DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

HoloAssist: Integrating an Augmented Reality Headset in a Flight Simulator

Matteo Nardini

DEPARTMENT OF INFORMATICS

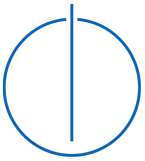
TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

HoloAssist: Integrating an Augmented Reality Headset in a Flight Simulator

HoloAssist: Integration eines Augmented-Reality-Headsets in einen Flugsimulator

Author:	Matteo Nardini
Supervisor:	Prof. Gudrun Klinker, Ph.D.
Advisors:	Michael Zintl
	Carsten Schmidt-Moll
	Sandro Weber
	Sven Liedtke
Submission Date:	15th March 2022



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15th March 2022 Matteo Nardini

Abstract

Flying an aircraft is a complex task, especially in abnormal situations. Although the cockpit environment has already been optimized to be as clear and effective as possible, there is the hypothesis that comparatively recent technologies like augmented reality (AR) could further improve the situation. Testing this hypothesis requires being able to quickly develop AR experiences that can be used on innovative hardware like the Microsoft Hololens in the context of a flight simulator and then be evaluated by professional pilots.

Achieving this result requires a flexible and easy to use platform that simplifies the creation of these experiences: such platform is the main contribution of this thesis. Additionally, the developed framework has been used to create some AR aids for landing at the Innsbruck airport, which have then been evaluated by real pilots on a fixed-platform aircraft simulator.

Contents

1. Introduction

For the entirety of recent history, augmented reality has been a staple of science fiction stories, regardless of their setting or of the medium used to portray them, be it written text, pictures, movies or video-games. This concept is so ingrained in popular culture that almost every technology enthusiast or sci-fi aficionado has a mental image of how a system using it could look like.

What is less known is that augmented reality prototypes have been a reality for a long time, since the late 1970s[1]. However, those early systems were difficult to use, as they required expensive devices and lengthy calibration procedures. This effectively restricted them to military research environments and, later, university research laboratories.

Luckily, this is not the case anymore: in the last ten years the AR field has moved forward very quickly, thanks to the fact that current-generation mobile devices pack enough performance to support these use cases and thanks to the commercialization of the associated field of virtual reality (VR), which finally has become available as an off-the-shelf product. This in turn caught the interest of big players like Microsoft, Apple, Google, Meta and others, which have been doing a lot of the work that is necessary to transform AR from a research "novelty" to a platform easily accessible by developers with a less specific background[2, 3, 4, 5].

The first concrete proofs of these advancements have been AR video-games like PokemonGO[6], which for the first time have brought an (admittedly limited) augmented reality experience to the general public, and the integration of an high-level AR application programming interface (API) in most of the major platforms, from the Unity game engine[7] to WebXR[8]. The most interesting result of the renaissance of this field has been the development of AR head-mounted displays (HMDs) like the Microsoft Hololens[4] (shown in Figure 1.1) and the Magic Leap One [9]. For the first time, they allow the development of AR experiences that are very close to the ideal ones imagined in science fiction stories, as such devices are fairly comfortable to wear, do not require lengthy calibrations, are capable of overlaying augmentations on a decent portion of the wearer's field of view (FOV) and are self-contained, meaning that they do not require external tethers.

This hardware availability alone already offers many possibilities for devel-

Figure 1.1.: The Microsoft Hololens 2.

opers and researchers wanting to experiment with AR, as these devices are cheaper and easier to use than what was previously available as a commercial product. However, another significant part of these improvements regards how developers interface with such devices: although HMDs have existed for a while, they often offered a very low level API that only allowed to specify the color of each individual pixel of the display. In order to obtain an AR-capable system, the developer had to acquire the information coming from an external tracker, process it and then generate the correct 2D image to be displayed on such HMD, which is non-trivial and requires a significant amount of software engineering work. A concrete example of this is the current version of the ROSIE[10] rotorcraft simulator, which uses a custom C++ 3D engine that integrates the pose information coming from an external tracker and the description of the desired virtual 3D scene to produce an image that is then used to drive the individual pixels of an old-generation HMD. Besides the initial development cost, this kind of setup also presents recurring costs, as new developers and researchers need to become familiar with a fairly complex setup, and often end up spending a significant amount of time on software engineering issues rather than their original research question.

Devices like the Hololens eliminate a meaningful portion of this complexity, as they abstract away the inner workings of the tracking system and other implementation details to offer a higher-level platform that explicitly targets the development of AR applications. Moreover, the industry decided to converge early on a device-independent API called OpenXR[8], which allows developers to write their application once and then be able to run it on a multitude of different AR devices. This caught the interest of game engine designers, who saw an easy opportunity to extend their platform to also support OpenXR as a target, leading to the current situation in which one can quickly write a Hololens

application in Unity, enjoying the features of a modern game engine, without having to focus too much on the low-level details of how the Hololens actually works. This allows developers and researchers to shift their focus from *building* an AR system to just *using* an AR system, allowing them to pursue their research question rather than work on software engineering issues.

1.1. Related work

Concrete evidences of these recent developments are readily available. Not only devices like the Microsoft Hololens[4] and the Magic Leap One[9] are commercially available, they are already being put to use. An example is the field of medical augmented reality, where researches are using the Hololens to aid the positioning of complex medical machinery[11, 12, 13] and train surgeons in complex procedures[14]. The Hololens has also already been used with success in real human surgeries[15]. Another example is the suite of commercially available AR applications available for use on the Hololens 2, both offered by Microsoft themselves[16] and by third parties[17]. A last example is given by the AR experiences already developed for rotorcraft simulators, which show promising results in highlighting obstacles and displaying approach paths[18]. Exploration is happening also in the direction of single-pilot cockpits[19].

1.2. Problem statement

Aircrafts are complex environments with a low tolerance for errors. Pilots have to monitor a number of different flight parameters while at the same time having to fly the aircraft, navigate the planned route and communicate with an external traffic control entity. Especially in abnormal situations, this requires deep knowledge of the aircraft and quick decision-making skills. Moreover, there is the risk of "cognitive overload", in which the pilot has to deal concurrently with too many information, which could lead to missing an important data point with potentially severe consequences.

A reasonable hypothesis is that a part of the complexity of these situations is accidental[20] and that it is due to how the information is presented to the pilot. The design of aircraft cockpits and their interfaces with the pilot have been studied for many years and are therefore already highly optimized for safety and clarity, making further improvements difficult to achieve. However, research focused primarily on improving standard and well-known technologies, like normal displays, gauges, switches and buttons. Due to the limitations mentioned

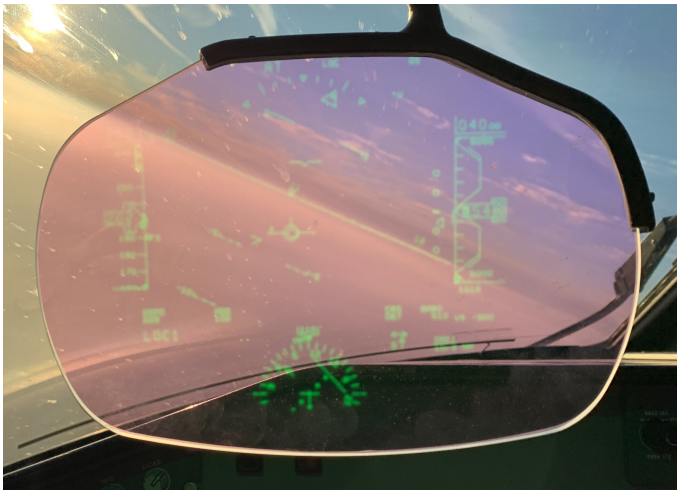


Figure 1.2.: The view shown by a traditional head-up display (HUD), composed only of 2D elements.

at the beginning of chapter 1, experimenting with AR solutions in these contexts has been difficult, with available HMDs solutions being cumbersome to setup and/or being limited to displaying only 2D elements like speed and altitude, as shown in Figure 1.2.

The recent availability of devices like the Microsoft Hololens has sparked interest in this area, especially as far as rotary-wing aircrafts are concerned[21, 18]. Since augmented reality experiences aim to blend seamlessly with the way humans usually experience reality, there are well-justified hopes that they will represent the next step in user interface and user experience design, as they should be significantly more intuitive to use than traditional display-based solutions. If this turns out to be true, then AR could be used to present information to pilots in a more efficient way, leading to even safer flight conditions.

Another context in which AR could potentially be beneficial are electric vertical take-off and landing aircrafts (eVTOLs): they will probably be commanded by a single pilot with limited training, who will therefore require all the assistance that technology can possibly offer.

The only way to prove (or disprove) this belief in augmented reality is to use modern AR HMDs first in simulators and then in real aircrafts. New, more advanced visualizations will have to be shown to pilots and their feedback on whether these solutions are better or worse than what is currently available will have to be collected. In order to do this, it must be as easy as possible to create AR experiences for flight simulators, to enable a rapid feedback cycle that will allow to test many different visualization ideas and to iterate on them quickly. Moreover, developing these AR experiments should require as little

programming knowledge as possible, because researchers working on them will likely have very different backgrounds and a varying level of computer science expertise.

To achieve this objective, the appropriate tooling and workflows to create these AR experiences have to be developed, and this is the main aim of this thesis. Moreover, to show that the resulting platform is actually helpful, an initial AR experience will be developed and presented to a small sample of experienced pilots, who will give an assessment of its usefulness. The aim of such experiment will be to create AR augmentations that help pilots to land at the Innsbruck airport in condition of limited visibility, as that landing is comparatively more difficult than many others due to the orographic formations that surround the airport. The developed tooling consist mainly of two parts:

1. "HoloAssist", a Unity application that runs on the Hololens and contains most of the logic required to display 3D augmentations in the setting of a flight simulator.

2. "HoloAssist Apps", a collection of scripts that use the API exposed by HoloAssist to draw the augmentations that have to be shown to the pilot in the context of an experiment.

TL;DR. In order to evaluate whether AR solutions are actually helpful in the context of an aircraft cockpit there is the need to quickly build AR experiences and test them on a flight simulator. This thesis project builds the tooling and workflow needed to achieve these capabilities.

1.3. Technical limitations

Although devices like the Hololens seem very promising, they are still targeted mostly for research environments and experimentation, as they still have rough edges and limitations, especially for more exotic use cases like the one presented in this document. First of all, they have a limited battery life (about two hours of actual usage on the Hololens 2): using current iterations of these devices in a real aircraft will therefore require some integration with the airplane's electrical system in order to keep them charged. Additionally, the computing power available on-device is fairly constrained, which limits the complexity of the augmentations that can be shown.

The Hololens also has a pretty significant problem specific to the use case presented in these pages. The inside-out tracking relies heavily on an inertial measurement unit (IMU) embedded in the device. This means that whenever the

device is used on a moving platform (like a car, a ship or an aircraft) the tracking is sub-optimal and often lost completely, making the device unusable. The eventual desire is to be able to show augmentations while being in full-motion flight simulators and in real aircrafts, but this will not be possible with the current iteration of the Hololens (unless an external tracking system is used, with all the complexity that it would entail). Fortunately, Microsoft is aware of this issue and working on a solution[22], which however is currently only effective only for low-jerk motions like those of a cruise ship (rather than the sudden changes in acceleration of a sport car).

1.4. Thesis overview

The remainder of this document describes the proposed solution for the problem statement expressed in section 1.2. Chapter 2 deals with the Hololens side of the solution: each section focuses on a different problem that HoloAssist had to solve in order to display the desired augmentations. A particular mention goes to section 2.3, which explains how such augmentations are represented in memory and rendered, which is arguably the most complex part of the Hololens application. Chapter 3 describes how HoloAssist's API can be used to create AR experiences inside a fixed-platform simulator and what was implemented to enhance the landing at Innsbruck's airport. Chapter 4 describes how the quality of the augmentations developed for Innsbruck's airport was assessed and chapter 5 summarizes the main outcomes of the whole project and some potential future steps.

Additional content is available in the appendices, which contain a detailed documentation of the HoloAssist API and a semi-serious description of some of the more puzzling bugs that were encountered during the development of the Hololens application.

2. HoloAssist

As explained in section 1.2 the main aim of this thesis is to build a solution that allows to easily display arbitrary augmentations in a fixed-platform flight simulator. The best way to achieve this result was to build "HoloAssist", a Hololens application developed using the Unity game engine that contains all the logic that is necessary to display a desired 3D mesh:

1. at a specific real-world location identified by some geographic coordinates, so that by wearing the Hololens inside the flight simulator and looking "outside" the cockpit window one would see that 3D mesh anchored to that specific geographical location. This type of augmentation is called "geo-fixed augmentation";

2. at a specific location inside the flight simulator cockpit, so that by wearing the Hololens and looking at that specific point one would see the 3D mesh anchored to it. This type of augmentation is called "plane-fixed augmentation".

Since developing directly for the Hololens is not trivial for someone with limited computer science experience, decoupling the creation of the augmentations from HoloAssist seemed to be the best approach. Therefore, HoloAssist exposes a generic API that can be used to draw arbitrary augmentations: the specific instances of the augmentations are created, drawn and handled by external scripts, called HoloAssist Apps. These scripts use the HoloAssist API via the local network and are relatively easy to develop also for someone with limited programming experience: an minimal HoloAssist Apps that can highlight a specific runway at an airport can be written in just fifteen lines of Python. A diagram outlining the overall data flow is available in Figure 2.1.

2.1. Aligning the virtual 3D world with the real one

The first problem to tackle was to align HoloAssist's 3D world with the real one. When a Unity application is started on the Hololens, the device establishes a world-fixed coordinate system centered at its current real-world position. This

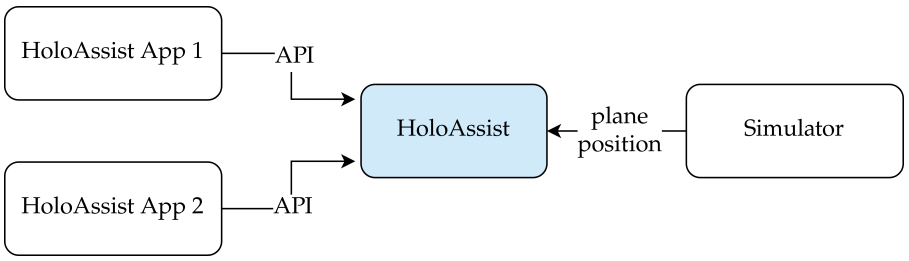


Figure 2.1.: A graphical representation of the interaction between HoloAssist, HoloAssist Apps and the flight simulator being used. All interactions happen through the local network. HoloAssist runs on the Hololens, whereas the various HoloAssist App run on a normal computer.

world-fixed coordinate system is then used by the game engine to position its own root coordinate system. Therefore, a 3D cube that is placed in Unity's root coordinate system at the point $(0, 0, 0)^T$ will appear in different real-world positions, depending on where the Hololens is in the real world when the user starts the Unity application. This is sufficient in many situations, but makes HoloAssist's use case more difficult, because correctly placing the augmentations requires aligning them with the real-world flight simulator.

