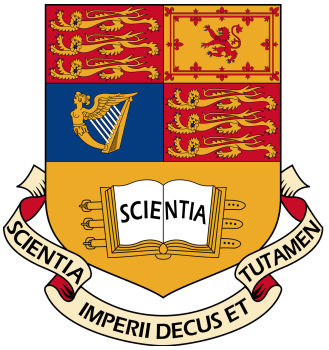

Travail de Fin d'Etudes

3D gaze interpolation in an immersive data visualization environment

Data Science Institute of Imperial College London

Student :
Andrianirina Rakotoharisoa

Supervisors :
Jean-Yves Martin, Centrale Nantes
Florian Guitton, Data Science Institute
Dr Ovidiu Șerban, Data Science Institute

Abstract

Being able to forecast the judgement of human experts when they visualise rich, heavy data is a field of research that remains relevant and thoroughly investigated today. Gaze estimation is the first step to understand how the brain works. Processes such as Machine Learning and Artificial Intelligence can take over once we have analyzed which data are important. This document will describe a method of gaze estimation based on face orientation in an immersive data visualization environment. This method has been implemented using a set of kinects. We will combine a face recognition algorithm to the body skeleton detector built in the kinect to construct our estimation. Further experiments will then be conducted to improve the software and results will be analyzed in a dedicated section. The issues tackled are the performance of our software, its robustness and its accuracy.

Acknowledgement

I am very grateful to Yike Guo and Florian Guitton for offering me the possibility to accomplish my internship at the Data Science Institute (DSI). The enthusiasm they have demonstrated since the beginning of my internship has been heart-warming and has really helped me to fit in quietly.

My deepest appreciation also goes to the GDO team. Ovidiu Serban, Senaka Fernando and James Scott-Brown have always been available when I had questions and their advices often helped shed new light upon my project.

The Data Science Institute has really been a stimulating place to work in, with many people passionate about their work. The glimpse I had when talking with researchers really awakened my curiosity. It made me try to educate myself on research fields that were not directly related to mine. Off work, the conversations I had were quite often enlightening. For this reason, I would like to extend my gratitude to the DSI and all its people, academic and non-academic. This internship has created possibilities that I probably wouldn't have discovered in the industry and brought a very welcomed diversity in my relations.

Contents

1 Introduction

Machine Learning and Artificial Intelligence are the source of numerous breakthrough and enabled various innovations. Though today we use Machine and Deep Learning in a large number of technologies (e.g. self-driving car, face recognition for mobile application), in other fields the judgement of a human specialist is often unmatched and still essential. For example, in medicine though artificial intelligence can assist the doctor, it's ultimately the human who establishes the medical diagnosis.

Data collection is required before being able to use Machine Learning techniques. In my project, the data we want to collect are provided by human interaction with machines, i.e. when the human specialist inspects information displayed on screens. This means that these data are indirect: we can only collect them by recording and analyzing the activity of people watching the screens. The main objective is thus to be able to interpolate the gaze of the people and ultimately learn which data can be defined as relevant and which are left aside.

For this project we will use a set of kinects located around the visualization environment which will capture the position of each person. Our software will then estimate the position of the gaze on the screens based on the position of the body and the face orientation.

2 Context

2.1 Imperial College of London

Imperial College (ICL) is a public research university specialized in science and medicine. It was founded in 1907 by royal charter. It is the result of the fusion of the Royal College of Sciences, the Royal School of Mines and City and Guilds of London Institute. Each year, 17000 students and 8000 staff members work and study in the university. More than 125 nationalities are represented.

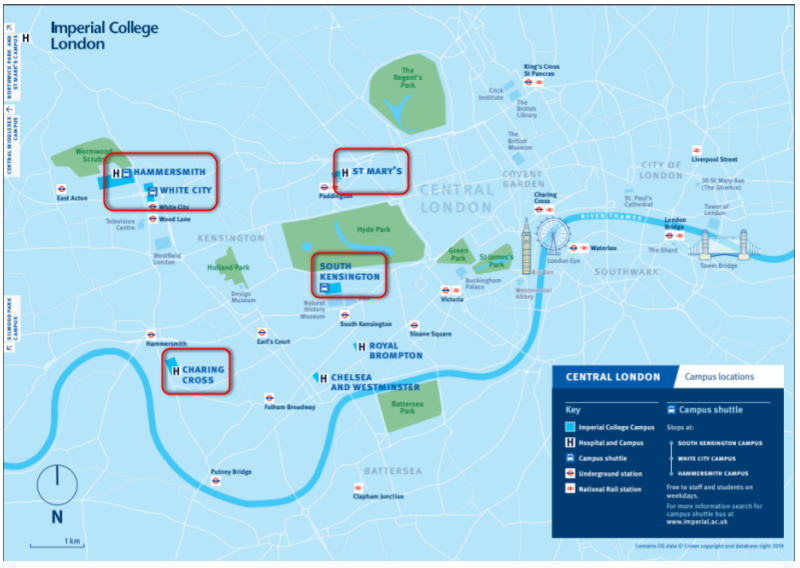

Figure 1: The campuses of Imperial College

The five campuses that welcome students are Hammersmith, Charing Cross, White City, St Mary's and South Kensington. My internship took place at the South Kensington campus.

Located in a touristic area of South Kensington, the main entrance (see Figure 2) opens onto Exhibition Road. In this neighbourhood, one can also find museums (the Science Museum, the Natural History Museum and the Victoria and Albert Museum) and other famous institutions such as the Royal Albert Hall and the Royal College of Music.

Figure 2: The main entrance on Exhibition Road

Today, it is organised in four faculties:

* Faculty of Engineering
* Faculty of Natural Sciences
* Faculty of Medicine
* Imperial College Business School

Each faculties is then subdivided into departments.

Aside from the faculties, Imperial College hosts Global Institutes to promote inter-disciplinary work and research:

- Grantham Institute for Climate Change
- Institute of Global Health Innovation
- Energy Futures Lab
- Institute for Security Science and Technology
- Institute for Molecular Science and Engineering
- **Data Science Institute**

The goal of these institutes is to work on and try to bring answers to critical issues and problematics that can't be tackled by individual academics or teams. In these institutes, researchers of different background work together on global challenges.

2.2 The Data Science Institute

Located in the William Penny lab, the Data Science Institute (DSI) tackles issues that vary from health inequalities and the dangers of global warming, to the opportunities created by big data and molecular engineering: the research themes of the institute are respectively Analytics, Bio-Medical Informatics Platforms, DataLearning, Image Informatics and **Visualization**.

These themes are developed, and thereby studies led, in seven multidisciplinary labs. In these labs, different technologies and innovations are developed and investigated through collaborations between different fields.

The different labs are the following:

- Algorithmic Society Lab
  - ⇒ Mission: Provide mechanisms and algorithms that allow to analyze behavioural large data-set while respecting privacy
- Behavioural Analytics Lab
  - ⇒ Mission: Understand and predict human and biological behaviour
- Business Analytics Lab
  - ⇒ Mission: Study how business analytics, data, and artificial intelligence will change business and society
- Data Assimilation Lab
  - ⇒ Mission: Assimilate data into advanced theoretical models and apply these to science and engineering
- Data Economy Lab
  - ⇒ Mission: Carry out research into the economic, legal and policy mechanisms required for the emerging Data Economy which consists of markets where the currencies are huge datasets
- Machine Learning Lab
  - ⇒ Mission: Develop autonomous decision-making systems, which close the perception-action-learning loop while learning from small amounts of data
- Social and Cultural Analytics Lab
  - ⇒ Mission: Study the evolution of music, social networks, online collaborations to discover how culture and society works

2.3 The Global Data Observatory

Opened in 2015, the Global Data Observatory (GDO) is an immersive data visualization environment. It provides opportunities for industry and academics to investigates visual and complex data in an open and multi-dimensional environment.

Figure 3: The Global Data Observatory

The GDO team is composed of a system architect, Senaka Fernando, a research associate, Dr. James Scott-Brown and a research fellow-Head of GDO, Dr. Ovidiu Șerban, whose work in this team focus on large-scale visualization. For the purpose of this project, my work is also followed and advised by my supervisor, Florian Guitton, research associate.

The GDO is composed of sixty-four screens, each one having a resolution of 1920*1080 pixels. The screens display the information sent by the computers. There are two displaying modes: the cluster mode and the section mode.

The Section mode is the default mode. On Section mode, four computers control twelve screens (green circles) and one computer controls the last sixteen screens (red circle) (see Figure 4).

On cluster mode, each computer handles two screens. In this configuration, the amount of data processed is increased and the computational capacity of the GDO is more important : it can be useful for very complex data and allows more adaptability.

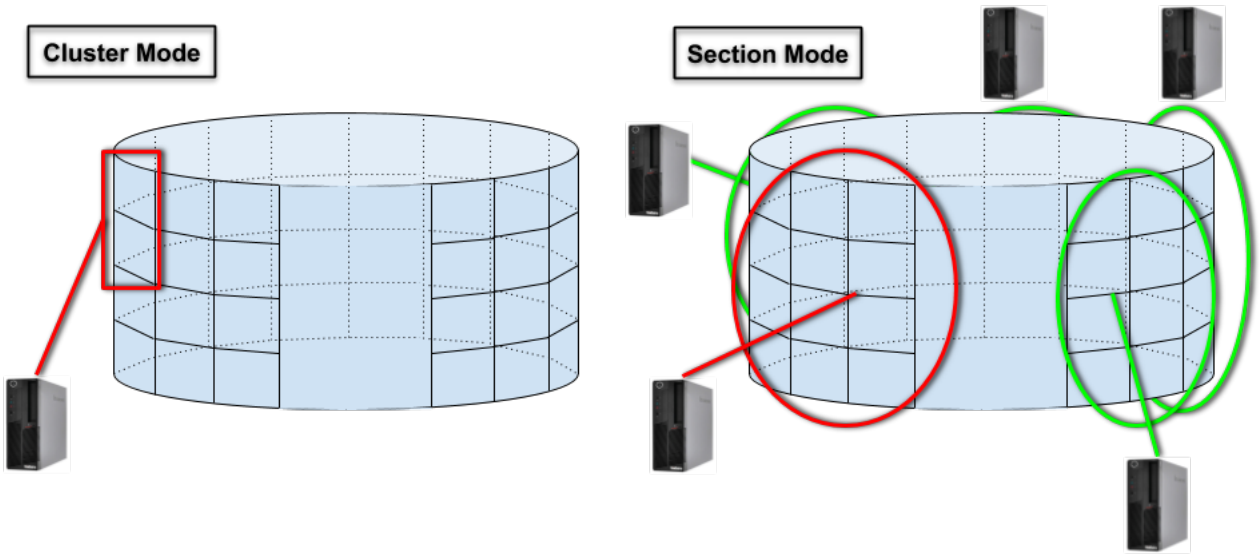

Figure 4: The two display mode

For both configuration, the kinects are placed above the screens (see Figure 5) and each camera feed goes to a different computer. The synchronization of the data takes place once all the feeds are received and then the information is sent to the server.

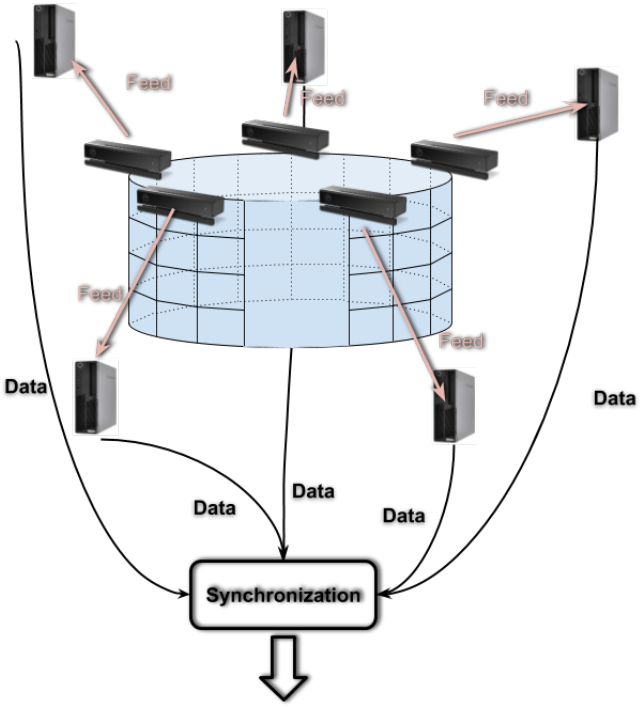

Figure 5: Kinects disposition in the GDO

To develop my software and execute my experiments, I used a development room. In this room, we have a replica of the GDO, the Data Observatory Development (DO-Dev): eight screens, which have a resolution of 3840*2160 pixels, and one kinect (see Figure 6). Unlike the GDO, each screen is monitored by a different computer.

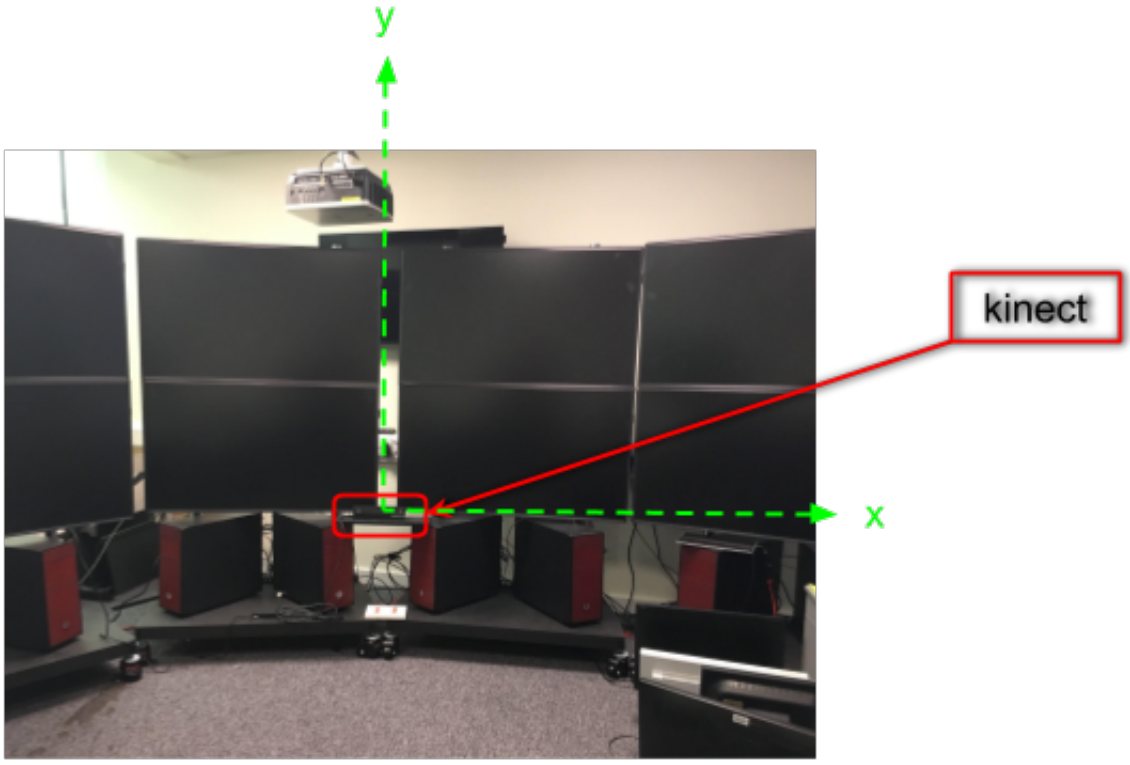

Figure 6: Disposition of the DO-Dev

3 Objective

3.1 Constraints

The main objective of this internship was the creation of a software that enables gaze estimation based on body posture and facial orientation.

The main constraints were the following:

- Adapt and work with the GDO team to bring something compliant with the new disposition and useful for possible future projects
- Base my project, when possible, on the results obtained from the previous project 4.1 of D. Birch and L.Chao

I had the choice of the language and the technology that would be used and was free to manage the project as I intended to.

3.2 Scope statement

In order to reach our objective, we had to design a software that would be able to estimate the gaze of GDO users in multiple configurations.

