# Glance

Biometric Systems and
Multimodal Interaction Report

luci.1665528

## Contents



## 1 Introduction

Although we (thankfully!) don't perceive it, because of the anatomy of the human eyes, when we are looking at something only a small region of the retina, the fovea centralis, is actually producing a sharp image of what we see. This is why in order to "take in" what we are looking at we must scan our object of

observation, by quickly moving our eyes to fixate on the details that interest us. These movements are called saccades, French for *jerk*, because the fixation point jumps from place to place as fast as the eyes can move sometimes even over or undershooting since the movement can't be corrected "on the fly". This scan pattern is interesting because it can tells us a lot about how we carry on visual tasks such as reading a text, browsing for a product, looking for something in an image, interacting with a person etc. This information can be used for designing GUIs, organizing the layout of a piece of content, placing and designing "eye catching" ads and much more.

In this project we are exploring the possibility of utilizing the scan pattern

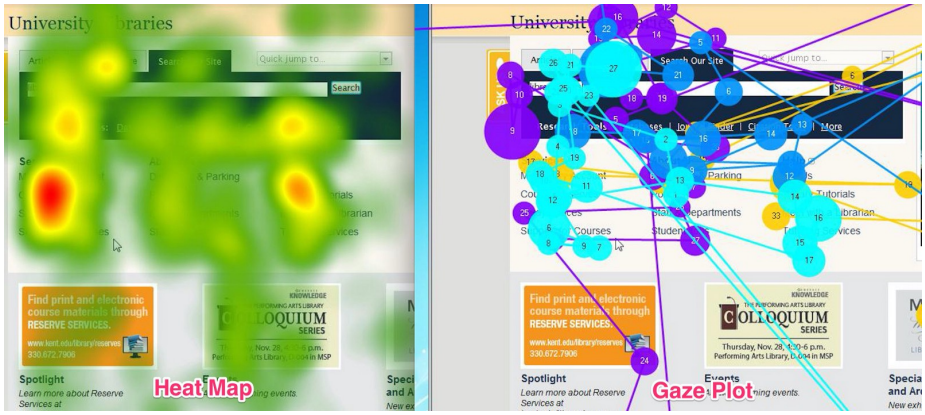


Figure 1: An Example of Heatmap and Scan Pattern for a Website

of an artificially constructed sequence as a behavioural biometric trait. We are also looking at a possible use of gaze as a way to interact with a machine in a multimodal fashion in particular as an additional gaming input along the traditional controllers.

# 2 Description

We will now proceed to take an overview of the presented modules.

## 2.1 Biometric Systems

The main idea behind using gaze tracking in order to perform verification, is that the scan pattern of each person is unique enough to be used to recognize them. However we don't want to have a lengthy enrolment process collecting tons of data about a person to train a recognizer since a normal user probably wouldn't bother going through it. Instead we are piggy backing on a function that the brains performs automatically and use it for our ends. In particular we are using the fact that our gaze tends to fall on familiar faces when shown a mixed set of familiar and unfamiliar ones [1]. What we are trying to test is

if that's true in general, i.e., if familiarity needs not be restricted to a face but anything the user "holds dear".
The user is thus asked to pick a small number (5+) of images that depict something that he knows well. During the enrolment process these images are shown, one by one, alongside an image classified as a neutral stimuli (no shocking images, no dangerous animals, bright colors etc.) and the gaze of the user is recorded. Pictures of the user's face are also taken to train a LBPH recognizer. The recognition happens in two phases: first the user is identified by the face recognizer, the identity claim is thus made by the system itself. The user is then shown the same sequence of images shown during the enrolment ( but with different neutrals ) and their gaze recorded. This trace is matched against the enrolment' one and we then use a simple threshold to accept or reject the user.
The threshold on the identification is kept purposefully low since we don't actually care that the user can make a different identity claim since he wouldn't be able to back it up during the verification phase. We also chose to only utilize the first match during face recognition cause we don't want the user to go through possibly many sequences of images that may not belong to them.

## 2.2 Multimodal Interaction

Since gaze tracking can now be done pretty fast even on non specialized hardware we decided to use it as a natural input device in a gaming context. Especially when considering first person games it seemed the obvious choice that our character gaze should follow ours, after all, it's weirdly owl-esque that in order to move the camera around our avatar either rotates their head or entire body.
Thus we imagined to split the controls in two. The mouse movements are now binded to the lower part of the body, i.e. from the neck down, while the gaze controls the head and eyes. Notice that we don't actually make a distinction between the 2 since we are only interested in the gaze position itself, but we record both the head orientation and gaze vectors so if one wanted they could animate head and eyes separately.
It's obligatory to note that this solution only works well for small angles around the centre of the screen. The reason why is that our screens can't move of course so if we look, for example, directly to the right our characters would follow our gaze and look correctly at what is on their right, but we'd be looking at a wall. Other solutions are possible, we could for example try to detect the object in our fixation point and try to bring it to the centre of the screen and these are definitely worth exploring in a future iteration but for now we are focusing on a simplified version.
In this first implementation we decided to use the mouse to control the camera rotation and the gaze to to determine where we are looking to interact with the game.
Feedback from the first users brought up that this system is in fact unusable, so a second implementation has been done where the system uses the gaze just a secondary input to control part of their character in a simpler, albeit less cool

way.