The first solution attempt relied on a manual alignment performed by the user on application startup: the user would start HoloAssist and then use their hand to grab a virtual pointer that would appear in front of them. The user would then move this virtual pointer with their hand until it was aligned with the real world corner of the simulator it represented: its new pose in the virtual world would then be used to position all the other scene objects. Unfortunately, this approach did not work at all. Aligning virtual and real world items in AR is a difficult problem[23] and it led to very poor results, especially since this would have to be done every time HoloAssist was started. This solution was therefore quickly discarded.

Since the Hololens has dedicated hardware to recognize QR codes and to estimate their real world pose, it can perform this task very efficiently[1] (see Figure 2.2). It was therefore decided to attach a single QR code to the flight simulator and use this custom hardware to recognize it and locate it in the virtual world. This estimated pose was then used to update the virtual-world position of a Unity object so that it would match the real-world QR code. This object would then be used to position all the other required elements in the real world.

[1]Opposed to approaches like the one used by the Vuforia library[24], which uses the Hololens camera directly and performs the QR code recognition and pose estimation in software.

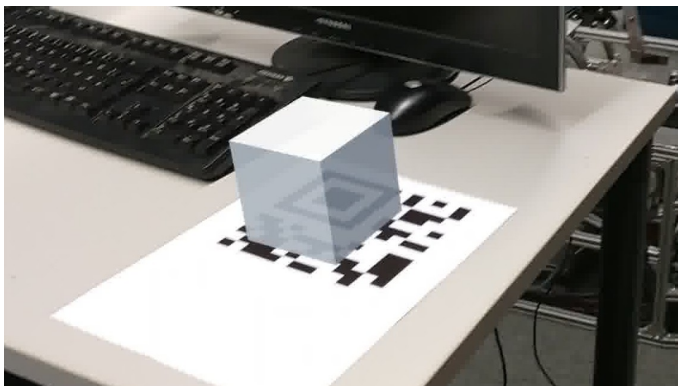


Figure 2.2.: A picture taken from the Hololens point of view of the first tests with its QR code detection hardware. This is also an example of how important the occlusion clue is for the authenticity of augmentations: although the cube is correctly placed with its center around the top-left corner of the QR code, it "looks wrong" due to the fact that the part of it below the piece of paper is not occluded by it.

The result was extremely more pleasant to use, less cumbersome, faster and more precise than the manual approach, but it was still sub-optimal: the pose estimation of the QR code is not perfect, especially as far as rotation is concerned. Due to how geo-fixed augmentation are processed (see section 2.3), they are very sensitive to errors in the rotational component of the alignment between the real and virtual world, and the precision offered by a single QR code was not sufficient. Therefore, it was decided to try to use multiple QR codes instead of a single one. This led to the discovery of the World Locking Tools library (WLT)[25], developed by Microsoft to solve a slightly different problem but that could work also for HoloAssist's use case. WLT solves the problem of providing a "frozen" Unity coordinate system, which remains fixed with respect to real world features also across application reboots and large-scale movements of the Hololens in the real world.

The Hololens API natively offers a way to pin a virtual 3D object to a real world location via the mechanism of "spatial anchors". While the application is running, it can request to the device a spatial anchor to a real-world point. After such request, the Hololens starts tracking that real-world point, continuously updating the virtual-world pose of virtual objects bound to the returned spatial anchor to correct for sensor errors. However, this only works if the spatial anchor remains close to the Hololens (less than around three meters). Moreover, corrections will be applied to each spatial anchor individually. Therefore, even though two real-world point are in a fixed position with respect to each other,

two spatial anchors bound to those point will not necessarily remain fixed relative to each other. Given these limitations, using multiple spatial anchors across large-scale AR experiences is difficult, because clusters of virtual objects bound to different spatial anchors will drift away relative to each other as the real world points referenced by the spatial anchors leave the field of view of the Hololens.

The World Locking Tools library solves this problem by introducing the concepts of "frozen", "locked" and "spongy" spaces. The default Unity coordinate system, where each individual anchor is free to move independently from all the others, is called spongy space, because reference points are continuously updated by the sensor refinements. For every frame, the Frozen World Engine offered by the library computes a rigid transformation that converts from the spongy space to the so-called locked space, a different Cartesian coordinate space which is as stable as possible, meaning that a stationary object placed in it will remain fixed with respect to real world features. However, the origin of this space is arbitrary. More formally[2], let $\mathbf{a}_i^t$ be the position in spongy space of the i-th spatial anchor at frame t and T_{SL}^t be the transformation matrix from spongy space to locked space for frame t. Then for frame $t+1$ WLT is trying to compute T_{SL}^{t+1} such that:

$$T_{SL}^{t+1} = \underset{T_{SL}^{t+1}}{\operatorname{argmin}} \sum_i \left(\frac{d_i^t + d_i^{t+1}}{2} \right)^{-1} \left\| T_{SL}^t \mathbf{a}_i^t - T_{SL}^{t+1} \mathbf{a}_i^{t+1} \right\| \tag{2.1}$$

Where $T_{SL}^t \mathbf{a}_i^t$ is the position in locked space of anchor $\mathbf{a}_i$ at frame t and $T_{SL}^{t+1} \mathbf{a}_i^{t+1}$ is the position in locked space of anchor $\mathbf{a}_i$ at frame $t+1$. The value $d_i^t \in \mathbb{R}$ is the real-world distance between the anchor $\mathbf{a}_i$ and the Hololens at frame t, and is used as a weighting factor to give more importance to the anchors that are closer to the device, as those are the ones for which the sensors have likely been able to determine a more precise position.

However, there are infinite valid locked spaces: any rigid transform applied to a locked space will yield another locked space. To transform from a locked space to the frozen space, persistent across reboots and large-scale movements, WLT introduces the concept of "space pins". A space pin is a virtual-world point that corresponds to a precise location in the real-world (e.g. the position of the top-left corner of a QR code). The application is responsible for supplying the pose of the space pins that are detectable in the real world, and then the WLT uses those to compute a transformation from the locked space to the frozen space in such a

[2]The following derivation is a plausible simplification of what is actually done by the WLT and is obtained from the library's documentation. Describing the exact implementation is impractical as it is closed-source and owned by Microsoft.

way that it best satisfies all the supplied space pins. More formally[3], let $\mathbf{p}_i^t$ be the locked space position of space pin i at frame t and $\mathbf{q}_i$ be its desired position in the virtual (frozen) world. $\mathbf{p}_i^t$ is obtainable by applying the transformation from spongy to locked space obtained for the current frame to the currently available sensor data. The Frozen World Engine is then computing a transformation T_{LF}^t from the locked space to the frozen space such that:

$$T_{LF}^t = \underset{T_{LF}^t}{\operatorname{argmin}} \sum_i \frac{1}{d_i^t} \left\| T_{LF}^t \mathbf{p}_i^t - \mathbf{q}_i \right\| \tag{2.2}$$

Where once again $d_i^t \in \mathbb{R}$ is the distance between the i-th space point and the Hololens at frame t and acts as a weight to give more importance to space pins closer to the device. The two computed transformations can then be combined into a single one that allows to convert spongy space poses to frozen world poses. Every frame, this combined transformation is cleverly applied to the Unity camera hierarchy in such a way that placing Unity objects normally in the game engine's root coordinate system results in them being automatically placed in the frozen space, hiding away all the complexity described above.

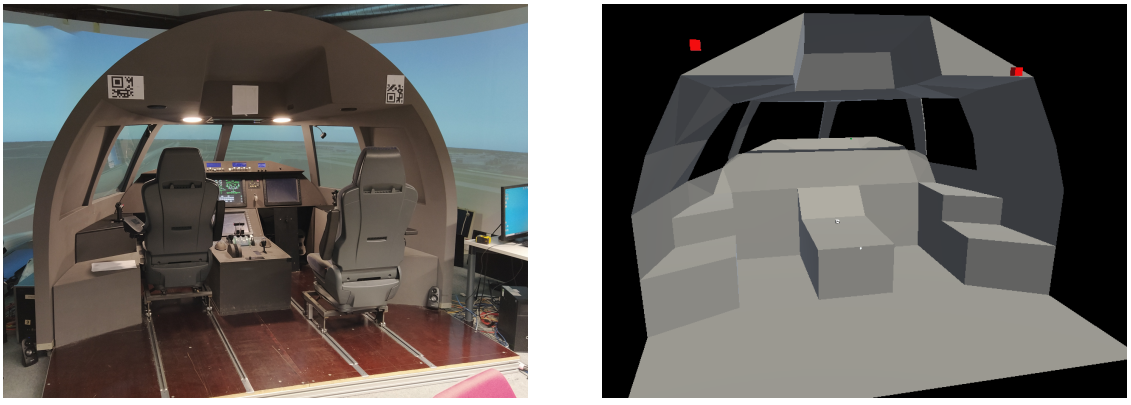
This behavior is exactly what is desired for HoloAssist: by simply placing some QR codes in the real world and using them as space pins it is possible to obtain a very precise alignment between the virtual world and the real one. Concretely, this requires measuring the real-world offset between the QR codes with a tape measure and then use Unity's `GameObjects` to replicate their structure in the virtual world as shown in Figure 2.3. HoloAssist then uses the Hololens API to detect these QR codes in the real world while the application is running and supplies their pose estimate to the WLT as space pins, matching each real world QR code with the correct `GameObject` representing the space pin in the virtual world. WLT then takes care of the rest.

Testing this solution quickly revealed that the QR codes position estimated by the Hololens is not perfect: it is reasonably precise, but changes over time, apparently within a sphere centered at the correct real-world position and with a radius of about half a centimeter. This effect is particularly noticeable if the QR code is further away than one meter from the Hololens and if the room illumination is insufficient (not necessarily dark, but simply not bright enough). The concrete result of this position detection error is that the alignment computed by WLT is sub-optimal, and easily results in visible alignment errors between real-world features and the geo-fixed augmentations shown on top of them. Since the reported position visually appeared to be randomly and uniformly distributed around the correct position, it was decided to filter the QR code

[3] Again, a plausible simplification.

position submitted to WLT by averaging the last ten QR code position estimates reported from the Hololens API. Moreover, if the distance between the Hololens and the estimated QR code position is greater than 80cm, that measurement is discarded directly and not taken into account when computing the average. This distance threshold has been choosen empirically and is determined by the fact that the precision of the QR code pose estimation produced by the Hololens decreases with distance from the QR code itself. Applying this filtering and averaging step significantly improved the final alignment and reduced the visual difference between real-world features and augmentations. Moreover, this makes the alignment procedure completely automatic, as it only requires the user to look at the QR codes for a few seconds. If for any reason the Hololens loses tracking or the alignment is broken, simply looking again at the QR codes is sufficient to re-establish.

As HoloAssist should be able to support multiple different flight simulator, the `GameObjects` that act as space pins are not hard-coded. They are generated dynamically from a JSON file in which their desired position in the virtual world is recorded. A detailed description of this JSON file is available in section 2.5. All the JSON files describing the currently supported flight simulators are packaged with the HoloAssist application and the correct one is selected depending on the first QR code that is detected.



(a) The real world QR codes

(b) The virtual space pins

Figure 2.3.: The images shows how the offset between the virtual space pins in Unity (in red) should match with the offset between the real world QR codes.

2.2. Acquiring the digital double of the simulator

Geo-fixed augmentations should only be displayed when the user is looking through the windows of the flight simulator's cockpit. Otherwise, they could end up covering important in-cockpit information, degrading the pilot experience instead of improving it. This can be achieved by acquiring a digital double of the simulator cockpit: this virtual 3D model could then be placed in the virtual "frozen" world described in section 2.1 so that the windows of the 3D model would be aligned with the windows of the real cockpit. The remaining part of the digital double would provide the required occlusion for the rest of the cabin.

A straightforward approach to build this digital double would be to use a tape measure and to build a computer aided design (CAD) model of the simulator shell with the correct dimensions. Moreover, at least for the use case of geo-fixed augmentations occlusion, the CAD model does not need to be particularly detailed: as long as the general shape is correct and the window holes are in the correct position it will satisfy the requirements. However, since this approach is traditional and well-known, it was decided to use this opportunity to experiment with more innovative technologies, in order to assess their viability and potentially discover better and/or faster workflows.

A first idea was to use the "spatial awareness" functionality offered by the Hololens API. It uses the integrated depth camera of the device to provide to the application an approximate 3D mesh of the environment surrounding the Hololens. Unfortunately, the results are not sufficient for HoloAssist's use case because:

1. The simulator's cockpit windows are made of transparent glass, which sometimes is reflective enough for the Hololens to detect it as opaque geometry.

2. The spatial awareness mesh is very imprecise, even at the highest quality setting, especially considering the small-scale details like the control inputs and buttons of the flight simulator cockpit. Although this is not strictly necessary for the geo-fixed augmentations occlusion, it will play a role in designing plane-fixed augmentations (see section 2.4).

It is also possible to save the spatial awareness mesh to a file, with the idea of using it as a starting point to then refine manually: attempting this resulted in the mesh visible in Figure 2.4. However, polishing this mesh proved to be significantly more time-intensive than expected, as it had plenty of holes and intersecting geometry. These considerations led to discard the approach based on spatial awareness.

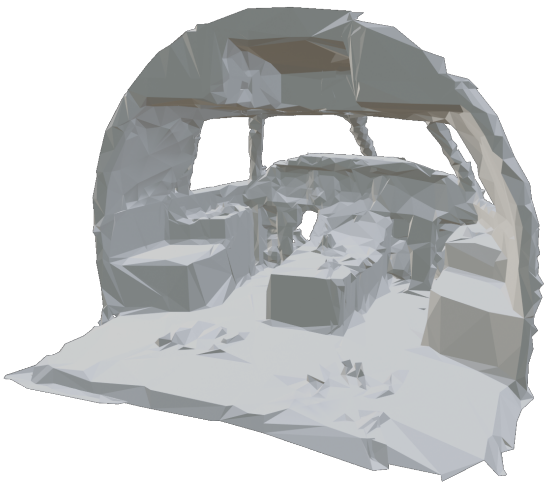


Figure 2.4.: The 3D scan of the simulator acquired via the spatial awareness feature of the Hololens.

Another idea entails applying a photogrammetry-based solution. This method relies on taking a number of partially overlapping pictures from different angles of the object that should be digitally reconstructed. The pipeline then proceeds with the following steps:

1. First of all, relevant features are extracted from each image. In this context, a "feature" is a small and highly recognizable area of the image. This step relies on common computer vision feature extractors like SIFT[26] and AKAZE[27] and yields a list of features for each image. An example is shown in Figure 2.5.

2. Then the same feature is matched across different images. As said above, this pipeline relies on the fact that the different pictures are partially overlapping: this step tries to automatically match the same real-world visual feature across two (or more) different images by comparing the feature identifier produced by the feature extraction step. This is possible because the feature extraction algorithm is able to produce similar identifiers for the same real-world feature even across different images.