→ The user(s) must have the possibility to:

- move across the GDO
- turn their body
- turn their head

→ It can be one user or people interacting:

- The software must handle up to six people [1]
- The number of people using the GDO can change during the use of the software

To check the accuracy of the estimator, we have to be able to see in real time the estimations displayed on the GDO:

- The estimations must be easy to follow on screen
- The delay between the the real gaze and the estimation displayed on screen must not be too long

The Figure 7 below sums up the tasks that our software must be able to realize:

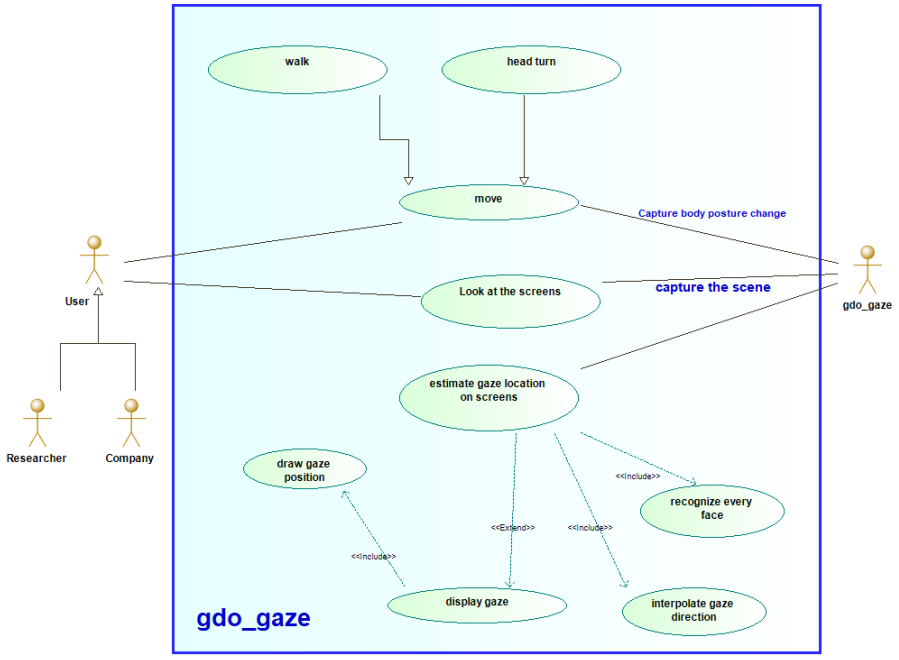

Figure 7: Use case diagram of the gdo_gaze project

[1] The kinect can only detect up to six people at the same time

4 State of the art

4.1 Previous studies

The GDO-Holokinect was a project created by Lu Chao and David Birch (1). This project originated the setup of the kinects seen in Figure 5. The aim of this project was also to create a software capable of estimating the gaze of people using the GDO.

After the synchronization and the fusion of all the frames provided by the five kinects, as seen in Figure 8, the gaze was estimated from the body and face orientations. The face orientation was obtained from a vector already provided by the kinect.

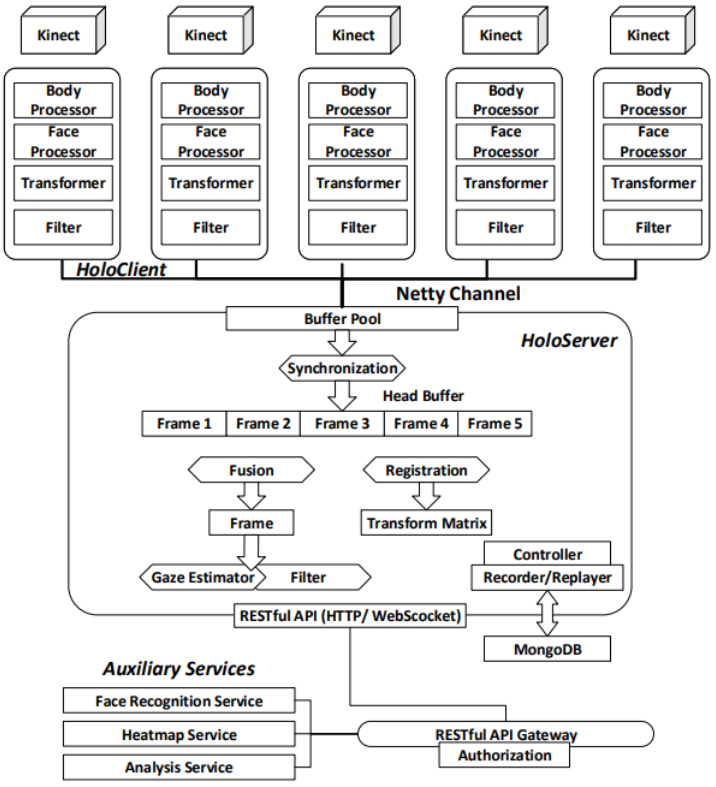

Figure 8: The architecture of the GDO-Holokinect project as published in (1)

The work from Alberto, Mora and Ordobez took a different approach (2). In this paper, the face is the center of the study. Body identification or orientation are not mentioned whereas face pose and eye orientation are thoroughly investigated (see Figure 9).
They use face models trained by neural networks (3) to approximate the face pose. After a normalization to obtain a frontal image of the face, the eye-region is then cropped and the direction is inferred through adaptive linear regression.

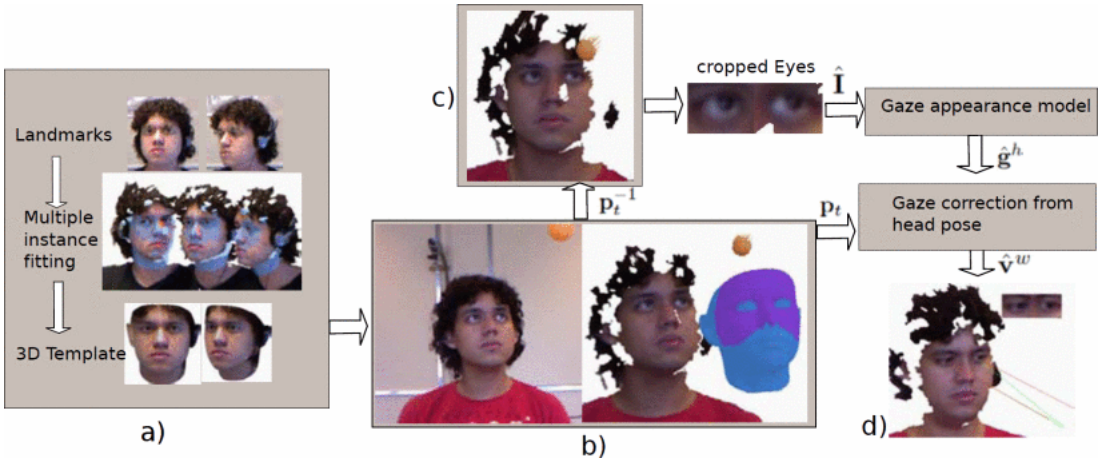

Figure 9: Method used for gaze estimation as published in their paper

This approach is more precise but also more difficult to scale to our visualization environment. In the GDO, people can move, therefore the major obstacle would be the robustness of the face fitting algorithm: a face occlusion would prevent the face fitting to take place. Nonetheless, I took the idea of using a face detection algorithm instead of the built-in kinect quaternion (a vector used to calculate the orientation of the x, y, z axis of the kinect) to improve the detection of face orientation. Study the pupil position in the eye region is also an interesting idea that could improve the accuracy of the gaze estimation.

4.2 Technology

4.2.1 Kinect v2

For this project we've decided to use the Xbox One Kinect Sensor. It was practical as this device combines a depth space representation, a 2D camera and an integrated skeleton detector. Moreover it is cheap and is already used in the previous project: therefore my work could reuse what has already been done before. One of the main downside of this technology is that it's not on sale anymore, therefore developing a technology using the kinect can seem risky.

Depth detection

To create a 3D representation of the space, the Kinect device uses the Time-Of-Flight ranging system. It requires three Infra Red (IR) projectors and one IR camera (see Figure 10). The depth data are then stored in an array, sized like the color frame captured by the Color camera of the Kinect. A mapping between the depth data and the pixels of the color frame takes place to form the 3D representation.

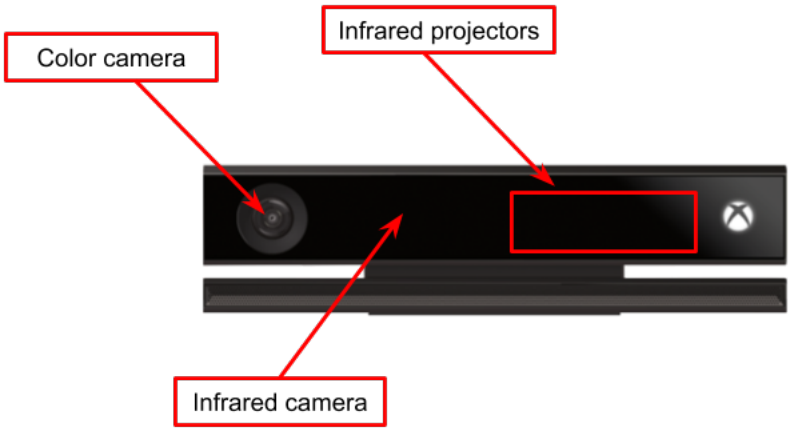

Figure 10: Kinect V2 structure

4.2.2 Language and libraries

Although for its majority the previous project of L. Chao and D. Birch has not been implemented into this language, we have decided with my supervisors to pursue my project in python. I already had some experience coding in this language thanks to my second year internship and many face recognition libraries and kinect APIs were already implemented in python.

I will present here some of the most important libraries and APIs that were of help during my research:

dlib:

dlib is a famous cross-platform software initially developed in C++ containing machine learning algorithms. In image processing, an important section of the library has been established in object detection. The *frontal face_detector* algorithm has been my first introduction to facial recognition.

This module allows the user to take an image as an input and returns the image with the recognised faces (see Figure 11).

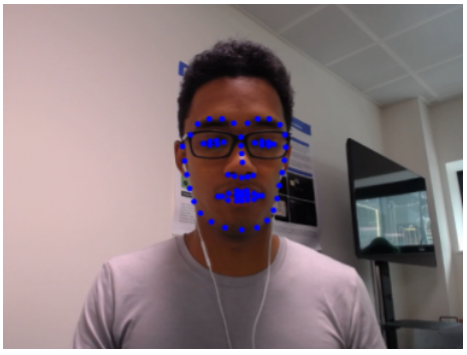

Figure 11: Face recognition with face landmarks

As we can see in the above figure, a face is recognised with *landmarks*. A landmark is a feature that can be detected after a Convolutional Neural Network (CNN) or Histogram of Oriented Gradients (HOG) training and that characterizes the object we want to detect, here a face.

This face detector has been used with the pre-trained model shape_predictor_68_face_landmarks.dat The idea of working with face landmarks to recognise a face and to estimate the gaze is a ground idea I have used in my project.

face_alignment:

The face_alignment library by Adrian Bulat (4), is a face recognition and face alignment library that overall produces the same results as the dlib face_detector section. The face alignment is obtained after a CNN training over a dataset of 230000 facial landmarks.

The main difference is that this library can also provide 3D face landmarks from a 2 dimensional image, which was a significant step forward as we want to project the gaze of a person into a three dimensional space (see Figure 12).

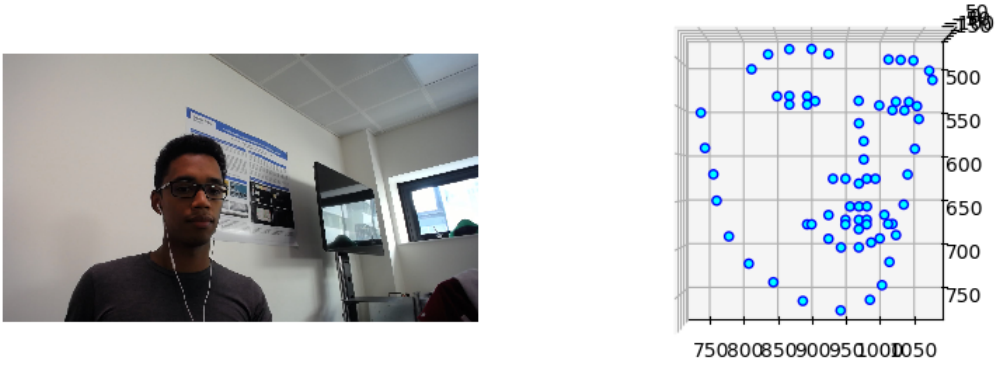

Figure 12: 3D landmarks detected by the face_alignment library

The x and y coordinates we see on the Figure above are the coordinates of the two dimensional pixel frame. The z coordinate, or depth coordinate, is deduced from the CNN training of the model based on the face landmarks dataset.

One other aspect that I took into account while using this library, was the possibility to bring some modifications. Unlike the dlib library which was originally implemented in C++, this library was originally implemented in python. This way I was able to try to improve the performance of the face alignment method (see 7.2.1).

PyKinect2:

PyKinect2 is a kinect python wrapper for the windows kinect SDK (Software Development Kit) that enables the user to access all the kinects' functions easily (see Figure 13).

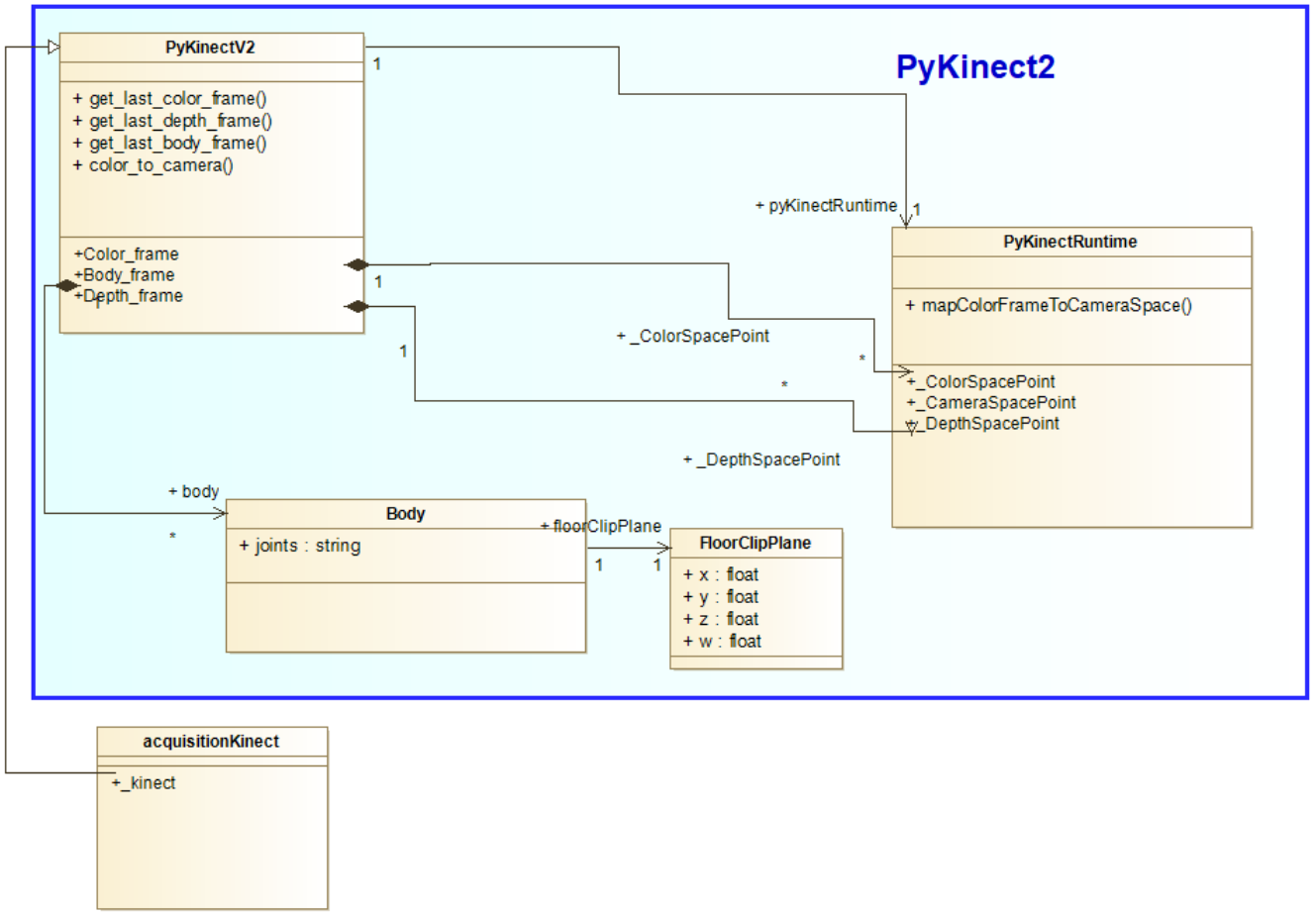

Figure 13: Simplified PyKinect2 Class Diagram

This wrapper allows us to acquire the depth, color and body frames as well as the body orientation for each detected skeleton. The FloorClipPlane present for each body in the diagram, is a quaternion indicating the orientation of the body in regard to the camera. It is by using this quaternion that we deduced the tilt angle of the kinect. By supposing in fact that the floor is horizontal, we are able to correct the oriented vision of the kinect (see Figure 14).

