Extrinsic Eye-Tracker Calibration

Ayush Chamoli[1] and Mahdi Chamsaddine[2]

[1] chamoli@rptu.del
[2] mahdi.chamseddine@dfki.de

Abstract. With the recent advancements in the augmented reality technology, eye-trackers are important now more than they have ever been. The gaze data is used to calculate where person is looking at in the space around them. This gaze data is not only relevant to the device which has the eye tracker, but it can also be used in another frame of reference. However, this functionality does not come with most of the eye-trackers. This project provides an end-to-end framework called PupilCalib which helps with the task. It provides a modular API that can be used with any eye tracker hardware to project the gaze information to another frame of reference.

Keywords: Eye Tracker, Camera Calibration, Pupil Detection

1 Introduction

Eye tracking has been a necessary aspect in order to study and understand human behaviour and their cognitive processes [7]. An eye tracker headset captures and interprets subtle movement of person's eyes. Typically, these systems involve combination of cameras, infrared sensors and image processing algorithms. With the use of this, these systems can track eye movement, distinguish between fixations (periods of subtle gaze) and saccades (rapid eye movements between fixations) and produce detailed gaze plots and heatmaps to visualize region of interest. Currently, it finds its use in the a variety of fields including psychology, market research, human computer interaction, medical diagnostics, virtual reality and gaming.

However, most of the eye tracking solutions out there aim to find out the position of the gaze of the user with respect to the eye tracking headset itself, which is done with the use of user's pupils and projecting the point on which the user's eye look onto the scene. This gaze information can also be useful and relevant outside the eye tracking headset itself. Even though most of the modern solutions provide API for accessing this gaze information, it is only relevant with respect to the camera module attached with the eye tracker headset. As it turns out, this gaze information can also be useful in another frame of reference. For example, in a vehicle with a camera mounted on it, it can be really useful to have information of a user's gaze with respect to the vehicle camera. It can also be helpful in various human-computer interaction and augmented reality applications.

This project aims to provide an end-to-end API that can be used to project gaze from eye tracker headset to another camera. In this report, the details about the eye tracker and other elements of the setup is discussed. It is then followed by the methodology used to implement the API solution, PupilCalib.

2 Related Work

In order to understand the working of PupilCalib, we will take a look at several hardware components and techniques used in computer vision.

2.1 Pupil Core

Pupil Core [4] is a headset which captures and records the gaze data of the person. It includes a scene camera and two infrared (IR) spectrum eye camera for pupil detection. The scene camera is mounted above the user's eye on the frame which aligns the scene camera optics with the user's eye. The scene camera captures a portion of the users field of view at 30Hz while the eye camera is mounted on the frame and it faces the user's pupil. It captures the user's pupil at 30Hz.

The Pupil API uses "dark pupil" detection method in order to detect pupil and project gaze on the scene camera output. Upon getting the region of interest via the strongest response for center-surround feature as proposed by Swirski [6], the edges are detected using Canny [1] in order to find contours in eye image. Contours are filtered and split into sub-contours based on criteria of curvature continuity. Pupil ellipse is then formed using ellipse fitting [3] on the subset of contours. The result is calculated as a ratio of supporting edge length and ellipse circumference. If this ratio is above a threshold, the pupil ellipse is reported. Otherwise the algorithm reports no pupil.

The algorithm is robust to reflections and can therefore work for user wearing contact lenses or eyeglasses. After calibration, detection rate is 80%.

2.2 Camera Calibration

Camera calibration is an important step in computer vision and it is used to determine the intrinsic and extrinsic parameters of a camera. Intrinsic camera parameters define how a 3D scene projects onto a 2D image. Intrinsic Calibration is necessary in order to remove radial distortion. Using a chessboard pattern is a common way to determine the intrinsic parameters of a camera. It is defined by:

$$camera\ matrix = \begin{bmatrix} \alpha_x & s & c_x \\ 0 & \alpha_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

where $\alpha_x = fk_x$ and $\alpha_y = fk_y$ represent the focal length in pixels along the x-axis and y-axis respectively, (c_x, c_y) represent the coordinates of the center of the image and s represents the skew. These parameters are crucial in order to convert the pixel coordinates of an image to camera coordinates and vice versa.