3. The next steps consist of applying a "structure from motion" algorithm, which relies on the fact that the same real-world point can be seen from different pictures, and therefore from different camera poses. This allows the algorithm to use the stereoscopic information embedded in such setup to estimate the 3D position of that real-world point. More formally, assume

that the same real-world point $\mathbf{p} = \left(p_x, p_y, p_z, 1\right)^T \in \mathbb{R}^4$ (in homogeneous coordinates) has been identified and matched in n different pictures. For each of these pictures we know the coordinates of the projected position of $\mathbf{p}$ (in picture space) from the feature extraction step. Let $\mathbf{q}_i = \left(u_i, v_i, 1\right)^T \in \mathbb{R}^3$ (in homogeneous coordinates) be this projected position of $\mathbf{p}$ in the i-th picture space. Assuming a pinhole camera model[28], we can write the following relationship:

$$\mathbf{q}_i = KE_i\mathbf{p} \tag{2.3}$$

Where $K \in \mathbb{R}^{4\times3}$ is the intrinsics matrix, which depends on the camera that is used to take the various pictures, and $E_i \in \mathbb{R}^{4\times4}$ is the extrinsics matrix, which describes the real world pose of the camera when it took the i-th picture. Given that K can be determined via camera calibration[29] and $\mathbf{q}_i$ has been determined during the feature extraction step, the following optimization problem can be solved to determine $\mathbf{p}$ and the various E_i:

$$\underset{E_i,\mathbf{p}}{\operatorname{argmin}} \sum_{i}^{n} d\left(\mathbf{q}_i, KE_i\mathbf{p}\right) \tag{2.4}$$

Where $d\left(\cdot,\cdot\right)$ is the Euclidean distance between two points. Extending this problem to all features (all points $\mathbf{p}$) across all images and solving it yields the real-world position of all the extracted and matched features (alongside with the real-world pose of the camera that took each of the pictures used in the pipeline). This produces a sparse 3D reconstruction of the scene (an example is shown in Figure 2.6).

4. The next step consists of using the sparse 3D reconstruction to build a dense 3D reconstruction, in which also the real world position of the points in-between the extracted features are determined.

5. The final step entails extracting a 3D mesh from the dense point cloud obtained at the previous step. These dense point clouds are merged into a single one, which is then used to perform a Delaunay-tetrahedralization. This yields a large number of triangles that potentially describe the real surface. A complex set of heuristics[30, 31] relying on line-of-sight hints is then used to decide which of the oriented triangles is actually on the surface and which instead should be discarded.

This entire pipeline has been implemented in the open source software Meshroom[32], which makes it almost trivial to use this comparatively sophisticated approach. Around 130 pictures of the fixed-platform simulator in which HoloAssist will be used were acquired with a phone camera and processed via Meshroom's photogrammetry pipeline implementation. Some of these images are

Figure 2.5.: One of the images processed by the photogrammetry pipeline. The features extracted by SIFT in the feature extraction step are highlighted in blue, the features matched with other pictures of the dataset are highlighted in orange. The points whose 3D position has been reconstructed in the structure from motion step are shown in red.

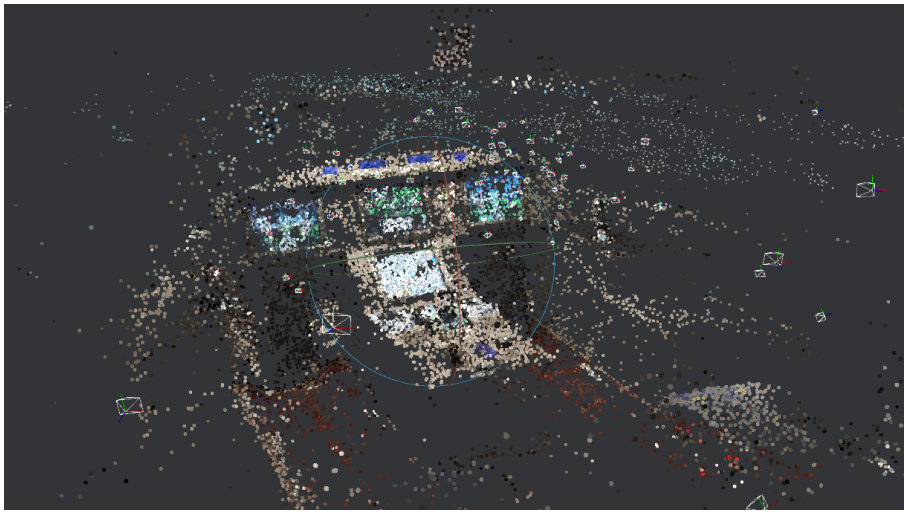


Figure 2.6.: A view of the sparse reconstruction produced by Meshroom.

shown in Figure 2.7. This yielded the final 3D-reconstructed mesh shown in Figure 2.8. The result is quite remarkable, especially giving the poor lighting condition under which the pictures were acquired and the fact that the only required hardware is a phone camera. With some manual cleanup it is definitely sufficient to obtain an occlusion mesh for the simulator cockpit's shell, and with better pictures and/or a better pipeline implementation[33] it would be a perfect solution to the problem, satisfying all of the desired use cases.

Another idea to acquire a digital double of the flight simulator comes from recent Apple iPads[34], which include a LIDAR sensor and an application that is able to use it to acquire a 3D scan of an environment. A quick attempt with this technology showed it to be superior than all of the other attempted approaches: the 3D scans it produces are precise and the texture that is generated for the 3D model allows to clearly distinguish also the positions of small-scale features like individual control knobs. Given the excellent results, it was decided to pause further investigation of photogrammetric techniques and utilize this approach instead.

Acquiring a full 3D scan of the simulator cockpit in a single step proved to be difficult: therefore, multiple different partially-overlapping 3D scans were independently acquired. However, these different meshes were not aligned: they were therefore imported in MeshLab[35] and aligned via its iterative closest point (ICP) implementation[36]. After the alignment, they were imported in Blender[37], where they were merged in a single mesh that could be used to handle the occlusion for geo-fixed augmentations. This also allowed some manual refinement, like the deletion of some unnecessary details (e.g. the pilot seats) and the automated simplification of the mesh to reduce its polygon count. This yielded the result shown in Figure 2.9. This mesh was then imported in Unity, where it was positioned in roughly the correct place with respect to the `GameObjects` acting as space pins (and therefore to what will be the QR codes positions in the real world). This rough positioning was obtained by measuring the real world offset between the QR codes and a particular point of the simulator cockpit shell with a tape measure and then replicating that offset in the Unity world between the space pins and the matching point on the simulator digital double. A further manual refinement was then performed by running the application on the Hololens and using the remote Unity editor (see section 2.6) to move the digital double until the overlap between it and the real simulator was visually satisfying. The resulting virtual position of the digital double was then integrated in the Unity project to be the default one by using the same simulator-specific JSON file in which the virtual positions of the space pins are defined. For the simulator in which HoloAssist was tested this looks like Figure 2.10.


Figure 2.7.: Two of the images used in the photogrammetry pipeline. As it can be seen, the images are relatively low-quality, as the lighting is not ideal, there are plenty of reflections and some of them had a bit of motion blur. Better images would most probably yield a better result, but this was sufficient for the test.


Figure 2.8.: The output of the Meshroom photogrammetry pipeline on the given image set. Despite the poor quality of the initial images, the result is more than acceptable.


Figure 2.9.: The final 3D scan.

```
{
    "planeMesh": [0.6909984, -1.27500093, 0.3499981],
    "spacePins": {
        // all the space-pins locations
    },
    // other simulator-related information not relevant here
}
```

Figure 2.10.: An extract of the JSON file that describes the virtual positions of the simulator digital double.

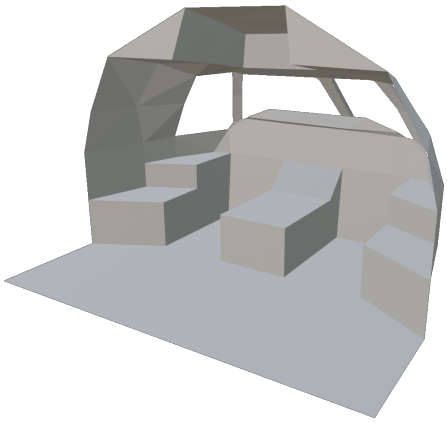


Figure 2.11.: The extremely low-polygonal mesh that is actually used as occlusion mesh on the Hololens.

In conclusion, HoloAssist is now able to place in the virtual world a digital double of the simulator cockpit's shell such that it is correctly aligned with the real one to provide the occlusion required for geo-fixed augmentations. Additionally, the acquired 3D scan is precise enough to help designing plane-fixed augmentations. Moreover, two easily accessible ways of acquiring the digital double were discovered, improving the workflow based on traditional CAD-based approach. Another advantage of these two approaches is that their complexity is irrespective of how intricate the simulator's shape is, whereas it is instead more difficult to design a CAD model for a more complex shape.

A final footnote should however be made. The mesh resulting from the 3D scan, despite the simplification done in Blender, still has a decent polygonal count (around 9000 triangles). Given the limited computational budget of the Hololens and the fact that the occlusion mesh does not need to be extremely detailed, it was decided to use the 3D scanned mesh to draw a simplified

occlusion mesh that only captures the essential details, as shown in Figure 2.11. This resulted in a significantly lighter mesh (around 200 triangles) which is still able to provide the required occlusion. Nevertheless, having the 3D scanned mesh allowed to create this simplified mesh very quickly, as it could be used as a reference without having to obtain any real-world measurement.

2.3. Geo-fixed augmentations

At this point, HoloAssist contains the foundations that are required to start working on actually displaying geo-fixed augmentation, which are augmentations that appear to be placed at a specific geographical point when looking outside the windows of the simulator cockpit.

2.3.1. Geodesy fundamentals

Understanding how geo-fixed augmentations are represented and how their rendering process works requires some familiarity with some geodesy fundamentals.

A point on the Earth can be identified precisely in different ways known as coordinate reference systems (CRSs). Different CRSs offer different tradeoffs (e.g. regarding precision vs. area of applicability) and use different parameters to represent points. The EPSG Geodetic Parameter Dataset[38] keeps track of the different CRSs and assigns an identifier to every one of them.

One of the most commonly known CRSs is EPSG:4326, which approximates the Earth surface with well-defined ellipsoid known as the WGS84 datum surface[39]. The ellipsoid's center is the Earth's center of mass and it has an equatorial radius (major semi-axis) of $a = 6378137$m and a flattening $f = 1/298.257223563$, leading to a computed semi-minor axis $b = 6356752.3142$m[40]. The resulting shape is shown in Figure 2.12. In this CRS a point on the Earth surface is identified by two numbers known as latitude and longitude: they describe, respectively, the angle between the equatorial plane and the point to be described and the angle between the IERS Reference Meridian (colloquially known as "the Greenwhich meridian") and the point to be described. A commonly used extension of this CRS is EPSG:4979, which adds a third coordinate, "height", to measure the distance between the point to be described and the ellipsoid along the ellipsoid's normal at that point. An example of a point in this CRS is shown in picture Figure 2.12.

However, EPSG:4979 does not use an Euclidean space, and this complicates some computations. Therefore, another CRS often used is EPSG:4978, which

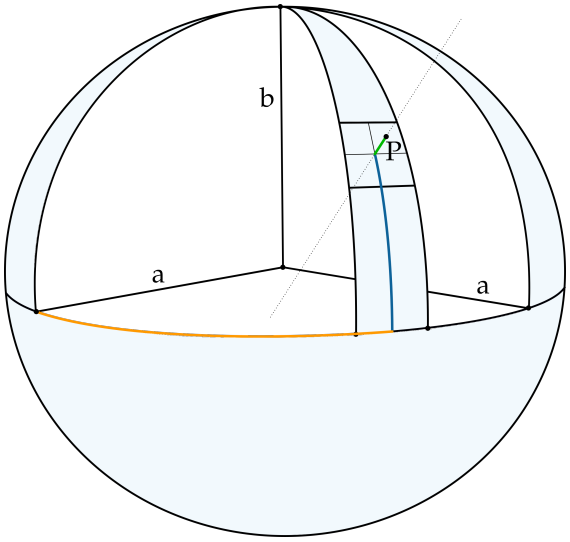


Figure 2.12.: The light-blue shape shows a depiction of the WGS84 datum surface: it is an oblate spheroid with a as the major semi-axis and b as the minor semi-axis. P represents a point in the EPSG:4979 CRS. The longitude of P is the angle implied by the orange arc, its latitude is the angle implied by the blue arc and its height is the length of the green segment. It should be noted that the height is computed along the normal to the ellipsoid surface at point P (the dotted line), which does not necessarily intersect the ellipsoid center.

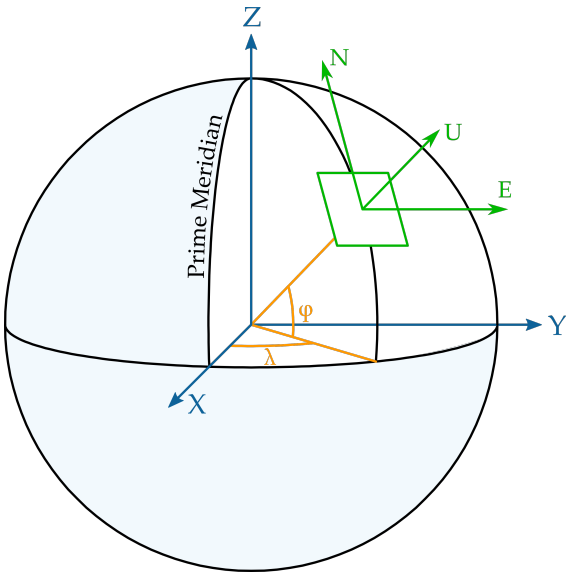


Figure 2.13.: A visual representation of a East North Up (ENU) topocentric CRS and its relationship with the EPSG:4978 CRS. Original image from Wikipedia[41].

instead *is* an Euclidean space. It is an Earth-centered, Earth-fixed (ECEF) coordinate system whose origin is at the center of the WGS84 ellipsoid. In this CRS, a point is represented by a triplet of numbers denoting the distance from the origin in meters along each of the three axes, respectively known as X axis, Y axis and Z axis. The X axis is on the equatorial plane and intersects the IERS Reference Meridian, while the Z-axis is parallel to the mean earth rotation axis. As EPSG:4978 is the only ECEF CRS used in this document, the terms ECEF and EPSG:4978 will be used interchangeably.

The last type of CRS that needs to be discussed is actually a class of CRSs known as topocentric coordinate reference systems. An example of such system is EPSG:5819. These CRSs define a plane that is tangent to the Earth's ellipsoidal surface at a specific location called "topocentric origin". This location is then used to define a Cartesian frame centered at it, with two axes lying on the tangent plane and the third one parallel to the normal of the plane. A visual depiction of this kind of CRS is shown in Figure 2.13. HoloAssist uses a type of topocentric CRSs known as ENU, which is a right-handed coordinate system in which one axis points towards the north pole, one points "away" from the ellipsoid surface (along the plane normal) and the third one is perpendicular to these two, which means it points "east". An ENU CRS offers two main advantages with respect to an ECEF CRS: it is more intuitive to use in the context of the position of an aircraft and its coordinates are smaller in absolute value, leading to fewer numerical precision issues during computations. However, it is a flat approximation of the Earth surface around a point, and is therefore accurate only in proximity of the topocentric origin.

HoloAssist also needs to be able to take points expressed in one CRS and convert them to another. Fortunately, conversion procedures between common EPSG CRS are well-known and often already implemented in geodesy libraries (like PyProj[42]) and tools (like QGis[43]). In the interest of producing a self-contained document, the conversion procedures between the CRSs used by HoloAssist have been sourced from the literature[40] and reported here without proof.

