one. There is no return value.

### **Camera calibration:**

Since the specific process of the camera calibration is neither the task of, or specified part in this thesis, a third party calibration code was used with minor modifications. The main function of this standalone calibration code was renamed to *calibrateCameraOnce*, which holds no parameters nor a return value. The specific implementation and use is explained later on in the Technical specifications section of this thesis. The *calibrateCameraOnce* function initiates the calibration process.

### **Calculation functions:**

The head pose estimation needs many different type of calculations to analyze the facial points. The following functions help to make this calculations to be easier and more comfortable to use in the main algorithm. These methods are strictly mathematical, and use and manipulate several types of mathematical structures.

1. *returnThreeDimensionalDistance* is a twice overloaded function. The first definition returns the distance of the three-dimensional point given as an input parameter from the origin. The second definition requires two three-dimensional points as input parameters, and it returns the distance between those two points. Both functions return a floating point number.
2. *returnCentroidOfThreePoints* takes four input parameters. The first parameter tells the dimension which the centroid is considered. For example, if the first parameter is '1', then the center of two points will be calculated along the x axis; likewise, if the first parameter is '3', then the centroid is calculated

in all three dimensions. The other three parameters of the function are three-dimensional points, of which the centroid is calculated. This function returns a three-dimensional point.

3. *returnUnitRemovedPoints* essentially translates a three-dimensional point around the origin – using its centroid –, and removes its pixel units. It requires four input parameters. The first parameter is the point, which need to be processed. The second parameter is the centroid of the point. By itself, this function only makes sense if the point is part of a set, where the number of points in the set is greater than two. However, this functions is used in the analysis of the input face points from the camera meaning that the number of points in the set is greater than two by prerequisite.
4. *returnMatrixOfTwoVectors* stores a three-by-three matrix which is the result of mutiplying two three-dimansional vectors together. The first vector must be a three-by-one dimensional, while the other must be a one-by-three dimensional vector in order for this function to work properly. This function creates the matrix which is used to calculate the matrix of rotation. The first parameter is a reference to a storage of a matrix. The second and third parameters are the pair of corresponding vectors. The theory behind this function is outlined in the previous chapter.
5. *returnDrawablePoint* serves as part of the response towards the user. This function returns with a two-dimensional point which can be drawn on the mesh response window. It takes four parameters. The first parameter is the three-dimensional point which needs to be translated onto a screen coordinate system. The second parameter is a coefficient, which the point needs to be multiplied with in order to enlarge it. The third and fourth parameters are the window width and height.
6. *returnMaximumOfTwoValues*, *returnAnglesBetweenThreePoints*, and *returnSumOfAnglesBetweenThreePoints* are functions that were developed and tested, but unfortunately were not used due to a change in the main algorithm method.

### **Head Position estimation and cursor control:**

The head position estimation is technically the main function of the previously described set of functions. It hold together all the pieces of code which is needed in order to solve the main problem. *runCameraCalib* and *runHeadCursor* are the two most important functions defined, as these are the methods that are used by the user interface. Both methods have an input parameter of a window pointer, which is needed for the possible messages to be placed relative to the main window of the application. Since these two functions are essentially define a collective use of the previously described functions, their inner mechanisms are detailed in the Technical specifications of this thesis.

### **Graphical user interface:**

The graphical user interface is composed of three parts. The first part is the definition of the window, which holds all the menu items, buttons, and texts. The second part renders functionality to the items on the user interface, defining user events, such as a button being clicked. The third part is a function that constructs and runs the entire application.

## **3.2.2 Dynamic workflow**

The running application can be represented with the following state diagram. The diagram shows how the user is able to interact with the application.

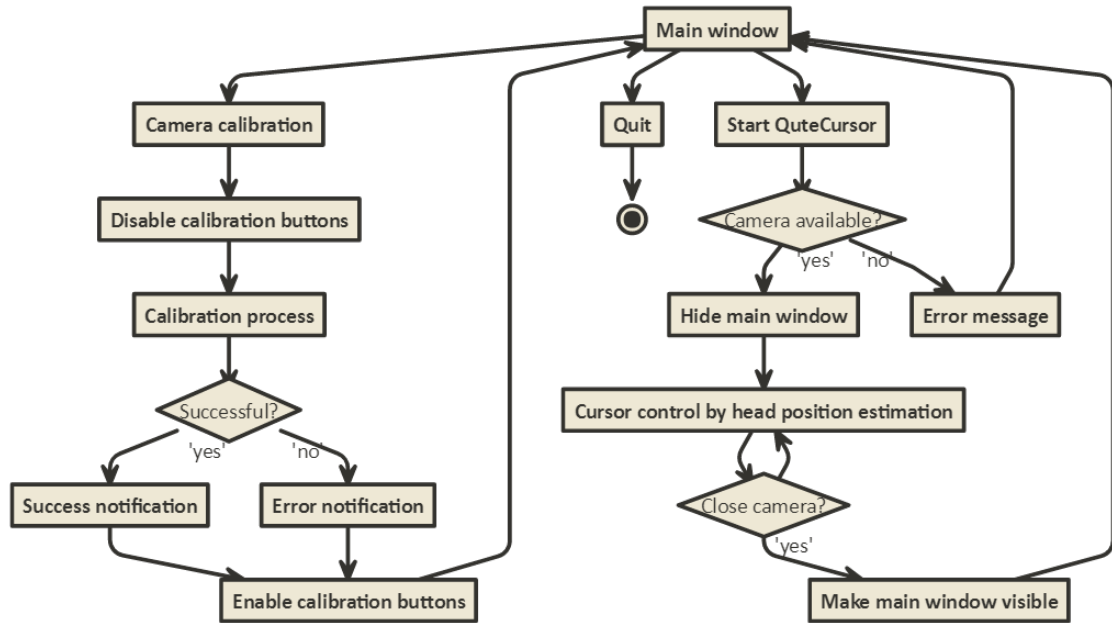


Figure 3.13: State diagram of the application

### 3.2.3 Implementation details

The purpose of this chapter is to give a detailed guide of the implementation of the algorithm, and the entirety of the project itself – from the very basics until the smallest details on the graphical user interface.

#### Mocking the cursor movement:

The first phase of the project is to test whether the system mouse cursor can be automated using any kind of programming language. Both C++ and JAVA language support mouse handling. As the official JAVA 7 documentation puts it: “[the `java.awt.Robot`] class is used to generate native system input events for the purpose of test automation, [...] and other applications where control of the mouse and keyboard is needed”[8]. The `Robot` class contains a `mouseMove(int x, int y)` method, which is satisfactory for positioning the mouse cursor to the given input parameters. Microsoft’s Development Network states that with the minimum of Windows 2000 operating system and the inclusion of the `Windows.h` header file, the Microsoft Visual C++ language supports a function called `SetCursorPos(_In_ int X, _In_ int Y)`. Similarly to JAVA’s `Robot.moveMouse` method, `SetCursorPos` is also satisfactory for placing the cursor to the given parameters[9]. The parameters



of both functions are referring to a point on the screen by their coordinates in the screen coordinate system.

Since the algorithm requires greater precision and possibly more efficient memory management – due to the quickly analyzed data - the Visual C++ language is used to implement the project. The development framework used for the implementation of the logic layer is Microsoft’s Visual Studio 2015 Community Edition.

### **Installation of dependencies and external libraries:**

Since the possibility of automating the mouse cursor is granted, the problem which remains to be solved is the head position estimation. The first step of this process is the facial recognition. Eötvös Loránd University already has experience in creating face recognition software, and the suggested API for the face recognition in C++ language is the Dlib C++ Library. This project uses Dlib version 19.2.

Dlib can be downloaded from [dlib.net](http://dlib.net). In order to include the libraries in a given project, a new system environment variable must be created pointing to the 'dlib' directory within the downloaded and unzipped Dlib library.

The next step is to use a package that supports camera access, and several more complex mathematical models, such as matrices, geometric shapes, and vectors. One such API is called OpenCV, and it is described as “... an open source computer vision and machine learning software library”, which “... has C++ [... and] JAVA interfaces and supports Windows [...]. OpenCV leans mostly towards real time vision applications...”[10]. Essentially, OpenCV supports the algorithms and models needed for solving the problem of head position estimation. This project uses OpenCV version 3.1 Windows edition, which is available for download on [opencv.org/releases.html](http://opencv.org/releases.html). Similarly to Dlib, OpenCV must be extracted somewhere on the computer, and a new system variable must be made that points to the 'build' directory of the OpenCV library. Furthermore, in order to include OpenCV source files into a project, the system path must be extended. The extension must point to the 'binary' directory of OpenCV, depending on the type of system and developer framework used. In this project, this path value points to 'OPENCV\_DIR/x64/vc14/bin' directory, where 'vc14' refers to Visual Studio 2015.