# 3 Architecture

## 3.1 Overview

The system is built using Unreal Engine 4 both for the Biometric and Multimodal part. The choice of implementing the Biometric project in a game engine is double : mainly for organizational purposes since it favours code reuse and we don't have to worry about the OS we're running on thanks to UE4's Hardware Abstraction Layer ( though it's more software abstraction ) and second Virtual Reality is opening up a whole new world of GUI design that is no longer limited to the WIMP model and I wanted to try and take a shot at designing a program that is meant to be explored in a fully 3D environment instead of a flat 2D one. We rely on the excellent OpenFace facial behavior analysis toolkit to perform pose and gaze tracking. The library performs landmark detection using a Multi-task Cascaded Convolutional Networks (MTCNN) and gaze tracking using a Constrained Local Neural Field (CLNF) and exposes a clear API to get information about the detected faces. We just feed it the stream from the webcam and retrieve the necessary information by the appropriate calls. A manager wrap this library to simplify the interaction with the game code and some glue code hides away the implementation since the engine compilation settings conflicted with the library itself. We thus need to pay an extra step of indirection, but at least we got it to work.

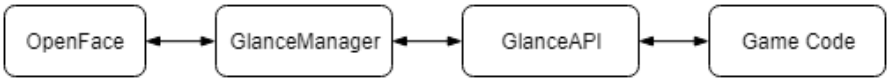


Figure 2: The Indirection Layers

To perform face recognition we make use of OpenCV's extra module simply called 'face' . This module implements a variety of recognizers (EigenFaces, FishersFaces, LBPH). We chose to use LBPH since it's robust to differences in illumination and can be updated with a new set of user faces without having to retrain the whole model. The user pictures are retrieved just before the end of the enrolment process. To detect the face we again rely on the OpenFace library which internally performs face detection by using the trained MTCNN. This is a bit of an overkill since the user is going to be mostly facing the camera, and in fact the library can be set up to utilize Haar cascades or a trained HOG SVM but since when we are snapping the pictures we'd been performing gaze tracking anyway we stick with the MTCNN.
Once we have recorded our traces we need to compute the similarity between them in order to determine whether to accept or reject the identity claim. To

do so we rely on LBImproved, a simple library that performs Dynamic Time Warping. We just feed it the base and trial series eventually adjusted as to have the same length ( by interpolating points on the shorter series ) and retrieve the distance score, this score is first processed based on the settings input by the user and a confidence score is then output.
For what concerns the gaze as input we again make use of OpenFace to get the gaze angle as an $\bar{g} = [x, y]$ vector (we don't need a z since these angle are given relative to the screen plane ). We then perform some adjustments to compensate for things the library can't know about in particular the distance of the user from the screen.

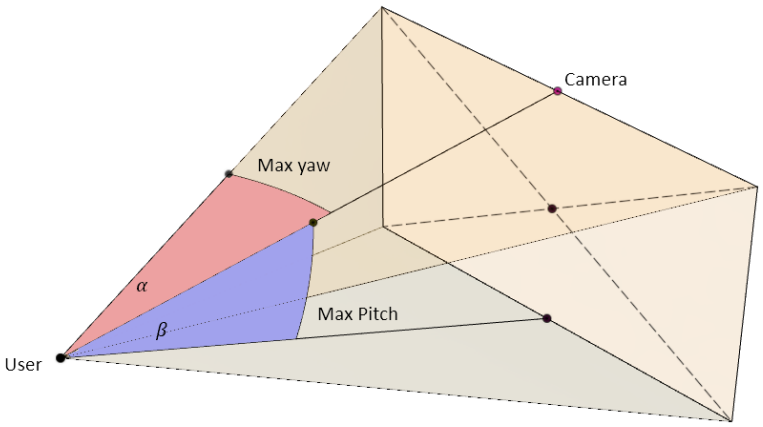


Figure 3: The Observer Model

These values determine the Max Yaw and Pitch that the user's gaze can assume. We can of course change these values to not reflect the actual geometry to obtain different effects. This is not used for the Biometric System' part where we want it to be as close to reality as possible but it's useful to adjust the in game gaze control.

## 3.2 A closer look

We will now explore some of the classes that make up the game code.

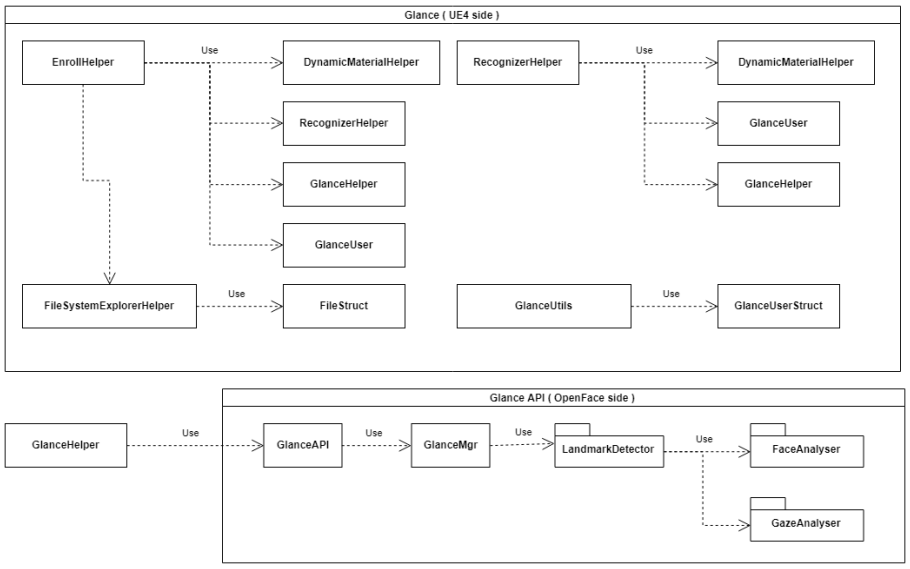


Figure 4: The Classes of Glance

The classes have been split into trees depending on the use cases. We will now explore them one by one.

- **GlanceHelper Tree** : This class acts as a glue between UE4 and the OpenFace package. The only 2 methods that are actually relevant for the game code are **GetCurrentAngle()** and **GetHeadOrientation()**, which are in fact the only methods exposed to the unreal blueprint environment.

