Communication Systems
Semester Project
Bachelor Semester 5 - Autumn 2019

Human Visual Attention for 3D models in Virtual Reality

Student: Manon Michel

Supervised by: Evangelos Alexiou
Prof. Dr. Touradj Ebrahimi

January 10, 2020

Multimedia Signal Processing Group
EPFL

Abstract

Virtual reality applications aim to faithfully reproduce real-world sceneries allowing high levels of immersion and engagement of the user with the virtual content. For specialized virtual reality applications, for example in usability or psychology research, among others, gaze information can be very useful. Specifically, by analyzing recorded gaze data, researchers can draw conclusions concerning the way users consume and behave with the contents under inspection.

The objective of this project is to get familiarized and use an eye-tracking device in order to record and collect gaze information from humans that consume 3D models in a virtual environment of 6-degrees of freedom. We acquire knowledge on human visual attention, and learn how to use the provided hardware for eye-tracking. Since the experimentation will be held in a virtual scene, we have the opportunity to acquire hands-on experience on a modern 3D development engine (e.g., Unity). Methodologies for processing and analysis of the acquired eye- and head-related streams will be developed in order to maintain useful gaze information from the recorded material. This data will be used to identify regions of interest from subjects that interact with the 3D models in the virtual world.

Contents

1 Introduction

Attention was defined by William James as implying *"withdrawal from some things in order to deal effectively with others. Focalisation, concentration, of consciousness are of its essence"* [1]. The ability to gather information on attention is extremely valuable for a number of applications. The insight that can be obtained regarding what draws attention and the perception of observers can be examined from a psychological viewpoint by analysing attentional behaviour but also from a physiological perspective on the neural mechanisms responsible for driving attentional behaviour. One way to understand attention, particularly from a psychological viewpoint, is through visual attention as the human visual is particularly subject to attention due to the fact that our inspection of visual scenes surrounding us relies on compiling small regions to build a coherent representation of scenes as a whole [1]. A particular medium to gather knowledge on visual attention is the tracking of eye movements. Eye tracking can be used in research on the visual system, in psychology, marketing, in human-computer interaction and in product design.

Assessing human visual attention is becoming increasingly popular in virtual reality (VR), both for the value of the gaze as an input and for the range of conditions that can be reproduced in VR through which visual attention can be analysed. Indeed, all the imaginable applications through which eye tracking can be used to examine human visual attention can be reproduced in VR without having to set up a physical experimentation environment and by achieving ideal experimental conditions, such as eliminating distractions or ensuring good lighting. On top of this, virtual reality in itself is expanding immensely both in its usage and in its development, making the ability to explore visual attention in such a unique environment an increasingly relevant topic of interest.

In this project, we gain insight on human visual attention in virtual reality by designing and developing a software in Unity to conduct subjective evaluation experiments with six degrees of freedom (6 DoF) in virtual reality. For this we use an eye-tracking hardware device, the Pupil Labs VR Binocular Add-on, together with the HTC Vive Pro head-mounted display (HMD). These experiments will evaluate the accuracy of recorded gaze as well as the extent of knowledge we can gather on gaze from various regions. For this we design an appropriate virtual reality experimental environment and obtain extensive data on the gaze and the corresponding displayed stimuli throughout the experiment. We also develop a procedure to post-process this data in order to extract viable information on human visual attention in our experiment. The rest of the report follows the development process of the project analysed in corresponding sections:

1. State-of-the-art of human visual attention, coding, display technologies and eye-tracking devices.
2. The functioning of the chosen eye-tracking device.
3. The software developed for evaluating visual attention.
4. Explaining methodologies to detect fixations and notably those used for this project.
5. Describing the experimental process and the post-processing of the recorded data.

2 State-of-the-art

2.1 Human Visual Attention

Visual attention can be considered to function in terms of the *what* and the *where*. The *what* consisting in the detailed inspection of a spatially limited region while the *where* consists of the visual selection for detailed inspection of specific regions of interest from the entire visual field [1].

Visual attention is often thought of as following a bottom-up model by which one begins with a low resolution vision of the entire scene and once the eye movement is complete, the fovea is directed at the region of interest and attention engages in order to perceive the latter in high resolution. This model assumes that the visual stimulus (e.g. image features) are solely responsible for attracting attention and fails to address the link between attention and eye movements thus making it incomplete [1].

In order to clarify such aspects, it is important to note that the human visual system responds with different intensity to different types of stimuli, for example edges drive a stronger response than homogeneous regions while a parallel pre-attentive stage acknowledges the presence of four basic features: color, size, orientation, and presence and/or direction of motion. This is driven by the regions in the brain responsible for responding to and interpreting such visual stimuli, notably three main neural regions implicated in eye movement [1]:

- Posterior parietal complex: disengages attention
- Superior Collicilus: relocates attention
- Pulvinar: engages, or enhances, attention

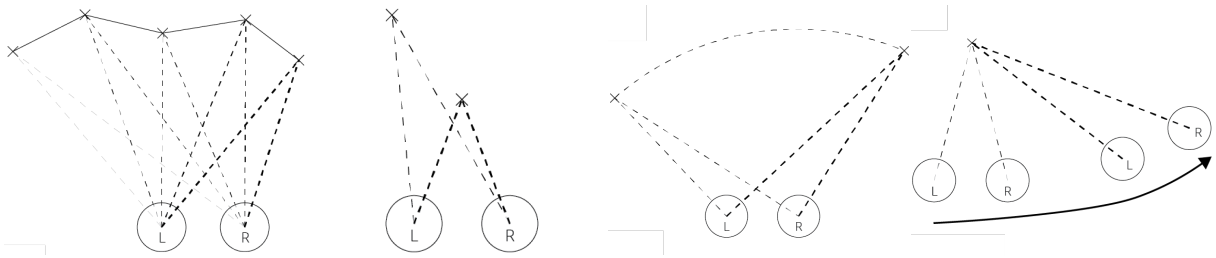
Another significant aspect of human visual attention, notably to this topic, is that of eye movements. The main models of eye movements can be broken down as such:

1. Saccadic - *saccades are rapid eye movements used in repositioning the fovea to a new location in the visual environment* (cf. Figure 1a)
2. Smooth pursuit - *occur when visually tracking a moving target* (cf. Figure 1c)
3. Fixations - *eye movements that stabilise the retina over a stationary object of interest.*
4. Vergence - *used for depth perception* (cf. Figure 1b)
5. Vestibulo-ocular - *movement compensating for the movement of the head* (cf. Figure 1d)
6. Physiological nystagmus - *miniature movements associated with fixations*

From these, the first three suffice in providing the most evidence for voluntary eye movement. Notably, saccadic movements are considered manifestations of the desire to voluntarily change the focus of attention while smooth pursuits and fixations correspond to the desire to maintain one's gaze on an object of interest. It is also important to note that approximately 90% of human viewing time is spent on fixations [1].

2.2 3D Development Engines

In order to asses human visual attention in virtual reality, an engine that supports such features is necessary. Many such engines are available for instance Unity, Unreal Engine 4 or even AppGameKit VR. For this project, the cross-platform real-time engine Unity 3D was chosen both for its ease-of-use and extensive features, especially in virtual reality, but also because it is extensible to our eye-tracking intent.



(a) Saccades (b) Vergence (c) Smooth pursuit (d) Vestibo-ocular

Figure 1: Eye Movements [2]

2.3 Eye-Tracking Devices for Virtual Reality

For the purpose of eye-tracking in this project we use the Pupil Labs VR/AR Binocular add-on for the HTC Vive Head-Mount Display (HMD) [3]. The Table 1 compares different devices available for eye tracking in virtual reality. The Pupil Labs device was chosen as it is way less costly, given its technical specifications, it provides open source scripts and software [6], and it can be used both for desktop and headsets.