With OpenCV and Dlib included in the system, the tools which the main algorithm can be solved is given. A new project can now be set up. Instead of creating

a new project in Visual Studio, CMake is used to set up the project for us. CMake can find and set dependencies (in this case OpenCV and Dlib libraries) which could not be straightforward if done manually within Visual Studio. For this project, CMake 3.7.0 is used. Specifically for this project, CMake requires that both Dlib and OpenCV directories are included in the system environment - as explained above - as well as Visual Studio 2015 installed with its Visual C++ compiler. In order to set up a project, CMake requires a CMakeLists.txt file, which holds the settings and characteristics of the project. To create the project, the 'CMakeLists.txt' and the 'face\_landmark\_detection\_ex.cpp' files were modified and used. Both of the files can be found inside the 'examples' directory of the Dlib library. The latter C++ source file is renamed to 'headPoseCursor.cpp'.

#### **Modifying the CMakeLists.txt file:**

PROJECT(HeadPoseCursor) defines the name of the project which is created by CMake.

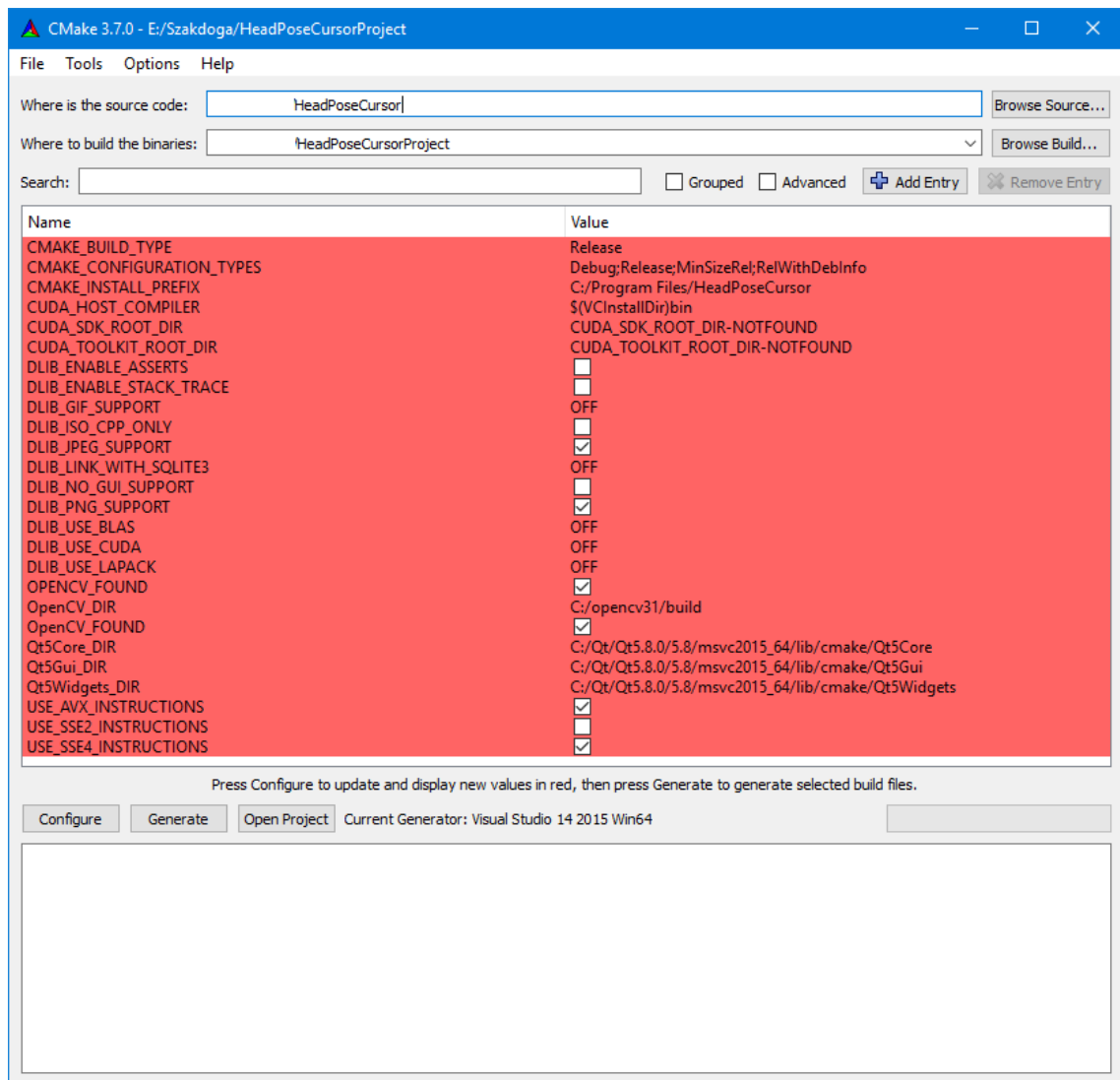
include(..../dlib-19.2/dlib/cmake) tells CMake to find the Dlib library, and include it in the generated project.

ADD\_EXECUTABLE(headPoseCursor headPoseCurosr.cpp) tells CMake which files to add in the generated project, and how to refer it as an executable.

TARGET\_LINK\_LIBRARIES(headPoseCursor dlib) tells CMake to add the Dlib libraries as dependencies of the headPoseCursor source.

The OpenCV libraries are included and linked similarly to Dlib.

Open CMake Gui, and select the folder where the source files and the CMakeLists.txt exists, and select a folder where CMake should generate the project. By clicking on "Configure", CMake will go through the given 'CMakeLists' file, and set the dependencies. For face detection, Dlib recommends the usage of AVX and / or SSE4 instructions to be selected. By default, only the SSE2 instructions are enabled. Click on "Configure" to finalize the configuration. If Dlib, OpenCV, and Visual Studio are set up correctly, the following window should appear.



Click on “Generate” to generate the Visual Studio 2015 project, then click on “Open Project”. The project is now set up along with the needed dependencies.

### Implementation:

The aim of this section is to go through the details of the main algorithm. Since Microsoft Visual Studio 2015 framework was chosen for the implementation of this project, it is important to point out several characteristics of the C++ language and the working environment. The main algorithm is composed of the functions explained in the Static architecture chapter. The signature and the implementation of these functions are separated in two different files: the header file and the source file. The header files are created such that they begin with the needed libraries that have to be included for the functions, and the signature of the functions to work. As it was advised by Zoltán Tóser, it is important that namespaces are never used within

a C++ header file, due to the high probability of ambiguating datatype structures that are used in the different source code. After adding the needed includes in the header files, the namespaces become available for use in their respective source files. If a function returns or requires a special datatype or structure, then they are referred to as “<namespace>::<data type>” in the header file.

An example for this can be found in the calculationFunctions. The header file declares the signiture of the function as

```
cv::Point3d returnCentroidOfThreePoints(  
    int dimensions,  
    cv::Point3d firstPoint,  
    ...  
);
```

It is important to keep in mind that in the actual source code - where the methods are defined - it is safe to use the “using namespace <namespace>;” declaration. Therefore, with the example given above, the source code refers to the same function as

```
Point3d returnCentroidOfThreePoints(  
    int dimensions,  
    Point3d first,  
    ...)  
{ ... }
```

where “using namespace cv;” is declared at the top of the code.

The source code communicate with each other by the header files, so it is important to include the headers where necessary.

This next section defines how the camera calibration is used during the project. As mentioned before, this thesis does not go into the details of camera calibration, and therefore uses a thrid party calibration algorithm which is provided by OpenCV. An example camera calibration code is available at ”OpenCV/sources/samples/cpp/tutorial\_code/calib3d/camera\_calibration”. The source code found inside the directory explains how the camera calibration should be used in order for the code to work. This source code was manually modified for the purposes of this project.