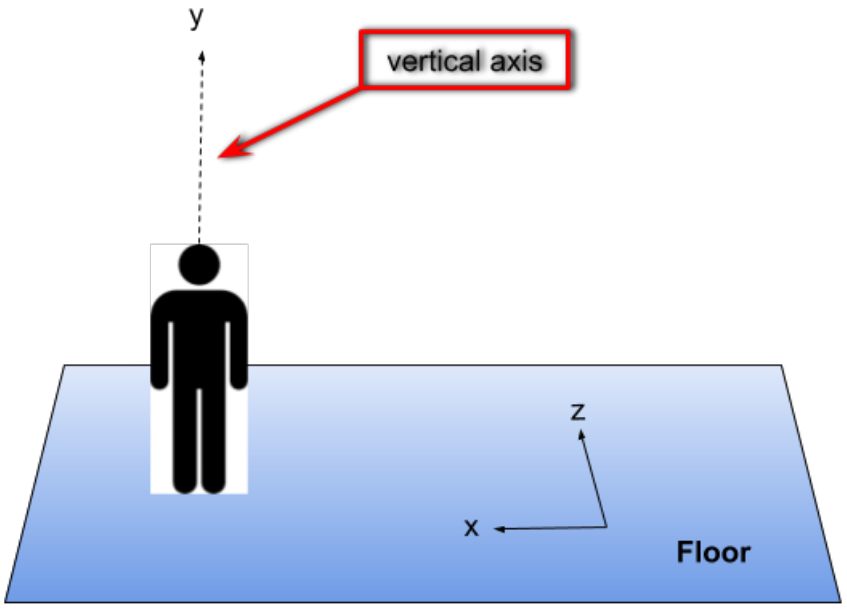

Figure 14: Finding a horizontal xz plane with the help of body orientation

We see on the figure above that by supposing that the vertical axis y can be inferred from the position of the person, the floor will then be a plan normal to this vertical vector and therefore a plan for the x and y vectors.

We created another class too, acquisitionKinect, so that the PyKinect2 wrapper remained unchanged. We brought all the modifications needed directly on the acquisitionKinect class.

CUDA and Pytorch:

CUDA is a parallel platform computation that enables us to use our GPU for general process computations. Pytorch is a library designed for computer vision amongst other things.

We will use Pytorch to modify our face recognition algorithm and try to improve our software. We will do so by moving most of the heavy computations from the CPU to the GPU, using the fact that Pytorch can be CUDA enabled.

5 Models

5.1 The coordinate systems

Each frame owns a particular coordinate system. It is possible to switch from one coordinate system to another by using a coordinate mapper. To obtain real world coordinates, we have to combine the information coming from the depth frame and from the color frame

- Color coordinates

  The color frame is a 1920 * 1080 matrix of pixels. A pixel is referenced by two coordinates x and y corresponding to the pixel position.

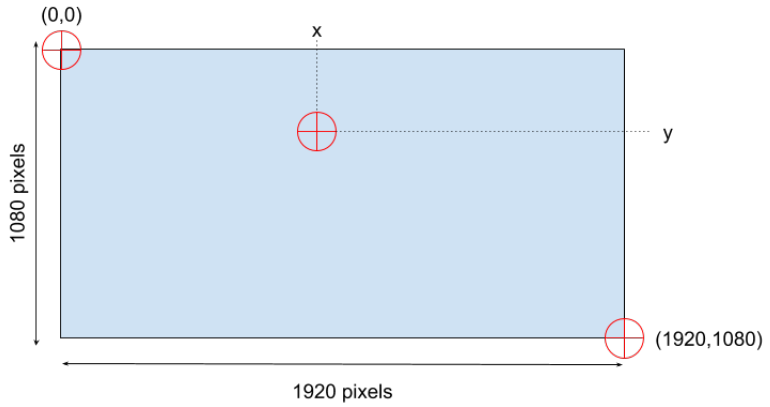

Figure 15: Pixels coordinates in the color frame

Each pixel contains the RGB information of the color frame at this position.

- Depth coordinates

  The depth frame resolution is 512*424 and the depth information is stored into a list of integers. This list of integers has been reshaped into a matrix of size 512*424.

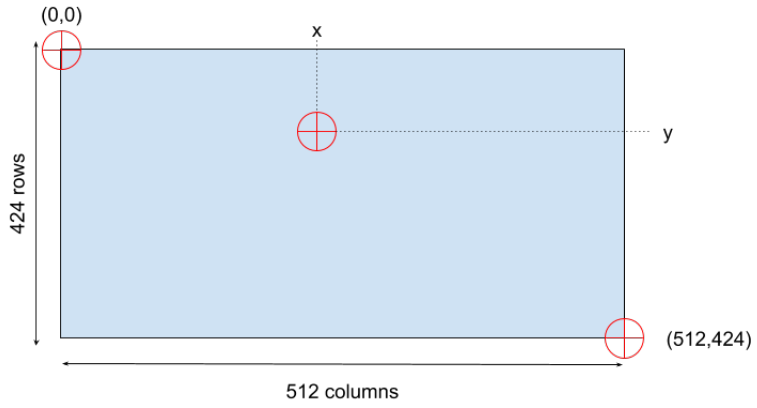

Figure 16: Depth coordinates in the depth frame

In opposition to the color frame, the depth frame doesn't contain RGB pixels but the depth of each element in meters.

- Camera coordinates

  The Camera coordinates are obtained by mapping [2] the color coordinates with the depth coordinates into a 1920*1080 resolution matrix (see Figure 17).

  After the mapping has occurred, the kinect is able to give the coordinates of every points of the color frame in meters. The origin is located on the color camera (see Figure 18).

[2] see Appendice A

$$\underbrace{\underbrace{\begin{bmatrix}(100,150,79) & \dots & \dots & \dots & \dots \\ \vdots & \ddots & \dots & \dots & \dots \\ \vdots & \dots & \ddots & \dots & \dots \\ \vdots & \dots & \dots & \ddots & \dots \\ (100,250,79) & \dots & \dots & \dots & \dots \end{bmatrix}}_{\text{1920*1080 pixel image}}}_{\text{Color frame}} + \underbrace{\underbrace{\begin{bmatrix}(4.759) & \dots & \dots & \dots & \dots \\ \vdots & \ddots & \dots & \dots & \dots \\ \vdots & \dots & \ddots & \dots & \dots \\ \vdots & \dots & \dots & \ddots & \dots \\ (5.018) & \dots & \dots & \dots & \dots \end{bmatrix}}_{\text{512*424 matrix}}}_{\text{Depth frame}}$$

Mapping

$$\underbrace{\begin{bmatrix}(-4.100,2.750,4.759) & \dots & \dots & \dots & \dots \\ \vdots & \ddots & \dots & \dots & \dots \\ \vdots & \dots & \ddots & \dots & \dots \\ \vdots & \dots & \dots & \ddots & \dots \\ (-4.100,2.750,5.018) & \dots & \dots & \dots & \dots \end{bmatrix}}_{\text{1920*1080 coordinates matrix}}$$
Camera space coordinates matrix

Figure 17: Mapping between the depth frame and the color frame

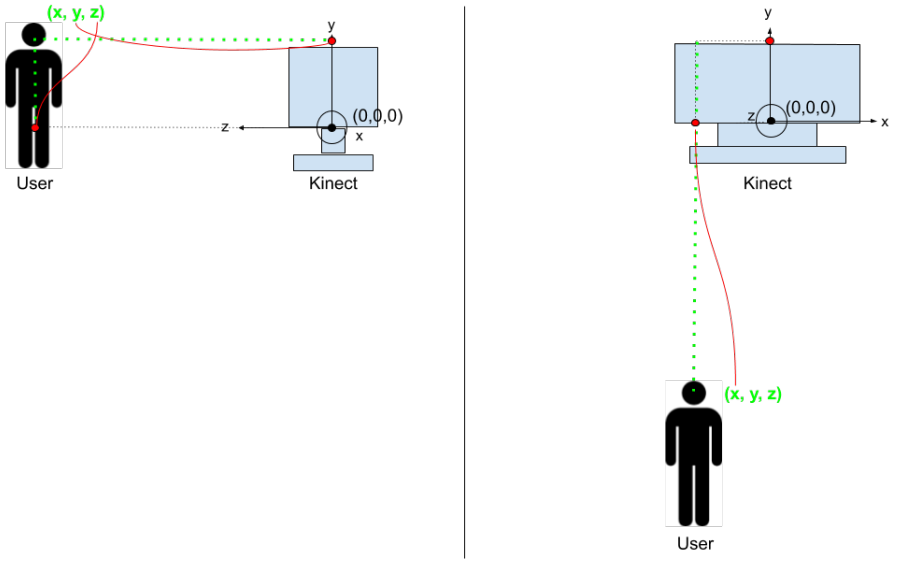

Figure 18: Camera coordinates after the mapping between depth and color coordinates

To correct errors inherent to the position of the kinect, the body frame provides a quaternion that detects the kinect angles based on the floor plan orientation.

Although we will assume that the kinect is locally parallel to the screens, we will take into account the tilt angle as it can significantly affect our estimations.

Influence of the tilt angle

Initial situation:

Let's suppose that the kinect has a tilt angle θ of 10 degrees and the person is located two meters away from the kinect. The kinect is located h = 0.70m above the ground (see Figure 19).

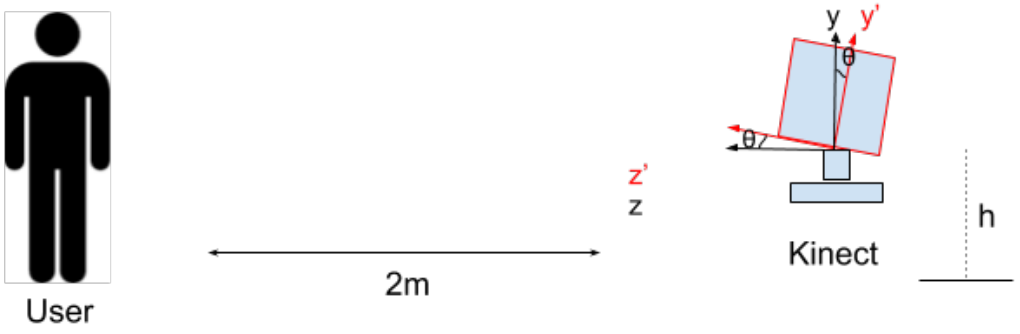

Figure 19: Tilt angle of a kinect

If we assume the person's height to be of 1.80m (so its eyes should be y_{eye} = **1.70m** high), the real coordinates of one eye should be:

$$eye_{real} = (x_{left/right}, y_{eye} - h, 2.00)$$

i.e $eye_{real} = (x_{left/right}, 1.00, 2.00)$

The x coordinate won't change due to the tilt angle so we didn't give it any value.

Having a tilt angle θ, we get the following rotation matrix:

$$R_\theta = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{bmatrix}$$

Therefore, the coordinates calculated by the kinect are the following:

$$eye_{kinect} = R_\theta * eye_{real} \tag{1}$$

In our example and after the calculation of (1), $eye_{kinect} = (x_{left/right}, 0.63, 2.14)$

If we consider the problem to be bi-dimensionnal and put aside the x coordinate, the relative error we get is:

$$error_{tilt} = \frac{\left\| eye^{2D}_{real} - eye^{2D}_{kinect} \right\|^2}{\left\| eye^{2D}_{real} \right\|^2}$$

D.A: $error_{tilt} = 0.07$

The relative error is 7% which means we would only be able to round our space coordinates to the first digit, i.e to the decimeter.

5.2 Architecture

The process of gaze estimation can be separated into two main parts:

- Create a 3D rendering of the face or at least essential facial features, the landmarks
- Interpolate the gaze direction based on face orientation and body posture

Once the estimation is completed, the coordinates are sent to the GDO machines and displayed on our visualization environment.

Result: JSON file sent to server
while *time* < *warmup_duration* **do**
    get Color_frame
    get Depth_frame
    Camera_points = MapColorFrameToCameraCoordinates()
end
while *not (Esc keyboard key)* **do**
    get Color_frame
    get Depth_frame
    Camera_points = MapColorFrameToCameraCoordinates()
    get Body_list
    face_landmarks = face_recognition_algorithm(Color_frame)
    assert length(Body_list) == length(face_landmarks)
    landmarks_3D = Camera_points[face_landmarks]
    face_orientation = get_face_orientation(landmarks_3D)
    message = interpolate_gaze_position(face_orientation, landmarks_3D)
    send_message_to_server(message)
end

Algorithm 1: Estimation of the gaze from face orientation

We added a warm up duration at the beginning of our software so that the infrared sensors of the kinect can be more accurate when capturing the depth frame.
Once the data are sent via a JSON file to the server, a websocket garanties that the GDO machines stay up to date as long as the program runs (see Figure 20).

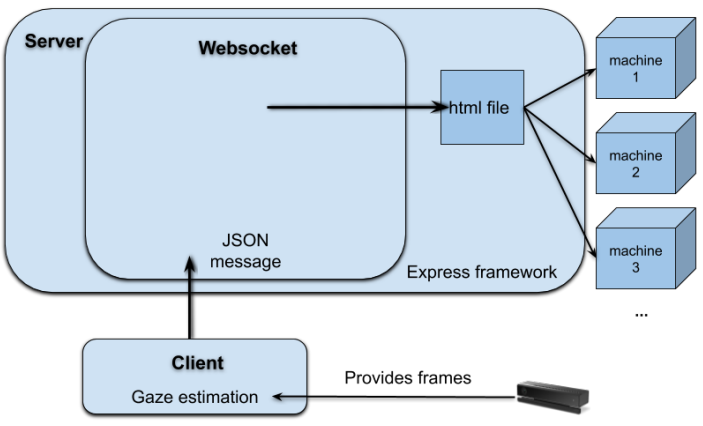

Figure 20: Server architecture

6 Implementation

6.1 Prediction of the gaze orientation

Selection of the library

The first step to collect facial landmarks is by using a face recognition algorithm. The algorithm will be applied on each color frame captured by the color camera of the kinect to detect every face.

We had the choice between two python libraries: dlib and face_alignment developed by A. Bulat.

Initially, these libraries are used on images and not on live stream camera feed. Therefore we had to make sure that this step would not last too long so that the rest of the process could take place in a reasonable amount of time. We tested them on a computer that had an Intel(R) Core(TM) i7-6700HQ CPU and an NVIDIA GeForce GTX 1060 GPU.
All the heavy computations were performed on the CPU for this test.

We summarized all the different characteristics of each algorithm in the following table:

Criteria	**face_alignment**	**dlib**
Speed (*)	0.5s	0.18s
Robustness	Can detect face landmarks even with a face partially hidden or masked (see Figure 23)	Must have an almost frontal face and can't detect an occluded face
Accuracy	Both algorithms are equally accurate (see Figure 22)	

Figure 21: Comparison between dlib and face_alignment

(*): The speed criteria has been determined having only one person in front of the camera and using the CPU to do all the heavy computations. The result is the mean of the time it took to detect the face landmarks for each loop of the software.

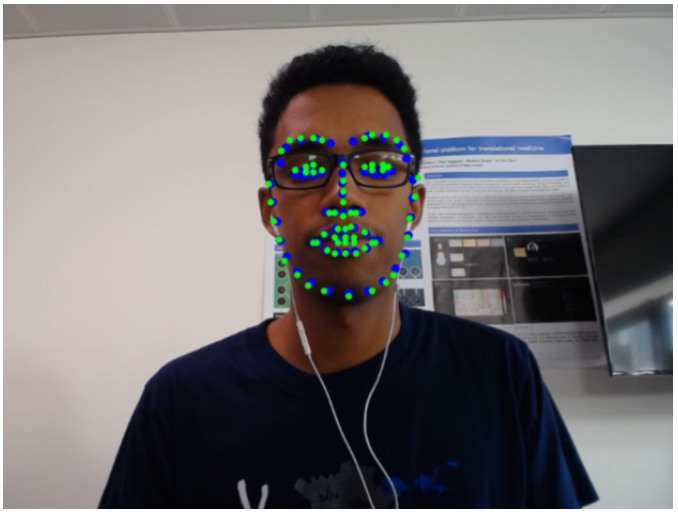

Figure 22: Superposition of landmarks detected by both landmarks

The Figure 22 shows that both libraries provide approximately the same landmarks position on a two dimensional color frame.

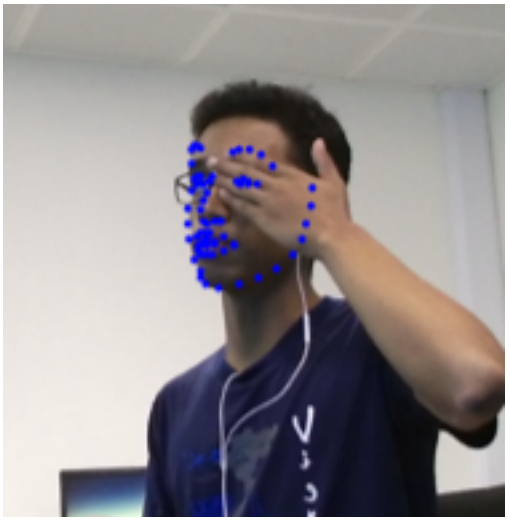

Figure 23: Landmarks detection on a partially hidden face with face_alignement

In the Figure 23, we can see how a face can be detected even when it's partially hidden and not in a frontal position with the face_alignment library.

