CHILI
computer-human interaction
in learning and instruction

LEARN
Center for Learning Sciences

Augmented Reality and Eye-Tracking in iOS Development

Author:
PRUNE OLLIER

Supervisor(s):
Lucas Burget

Professor:
Pierre Dillenbourg

January 5, 2024

Contents

1 Introduction

This report explores the development of an iOS eye-tracking app using Apple's ARKit technology. Executed within the CHILI research lab at EPFL in the context of my Bachelor's semester project, the focus was not merely on app performance but on the research and development that lead to the accuracy analysis of this technology. The project's core objective was to harness the eye-tracking capabilities within Apple's ARKit framework to build an iOS app. While the current iteration is optimized for iPhone, strategic considerations anticipate future adaptations for iPads, broadening the potential applications of this technology. Key aspects include understanding ARKit and SpriteKit referentials, the process of estimating and projecting gaze on the screen, calibration techniques, and the results of these implementations. The project's code is available via this link : `https://github.com/Pruneollier/Eye_Tracking_PDS`

2 Method and Material

2.1 Methodology

The project used Swift, Apple's programming language, to harness the different frameworks. The development process included:

- Learning Swift and understanding basic iOS app structures.[4]
- Exploring ARKit and SpriteKit through hands-on experimentation and Apple's documentation and tutorials.[2] [6]
- Implementing AR and eye-tracking features in an iOS app, with a focus on integrating ARKit's real-world tracking and SpriteKit's graphical capabilities.
- Iterative testing and refining for optimal performance.
- Analysis of the results.

2.2 Apple Frameworks

Understanding the various frameworks we worked with form the backbone of iOS development, each offering unique functionalities crucial for augmented reality (AR) and eye-tracking integration. ARKit enables the AR experience, SpriteKit manages 2D animations and graphics on the screen, and UIKit handles the user interface. A comprehensive understanding of these tools is essential for effectively using their capabilities.

ARKit

ARKit is Apple's framework for building augmented reality (AR) experiences for iOS devices. It allows developers to blend digital content with the real world through the device's camera. Key features include tracking the device's position and orientation in the real world, detecting and tracking human faces (**ARFaceAnchor**), anchoring virtual objects in the real environment, and the estimation of the user's gaze in the 3D space **lookAtPoint**. The virtual 3D objects in ARKit are **SCNNode**'s and the measuring unit is the **meter**. [1]

SpriteKit

SpriteKit is Apple's 2D game development framework, used for creating rich graphical animations and interactions. It's integrated with ARKit to render 2D content in an AR environment, offering tools for animations, making it ideal for creating interactive 2D content within AR apps. The virtual 2D objects in SpriteKit are the **SKNode**'s and the measuring unit is the **CGPoint**.[3]

UIKit framework

While UIKit is an essential framework for designing and managing user interfaces in iOS apps, its role in this project was minimal. It was employed primarily as a formality to ensure the app's compatibility and operability within the iOS environment. The focus was largely on the capabilities of ARKit and SpriteKit, with UIKit serving only to facilitate the basic running of the app rather than playing a significant part in the app's overall functionality or user experience.[5]

3 Understanding ARKit referentials

The development of the app involved a deep understanding of various referentials and coordinate systems. Due to the limited (or confusing) documentation available of those used in ARKit, several hands-on experiments were conducted to ensure the correct comprehension of the overall system. These consisted in printing the position of a fixed point in the different referentials. This practical approach allowed us to understand and effectively work with the correct referentials.

3.1 World referential

- **Orientation and Position:** In ARKit, the world referential's origin is located at the device's frontal camera. The orientation of this referential is characterized by the z-axis pointing behind the screen, the x-axis pointing to the left, and the y-axis pointing downward. Notably, this setup does not conform to the conventional right-hand rule. See Figure 1 for a modelisation of the world referential.

- **Application in ARKit:** The world referential is crucial for positioning objects in the 3D world space as perceived by the camera. In the case of the **ARFaceAnchor**, its position is defined within this world referential, meaning its location is relative to the camera's position in real time.