The first modification was splitting the given `camera_calibration.cpp` source file into a header and a C++ source file. The newly created `cameraClairbration.h` header file includes the C++ `<iostream>`, `<ctime>`, and `<cstdio>` for possible standard library outputs used for quick testing during the development of the project, and includes “`messages.h`”, which is a separately written code that is used for outputting instructions on the console. OpenCV source files are also included here. The “Settings” class as well as the signatures of all the functions are moved to this header file. Since it is not safe to use namespaces in headers - as explained above - it is required that all of the OpenCV datatypes and structures are given the “`cv::`” namespace, as well as that the standard library functions are given the “`std::`” namespace. The “Settings” class and the signature of the functions are otherwise left as they were already present. The “main” function is renamed to “`calibrateCameraOnce`”, and it requires a `QWidget` window pointer as its only parameter. The usage of this parameter is explained later in the graphical user interface implementation.

Moving on to the C++ source code, it includes its corresponding header, and the “`fileHandlerFunctions.h`” header file. It uses the `std` and `cv` namespaces.

As mentioned several times, OpenCV’s default camera calibration example code was used, thus only the algorithm and the modifications of `calibrateCameraOnce` function is detailed in this thesis. The function starts with outputting some text about the usage and the files it is required to run. This function is designed to run only after the user confirmed their desire of calibrating the camera, therefore the first modification of this method was to show a message to the user to wait until the calibration is finished. This message was first written as an output message while the program ran in console, but considering more user-friendly user interface, it was later redefined to appear in a `QMessageBox`, which is a Qt development framework specific datatype. This will be detailed later in the graphical user interface implementation rather than in the implementation of the logic section. The algorithm then reads a setting file, that holds all the information about the type of calibration used. This file is left as “`default.xml`” and it is a crucial file for the program to work. The “`default.xml`” file can be easily modified using any kind of text editor program, and it is well commented for anyone to be able to use. This file is configured for the user to use the default chessboard pattern image (that can be found in the default

OpenCV library, under "OpenCV/sources/doc directory") printed fully on an A4 sized paper. This file also tells the Settings process images for the calibration, with the maximum of 50 photos processed. The path of the photos are stored in the "VID5.xml" document. An example of the contents of the "VID5.xml" document:

```
<?xml version="1.0"?>
<opencv_storage>
  <images>
    images/xx1.jpg
    images/xx10.jpg
    images/xx7.jpg
    images/xx8.jpg
    images/xx9.jpg
  </images>
</opencv_storage>
```

After the settings file is read and accepted as a valid input file, the function creates two matrix datatypes.

When the calibration is finished, the cameraMatrix data holds the intrinsic camera parameters, such as the focal lengths, which are needed in the head position estimation. At the same time, the program shows the result of the calibration by applying the camera matrix and distortion coefficients on the input images. Originally, it was possible for the user to quit during the time the program calibrated and showed the results of the calibration by closing the shown windows. However, based on experimentation and testing, breaking any part of the calibration process seem to crash the program. For this reason, the code – which is responsible for allowing the user to quit the calibration after it has started – has been commented out. At the end of the method, all the windows are closed, the calibration results are written in a file, and a message notifies the user that the calibration was successful.

```
if(waitMessage.isVisible())
{
    waitMessage.close();
}
destroyAllWindows();
```

```
writeCameraMatrixToAFile(cameraMatrix);  
showCalibrationSuccessMessage(parent);
```

In order to ensure the safety of the process, the user is required to wait until the calibration is fully completed. The calibration message shown at the beginning can be closed, or left open, therefore before closing all the windows, it is checked whether the user closed the message or not.

The *writeCameraMatrixToAFile* function writes the intrinsic camera parameters in a file<sup>1</sup>.

The camera matrix is a three-by-three matrix, and it is written in a file such that it reflects the look of a three-by-three matrix. The elements of a `cv::Mat` matrix can be accessed by the `cv::Mat::at<Tp>(i,j)` function, where `<Tp>` is the type of data stored in the *i*-th row *j*-th column of the given matrix. In this case, the types are floating point numbers, and the elements are indexed from (0, 0) to (2, 2).

Once the calibration is done, four files in the directory should be created or updated, depending whether they existed prior to calibration or not. These four files are:

1. cameraMatrixFile
2. settings.yml
3. out\_camera\_data.xml
4. VID5.xml

The cameraMatrixFile is written as the result of the *writeCameraMatrixToAFile* function detailed above. The settings.yml and the out\_camera\_data.xml files are both created by the OpenCV camera calibration process. The VID5.xml document is created prior to the camera calibration, and it is initiated in the head pose estimation source code.

### **Head pose estimation:**

The headPoseCursor files are the last layer of the logic section, and they essentially connect all of the used functions which were mentioned in the Architecture

---

<sup>1</sup>QuoteCursor/fileHandlerFunctions.cpp, lines 22-42

section. The header file include every other source code (headers) that are needed for the head position estimation algorithm, and declares two function signatures, which are:

1. `void runCameraCalibration();`
2. `int runHeadCursor();`

In the final form of the application, these functions use a `QWidget` pointer type parameter, which will be explained later in the graphical user interface implementation section.

The `headPoseCursor` source file uses the `std`, `dlib`, and `cv` namespaces, and only includes its header.

The first function defined in this source file is the one that initiates the camera calibration<sup>2</sup>.

The first step of this function is to write the contents of the “images” folder to a file. More specifically, it’s purpose is to generate the `VID5.xml` document, which stores the paths of the images that are used in the camera calibration. An example of the contents of this file can be found in the previous section. The generation of the required xml document can be found in the directory access functions, and the code which is responsible for the generation is the following:

```
DIR *imagesDirectory;
struct dirent *entity;

void writeFolderContentsToFile()
{
    int lineCounter = 0;
    ofstream imageCalibrationInputFile;
    imageCalibrationInputFile.open("VID5.xml");

    imageCalibrationInputFile
        << "<?xml version=\"1.0\"?>"
        << endl
```

---

<sup>2</sup>`QuteCursor/headPoseCursor.cpp`, lines 45-52



```

        << "<opencv_storage>" << endl
        << "\t<images>" << endl;

if ((imagesDirectory = opendir("images\ ")) != NULL)
{
    while ((entity = readdir(imagesDirectory)) != NULL)
    {
        lineCounter++;
        if (lineCounter > 2)
            imageCalibrationInputFile
                << "\t\timages/"
                << entity->d_name
                << endl;
    }
    imageCalibrationInputFile
        << "\t</images>"
        << endl
        << "</opencv_storage>";

    closedir(imagesDirectory);
    imageCalibrationInputFile.close();
}
else
{
    cerr << "Could not open this directory." << endl;
}
}

```

The header of the directory access functions includes a “dirent.h” header file. The “dirent.h” header essentially implements directory access similar to that of Linux. “dirent.h” header is the created by Toni Rönkkö in 2015 under the MIT License, as it is presented in the file itself. The function uses a DIR pointer data type, which points to a directory – in this case, the directory the function needs to iterate through in order to discover its elements. It also uses a dirent pointer type, which is basically referring to a content inside a folder.

The function itself starts with the declaration and the initialization of an integer type counter that is set to zero. Then, an output stream file is created and opened. Let the name of this file always be “VID5.xml”, so the calibration function has an easier job of finding this file.

The beginning of the xml document is set up by writing the first few lines of the document, which is based on the given “VID5.xml” example. The function then checks, if it can open the “images” directory. If this fails, then the function returns with an error regarding the missing directory. Once the “images” folder is found and opened, the method iterates through its contents one-by-one in a loop. If it finds an element, the counter (defined at the start of the method) is incremented, and checked whether it is greater than two. If it is, then two tabulations, the images folder name, and the name of the entity are written in the file. The reason behind the counter is due to experimentation. While iterating through the contents of a folder, two entities “.” and “..” are always recognized, even if the directory itself is empty. To avoid writing these two lines in the generated file, they are skipped by checking the value of the line counter. Once the directory is fully read, the output file is properly finished by closing xml notation, the directory is closed, and finally, the output file itself is closed.

After the proper input file is generated for the calibration, the camera calibration process is started. If the calibration happens to fail, then the user is notified with an error message. The *showCalibrationFailureMessage* function is outlined in the graphical user interface section.

## Implementation of the head position estimation and cursor control

The final and most important function in the logical layer is the *runHeadCursor* function. It is a detailed step-by-step implementation of the main aim of this project, which uses all of the functions outlined in the Architecture chapter. Since the main algorithm requires a great number of resources on the computer – such as files, dependent libraries, directories, and camera access –, the algorithm is encapsulated within a try-block exception handler. If the block throws an error, then the error is displayed on the screen, and the function returns with a negative value indicating the error.