This difference of robustness will be the decisive factor in our case. We will choose the face_alignment library of A. Bulat instead of the dlib library, even if it is three tenths of a second slower per loop, due to its better robustness and access to its code.

Collect 3D facial features

Once the face alignment algorithm has been executed on the color frame, we have our face landmarks stored into an array. This array contains the coordinates of each landmarks on the color frame. We then select those which are going to be useful to establish the face orientation (see Figure 24).

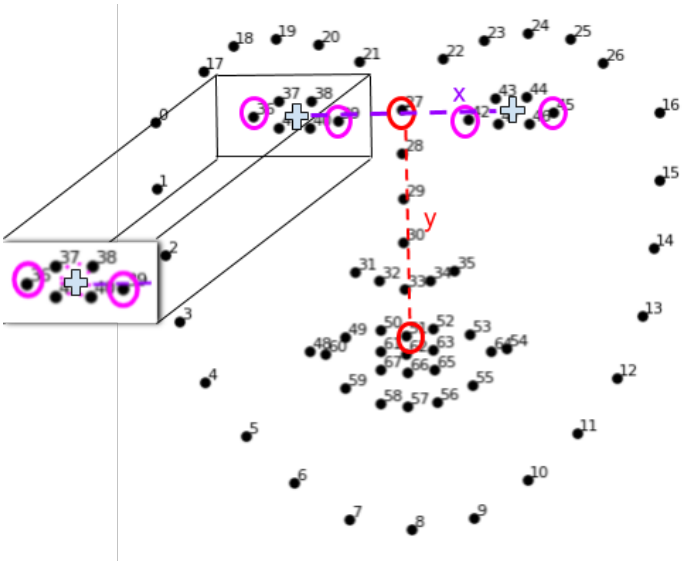

Figure 24: Face landmarks and face plan

The six landmarks we select from the landmarks array are those circled on the Figure 24.

As shown on Algorithm 1, the selected facial landmarks detected on the Color frame are then projected into the camera space coordinates by a mapping between the Color and the Depth frames.

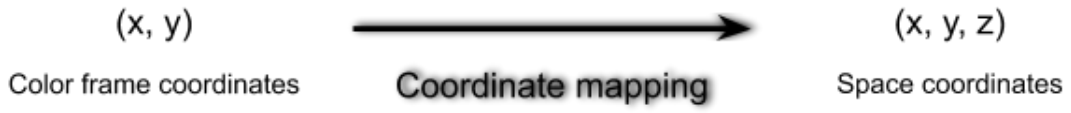

Figure 25: Coordinate mapping

Once we have three dimensional landmarks, The two landmarks circled in red form the dotted red y vector.

The four landmarks circled in purple around the eyes area are used to form the dotted purple x vector. The landmarks 36 and 39 form a pair and 42 and 45 the other pair. The blue crosses represent the middle between each pair of landmarks and what we will suppose to be the location of the pupil of each eye.

$$left\_eye = \frac{left\_eye_{inside} + left\_eye_{outside}}{2} \tag{2}$$

$$right\_eye = \frac{right\_eye_{inside} + right\_eye_{outside}}{2} \tag{3}$$

The vector x is the vector linking the two crosses together.

Interpolate the gaze and estimate the position on screens

The two vectors x and y form a plan and our assumption is that they are orthogonal (see Figure 26).

We deduce the direction of the gaze by creating a vector orthogonal to the face plan: the vector k on the figure above.

Then we make a projection in this direction until the projection encounters a screen, ie until the z_{kinect} coordinate equals 0.

Calculation of the projection coordinates

Let k be the director vector of the gaze, calculated by the method described under the Figure 26.

$$\begin{cases} eye_{coordinates} = (x, y, z) \\ k = (k_x, k_y, k_z) \end{cases}$$

Then we solve the equation:

$$eye_{coordinates} + m * k = (x_{proj}, y_{proj}, 0) \tag{4}$$

With m being the unknown variable.

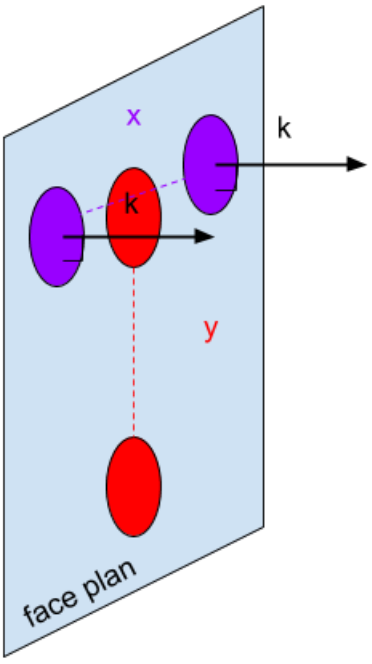

Figure 26: Gaze direction

By reducing the equation (4) to one dimension we get:

$$z + m * k_z = 0 \tag{5}$$

Which gives us: $m = -\frac{z}{k_z}$

Then x_{proj} and y_{proj} are the coordinates of the gaze location on the screens (see Figure 27).

$$\begin{cases} x_{proj} = x + m * k_x \\ y_{proj} = y + m * k_y \end{cases} \tag{6}$$

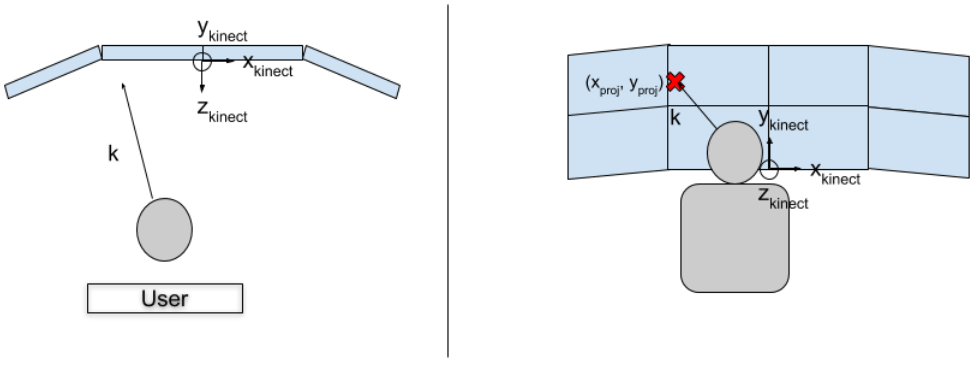

Figure 27: Projection

We can make this projection as every screen is located in the kinect space. We use the top right corner of each screen to get their coordinates (see Figure 28).

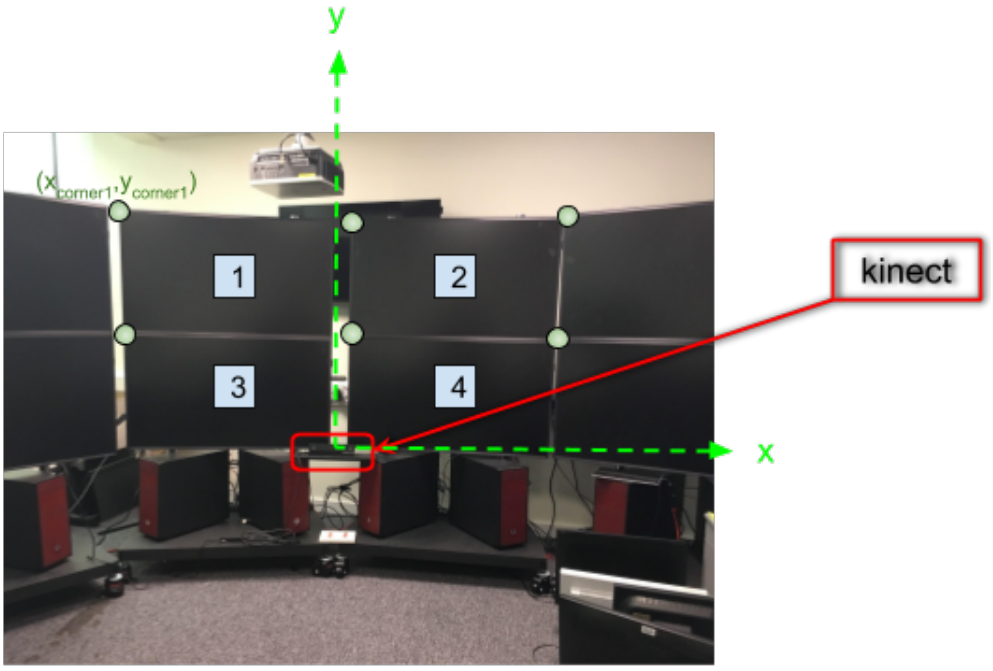

Figure 28: Coordinates of each screen in the DO-Dev

So to get the projection coordinates on every screen we have to remove the offset corresponding to the screen position in the camera space:

- For screen 1, we have $(x_1, y_1) = (x_{proj}, y_{proj}) - (x_{corner1}, y_{corner1})$
- For screen 2, we have $(x_2, y_2) = (x_{proj}, y_{proj}) - (x_{corner2}, y_{corner2})$

  ⋮

Finally we convert the coordinates, which are still in meters, in pixels by the following method:

$$\begin{cases} x_{screen\_i} = \frac{x_i * canva\_x}{screen_{width}} \\ y_{screen\_i} = \frac{y_i * canva\_y}{screen_{height}} \end{cases} \tag{7}$$

In this case, canva_x and canva_y represent the resolution of the screens in pixels and, for every screen, the width is 1.10m and the height 0.633m.

6.2 Multiple face detection

Given that this software will potentially be used by professionals and researchers, we had to make sure that many gazes could be estimated as the same time, so that the GDO could be employed simultaneously by multiple users.

The kinect provides a body tracker and can detect up to six people at the same time. The tracker assigns a number to each body and this number doesn't change when the kinect update the color and depth frames.
To each body detected, the body frame associates a skeleton composed of twenty-five body joints (see Figure 29).

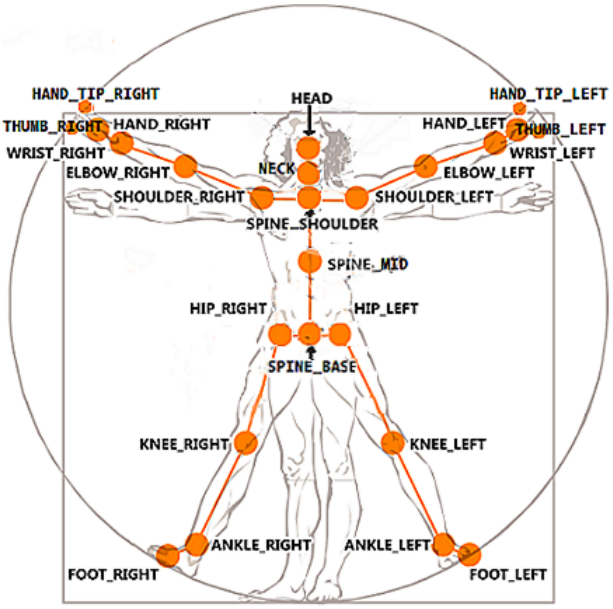

Figure 29: Body joint-skeleton. Image from (5)

```
while not (Esc keyboard key) do
    get Color_frame
    get Depth_frame
    Camera_points = MapColorFrameToCameraCoordinates()
    get Body_list
    face_landmarks = face_recognition_algorithm(Color_frame)
    assert length(Body_list) == length(face_landmarks)
    landmarks_3D = Camera_points[face_landmarks]
    face_orientation = get_face_orientation(landmarks_3D)
    message = interpolate_gaze_position(face_orientation, landmarks_3D)
    send_message_to_server(message)
end
```

Figure 30: Checking the numbers of faces and bodies detected

The test of Algorithm 1 circled in red in the Figure 30 is checking that the number of faces detected by the face recognition library equals the number of bodies detected by the kinect body frame.

Once the number of faces corresponds to the number of bodies, we also have to make sure that every face is linked to the right body. If this association is done correctly, the kinect body tracker then makes sure that even if the faces are not recognised in the same order by the face recognition algorithm, they are still associated to the same people.

We therefore added some steps into Algorithm 1 (see Algorithm 2).

```
Result: JSON file sent to server
while time < warmup_duration do
    get Color_frame
    get Depth_frame
    Camera_points = MapColorFrameToCameraCoordinates()
end
while not (Esc keyboard key) do
    get Color_frame
    get Depth_frame
    Camera_points = MapColorFrameToCameraCoordinates()
    get Body_list
    face_landmarks = face_recognition_algorithm(Color_frame)
    assert length(Body_list) == length(face_landmarks)
    landmarks_3D = Camera_points[face_landmarks]
    for every face detected do
        associate_face_to_body(landmarks_3D, Body_list)
    end
    face_orientation = get_face_orientation(landmarks_3D)
    message = interpolate_gaze_position(face_orientation, landmarks_3D)
    send_message_to_server(message)
end
```

Algorithm 2: Handling multilple users

For each face, we calculate the distance between the face and every body. The face gets the number of the closest body.

```
Result: Body number associated to a face
body_number = 0
dist = distance(Body_list[0], face)
for i in Body_list do
    if distance(Body_list[i], face) < dist then
        body_number = i
    end
end
```

Algorithm 3: Association of a face and a body

To calculate the distance between a skeleton and a face, we use the skeleton-joint associated to the head and the face landmark associated to the nose (see Figure 29 and landmark 30 on Figure 24).

If we define the vector difference as difference $=$ Body_list[i][joint_head] $-$ face_landmark[30], the distance between the face and the Body i is defined by:

$$distance_i = \sqrt{difference.x^2 + difference.y^2 + difference.z^2} \tag{8}$$

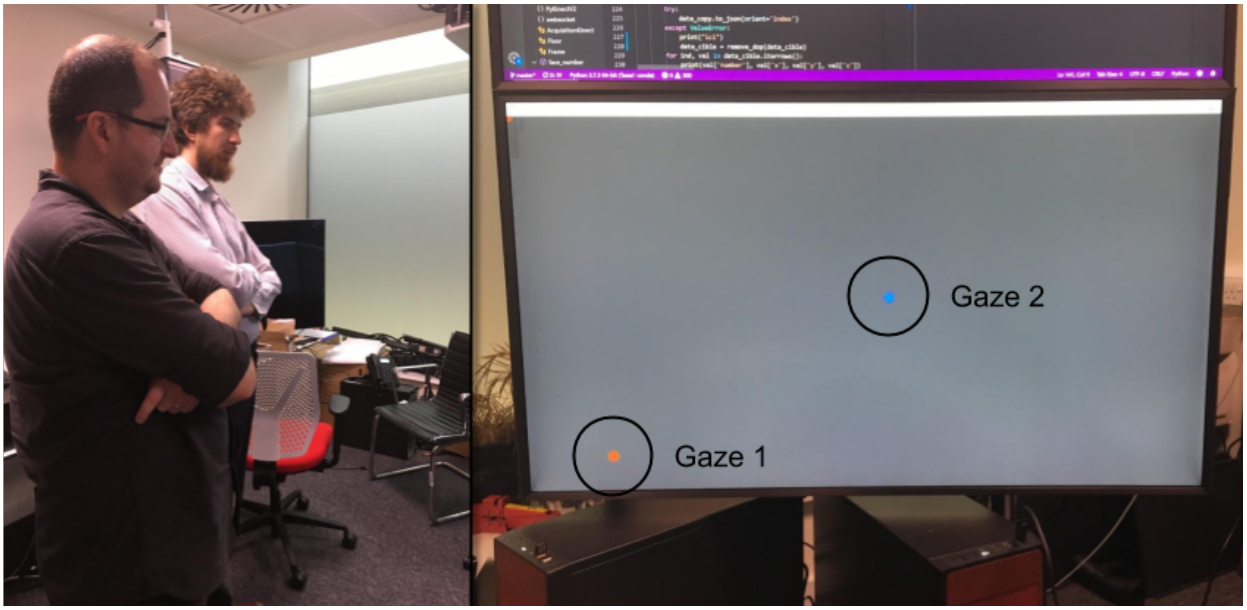

Figure 31: Multiple people using the gaze estimator

Throughout the use of the software, a person will always be associated to the same color (see Figure 31). We assigned a color to every detected person.

Specific Case

It sometimes may happen that the face recognition algorithm recognizes a same face multiple times.
In this particular case, when the same body number is assigned more than once, we remove the clone by keeping only the face which is closer to the body distance-wise.

6.3 Sending the data to the server

We use the express framework to help manage our server.

Our HTML file is being loaded by all the machines of the GDO and is being updated during the software execution.

We use a websocket to send the coordinates calculated from the kinect and to update the html file.