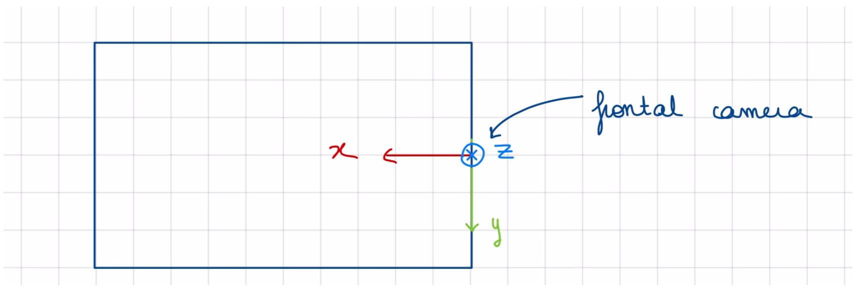


Figure 1: World referential

3.2 Face referential

- **Orientation and Position:** The face referential in ARKit is centered around an ARFaceAnchor, which is an embedded feature in ARKit. The ARFaceAnchor is defined as an anchor for a unique face visible in the front-facing camera. The orientation of axes in the face referential follows the right-hand rule applied to the face: the x-axis points to the user's left, the z-axis points towards the screen, and the y-axis points upwards. See Figure 2 for a modelisation of the face referential.

- **Application in ARKit:** This referential is essential for tracking the user's eye position and gaze. For instance, the position **lookAtPoint**, returned by an embedded function in ARKit that approximates the user's gaze position, is expressed in this face referential.

3.3 rightEyeTransform

- **Orientation and Position:** The **rightEyeTransform** is an embedded transform matrix indicating the position and orientation of the user's right eye (from the camera's point of view). Its position is expressed in the face referential, and its orientation is expressed through the right eye referential, which follows the same orientation and right-hand rule as the face referential, only its origin is the user's eye. It is important to note that, due to the camera's mirror effect, the origin of this referential is actually located on the user's left eye. See Figure 2 for a modelisation of the right eye referential.

- **Application in ARKit:** The rotational components of this matrix denote the orientation of the eyeball. Rotations about the x-axis indicate upward or downward movement of the pupil, while rotations about the y-axis indicate left or right movement. There is no rotation about the z-axis for the eye.

See Figure 3 for an all-together representation of the referentials when using the app, seen from the user's left.

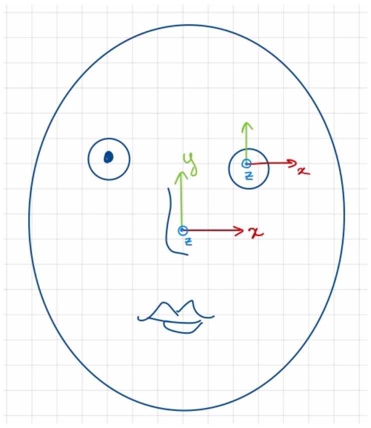


Figure 2: Face referential

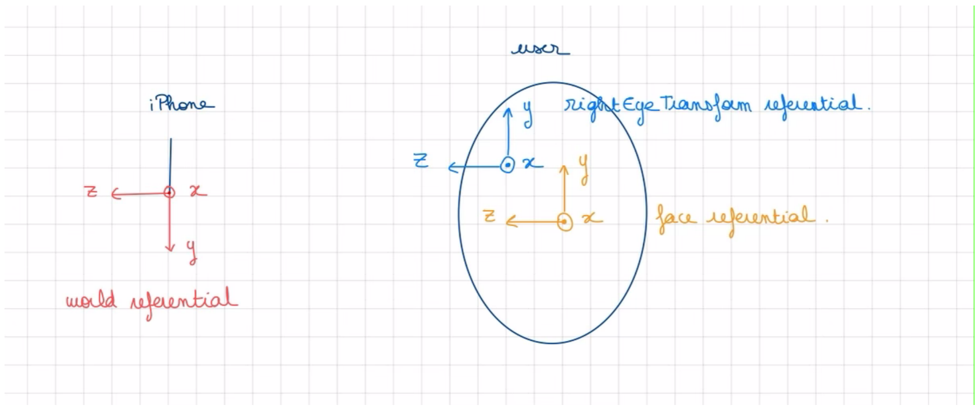


Figure 3: ARKit referentials

4 Understanding the SpriteKit referential

4.1 SpriteKit Scene Overlay

- **Concept of overlay:** In our application, we use a SpriteKit scene (SKScene) as an overlay. An overlay in this context is a Sprite Kit scene that is rendered on top of the SceneKit content. In simpler terms, it's like layering a SpriteKit scene (which is a 2D scene) over a SceneKit view (which is a 3D scene). This is especially relevant for our project as the overlay is used over the ARSceneView content, which is the main content of ARKit.

