Project Report: Integrate Eye-tracking with BCI System for Assistive Robots

Jingxi Xu*
Department of Computer Science
Columbia University
jingxi.xu@columbia.edu

Abstract

In this project, eye-tracking is used in the Brain Computer Interface (BCI) system [Akinola et al.] for choosing rooms and tables. I use camera calibration to empirically find the intrinsic parameters of RViz camera which then allows me to develop a function to return the image plane pixel coordinate of any point on the map, given their world coordinate and the camera position. The only requirement for this method to work is that the image plane does not change position on the screen and its size is fixed during the experiment. This method will be very helpful for future eye-tracking experiments in that it allows developer to define an arbitrary number of cube markers in any orientation and position, and no matter how the camera position is changed during experiments, it can always return an accurate pixel coordinates of that marker. I also run experiments to find that the location inaccuracy of finding the pixel coordinate is at most 4 pixels for this system and the time to run eye-tracking can be theoretically reduced to 0.0115 second to correctly find the choice of user.

1 Introduction

As humanoid robots become more capable, the interaction between human and robots become more important. Shared autonomy is when a human and a robot agent collaborate to accomplish a task utilizing the strengths of both. This is not only important for developing assistive robots for disabled people with special needs, but also for developing more efficient active robots which know how to use the feedback from human to achieve better performance. Using BCI with electroencephalography (EEG) signals via Steady-State Visual Evoked Potentials (SSVEP) has become a hot topic in research on shared autonomy, but the success of such system has been limited outside laboratory conditions due to a list of constraints [Galway et al., 2015].

The goal of this project is to replicate the BCI system in [Akinola et al.] but using the eye-tracker as one of the input devices. Experiments will be done in a controlled simulated gazebo world.

2 System Setup

This projects uses the following hardware devices:

- A Linux Machine running ubuntu 14.04 LTS (display resolution: 1920×1080 pixels).
- A Windows machine running Windows 10 Pro (display resolution: 1366×768 pixels).

*This work is done under the supervision of Prof. Peter Allen and PhD candidate Iretiayo Akinola at Robotics Lab at Columbia University. You can also find this report in the GitHub Repo: https://github.com/CRLab/bci_ssvep_eye_ws

- Tobii eye-tracker 4C.

Tobbi eye-tracker only provides software dirver on Windows platform, and this is why we will need the Windows machine to drive the eye-tracking device. However, the gazebo simulated world and robot operating system (ROS) can only be run under Linux platform, so we choose to use Lab Streaming Layer to stream the gaze data of the user from Windows machine to Linux machine through network.

We will mount Tobbi eye-tracker on the Linux machine and calibrate it with regards to the screen of Linux. To be more specific, one person runs the software driver on Windows to start the calibrating process; when the Tobbi application on Windows asks the user to focus on center, up-right corner, bottom-left corner, etc., of the screen, the other person will then look at those corresponding positions on the Linux screen (has to use some sort of imagination since the calibration user interface only appears on the Windows machine).

After the eye-tracker is successfully mounted and calibrated on the Linux machine, we write a program in C# to get the streaming gaze data on the Windows machine (Tobbi only provides API in C# for free use) and these data are actually where the user is gazing at on the Linux machine where the eye-tracker is mounted. These data are then streamed through network to the Linux machine for eye-tracking service.

The code for using LSL (sending in C# on Windows, receiving in Python on Linux) can be found in this Github Repo: `https://github.com/CRLab/eye_tracking_device.git`

3 Methods

3.1 Camera Calibration

The biggest challenge and also the biggest contribution of the project is to develop a function to find the pixel coordinate of a point on map, given its corresponding map coordinate and a particular camera position. This function works accurately no matter how the camera is moved and the only request is that the intrinsic parameters and the output image size do not change. In this section, I will explain how we solve the problem by empirically finding the $Intrinsic$ matrix of the RViz camera, formed by (D_x, D_y, u_0, v_0, f).

The code for camera calibration can be found in this Github Repo: `https://github.com/CRLab/camera_calibration`.