- **EnrollHelper tree** : the enroll helper is called during the enrolment process. At the end of this case a new user is added to the system. The caller must provide a username, it is then prompted to select the user images through the use of the FileSystemExplorerHelper. The user is then prompted to go through the tracing of their gaze. The system draws the images with the help of a DynamicMaterialHelper class. The way this is done is a bit unorthodox since we're dynamically rewriting texture data but works pretty well! After tracing the gaze the EnrollHelper snaps pictures of the user, it thus makes use of the RecognizerHelper to store this data for the recognition phase. All throughout the process this class makes use of the GlanceHelper to get the data about the gaze as well as the user pictures, that are saved as OpenCV Mat .

- **RecognizerHelper tree** : the recognizer helper handles the recognition and verification process (but the name would have been too long). The user is first recognized by face recognition (this is all handled by OpenCV) and the correspondent GlanceUser is constructed from the database. The

user must then prove their claim, the class uses the DynamicMaterial-helper to draw the images on the screen while it records their gaze, it then computes the distance and outputs the result on the screen.

It's worth spending a word on the way UE4 handles game logic programming:

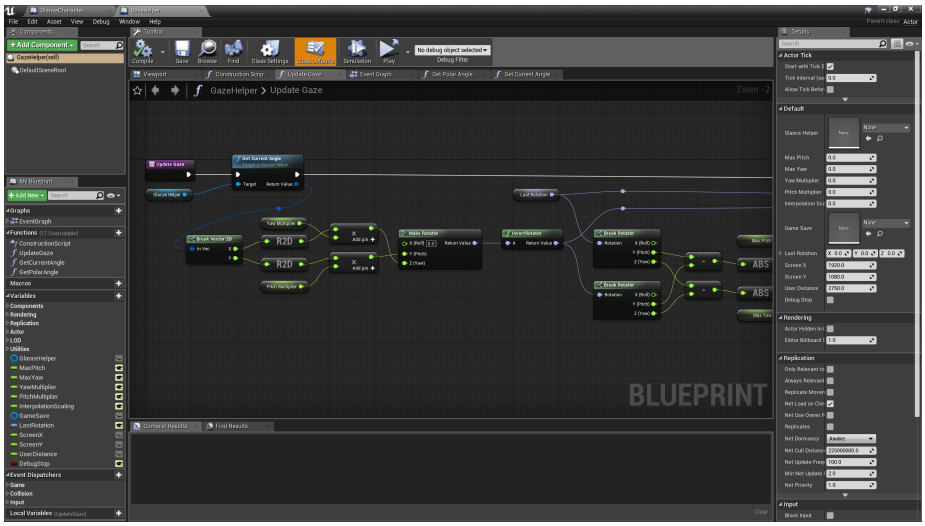


Figure 5: The Unreal Blueprint Environment

Here we can see the blueprint for an Actor, an entity in the game engine. Blueprint classes are for all intents and purposes C++ classes with additional metadata that gets interpreted by the engine before the actual compilation process begins. Blueprints make some tasks very easy, such as interacting with the game world entities, but fail when additional complexity is needed often getting really messy really quickly. The class in the figure is the GazeHelper class, an additional wrapper to the GlanceHelper and basically acts as a filter to provide a smoother "output signal" from the raw data of the collected gaze which often presents some errors that must be hidden to the end user.

# 4 Usage

We will now go through a sample session to explain the system usage.

## 4.1 Biometric Systems

A user that wishes to enroll must open the enrollment level. It will be presented with a simple 3D space composed of an elevated platform and a screen. The user must first select the images that he wishes to be shown and will be used during the recording of the trace, they can also select a username if they wish

to otherwise a random one will be assigned to them. After they've selected the images they will be able to start the enrolment. To do so they must go on the platform and interact with the button by pressing E. The screen in front of them will show the images in succession while recording the trace. When that's done, the system will ask the user to look directly into the camera while it is snapping the pictures. This concludes the enrolment process.
After a user is enrolled, they can open the verification level. They will be presented with the same space as before. Again using the button, the first press will snap a picture of the user, pass it through the recognizer and output the identity claim. The second press will start the image show. The system will record the trace and output the result of the matching process on screen.
The user can press P to pause and Exit when they're done. The relevant settings to this module are only the interval at which the image change on screen.

### 4.2 Multimodal Interaction

Two modules have been implemented for this part. The first module is labelled "Debug Level" while the other is the "Play Level". The Debug Level is so called because it's an implementation of the aforementioned FPS, however it's not very usable and thus has been relegated to just being a demo of what can be something interesting to explore with, for example, a VR headset, where the screen surrounds the user's FoV. The Play Level is a minigame based around a reduced version of the gaze controller. The game is a simple shooting game. The user uses the gaze to move the cannon CW or CCW, the mouse to pitch it up or down and the spacebar to shoot. Additionally the user can bring up a gaze controlled widget by pressing Q. The widget consists in a sectored wheel. The user can then look at the section of the wheel he is interested in to make a selection. Different projectiles can be selected this way.

## 5 Functional Details

### 5.1 Global Settings

A number of parameters control the user experience:

- **Face Threshold**: this value determines the threshold for the face recognizer, under which the user does not passes the first phase of the recognition.
- **Gaze Threshold**: this value determines the value of the DTW distance used to decided whether a user is who they claim to be.
- **Ray Distance**: used in the debug level. The ray distance is used to intersect objects with the gaze vector.

- **Interpolation Scaling**: the speed at which the character gaze follows the user gaze. This value is used to stabilize the camera around the user gaze vector. Since the gaze detection is bound to be a little imprecise we don't want the camera to bounce around due to errors. Thus the desired position is reached in a certain time that is dependent on the angle between the last and current vector and this interpolation factor. The greater the angle the faster the camera will follow, so big movements feels responsive, but small movements take a longer time making the experience feels smoother.