- **Purpose in the app:** The primary function of this overlay in our app is to display the calculated estimation of the user's gaze. The overlay acts as a canvas where we can draw or place 2D elements, like an indicator of where the user is looking.

4.2 SpriteKit referential

- **Orientation and position:** In SpriteKit, the referential or coordinate system used has its origin at the bottom left corner of the iPhone when the iPhone is held in portrait mode. However, our app requires the phone to be held horizontally with a locked orientation and the camera to the right. In this horizontal orientation, the origin of the referential does not shift. The y-axis now points to the right of the screen (which was the top in portrait mode). The x-axis points downwards (which was towards the right in portrait mode). Coordinates are expressed in CGPoints.

- **Implications in eye tracking:** The change from ARKit's world referential to SpriteKit's referential is crucial for our application. The graphical representation of the gaze estimation indicator, needs to be positioned correctly in this modified referential to ensure they accurately reflect the user's gaze direction on the screen. We will see later how to convert ARKit coodinates to SpriteKit coordinates. See Figure 4 for a representation of both the SpriteKit and ARKit's world referentials.

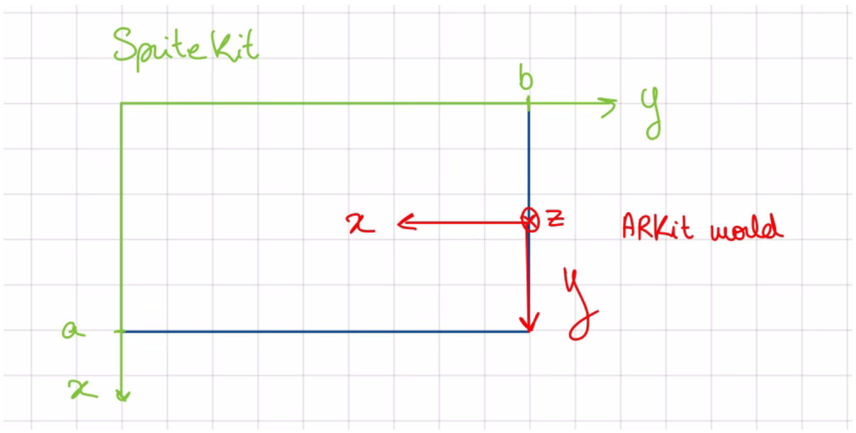


Figure 4: SpriteKit and ARKit world referentials

5 Estimate and project gaze on the screen

5.1 Estimation of the gaze

The embedded function **lookAtPoint()** in ARKit returns a simd_float3 matrix, representing a position in the ARKit world referential. This matrix serves as an estimation of where the user is looking in the 3D space. However, our goal is to have an on-screen indicator that dynamically follows the user's gaze when they look at the screen. To achieve this, we project the estimated gaze point onto the 2D SpriteKit overlay. This projection ensures that the indicator is displayed only if the user is looking at a point on the screen. If the user's gaze is directed elsewhere, the projected point's coordinates will fall out of the display range of the overlay.

5.2 Projection Process

We implemented a function **linePlane()** to compute the intersection between the screen plane and the z-coordinate of the user's gaze (derived from lookAtPoint). This function returns a SIMD3<Float> matrix, representing the position of this intersection in the 3D space. The z-coordinate of this intersection point is zero, as the intersection occurs on the plane of the screen.

5.3 Conversion to SpriteKit Referential

Once we have this 3D point (ie. SIMD3<Float> matrix) expressed in the world referential of ARKit, the next step is to convert it into a 2D point (ie. CGPoint) within the SpriteKit referential. The function **convertARKitPositionToSpriteKitPoint()** is used for this conversion. At each call of this function (ie. everytime the session frame is updated), the value returned is stored in an array pts, for future purposes.

Let's break down how this function works.