3.1.1 RViz Camera Position Configuration

The position of Rviz camera is configured by three 3-dimensional points:

- Eye point: the world coordinate of the camera position on the map.
- Focus point: the world coordinate of the point that the camera is looking at on the map.
- Up vector: the vector in the world frame which always points straight up (-y) in the final output image.

The above parameters determine what we see in RViz.

3.1.2 Homogeneous Transformation Matrix

We first talk about how to find the 4 by 4 homogeneous transformation matrix T_{world}^{camera} to translate a point from world coordinate to camera coordinate given the above arguments - eye point, focus point and up vector.

Define:

$$P = \text{eye point} \quad (1)$$
$$F = \text{focus point} \quad (2)$$
$$\vec{u} = \text{up vector} \quad (3)$$

It is easier to first find T_{camera}^{world} which transforms a point in camera frame to world frame, because the first three columns of T_{camera}^{world} will simply be its normalized $\vec{x}$, $\vec{y}$, $\vec{z}$ axes and append 0 in the end. The last column is simply the origin (eye point) of the camera frame expressed in world frame, appended by 1.

By convention, we define $\vec{z} = F - P = \overrightarrow{PF}$. **The $\vec{y}$ axis is the intersection of 2 planes**: **the first plane passes F and is parallel to $\vec{u}$, the second plane passes P and is perpendicular to $\vec{z}$.** Then, $\vec{x}$ is simply $\vec{y} \times \vec{z}$. Finally, normalize $\vec{x}$, $\vec{y}$, $\vec{z}$ we can get T_{camera}^{world} and we have:

$$T_{world}^{camera} = (T_{camera}^{world})^{-1} \tag{4}$$

3.1.3 Find Intrinsic Parameters

We know that:

$$P^{image} = T_{persp}^{image} \cdot T_{camera}^{persp} \cdot T_{world}^{camera} \cdot P^{world} \tag{5}$$

Or write it in matrix form:

$$\begin{bmatrix} s \cdot u \\ s \cdot v \\ s \end{bmatrix} = \begin{bmatrix} \frac{1}{D_x} & 0 & u_0 \\ 0 & \frac{1}{D_y} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^wX \\ {}^wY \\ {}^wZ \end{bmatrix} \tag{6}$$

T_{world}^{camera} is the homogeneous transformation matrix from world frame to camera frame. T_{camera}^{persp} transforms a point in camera frame to its 2 dimensional image plane. T_{persp}^{image} scales the coordinates of points in image plane to final pixel values. Given a particular camera, there are 5 fixed intrinsic parameters: focal length f, scaling factor from image plane to pixel value D_x and D_y, origin of image plane on the final picture u_0 and v_0. These parameters does not change when changing the physical location of the camera. There are another 6 external parameters in T_{world}^{camera} which are determined by the rotation and translation of camera position.

We can rewrite equation (5) in the following way:

$$P^{image} = C \cdot P^{world} \tag{7}$$

where C is the overall 3 by 4 transformation matrix for a given camera and a given position of that camera. So we have:

$$C = Intrinsic \cdot External \tag{8}$$

where $Intrinsic = T_{persp}^{image} \cdot T_{camera}^{persp}$ and $External = T_{world}^{camera}$.

$Intrinsic$ is a 3 by 4 matrix determined by the intrinsic parameters of camera (D_x, D_y, u_0, v_0, f), while $External$ is equal to the homogeneous transformation matrix from world frame to camera frame. The $Intrinsic$ matrix does not change with the position configuration of the camera, only $External$ does. Because the intrinsic parameters are fixed given a particular camera, but the homogeneous transformation matrix from world frame to camera frame will change with different translations and rotations of this camera.

Thus, we can empirically find the fixed $Intrinsic$ matrix given a particular camera position and at least 6 points with their world coordinates and pixel coordinates. C can be overdetermined by the 6 points by solving a linear system, $External$ can be determined by camera position configuration, and then $Intrinsic = C \cdot External^{-1}$.