Name	Type	Price	Accuracy	Tracking frequency	Trackable FoV
Pupil Labs	HMD Add-on	1400 €	≈1.0°	200Hz	110°
VIVE Pro Eye	Headset	1700 €	0.5°–1.1°	120Hz	110°
Fove	Headset	540 €	<1°	120Hz	Up to 100°

Table 1: Comparison table for Eye Tracking devices in VR [3][4][5]

3 Pupil Labs

3.1 Pupil Detector

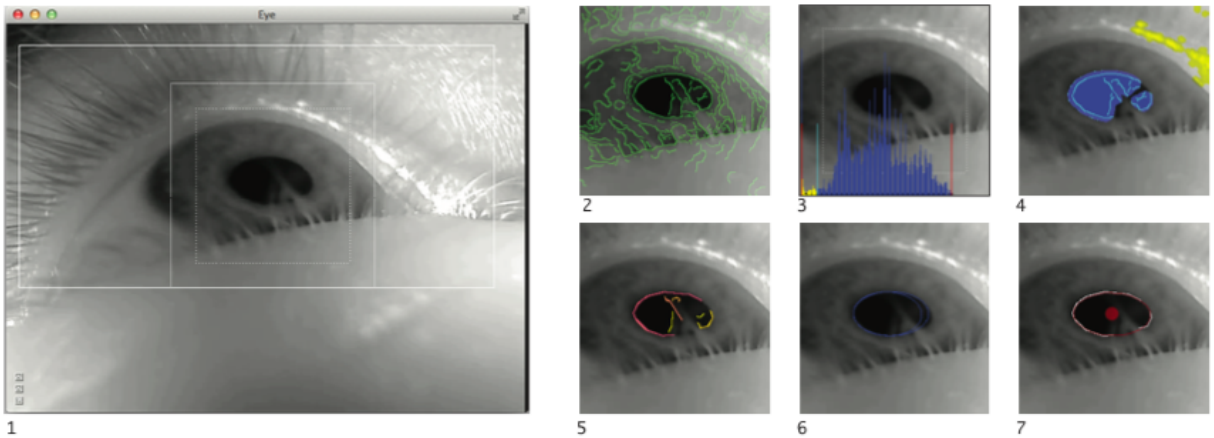


Figure 2: Visualisation of the Pupil Detection Algorithm

The pupil detection algorithm used by Pupil Labs uses the "dark pupil" detection method, i.e. it essentially locates dark regions to detect the pupil. The breakdown of the algorithm follows the visualisation of Figure 2.

First, the eye image is converted to gray-scale and the pupil region is estimated (dashed line squares). The outer white rectangle represents the user region of interest. The second step

consists of Canny edge detection [9] which locates contours in the eye image by filtering edges based on neighbouring pixel intensity. Thirdly, the algorithm defines darker areas (blue regions) as a user set offset of the lowest spike in the histogram of pixel intensities within the eye image. In the fourth step the edges are filtered to exclude spectral reflections (yellow regions). In the fifth step we see the remaining edges are extracted into contours, this is achieved using connected components. These contours are then split into sub-contours based on curvature continuity criteria, this is visualised by the multicolored lines. Then candidate pupil ellipses are formed using ellipse fitting on a subset of contours using the least square method to find good fits and a user defined range. These candidate pupil ellipses are represented by the blue ellipses in step 6. Finally, a final ellipse fit is found using an augmented combinatorial search, the center and ellipse are visualised in red and the supporting edge pixels are drawn in white [7].

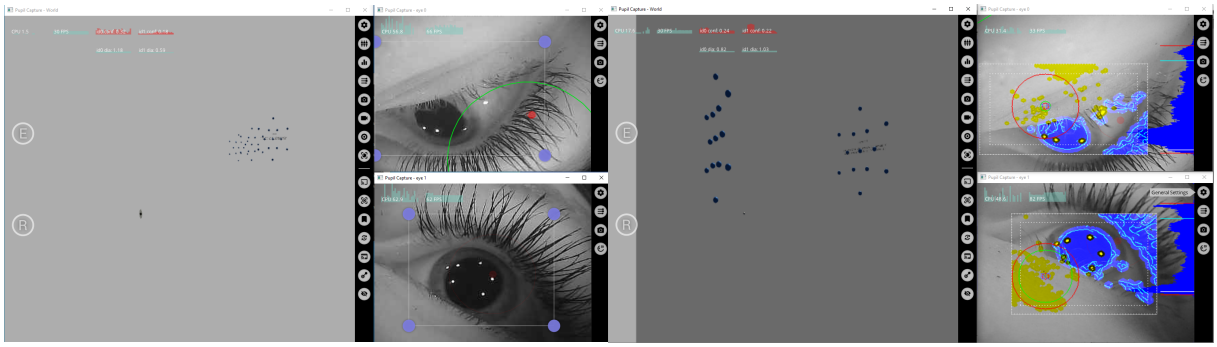
The algorithm also provides a "confidence" value for the detected pupil. This confidence is the ratio of supporting edge length and the ellipse circumference [7].

3.2 Pupil Software

3.2.1 Pupil Capture

Pupil Capture is the software that reads the video streams coming from the Pupil Labs cameras. The software uses these streams for pupil detection, gaze tracking, video recording and real-time data streaming [8]. Pupil Capture also detects and tracks markers in the users environment, this environment is typically the real world when a world camera is available, however, this feature is not part of the add-on as our environment of interest is the virtual-world. For example, we can visualise the virtual scene from unity in Pupil Capture.

Pupil Capture is composed of 3 windows. The main window is the "World Window" and displays a view of the environment. On top of this, performance graphs are displayed for CPU, frames per second (FPS), pupil algorithm detection confidence and pupil diameters. Pupil Capture also centralises the plugins in use, both with system plugins (e.g. Recorder, HMD Calibration) and with user added plugins such as the Blink Detector, Fixation Detector or Time Synchroniser. The two remaining windows display both eye images in real-time and allow users to tweak parameters such as pupil region of interest (Figure 3a), pupil size range, model sensitivity, resolution, absolute exposure time and more. The eye windows also allow visualisation of the eyes with the algorithm view (Figure 3b) from which we can visualise the dark areas as blue regions and the spectral reflections as the yellow regions as well as other pupil detection features.



(a) ROI View (b) Algorithm View

Figure 3: Pupil Capture

3.2.2 Pupil Service

Pupil Service is essentially like Pupil Capture but with less features, notably it does not have a world video feed or GUI. It is designed to run in the background and to be controlled via network

commands only. Although Pupil Service is intended to be used with VR eye tracking setups we chose to use Pupil Capture instead, particularly for plugins, data recording and for visualising the VR view in realtime [8].

3.2.3 Pupil Player

Pupil Player is a media and data visualiser from which one can observe Pupil Capture recordings. Once a recording is loaded into Pupil Player, it appears to be very similar to Pupil Capture, with the performance graphs and plugins. In addition to this though, it provides a timeline panel for trimming and frame stepping and from which we can visualise certain events such as blinks and pupil confidence over time, as shown in Figure 4. Pupil Player offers a multitude of useful plugins. Namely the Eye Video Overlay can be used to overlay the eye video on top of the world video. Another useful plugin for eye tracking in virtual reality is the eye movement detector which classifies segments of variable lengths into different eye movements such as fixation, smooth pursuit, post-saccadic oscillations and saccades. Besides the various plugins available, a key feature of Pupil Player is the ability to export data. From this Pupil Player yields eye and scene recordings with added gaze markers and pupil detection markers but also numerous `.csv` files for timestamps, gaze positions, pupil positions and also `.csv` files for the plugins such as blink detection, fixation detection or annotations. The export directories also contain `.npy` timestamp files for the three video files [8].

In order for Pupil Player to function, it takes as input a recording directory which has usually been output by Pupil Capture/Service through the recording plug-in. In order to be recognised by Pupil Player, these directories contain files of the following formats:

Meta Files: These files are essential for Pupil Player to recognize the directory as a Pupil Recording. This file is named `info.player.json` [8].