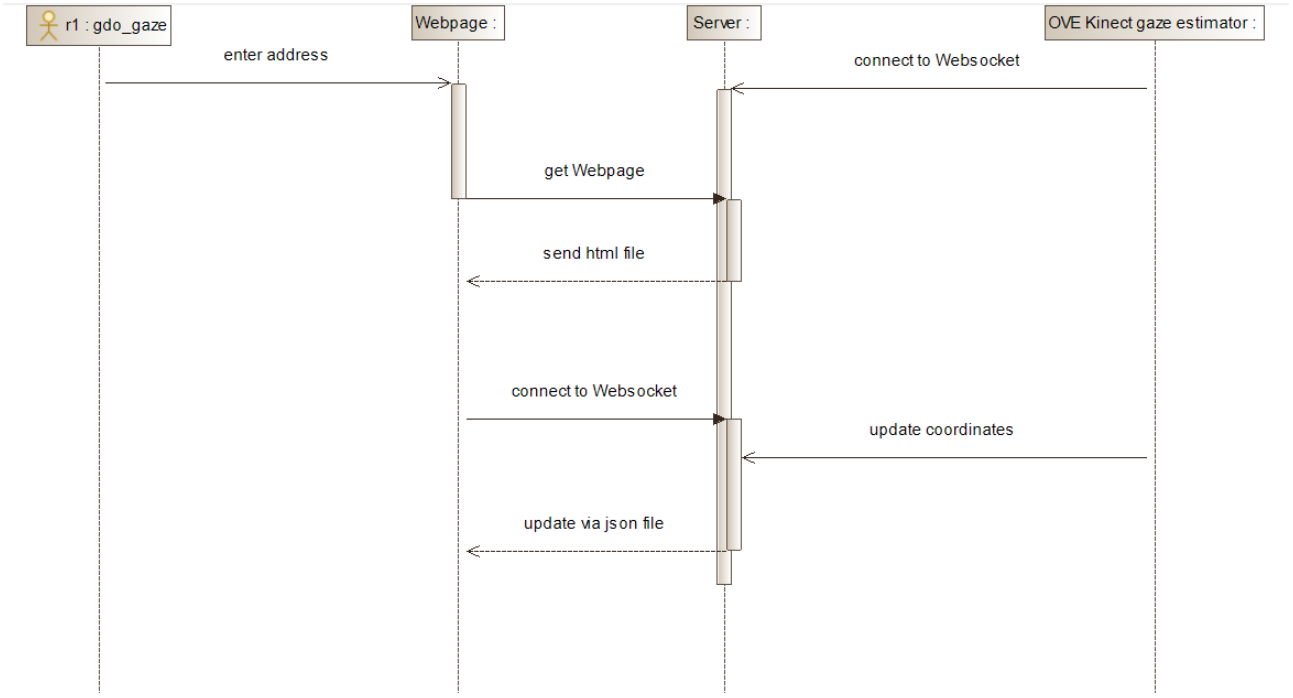

Figure 32: Sequence diagram representing the sending of the coordinates to the GDO machines

As the HTML file is the same for all the machines, we had to find a way to make the screens aware of their coordinates so that the offset specific to each screen could be removed.

We've decided to manually add the x and y coordinates of the top right corner of each screen at the end of the URL:

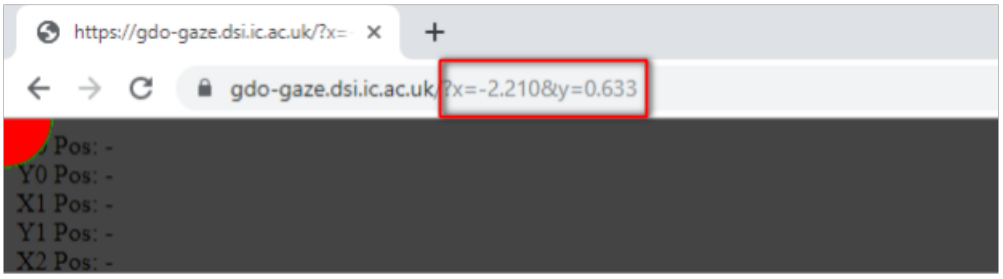

Figure 33: URL with the screen coordinates

7 Experiments

Specification

In all our experiments, the kinect was connected to a computer using an Intel(R) Core(TM) i7-8700K CPU, a NVIDIA GeForce GTX 1080 GPU and having 16GB of RAM.

7.1 Reducing the impact of outliers

One major issue of the first version of my project was the sensitivity to one-time errors, also called outliers. If the face coordinates are inaccurately detected, the gaze prediction can become be inconsistent.

In this part, we don't take the residual errors into account.

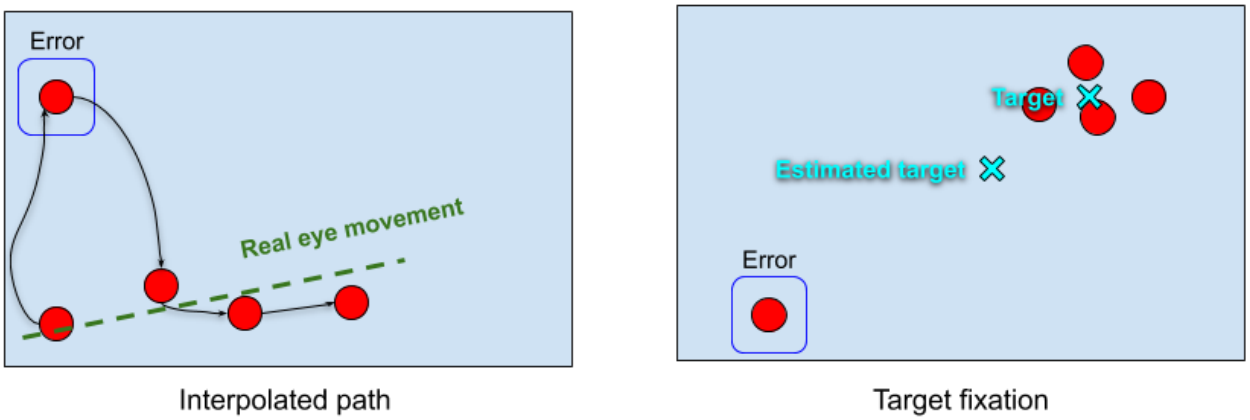

Figure 34: Situations where one-time errors could cause problems

Such inconsistent predictions can bring discomfort to the user who would see his gaze jumping from one part of the screen to another. Moreover, this could also alter later studies if analysis should be conducted based on these datasets.

7.1.1 Protocol

We decided to use moving averages and means to try to smoothen the predictions and to reduce the impact of potential errors.

To compare the influence of moving average on our results, we constructed our data by recording the gaze under three conditions, called modes:

- **Mode 1**: Gaze interpolation based on immediate face landmarks estimation
- **Mode 2**: Average estimation of face landmarks prediction (see Figure 35) (*)
- **Mode 3**: Moving average on gaze estimation coupled with average estimation on face landmarks (see Figure 36) (**)

Expected benefits of Modes 2 and 3:

(*): If the landmarks are totally misplaced after one loop of the face recognition algorithm, the mean will erase or diminish this error as it is unlikely that the face recognition algorithm will fail three times in a row. (**): Adding the moving average on top of the previous process will smoothen the gaze interpolation and make the gaze display more fluid on screen, especially when the task executed by the user will be an exploration of the information displayed on the screens. It will be easier to see the global trends of the eye movement.

For Mode 2, we will make an average based on the last three face landmarks estimations (see Figure 35). It is a good compromise between a fast software and an efficient average.

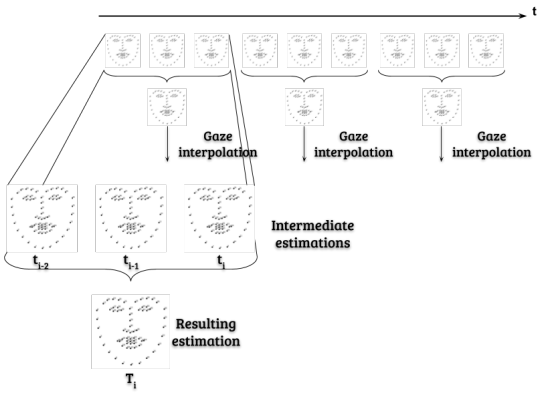

Figure 35: Average estimation on face landmarks prediction

For Mode 3, we decided to have three gaze estimations per window (see Figure 36). Despite the time it takes to make the first interpolation, the loop is quick once we've got our first moving average. Taking only three estimations per windows enables the gaze to be dynamic and not too static [3] as the influences of past estimations decrease fast.

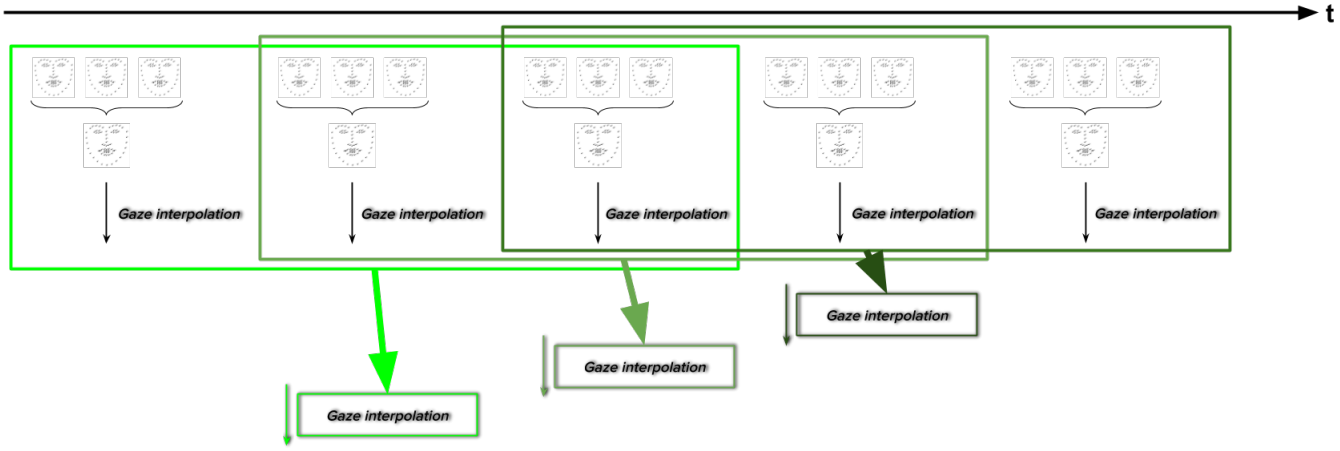

Figure 36: Moving average on gaze estimation coupled with average estimation on face landmarks

All the data were collected from one person.

[3] With 3 estimations per window, a loop takes less than 2seconds. The prediction stays fluid and comfortable for the user to follow.

7.1.2 Experiment

To demonstrate the influence of the two process, we will apply them consecutively on a task of target fixation.

The subject will fix 5 locations on the Data observatory of the Development Room (shortened DO-Dev) (see Figure 37).

Protocol

1. Mode a
   For every target
   • Kinect warm-up: 30s
   • Recording: Fixing the target for 30s
2. Mode b
   For every target
   • Kinect warm-up: 30s
   • Recording: Fixing the target for 30s
3. Mode c
   For every target
   • Kinect warm-up: 30s
   • Recording: Fixing the target for 30s

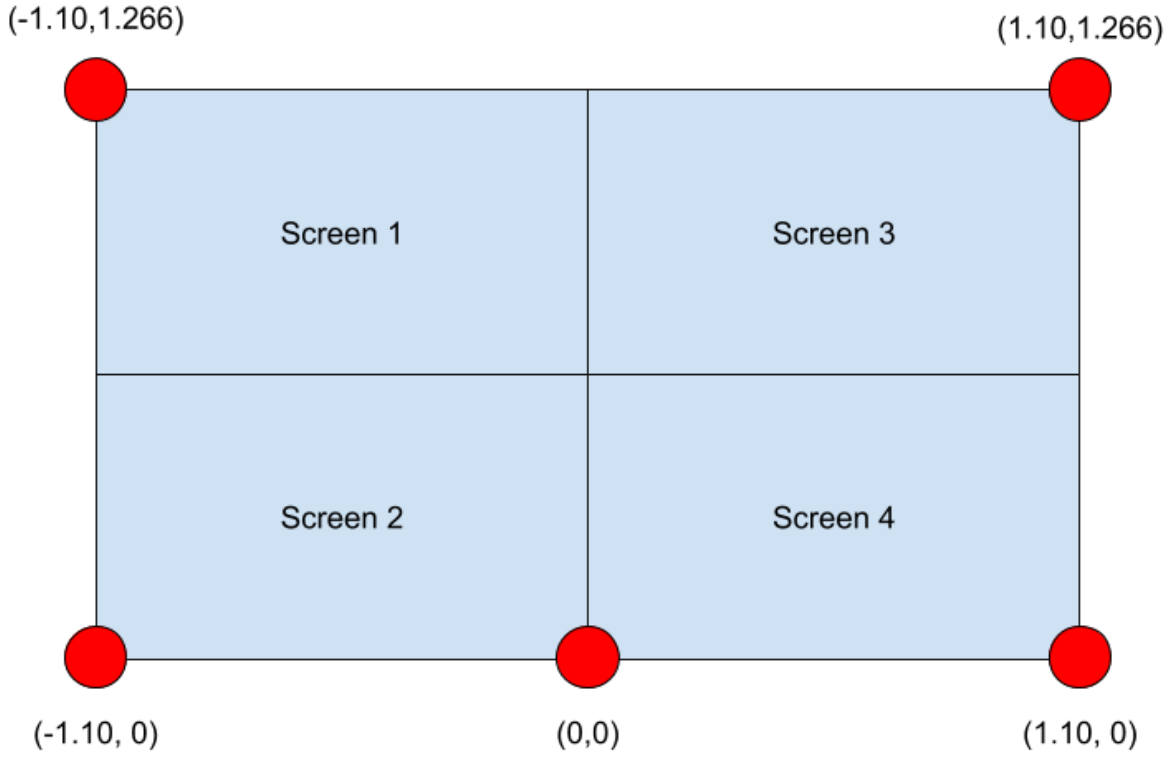

Figure 37: Coordinates of the five targets used in the experiment

The modes and targets are ordered after a draw to minimize bias coming from the kinect or the tiredness of the subject.

The subject is standing two meters away from the kinect, set slightly aside from the center so that the head is not totally frontal during the experiment (see Figure 38).

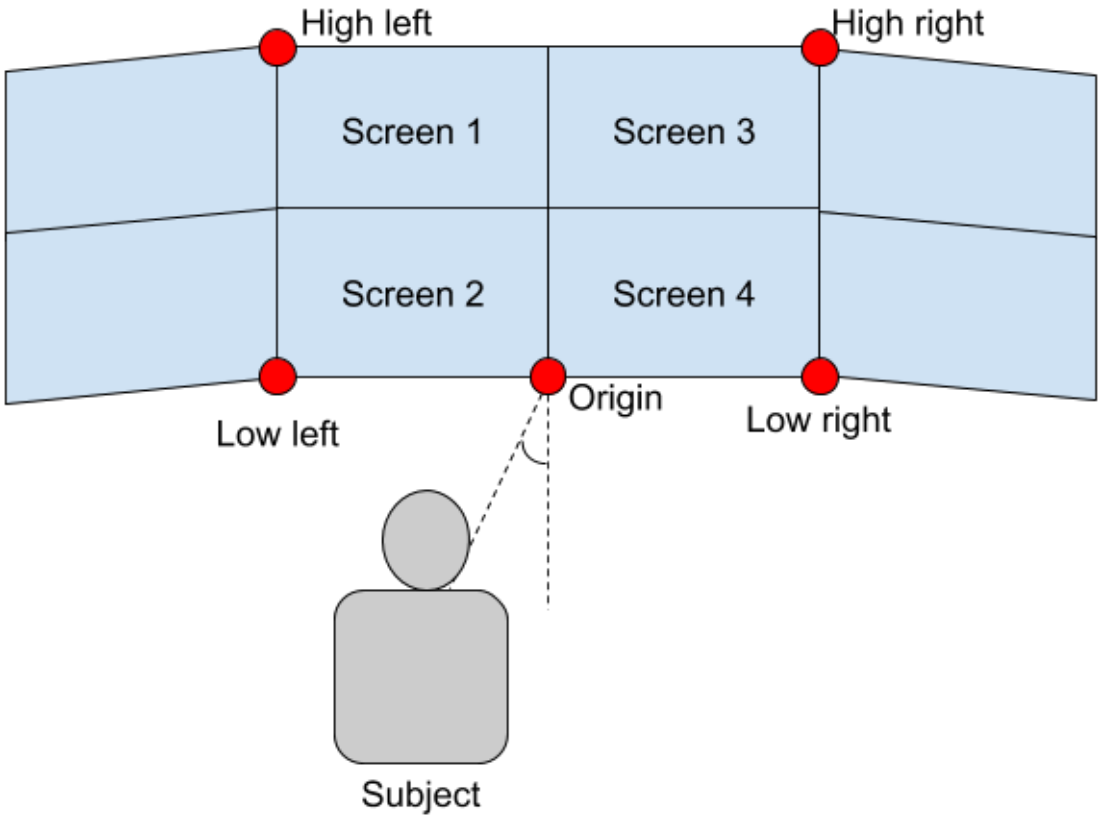

Figure 38: Experiment disposition

7.1.3 Analysis

Clean the data

We had to clean the data coming from the experiment before analysing them. The warm-up part at the beginning of every recording was set to reduce the odds of huge outliers to happen.