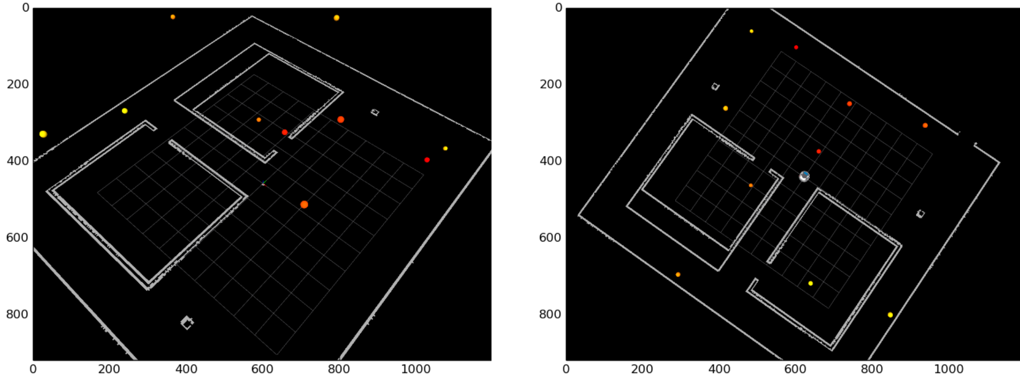
Then, when the camera moves, we can easily get the new $C' = Intrinsic \cdot External'$ ($External'$ is the new homogeneous frame transformation matrix). With C', we can easily get the pixel coordinate of a point in the world.

We first generate 10 randoms points on the map and record their world coordinates, shown in figure 1a. We can then read the pixel coordinates of these points so that we get 10 pairs of world coordinates and pixel coordinates. Arbitrarily choose 6 pairs to form a linear system where $A \cdot C_0 = B$ and A is 12 by 11, C_0 is 11 by 1, B is 12 by 1. For details on how to form these matrices, see this note: `http://www.cs.columbia.edu/~allen/F17/NOTES/Lecture_5_rev_3.pdf`. We can get C_0 which is a flattened matrix of C without c_{44} by:

$$C_0 = (A^\intercal A)^{-1} \cdot A^\intercal B \tag{9}$$

We then append 1 to C_0 and reshape it to a 3 by 4 matrix so that we obtain C. We can use the rest 4 pairs of coordinates to test the accuracy of the overall transformation matrix C for this camera position.

After we make sure that C is correct, we can get the $Intrinsic$ matrix by $C \cdot External^{-1}$, where $External$ is the homogeneous transformation matrix for this camera position and can be obtained using methods in section 3.1.2.



(a) View over 10 randomly generated points with the camera position configuration for calibration.

(b) View over another set of 10 randomly generated points with a new camera position for testing.

Figure 1: Different views over 10 points from RViz.

To validate the correctness of $Intinsic$ matrix for RViz camera, we move the camera to another position which gives us another view of the same 10 points, shown in figure 1b. We subsequently obtain the new homogeneous transformation matrix from world frame to camera frame for this position denoted by $External'$. Then, the new overall transformation matrix from world coordinate to pixel coordinate C' is simply $Intrinsic \cdot External'$. We multiply this C' with all 10 points (appended by 1) and verify if we can correctly obtain their pixel coordinates.

With the empirically found $Intrinsic$ matrix, we are able to develop the function to return the pixel coordinate of a world point no matter how we change the position of the camera.

NOTE: The camera calibration gives the transformation from a 3D point in the ROS world to a pixel point of the RViz image. The important portion (image plane) of the RViz window typically occupies only a fraction of the window. So, there is an extra parameter that determines the coordinate of the top left point that starts the important part of the screen. This parameter (`X0`, `Y0` in `eye_tracking_engine.py`) is used as an offset to get the pixel coordinate of the 3D point in the full screen tracked by the eye-tracking device.

3.2 Eye-tracking Algorithm

The choices are visualized using cube markers in RViz. One dimension of the cube has a very small scale so that this cube can approximate a plane. The other 2 dimensions have the same scale. This cube will be transformed to a convex quadrilateral in the image whose edges might not be parallel.