Timestamp Files: These files are of the form `<name>_timestamps.npy` which correspond to data files such as `eye0.mp4` or `blinks.pldata`. The data and timestamp are synchronised through indices, i.e. the `i`th timestamp in `world_timestamps.npy` belongs to the `i`th video frame in `world.mp4`. The timestamps are saved in the NPY binary format and can be accessed by using `numpy.load()` in Python [8].

Audio Files: Such files are only recognisable by Pupil Player's playback plugin if the file is named `audio.mp4` and has a corresponding `audio_timestamps.npy` file containing the correct audio timestamps [8].

Video Files: These correspond to video files for the scene, right eye and left eye and can be of variable formats (`.mp4`, `.mkv`, `.avi`, `.h264`, `.mjpeg`) [8].

`pldata` **Files:** These files are a specific Pupil Labs format which contain a sequence of independently msgpack-encoded messages each consisting of two frames. The first frame corresponds to the payload's topic as a string e.g. `"pupil.0"` and the second frame corresponds to the payload, e.g. a pupil data, encoded as a msgpack (so this frame is encoded twice). These messages are decoded by Pupil Player into `file_methods.Serialized_Dict`. The `.pldata` files can be read and written using `file_methods.load_pldata_file()` and `file_methods.PLData_Writer` [8] .

3.3 Pupil Demos

Pupil Labs offers a number of demo scenes which demonstrate how to use their features in Unity. The demos show how to subscribe to `pupil` topics and also how to interpret the dictionary received in Unity from the Pupil Labs software, for example blinks (see Figure 5a) or gaze

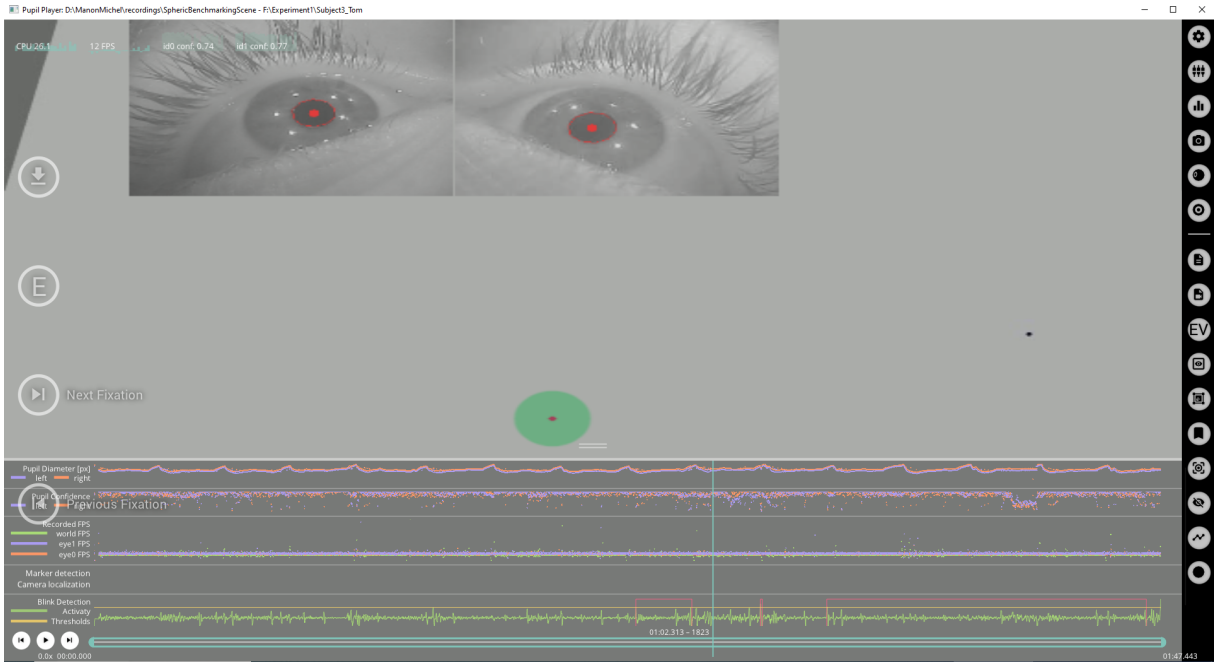
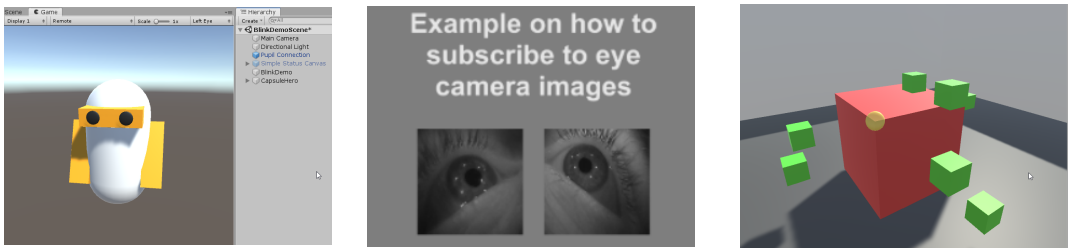


Figure 4: Pupil Player

visualisation (see Figure 5c). They also present how to communicate with Pupil Labs for example to get eye video frames, as shown in Figure 5b, which is especially useful for calibration. Demos are also available to show how to record data directly from Unity or how to cast the virtual scene from Unity to Pupil Capture's "World view" [6].



(a) Blink Demo (b) Frame Publishing Demo (c) Gaze Demo

Figure 5: Pupil Labs Units Demos

3.4 Communication between the Pupil Add-on and Unity

The Pupil Labs eye-tracking device and Unity communicate through plugins and dictionaries. There are various dictionaries for corresponding `topics`, for example the `blink` topic, when subscribed to, will yield access to a dictionary containing the blink `type` (onset or offset), `confidence`, `base_data` (gaze points) and the `timestamp`. To access Gaze Data, the `GazeListener` subscribes to the `gaze` topic provided by Pupil Capture/Service and provides C# events containing already parsed `GazeData`. The `GazeController` wraps the listener as a `MonoBehaviour` to simplify the setup in Unity. From this, one can easily access Pupil gaze data such as gaze direction directly in Unity.

Direct communication from Unity to Pupil Labs is facilitated by the presence of annotations. Annotations are messages that appear in real-time in Pupil Capture/Service. Each annotation also has a corresponding timestamp and index and can then later be exported through Pupil Player.