5.3.1 Parameters and Components

- **worldPosition (SIMD3<Float>):** This parameter represents the 3D position in the ARKit world referential. It's given in a SIMD3<Float> format, which holds the x, y, and z coordinates in 3D space.
- **a and b (Dynamic Measurements): a** and **b** respectively represent the width and length of the iPhone's screen, in CGPoint. These values are dynamically computed at the beginning of the session to adapt to the device's size. In fact, these values are embedded in the ARSCNView's frame properties [7]. In the context of the app, since the phone is held horizontally, **a** would correspond to the screen's shorter side, and **b** would be the longer side (see Figure 4).
- **pointsPerMeter (Calibration Value): pointsPerMeter** is a calibrated value determined during the calibration phase of the app. It represents an approximation of the number of CGPoints per meter in the real world. This value is critical for translating real-world distances (in meters) to screen distances (in CGPoints). We will discuss how was this value computed later in Section 6.

5.3.2 Function mechanics:

- **X-Axis Conversion:** The x-coordinate in the SpriteKit scene (CGPoint.x) is calculated by taking half of **a** (the screen's width) and adding the world position's y-coordinate (note the axis switch due to the different orientations of the referentials) multiplied by pointsPerMeter.
- **Y-Axis Conversion:** The y-coordinate in the SpriteKit scene (CGPoint.y) is derived by subtracting the product of the world position's x-coordinate and **pointsPer-**

Meter from **b** (the screen's length). This inversion is necessary because the y-axis in SpriteKit starts on the left and increases to the right.

5.4 Stabilizing the gaze indicator

To prevent the gaze estimation indicator from being overly sensitive and unstable, an averaging strategy is employed. The function **averageOfLastKCGPoints()** is used to calculate the average value of the last 10 projections of the user's gaze, stored in the array **pts**. This averaging approach helps in smoothing out the movement of the indicator, avoiding a jittery or shaky appearance.

6 Calibration

The calibration process in our iOS eye-tracking app is a critical step to ensure accurate gaze tracking and projection. It operates by guiding the user to look at specific points on the screen. These points are strategically positioned at the upper left, upper right, and lower left corners of the screen. The function then captures the user's gaze position at each of these points, which allows the app to estimate the screen's dimensions in meters and establish a conversion rate between ARKit's 3D points and SpriteKit's 2D points. Here's a detailed explanation of this process:

- **Recording of the gaze's position with a calibration point:** At the start of the session, a black point (**calibNode**) appears in the upper left corner of the screen. The user should look at this point (here we could add an instruction for better understandability). The position of the **lookAtPoint** (the user's gaze point in the ARKit world) at this moment is then stored in a SCNNode. This action is repeated when the **calibNode** is moved to the upper right corner of the screen and one last time when it is to the lower left corner. Each time, the position of the **lookAtPoint** is stored in a different SCNNode.

- **Calculation of Screen Dimensions:** An estimation of the length in meters of the screen is calculated by subtracting the x-coordinate of the upper right corner position from the x-coordinate of the upper left corner position. Similarly, the width in meters of the screen is computed by subtracting the y-coordinate of the upper right corner position from the y-coordinate of the lower left corner position.

- **Computing PointsPerMeter:** With the length and width of the screen determined in meters, the **pointsPerMeter** value can be calculated. It is computed by dividing **a** (the width of the screen in CGPoints, computed as explained in Section 5.3.1) by the previously computed width in meters. This value represents the number of CGPoints per meter and is essential for converting 3D points in ARKit to 2D points in SpriteKit.

7 Results

The project's performance was evaluated in three experimental assessments:

- The first one consisted in having the user follow their finger with their gaze while moving it randomly across the screen. This movement was designed to be unstructured, encouraging the finger to cover various screen areas. This approach allowed for a comprehensive analysis of the gaze estimation accuracy over varied movements and timeframes. This approach allowed for a comprehensive analysis of the gaze estimation accuracy. See Section 7.1.

- The other two were conducted to analyze how different parameters might influence the gaze accuracy. Key variables included the rotation of the user's head and its position in the world's referential. In contrast to the first experiments, the user would touch a point on the screen in the beginning of the session, which remained fixed for the remaining time of the session. This approach allowed for the certainty that the analyzed parameters were the only factors affecting the gaze estimation's position. See Sections 7.2 and 7.3.