The following part focuses on implementing the head position estimation and how the estimation is analyzed in order for the cursor to be moved.

The first step is to open the primary camera attached to the device. If the camera is not available for use – due to it not being attached, or it being disabled on the device – then an error is thrown, and the function stops<sup>3</sup>.

As it is officially documented by OpenCV, VideoCapture is a “class for video capturing from video files, image sequences or cameras”[11], therefore this class is used to open the camera of the device. The parameter 0 in the constructor of this class indicates the request for using the default, or primary camera of the device.

The algorithm allocates memory to an input stream file, and tries to open a file called “cameraMatrixFile” (without any file extensions). The “cameraMatrixFile” stores the intrinsic parameters of the camera, which is stored after the camera calibration process. However, the camera calibration is an optional function, meaning it is not necessarily ran during the runtime of the application. In this case, the file may not be present in the directory of the application, and therefore a default file must be created<sup>4</sup>.

The default file is created by the function called *createCameraMatrixFile*. This method is defined in the file handler functions<sup>5</sup>.

The method basically fills up a file with intrinsic camera parameters, which were the results of the calibration on the device that this application was developed on.

Then, two floating point variables are defined for the camera focal lengths along

---

<sup>3</sup>QuteCursor/headPoseCursor.cpp, lines 59-65

<sup>4</sup>QuteCursor/headPoseCursor.cpp, lines 67-75

<sup>5</sup>QuteCursor/fileHandlerFunctions.cpp, lines 14-20

the x and the y axes, and their values are set by a function, which reads the values from the "cameraMatrixFile".<sup>6</sup>.

The method that sets the values are defined as such:

```
void readAndSetCameraClaibrationValues(  
    double &focallenght_x,  
    double &focallength_y  
)  
{  
    int doubleCounter = 0;  
    double temporaryValue = 0;  
    ifstream cameraMatrixFile;  
    cameraMatrixFile.open("cameraMatrixFile");  
    while (cameraMatrixFile >> temporaryValue)  
    {  
        doubleCounter++;  
  
        if (1 == doubleCounter)  
            focallenght_x = temporaryValue;  
  
        if (5 == doubleCounter)  
            focallength_y = temporaryValue;  
    }  
    cameraMatrixFile.close();  
}
```

After the focal lengths are set, two windows are created.

```
image_window win;  
win.clear_overlay();  
win.set_pos(0, 0);  
  
image_window win2;  
win2.clear_overlay();  
win2.set_size(640, 480);
```

---

<sup>6</sup>QuteCursor/headPoseCursor.cpp, lines 78-81

```
win2.set_pos(640, 0);
```

The first window (named “win”) is used to show the input from the camera as well as the outline of the recognized face. The purpose of the second window (named “win2”) is to show the default face mesh, and its rotation based on the head pose estimation. The windows are cleared and positioned beside each other. They receive the data that needs to be shown at the end of the algorithm.

The next step is to read the three-dimensional face mesh reference points from a file, and store it in an array so the points can be multiplied by the matrix of rotation in a loop, used later in the algorithm<sup>7</sup>.

The array consists of 49 predefined three-dimensional points which are read by the method called “returnMeshPOintsFromAFile”. This method requires a pointer of an array as an input parameter, which contains the three-dimensional points<sup>8</sup>.

The file that contains the face points is opened, and floating point variables are declared. The floating point variables hold the x, y, and z-coordinates of a point respectively. A line counter is then defined, that is used to index the “meshPoint” array, that needs to be filled. The points are read line by line, storing the coordinates separately, then defining a three dimensional point that is placed in the  $(n - 1)$ -th index of the array, where  $n$  is the number of cycles. As explained in the Algorithm background chapter, the file might contain such orientation of a coordinate system, which does not assimilate that of the used coordinate system for comparing the points of the recognized face and the points of the face mesh. In this special case, the y-coordinate of the mesh points have to be negated.

After the face mesh is stored, three particular points are defined, which are the points that are used for the comparison<sup>9</sup>.

For face recognition, Dlib’s frontal face detector and shape detector are being used.

```
frontal_face_detector detector = get_frontal_face_detector();  
shape_predictor sp;  
deserialize("shape_predictor_68_face_landmarks.dat") >> sp;
```

<sup>7</sup>QuteCursor/headPoseCursor.cpp, lines 94-95

<sup>8</sup>QuteCursor/fileHandlerFunctions.cpp, lines 62-73

<sup>9</sup>QuteCursor/headPoseCursor.cpp, lines 97-99

The shape detector accepts an input parameter, containing the shape which needs to be recognized on an image. This algorithm uses Dlib's 68-point face landmark predictor. An image of how these 68 points are scattered on a face can be seen in the appendix of this thesis.

The algorithm creates a loop which only stops once the first window is closed. The loop encapsulates all the calculation which is needed for the head position estimation.

```
while(win.is_closed()) { ... }
```

The loop starts by storing the camera information in a matrix datatype.

```
Mat cameraInformationUnflipped;  
cap >> cameraInformationUnflipped;
```

In the early stages of the development, it was found that the information from the camera returns a horizontally flipped image rather than a 'mirror-like' image. Therefore the next step is to horizontally flip the matrix. OpenCV has a built-in function called "flip", which is capable of doing this<sup>10</sup>.

Since the windows – that were created in the beginning of the algorithm – are the classes of Dlib, the camera data have to be converted to an image format that the Dlib window can work with.

```
cv_image<bgr_pixel> image_cameraInformation(cameraInformation);
```

Another matrix is defined for reading a background image for the second response window. Similarly to the camera information matrix, this background image matrix is also converted to an image format that Dlib's window can handle. The purpose of this is that the points of the rotated face mesh can be easily drawn on this background image.

```
Mat bg;  
bg = imread("bg.jpg");  
cv_image<dlib::bgr_pixel> info(bg);
```

After storing the correct camera input, it is given to Dlib's face detector which

---

<sup>10</sup>QuteCursor/headPoseCursor.cpp, lines 126-128

stores the bounding boxes of the detected faces in a C++ standard library vector. Another vector is defined, which stores the actual shape of the recognized faces within the bounding boxes<sup>11</sup>.

With the detected faces, the algorithm is ready to analyze the points and start the head position estimation. The face detector is capable of recognizing multiple faces on a given image. Even though in the special case of this project, it only makes sense to analyze one face, it is unknown in what order Dlib's face detector stores the bounding boxes of the recognized faces. Therefore, a for-loop is created to analyze all the recognized faces. As face detection is a complicated computational process, it works best in a well-lit environment, where all the parts of the face can be easily "seen" by the camera. The Algorithm background chapter also describes why it is important for the algorithm to recognize only one human face<sup>12</sup>.

### **Extracting the necessary points:**

The first step of extraction is to use the shape predictor to store the points of the recognized face, in the vector. One element in the vector stores a set of points (the 68 points described above), where each point can be addressed separately. Each of these points have an x and a y-coordinate, indicating their screen coordinate position within the camera image. Based on the 68 point layout, three points are chosen that are adequate for the head position estimation. For this project, the outer corners of the eyes, and the nose tip points are used. As it is described in the Algorithm background, the points are set in the image coordinate system, which is a two dimensional plane along the x and y-axes. In this coordinate system, the y-coordinate increases downwards, meaning that the eyes have a lower y-coordinate attribute than that of the nose tip (if looked from a normal, upright position). A third-dimensional attribute is added to the extracted points.

```
Point3d rightEye_raw((shapes[0].part(45).x()),
                    (shapes[0].part(45).y()),
                    1);

Point3d leftEye_raw((shapes[0].part(36).x() ),
                    (shapes[0].part(36).y()),
```

---

<sup>11</sup>QuteCursor/headPoseCursor.cpp, lines 134-137

<sup>12</sup>QuteCursor/headPoseCursor.cpp, line 138

```

1);

Point3d noseTip_raw((shapes[0].part(30).x() ),
                    (shapes[0].part(30).y()),
                    1);

```

The necessary points are now set in a three-dimensional space, but still lie on the same plane.

The next step is to pull the points around the origin. For this, the centroid of the three points is used, which is calculated separately with a function<sup>13</sup>.

The *centroidOf\_rawFacePoints* function needs four parameters. The first parameter tells the function in what dimension should the centroid be calculated. For example, if ‘1’ is given, then the average of the x-coordinates of the points will be calculated. If ‘2’ is given, then the average of the x and y-coordinates will be calculated. If this parameter is set to ‘3’, then the centroid will be calculated in the three-dimensional space. The second, third, and fourth parameters are the points that we want to calculate the centroid of. The default case dictates that the centroid in the 3D space is calculated<sup>14</sup>.