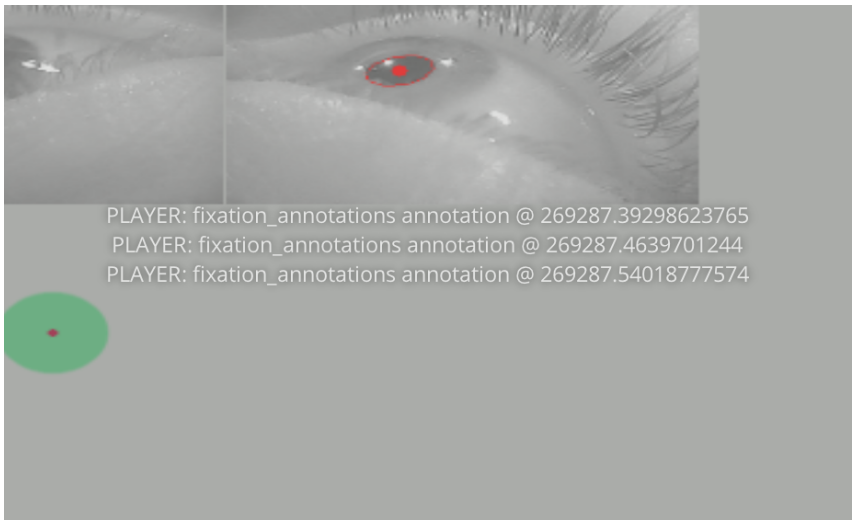
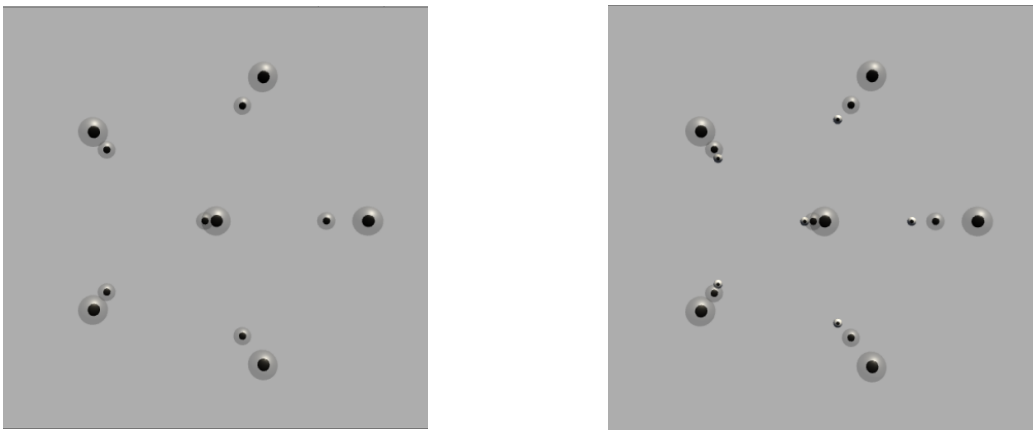


Figure 6: Annotations in Pupil Player

4 Bench-marking

In order to assess human visual attention in virtual reality through eye tracking, the accuracy of our ability to track the gaze in virtual reality needs to be assessed. This accuracy goes beyond the scope given by the technical specifications of the devices used and for this purpose it was necessary to create a Bench-mark Scene.

Each scene begins with a Pupil Labs calibration subscene. This calibration allows Pupil Labs to establish a mapping between pupil positions and the gaze point. It can be added to any scene through the `CalibrationController` component. The calibration settings are modifiable directly from Unity and allow flexibility of calibration. Mainly, users can choose between Fast Settings (1 s between each marker) and Default Settings (2 s between each marker) and also between Few Targets (12) and Default Targets (18) as in Figure 7. Once calibration is complete, the data is sent to Pupil software which then responds with either a `success` or `failure` message [6].



(a) Few Targets (b) Default Targets

Figure 7: Pupil Labs Calibration Targets

4.1 Cubic scene

The initial idea was to create a simple benchmark scene with markers in order to estimate the accuracy of the eye tracking. The initial benchmark scene was composed of 12 cubes positioned in various areas of a simple grey room, appearing one after the other for two seconds. During this time, the angular error between the gaze direction and the direction to the cubes position is output to evaluate the accuracy of the gaze. Figure 8 shows a visualisation of the gaze rays, with the predicted gaze ray in green which is aimed at the center of the cube and the real gaze ray in magenta. The angular error we get from this bench-marking scene is the one between these two rays. This scene is attached to the `CubicBenchmarkBehaviour.cs` script, which is structured as described in the following section.

4.1.1 CubicBenchmarkBehaviour.cs

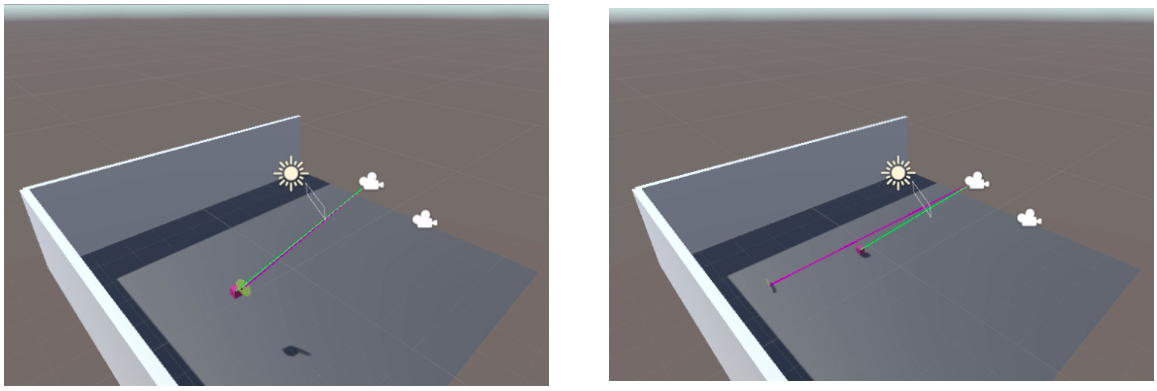
Start() The `Start()` method is called before the first frame update and creates a list of `GameObject`s with the 12 cubes and hides these cubes.

Update() The `Update()` method is called once per frame and listens to the `S` key that displays the gaze visualisation and activates the coroutine that displays the cubes through `StartCoroutine(displayCubes()`. The method also uses the `GazeController` to receive gaze data and calls `ShowProjected()`.

displayCubes() This method displays each cube one by one for 2 seconds and prints the angular error between the predicted gaze and the actual gaze to the console. We wait 0.5 seconds before printing this angular error as a way to ensure the subject has had time to move their gaze from the previous cube to the current one. Note that this was a temporary solution and a more elaborate approach was used for the Spherical Scene, namely fixation detection. At this point we also display the predicted gaze ray in green in the Unity Scene view as in Figure 8.

ReceiveGaze() This method takes as input `gazeData` which is received directly from the `GazeController`. We verify that the `gazeData` has a confidence above the given threshold and that the gaze is binocular. If all conditions are met, we get the `GazeDistance` and `GazeDirection` from the `gazeData`.

ShowProjected() This method uses `Physics.SphereCast()` to detect a potential collision between the gaze direction and an element in the scene (ideally, the cube). We then display the subjects gaze ray in magenta in the Unity Scene view as in Figure 8



(a) On target

(b) Close to target

Figure 8: Gaze Rays in Cubic Scene

4.2 Spherical scene

While the cubic scene above is functional and capable of giving some indication on the limitations of eye tracking in virtual reality, in order to be more precise it was decided to create a second scene to assess the angular error in a more structured way. For this the cubes were replaced by 39 spherical markers placed at 3 different depths (1 m, 3 m and 5 m) and at 13 different angles from the center, as in Figure 9. The outer and innermost spherical markers at depth 1 m are respectively at an angle of 22°and 12°from the center, at depth 3 m they are at respectively 9°and 5°from the center and at depth 5 m they are at an angle of 5°and 3°from the center.

In order to minimise the amount of distractions while visualising the spherical markers, the scene was transformed from a room to a uniform and neutral background, as can be seen in Figure 9.

A major addition to this bench-marking scene is that the angular error is output only when a fixation is detected, this guarantees that we are assessing the angular error between the predicted gaze and the real gaze only when this real gaze is intentional. This is achieved by subscribing to the Pupil Labs `fixation` topic from Unity in order to get information on detected fixations from Pupil Labs' Online Fixation Detection.

This scene is attached to the `SphericalBenchmarkBehaviour.cs` script, which is structured as described in the following section.