- **Max Yaw** and **Max Pitch**: these values are used to bound the camera angle such that the user can only look at what's on the screen. The distance from the screen and screen size determine those values, but they can be tweaked to the user preference.

- **Yaw Multiplier** and **Pitch Multiplier**: these values are used to tune the dependency between the gaze angle and the camera angle. The relation is linear so $\vec{C_a} = \vec{M} \odot \vec{G_a}$ where $\odot$ is the element-wise multiplication operator.

- **Image Show Period**: this value determines the time the user images will be shown on screen. Note that this value should not be changed on a per user basis, unless one wants to get inconsistent results when comparing users' scores.

## 5.2 Gaze Estimation Calibration

The user can help the system perform better by going through a calibration procedure. The procedure attempts to correct for errors by collecting gaze data for fixed points and then using the known points to subdivide the screen in sector on which the gaze gets interpolated to produce the final position.

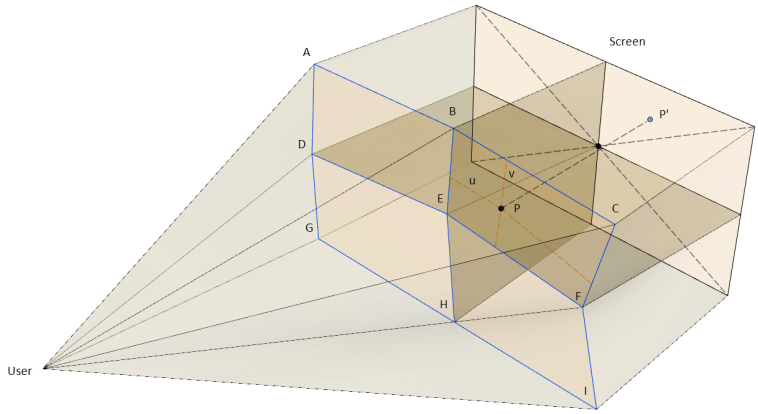


Figure 6: The Calibration Procedure

The programs ask the user to follow a dot around the screen that fixates in the points $A$ to $I$. This effectively subdivides the screen in 4 sector. The positions of the points are given by the gaze detector thus they do not necessarily form rectangles (in fact they almost never do) but they do form quadrilaterals. All we need to do is thus map points from the quadrilaterals to the screen sectors. In short:
Given a point $P$ determine the quadrilateral it belongs.
Starting from the top-left corner this gives us 4 vertices $V_0$ to $V_3$ with positions $Pos(V_i)$.
Determine the distance of $P$ from the 4 edges of the quadrilateral $V_0\bar{V}_1$ to $V_3\bar{V}_0$.
Compute the $uv$ coordinates of the point $P$, that is for

$$u = \frac{\text{dist}(P, V_3\bar{V}_0)}{2(\text{dist}(P, V_3\bar{V}_0) + \text{dist}(P, V_1\bar{V}_3))}$$

and similarly for $v$ but using the top and bottom edges.
Use the uv coordinates to map the point $P$ to a point $P'$ on the screen, adding 0.5 when needed since we are working on a quarter section of the screen.

## 5.3 NPC character movement

The minigame includes Non Playing Character that need to reach random objectives around the map, given a random starting position. This is a well known problem in game design and in fact it would've been much easier to use a premade solution, such as navigation maps and A* pathfinding algorithm. Unfortunately the characters in our game effectively float around the map on a collision free mesh, thus a navigation algorithm needed to be rewritten. Since

the well known algorithms are not very interesting to reimplement an adaptation of one in the class of Bug Algorithms has been used.

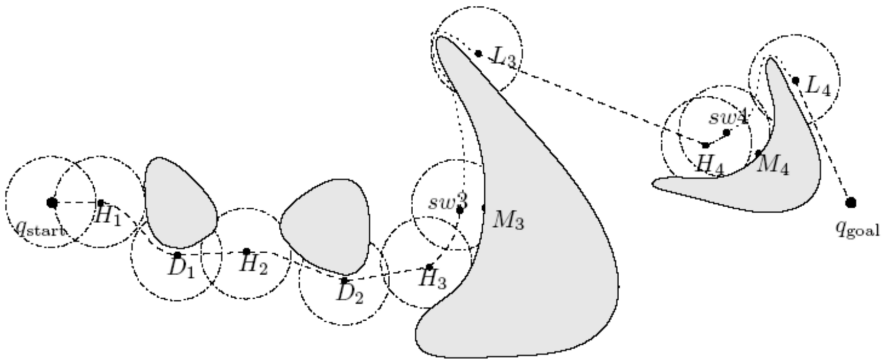


Figure 7: A Finite Sensor Range Algorithm (16-735, Howie Choset with slides from G.D. Hager and Z. Dodds)

These algorithms are mainly used in robotics since they are greedy pathfinding procedure that do not require the character to be omniscient about the topography of the map and are in fact used, for example, for unmanned rovers that need to explore an area. The boat in our game doesn't have sensors of cours and instead makes use of raycasting in order to determine forward and lateral collision, additionally, it checks for collision towards the target. At each step it then determines the vector toward which to steer to try to reach its destination. While this algorithm works well when there aren't many obstacles it is far from optimal if the area is very crowded since it will have the boat try to circumnavigate each obstacle that is on the direct path between the current position and the target position.

# 6 Experimental Setup and Data Collection

In order to test the system a small (7) number of subjects have been asked to go through the enrolment procedure. Each subject chose exactly 5 images. They then tried for 3 times to be recognized using their own images and the traces and scores have been recorded. Each subject then tried to impersonate the others, by looking at their image sequences, again scores and traces have been recorded. All the tests have been performed in well lit areas, using the same hardware, and trying to reduce the variance between the distance to the screen. The data has then been collected and analyzed.