We've decided to put a threshold of 1.5m*1.5m and remove all data outside of this square (see Figure 39). We notice that almost all the outliers are captured during the Mode 1. It is logical, according to the fact that all values are registered without any process to smoothen the errors.

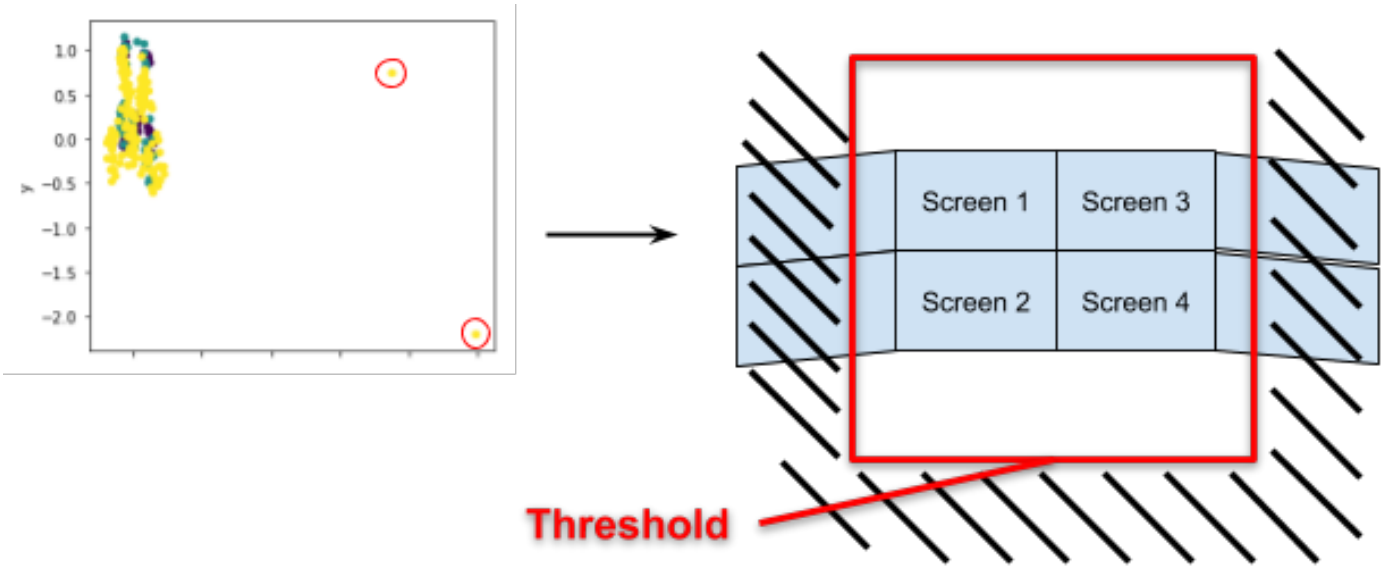

Figure 39: Removing outliers

Once the cleaning process is done, we get the following plot:

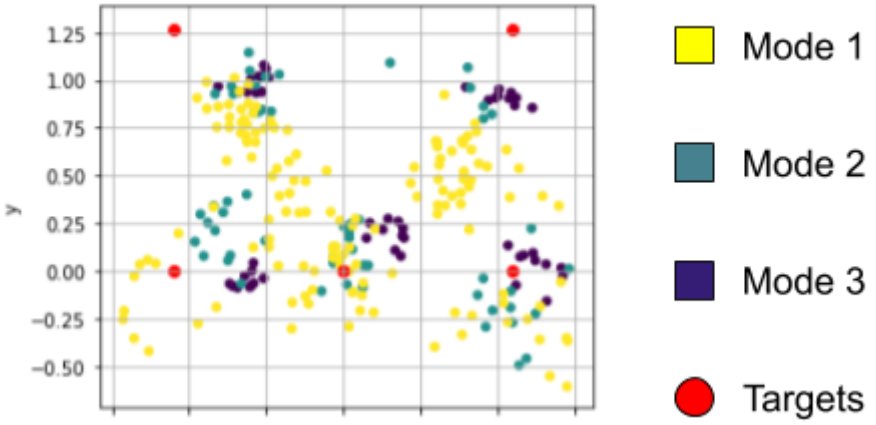

Figure 40: Results

General tendencies

We can observe that the data coming from Mode 1 have a much higher variance than the data coming from Modes 2 and 3. The gap between Mode 2 and 3 is less important (see Figure 41).

In Mode 1, the mean variance along the x axis equals 0.12 and 0.03 along the y axis. In Mode 2, it is divided by more than 5 along the x axis and slightly smaller along the y axis. On the other hand, in Mode 3, the variances alongside the x and y axis are in average 10 times lower than in Mode 1 (respectively 0.010 against 0.12 and 0.003 against 0.03).

	Mode 3		Mode 2		Mode 1	
	x	y	x	y	x	y
location						
high_left	0.009078	0.002661	0.013785	0.010247	0.023888	0.022463
high_right	0.013843	0.001158	0.059386	0.015823	0.034012	0.022978
low_left	0.004703	0.001821	0.016983	0.019639	0.222913	0.053133
low_right	0.014821	0.007737	0.028836	0.038826	0.321398	0.047113
origin	0.008121	0.004106	0.007636	0.016882	0.038148	0.043109

Figure 41: Variance depending on the mode and the target location

Impact of target position

If we compare the results on the target location and not only on the modes this time, we can make these assertions:

- **The variance is bigger on the right part of the DO-Dev**. At the same level (high or low) and for the same mode, the variance is at least 1.5 times higher on the right targets. This effect is more present for the low targets. For example, in Mode 3, the variance is 3 to 4 times higher for the low right target than the low left target.
- **The variance is bigger in the low part of the DO-Dev**. Except when the variance is already really low (for example in Mode 3, in the left part of the DO-Dev), the variance can be up to 10 times lower on the high part of the DO-Dev: in Mode 1, regardless of the side (left or right), the variance is 10 times bigger alongside the x axis and 2 times bigger alongside the y axis.
- **Using average and moving average doesn't delete residual errors**. It was predicted but the Figure 42 proves it. The residual error is reduced in the high part of the DO-Dev but doesn't disappear. Modes 2 and 3 tend to obtain the same results in terms of precision.

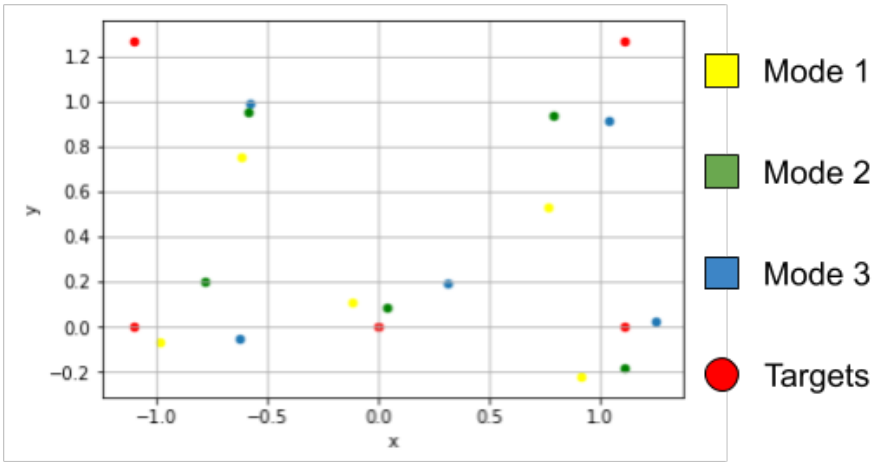

Figure 42: Mean depending on the mode and the target location

Differences between x and y axis

- **The variance is bigger alongside the x axis than the y axis**. Especially in the high right part of the DO-Dev, where in Mode 3, the variance is more than 10 times bigger alongside the x axis than the y axis. In Mode 1, it is in the low part of the DO-Dev that the variance is much bigger along the x axis: 4 time bigger on the left and more than 6 times bigger on the right.
- **The interpolation is better alongside the y axis for the low targets**. For the Mode 3, except for the high right target, the interpolation is in average 3 times more accurate along the y axis than along the x axis.
- **For the high targets(*), the interpolation is more precise alongside the x axis but overall not really accurate**. It could be interesting to add a calibration step right after the warm-up step to increase the interpolation accuracy.

(*): The high targets are the high_left and high_right targets on Figure 37.

Conclusion

This experiment enabled us to demonstrate that with normal average and moving average, the gaze interpolation dispersion was significantly reduced. Moreover, except for the left low target, the estimations were in average more accurate.

By adding a threshold, we removed the most obvious outliers. The lower variance for the interpolations with average and moving average underlined that there were less outliers during these mode.

However, to demonstrate the utility of using moving averages on top of regular averages, it could be appropriate to conduct another experiment based on gaze interpolation with moving targets. The moving average process would be more legitimate as there would be head movement and a trend to detect. In our experiment, the advantages are not sufficient to justify the double layer process.

To remove the residual errors, inherent to the kinect, we could add a calibration phase at the beginning of the use of the software. This way the offsets we see, particularly for the high targets could be artificially corrected.

7.2 Increasing the software performance

One of the biggest issue of our software was and still is its speed. We noticed that a whole loop (a landmarks detection followed by a gaze interpolation, then the sending of the coordinates to the server), takes often more than one second to complete.

7.2.1 Profiling

We did a profiling of our code to see which part was kinetically significant. By using Snakeviz, a graphical viewer of Python cProfile's module, we obtained the following results:

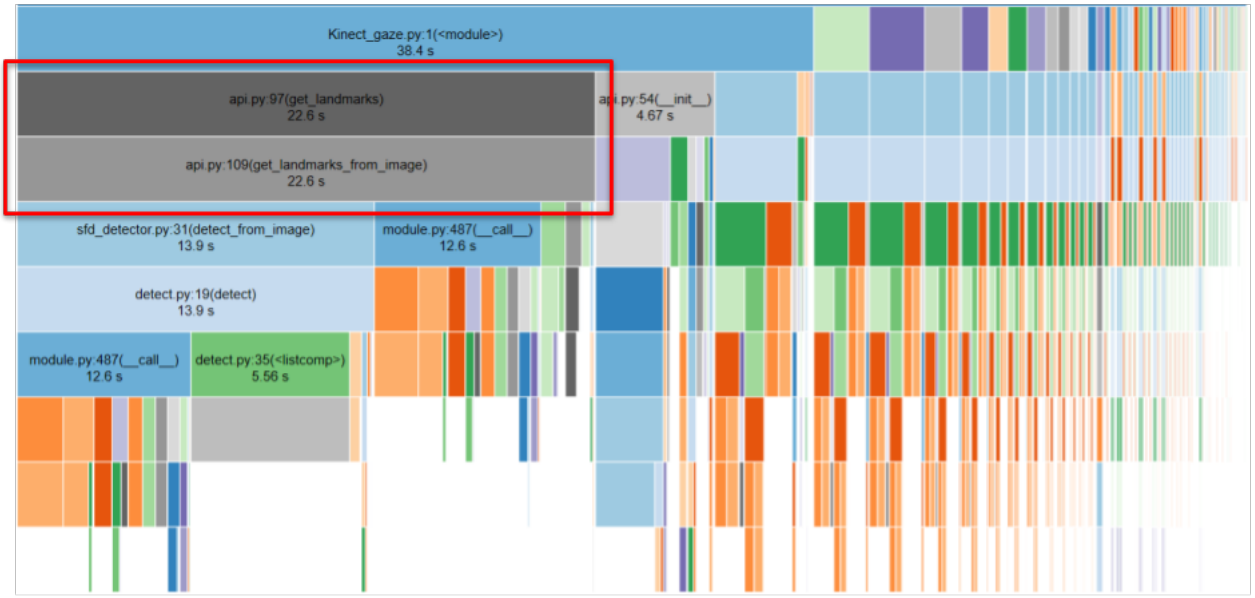

Figure 43: Profiling visualisation with Snakeviz

We let the software run for nearly forty seconds and more than half of the time it was the face recognition algorithm that was running. To check this information, we looked the per call duration of each submodules of the landmarks detector(see Figure 44).

percall	filename:lineno(function)
0.6862	api.py:97(get_landmarks)
0.6861	api.py:109(get_landmarks_from_image)
0.4213	sfd_detector.py:31(detect_from_image)
0.4206	detect.py:19(detect)
0.1848	net_s3fd.py:70(forward)

Figure 44: Per call duration of every submodule of the face recognition algorithm

Note: The results shown in Figure 44 take into account the first loop of the software which takes around 10s and therefore skews the results. It gives information however on which submodules of the face recognition library are significantly long.

On another hand, all our results and analysis until now were conducted with the hypothesis that only one person used the software. The problem is that the more people use the software, the slower the face landmarks algorithm gets (see Figure 45).

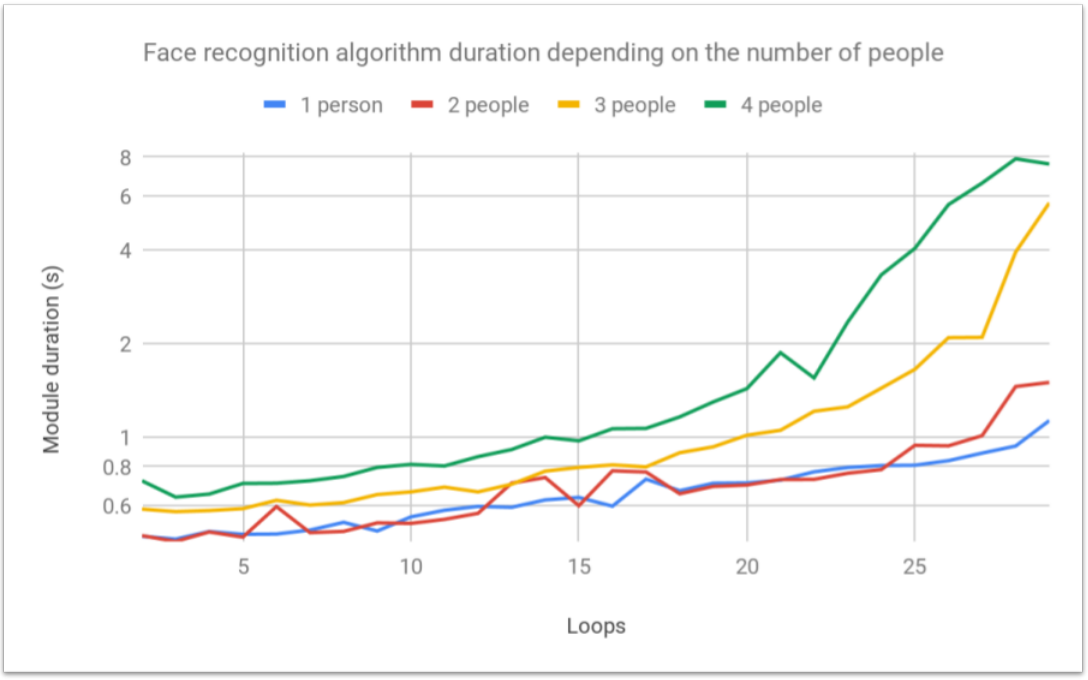

Figure 45: Table of face recognition algorithm duration per loop

We see that above two people, the duration of face recognition makes the software unusable. It never takes less than 0.6 seconds. It is problematic when the objective of the software was to be used by multiple people at the same time.

This chart also underlines the fact that the duration of the face recognition module increases over time. The cause of this problem is that every calculation is done on the CPU, on top of the kinect which is already using 70% of the CPU just for frame processing (see Figure 46).

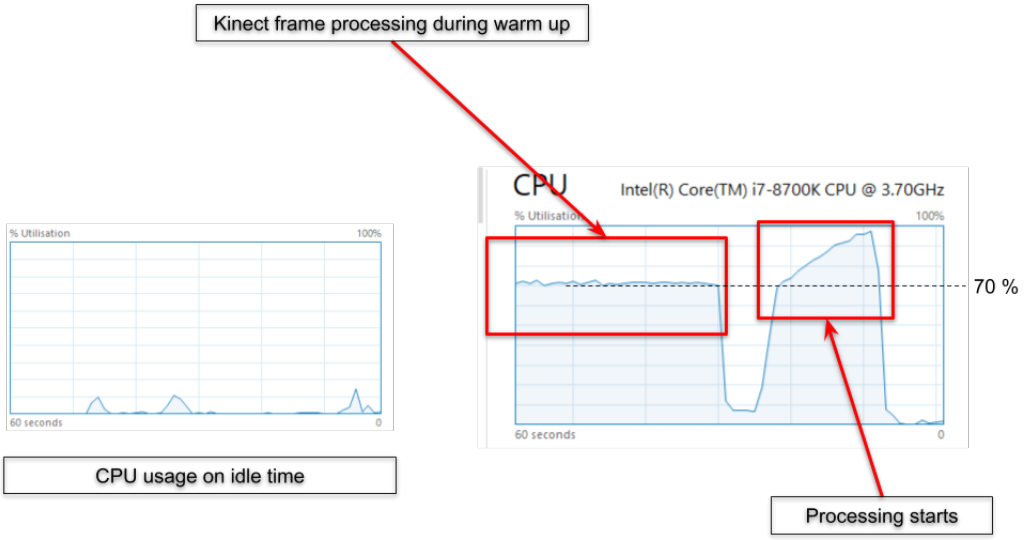

Figure 46: CPU usage while the software is running

7.2.2 Transfer heavy computation from the CPU to the GPU

To compensate for the increasing duration of the algorithm and to relieve the CPU of our computers, we tried to focus on the submodules detected in Figure 44. We modified these submodules by replacing every **numpy** element with its **Pytorch** equivalent when possible. The goal was to use more our GPUs instead of our CPUs.