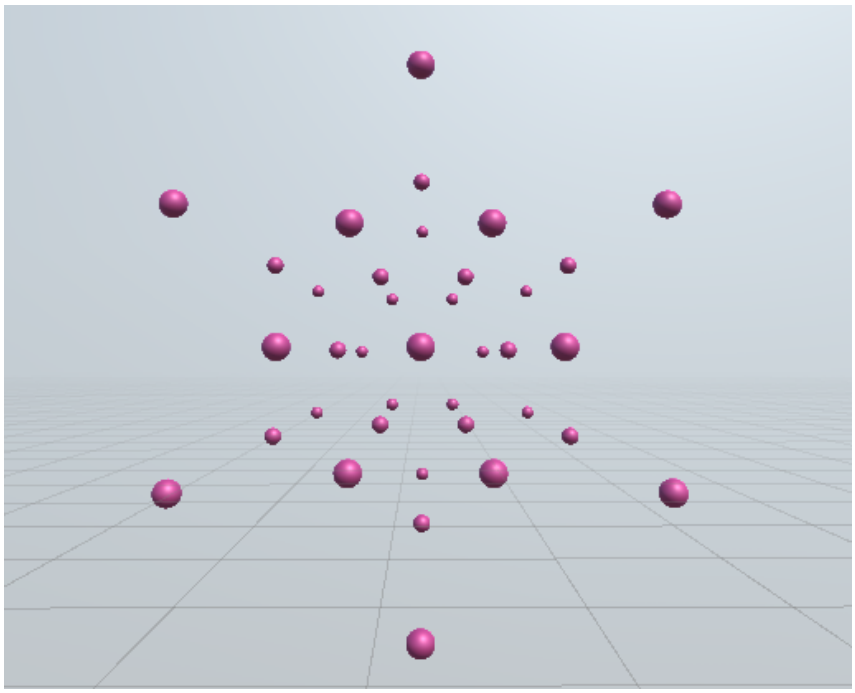


Figure 9: Spherical Markers

4.2.1 SphericalBenchmarkBehaviour.cs

Start() The `Start()` method is called before the first frame update and creates a list of `GameObject`s with the 39 spheres and hides them. It also activates a wall that ensures that the spheres are viewed without visual distractions from lighting or background scenery.

Update() The `Update()` method here is effectively the same as for CubicBenchmarkBehaviour.cs.

displaySpheres() This method displays randomizes the order of appearance of the spheres and displays each spherical marker one by one for 2 seconds. Data regarding the spherical markers

is collected at this point for later use, in particular the label corresponding to the current sphere being displayed.

ReceiveGaze() This method takes as input `gazeData` which is received directly from the `GazeController`. We verify that the `gazeData` has a confidence above the given threshold and that the gaze is binocular. If all conditions are met, we get the `GazeDistance` and `GazeDirection` from the `gazeData` and compute the angular error between the predicted gaze and the received gaze.

ShowProjected() The `ShowProjected()` method here is effectively the same as for CubicBenchmarkBehaviour.cs.

recordData() This method records data to a `.txt` file created in Start(). This data is intended to be recorded at all times during the experiment, i.e. not only during fixations, it essentially records the same data as the annotations. This method isn't used in the project as it creates too much lagging.

SendFloatAnnotation() This method takes as input a string, `aLabel` and a float `value` that sends the given value with the given label as an annotation to Pupil Capture.

SendStringAnnotation() This method takes as input a string, `aLabel` and a string `value` that sends the given string with the given label as an annotation to Pupil Capture.

SendVectorAnnotation() This method takes as input a string, `aLabel` and a Vector3 `value` that sends the given vector with the given label as an annotation to Pupil Capture.

SendBenchmarkAnnotation() This method takes as input a float, `angularError`, a Vector3 `gazeDir`, a Vector3 `sphereDir`, a Vector3 `gazePoint`, a Vector2 `normPos`, a Vector3 `spherePos`, an object `duration`, an object `confidence`, an object `dispersion` and an object `id`. It sends the given data as an annotation to Pupil Capture. This data is further detailed in section Data.

StartFixationSubscription() This method subscribes to the `fixation` topic and indicates the desired `max_dispersion` and `min_duration`.

CustomReceiveData() This method is called when a fixation is detected and in turn calls `SendBenchmarkAnnotation()`.

5 Fixation Detection

The ability to detect fixations is crucial to analysing human visual attention, amongst many applications. Methods to detect fixations vary and can be classified as dispersion- or velocity-based as well as data-driven. Dispersion-based methods depend on the spatial scattering of the gaze within a chosen time interval while velocity-based methods depend on point-to-point velocities of the gaze. Both of these methods rely only on gaze data and therefore do not take into account information such as the target being observed [10]. Data-driven methods are based on the clustering of eye positions, using projections and projection aggregation applied to static pictures [11].

Pupil Labs' fixation detectors implement a dispersion-based method which identifies fixations as groups of consecutive points within a particular dispersion (or maximum separation) and that attain the time threshold set defining the minimum duration of a fixation. However the precise

approach of the fixation detection depends on whether fixations are detected in an online or offline context (i.e. real-time or post-recording) [8].

5.1 Online Fixation Detection

The Online Fixation Detection is the method used in Pupil Capture when detecting fixations in real-time. These fixations are based on three user-set parameters. First the `Maximum Dispersion`, a spatial parameter in degrees which represents the maximum distance between all gaze locations during a fixation and is set to 3° on setup. Secondly, the `Minimum Duration`, a temporal parameter in milliseconds indicating the minimum duration in which this dispersion threshold must hold and typically set to 300 ms on setup. Thirdly, the `Confidence Threshold`, a qualitative parameter between 0 and 1, typically set to 0.75. Any data below this threshold will be disregarded during fixation detection [8].

This approach to fixation detection is advantageous as it provides a real-time solution and its results can be directly used in Unity through subscription to the `fixation` topic. Since fixations are published as soon as they comply with the constraints this can result in a series of overlapping fixations.

5.2 Offline Fixation Detection

Pupil Player deals with complete recordings and thus has an Offline Fixation Detection plug-in. This approach also detects fixations using a dispersion threshold in degrees but unlike the online method, this threshold is not evaluated above a minimum duration but within a given duration window. Therefore an additional temporal parameter is also required, the `Maximum Duration`, in milliseconds. From this, we obtain a duration window within which the length of classified fixations are maximised. As an example, instead of creating two consecutive fixations of length 300 ms it creates a single fixation with length 600 ms [8].

This procedure is beneficial in that fixations do not overlap and thus the data is more concise but it can only be performed on a complete data-set.

6 Conducting the Experiment and Data Post-processing

6.1 Experimental Setup

In order to further assess human visual attention in virtual reality using eye-tracking we used the spherical bench-marking scene to conduct experiments. The aim for this experiment is for subjects to observe each spherical marker of the scene and this data to be recorded from Unity and processed. For the experiment, the spherical bench-marking scene was setup to ensure natural gaze comfort and representative data. The largest angle between a sphere and the central ray is 22°(for the six closest outer markers). Such an angle allows us to reach a high variety of data whilst ensuring comfort of gaze as it does not exceed 25°. In terms of depth, the spheres are placed at 1 m, 3 m and 5 m. These depths were also selected to ensure the visualisation remains natural while achieving good coverage. In addition to this, the 39 spherical markers are set to appear one at a time for 2 seconds and in a random order such that order does not impact the gaze accuracy. For the experiment, we set the fixation detection parameters with a maximum angular dispersion of 1°and a minimum duration of 300 s.