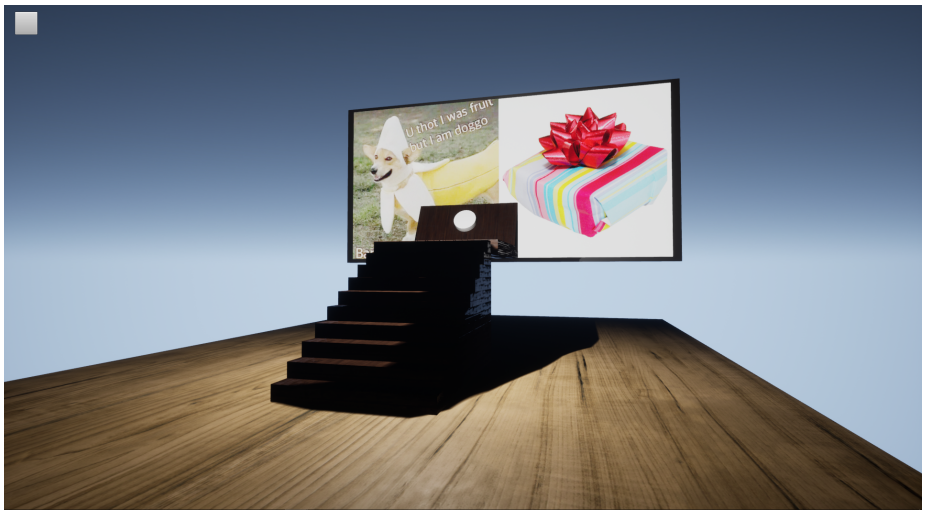


Figure 8: The Enrolment and Recognition Platform

We can first take a look at a raw trace from the enrolment of one user.

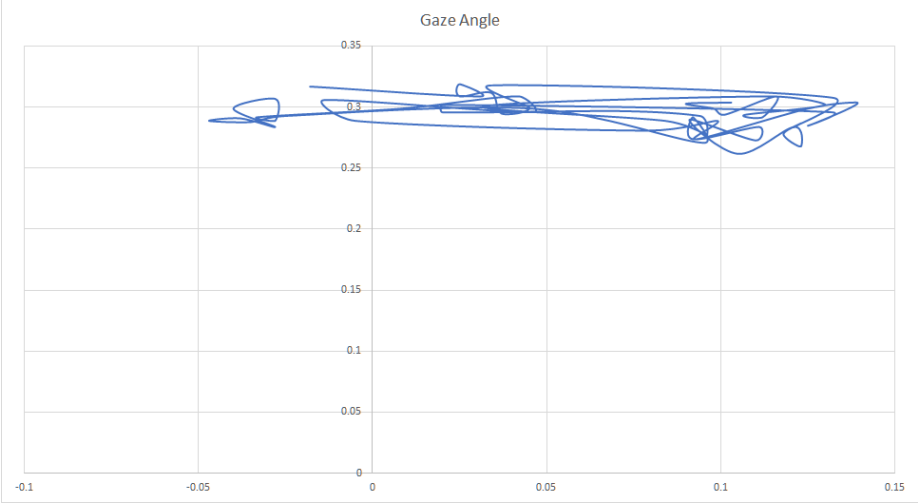


Figure 9: An Example of a Gaze Trace

This is what is collected from the program without any adjustment, the line connects the data points as a pair of $(x, y)$ angles in radians. This isn't very useful per se since it's not easily comprehensible at a glance. However the pattern becomes clear if we separate the $x$ and $y$ angle and plot them separately over time.

Now we can recognize that the user mostly kept their gaze at a fixed height

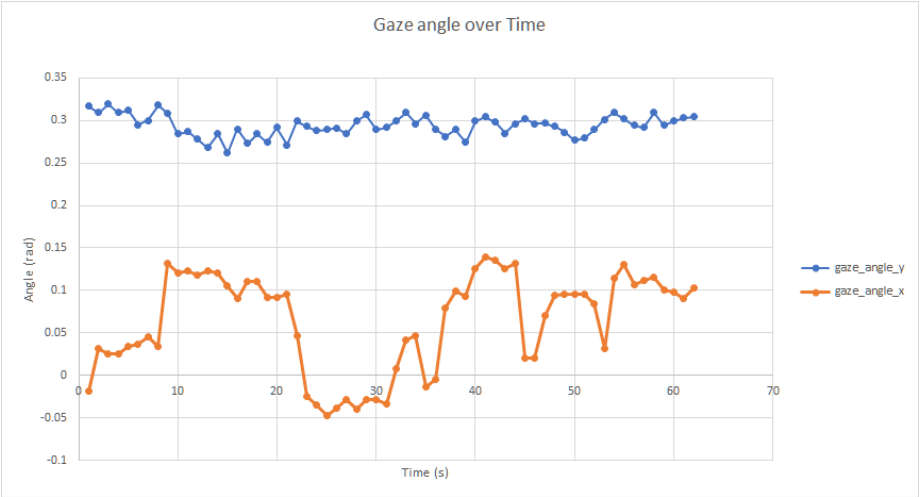


Figure 10: The Gaze Angles Over Time

but darted from left to right to look at the images in a recognizable pattern. One could be tempted to discard the $y$ information since it does not seem to be very indicative of the trace, however this is just an isolated case since the user can fixate on any part of the image making the $y$ angle just as important.

# 7 Results

We now present the results of the tests using the standard methods. Again we don't report the results for the face recognition part since those are well known in the literature.
The first thing we are interested in is the FAR and FRR.