First of all, HoloAssist needs to be able to convert between EPSG:4979 and EPSG:4978. Given the parameters of the WGS84 datum:

$$a = 6378137m \tag{2.5}$$

$$b = 6356752.3142m \tag{2.6}$$

$$f = \frac{a-b}{a} \tag{2.7}$$

$$e = \sqrt{f(2-f)} \tag{2.8}$$

The radius of curvature in the prime vertical of the WGS84 ellipsoid at a particular latitude φ can be computed as follows:

$$N(\varphi) = \frac{a}{\sqrt{1 - e^2 \sin^2 \varphi}} \tag{2.9}$$

Given a point $\mathbf{p}_{\text{EPSG:4979}} = (\text{latitude}, \text{longitude}, \text{altitude})^T = (\varphi_p, \lambda_p, h_p)^T$ in the EPSG:4979 CRS, it can be converted to ECEF with the following function:

$$\mathbf{p}_{\text{ECEF}} = \text{convert}_{\text{EPSG:4979} \to \text{ECEF}}(\mathbf{p}_{\text{EPSG:4979}}) \tag{2.10}$$

$$= \begin{pmatrix} (N + h_p) \cos \varphi_p \cos \lambda_p \\ (N + h_p) \cos \varphi_p \sin \lambda_p \\ ((1 - e^2) N + h_p) \sin \varphi_p \end{pmatrix} \tag{2.11}$$

The other conversion that HoloAssist needs to be able to do is from EPSG:4978 to a topocentric ENU CRS and vice versa. Let $\mathbf{o} = (\varphi_o, \lambda_o, h_o)$ be an EPSG:4979 point describing the topocentric origin of the destination ENU system and $\mathbf{p}_{\text{ECEF}}$ be the EPSG:4978 point that we want to convert. Moreover, let:

$$\mathbf{o}_{\text{ECEF}} = \text{convert}_{\text{EPSG:4979} \to \text{ECEF}}(\mathbf{o}) \tag{2.12}$$

Then the desired conversion is:

$$\mathbf{p}_{\text{ENU}} = \text{convert}_{\text{ECEF} \to \text{ENU}}(\mathbf{p}_{\text{ECEF}}, \mathbf{o}) \tag{2.13}$$

$$= \begin{pmatrix} -\sin \lambda_o & \cos \lambda_o & 0 \\ -\sin \varphi_o \cos \lambda_o & -\sin \varphi_o \sin \lambda_o & \cos \varphi_o \\ \cos \varphi_o \cos \lambda_o & \cos \varphi_o \sin \lambda_o & \sin \varphi_o \end{pmatrix} (\mathbf{p}_{\text{ECEF}} - \mathbf{o}_{\text{ECEF}}) \tag{2.14}$$

This conversion can be straightforwardly inverted to obtain the function $\text{convert}_{\text{ENU} \to \text{ECEF}}(\mathbf{p}_{\text{ENU}}, \mathbf{o})$, which converts a point from an ENU system with topographic origin $\mathbf{o}$ to the ECEF CRS:

$$\mathbf{p}_{\text{ECEF}} = \text{convert}_{\text{ENU} \to \text{ECEF}}(\mathbf{p}_{\text{ENU}}, \mathbf{o}) \tag{2.15}$$

$$= \mathbf{o}_{\text{ECEF}} + \begin{pmatrix} -\sin \lambda_o & \cos \lambda_o & 0 \\ -\sin \varphi_o \cos \lambda_o & -\sin \varphi_o \sin \lambda_o & \cos \varphi_o \\ \cos \varphi_o \cos \lambda_o & \cos \varphi_o \sin \lambda_o & \sin \varphi_o \end{pmatrix}^T \mathbf{p}_{\text{ENU}} \tag{2.16}$$

2.3.2. Geo-fixed augmentations representation

A good starting point is to determine the best way to represent this kind of augmentations: regardless of how HoloAssist will handle them internally, it is desirable to expose to the API's users a straightforward and easy to use representation which at the same time maintains as much flexibility as possible.

Usually, computer graphics applications represent 3D meshes as a set of triangles, encoded with two lists:

1. A list of vertices, containing the 3D coordinates of the vertices of all the triangles;

2. And a list of indices, in which every elements points to an item of the list of vertices. Starting from the beginning of this list, each triplet of elements describes a triangle of the mesh.

This representation is preferred because the same vertex is usually part of multiple triangles: therefore, storing its coordinates only once allows to save memory. Since this representation is so ubiquitous and flexible, it was used as a starting point to design HoloAssist's interface.

HoloAssist's API allows users to create, edit and delete entities called geo-fixed external line meshes (GF-ELMs)[4], which analogously to common 3D meshes consist mainly of two lists: one containing vertices and one containing indices. The main difference is how a vertex is described: normal 3D meshes use a triplet of numbers describing the vertex position in an Euclidean space (usually called "model space") whereas GF-ELM vertices are described by a more complex structure called `GeoFixedVertex`. It consists of the following elements:

1. An EPSG:4979 point describing the topocentric origin of the ENU system in which this vertex is positioned.

2. A triplet of number describing the local translation of the vertex with respect to the ENU's topocentric origin. Although these numbers describe the "east", "north" and "up" components of the local translation, the API uses (respectively) the names "x", "z" and "y", as they match with the respective Unity's axes. The details of this relationship are described below, but this naming makes the API interface more intuitive to reason about when actually trying to design a geo-fixed augmentation, because the user does not need to mentally map between Unity's axes and the ENU axes and can focus on Unity's naming.

3. A triplet of number containing the intrinsic Euler angles that describe the local rotation of the whole ENU space around it's origin. Even though not strictly necessary, this field is useful because rotations are often cumbersome to implement, and allowing the API users to just specify the desired

[4] In computer science there are two difficult problems: cache invalidation and naming things. And off by one errors[44].

rotation without having to apply it significantly simplifies some HoloAssist Apps. Although these numbers describe a rotation around the "east", "north" and "up" axes, the API uses (respectively) the names "x", "z" and "y", for the same reason as the previous point.

4. A triplet of number describing the color of the vertex as a combination of red, green and blue components.

This vertex representation allows to naturally cover a number of different use cases that were identified, such as:

1. Meshes that are entirely composed of geo-referenced vertices. An example of this would be a mesh with four vertices representing the corners of an airport runway, where the geographical coordinates of such vertices have been obtained from a map. In this case, both the local translation and the local rotation of the `GeoFixedVertices` would be the zero vector.

2. "Common" 3D meshes (e.g. a 3D model created in Blender) positioned somewhere on the Earth. For example, the 3D model of a map pin could be placed above a geographical feature to highlight it. In this case, all the vertices of the mesh would share the same topocentric origin (that is, the position of the geographical feature to be highlighted), but the local translation of each vertex would be equal to the vertex position in the "normal" model space in which the mesh is created. The local rotation would once again be the zero vector or could be used to easily rotate the whole mesh (by assigning to every `GeoFixedVertex` the same local rotation).

3. Tunnel-like meshes, which are commonly used to show approach trajectories. An example of such mesh is shown in Figure 2.14. In such meshes there are multiple cross-sectional slices of the tunnel whose corners are connected by straight lines. Each of these cross-sectional slices usually needs to be placed at a different geographical position and to be oriented in a different way, but their corners need to be connected to other such slices. In this case, vertices of the same cross-sectional slice would share the same topographic origin and local rotation, whereas the local translation would allow them to be positioned around the topographic origin to form the desired shape (e.g. a square). The indices list then naturally allows to describe connections across different cross-sectional slices, as the vertices forming them are still part of the same mesh.

4. Any combination of the above.

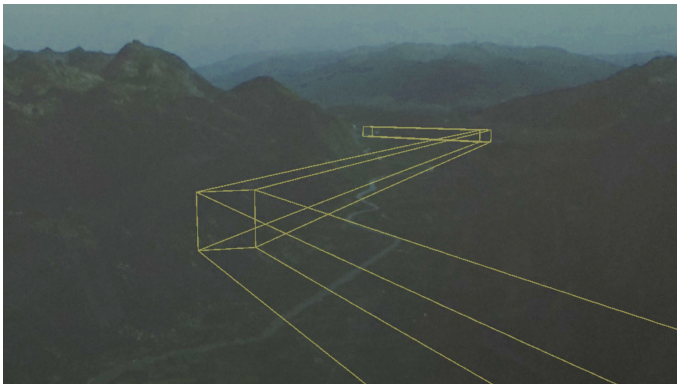


Figure 2.14.: A visual example of a tunnel mesh. Each cross-sectional slice (vertical square) is placed at different geographical point with different orientations and is then connected to the others with straight segments.

The details of HoloAssist's API for geo-fixed augmentations are described in Appendix A, but in summary it is designed to offer a straightforward way of manipulating the vertex and index list for each GF-ELM.

2.3.3. Rendering geo-fixed external line meshes

The next step entails finding a way to convert such representation to something that can be understood by Unity, as computer graphics usually assumes a single, Euclidean space in which the various elements of the 3D scene are placed. As suggested in [40], this result can be achieved by converting the vertices of all the GF-ELMes to a common ENU coordinate reference system that is centered on the current geographical position of the aircraft, as computed by the flight simulator. Such ENU CRS is (rather unimaginatively) called the "airplane ENU CRS" in this document. This single CRS is then mapped to the left-handed coordinate system used by Unity in such a way that it matches with the simulator's digital double described in section 2.2. This allows to treat all the GF-ELMes as normal Unity meshes which are then rendered by the standard game engine rendering pipeline. A diagram summarizing the CRSs involved in these conversions is available in Figure 2.15. Implementing this algorithm requires:

1. Being able to obtain the airplane's current geographical position (and orientation) as computed by the flight simulator.

2. Being able to convert a `GeoFixedVertex` to the airplane ENU CRS.

3. Being able to convert a point from the airplane ENU CRS to the Unity

"frozen" coordinate system in which the simulator digital double from section 2.2 is placed.

Acquiring the airplane's current geographical position and orientation from the flight simulator is straightforward, as most flight simulators have a way of broadcasting this information via user datagram protocol (UDP) packets. HoloAssist is therefore capable of receiving this kind of information by listening on a UDP port where it expects binary UDP packets with the structure defined in Table 2.1. In this document, such packets are called "simulator status update" packets. Flight simulators can directly send simulator status update packets with that structure to HoloAssist or an intermediate script can be written to translate the format used by the simulator to the one used by HoloAssist.

Once the current geographical position of the aircraft is available to HoloAssist, it becomes possible to compute the position of each `GeoFixedVertex` in the airplane ENU CRS. This happens in two phases. The first phase entails converting each `GeoFixedVertex` to the ECEF CRS: this conversion is perfomed once when the vertex is created via the API. Denoting with the EPSG:4979 point $\mathbf{o}$ the topocentric origin of the `GeoFixedVertex`, with $\mathbf{t} \in \mathbb{R}^3$ its local translation and with $R_l \in \mathbb{R}^{3\times3}$ the rotation matrix describing its local rotation, the ECEF coordinates are computed as follows:

$$\mathbf{v}_{\text{ECEF}} = \text{convert}_{\text{ENU}\to\text{ECEF}}\left(R_l\mathbf{t}, \mathbf{o}\right) \tag{2.17}$$

Where $\text{convert}_{\text{ENU}\to\text{ECEF}}\left(\cdot,\cdot\right)$ is the function defined in Equation 2.15. The second phase happens once per rendered frame and entails converting each $\mathbf{v}_{\text{ECEF}}$ to the airplane ENU CRS:

$$\mathbf{v}_{\text{AirplaneENU}} = \text{convert}_{\text{ECEF}\to\text{ENU}}\left(\mathbf{v}_{\text{ECEF}}, \mathbf{q}\right) \tag{2.18}$$

Where $\mathbf{q}$ is the current airplane position in EPSG:4979 as received from the simulator and $\text{convert}_{\text{ECEF}\to\text{ENU}}\left(\cdot,\cdot\right)$ is defined in Equation 2.13. Technically, this conversion would only need to be performed once per simulator status update. However, since the simulator in which HoloAssist has been tested broadcasts updates frequently enough to be comparable with the Hololens' framerate, it was decided to recompute it once per frame. Moreover, this keeps the workload consistent across frames, improving hologram's stability[45].

Now that all the vertices of the GF-ELMes are expressed in a single Euclidean space (the airplane ENU CRS), the last step is to map it to the same Unity's frozen coordinate space in which the simulator digital double is positioned as described in section 2.2. In order to simplify this step, some additional requirements on the coordinates that are chosen for the space pins described in section 2.1 are

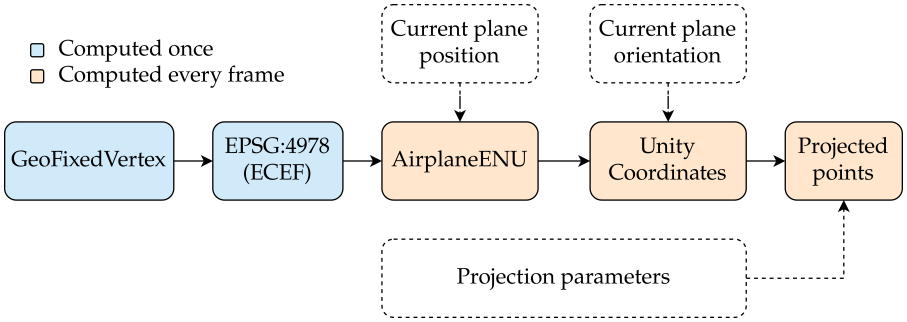


Figure 2.15.: An overview of the transformations required to display a `GeoFixedVertex` at the correct position. Each of these conversions is explained in details in the main text. The last transformation is explained in subsection 2.3.4.

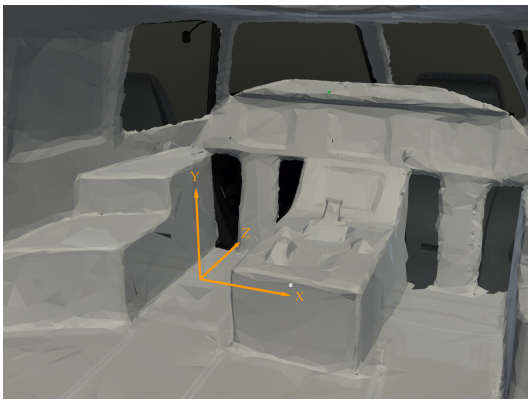
Offset	Length	Type	Description
0	1	-	Fixed value (00000000)
1	8	double	Airplane current latitude in radians
9	8	double	Airplane current longitude in radians
17	8	double	Airplane current altitude in meters
25	8	double	Airplane current roll angle in radians
33	8	double	Airplane current pitch angle in radians
41	8	double	Airplane current yaw angle in radians

Table 2.1.: Describes the binary structure of a simulator status update UDP packet. Field offset and length are in bytes. The field type "double" is a shorthand to indicate a IEEE 754 double-precision number encoded in little endian. The first byte of the packet is used to differentiate a simulator status update packet from other types of UDP packets received on the same UDP port.

ENU axis	Unity axis
Positive North	Positive Z
Positive Up	Positive Y
Positive East	Positive X

Table 2.2.: Mapping between Unity's coordinate system and the airplane ENU CRS when the airplane has zero yaw, pitch and roll.