6.2 Methodology

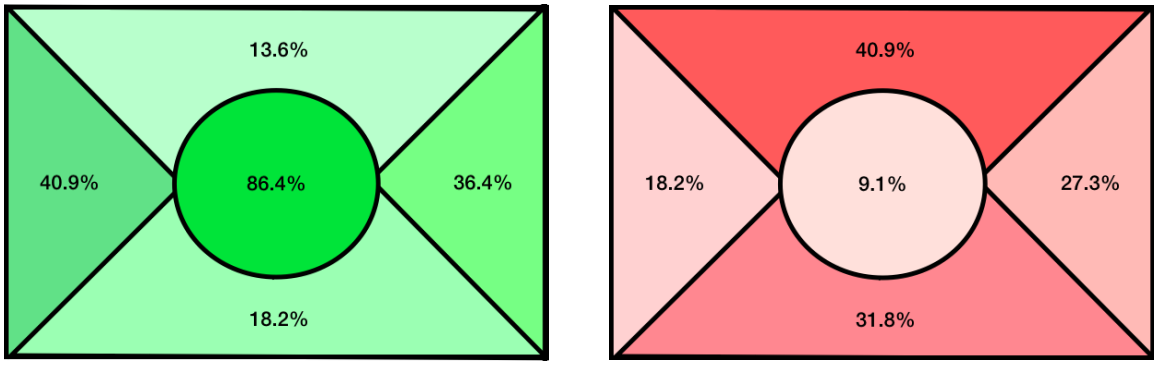
The test begins by seating each subject and adjusting the headset to their head to ensure comfort but also to make sure the Pupil cameras are able to correctly detect the eyes. We also verify that no sound comes from the headset during the experiment. Then the region of interest is

adjusted for each subject in Pupil Capture by ensuring it captures the pupil even when looking completely up, down, left and right and ensuring that pupil confidence is high (i.e. above 0.8).

Once the subject is ready and everything is adjusted, the experiment can begin. The experiment conductor explains to the subject that they must try to fixate each calibration target and then presses the `C` key on the keyboard to activate the Pupil Labs calibration. This calibration is set to the high-level settings as in Figure 7b in order to maximise the accuracy of the eye-tracking. Once calibration has succeeded, the conductor instructs the subject they will have to fixate on a series of spherical markers and then proceeds to recording the data by pressing the `R` key and starts displaying the spherical markers by pressing the `S` key. Once all the markers have been displayed, the recording is stopped by again pressing `R` and the experiment is over. Each subject then fills in a questionnaire to collect information on the subject and feedback on the experience.

6.3 Subjects

The experiments were conducted over 2 days with 22 different subjects aged 18 to 23 from which 16 were male, 6 were female, 68% have brown eyes, 7 wear glasses, 3 wear contact lenses and 12 wear no eye corrections. On a scale from 0 to 5, where 0 is "uncomfortable" and 5 is "comfortable", subjects rated the experience at an average of 4.3. Subjects were also given the opportunity to provide feedback on which regions were easier or harder to observe as can be seen in Figure 10. Note that for the harder to observe regions, one subject responded with "None" and another with "Edges". Finally, when asked how observable the spheres were as a whole, 91% of subjects responded with "Completely fine" while one responded with "Too far (in depth)".



(a) Easy-to-observe regions

(b) Hard-to-observe regions

Figure 10: Ease of observation

6.4 Data

The data received from the experiments is sent directly from the `SphericalBenchmarkBehaviour.cs` script that controls the behaviour of the Spherical Bench-marking scene. When a fixation is detected in the script, the `CustomReceiveData()` method is called, this method mainly warns the console that a fixation has been detected and in turn calls the `SendBenchmarkAnnotations()` method. The latter sends a number of annotations to Pupil Capture in the form of a dictionary. This dictionary contains the following data:

Fixation data: Composed of the `fixation_id` which can be used to merge overlapping fixations, the `fixation_duration` which represents the duration of the fixation in milliseconds, the `dispersion` which represents the angular dispersion in degrees, the `avg_pupil_confidence` representing the angular pupil confidence over the duration of the fixation and `norm_pos_x`, `norm_pos_y` the normalized position of the fixation's centroid.

Gaze data: Which englobes the gaze direction (`gaze_direction_x`, `gaze_direction_y` and `gaze_direction_z`) and the gaze position (`gaze_position_3d_x`, `gaze_position_3d_y` and `gaze_position_3d_z`)

Spherical marker data: Containing the sphere direction (`sphere_direction_x`, `sphere_direction_y` and `sphere_direction_z`), the sphere position (`sphere_position_3d_x`, `sphere_position_3d_y` and `sphere_position_3d_z`), the `sphere_label` to indicate which sphere is being viewed and the `sphere_angle` which represents the angle from the central gaze ray to the current sphere's position.

Head data: Which consists of the head position (`head_position_x`, `head_position_y` and `head_position_z`) and the head rotation (`head_rotation_x`, `head_rotation_y` and `head_rotation_z`)

Angular error: The angle between the predicted gaze ray (the spherical markers position) and the real gaze ray (the gaze position) is output as `angularError` in degrees. Since the experiment is recorded through Pupil Capture, this data can be exported through Pupil Player by using the Annotations Plug-in. This yields an `annotations.csv` file for each recording, which can be found in the `exports` folder of the recording. This file is thus composed of the data mentioned previously as well as a `timestamp` and `label` for each annotation. Note that this `label` is to be defined when sending the annotations, here it is defined in `SendBenchmarkAnnotations()` as `fixation_annotations`.

While the `annotations.csv` file contains the most valuable information for our analysis, a number of data is also made available from recording the experiments through Pupil Capture. This data corresponds to the files mentioned in the Pupil Player section.

6.5 Post-Processing

The post-processing of the data is dealt with through the `process_annotations.py` script. The main focus of this post-processing is on the angular error measured between the predicted gaze and the actual gaze of each subject. This is broken down into three parts which constitute the three main methods of the script.

`avg_per_sphere()` This method takes as input a string `path` corresponding to the path of the experiment directory, i.e. the directory containing the recordings for the experiment session, and two strings, `start` and `end` corresponding to an interval of subject recording directories to process. With these inputs, we iterate over each recording and open their respective `annotations.csv` file previously exported from Pupil Player. The `sphere_label` and `angularError` are extracted for each row of the file and if the `angularError` is inferior to $5°$ we add it to a list corresponding to that particular spherical marker. The data is filtered as such as we consider that angular errors greater than $5°$ suggest the subject is not observing the target. Once we've iterated over every row of the file, we have a complete dictionary `sphere_lists` with each individual spherical marker mapped to a list of valid angular errors corresponding to it. From this dictionary we compute the average angular error for each individual spherical marker and create a new file, `avg_angularError_per_sphere_<id>.csv`, with each spherical marker label and its corresponding average angular error, where id is the string identifying the subject, e.g. "12". Note that if the data for a particular spherical marker is unusable, the average will simply be noted as `N/A`.

`collective_stat()` This method takes the same inputs as `avg_per_sphere()`. It then proceeds to iterate over the `avg_angularError_per_sphere_<id>.csv` files for each subject which are in a

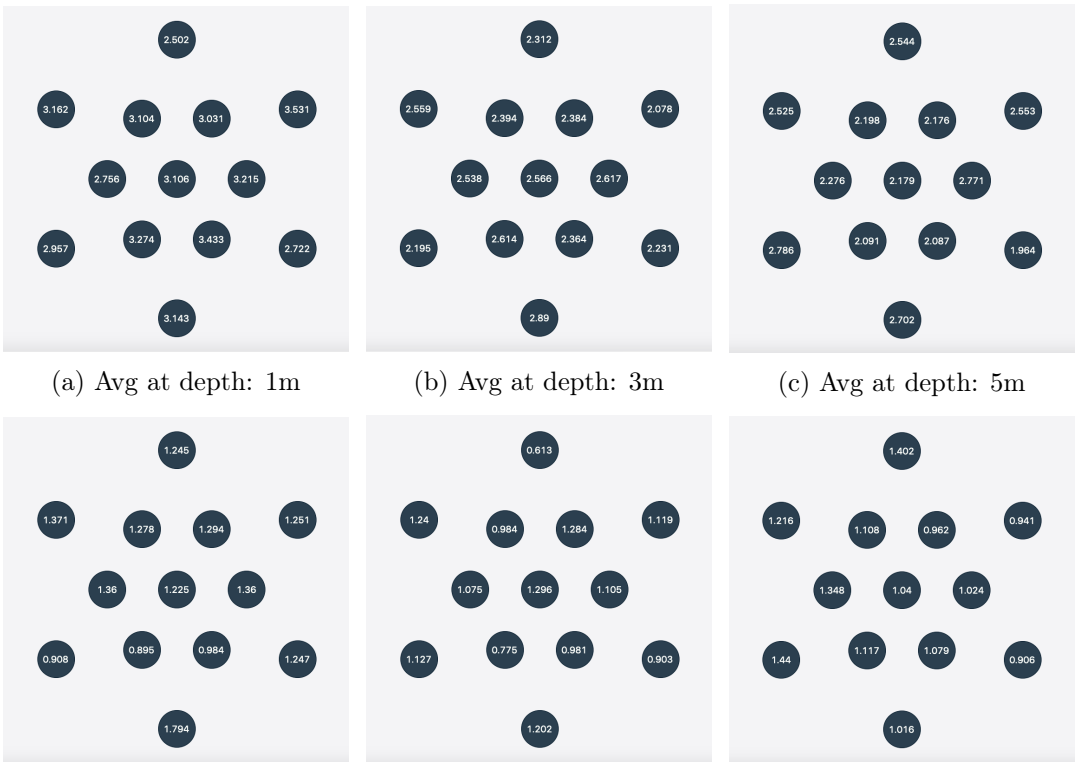
`post_processing` directory in the experiment directory. From these, it extracts the average angular error per spherical marker for each subject and computes the total average and the standard deviation over the subjects. From this we create two new files, `collective_avg_sphere.csv` and `collective_stdev_sphere.csv`, in the same directory, with the total average angular error and standard deviation for each spherical marker over all subjects.