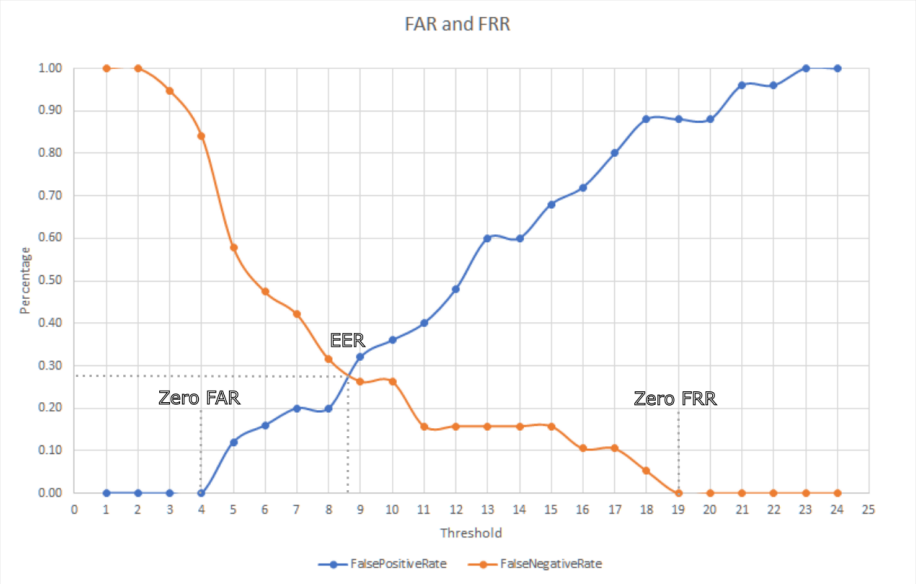


Figure 11: The FAR and FRR

Note that the graph is "flipped" along the y axis, i.e., increasing the threshold in our case means lowering the bar for acceptance since that is the maximum distance we are comparing against for deciding whether a user is genuine. While the curve for the FRR has an expected sharp drop at a value of 4 the growth of the FAR curve looks almost linear ($R^2 = 0.98$). This is not something that we want of course. This is even more evident when we look at the distribution of the scores for the genuine and false probes.

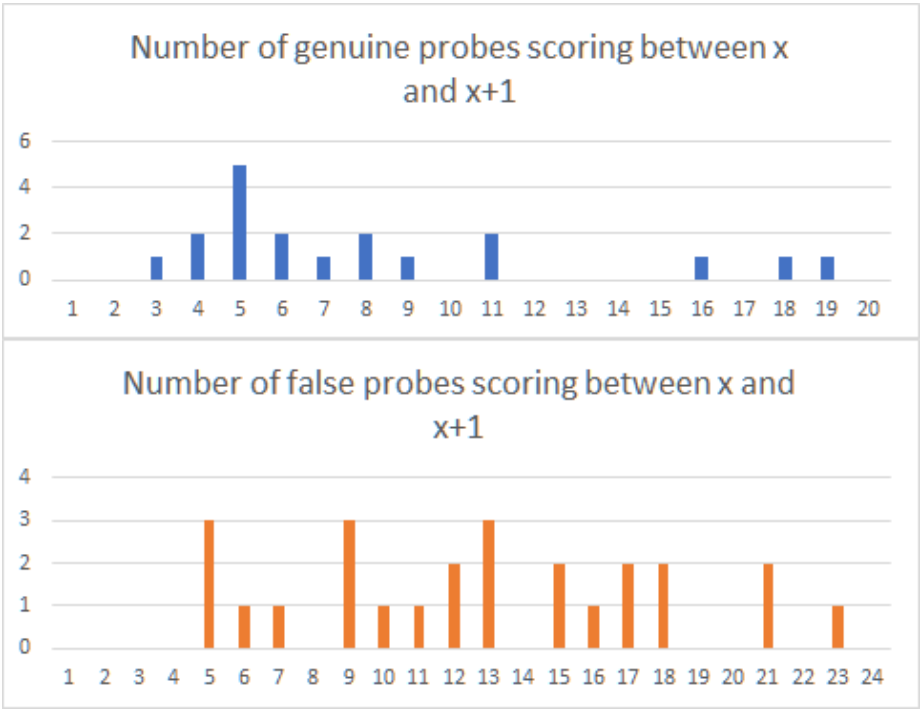


Figure 12: The Results of the Probes

For the genuine claims we have a nice distribution centered around the value of 5 with a few outliers on the right. The false claims however don't seem to follow any type of distribution at all. Can we make sense of this? Well, consider a simplified model of the verification process, where the image the user looks at is either completely left or right. Thus a trace in our gallery is a binary string in the form:

$$\underbrace{00000}_{\text{with the sequence as long as the show period time}}111110000000000\ldots$$

And a genuine and false claim may look respectively like:

$$\text{Genuine} : 00001111110010000001$$
$$\text{False} : 01001100100110101100$$

Save for imprecisions, $\mathbb{E}(T) = 0$ where $T$ is the distance from the trace for genuine claims. While $\mathbb{E}(F) = 0.5len$ for false claims. We can also compute $\Pr(F = k)$ for $0 \leq k \leq len$ since this is just a binomial variable. So should we despair since our data looks nothing like a binomial distribution? Probably not, since our sample size is extremely limited so if we wanted more significant results we should probably increase the number of subjects.
However one may object that this is a gross oversimplification of the actual

problem. So what does the actual problem look like? We call $G$ and $R$ the two traces.