(a) Airplane ENU CRS's East, North, Up axes



(b) Unity's X, Y, Z axes

Figure 2.16.: An image taken from the Hololens while running HoloAssist that shows the alignment between the real simulator, the Unity coordinate system and the airplane ENU CRS. When the airplane has zero yaw, pitch and roll, the simulator's longitudinal axis is aligned with Unity's Z axis and the airplane ENU CRS N axis. The same happens for the simulator's vertical axis, Unity's X axis and the airplane ENU CRS U axis.

necessary. As of now, the concrete values of the coordinates of the space pins have been irrelevant, as long as the relative offset between them is consistent with the real-world offset between the QR codes. However, in order to easily map the airplane ENU CRS to Unity's coordinate system, those coordinates need to be picked in such a way that the simulator digital double's longitudinal axis is aligned with Unity's Z axis and that the digital double's vertical axis is aligned with Unity's Y axis.

When this additional requirement is added we can observe that, if the plane geographical orientation computed by the simulator is zero on all three axes (yaw, pitch, roll), then Unity's Z axis (and therefore the digital double's longitudinal axis) is aligned with the airplane ENU CRS north axis, as an airplane with a yaw of zero degrees is pointing north. Moreover, due to the roll and pitch of zero degrees, the airplane ENU CRS up axis is aligned with Unity's Y axis (and therefore with the digital double's vertical axis). This means that, if the plane has no geographical rotation and all the requirements are uphold, the airplane ENU CRS space can be mapped to Unity's coordinates with the mapping described in Table 2.2 and the result will visually match in the real world with what can be seen from the simulator cockpit's windows. An example of this axes alignment is shown in Figure 2.16.

However, this simple mapping is not sufficient. The aircraft position computed

```
{
    "planeCenter": [0,2.1336, -10.0584],
    "spacePins": {
        // all the space-pins locations
    },
    // other simulator-related information not relevant here
}
```

Figure 2.17.: An extract of the JSON file that describes the virtual positions of the plane center of mass.

by the flight simulator is usually the geographical position of the aircraft's center of mass. In order to obtain correctly placed geo-fixed augmentations we need to take into account the offset between this point and the simulator digital double. The offset between the airplane center of mass and the simulator cockpit is usually available in the simulator's documentation. HoloAssist expects that the position of the aircraft's center of mass in the Unity frozen world (see section 2.1) is supplied as a simulator-specific value in the same JSON file in which the virtual locations for the various space pins are recorded. For the simulator in which HoloAssist was tested, this looks like Figure 2.17.

The last piece of the puzzle regards the airplane geographical orientation, which is supplied by the simulator as part of the simulator status update. Since HoloAssist is designed for fixed-platform simulators, when the geographical orientation of the aircraft changes, the simulator cockpit doesn't actually move in the real world: the only thing that changes is the imagery that is visible through the cockpit windows. Therefore, the virtual position of the simulator digital double cannot be changed, as it needs to remain aligned with the real cockpit: thus, when the airplane changes geographical orientation, instead of rotating the digital double, the inverse of such rotation is applied when matching the airplane ENU CRS with the Unity's coordinate system.

Combining all these considerations allows for a mapping from the airplane ENU CRS to the Unity's coordinate system that ensures visually correct geo-fixed augmentations. More formally, let $\mathbf{b} \in \mathbb{R}^3$ be the position of the airplane center of mass in Unity's coordinate system, let $R_g \in \mathbb{R}^{3\times3}$ be the rotation matrix[5] describing the current airplane rotation as obtained from the simulator and let

[5] Although the descriptions in this document represent rotations with 3×3 matrices, internally HoloAssist uses quaternions[46], as they are the standard implementation for handling rotations in game engines. However, quaternion details tend to be less known, and therefore the more common matrix representation has been chosen.

$\mathbf{v}_{\text{AirplaneENU}} \in \mathbb{R}^3$ be the position of a `GeoFixedVertex` converted to the airplane ENU CRS for the current frame. Then that point's position in the Unity space is computed as follows:

$$\mathbf{v}_{\text{Unity}} = \mathbf{b} + R_g^{-1} M \mathbf{v}_{\text{AirplaneENU}} \tag{2.19}$$

With M encoding the mapping described in Table 2.2:

$$M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \tag{2.20}$$

The airplane's geographical orientation coming from the simulator status updates is assumed to be in a right-handed coordinate system with the "up" axis pointing downwards: when computing R_g, appropriate care should be taken to convert the rotation to the left-handed coordinate system that Unity uses and to also account for the different orientation of the "up" axis.

At this point, the process to convert from GF-ELM vertices to positions that are understood by Unity is complete: these converted positions are then handed to the game engine as normal 3D meshes (instances of the `CoreModule.Mesh` class), which are rendered by the standard rendering pipeline, yielding a visual result which shows the virtual augmentation at the desired geographical position when looking outside the simulator windows in the real world.

2.3.4. Accounting for projection error

Although the steps described in the previous sections are correct for a real aircraft, in a fixed-platform flight simulator they will yield a result that is visually wrong, as shown in Figure 2.18. This is due to how fixed-platform simulators usually work. As shown in Figure 2.19, a fixed-platform simulator is usually composed of a cockpit, which resembles the real plane and where the pilot sits, and a (usually cylindrical) screen placed in front of it. The simulator then uses some projectors to show on this screen what the pilot would see from the cockpit's window at the current geographical position of the plane. In order to do this, the simulator engine has to project geographical features that in the real world would be placed kilometers away from the airplane to the cylindrical screen a few meters in front of the cockpit: this kind of projection can be computed correctly only for a single point, known as the projection eye point (PEP), which is usually placed inside the cockpit, near the head of the pilot. A viewer placed in any other position than the PEP will see some projection error in what is shown on the outside screen.

Figure 2.18.: An image taken from the Hololens while running HoloAssist that shows a geo-fixed augmentation for the runway of an airport. Although the computations are correct, the result does not visually match the image produced by the simulator.

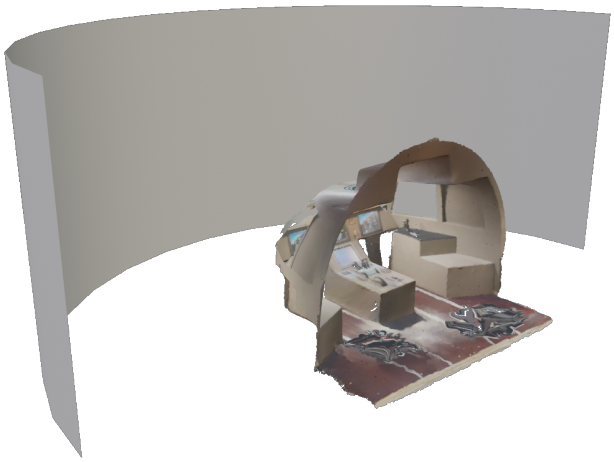
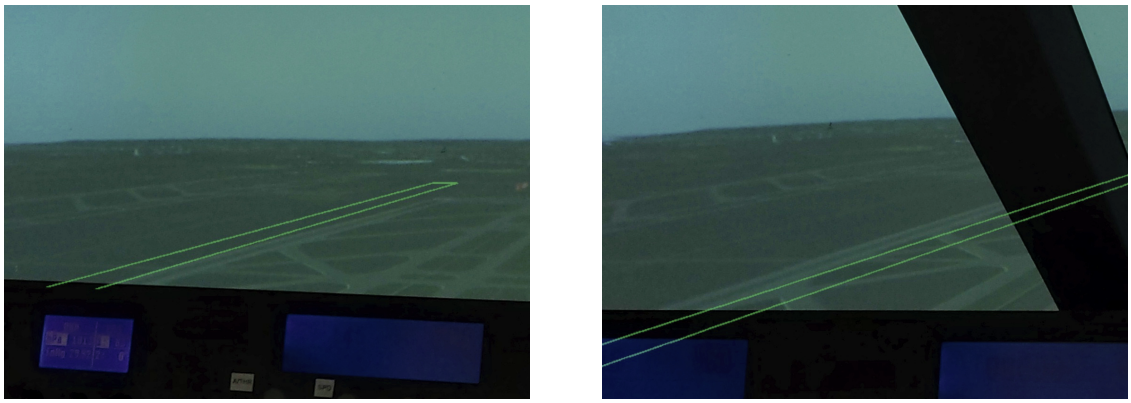


Figure 2.19.: An overview of the structure of a fixed-platform flight simulator. In the center there is a replica of the cockpit of the airplane being simulated, and in front of it there is a (often cylindrical) screen on which the imagery generated by the simulator is projected.



(a) Left of the PEP

(b) Right of the PEP

Figure 2.20.: Images taken from the Hololens of a geo-fixed augmentation being shown without the correction for the projection error. Each image shows the visual output at a different position with respect to the PEP.

This is usually not a huge issue, as the projected imagery still looks real and precise, but becomes relevant when we want to add some augmentations on top of real world features. The computations described in the previous sections do not account for this projection error, and therefore, despite being correct, they appear visually wrong if the Hololens is positioned in any point other than the PEP, as shown in Figure 2.20. In order to obtain visually correct results in every place inside the simulator we need to apply the same projection used by the flight simulator to the $\mathbf{v}_{\text{Unity}}$ coordinates computed in subsection 2.3.3.

In order to do this, a few additional simulator-specific parameters are needed. They are:

1. The shape of the projection screen. In the simulator in which HoloAssist is tested the projector screen is assumed to be a cylinder with its axis parallel to the airplane's vertical axis.

2. The position of the center of the cylinder in the virtual world. Let this point in the virtual Unity frozen space be called $\mathbf{c} \in \mathbb{R}^3$.

3. The radius of the cylinder, which will be called $r \in \mathbb{R}$.

4. The position of the PEP in the virtual world. Let this point in the virtual Unity frozen space be called $\mathbf{e} \in \mathbb{R}^3$.

These values can usually be derived from the simulator documentation. However, as the simulator in which HoloAssist is tested is a non-commercial research

```
{
    "projection": {
        "type": "CYLINDER",
        "cylinderCenter": [0.960499, -1.262993, 0.286 ],
        "cylinderRadius": 4.012081,
        "eyePoint": [0.989999, -0.552001, 0.686999]
    },
    "spacePins": {
        // all the space-pins locations
    },
    // other simulator-related information not relevant here
}
```

Figure 2.21.: An extract of the JSON file that describes the parameters needed to account for the projection error.

simulator, these value had to be obtained in another way, described in subsection 2.3.6. Once known, these values are stored in the same simulator-specific JSON file in which the virtual position of the space pin are stored. An example is shown in Figure 2.21.

Given the above-mentioned parameters $\mathbf{c}$, r and $\mathbf{e}$ and a point $\mathbf{v}_{\text{Unity}}$ as computed above, the projection $\mathbf{v}_{\text{proj}}$ of such point to the cylinder surface can be computed similarly to how it is done in [21]. In the cylindrical screen case, the geometrical setup of the problem is shown in Figure 2.22, in which the unknown is $\mathbf{v}_{\text{proj}}$. The direction of the line connecting $\mathbf{v}_{\text{Unity}}$ and the PEP can be computed as follows:

$$\mathbf{n} = \frac{\mathbf{v}_{\text{Unity}} - \mathbf{e}}{\|\mathbf{v}_{\text{Unity}} - \mathbf{e}\|} \tag{2.21}$$

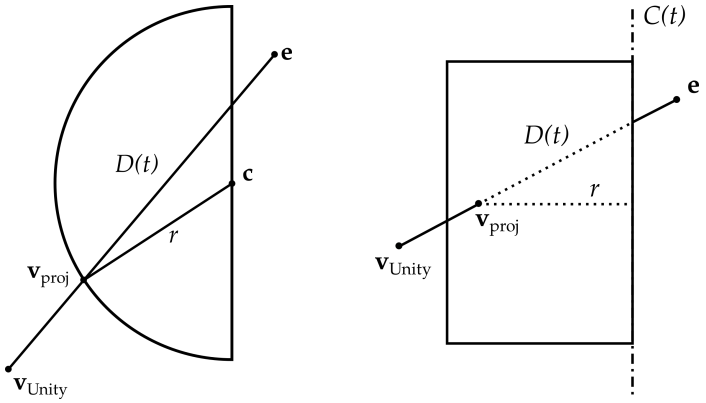
Therefore, the line connecting them is:

$$D(t) = \mathbf{e} + t\mathbf{n} \tag{2.22}$$

Similarly, the axis of the cylinder can be written as:

$$C(t) = \mathbf{c} + t\hat{\mathbf{y}} \tag{2.23}$$

Where $\hat{\mathbf{y}}$ is the unit-length vector denoting the Unity Y axis, which is parallel to the simulator digital double vertical axis and to the cylinder axis. In order to find $\mathbf{v}_{\text{proj}}$ we need to find the value $d \in \mathbb{R}$ for which the line $D(d)$ intersects the



(a) Top view (b) Side view

Figure 2.22.: The top and side views of the geometrical setup for the projection of a point in the airplane ENU CRS space to the cylindrical screen.

cylinder, that is, the value of d for which the point $D(d)$ is at distance r from the line $C(t)$:

$$\text{distance}\left(C(t), D(d)\right) = r \tag{2.24}$$

Given a line $\mathbf{a} + x\mathbf{b}$, its distance from a generic point $\mathbf{p}$ is given by[47]:

$$\text{distance}\left(\mathbf{a} + x\mathbf{b}, \mathbf{p}\right) = \|(\mathbf{p} - \mathbf{a}) - ((\mathbf{p} - \mathbf{a}) \cdot \mathbf{b}) \cdot \mathbf{b}\| \tag{2.25}$$

Equation 2.24 can be rewritten as follows:

$$\begin{aligned}
\text{distance}\left(C(t), D(d)\right) &= r && (2.26)\\
\text{distance}\left(\mathbf{c} + t\hat{\mathbf{y}}, \mathbf{e} + d\mathbf{n}\right) &= r && (2.27)\\
\|(\mathbf{e} + d\mathbf{n} - \mathbf{c}) - ((\mathbf{e} + d\mathbf{n} - \mathbf{c}) \cdot \hat{\mathbf{y}}) \cdot \hat{\mathbf{y}}\| &= r && (2.28)\\
\dots && (2.29)\\
\left\| \begin{pmatrix} e_1 + dn_1 - c_1 \\ 0 \\ e_3 + dn_3 - c_3 \end{pmatrix} \right\| &= r && (2.30)\\
(e_1 + dn_1 - c_1)^2 + (e_3 + dn_3 - c_3)^2 &= r^2 && (2.31)
\end{aligned}$$

Where the x_i syntax denotes the i-th element of vector $\mathbf{x}$. The last step of the derivation is a second degree equation in d that can be solved with standard techniques, yielding two solutions d_1 and d_2, one for the "entry point" of the line in the cylinder and one for the "exit point". Assuming the direction from

(a) Left of the PEP

(b) Right of the PEP

Figure 2.23.: Images taken from the Hololens after applying the correction for the projection error. The geo-fixed augmentation now appears visually correct from every head position.

$\mathbf{v}_{\text{Unity}}$ to $\mathbf{e}$, the desired point is always the entry point, which given this setup is the closest to $\mathbf{v}_{\text{Unity}}$. Therefore, the desired d is:

$$d = \min\left(\|D(d_1) - \mathbf{v}_{\text{Unity}}\|, \|D(d_2) - \mathbf{v}_{\text{Unity}}\|\right) \tag{2.32}$$

Thus, the actual final output of the conversion of a `GeoFixedVertex` to Unity coordinates is:

$$\mathbf{v}_{\text{proj}} = D(d) \tag{2.33}$$

After applying this step, geo-fixed augmentation appear visually correct from all points of the simulator, as shown in Figure 2.23.