Extrinsic calibration is necessary to determine the camera's position and orientation in the real world with respect to the camera coordinate system. It defines the camera's position and orientation relative to the world coordinate system. It is defined by:

$$\begin{bmatrix} \mathbf{R}_{3\times3} & \mathbf{t}_{3\times1} \\ 0_{1\times3} & 1 \end{bmatrix} \tag{2}$$

where R is the rotation matrix which specifies the orientation of the camera with respect to the world coordinate system and T is the translation vector which describe the camera position with respect to the world coordinate system.

It is useful in converting a point from the camera coordinate system to the world coordinate system and vice versa.

2.3 AprilTags

AprilTags [5] are fiducial visual marker system which are commonly used in computer vision and robotics application. They are similar to QR codes, that is they have unique black-and-white square patterns which are encoded identifiers and it makes it easy for the camera to distinguish them. They are robust to varying lighting conditions and also partial occlusion to perspective. With a proper detection algorithm, they are ideal in the field of robotics and augmented reality. They help in tracking and localizing objects in 3D space.

3 Implementation details

This project uses Pupil Core as the eye tracker headset and IDS 3080CP camera. The camera on top of pupil headset is referred as the scene camera and the IDS camera is referred as the world camera.

The use of Pupil API involves the use of Pupil Core Software which does the heavy lifting part of detecting the pupils and providing the gaze information. It also provides with the camera feed for the scene camera and both the infra-red (IR) eye camera. Along with this, it constantly provides the confidence of the gaze data. The IDS camera uses it's own API in order to access the camera feed and is available to download and use on their website.

The software solution, PupilCalib, is made using PyQt. It includes classes for managing camera and is overloaded for each specific camera which makes the entire API modular. The main idea for the end-to-end solution is to make sure that the solution works with any eye-tracking headset and any camera with minimal effort.

3.1 Intrinsic camera calibration for world and scene camera

The initial step involves determining the intrinsic camera parameters for world camera and scene camera. In order to do this, a chessboard pattern is used. The chessboard is placed at different positions and the images are taken. These images are then used to calculate the intrinsic parameters of the camera. This is done for both, the world and the scene camera.

The API comes with the functionality of saving and loading these parameters in case of using the same cameras.

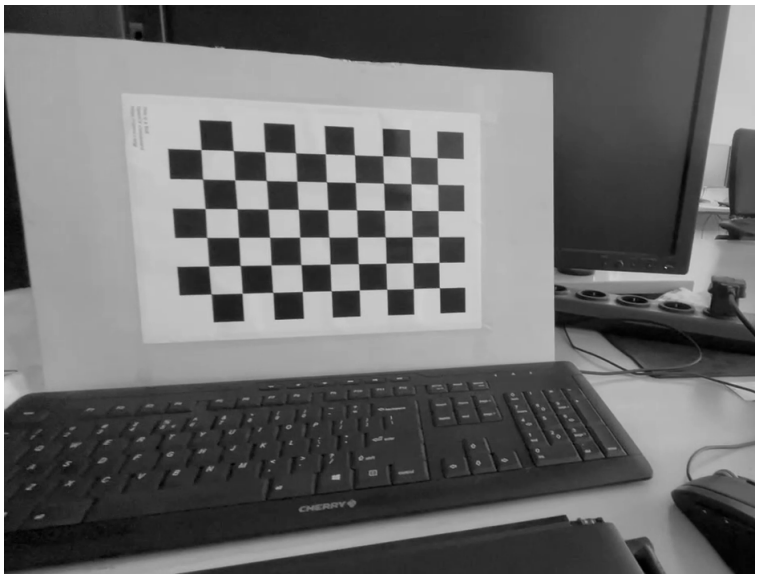

Fig. 1: Intrinsic Camera Calibration for world camera

3.2 Detecting Gaze and projecting it on the scene camera

The information of the Gaze can be accessed by the Pupil Core API. The gaze data is accurate upto a distance of 1 meter in front of the user and the accuracy of this data drops as the user starts

looking at object further away. This can be projected on the scene camera which indicates the point at which the user is looking. The gaze point is represented in the scene camera coordinate system as

$$\begin{bmatrix} x_{gaze} \\ y_{gaze} \\ z_{gaze} \\ 1 \end{bmatrix} \tag{3}$$

Now with the use of the intrinsic camera coordinates for the scene camera, the pixel coordinate of the gaze can be calculated as

$$\begin{bmatrix} u_{gaze} \\ v_{gaze} \\ w_{gaze} \end{bmatrix} = \begin{bmatrix} \alpha_x & s & c_x & 0 \\ 0 & \alpha_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}_{scene} \begin{bmatrix} x_{gaze} \\ y_{gaze} \\ z_{gaze} \\ 1 \end{bmatrix} \tag{4}$$