Each assessment was conducted throughout one entire app session, from start to close, with the session's duration displayed on the graphs in a "mm:ss" format.

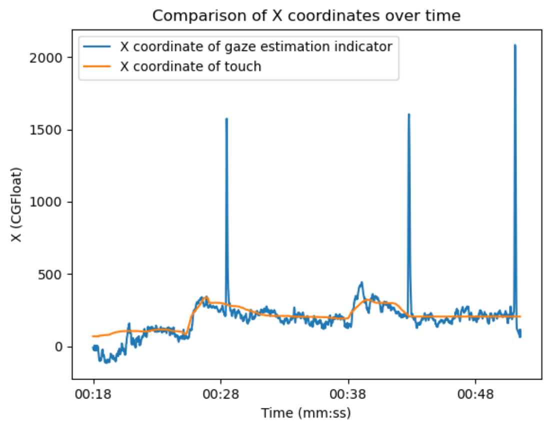
7.1 Analysis of Gaze Estimation Accuracy

Graphs 5 and 6 illustrate the evolution of the x (vertical) and y (horizontal) coordinates for both the touched point and the estimated gaze point. These graphs reveal sudden peaks in the estimation coordinates, which we attribute to the user's blinks. The computed average difference in x coordinates was 54.44 CGPoint units, while for y coordinates, it was 69.51 CGPoint units. This data suggests that the app demonstrates better performance in estimating the gaze's position on the x-axis compared to the y-axis, highlighting an area for potential improvement in horizontal gaze tracking accuracy.

7.2 Head Rotation and Gaze Estimation Correlation

To understand the influence of the user's head rotation on their gaze estimation accuracy, two separate pairs of graphs were plotted (Figures 7 and 8), each representing the head's rotation about the x and y axes (orange curve). Alongside these, plots of the absolute differences between the x and y coordinates of the touched and estimated gaze points were added (red curve). Head rotation about the z-axis wasn't included in the analysis because it usually shows minimal rotation, making it irrelevant for this particular study.

For the x-axis rotation, a clear correlation was observed on Figure 7 with the x-coordinate differences: peaks in x-axis rotation often coincided with peaks in the x-coordinates difference. However, there was little to no correlation between the y-coordinates difference and the x-axis rotation, as these two curves did not display coordinated trends.



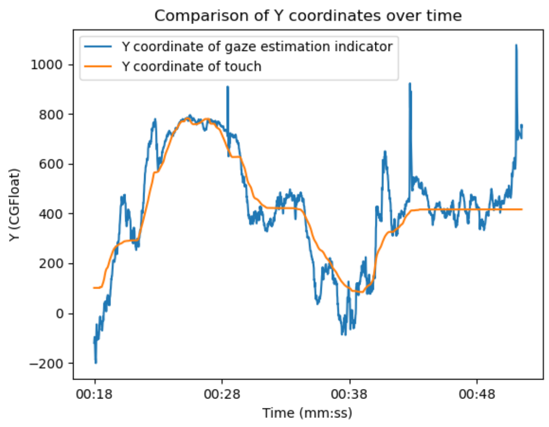


Figure 5: x coordinates of touch and gaze over time

Figure 6: y coordinates of touch and gaze over time

Conversely, the y-axis rotation showed on Figure 8 a strong influence on the difference in the y-coordinates between the touched and estimated points, with minimal impact on the x-coordinates difference. This analysis highlights the significant role of head rotation in gaze estimation accuracy, particularly its axis-specific effects.

