Eye Tracker Game Controller

Nika Eržen (63150400), Anej Placer (63080315), Kevin Sedevčič (63140373)

June 13, 2016

1 Goal

The goal of this project was to create a plugin for the Pupil-Labs eye tracker. The plugin would allow the user to perform keyboard actions by only using their sight, therefore being able to play any computer game hands-free. The plugin needs to recognize simple and advanced actions.

2 Installation

The plugin takes advantage of the plugin loading system where you only need to put the plugin into the proper folder (capture_settings/plugins/plugin.py), run the capture application and select the plugin you want to run from the drop down menu.

Python, along with libraries like scikit-learn and numpy was used for development.

3 Workflow

After calibrating the eye tracker and loading the plugin, it begins to receive gaze data (location on the screen the user is looking at) every frame. The gaze history is of a fixed length. The data is run through a smoothing filter for denoising purposes. Since the calibration doesn't always work perfectly, we added our own additional calibration. Then comes the learning portion where user inputs nine basic actions and two special ones (there could be arbitrarily many if needed) multiple times for the learner to construct a prediction model. After all the prerequisites are complete the plugin starts to check if any action happened. If it is detected the appropriate key press is simulated. The application can also listen for actual keypresses in a separate thread, so you are able to stop or pause it.

3.1 Calibration

First a consistent amount of measurements (gaze positions) is needed to start the calibration. The gaze values are between 0 and 1 for the X and Y axis. When sufficient amount of gaze positions is passed to the calibrator, it finds the min and max values for each axis. A simple median from the two gives us the center. We chose to calculate the new center because the values can be slightly shifted from the real center of the final interval (0.5) let's say 0.3 and 0.5,

where the center is 0.4. So here a linear normalization is not possible. Then the proportion of the points which are left and right of the new center need to be found. When the initialization is done, the distance from min (if left from the center) and max (if right from the center) is calculated for every point which is transformed to the interval between 0 and 1. Obviously there is a chance that values bigger than the first min and max are found. Those are trimmed to the min and max.

3.2 Learning

The screen is divided into portions according to the Figure 1. These are the basic nine actions. The special actions can be of whatever shape, but we chose those to be a V-shape (see Figure 2 and a circle (not practical).

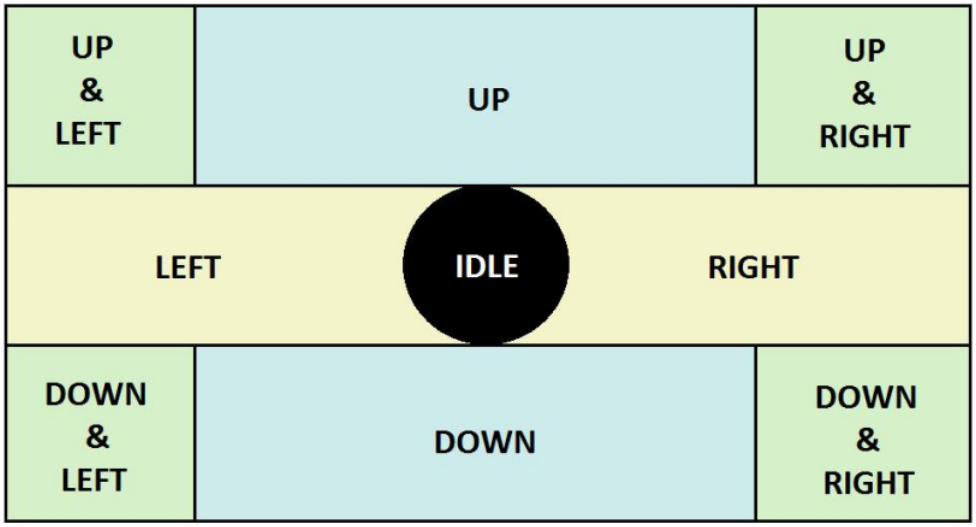

Figure 1: The gaze map.

The user needs to set the amount of repetitions per action so the learner gets the extracted features for each take and constructs a model which is then used to predict actions. For each take the user needs to press the mouse button before beginning the action, and release it after completion.