$$x_{gaze}^{pix} = \frac{u_{gaze}}{w_{gaze}} \tag{5}$$

$$y_{gaze}^{pix} = \frac{v_{gaze}}{w_{gaze}} \tag{6}$$

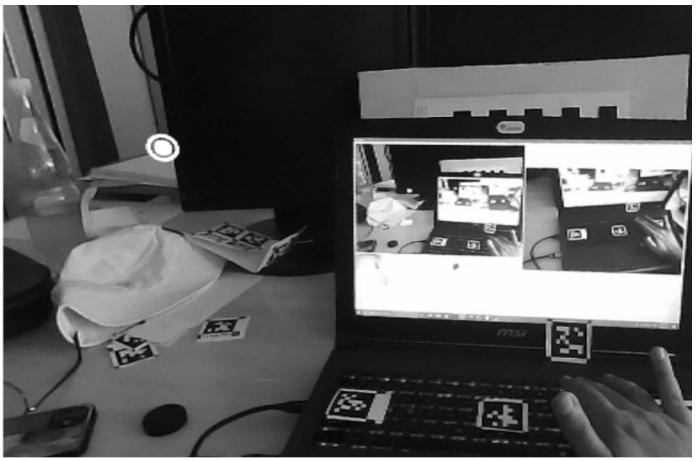

Fig. 2: Gaze point in the scene camera

3.3 Detecting AprilTags in the world and scene camera

In the setup, the AprilTags are placed at known locations with known world coordinates. The corresponding world coordinates are fed to the API before detecting the AprilTags. Once this is done, the AprilTags are located on each of the image using a detection algorithm and the pixel coordinates of the corners of each AprilTag is saved for further processing.

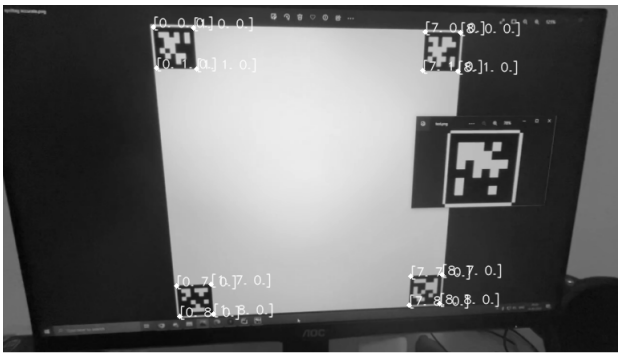

Fig. 3: AprilTag detection on the scene camera

3.4 Extrinsic calibration for world and scene camera with respect to the AprilTags

With the use of the pixel coordinates of the corners of the AprilTags, the extrinsic parameters of each of the camera is calculated. This is done for each frame with the use of Perspective-n-Point (PnP) pose computation [2]. It solves this problem by solving for rotation and translation which minimize the re-projection error from the 3D-2D point correspondences.

3.5 Projecting the gaze on the world camera

Upon obtaining the intrinsic and extrinsic parameters for both the cameras, the gaze point is projected in the world camera.

First the gaze point is obtained in the world coordinate system.

$$\begin{bmatrix} X_{gaze} \\ Y_{gaze} \\ Z_{gaze} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_{scene} & \mathbf{t}_{scene} \\ 0 & 1 \end{bmatrix}^{-1}_{4\times4} \begin{bmatrix} x_{gaze} \\ y_{gaze} \\ z_{gaze} \\ 1 \end{bmatrix} \tag{7}$$

Now, this gaze point is used to calculate the gaze point in the world camera coordinate system by using the extrinsic parameters of the world camera.

$$\begin{bmatrix} \bar{x}_{gaze} \\ \bar{y}_{gaze} \\ \bar{z}_{gaze} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_{world} & \mathbf{t}_{world} \\ 0 & 1 \end{bmatrix}_{4\times4} \begin{bmatrix} X_{gaze} \\ Y_{gaze} \\ Z_{gaze} \\ 1 \end{bmatrix} \tag{8}$$

Finally, in order to project this point on the image, we transform the world camera coordinate point to pixel coordinates using the intrinsic camera parameters of the world camera.