The extracted points are then changed by pulling them around the origin, and divided by the focal lengths of the camera<sup>15</sup>.

The function which applies these changes is defined as:

```

Point3d returnUnitRemovedPoints(Point3d point,
                                Point3d centroid,
                                double focalLength_x,
                                double focalLength_y)
{
    return Point3d((point.x - centroid.x) / focalLength_x,
                  (point.y - centroid.y) / focalLength_y,
                  1);
}

```

<sup>13</sup>QuteCursor/headPoseCursor.cpp, line 149

<sup>14</sup>QuteCursor/calculationFunctions.cpp, lines 28-43

<sup>15</sup>QuteCursor/headPoseCursor.cpp, lines 151-153



The algorithm now possess a set of points which are scattered around the origo, and have the z-coordinate attribute of 1.

The next step is to project the set of points onto a unit sphere. This is done by calculating the three dimensional distances from the origo of each point, and divide each point with their corresponding three dimensional distance.

```
double normalizingCoefficientOf_rigthEye
    = returnThreeDimensionalDistance(rightEye_raw);
double normalizingCoefficientOf_leftEye
    = returnThreeDimensionalDistance(leftEye_raw);
double normalizingCoefficientOf_noseTip
    = returnThreeDimensionalDistance(noseTip_raw);

// points on the unit sphere
Point3d rightEye_normalized
    = rightEye_raw / normalizingCoefficientOf_rigthEye;
Point3d leftEye_normalized
    = leftEye_raw / normalizingCoefficientOf_leftEye;
Point3d noseTip_normalized
    = noseTip_raw / normalizingCoefficientOf_noseTip;
```

The next step enlarges the points from the unit sphere, and positions them on the middle of the second response window. These points are represented as three circles: a green circle for the left eye, a blue circle for the right eye, and a red circle for the nose tip. The first parameter of the "circle" function tells the program to which image to use as a drawing panel. In this case, it is the background image that is being drawn on<sup>16</sup>.

The drawable points are two dimensional points and are created with the following function:

```
Point returnDrawablePoint(Point3d point,
                          double norm_coeff,
                          int window_width,
                          int window_height)
```

---

<sup>16</sup>QuteCursor/headPoseCursor.cpp, lines 167-174

```

{
    return Point(point.x * norm_coeff + window_width,
                 point.y * norm_coeff + window_height);
}

```

Before the face mesh reference points and the unit sphere points are compared, the points on the unit sphere have to be enlarged so that the order of magnitude of the values are similar with each corresponding point. The constants used to enlarge the points are arbitrary, and were chosen based on experimentation. Let the name of the enlarged points be *corresponding points*. It is important to mention, that the face mesh reference points used in this project is a set of points that are scattered around the origin in all hemispheres of the three-dimensional space. However the points on the unit sphere are all positioned on the part of the sphere where the z-coordinates are positive. Therefore, the corresponding points are divided by a certain constant which positions the eyes behind, and positions the nose tip in front of the x-y plane. The calculated points are then drawn as smaller yellow circles on the second response window<sup>17</sup>.

The algorithm can now compare and translate the two set of points onto each other. Let the first set of points be points of the face mesh reference, and the second set of points the corresponding points that are scattered around the origo. The comparison of the two sets is defined as translating the second set of points to the first. This process can be done by the following equation:

$$B = R * A + t$$

where ‘B’, is the set of mesh reference points, ‘A’ is the set of corresponding points, ‘R’ is the matrix of rotation around the three major axes, and ‘t’ is the vector of translation[6].

The model of this algorithm is created such that it does not consider the vector of translation in the cursor movement. The cursor movement will always be calculated assuming that the recognized face is somewhere at the center of the camera view. The vector of translation is essentially the previously calculated centroid of the three

---

<sup>17</sup>QuteCursor/headPoseCursor.cpp, lines 183-197

face points in the screen coordinate system.

The steps of calculating the centroids and translating both set of points around the origin is done during the previous two steps - it is assumed that the face mesh reference points (read from the file) are predefined such that they already have this characteristic.

The next step is accumulate a three-by-three matrix based on the multiplication of the matching mesh points and corresponding points. The accumulation is defined by the equation on Figure 3.10.

The matrix is accumulated by creating three different matrices for the three matching points, filling up the matrices with the necessary data (multiplying the points in their vector form), and adding the three matrices together<sup>18</sup>.

Multiplying the two matching vectors is calculated by the following function:

```
void returnMatrixOfTwoVectors(Mat &mat, Point3d mesh, Point3d face)
{
    Mat_<double> ret = Mat_<double>::ones(3, 3);
    ret(0, 0) = mesh.x * face.x;
    ret(0, 1) = mesh.x * face.y;
    ret(0, 2) = mesh.x * face.z;
    ret(1, 0) = mesh.y * face.x;
    ret(1, 1) = mesh.y * face.y;
    ret(1, 2) = mesh.y * face.z;
    ret(2, 0) = mesh.z * face.x;
    ret(2, 1) = mesh.z * face.y;
    ret(2, 2) = mesh.z * face.z;
    mat = ret;
}
```

The accumulated matrix is then used with the Single Value Decomposition to find out the matrix of rotation, which is detailed in the Algorithm chapter.

It is stated that the SVD algorithm might return a reflection matrix, which mathematically makes sense, however in reality, it does not. The reflection matrix can be avoided by checking the signature of the matrix's determinant, and multiply

---

<sup>18</sup>QuteCursor/headPoseCursor.cpp, lines 215-221

the last row by -1, if it is negative. This makes sure that the determinant of the matrix of rotation is always positive<sup>19</sup>.

The rotations around the three major axes can be calculated by the following functions:

$$\alpha = R_x = \tan^{-1}(r_{21}/r_{11})$$

$$\beta = R_y = \tan^{-1}(-r_{31}/\sqrt{(r_{32}^2 + r_{33}^2)})$$

$$\gamma = R_z = \tan^{-1}(r_{32}/r_{33})$$

given that  $R^{3 \times 3}$  is:

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

The implementation is referenced here<sup>20</sup>.

The mouse cursor position is calculated based on the different angles of rotation around the three major axes. The angles of deviation are explained in the Algorithm chapter.

At this stage, the head position estimation is done. Before the mouse cursor is moved, the face mesh reference points are multiplied by the matrix of rotation, and drawn on the second response window. This allows the user to see a rotating mesh of face that corresponds to the users head position movements, and have a feedback on how their head position is interpreted by the algorithm.

```
Mat_<double> meshContainer;
Point3d mesh3d;
Point drawablePoint;
Scalar deepblue(143, 72, 21);
for (int i = 0; i < 49; i++)
{
    meshContainer = matrixOf_Rotation * Mat_<double>(meshFacePoints[i]);
    mesh3d = Point3d(meshContainer.at<double>(0, 0),
                     meshContainer.at<double>(0, 1),
                     meshContainer.at<double>(0, 2));
}
```

<sup>19</sup>QuteCursor/headPoseCursor.cpp, lines 225-229

<sup>20</sup>QuteCursor/headPoseCursor.cpp, lines 231-233

```

drawablePoint = returnDrawablePoint(mesh3d, 1, 320, 240);
circle(bg, drawablePoint, 2, deepblue, 2, 0, 0);
}

```

The last step of the algorithm is to move the mouse cursor.

Figures 3.11 and 3.12 show how the values of rotation on each of the three major axes are interpreted. The interpretation was based on experimentation.

During the developmental testing of the algorithm, the algorithm produced calculations with bigger uncertainties than expected. These uncertainties changed with every analyzed frame, and resulted in "jumping" cursor movement. To avoid the hectic jumps of the cursor based on the head position, a moving average is taken by the previous and the currently calculated cursor position. This modification was suggested by Zoltán Tősér, and greatly improved the reliability of the cursor position.