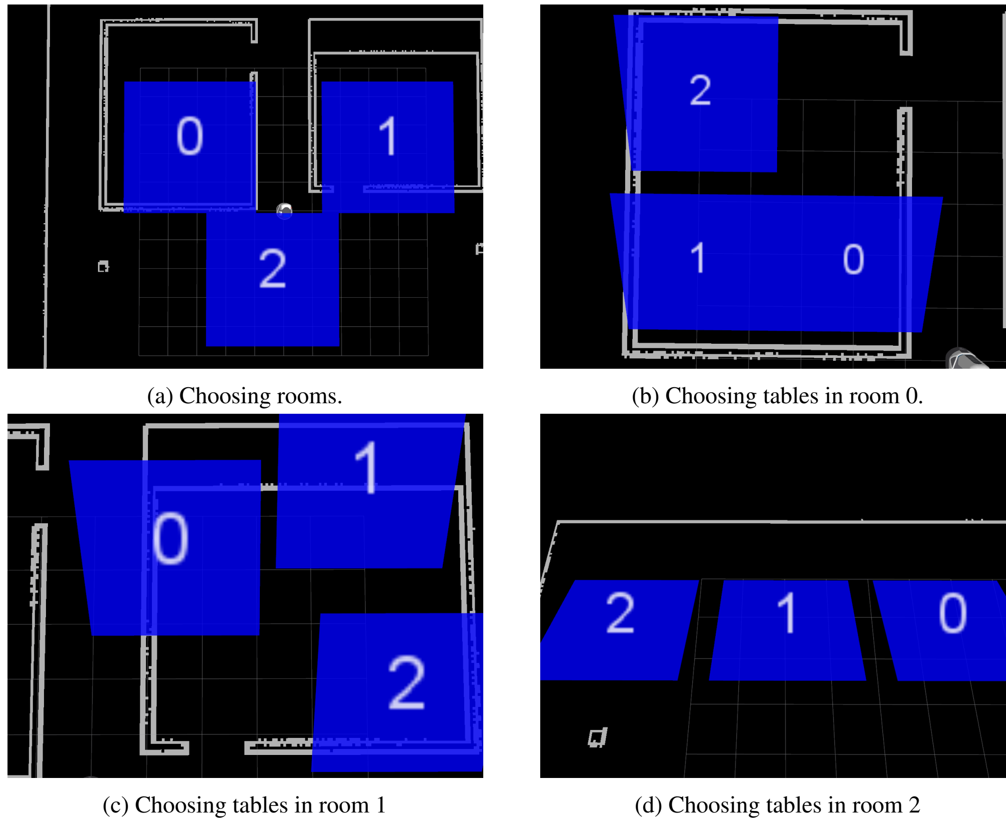
After the gaze data are obtained, we first filter out all the noisy gaze positions, namely, those out of all quadrilaterals. Then we get the mean of these gaze positions and use the mean value to decide the final choice of user.

4 Experiments

We use eye-tracking for two tasks: choosing rooms and then choose a table in that room. The simulation environment is a gazebo world with a predefined map. The system first ask the user which

room to go, shown in figure 2a, and then the camera hover over and zoom into the chosen room and present the choices of three tables, shown in figure 2b to 2d.

The code for running these experiments is under this Github Repo: `https://github.com/CRLab/bci_ssvep_eye_ws`.



(a) Choosing rooms.

(b) Choosing tables in room 0.

(c) Choosing tables in room 1

(d) Choosing tables in room 2

Figure 2: Different choices presented by the system using cubes

5 Accuracy

5.1 Time Accuracy

The eye-tracker collects around 87 gaze positions in 1 second, which means it can collect 1 gaze position in 0.0115 second. As long as we run the eye tracking process for a period over this value, it is able to clearly identify the choice of the user.

5.2 Location Accuracy

The location accuracy means the ability of the eye tracker correctly locating the position on screen where the user is looking at. These performance is affected by the following factors:

Eye-tracker calibration This is the error coming from setting up the eye-tracker on screen. The calibration process provided by the Tobbi software driver introduces error and this is internal with the eye-tracker, which is small enough to be ignored and cannot be optimized.