However, applying the projection in this way introduces a limitation to the kind of meshes that the current version of HoloAssist can show. Flattening all vertices to the same cylindrical surface as the projection does completely removes all the depth information that would normally be associated with the mesh's triangles and that would be used to distinguish which triangle should be drawn in front of the others during the normal rendering pipeline (in a step called depth test[48]). Although this problem should be solvable by manually reimplemening a mechanism analogous to the depth test applied to the cylinder surface, it was decided to not deal with this in the current version of HoloAssist. It was instead decided to (more simply) limit the kind of mesh that could be shown: rather than allowing meshes with triangular topology, only meshes with linear topology (i.e. composed by lines) are permitted. This is the reason for which GF-ELMes have `Line` in their name. Although this limits the kind of

Figure 2.24.: An image taken from the Hololens that shows a geo-fixed augmentation highlighting the runway of an airport.

AR augmentations that can be shown, it was deemed an acceptable trade-off because:

1. The currently planned geo-fixed augmentations would have been line-based anyway;
2. It was preferred to obtain something working earlier that could then be expanded in the future rather than delaying deployment until all features could be implemented[49].

2.3.5. Automatic interpolation

Although the solution to the projection error shown in subsection 2.3.4 works, it sometimes leads to an unexpected behavior. Take for example a geo-fixed augmentation highlighting a landing strip of an airport, as shown in Figure 2.24. If the airplane flies directly over the landing strip at a low altitude, the unexpected outcome shown in Figure 2.25 happens. This happens because the geo-fixed augmentation only specifies four vertices for its GF-ELM, one for each corner of the landing strip: these four vertices are then projected to the cylinder (correctly, as shown by the orange circle in Figure 2.25), but then are connected by straight lines, which therefore "cut through" the projection cylinder instead of wrapping around it as expected.

This problem can be solved by adding additional vertices to the mesh in intermediate positions between the original four vertices: in this way, each of

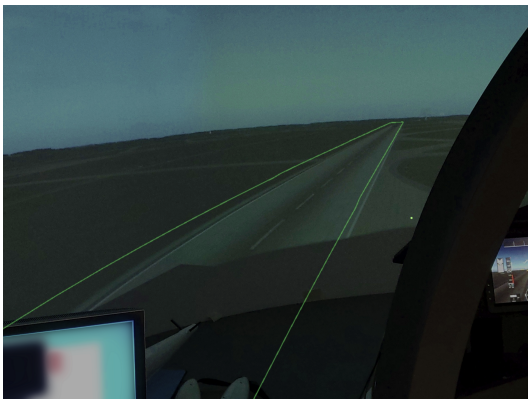
Figure 2.25.: An image taken from the Hololens that shows a geo-fixed augmentation highlighting the runway of an airport and the error induced by the combination of the limited number of vertices of the mesh and the projection applied by HoloAssist. The vertices submitted through the API (two are in the orange circle) are placed correctly, but they are connected with straight lines that do not follow the cylindrical projection screen.

the intermediate points will be projected to the cylinder and the error due to the fact that they are connected by straight lines will be less evident, as shown in Figure 2.26.

However, forcing the user to deal with this problem (which effectively is an implementation detail of HoloAssist) for every geo-fixed augmentation that is created goes against the basic principle of making it easy for HoloAssist users to create AR experiences. Therefore, HoloAssist performs this interpolation automatically by default when a GF-ELM is created through its API. Nevertheless, users working on more advanced use cases are still free to specify a maximum real-world distance between vertices after the interpolation and, if preferred, to opt-out of this automatic procedure entirely.

In order to implement this, the interpolation routine assumes a mesh with line topology, as the current version of HoloAssist only supports that kind of topology (see subsection 2.3.4). This means that, for every GF-ELM, the list of indices can be seen as a list of pairs of numbers, with each pair denoting a line segment that could potentially have to be interpolated. Therefore, after converting the vertices to the ECEF CRS as described in Equation 2.17, HoloAssist goes through each of these line segments and, if needed, adds intermediate vertices via linear interpolation. More concretely, for each pair of indices that denotes a segment,

(a) Coarse automatic interpolation



(b) Fine automatic interpolation

Figure 2.26.: An image taken from the Hololens that shows how adding intermediate vertices allows the projection to behave as expected. As the coarse interpolation shows, the additional added vertices (shown in the orange circle) are correctly projected on the cylindrical screen, but there still are not enough vertices to provide a sufficiently smooth result.

let $\mathbf{v}_{\text{ECEF}}$ and $\mathbf{u}_{\text{ECEF}}$ be the coordinates of the vertices forming that segment in the ECEF CRS. Since ECEF is an Euclidean space, the length of this segment can be computed with the Euclidean distance of these two points. If this value is above a certain threshold $t \in \mathbb{R}$ chosen by the user:

$$\|\mathbf{v}_{\text{ECEF}} - \mathbf{u}_{\text{ECEF}}\| \geq t \tag{2.34}$$

Then the segment is divided in n equal sized pieces with:

$$n = \left\lceil \frac{\|\mathbf{v}_{\text{ECEF}} - \mathbf{u}_{\text{ECEF}}\|}{t} \right\rceil \tag{2.35}$$

Each of the new intermediate point $\mathbf{w}_i$ is computed by linear interpolation:

$$\mathbf{w}_i = \mathbf{v}_{\text{ECEF}} + \left(\frac{1}{n} \cdot i\right)(\mathbf{u}_{\text{ECEF}} - \mathbf{v}_{\text{ECEF}}) \tag{2.36}$$

These intermediate vertices are added to the mesh in a way that is transparent to the API user and the indices are patched to correctly use also these intermediate vertices. This interpolation happens only once when the user changes the mesh.

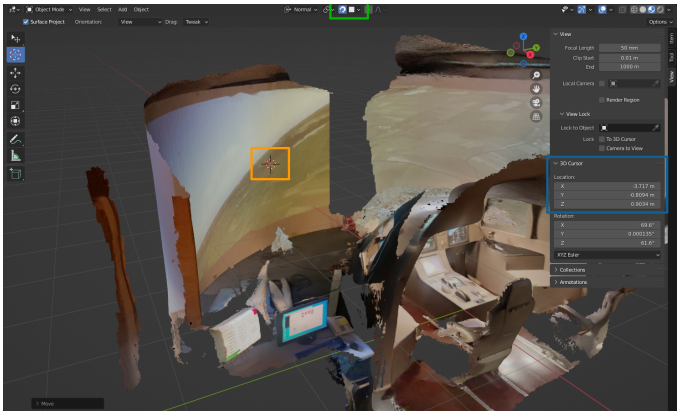


Figure 2.27.: After importing the 3D scan in Blender, the "cursor" tool (shown in the orange box) can be used to obtain the coordinates of various points on the 3D model. In this case, the "snap" feature of the cursor is enabled (green box): this allows the cursor to stick to the faces of the model. On the right (blue box), Blender allows to retrieve the current position of the cursor.

2.3.6. Measuring the simulator's parameters

As described in subsection 2.3.4, a number of simulator-specific parameters are required to correctly compute the projection that HoloAssist must apply. In commercial simulators these parameters are usually available as part of the documentation and are therefore straightforward to obtain. However, the simulator in which HoloAssist has been tested (shown in all the previous pictures) is a research one, and has been built incrementally over several years from multiple different researchers involved in different projects: this results in a lack of official documentation and in the necessity of figuring out these parameters in an alternative way.

The most intuitive approach involves a tape measure, but the structure and scale of the simulator make this difficult. Determining exactly whether the projector screen is actually cylindrical, where the center of such cylinder is and how to measure its diameter/radius is not straightforward, as it requires measuring lengths across the whole room, with limited maneuver space and potentially intersecting various physical objects with irregular surfaces that cannot be moved (like the simulator cockpit itself). It was therefore decided to determine these parameters indirectly, by using the simulator's 3D scan acquired as described in section 2.2. After opening the 3D scan in Blender, its "cursor" tool was used to obtain the virtual 3D position of a number of points on the projector screen, as shown in Figure 2.27.

For each of these points, the Y coordinate has been discarded and the X and

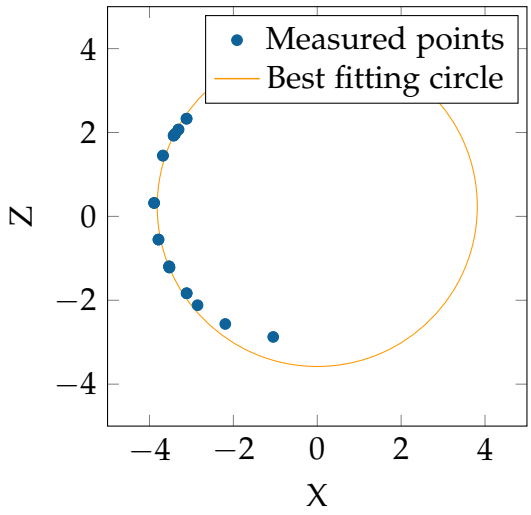


Figure 2.28.: The plot obtained by trying to fit a circle to the points on the projection screen measured using the 3D scan. As it can be seen, the measured point are not really on a circumference and rather form an ellipse.

Z coordinates have been used to determine the best fitting circle, obtaining both the cylinder center and the cylinder radius. As the result in Figure 2.28 clearly shows, the projector screen is definitely not cylindrical and would be better approximated with an elliptic cylinder. However, since this would require deriving a different projection with respect to the one shown in subsection 2.3.4 and since the augmentations were visually mostly correct also by keeping the regular cylinder projection, this improvement was not implemented and the parameters of the best fitting circle were used. Nevertheless, if the user moves outside the simulator cockpit the error is significantly more visible, as shown in Figure 2.29.

The last parameters to determine are the PEP and the airplane center of mass. The rigid offset between these two points has been obtained from the simulator engine configuration, as it is a parameter of the simulation. The PEP was then placed in roughly the correct position with respect to the simulator digital double, and its position was further refined manually via the remote Unity editor described in section 2.6 until the geo-fixed augmentation would visually match with the geographical features shown on the projection screen by the simulator. While doing this, it is important to test the positioning of the PEP by looking at the same augmentation from several different airplane positions and orientations, as it easily happens that the augmentation appears to be completely aligned with real world features from one aircraft position but then loose the alignment as soon as the position changes.

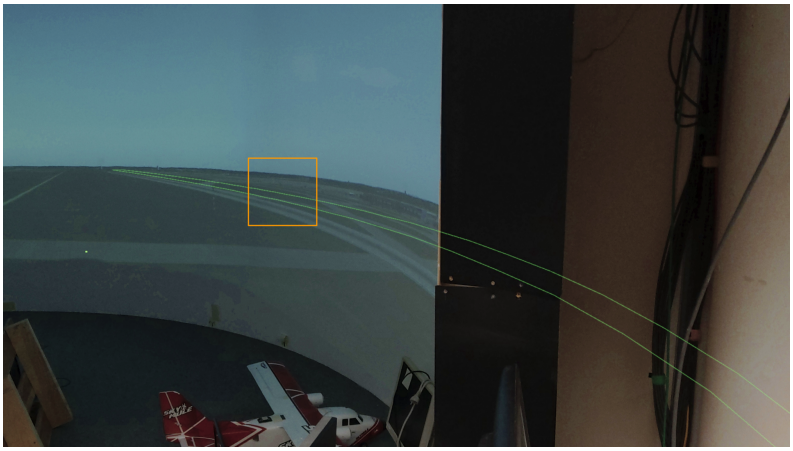


Figure 2.29.: Image taken from the point of view of the Hololens that shows the error between the computed cylinder and the real surface of the projector screen. Since it is difficult to show in a static image, the point in which the projection cylinder "goes behind" the real projection screen is highlighted in orange: the augmentation on the right of this point is only visible because the spatial awareness feature of the Hololens is disabled. Inside the simulator cockpit this error is visually much less relevant.

2.3.7. Limitations due to floating-point precision

The transformations to convert a ECEF point to Unity coordinates and to project it on the cylindrical surface can be performed in `float` or `double` precision and need be executed every frame either on the CPU or on the GPU in a vertex shader. Due hardware limitations of the current version of the Hololens, the device's GPU only supports `float` operations. It was therefore decided to implement a GPU-based version of these transformations that uses `float` precision and a CPU-based version that uses `double` precision, and then compare the results. As can be seen in Figure 2.30, when using the higher-precision version the result is smoother, both spatially and temporally[6]. This is likely due to the fact that ECEF coordinates tend to be fairly big numbers (in absolute value) whereas the final Unity coordinates tend to be fairly small: in this situation, `float` arithmetic easily leads to a loss of precision that manifests with these artifacts. On the other side, the GPU implementation should be faster, as vertex shaders are designed exactly to perform this kind of computation for every vertex. However, since both the CPU-based and the GPU-based implementation are performant enough to achieve a stable framerate for the currently developed geo-fixed augmentations, the performance difference was not tested in details.

[6] Although this second aspect is more difficult to show in a static medium like a printed thesis.

(a) 32-bit float precision (b) 64-bit double precision

Figure 2.30.: Images acquired by the Hololens that show the same geo-fixed augmentation rendered with the two different precisions. The 32-bit version presents some artifacts. The temporal inconsistency of these artifacts manifests as a small jitter of the geo-fixed augmentation around the correct position.

Future versions of the Hololens should include GPU support for `double` precision operations and should remove the necessity of choosing between visual quality and performance.

2.4. Plane-fixed augmentations

The other kind of augmentations supported by HoloAssist is called "plane-fixed augmentation" and consist of augmentations that are shown in a fixed position with respect to the airplane's cockpit. Some envisioned use cases for this kind of augmentations are highlighting commands inside the cockpit or showing virtual displays to the pilot.

The setup described in section 2.1 and section 2.2 already offers a coordinate system aligned with the simulator digital double (and therefore with the real simulator cockpit) in which this kind of augmentations can be shown. Therefore, offering this functionality to the user only requires extending HoloAssist's API to enable creating, editing and deleting normal 3D meshes (that is, instances of `UnityEngine.CoreModule.Mesh`), which are then added to the Unity scene at a position and orientation specified on mesh creation. The resulting API is very close to the one for geo-fixed augmentations and allows to modify the vertex list and index list of every mesh created through the API. The main difference is that instead of defining vertices as `GeoFixedVertex`, they only consist of a 3D position and a color. In this case, neither interpolation nor projection is required.

Figure 2.31.: Image acquired from the Hololens that shows an example plane-fixed augmentation that consists of a colored rectangular outline surrounding the gear lever near the center of the picture.

An example of a plane-fixed augmentation which highlights the gear lever is shown in Figure 2.31.

The main problem posed by this kind of augmentation regards the precision required: although the alignment between the real and the virtual world is fairly precise, it still has an error of around 1cm. Therefore, the precision is not sufficient to highlight particularly small elements, like a single knob in the cockpit. This partially limits the currently available use cases for which plane-fixed augmentations can be used.