3.2.1 Feature extraction

Features are extracted from a portion of the recent gaze history (how big that portion is, is a matter of testing). The following features are extracted:

- coordinates of the centroid point of the last few points in the gaze history (denoted as short gaze history) and standard deviation of the distance of points in the short gaze history to the centroid point; (3 features: x and y coordinate and standard deviation);

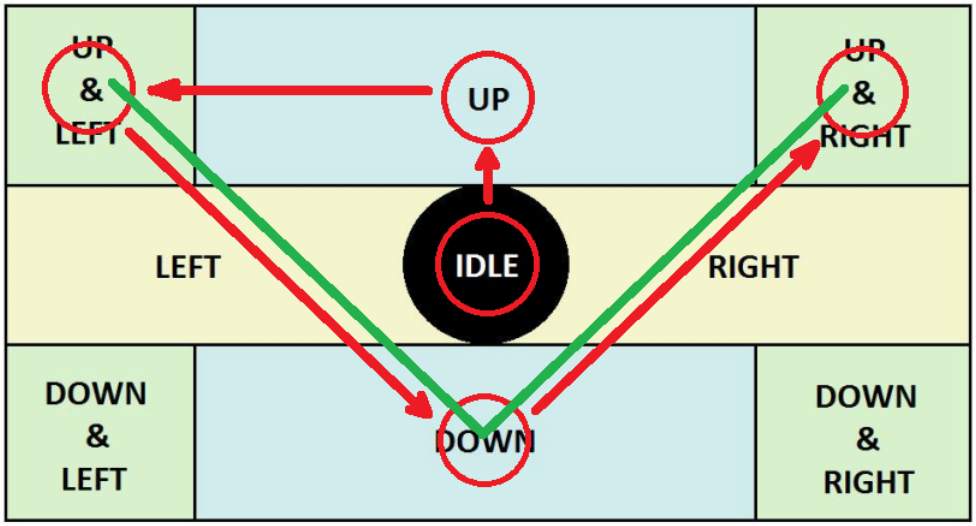

Figure 2: The special action trace.

- coordinates of the highest and the lowest point in the gaze history; (4 features: x and y coordinates of two points);
- coefficients of a polynomial curve (of second degree) fitted on the gaze history points and the error of this approximation given as residuals of the least-squares fit; (4 features: 3 coefficients and residuals).

The first four features are more important for predicting simple actions. Since simple actions were defined as an approximately steady gaze at a point, the standard deviation of the distance helps determine whether the gaze is steady (if it is, it should indicate a steady action) and the centroid point should determine which of the simple actions the user wanted to execute. Because simple actions take less time than special actions only short gaze history is used.

The remaining seven features are designed for special actions. Chosen special actions were defined as a V shaped gaze track. These features should point to the approximate shape that the gaze history points make, the size of the shape and how accurate the user was while tracking the shape with his gaze.

3.2.2 Predictor

To create the prediction model the random forest classifier with up to 20 estimators was used. After plugging a set of features in to the predictor a predicted class (action) is returned, along with the probability that this is the correct action.

3.3 Action selection

There will always be an active action. If it's not a special one it will be one of the location bound simple ones, the area of which covers the whole screen. Even so, the features, extracted

from gaze data, are passed in to the model every frame. In the case that predictor can't predict an action with a high enough certainty, the process of secondary check for simple actions begins. On this step we again use the centroid point of short gaze history and the standard deviation of the distance. If the latter is lower than the estimated threshold the action is predicted based on the position of the centroid point.

3.4 Action simulation

To simulate key presses using Python, we used a linux program *XTE*, which allows key-down and key-up actions for any key available. Every action has an accompanying key which is to be "pressed" if that action is chosen. Before the next key press is simulated, the previous one needs to be stopped. In the case of actions like up-left and down-right two keys are pressed and both need to be stopped before the next key press.

4 Results

Because we ran into many problems with the eye tracker, the testing was not easy. We however managed to test it by recording action takes from multiple people, train the predictor and perform pre-recorded actions. The results were mostly good (actions recognized; possible to play a game) except for the more complex special actions. It would all be much easier to develop and test if there weren't so many unexpected problems. When using linux no less!

5 Problems

We had some serious problems with pupil labs software. We tried using it on Linux OS (Ubuntu 14.04 as virtual OS and as independent OS and Ubuntu 16.04 as virtual OS). We followed the installation guide for dependencies. We also installed OpenCV library (newest and older versions, since the new one doesn't work with the pupil capture program). Tried multiple versions of TurboJpeg and other relevant libraries. We also tried installing older versions of pupil labs calibrator (e.g. pupil_calibrator 0.6.20). We always got the same error:

world - [ERROR] uvc : Turbojpeg jpeg2yuv error: Premature end of JPEG file
world - [ERROR] uvc : Turbojpeg jpeg2yuv error: Corrupt JPEG data: premature end of data segment.

We tried solving the error by installing drivers for the eye camera and we searched online for answers but no luck. We finally managed to find functional version of pupil labs for Windows but we didn't manage to find a way to implement our work in the program, since the plugin importer seemingly doesn't work on bundled version. We however did manage to make recordings and use them in testing.