visualise_data() This method takes as input a string `stat_file` which corresponds to the path to a file with a `sphere_label` field and a `avg_angularError` or `stdev_angularError` field, i.e. one of the files output from the two previous methods, a string `field` indicating which type of statistic we want to visualise, and an integer `depth` corresponding to the marker depth we wish to visualise. We then read the file and extract the given statistic for the spherical markers of the given depth and use the latter to create a visualisation of the data as shown in Figure 11 and 11. This visualisation is created using the Tkinter Canvas Python module.

6.6 Analysis

The data visualisation shown in Figure 11 represents the average angular error and standard deviation for each individual sphere over the 22 subjects that conducted the experiment. Note that for some subjects there were software issues which caused a black screen to appear at times or glitches in the spheres positions. However these bugs do not hinder the overall data as they resulted in angular errors that were filtered by our 5° threshold. These limitations seemed to have been caused by updates of the software, SteamVR, used for communication between Unity and the HTC Vive Pro HMD. In particular, it is important to update the drivers after having updated all the necessary software.

6.6.1 Average



(a) Avg at depth: 1m (b) Avg at depth: 3m (c) Avg at depth: 5m

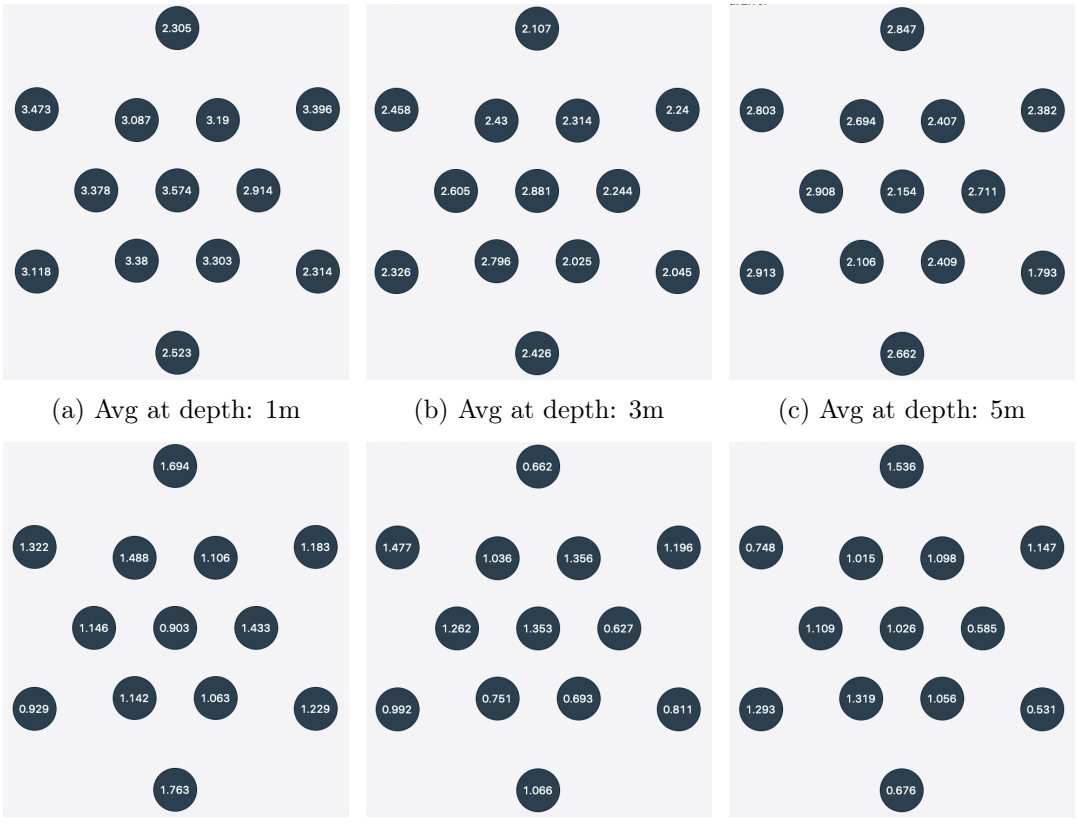
(d) Std at depth: 1m (e) Std at depth: 3m (f) Std at depth: 5m

Figure 11: Data Visualisation over all subjects

From our results we can see that the angular error between the predicted gaze and the recorded gaze was overall much higher for the spherical markers at a closer depth. A possible reason that would explain this is the fact that closer spheres have a higher viewing surface making the probability for a subject's gaze to hit exactly the middle of the sphere a lot weaker.

We can also analyse the results by comparing them to the ease of observation indicated by the subjects in Figure 10. One main observation from the subjects is that they found the centre easier to observe, however Figure 11 indicates that the angular error was not necessarily the lowest in the centre. While the inner circle does have a slightly lower angular error overall for further depths, the difference is far from being as overwhelming as one could expect. Likewise, most subjects designated the harder to observe regions as being "Up" despite the fact that the "highest" sphere of the two closest depths was one of, if not the, most accurately visualised sphere. These disparities could be explained by the fact that since subjects struggled more with certain regions, they would focus more intensely on properly fixating on the latter, while when they found targets easy to visualise, such as the centre, they did not put as much effort in the accuracy of their gaze.

As an overall conclusion, it seems that in order for the gaze predictions to be close to the recorded gaze, the targets for this predicted gaze should be as precise as possible, i.e. have a small viewing surface and the viewing conditions should limit distractions as much as possible in order to encourage a high-ability to focus on targets. It is also important to note that the Pupil Labs eye-tracking device has an accuracy of approximately 1°(see Table 1) and this should be taken account when evaluating the angular error data we receive.