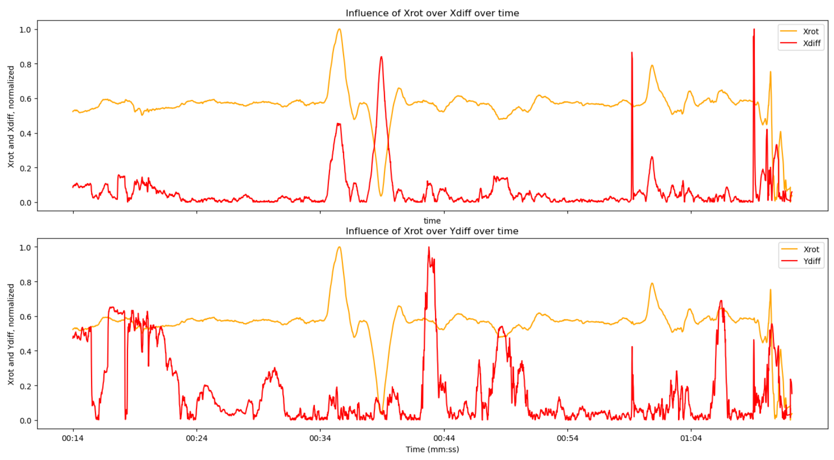


Figure 7: influence of rotation over x-axis on the gap

7.3 Head Position Influence on Gaze Estimation

To assess how head position changes affect gaze estimation, three pairs of graphs were plotted (Figures 9, 10 and 11). Each pair showed the differences between the x and y coordinates of the touched and estimated points. The first pair analyzed the x-coordinate, the second the y-coordinate, and the third the z-coordinate of the user's head position in the real world.

The results indicated that the app's performance was consistent regardless of the head's z-coordinate (Figure 11). However, a strong correlation was observed between

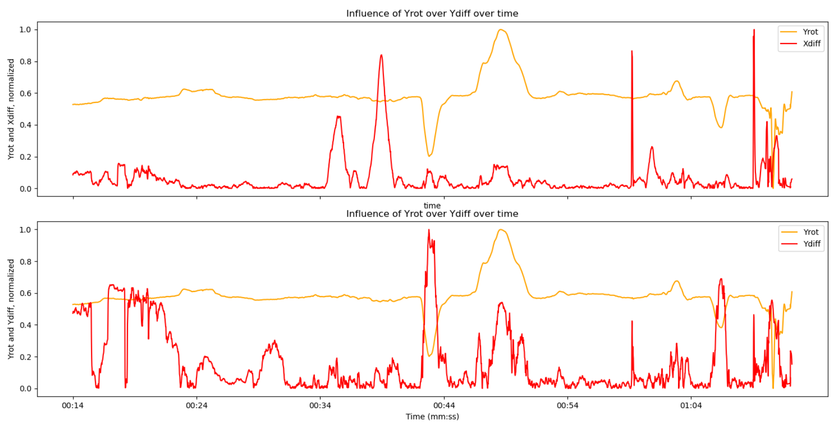


Figure 8: influence of rotation over y-axis on the gap

the head's x-coordinate and the gaze's y-coordinate (Figure 9). Additionally, the head's y-coordinate showed a visible correlation with both the x and y coordinates of the gaze estimation (Figure 10), highlighting the significant influence of head position on gaze accuracy.

Summary of Results

The performance evaluation essentially involved experiments where the user followed their finger with their gaze across the screen. The experiments showed that gaze estimation accuracy was influenced by head rotation and position.

- The gaze's x-coordinate was primarily affected by the head's y-coordinate and rotation around the x-axis, highlighting the differences between ARKit and SpriteKit referentials.
- Conversely, the gaze's y-coordinate was influenced by the head's x and y coordinates and rotation about the y-axis, indicating better performance in estimating gaze along the x-axis and less stability for the y-coordinate.
- The head's distance to the screen and rotation about the z-axis had minimal impact on gaze estimation.

However, it's important to note that this was preliminary work, as each assessment was conducted over only one session. This limits the breadth of the data but provides a valuable starting point for further research.

8 Conclusion

This report documents a comprehensive exploration of ARKit and SpriteKit frameworks in developing an iOS app for eye-tracking. It highlights the challenges encountered during the project, particularly in gaze estimation accuracy influenced by factors like head rotation and position. The findings suggest a strong correlation between these factors and gaze accuracy, providing valuable insights for future development.