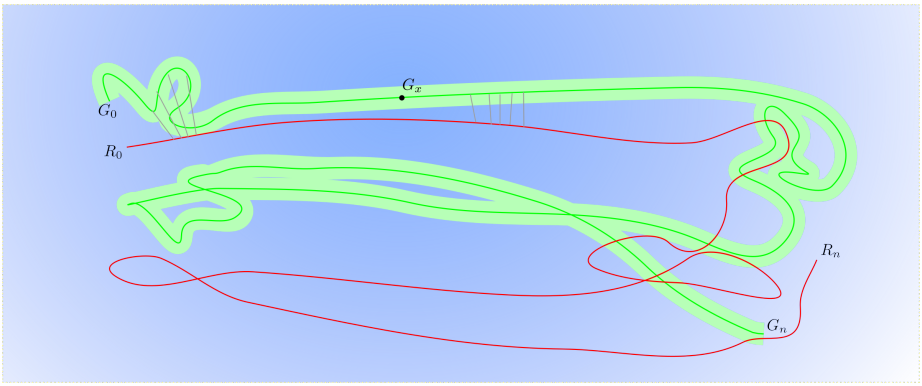


Figure 13: A Possible False Claim

The green one will be the genuine claim, showing other possible paths that fall in an offset of the enrolment trace. The red one will be a random trace. The point density is not shown but since these are not continuous lines ( we don't have infinite FPS! ) we assume that they can be broken into the same number of $n$ segments. What we have is thus matrices $G$ and $R$ of size $n \times 2$ that we feed to the DTW algorithm. Intuitively however we can skip this passage since the DTW is only going to lower bound our results, so let's say instead we just compute the distance between these matrices as

$$\sum_{i=0}^{n} dist(row_i(G), row_i(R))$$

So now we can start formalizing the problem. Given a bounding area $A$ and a curve $C$ we call $D$:

$$d_i = \frac{1}{A} \iint_A \operatorname{dist}(C_i, P(A)) \, \mathrm{d}A$$
$$D = \sum_C d_i$$

Which is the average of the distance of every point in the bounding Area summed over the points of the curve. Intuitively this tells us that a random walk in the bounding area should have a distance that falls within a range of this value. However the distributions are unique for each curve, in fact, look at the blue gradient in the figure. The points with the same colour have the same distance to the point $G_x$. Now imagine the distance distribution for a random sampling in the bounding area, for a fixed point. It would look quite weird since

the area is of course not symmetrical around each point of the curve. Is the sum of these pdf a nicer distribution? Of course we have the simplified case that we proved earlier so the answer is true for at least one particular scenario, but what about the general case? Unfortunately I can't tell, so that's going to stay uncertain, until some brighter bulb comes along.

Finally we report our ROC curve.

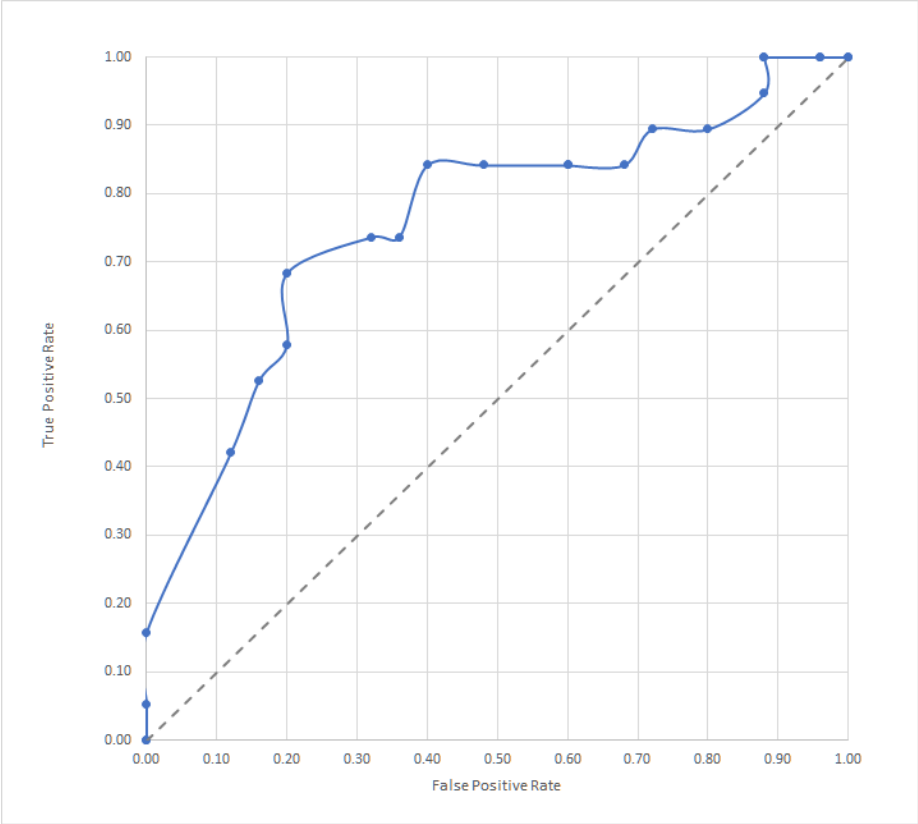


Figure 14: The ROC Curve

Which at least is looking good enough to tell us that this may be in fact a possibility to explore for a verification method.