These were the two main reasons:

- The GPUs were almost unused up to this point.
- For parallel calculations regarding floating numbers and vectors, GPUs architecture are more effective.

We used CUDA with Pytorch to enable the computations to take place on the GPU (see Appendice C for more details).

We compared the results acquired on two factors:

- The speed of the face recognition algorithm in the software.
- The percentage of CPU used for calculations that are not linked to the kinect.

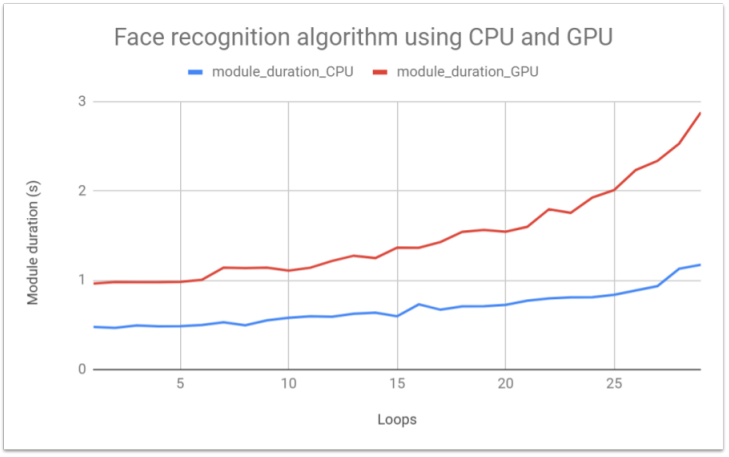

Figure 47: Algorithm duration using GPU versus using CPU

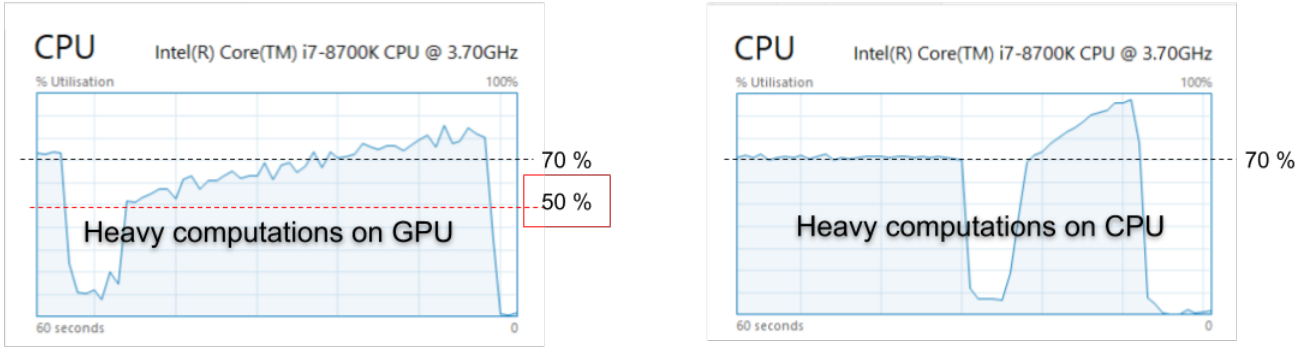

Figure 48: CPU use depending on heavy computations made on GPU vs CPU

Figures 47 and 48 show mixed results as detailed below:

On one hand we have a better management of CPU capacity (Figure 48): moving all the heavy computations from the CPU to the GPU allowed us to have some margin. We see that at the beginning of the software, only 50% of the CPU is used and it increases only to 80%. Moreover the raise is lessened in terms of CPU capacity but also in terms of speed. The software has run a lot longer with the GPU (almost one minute) and the raise of capacity used was less pronounced that with the CPU only.

On the other hand, the software is still faster with the CPU only (Figure 47). Possible causes for these results are the numerous transfers of data from numpy to Pytorch or the fact that the GPU is slower than the CPU for the computations that are required by the face recognition library.

Conclusion

It would require more time to understand the way Pytorch works with the GPU and which calculations would be interesting to pass on the GPU and which to keep on the CPU. On top of improving our work on the face recognition algorithm, it could also be interesting to see if it is possible to reduce the CPU capacity used by the kinect. We see on Figure 46 that the frames processing itself already request up to 70% of the CPU capacity.

8 Difficulties

8.1 Reliability

Depth sensors of the kinect

We noticed that by adding a warm up duration at the launching of our software, we reduced the chances of getting completely false face landmarks coordinates. Once the infrared LEDs warm up, the depth frame coordinates obtained in the Depth frame are often accurate. Therefore the step consisting of mapping color frame coordinates with depth information to collect space coordinates poses no problem. However, it doesn't completely solve the problem as errors can still punctually occur.

Addind a threshold in our experiment was a solution only adapted to the DO-Dev. The GDO has a cylindrical geometry, thus we can't consider the gazes to be constrained horizontally: the user will almost have a 360 degrees vision (see Figure 49). However, it will be possible to put a vertical threshold: we can just suppose that above a particular height, the gaze estimation will be considered as an outlier.

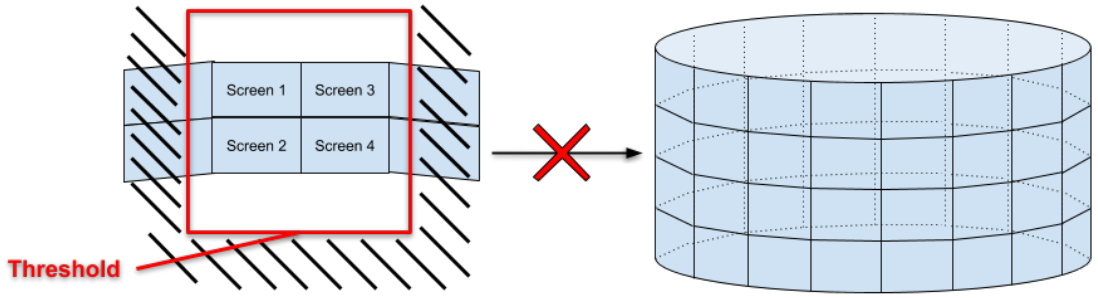

Figure 49: Problem with square-based threshold

We also have to take into account the fact that a future user of the GDO will most certainly move and not stay in a restricted area: the user's eyes won't stay in a predetermined region (unless stated so). Therefore we will have to use dynamic thresholds.

Detecting software errors and separating them from real eye movements will require further studies. We haven't been able yet to find a way to rule out automatically these errors.

Face landmarks detection

It sometimes happen that the face recognition library detects the same face twice (see Figure 50).

The red circles are the landmarks where you can see the superposition appearing clearly. These detected faces are always matched with the same skeleton. As a result, we manually delete the set of face landmarks that is the farthest from the skeleton associated.

It is a problem that could potentially be penalizing in terms of computation capacity or execution duration.

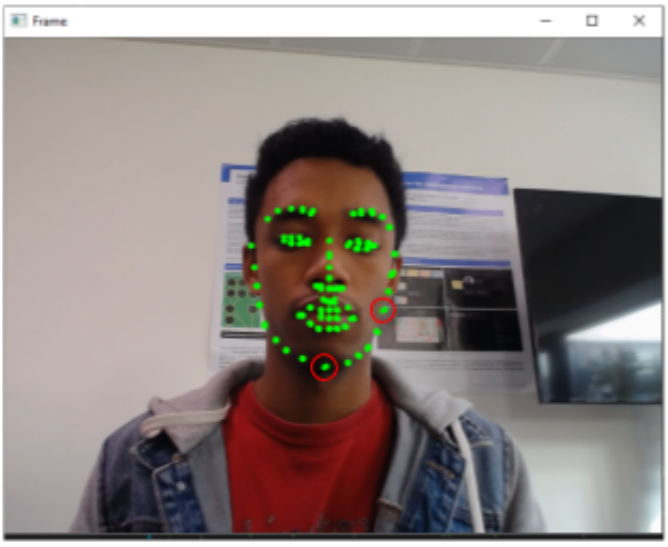

Figure 50: Face detected multiple times

8.2 Accuracy

To improve the accuracy of our software, we tried to detect the eyes direction and add this information in the gaze estimation process. It is interesting as we currently assume that a person is always looking straight when using the GDO.

This paper (6) shows the horizontal and vertical visual fields of a human:

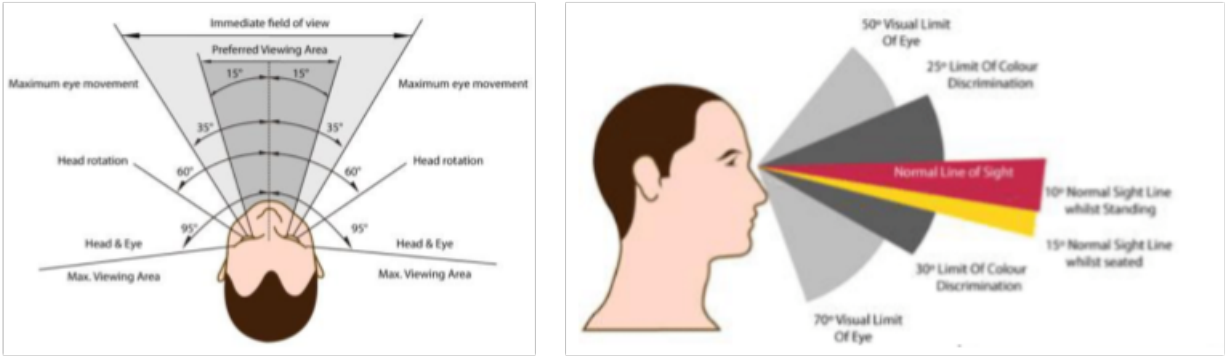

Figure 51: Average visual fields of a human

In the Figure 51 we can see that horizontally, the preferred viewing area is 15 laterally on the left and on the right and 10 in elevation and depression.

If the GDO user stands 1m from the GDO when processing information, that means that his gaze lies inside a 53cm wide and 35cm high rectangle. With an eye tracking step to estimate the gaze, we could reduce the uncertainty aof the estimation.

Protocol

We'll describe the protocol we followed when we experimented this method on images.

The steps are the following:

- Crop the eye region
- Get a grayscale version of the cropped region
- Divide the grayscale image in tiles and get the light intensity of each tile
- Classify the image based on our dataset

We have to make the hypothesis that the face is in a frontal disposition here. It is only the eyes direction that we are estimating.
To crop the eye region, we used our face recognition library (see Figure 52). The landmarks in red form a rectangle that we extract from the original picture.

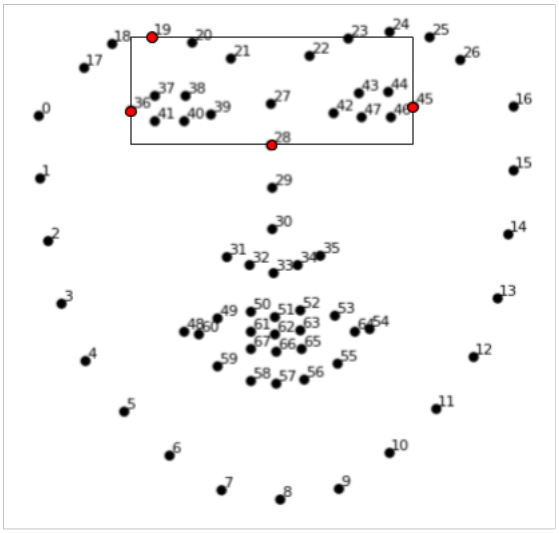

Figure 52: Cropping of the eye region

Then with the cv2 library we get a grayscale version of this image. We divide this grayscale image in fifteen tiles (see Figure 53).

Figure 53: Tiling of the grayscale image

For each tile, we calculate the **intensity level**. The intensity level is the mean intensity of all the pixels inside the tile. The closest from 0 it gets the darker it is. If it is near 255, then the tile is almost white.

If each tile k has an intensity i^k, we define the normalized intensity i_n^k as:

$$i_n^k = \frac{i^k}{I} \tag{9}$$

where $I = \sum_{k=0}^{14} i^k$

We normalize every intensity to cope with the fact that every image doesn't have the same luminosity. By doing so we keep the relative difference of intensity between each tiles but we can also compare images between themselves.

At the end of this process we've got the following normalized intensity vector $i = \begin{bmatrix} i_n^0 \\ i_n^1 \\ \vdots \\ i_n^{14} \end{bmatrix}$

We finally assign a direction to the eyes by classifying the vector i with the help of a set of vectors: these vectors are already linked to a certain direction and we use a Nearest Neighbours technique to estimate the direction associated to the vector i.

To collect the gaze estimation, we combine the eyes direction calculated by this method and the face orientation we described page 23 (see Figure 54).

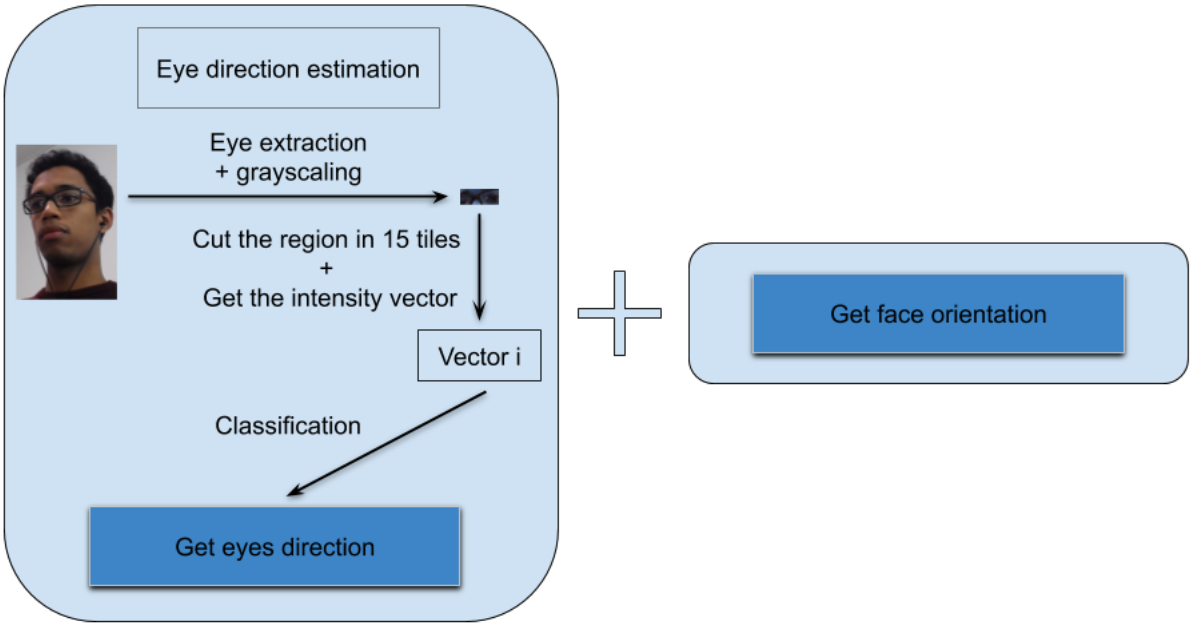

Figure 54: Combination of two processes to collect gaze estimation

We haven't been able to execute this process efficiently in our research. The results we obtained from the classification were not consistent enough.

The most probable causes of this failure are:

- The resolution of the kinect frames has been too low
- If the face is not frontal enough, the classification part doesn't work
- Our set of vectors is not big enough so we encounter a problem of overfitting

9 Conclusion & future work

During this internship, I have been able to develop the gdo-gaze project significantly: it is composed of a software able to predict the gaze of users in the Do-Dev and a server. The server shares the estimations coordinates with the Data Observatory's machines. These gaze estimations are calculated from the face orientation of the users and we detect the face orientation by using a face alignment library. To develop the software, I have used existing tools such as the face alignment library, but also kinects because of the many advantages they brought.
Although the software works, I encountered some obstacles that prevented a good user experience. Therefore I made experiments to try to solve the most obvious and inconvenient problems. And though these experiments enabled some improvements, there are still many issues that must be dealt with before considering the software as usable. For example, the CPU percentage required to acquire the face landmarks keeps increasing during the execution of the software. This problem prevents the GDO users to estimate their gaze during a long amount of time, even more so if there are multiple users. Another example is the detection of errors that would occur during the face recognition process. These errors would be amplified by the gaze estimation afterwards.
On another hand, one important step would be to adapt the software from the DO-Dev to the GDO. For this reason, I will try to modify the software to work simultaneously with multiple kinects. Doing this and solving the problems mentioned before are the main tasks I will try to execute for the rest of my stay at the DSI.