The two halves of the screen are handled separately, as it was found out that the angles of rotation on the x-axis below and above the plane of horizon are not symmetrical (as seen on Figure 3.11). The *setCursorPos* function receives its input parameters as integers of the calculated two-dimensional point. The cursor position is set by the following implementation:

```

POINT currentCursorPos;
GetCursorPos(&currentCursorPos);

double move_on_y;
double move_on_x;

if ((abs(rotationRadiansAroundZAxis) < 0.20))
{
    leftMouseButtonReleaseEvent();
}
else
{
    //cout << "left click" << endl;
}

```

```

    leftMouseButtonDownEvent();
}

if (rotationRadiansAroundXAxis < 0)
{
    move_on_y = 2386.86 * -rotationRadiansAroundXAxis + 1370.5;
}
else
{
    move_on_y = 2741 * -rotationRadiansAroundXAxis + 1370.5;
}

move_on_x = 16522.1 * rotationRadiansAroundYAxis + 1440;

Point2d smoothTransition(0.85*currentCursorPos.x + 0.15*move_on_x,
                          0.85*currentCursorPos.y + 0.15*move_on_y );

SetCursorPos((int)smoothTransition.x, (int)smoothTransition.y);

```

An extra function included in the algorithm is the ability to hold down the left mouse button by head gestures. If the recognized face is tilted by more than a certain angle along the z-axis, then the left mouse button is held down<sup>21</sup>.

Since these functions do not always work properly, they are left as developmental functions.

At the very end, the algorithm shows the processed data on the two feedback windows<sup>22</sup>.

If the camera window is closed, then the function stops, shows messages on the console, and returns with the value of 0, indicating normal termination<sup>23</sup>.

### **Implementation of the graphical user interface:**

The graphical user interface for the head pose estimation program was created using Qt 5.8. framework. It is designed to give a user-friendly control for the camera

<sup>21</sup>QuteCursor/developmentalFunctions.cpp, lines 63-85

<sup>22</sup>QuteCursor/headPoseCursor.cpp, lines 291-296

<sup>23</sup>QuteCursor/headPoseCursor.cpp, lines 299-311

calibration and the head position estimation mouse cursor control, without having to restart the application.

The main window of the application consists of a menu bar, and a panel which holds the user manual along with three buttons for initializing the calibration, initializing the head position estimation with mouse cursor control, and a button which handles the window width for a more compact view. The menu bar essentially holds menu items, which reflect the functions of the buttons. Both the menu items and the buttons on the user interface can be accessed with the keyboard.

The logic of this project was created with CMake, however, Qt uses QMake, which is a different kind of build file. Due to this, a new project is created with the Qt 5.8 framework.

To use OpenCV libraries within a Qt project, OpenCV must be built, which can be done using CMake and Microsoft Visual Studio 2015. The following official guide gives a great explanation of how OpenCV can be set up for a Qt project.

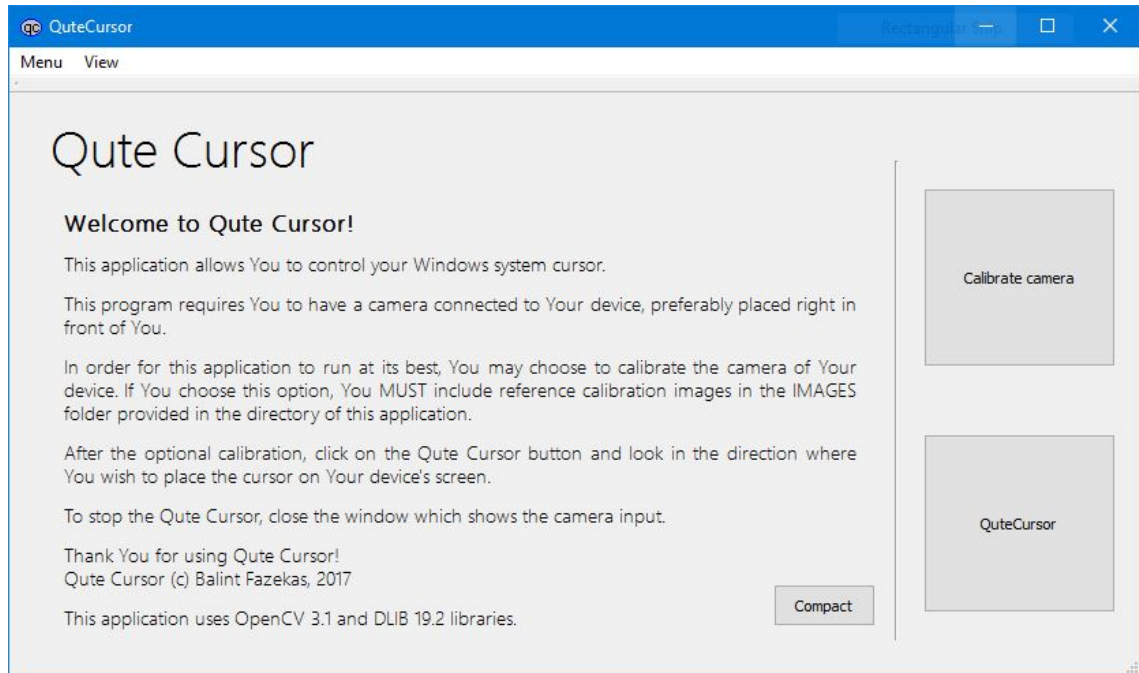
[https://wiki.qt.io/How\\_to\\_setup\\_Qt\\_and\\_opencv\\_on\\_Windows](https://wiki.qt.io/How_to_setup_Qt_and_opencv_on_Windows)

Once OpenCV is compiled, a new QWidgets application can be created using the Qt framework. The guide (available at the link above) outlines how the *.pro* file of the Qt project should be modified so that the OpenCV libraries are included. The *.pro* file also have to include the Dlib libraries. The guide on the link below tells how it should be modified in a few short steps, to include Dlib libraries in the Qt project. <http://valeriobiscione.com/tag/qt/>

Once the necessary libraries are included, the user interface can be created for the logical layer of the project. When creating a new project in the Qt framework, the project is initialized with five files:

1. a *.pro* file, which is Qt's makefile
2. a *.ui* file, which defines the user interface layout in xml format
3. a *main.cpp* sourcefile, which is responsible for running the graphical user interface application
4. a *.cpp* sourcefile and a *.h* headerfile, which control the connection between the user interface and the logical layer, that is already complete.

Double clicking the `.ui` (user interface) file within Qt framework, allows visual, easy to understand modification of the main window of our application. The design of the interface is created:



The menu bar is used to add manu items which help the navigation of the application. Three buttons are added for a more convenient use. The title and the user manual are written inside QPanels, that allows RichText input and basic text formatting. To add functionality to any of the buttons, simply right click on a specific button, and select the “Go to slot...” option. This takes us to the `.cpp` source file, and creates a function for when the button is clicked.

To make the editing and management of the Qt project easier, the already existing header and source files of the logic layer are copied in the directory of the Qt project. The files are then included in the project by right clicking on the Sources folder, and selecting “Add existing files...”. The header files can be included analogically.

The signatures of the functions for each clicked button on the user interface, and a private boolean value is added to the header of the original C++ source file. The “headPoseCursor.h” header file is also included so that the methods in the logic layer can be used<sup>24</sup>.

The clicked buttons and the triggered menu items run one of the four action

---

<sup>24</sup>QuteCursor/qutecursor.h



methods.

The quit action asks the user to confirm exiting the application<sup>25</sup>.

A message box with standard “Yes” and “No” buttons is defined. If the user clicks the yes option, then the program terminates. The first parameter in the constructor of the message box refers to the parent window, which the message box is placed relative to.

The compact view action method changes the dimensions of the main window based on the private boolean value of the class. If it is set smaller, then both calibration and head pose estimation buttons are disabled. If the window is made bigger, then the visibility of the calibration button is checked, and set appropriately. The user is allowed to change the view during calibration, where the calibration disables the calibration button. If the visibility of the button would not be checked before setting it, then the user could potentially run the calibration several times, while the calibration is still in progress.

The function, which is responsible for running the calibration, disables the calibration button during the calibration, and resets it after it is done.

```
void QuteCursor::calibrateAction()
{
    if (confirmCalibration(this->window()))
    {
        ui->calibrate->setText("Calibrating...");
        ui->calibrate->setDisabled(true);
        ui->menuMenu->setDisabled(true);
        runCameraCalib(this->window());
        ui->calibrate->setText("Calibrate camera");
        ui->calibrate->setEnabled(true);
        ui->menuMenu->setEnabled(true);
    }
}
```

The method that controls initiating the head position estimation hides the main window, and resets it once the camera window – opened in the head position esti-

---

<sup>25</sup>QuteCursor/qutecursor.cpp, lines 68-76

mation process – is closed.

```
void QuteCursor::quteCursorAction()
{
    ui->cursor->setDisabled(true);
    this->hide();
    runHeadCursor(this->window());
    this->show();
    ui->cursor->setEnabled(true);
}
```

To make the application more user friendly, a messages header and source file is created that contains the possible error messages and notifications, which the user might come across during the running of the application. The methods within this source file are made such that they all accept an input parameter. This input parameter is the parent window. Without indicating the parent window, the opened message boxes would be placed to an uncertain position on the screen. The parent window parameter makes sure that the opened windows are always placed relative to the main window. The message functions can be found here<sup>26</sup>.