Due to the 3D scan acquired as described in section 2.2, designing a plane-fixed augmentation is fairly easy: it is sufficient to open the 3D scan in a digital modeling tool like Blender and use the software to design the desired augmentation as a normal 3D mesh, using the imported scan as a reference for dimensions and positions. The designed mesh can then be exported (e.g. as a Wavefront OBJ file[50]) and drawn via the HoloAssist API. This is further simplified by the fact that the 3D scan not only contains the 3D geometry of the simulator, but also a projected texture which shows the real-world color of each point of the scan as shown in Figure 2.9: this allows to distinguish small scale details that do now appear in the raw geometry. An example of an augmentation drawn in Blender is available in Figure 2.32.

Creating this kind of augmentations without such a precise 3D scan would be much more cumbersome and time consuming: this is one of the reasons for which some of the 3D scan attempts described in section 2.2 (like using the Hololens spatial awareness mesh) were discarded.

Figure 2.32.: The gear lever plane-fixed augmentation as it appears in Blender. As it can be seen, the textured 3D scan is used as a base to draw the mesh. The result will then be exported and loaded through a HoloAssist App.

As a last remark, it should be noted that currently also plane-fixed augmentations are limited to only accept meshes with line topology. In this case, the limitation is not due to rendering issues, as in this case the normal pipeline would have worked without problems, but due to time-related constraints: supporting meshes with triangle topology would also have required an API capable of supporting materials (to handle triangle colors, transparency and more). Designing such API is time-consuming and, since plane-fixed augmentation are a comparatively less important use case for HoloAssist, it was decided to deprioritize this feature.

2.5. Integrating a new simulator

HoloAssist is designed in a way that allows integrating a new fixed-platform simulator with minimum additional effort. In order to do this, the first step is to acquire a 3D scan of the new simulator using a recent iPad[34] (or, alternatively, the photogrammetry pipeline). If acquiring the whole simulator in a single step is too difficult, different partially-overlapping scans can be aligned with Meshlab's ICP tool[36]. Blender can then be used to prepare a simplified occlusion mesh that can be exported to a Wavefront OBJ file.

The next step requires creating and printing at least two QR codes. They need to adhere to the "Micro QR Code"[50] standard and have a printed size of at least 10cm $\times$ 10cm, the bigger the better. Each QR code must contain a string that

matches the pattern `<simulator_name>-<number>`, where `<simulator_name>` is a string identifying the simulator (and therefore equal among all the QR codes) and `<number>` is a progressive number. These QR codes need to be placed on the simulator according to a few criteria:

1. The further they are away from each other the better.
2. The relative offset between two QR codes should involve more than a single axis. If the offset can be described with a translation along a single one of the three main axes the resulting alignment will be sub-optimal.
3. At least one QR code should not lie on the same plane, in order to improve the orientation estimation.
4. The relative offset between the QR codes should be easy to measure.

At this point, a new folder should be created inside the HoloAssist code repository at the path `//Assets/Resources/Planes/<simulator_name>`, where `<simulator_name>` is the same string printed on the QR codes. Inside this folder, two files should be added: one is a `plane-mesh.obj` file that contains the occlusion mesh that was obtained from the 3D scan and one is a `plane-measures.json` file that contains all the simulator-specific parameters.

The first part of the `plane-measures.json` that should be filled regards the space pins. The numerical values for the first space pin can be chosen arbitrarily, but the other two need to match the real world offset between the QR codes that was previously measured. For example, imagine the real-world setup shown in Figure 2.33, where the two QR codes are on the same plane, but one is rotated 90° to the left (to ensure that the offset between them is on at least two axes). The relative distances between the QR codes are as shown in the figure. The space pin part of the JSON file for such a setup is shown in Figure 2.34.

Please note that for this QR code disposition and for this coordinate choice, the Z and Y axes of the resulting Unity coordinate system will be respectively aligned with the real simulator's cockpit longitudinal axis and vertical axis as required by HoloAssist (see subsection 2.3.3). Running HoloAssist at this point and scanning the QR codes should result in a small colored cube appearing in the top left corner of each QR code.

The next step entails correctly placing the simulator digital double in the virtual space. This can be done by specifying a random value for the `planeMesh` key of the JSON file. Then, after starting HoloAssist, the mesh can be visually aligned by using the remote Unity editor (see section 2.6), which allows the live editing of position and rotation of Unity objects. The final position of the mesh

Figure 2.33.: An example real-world setup for placing the required QR codes.

```
{
    "spacePins": {
        "<simulator_name>-1": [0.0, 0.0, 0.0 ],
        "<simulator_name>-2": [2.0, 0.10, 0.0 ],
    },
    // other simulator-related information
}
```

Figure 2.34.: An extract of the JSON file that describes the virtual positions of the space pins for the setup shown in Figure 2.33.

can then be read and be written as the final value of the `planeMesh` key. At this point, HoloAssist should be able to scan the QR codes and correctly place the digital double.

If the projector screen shape is not cylindrical, then an additional projection method should be implemented. If it instead is cylindrical, then the position of the cylinder center, cylinder radius and PEP should be obtained, either from the simulator's documentation or as described in subsection 2.3.6. Their position in Unity's frozen coordinate system should then be stored in the JSON file. Once again, refining these points (especially the cylinder radius) via the remote Unity editor is possible.

The last step entails ensuring that the simulator broadcasts the airplane position and orientation correctly to HoloAssist. However, this is a simulator-specific step, and therefore cannot be detailed here. The required data format is available in Table 2.1. At this point, HoloAssist should be ready to show augmentations in the new simulator.

This procedure has been used to succesfully integrate HoloAssist in a second fixed-platform simulator available at the research institute in which this project was developed. The integration required only a couple of days of work, with the main difficulties being networking issues and finding the correct positioning of the PEP.

2.6. Developer utilities

While working on HoloAssist, it was deemed useful to develop two additional utilities that would help simplifying the development process: the remote Unity editor and the simulator position updater.

The remote Unity editor is composed by two parts: a Unity `Component` that listens for some particular UDP messages and a Rust application designed to be run on a computer and to interact with this component via the local network. The main use case for this utility is that, in some cases, it is very helpful to be able to wear the Hololens, start HoloAssist, load some holograms and then modify the position and rotation of such holograms without having to recompile and redeploy the Unity application to the device. The remote Unity editor allows to select a Unity `GameObject` by name and then use the computer's keyboard to translate it or rotate it while HoloAssist is running. Using a Bluetooth keyboard instead of a wired one lets the user easily tweak the position of holograms while moving around in the AR experience and looking at them through the Hololens. This is useful, for example, for refining the position of plane-fixed augmentations or for tweaking the position of the PEP as described in subsection 2.3.6. It is

also possible to dump the current position and orientation of a `GameObject` to a UDP packet which can be received and inspected via Wireshark, allowing to retrieve the refined pose of any object of the scene. This UDP packet is a UTF-8 encoded string which is automatically decoded by Wireshark and contains a self-descriptive JSON object.

Related to this is the possibility of using Wireshark to read debug UDP packets sent from the Hololens to the computer. This is useful because, when compiled in release mode and deployed to the device, HoloAssist is very difficult to debug. This is due to the fact that logs lack debug symbols, often are very minimal and do not report invocations `Debug.Log` (the standard logging utility in Unity). As a work around, HoloAssist can send log messages via UDP, which offers a limited way of debugging problems that only appear when the application is compiled in release mode or if the debug build is too slow to be usable. A late discovery was the possibility of enabling Unity Development Builds, which still apply all the release optimization but offer additional introspection capabilities, like showing the logs printed through `Debug.Log` and being able to profile the application through Unity's profiler.

The other utility that was developed is the "simulator position updater", which allows to send simulator status updates to HoloAssist without a real flight simulator broadcasting updates. This utility was created to enable the development of HoloAssist also when working from home or when the simulator was being used by someone else. The application allows to quickly set the simulator position to a few hardcoded presets and to tweak the plane geographical position and orientation by using a keyboard. This is achieved by using the same UDP interface through which real simulator status updates are received, using the packet structure described in Table 2.1.

3. HoloAssist Apps

As previously stated, one of the main aims of this project is to enable researchers with limited computer science experience to create and test AR experiences inside a flight simulator. One of the reasons for which HoloAssist has been designed as a standalone application that exposes an high-level API was precisely this: asking someone with limited programming background to become familiar with C#, Unity, the Mixed Reality Toolkit and the workflow required to develop on the Hololens was simply not feasible. Using HoloAssist's API over the local network is significantly easier, but it can be simplified further.

To do this, it was decided to create a "paved road"[51] that could act as a starting point and preferred way to use HoloAssist's API. The Python programming language[52] was the obvious choice for the foundation of this paved road: its pseudo-code-like syntax, dynamically typed nature and runtime safety make it (comparatively) easy to use, also for inexperienced programmers. The trade-off is that Python is a really slow interpreted language, even when compared with similar languages. Fortunately, this disadvantage is overcome by the immense collection of open-source libraries and packages that are available online[53]: some of them, like Numpy[54] or Scypi[55], offer easy to use Python bindings to extremely performant scientific computation libraries written in low-level languages like C, C++ or Rust. Installing these packages and using them is almost trivial (especially if compared to the alternatives) and this allows to deal with Python's performance problems. Combining this with the availability of other packages that cover a large number of scientific computing necessities (e.g. PyProJ[42] for geodesy, Skyfield[56] for astronomy, Matplotlib bindings[57] for plotting, PyTorch[58] for deep learning, and many more) makes for a very strong value proposition, which has lead to an ever increasing adoption of the Python language and ecosystem (see Figure 3.1). This makes it therefore more likely for aspiring users of HoloAssist to already be familiar with this language.

After deciding the language for the paved road, the next step was to create a Python class (`HoloAssistService`) that would act as a wrapper for HoloAssist's API: in this way, users only have to invoke normal Python methods on instances of this class and not have to worry about the details of the network protocol and the format chosen for data serialization. After creating this class, the only remaining steps were to write the appropriate documentation and to create a

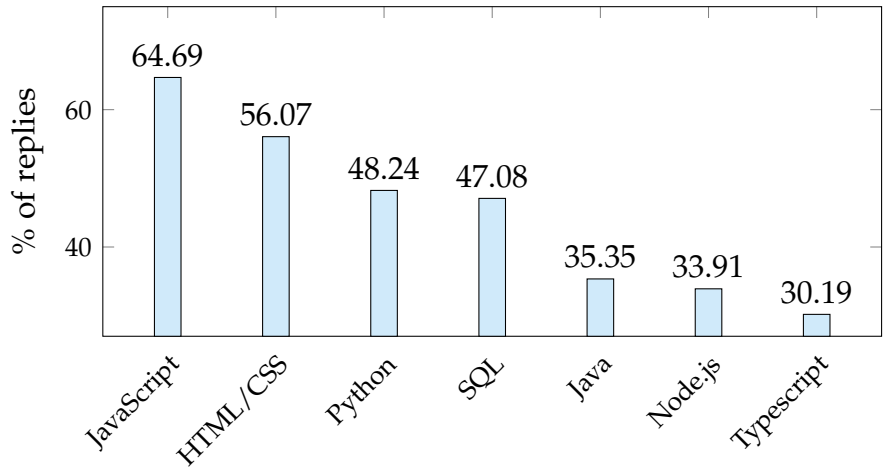


Figure 3.1.: The 2021 Stackoverflow Developer Survey[59] reports Python as the third most-popular language among developers, up one position with respect to the previous year. The plot shows the results of the question "Which programming, scripting, and markup languages have you done extensive development work in over the past year?"

few example HoloAssist Apps that would use this paved road and could act as a starting point for others that will be developed in the future.

3.1. Example HoloAssist Apps

The simplest HoloAssist App that could be developed is one that highlights a particular landing strip of an airport, and is reported in its entirety in Figure 3.2. Obtaining the geographical coordinates of the corners of the landing strip can be done through Google Maps. Once these coordinates are known, the HoloAssist API can be used to create the rectangular line mesh that will highlight the desired runway. Doing this requires building a list containing the vertices of the rectangle and a list that defines the lines connecting these vertices. This second list is known as the list of indices, because its elements are the numerical indices of the vertices defined in the first list. Each pair of elements of the list of indices describes a line with endpoints the two vertices identified by the corresponding indices. This approach is similar to how triangular meshes are commonly represented in 3D computer graphics, and its details are explained in subsection 2.3.2. Meshes created through the HoloAssist API in this way are known as GF-ELM in this document. In order to send the list of vertices and the list of indices to HoloAssist, the `HoloAssistService` class offered by the

```
from lib import prepare_holo_assist_instance
from lib.holo_assist_types import Color, GeoFixedVertex, WGS84Point

c = Color(0.0, 1.0, 0.0) #RGB
vertices = [
  GeoFixedVertex(WGS84Point.from_degrees(48.3630, 11.7675, 453), c),
  GeoFixedVertex(WGS84Point.from_degrees(48.3671, 11.8210, 453), c),
  GeoFixedVertex(WGS84Point.from_degrees(48.3666, 11.8212, 453), c),
  GeoFixedVertex(WGS84Point.from_degrees(48.3625, 11.7676, 453), c)
]

indices = [0, 1, 1, 2, 2, 3, 3, 0]
MESH_ID = "EDDM␣08␣L"

service = prepare_holo_assist_instance()
service.create_mesh(MESH_ID)
service.add_mesh_vertices(MESH_ID, vertices)
service.add_mesh_indices(MESH_ID, indices)
service.commit_mesh_changes(MESH_ID)
service.activate_mesh(MESH_ID)
```

Figure 3.2.: An example HoloAssist App that highlights the `08 L` runway of the Munich airport (`EDDM`). The explanation of what each API call does is available in Appendix A.

paved road can be used. It offers a wrapper around the raw API which allows to invoke it by simply calling normal Python methods. The result created by this HoloAssist App is shown in Figure 3.3.

Another example HoloAssist App that was implemented shows how to take an arbitrary Wavefront OBJ line mesh and draw it at a specific geographical point. To achieve this result, a couple of utility methods to load such mesh file and to convert it to `GeoFixedVertices` have been developed and offered as part of the paved road. An example usage of these methods is shown in Figure 3.4: the vertex and index lists it generates can then be used with the HoloAssist API as usual. The result that can be obtained from such HoloAssist App is shown in Figure 3.6. Creating a 3D mesh like this can be done in Blender, by drawing the desired shape, deleting all the faces (keeping therefore only vertices and edges) and by exporting the result as OBJ. As it can be seen in Figure 3.5, the mapping between the original mesh vertices and the `GeoFixedVertex` datastructure is

Figure 3.3.: An image acquired from the Hololens that shows the visual result of one of the simplest HoloAssist Apps that can be created.

straightforward and does not involve any computation, just a direct conversion.

The last example HoloAssist App focused on geo-fixed augmentations that has been developed is an adapter that is able to use HoloAssist's API to draw a tunnel mesh starting from a CSV file written in a custom format in use at the research institute in which this project was developed. Luckily, each CSV line could be mapped one-to-one to an HoloAssist API call, therefore writing the adapter was fairly straightforward.

An example HoloAssist App that creates a plane-fixed augmentation has also been created, obtaining the result shown in Figure 2.31.

3.2. Augmenting the Innsbruck approach

Besides the example HoloAssist Apps described in section 3.1, the main test case for the developed platform was enhancing the landing experience for the Innsbruck airport. As the approach chart in Figure 3.7 shows, the airport is surrounded by mountains: this complicates the landing, especially in conditions of limited visibility. The main idea is that AR could be used to highlight the elements relevant for the approach, improving the situational awareness of the pilot and leading to a safer landing.