Screen size discrepancy Remember that we are calibrating the eye-tracker on a Linux machine screen and running the software driver on a Windows machine. Because this two machines have different screen size, we need to scale the gaze data in one screen (1366 $\times$ 768 pixels) to another screen (1920 $\times$ 1080 pixels). In addition, the person looking at the the Linux screen can only use imagination to guess the relative positions of calibration markers shown by Tobbi software on

Windows machine, which has a much larger calibration error than doing calibration using a single screen. These errors are much more significant than those brought by regular eye-tracker calibration on a single screen. It can be reduced if we choose the 2 screens to be the same. But the error introduced by invisibility of calibration markers on Linux cannot be further avoided.

RViz camera calibration Another source of error is the process of calculating the intrinsic parameters of RViz camera. While doing camera calibration, we need to manually read the pixel coordinate for several situations:

- Read the pixel coordinates of image plane in the RViz window.
- Read the pixel coordinates of the 10 random points. We define the points to be marker balls with a scale of 0.2 meter on map. After moving the camera to the position where we do the calibration, these balls can have a diameter varying from 20 pixels to around 10 pixels. Reading the pixel coordinate of ball center can be erroneous. Error can be reduced by reducing the size of balls but it might result in balls which are too far away from the camera being very small in image.

The matrix calculation for camera calibration can also accumulate errors from rounding. The overall maximum error for camera calibration, namely, finding the pixel coordinate of a world point is 4 pixels (from multiple tests).

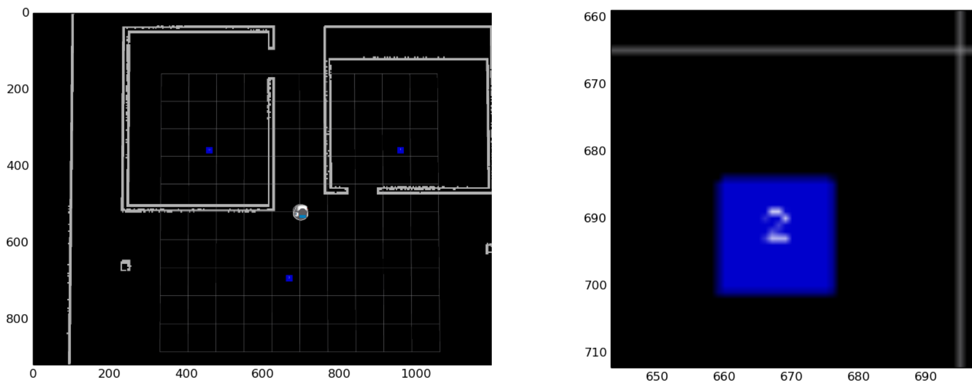
Bounding box vertices calculation Calculating the vertices of a bounding box from the cube markers' quaternion and pivot point position can also introduce errors due to matrix computation.

5.3 Accuracy Test

We create a test to find the minimum pixel sizes of markers that the whole system can work correctly, taking into account all these sources of errors mentioned above. This test is under the assumption that the eye-tracker has been properly calibrated as said in section 2, which means that all external conditions should not change, for instance, height of seat, experiment subject, etc. A particular size of marker which works properly for a user might not work correctly if he changes the seat height or if there is another user of different figure, etc., unless eye-tracker is calibrated again.

We use binary search to continue decrease the size of markers until the system cannot correctly identify the user's choice. We carry out this experiment under the choosing-room scenario and we find that the minimum size of the marker in image that the system can classify is around 16 pixels, as shown in figure 3.

The code for this test is under the same Repo as mentioned in section 4.



(a) Test under choosing room scenario.

(b) Zoomed in view of marker for room 2.

Figure 3: Accuracy test.