### 3.2.4 Testing

For this project, the Google C++ Test API is used to test the functions in the logic layer. To use the GTest API in a Visual Studio project, right click on the project, select *"Manage NuGet Packages"*, search and download Google Test. After it is downloaded, the Google Test API is automatically available to use for testing. To use the Google Test API in a source code, the *"gtest/gtest.h"* header file must be included.

The test cases written for the head pose estimation are checking that the functions used in the process are working correctly. These test cases are placed in a separate project named as *HeadPoseCursorTest*.

Each test method requires a name and a case, which are to be given as parameters. The name defines a package for the cases. An example of a test function can

---

<sup>26</sup>QuteCursor/messages.cpp, lines 8-43

be seen below:

```
TEST(testReadFromFile, faceMesh)
{
    ifstream input("pose_mean_shape");
    ASSERT_TRUE(input.is_open());
    Point3d points[49];
    EXPECT_DOUBLE_EQ(49 * sizeof(Point3d), sizeof(points));
    returnMeshPointsFromAFile(points);
    input.close();
    ASSERT_FALSE(input.is_open());
}
```

The ASSERT criteria requires to comply with the tests expectations, otherwise it breaks the test procedure. The EXPECT returns if the checked condition is met. Unlike the ASSERT criteria, the EXPECT methods continue to run the test, even if the condition they check is not satisfied.

The tests are ran within the main function of the test project by the following code:

```
#include "tests.h"
using namespace std;

int main(int argc, char *argv[])
{
    ::testing::InitGoogleTest(&argc, argv);
    RUN_ALL_TESTS();
}
```

The following list of test cases were defined and implemented: The following tests were created to check the maximum values of two floating point numbers:

- ☐ first number greater
- ☐ second number greater
- ☐ equal numbers

- ☐ negative numbers
- ☐ one negative, the other is positive
- ☐ one positive, the other is negative

The following tests check the accuracy of removing the pixel units from three-dimensional points:

- ☐ Points at the origin, with the magnifying scale of 1
- ☐ Points at the origin, with the magnifying scale of 5
- ☐ Point located in the first quartile of the Descartes coordinate system
- ☐ Point located in the second quartile of the Descartes coordiante system
- ☐ Point located in the first quartile of the Descartes coordinate system, with different focal lengths

The following tests check the accuracy of determining the three-dimensional distance of three-dimensional points from the origin, or between two points:

- ☐ Distance of the origin from the origin
- ☐ Distance of a point located in the first quartile of the Descartes coordinate system from the origin
- ☐ Distance of a point located in the second quartile of the Descartes coordiante system from the origin
- ☐ Distance of a point located in the third quartile of the Descartes coordiante system from the origin
- ☐ Distance of a point located in the fourth quartile of the Descartes coordiante system from the origin
- ☐ Distance between two points that are located in the first and second quartiles of the Descartes coordinate system
- ☐ Distance between two points that are located in the third and fourth quartiles of the Descartes coordinate system

These following tests check the accuracy of calculating the centroid of three three-dimensional points:

- ☐ One-dimensional centroid of points of a 60-60-60 degrees triangle around the origin
- ☐ Two-dimensional centroid of points of a 60-60-60 degrees triangle around the origin
- ☐ Three-dimensional centroid of points of a 60-60-60 degrees triangle around the origin
- ☐ Three-dimensional centroid of points of a 60-60-60 degrees triangle around the origin, in a three-dimensional space

These following test cases check the validity of calculating the matrix product of two vectors:

- ☐ Matrix filled with zeros
- ☐ Matrix filled with ones
- ☐ Matrix filled with the product of the indices

Additional tests:

- ☐ Display points around the origin in the center of the screen coordinate system (relative to a window)
- ☐ Check the sum of the angles between three points that form a triangle
- ☐ Check the angle of rotation on the main axes from a face mesh that is rotated 90 degrees along the y-axis
- ☐ Checking the validity of reading face mesh points from a file
- ☐ Checking the validity of writing a default camera matrix file
- ☐ Checking the validity of writing an error file

The source code of the test cases is referenced here<sup>27</sup>. The following images show the result of the previously described test cases:

---

<sup>27</sup>QuteCursorTest/tests.cpp

```
C:\WINDOWS\system32\cmd.exe

**** VIDEOINPUT LIBRARY - 0.1995 - TFW07 ****

[=====] Running 31 tests from 12 test cases.
[-----] Global test environment set-up.
[-----] 6 tests from testMaximumOfTwoValues
[ RUN ] testMaximumOfTwoValues.firstGreater
[ OK ] testMaximumOfTwoValues.firstGreater (0 ms)
[ RUN ] testMaximumOfTwoValues.secondGreater
[ OK ] testMaximumOfTwoValues.secondGreater (0 ms)
[ RUN ] testMaximumOfTwoValues.equalNumbers
[ OK ] testMaximumOfTwoValues.equalNumbers (0 ms)
[ RUN ] testMaximumOfTwoValues.negativeNumbers
[ OK ] testMaximumOfTwoValues.negativeNumbers (0 ms)
[ RUN ] testMaximumOfTwoValues.oneNegativeOnePositive
[ OK ] testMaximumOfTwoValues.oneNegativeOnePositive (0 ms)
[ RUN ] testMaximumOfTwoValues.onePositiveOneNegative
[ OK ] testMaximumOfTwoValues.onePositiveOneNegative (0 ms)
[-----] 6 tests from testMaximumOfTwoValues (2 ms total)

[-----] 5 tests from testUnitRemovedPoints
[ RUN ] testUnitRemovedPoints.origoPoints_1
[ OK ] testUnitRemovedPoints.origoPoints_1 (0 ms)
[ RUN ] testUnitRemovedPoints.origoPoints_2
[ OK ] testUnitRemovedPoints.origoPoints_2 (0 ms)
[ RUN ] testUnitRemovedPoints.aroundOrigo_1
[ OK ] testUnitRemovedPoints.aroundOrigo_1 (0 ms)
[ RUN ] testUnitRemovedPoints.aroundOrigo_2
[ OK ] testUnitRemovedPoints.aroundOrigo_2 (0 ms)
[ RUN ] testUnitRemovedPoints.aroundOrigo_3
[ OK ] testUnitRemovedPoints.aroundOrigo_3 (0 ms)
[-----] 5 tests from testUnitRemovedPoints (1 ms total)

[-----] 5 tests from testThreeDimensionalDistance1
[ RUN ] testThreeDimensionalDistance1.origoPoint
[ OK ] testThreeDimensionalDistance1.origoPoint (0 ms)
[ RUN ] testThreeDimensionalDistance1.firstQuartile
[ OK ] testThreeDimensionalDistance1.firstQuartile (0 ms)
[ RUN ] testThreeDimensionalDistance1.secondQuartile
[ OK ] testThreeDimensionalDistance1.secondQuartile (0 ms)
[ RUN ] testThreeDimensionalDistance1.thirdQuartile
[ OK ] testThreeDimensionalDistance1.thirdQuartile (0 ms)
[ RUN ] testThreeDimensionalDistance1.fourthQuartile
[ OK ] testThreeDimensionalDistance1.fourthQuartile (0 ms)
[-----] 5 tests from testThreeDimensionalDistance1 (4 ms total)

[-----] 2 tests from testThreeDimensionalDistance2
[ RUN ] testThreeDimensionalDistance2.firstAndSecondQuartile
[ OK ] testThreeDimensionalDistance2.firstAndSecondQuartile (0 ms)
[ RUN ] testThreeDimensionalDistance2.thirdAndFourthQuartile
[ OK ] testThreeDimensionalDistance2.thirdAndFourthQuartile (0 ms)
[-----] 2 tests from testThreeDimensionalDistance2 (2 ms total)

[-----] 4 tests from testCalculateCentroid
[ RUN ] testCalculateCentroid.aroundOrigo1
[ OK ] testCalculateCentroid.aroundOrigo1 (0 ms)
[ RUN ] testCalculateCentroid.aroundOrigo2
[ OK ] testCalculateCentroid.aroundOrigo2 (0 ms)
[ RUN ] testCalculateCentroid.aroundOrigo3
[ OK ] testCalculateCentroid.aroundOrigo3 (0 ms)
[ RUN ] testCalculateCentroid.aroundOrigo4
[ OK ] testCalculateCentroid.aroundOrigo4 (0 ms)
[-----] 4 tests from testCalculateCentroid (5 ms total)
```