Overall, this internship at the DSI was a very good experience. I had to manage the whole project and I really appreciated the autonomy that I had. Moreover, it was my first encounter with a research environment. Therefore having to choose how to build the software and see my plans evolve as I developed it was stimulating. I had to be completely focus on the same project for the whole 6 months of my stay and to see the result at the end of this period was really satisfying. Although everything didn't go exactly as planned, it was truly interesting to develop this software and try to solve the many problems appearing almost simultaneously.
The work environment really helped me: the GDO team and Florian Guitton were and still are precious help I can still rely on whenever I feel stuck or not sure about the step to follow.

I hope to stay in the research environment after my graduation for a while to continue to work on intellectually stimulating projects such as this one. Thus, I could explore other fields and keep improving my knowledge in computer science while gaining some experience.

Appendices

A PyKinect2 mapping

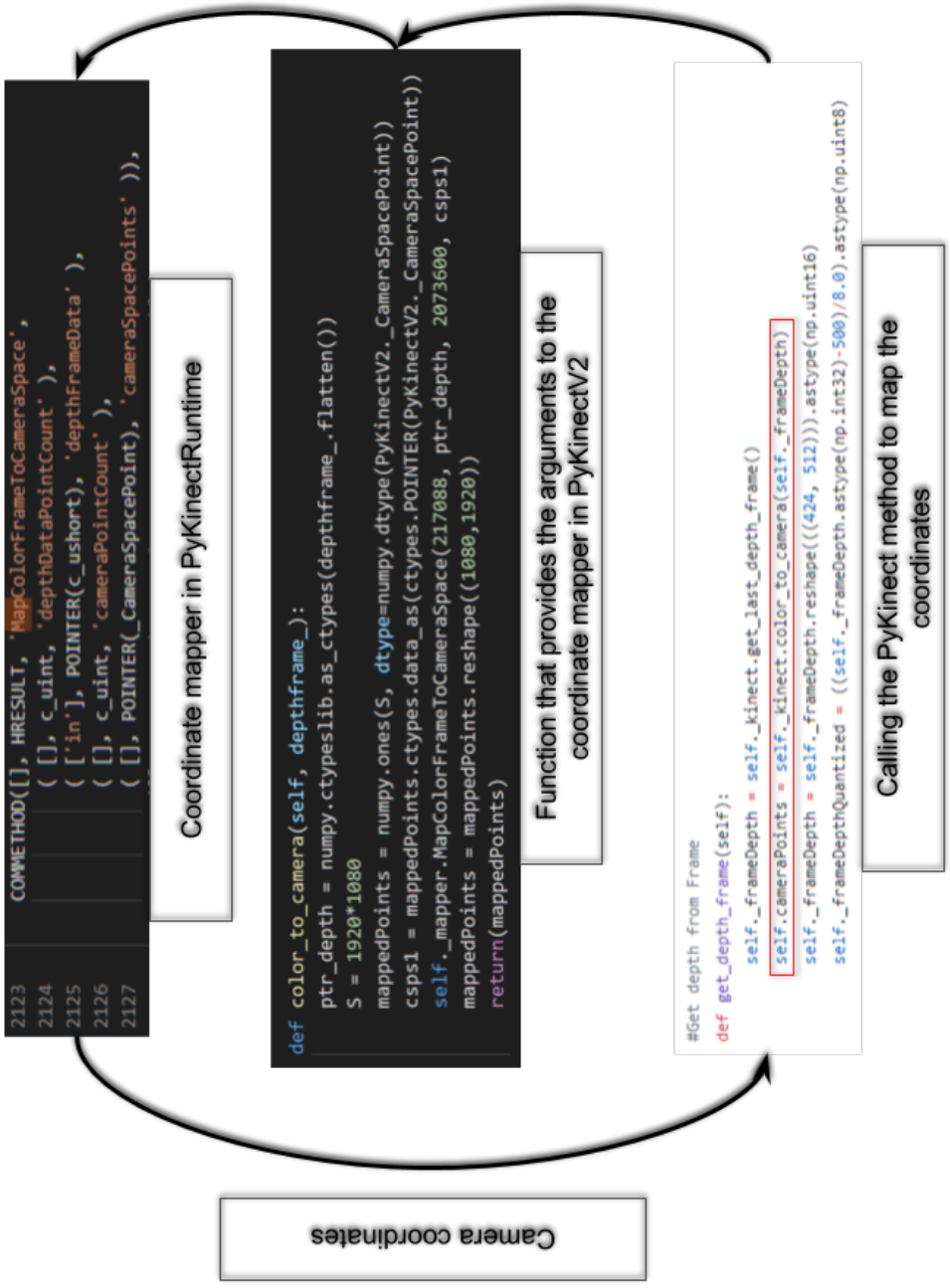

Figure 55: Coordinate mapper as defined in PyKinect2

B gdo-gaze server

Figure 56: HTML file loaded by the Data Observatory machines

C Pytorch and GPU

Enabling CUDA on Pytorch enables the heavy computations to take place in the GPU.

Numpy

```
bboxlist = []
for i in range(len(olist) // 2):
    olist[i * 2] = F.softmax(olist[i * 2], dim=1)
olist = [oelem.data.cpu() for oelem in olist]
for i in range(len(olist) // 2):
    ocls, oreg = olist[i * 2], olist[i * 2 + 1]
    FB, FC, FH, FW = ocls.size()  # feature map size
    stride = 2**(i + 2)    # 4,8,16,32,64,128
    anchor = stride * 4
    poss = zip(*np.where(ocls[:, 1, :, :] > 0.05))
    for Iindex, hindex, windex in poss:
        axc, ayc = stride / 2 + windex * stride, stride / 2 + hindex * stride
        score = ocls[0, 1, hindex, windex]
        loc = oreg[0, :, hindex, windex].contiguous().view(1, 4)
        priors = torch.Tensor([[axc / 1.0, ayc / 1.0, stride * 4 / 1.0, stride * 4 / 1.0]])
        variances = [0.1, 0.2]
        box = decode(loc, priors, variances)
        x1, y1, x2, y2 = box[0] * 1.0
        # cv2.rectangle(imgshow,(int(x1),int(y1)),(int(x2),int(y2)),(0,0,255),1)
        bboxlist.append([x1, y1, x2, y2, score])
bboxlist = np.array(bboxlist)
if 0 == len(bboxlist):
    bboxlist = np.zeros((1, 5))

return bboxlist
```

Pytorch

```
bboxlist = torch.cuda.FloatTensor([])
for i in range(len(olist) // 2):
    olist[i * 2] = F.softmax(olist[i * 2], dim=1)
olist = [oelem.data.cpu() for oelem in olist]
compteur = 0
for i in range(len(olist) // 2):
    ocls, oreg = olist[i * 2], olist[i * 2 + 1]
    FB, FC, FH, FW = ocls.size()  # feature map size
    stride = 2**(i + 2)    # 4,8,16,32,64,128
    anchor = stride * 4
    poss = zip(*np.where(ocls[:, 1, :, :] > 0.05))
    for Iindex, hindex, windex in poss:
        compteur += 1
        axc, ayc = stride / 2 + windex * stride, stride / 2 + hindex * stride
        score = ocls[0, 1, hindex, windex].to(device)
        loc = oreg[0, :, hindex, windex].contiguous().view(1, 4).to(device)
        priors = torch.cuda.FloatTensor([[axc / 1.0, ayc / 1.0, stride * 4 / 1.0, stride * 4 /
        variances = [0.1, 0.2]
        box = decode(loc, priors, variances)
        x1, y1, x2, y2 = box[0] * 1.0
        step = torch.stack((x1, y1, x2, y2, score))
        # cv2.rectangle(imgshow,(int(x1),int(y1)),(int(x2),int(y2)),(0,0,255),1)
        bboxlist = torch.cat((bboxlist, step))
bboxlist = torch.reshape(bboxlist, (compteur,5))
#bboxlist = np.array(bboxlist)
if 0 == len(bboxlist):
    bboxlist = np.zeros((1, 5))
```

Numpy

```
def nms(dets, thresh):
    if 0 == len(dets):
        return []
    x1, y1, x2, y2, scores = dets[:, 0], dets[:, 1], dets[:, 2], dets[:, 3], dets[:, 4]
    areas = (x2 - x1 + 1) * (y2 - y1 + 1)
    order = scores.argsort()[::-1]

    keep = []
    while order.size > 0:
        i = order[0]
        keep.append(i)
        xx1, yy1 = np.maximum(x1[i], x1[order[1:]]), np.maximum(y1[i], y1[order[1:]])
        xx2, yy2 = np.minimum(x2[i], x2[order[1:]]), np.minimum(y2[i], y2[order[1:]])

        w, h = np.maximum(0.0, xx2 - xx1 + 1), np.maximum(0.0, yy2 - yy1 + 1)
        ovr = w * h / (areas[i] + areas[order[1:]] - w * h)

        inds = np.where(ovr <= thresh)[0]
        order = order[inds + 1]

    return keep
```

Pytorch

```
def nms(dets, thresh):
    if 0 == len(dets):
        return []
    x1, y1, x2, y2, scores = dets[:, 0], dets[:, 1], dets[:, 2], dets[:, 3], dets[:, 4]
    areas = (x2 - x1 + 1) * (y2 - y1 + 1)
    order = torch.argsort(scores, descending = True)

    keep = []
    while list(order.size())[0] > 0:
        i = order[0]
        keep.append(i)
        xx1, yy1 = torch.max(x1[i], x1[order[1:]]), torch.max(y1[i], y1[order[1:]])
        xx2, yy2 = torch.max(x2[i], x2[order[1:]]), torch.max(y2[i], y2[order[1:]])

        w, h = torch.max(torch.cuda.FloatTensor([0.0]), xx2 - xx1 + 1), torch.max(torch.cuda.Float
        ovr = w * h / (areas[i] + areas[order[1:]] - w * h)

        inds = torch.cuda.LongTensor([])
        j = 0
        while list(ovr.size())[0] > 0:
            if (ovr[0] <= thresh):
                inds = torch.cat((inds, torch.cuda.LongTensor([j])))
            ovr = ovr[1:]
            j += 1
        order = order[inds + 1]

    return keep
```

Numpy

```
    """ Crops the image around the center. Input is expected to be an np.ndarray """
    ul = transform([1, 1], center, scale, resolution, True)
    br = transform([resolution, resolution], center, scale, resolution, True)
    # pad = math.ceil(torch.norm((ul - br).float()) / 2.0 - (br[0] - ul[0]) / 2.0)
    if image.ndim > 2:
        newDim = np.array([br[1] - ul[1], br[0] - ul[0],
                           image.shape[2]], dtype=np.int32)
        newImg = np.zeros(newDim, dtype=np.uint8)
    else:
        newDim = np.array([br[1] - ul[1], br[0] - ul[0]], dtype=np.int)
        newImg = np.zeros(newDim, dtype=np.uint8)
    ht = image.shape[0]
    wd = image.shape[1]
    newX = np.array(
        [max(1, -ul[0] + 1), min(br[0], wd) - ul[0]], dtype=np.int32)
    newY = np.array(
        [max(1, -ul[1] + 1), min(br[1], ht) - ul[1]], dtype=np.int32)
    oldX = np.array([max(1, ul[0] + 1), min(br[0], wd)], dtype=np.int32)
    oldY = np.array([max(1, ul[1] + 1), min(br[1], ht)], dtype=np.int32)
    newImg[newY[0] - 1:newY[1], newX[0] - 1:newX[1]
           ] = image[oldY[0] - 1:oldY[1], oldX[0] - 1:oldX[1], :]
    newImg = cv2.resize(newImg, dsize=(int(resolution), int(resolution)),
                        interpolation=cv2.INTER_LINEAR)
    return newImg
```

Pytorch

```
    """ Crops the image around the center. Input is expected to be an np.ndarray """
    ul = transform([1, 1], center, scale, resolution, True)
    br = transform([resolution, resolution], center, scale, resolution, True)
    # pad = math.ceil(torch.norm((ul - br).float()) / 2.0 - (br[0] - ul[0]) / 2.0)
    if len(image.size()) > 2:
        newDim = torch.cuda.IntTensor([br[1] - ul[1], br[0] - ul[0],
                                       image.shape[2]])
        newImg = torch.zeros(newDim.tolist(), dtype=torch.uint8, device = torch.device("cuda:0"))
    else:
        newDim = torch.cuda.IntTensor([br[1] - ul[1], br[0] - ul[0]])
        newImg = torch.zeros(newDim.tolist(), dtype=torch.uint8, device = device)
    ht = image.shape[0]
    wd = image.shape[1]
    newX = torch.cuda.IntTensor([max(1, -ul[0] + 1), min(br[0], wd) - ul[0]])
    newY = torch.cuda.IntTensor([max(1, -ul[1] + 1), min(br[1], ht) - ul[1]])
    oldX = torch.cuda.IntTensor([max(1, ul[0] + 1), min(br[0], wd)])
    oldY = torch.cuda.IntTensor([max(1, ul[1] + 1), min(br[1], ht)])
    newImg[newY[0] - 1:newY[1], newX[0] - 1:newX[1]
           ] = image[oldY[0] - 1:oldY[1], oldX[0] - 1:oldX[1], :]
    newImg = cv2.resize(newImg.cpu().numpy(), dsize=(int(resolution), int(resolution)),
                        interpolation=cv2.INTER_LINEAR)
    return newImg
```

Numpy

```
        torch.set_grad_enabled(False)
        landmarks = []
        for i, d in enumerate(detected_faces):
            center = torch.FloatTensor(
                [d[2] - (d[2] - d[0]) / 2.0, d[3] - (d[3] - d[1]) / 2.0])
            center[1] = center[1] - (d[3] - d[1]) * 0.12
            scale = (d[2] - d[0] + d[3] - d[1]) / self.face_detector.reference_scale

            inp = crop(image, center, scale)
            inp = torch.from_numpy(inp.transpose(
                (2, 0, 1))).float()

            inp = inp.to(self.device)
            inp.div_(255.0).unsqueeze_(0)

            out = self.face_alignment_net(inp)[-1].detach()
            if self.flip_input:
                out += flip(self.face_alignment_net(flip(inp))
                            [-1].detach(), is_label=True)
            out = out.cpu()

            pts, pts_img = get_preds_fromhm(out, center, scale)
            pts, pts_img = pts.view(68, 2) * 4, pts_img.view(68, 2)
```

Pytorch

```
torch.set_grad_enabled(False)
landmarks = []
image = torch.from_numpy(image).to(self.device)
for i, d in enumerate(detected_faces):
    center = torch.cuda.FloatTensor(
        [d[2] - (d[2] - d[0]) / 2.0, d[3] -
         (d[3] - d[1]) / 2.0])
    center[1] = center[1] - (d[3] - d[1]) * 0.12
    scale = (d[2] - d[0] +
             d[3] - d[1]) / self.face_detector.reference_scale

    inp = crop(image, center, scale)
    inp = torch.from_numpy(inp.transpose(
        (2, 0, 1))).float()

    inp = inp.to(self.device)
    inp.div_(255.0).unsqueeze_(0)
    time2 = time.time()

    out = self.face_alignment_net(inp)[-1].detach()
    if self.flip_input:
        out += flip(self.face_alignment_net(flip(inp))
                    [-1].detach(), is_label=True)
    #out = out.cpu()

    # pts, pts_img = get_preds_fromhm(out, center, scale)
    pts_img = get_preds_fromhm(out, center, scale)
    #pts, pts_img = pts.view(68, 2) * 4, pts_img.view(68, 2)
    pts_img = pts_img.view(68, 2)
```

References

[1] L. Chao, D. Birch, Z. Xu and Y. Guo. *HoloKinect: A Real-time Scalable Fusion Framework for Multiple Target Tracking*, 2017

[2] K. Alberto, F. Mora, J. Ordobez. *Gaze Estimation from Multimodal Kinect Data*, 2012

[3] http://gravis.dmi.unibas.ch/PMM/

[4] A. Bulat, G. Tzimiropoulos. *How far are we from solving the 2D & 3D Face Alignment problem? (and a dataset of 230,000 3D facial landmarks)*, International Conference on Computer Vision, 2017

[5] A. Bernardino, C. Vismara, F. Baptista, F. Carnide, S. Oom, S. Bermudez i Badia,E. Gouveia and H. Gamboa. *A Dataset for the Automatic Assessment of Functional Senior Fitness Tests using Kinect and Physiological Sensors*,1st International Conference on Technology and Innovation in Sports, Health and Wellbeing, 2016

[6] A. Torrejon, V. Callaghan, H. Hagras, *Panoramic Audio and Video: towards an Immersive Learning experience*, 2013