6 Conclusion

We develop a method using camera calibration to empirically find the intrinsic parameters to return the corresponding pixel coordinate of a world point. This method works accurately under several constraints which makes sense in real world application of assistive robots. We then use eye-tracker as an input device with the BCI system [Akinola et al.], enabling the user to make choices with their eyes. The main contributions of this project are:

- A function which can return the homogeneous transformation matrix from world frame to camera frame, given the position configuration of RViz camera.
- A function which returns the pixel coordinate of a world point given its coordinate on map and the camera position. This function utilizes the previous one.
- An integrated BCI system with eye-tracking.
- Analyses on the time accuracy and location accuracy of this eye-tracker.

7 Future Work

7.1 Constraints

For this method to work accurately, there are three constraints:

- The position of the image on screen returned by RViz camera is known and unchanged.
- The size of the image is known and unchanged.
- The intrinsic parameters of RViz camera are unchanged.

These three constraints are reasonable assumptions in real world application of assitive robots for disabled people. We would not expect the diabled users to drag the RViz window around or change the layout of RViz, otherwise we would not need the help of BCI or eye-tracking. Thus, constraints 1 and 2 are guaranteed.

The way RViz change its view is by moving the physical location of the camera instead of changing the focal length f, and since the size of the output image is also fixed, the scaling factors and image origin (D_x, D_y, u_0, v_0) are fixed, constraint 3 is satisfied.

Another way to figure out the intrinsic parameters of RViz camera is to look into the code of RViz. This alternative way has its advantages and drawback. The advantage is that we can write a ROS message and topic to reflect these information while running the system, so that we do not need to worry about the change of image size, because the RViz controller node can always publish the updated information on the topic. Namely, we can eliminate constraint 2. But in the current system, if we change the size of the output image, D_x, D_y, u_0, v_0 are changed, so that we need to calibrate the camera again. Another advantage is that we eliminate the error source of reading the ball marker pixel coordinates, which gives a better accuracy than the empirical calibration method. The drawback is that we do not know how these intrinsic parameters are represented in RViz controller. I did look into the code a lot, but did not manage to locate those features which reflect the intrinsic parameters. Future researchers can try finding the intrinsic parameters in this way to see if it turns out to be better than empirical camera calibration in terms of accuracy and time complexity.

If we want to eliminate constraint 1, we will need to understand how the RViz window is generated and how the layout is controlled by the code. Currently, we need to know the distance from the image to the right of the screen x_0 and to the top of the screen y_0. If x_0 and y_0 are changed, we need to update the corresponding `X0` and `Y0` in `eye_tracking_engine.py`. Note that we do not need to calibrate the camera again in this situation because the intrinsic parameters do not change when we move image plane on the screen. If we can get updated values of x_0 and y_0 from the code, than we can write a topic to broadcast this information to the eye-tracking node which uses it.

7.2 Time Complexity

Finding the homogeneous transformation matrix from world frame to camera frame uses matrix inverse in multiple places, which has the time complexity of $\mathcal{O}(n^3)$, where n is the length of the

first row of this matrix. For each marker, we need to calculate all its 4 vertices and then transform them into pixel coordinate, which needs the result from $External$ matrix. This calculation takes relatively a long time to finish. Future researchers can focus on using optimized methods or packages to improve the computing time.

8 Code and Demo

The code for this project can be found in the following GitHub Repos:

- `eye_tracking_device` (https://github.com/CRLab/eye_tracking_device): illustrates how to setup the eye-tracker across two machines using LSL.
- `camera_calibration` (https://github.com/CRLab/camera_calibration): elaborates and goes through the process of camera calibration.
- `bci_ssvep_eye_ws` (https://github.com/CRLab/bci_ssvep_eye_ws): top level workspace of this project, contains the code for experiments and accuracy test.

A demo video of the experiments and accuracy test can be found in this YouTube link: https://youtu.be/OH_F1D3WOOA

References

I. Akinola, B. Chen, J. Koss, A. Patankar, J. Varley, and P. Allen. Task level hierarchical system for bci-enabled shared autonomy.

L. Galway, C. Brennan, P. McCullagh, and G. Lightbody. Bci and eye gaze: collaboration at the interface. In *International Conference on Augmented Cognition*, pages 199–210. Springer, 2015.