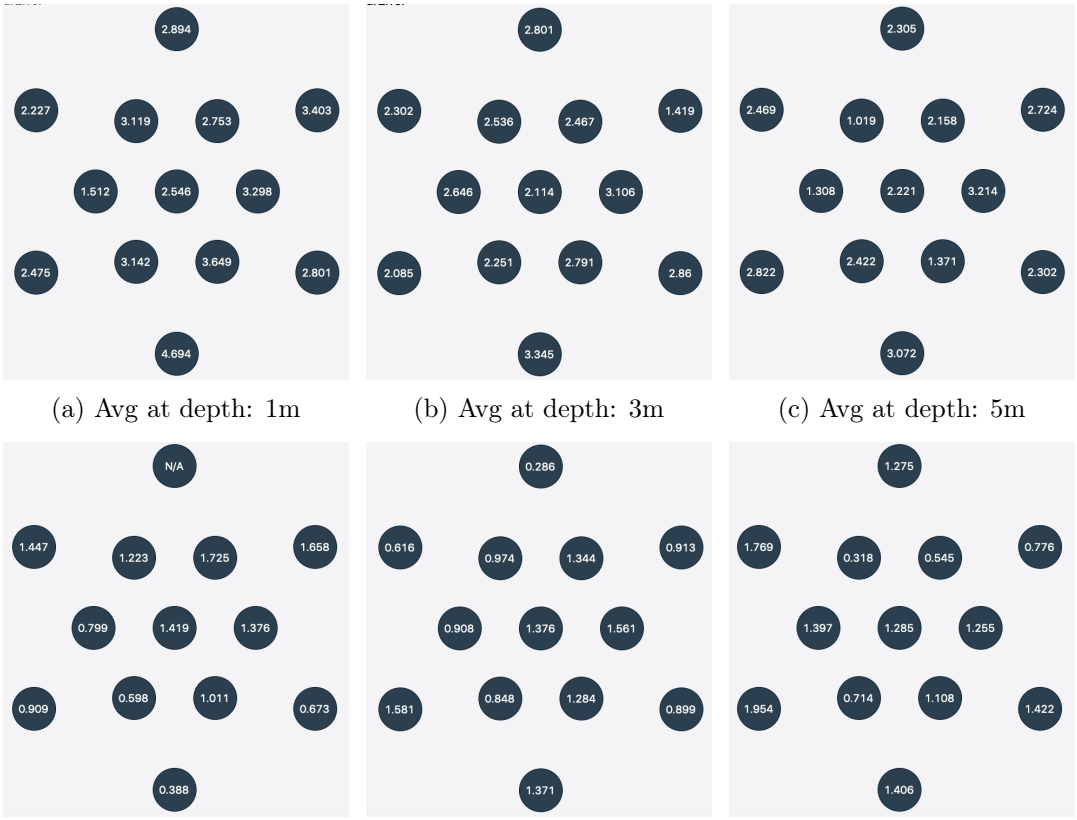
(a) Avg at depth: 1m (b) Avg at depth: 3m (c) Avg at depth: 5m

(d) Std at depth: 1m (e) Std at depth: 3m (f) Std at depth: 5m

Figure 12: Data Visualisation of Subjects without Glasses

6.6.2 Standard Deviation

Figure 11 shows the overall standard deviation for each spherical marker. We can see that the standard deviation tends to be higher for the closest depth of markers, which again makes sense considering the fact the visual surface is higher, so the angular error over each marker will also be higher. It may also be due to the smooth pursuits that take place in order to focus on something that is too close. For most markers, it seems to be the case that the further it is, the lower the standard deviation. Overall, standard deviation seems to be higher for spherical markers closest to the edges which indicates contrasting abilities to accurately visualise such areas.



(a) Avg at depth: 1m (b) Avg at depth: 3m (c) Avg at depth: 5m

(d) Std at depth: 1m (e) Std at depth: 3m (f) Std at depth: 5m

Figure 13: Data Visualisation of Subjects with Glasses

6.6.3 Influence of Eye Corrections

We can also compare subjects who wear glasses (Figure 13 with those who don't need glasses or who were wearing contact lenses (Figure 12. Note that subjects who need glasses were unable to wear them during the experiment as this would cause too many reflections from the eye-tracking device. We can first note that the standard deviation is much lower when we separate into these two categories, which makes sense given the fact these are more specific categories. Another key observation from this data is that the average angular error for subjects with glasses is generally lower than for subjects who don't need glasses or who were wearing contact lenses. While this can seem surprising at first, it may be due to the fact subjects who have lower eyesight concentrated more on the targets as they knew they couldn't see as well.

7 Conclusion

In this project, we began by understanding the value of human visual attention and the ways in which it can be analysed in order to then compare different state-of-the-art technologies for eye-tracking and adopt the best-suited one for our project. One of the main reasons for choosing the Pupil Labs eye-tracking device was for its extensive functionalities and the development resources it provides. Therefore familiarisation with this device and its features was a key part to this project and notably to further understanding of the eye-tracking process and the integration of the latter in virtual reality. Once sufficient knowledge on Unity and the Pupil Labs features had been acquired, this was used to create a bench-marking scene through which we would be able to evaluate the accuracy of the eye-tracking device in virtual reality, this was a two-step process which resulted in two separate scenes that follow the development process of the project. An important aspect in evaluating human visual attention in virtual reality is the ability to detect fixations as they give the most indication on intended gaze. Thus gaining insight on ways to detect fixations, notably with the Pupil Labs device, was of much importance to the project. Finally, an experiment was conducted using the Spherical Bench-marking scene to assess the angular error between predicted gaze and recorded gaze. From this experiment, substantial data was collected allowing for us to carry out post-processing and data visualisation from the latter.

The scenes created and the data collected from the experiment can be used to carry through further analysis of human visual attention in virtual reality. In particular offline fixation detection can be used for a more precise post-processing, i.e. without overlapping fixations.

One further initial aim for the end of this project was to use the data to identify regions of interest from subjects that interact with 3D models, such as point clouds, in the virtual world. This aim was envisioned with re-usability of scripts from a previous project in mind, notably scripts related to point clouds and fixation detection that also used features from Pupil Labs [12][13]. However, the ever-changing nature of Pupil Labs open-source software and scripts rendered the re-usability of these scripts more laborious than anticipated, especially due to the fact the eye-tracking was previously performed in 2D as opposed to 3D, thus starting the project from scratch was in fact a more viable and time-worthy solution. In addition to the changing eye-tracking software, limitations were also reached regarding updates of the virtual reality device and software used to run it as evoked in the Analysis section.

Since the project is fully adapted to 3D eye-tracking, it should be adaptable to carrying out an evaluation of regions of interest on 3D models in the virtual world, particularly under the form of heat-maps. The assessment of the eye-tracking accuracy implemented in this project can also be used to then evaluate this interaction with 3D models.

References

[1] Andrew T. Duchowski. *Eye Tracking Methodology.* Springer, Theory and Practice, Third Edition, 2017.

[2] iMotions: Types of Eye Movements
`https://imotions.com/blog/types-of-eye-movements`

[3] Pupil Labs | VR/AR
`https://pupil-labs.com/products/vr-ar/`

[4] VIVE Pro Eye with Tobii Eye Tracking
`https://vr.tobii.com/products/htc-vive-pro-eye/`

[5] Fove
`https://www.getfove.com/`

[6] Pupil Labs Repositories
`https://github.com/pupil-labs`

[7] M. Kassner, W. Patera, and A. Bulling. *Pupil: An open source platform for pervasive eye tracking and mobile gaze-based interaction.* April 2014.
`http://arxiv.org/abs/1405.0006`

[8] Pupil Labs Documentation
`https://docs.pupil-labs.com/`

[9] J.Canny *A computational approach to edge detection.* Pattern Analysis and Machine Intelligence, IEEE Transactions on, 6 (1986), 679–698.

[10] J.Steil, M.Huang, A.Bulling *Fixation Detection for Head-Mounted Eye Tracking Based on Visual Similarity of Gaze Targets.* 2018

[11] T.Urruty, S.Lew, N.Ihadeddene, D.Simovici *Detecting eye fixations by projection clustering.* 2007
`https://dl.acm.org/doi/10.1145/1314303.1314308`

[12] E. Alexiou, T. Ebrahimi, P. Xu *Towards Modelling of Visual Saliency in Point Clouds for Immersive Applications* 2019
`https://ieeexplore.ieee.org/abstract/document/8803479/`

[13] Peisen Xu *Quality assessment of 3D representations in Head Mounted Displays (HMDs).* 2019, MMSPG, EPFL

A Appendix

`https://github.com/manonmichel/EyeTracking`