# 8 The Game

As introduced earlier the use of gaze tracking for games has been implemented in a very simple shooting game of which we can see a screenshot below:


Figure 15: The Game with Debug Elements Enabled

The colored lines are the rays casted from the boat in order to determine the collisions that we use for navigation and are of course only enabled in debug mode. The player can make use of 4 different projectiles to try to take down the boats, each with different characteristics. These can be selected by holding Q and looking at the type of projectile wanted. The selector has only four sectors

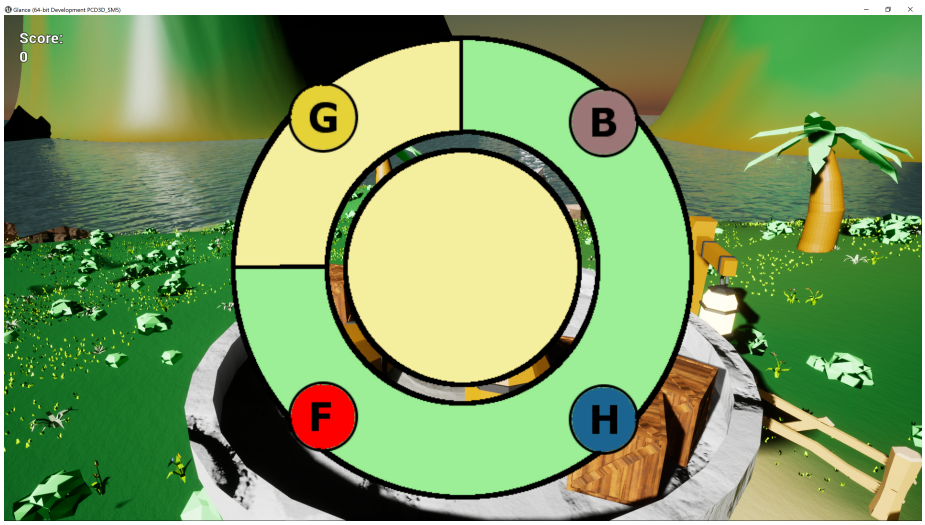


Figure 16: The Wheel Selector

since this proved to be the best compromise between speed and accuracy (we only need to get the quadrilateral right). An earlier version was divided into

more sectors but proved to be too unreliable and thus was discarded.
A simple score counter keeps tracks of the boats destroyed.
Finally we present a screenshot of the first implementation of the game which is now only kept for "historic" purposes.

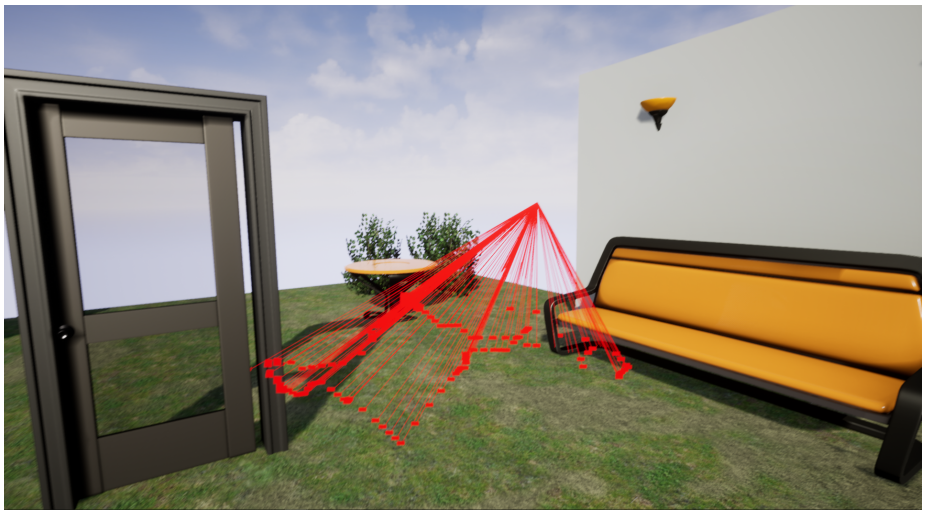

Figure 17: The Debug Level. This project, much like the door in the figure, lead to absolutely nowhere.

We can see from a third person the player' gaze darting around. The model for the character is empty since the system never got to the point of being usable.

# 9 Conclusions

Gaze tracking is an interesting area of research since it can now be performed online at an acceptable FPS even on consumer products in a non controlled setting. Of course it would be nice to explore this with more sensible hardware such as VR visors with integrated cameras since this would open for a wider range of applications.
The usage of gaze tracking for biometrics purposes also needs further research since the results presented here cannot be said to be conclusive.
On a technical note especially regarding the multimodal interaction part, while the 30 FPS mark is "ok" for such a demo, the gaze tracking shouldn't be what the CPU or GPU are busy doing, after all we don't expect an input device to be hogging any resources at all. Furthermore this performance bottleneck limits what can be explored regarding the biometric systems' part. Ideally we would want to show the images for a fraction of the time we do now, and still collect a comparable amount of data, i.e. the user shouldn't be able to "consciously"

think about where to look to obtain truly useful insights on the scan pattern and deter malicious users from trying to learn the sequence.

# References


[1] van Belle, Goedele et al. Fixation patterns during recognition of personally familiar and unfamiliar faces Frontiers in psychology vol. 1 20. 17 Jun. 2010, doi:10.3389/fpsyg.2010.00020

[2] OpenFace 2.0: Facial Behavior Analysis Toolkit Tadas Baltruaitis, Amir Zadeh, Yao Chong Lim, and Louis-Philippe Morency, IEEE International Conference on Automatic Face and Gesture Recognition, 2018

[3] Convolutional experts constrained local model for facial landmark detection A. Zadeh, T. Baltruaitis, and Louis-Philippe Morency. Computer Vision and Pattern Recognition Workshops, 2017

[4] Constrained Local Neural Fields for robust facial landmark detection in the wild Tadas Baltruaitis, Peter Robinson, and Louis-Philippe Morency. in IEEE Int. Conference on Computer Vision Workshops, 300 Faces in-the-Wild Challenge, 2013

[5] Rendering of Eyes for Eye-Shape Registration and Gaze Estimation Erroll Wood, Tadas Baltruaitis, Xucong Zhang, Yusuke Sugano, Peter Robinson, and Andreas Bulling in IEEE International. Conference on Computer Vision (ICCV), 2015