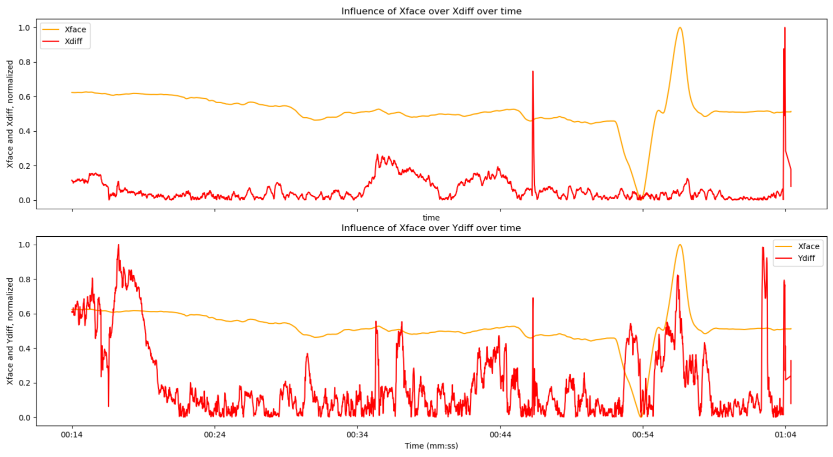


Figure 9: influence of change of X-coordinate of the face on the gap

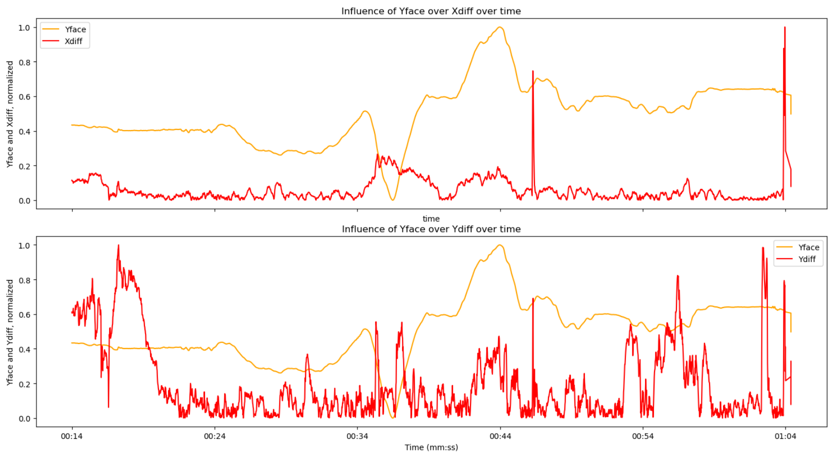


Figure 10: influence of change of Y-coordinate of the face on the gap

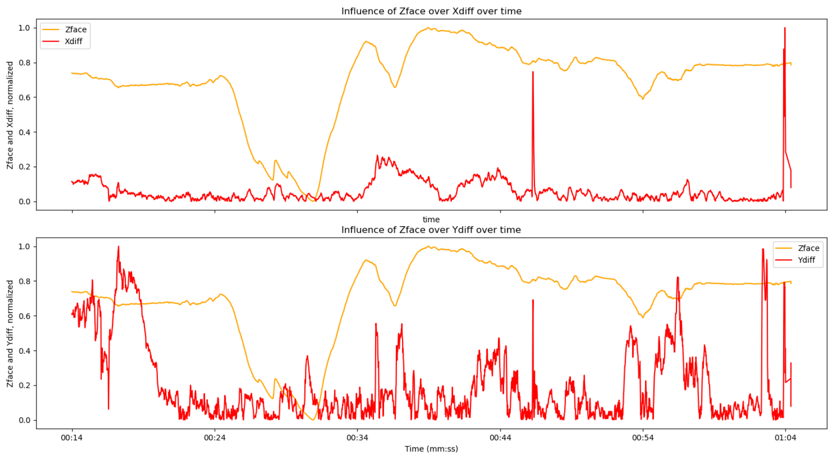


Figure 11: influence of change of Z-coordinate of the face on the gap

References

[1] *ARKit Documentation.* `https://developer.apple.com/documentation/arkit/`.

[2] *ARKit Tutorial.* `https://www.kodeco.com/378-augmented-reality-and-arkit-tutorial?page=1#toc-anchor-001`.

[3] *SpriteKit Documentation.* `https://developer.apple.com/documentation/spritekit/`.

[4] *Swift Documentation.* `https://docs.swift.org/swift-book/documentation/the-swift-programming-language/thebasics`.

[5] *UIKit Documentation.* `https://developer.apple.com/documentation/uikit`.

[6] *UIKit Tutorial.* `https://developer.apple.com/tutorials/app-dev-training#uikit-essentials`.

[7] *UIView.frame.* Apple Documentation. `https://developer.apple.com/documentation/uikit/uiview/1622621-frame`.