Figure 3.14: Result of the test cases

```
C:\WINDOWS\system32\cmd.exe

[-----] 3 tests from testMatrixOfTwoVectors
[ RUN    ] testMatrixOfTwoVectors.zeros
[ OK     ] testMatrixOfTwoVectors.zeros (1 ms)
[ RUN    ] testMatrixOfTwoVectors.ones
[ OK     ] testMatrixOfTwoVectors.ones (0 ms)
[ RUN    ] testMatrixOfTwoVectors.sqaured
[ OK     ] testMatrixOfTwoVectors.sqaured (0 ms)
[-----] 3 tests from testMatrixOfTwoVectors (2 ms total)

[-----] 1 test from testDrawablePoints
[ RUN    ] testDrawablePoints.window
[ OK     ] testDrawablePoints.window (159 ms)
[-----] 1 test from testDrawablePoints (159 ms total)

[-----] 1 test from testFacePose
[ RUN    ] testFacePose.first
[ OK     ] testFacePose.first (231 ms)
[-----] 1 test from testFacePose (231 ms total)

[-----] 1 test from testRotateTwoSetOfPoints
[ RUN    ] testRotateTwoSetOfPoints.alongYAxis
[ OK     ] testRotateTwoSetOfPoints.alongYAxis (885 ms)
[-----] 1 test from testRotateTwoSetOfPoints (887 ms total)

[-----] 1 test from testReadFromFile
[ RUN    ] testReadFromFile.faceMesh
[ OK     ] testReadFromFile.faceMesh (1 ms)
[-----] 1 test from testReadFromFile (1 ms total)

[-----] 1 test from testWriteMatrixToFile
[ RUN    ] testWriteMatrixToFile.defaultMatrix
[ OK     ] testWriteMatrixToFile.defaultMatrix (11 ms)
[-----] 1 test from testWriteMatrixToFile (11 ms total)

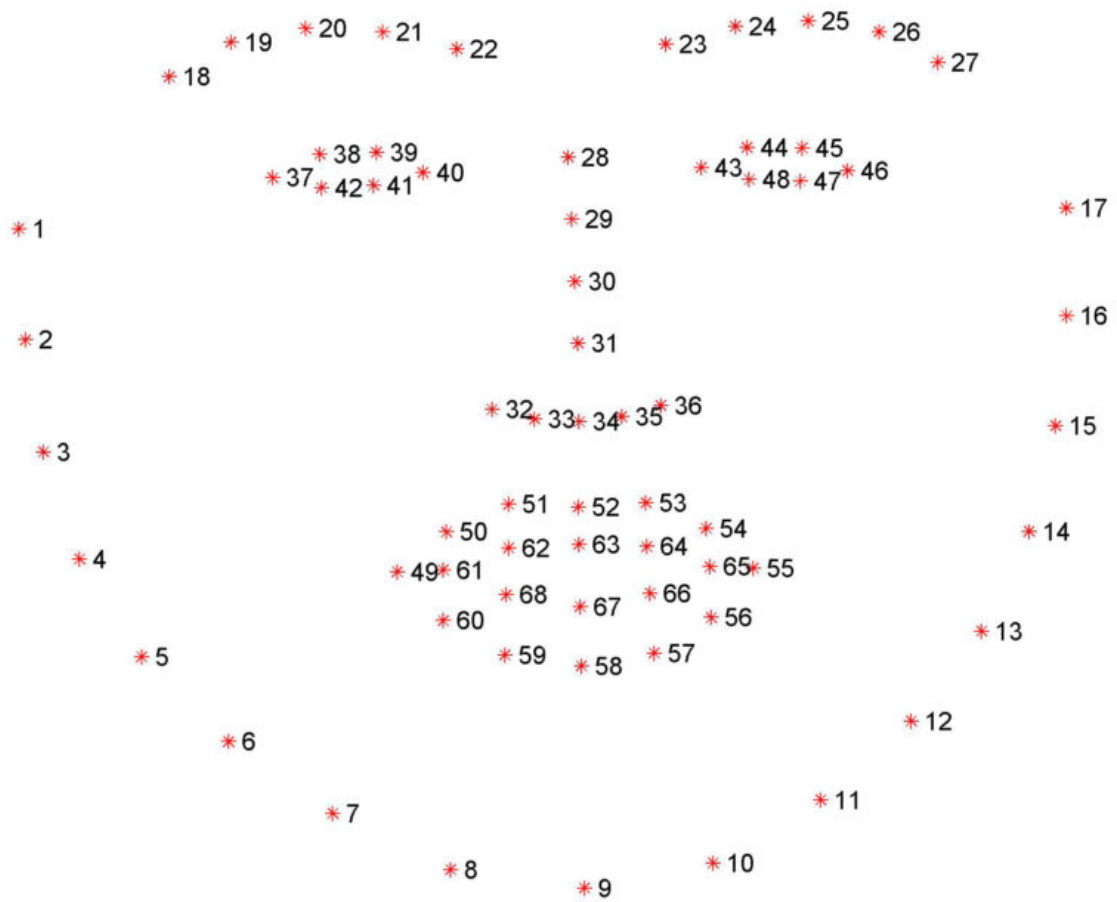
[-----] 1 test from testWriteErrorFile
[ RUN    ] testWriteErrorFile.errorFile
[ OK     ] testWriteErrorFile.errorFile (10 ms)
[-----] 1 test from testWriteErrorFile (12 ms total)

[-----] Global test environment tear-down
[=====] 31 tests from 12 test cases ran. (1329 ms total)
[ PASSED ] 31 tests.
Press any key to continue . . .
```

Figure 3.15: Result of the test cases



# Appendix



Mapping of the 68 points used by Dlib's face recognizer [\[12\]](#)

# Bibliography

- [1] H.M. L. Sensorama simulator, August 28 1962. US Patent 3,050,870.
- [2] Morton Heilig. The cinema of the future. *Translated by Uri Feldman. In Multimedia: From Wagner to Virtual Reality. Edited by Randall Packer and Ken Jordan. Expanded ed. New York: WW Norton*, pages 239–251, 2002.
- [3] Virtual reality – definition. Available at: <https://www.merriam-webster.com/dictionary/virtual%20reality>.
- [4] Xiao-Shan Gao, Xiao-Rong Hou, Jianliang Tang, and Hang-Fei Cheng. Complete solution classification for the perspective-three-point problem. *IEEE transactions on pattern analysis and machine intelligence*, 25(8):930–943, 2003.
- [5] The p3p (perspective-three-point) principle, May 2012. Available at: <http://iplimage.com/blog/p3p-perspective-point-overview/>.
- [6] Nghia Kiem Ho. Finding optimal rotation and translation between corresponding 3d points, May 2013. Available at: [http://nghiaho.com/?page\\_id=671](http://nghiaho.com/?page_id=671).
- [7] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.
- [8] Class robot, Jan 2016. Available at: <https://docs.oracle.com/javase/7/docs/api/java/awt/Robot.html>.
- [9] Setcursorpos function. Available at: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms648394\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms648394(v=vs.85).aspx).
- [10] Opencv - about. Available at: <http://opencv.org/about.html>.

- [11] Reading and writing images and video. Available at: [http://docs.opencv.org/2.4/modules/highgui/doc/reading\\_and\\_writing\\_images\\_and\\_video.html#videocapture](http://docs.opencv.org/2.4/modules/highgui/doc/reading_and_writing_images_and_video.html#videocapture).
- [12] Facial landmarks with dlib, opencv, and python, Apr 2017. Available at: <http://www.pyimagesearch.com/2017/04/03/facial-landmarks-dlib-opencv-python/>.

All online sites were last accessed on: July 6, 2017.

# Acknowledgements

I would like to emphasize my gratefulness towards Zoltán Tóser, my supervisor for this thesis, who always gave a guiding hand when I encountered an algorithmic or technical problem.