$$\begin{bmatrix} \bar{u}_{gaze} \\ \bar{v}_{gaze} \\ \bar{w}_{gaze} \end{bmatrix} = \begin{bmatrix} \alpha_x & s & c_x & 0 \\ 0 & \alpha_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}_{world} \begin{bmatrix} \bar{x}_{gaze} \\ \bar{y}_{gaze} \\ \bar{z}_{gaze} \\ 1 \end{bmatrix} \tag{9}$$

$$\bar{x}^{pix}_{gaze} = \frac{\bar{u}_{gaze}}{\bar{w}_{gaze}} \tag{10}$$

$$\bar{y}^{pix}_{gaze} = \frac{\bar{v}_{gaze}}{\bar{w}_{gaze}} \tag{11}$$

Here, $(\bar{x}^{pix}_{gaze}, \bar{y}^{pix}_{gaze})$ is the pixel coordinates of the gaze in world camera. This point is drawn on the image of the world camera and only appears if this point lies in the visible range of the image.

3.6 Exporting this information

The $(\bar{x}^{pix}_{gaze}, \bar{y}^{pix}_{gaze})$ and $(\bar{x}_{gaze}, \bar{y}_{gaze}, \bar{z}_{gaze})$ is accessible to the end user via the API which can then be used for further analysis.

4 Experiments

The PupilCalib API is tested with the help of the AprilTags. Upon setting up the devices and calibrating them, a new target AprilTag is placed in the views of both of the camera. Now, the user wearing the Pupil Core looks at the target and the information is used to project the 3D gaze point to the world camera. The error here is calculated as the pixel distance between the position of the new AprilTag in the world camera and the projection of the 3D gaze in the world camera.

Target Distance	Light Settings	Obervations	Mean	Standard Deviation
$<= 1m$	Well Lit Room	30	74.38	12.57
$> 1m$	Well Lit Room	20	102.82	15.90
$<= 1m$	Dim Lit Room	45	93.34	28.53
$> 1m$	Dim Lit Room	30	135.78	41.45

Table 1: PupilCalib performance with IDS 3080CP and Pupil Core in different settings

Table 1 shows that PupilCalib performs the best in a well lit condition when the 3D gaze point is less than 1m away from the user. However, a significant drop in performance is also observed in dim lit conditions as it generates a lot of noise in the world camera and removing the noise by blurring affects the extrinsic world camera calibration. A similar drop in performance is also observed as the target AprilTag is placed further away and Pupil Core API doesn't return 3D gaze point with a high confidence resulting in inaccurate calculation for the gaze point in world camera.

5 API Details

PupilCalib has a modular design and is therefore implemented with the idea of using different piece of hardware. It has a class based design which helps in achieving this task. The base class of the API performs all the tasks necessary and needs implementation of one to two functions only.

The world camera must inherit from the class `CameraManager`. This inherited class needs an updated `captureCurrentFrame()` function which basically stores the current buffer of the camera in it's `m_current_frame` variable. The image must also be converted to 8-bit black and white image.

The scene camera must inherit from class `CoreManager`. This inherited class also needs an update for the `captureCurrentFrame()` function. The current frame should be converted to 8-bit black and white image. In case the hardware also captures images of pupils, those can be stored in `m_current_left` and `m_current_right`. Also, the 3D gaze point must be stored in the class in the variable `interest_point`.

Once these new classes are implemented, they can be plugged in the PupilCalib API and can be used to project the 3D gaze point from the new scene camera to the new world camera.

6 Conclusion

In this report, we've described about our API, PupilCalib, which provides an easy-to-use method that provides a way to project 3D gaze point from the eye tracker hardware to another camera. The API is modular and can therefore be expanded to be used with any eye tracker hardware. The observations are taken in different testing conditions and the results are described.

References

1. John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698, 1986.
2. Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, jun 1981.
3. Andrew W Fitzgibbon, Robert B Fisher, et al. *A buyer's guide to conic fitting*. Citeseer, 1996.
4. Moritz Kassner, William Patera, and Andreas Bulling. Pupil: An open source platform for pervasive eye tracking and mobile gaze-based interaction, 2014.
5. Edwin Olson. AprilTag: A robust and flexible visual fiducial system. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3400–3407. IEEE, May 2011.
6. Lech Świrski, Andreas Bulling, and Neil Dodgson. Robust, real-time pupil tracking in highly off-axis images. In *Proc. ACM International Symposium on Eye Tracking Research and Applications (ETRA)*, pages 173–176, 2012.
7. Nicholas J. Wade and Benjamin W. Tatler. *The Moving Tablet of the Eye: the origins of modern eye movement research*. Oxford University Press, United Kingdom, 2005.