It was decided to focus on the standard procedure `RNP-Y-RWY-08`, shown in Figure 3.7. This approach is for runway 08 which, due to the mountains surrounding the airport, requires performing a left turn shortly before actually landing. Especially with limited visibility conditions, following the correct

```
COLOR = Color(0.0, 0.8, 0.0)
POSITION = WGS84Point.from_degrees(48.36306, 11.76755, 453)
ROTATION = Rotation.from_degrees(0, 0, 0)

map_pin_obj = simple_obj_importer.load_obj_line_mesh("/path/to/mesh.
    ↪ obj")
(vertices, indices) = convert_obj_to_geo_fixed_mesh(
  map_pin_obj, COLOR,
  POSITION, ROTATION
)
```

Figure 3.4.: An example usage of the utility functions that convert a normal Wavefront OBJ line mesh to a format that can be sent as-is to HoloAssist's API.

```
  def convert_obj_to_geo_fixed_mesh(
    mesh: ObjLineMesh, mesh_color: Color,
    mesh_geo_position: WGS84Point,
    mesh_local_rotation: Rotation):

    vertices = [
      GeoFixedVertex(
        mesh_geo_position, mesh_color,
        Vector3(-point[0], point[1], point[2]),
        mesh_local_rotation
      ) for point in mesh.vertices
    ]

    flattened_indices = itertools.chain.from_iterable(mesh.lines)
    indices_in_base_zero = [i - 1 for i in flattened_indices]

    return (vertices, indices_in_base_zero)
```

Figure 3.5.: The paved road implementation of the conversion from Wavefront OBJ to `GeoFixedVertex`. It is fairly straightforward, although care should be taken to convert from Blender's right-handed coordinate system to Unity's (and HoloAssist's) left-handed coordinate system. That's the reason of the "-" sign applied to one of the coordinates of the mesh.

Figure 3.6.: An image acquired from the Hololens that shows how HoloAssist Apps can be used to render arbitrary line meshes.

trajectory can be fairly challenging. For such procedure, three elements were identified as relevant and useful to be highlighted:

1. The airport landing strip;
2. The mountains on the left side of the approach trajectory, as the aircraft needs to fly quite close to them to correctly follow the procedure;
3. The desired approach trajectory itself, in the form of a tunnel inside which the airplane should stay.

Each of these three elements was implemented independently in its own HoloAssist App.

The geographical coordinates of the runway corners were obtained via QGis, an open-source software designed for reading and writing geographic information system (GIS) data. The QGis project consisted of two layers: a base layer with the satellite imagery from Google Maps and a feature layer on which points and shape could be drawn. After drawing the runway outline on this layer, its content was exported to a GeoJSON[61] file. The `geojson` Python package[62] was then used to read this file into an HoloAssist App that mapped its content to the datastructure required by HoloAssist's API. The result is shown in Figure 3.8.

Highlighting the mountains on the left side of the approach trajectory was less straightforward, as it required drawing a 3D mesh that would match their shape, at least roughly. Fortunately, the Austrian government (through its open data platform) has released a publicly available digital elevation model for

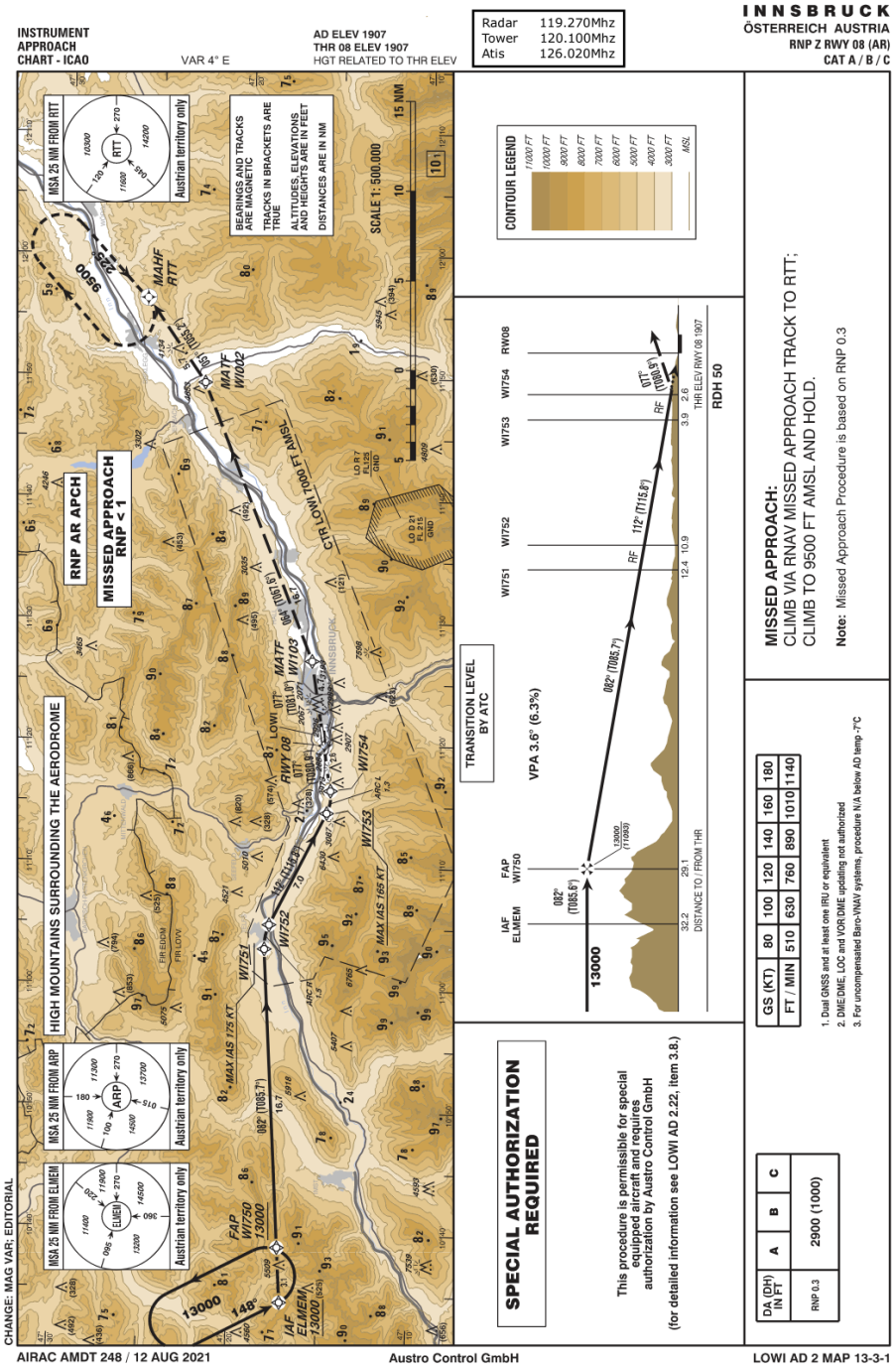


Figure 3.7.: The approach chart to the Innsbruck airport for the procedure `RNP-Y-RWY-08`. This approach chart is offered for simulation use only by VACC-Austria[60].

Figure 3.8.: The HoloAssist App that highlights the Innsbruck airport's runway.

the entire Austrian territory with a 10m resolution, which is precise enough for this use case[63]. In this model, each pixel represents a real world square with a side of 10m and the color of the pixel encodes the average height of that real world square. The parameters of the mapping between the color and the real-world height are embedded in the the image metadata, as it is common for the GeoTIFF[64] format in which it is distributed. Such metadata allows to effortlessly import it in the QGis project as another geo-referenced layer, which is then trimmed to the appropriate area of interest as shown in Figure 3.9. This area, which surrounds the Innsbruck airport, is small enough for the Earth curvature to be negligible. This means that this chunk of the elevation model can be exported from QGis as a normal gray-scale image that can then be imported in Blender and be used as an height map to create virtual geometry that roughly matches the real world terrain. Such geometry can then be enhanced by exporting the Google Maps satellite view of that same area from QGis as a normal RGB image that can be then applied to the virtual geometry in Blender. This process results in an approximate digital reconstruction of the real-world that surrounds the Innsbruck airport. Such reconstruction is precise enough to be used as a starting point to draw a 3D mesh that could be used to highlight the mountains, as shown in Figure 3.10. This 3D mesh can then be exported to an OBJ file[1] which can in turn be used by a HoloAssist App to be shown in the real simulator, as shown in Figure 3.11.

[1]Technical note. Before exporting from Blender, the mesh origin (that is, the Blender object position) must be moved back to $(0,0,0)^T$, otherwise the resulting augmentation will be misaligned

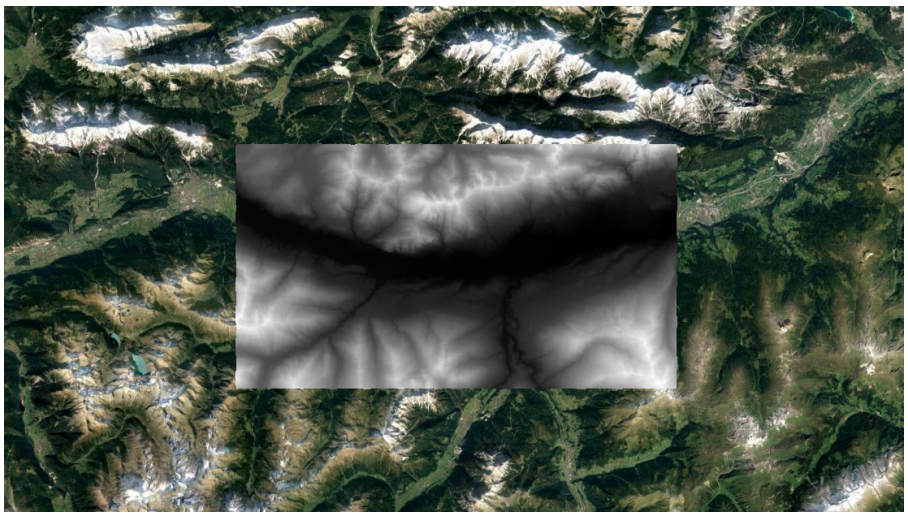


Figure 3.9.: The QGis project used for this HoloAssist App. It shows the satellite imagery of the Innsbruck airport area and the relevant part of the digital elevation model.

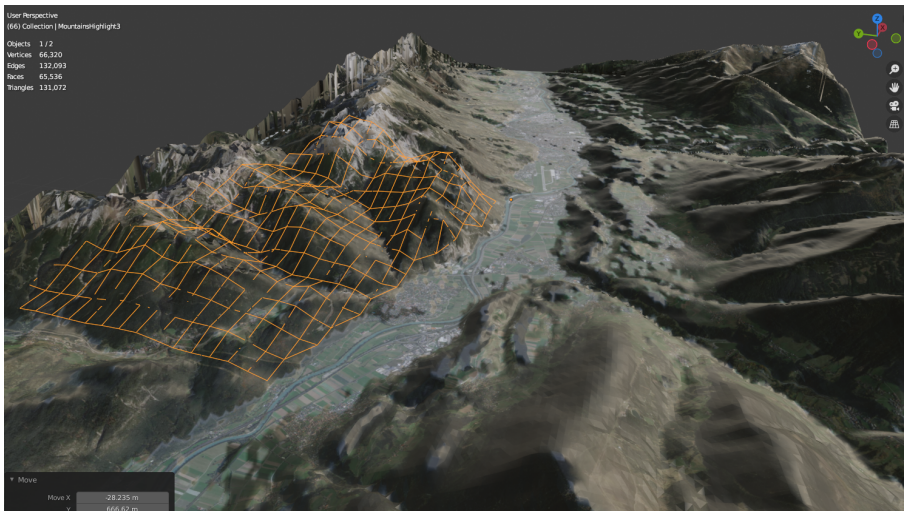


Figure 3.10.: A screenshot of the Blender project that was used to create the terrain 3D mesh for the area surrounding the Innsbruck airport. The mesh is highlighted in orange, whereas the rest of the 3D scenes is occupied by the digital reconstruction obtained by using the digital elevation model as a height map.

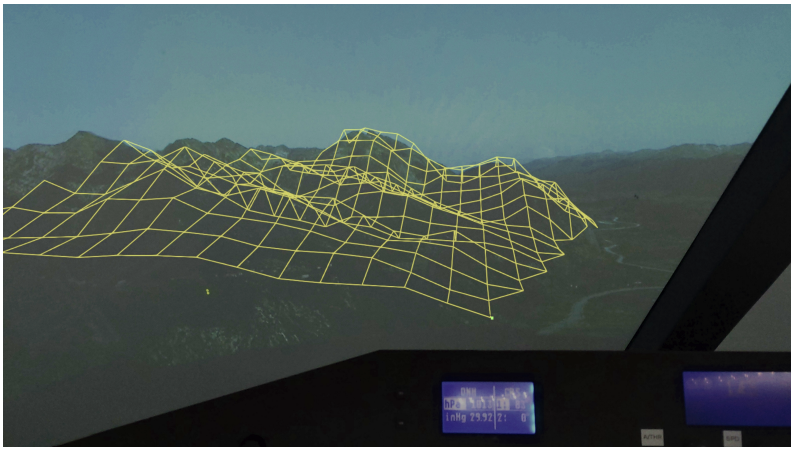


Figure 3.11.: The final appearance of the highlighted terrain in the flight simulator.

During early testing, it was suggested that always showing the whole mountain mesh could easily become overwhelming. It was therefore decided to extend the terrain HoloAssist App so that only the part of the terrain mesh that is closer than a certain threshold is shown. The same simulator position data that updates HoloAssist was therefore duplicated and made available also to the terrain HoloAssist App. While running, the app uses PyProj to convert all the `GeoFixedVertices` that compose the terrain mesh to their ECEF position, using an algorithm similar to the one described in Equation 2.17. The current airplane position is also converted to the ECEF CRS, and the distance between each mesh point and the current airplane position is computed. If the distance is greater than the given threshold then the slots of the mesh's index list that refer to those vertices are set to zero (resulting in those mesh lines disappearing). If the distance goes back below the threshold then they are restored to their original value (making those mesh lines appear again). A few of the physical buttons of the simulator have been integrated (via their UDP interface) in this HoloAssist App and allow the pilot to dynamically change the distance threshold at which vertices (and therefore parts of the terrain augmentation) are hidden. The fact that this is possible is a demonstration of the flexibility of the API exposed by HoloAssist, and the fact that the entire app only requires around 150 lines of Python shows its ease of use and the maturity of the Python ecosystem for this kind of tasks.

The last HoloAssist App that has been developed for the Innsbruck airport shows a tunnel representing the correct approach trajectory that should be maintained by the aircraft to follow the given approach procedure. The approach procedure chosen for this example is `RNP-Y-RWY-08` (shown in Figure 3.7), which

Figure 3.12.: The visual result from of the first iteration of the HoloAssist App displaying the approach tunnel for the Innsbruck airport.

is a required navigation performance procedure: this means that instead of describing the trajectory in terms of ground-based localization equipment, it uses a sequence of geographical points that the aircraft computer follows[2]. This sequence of geographical points for `RNP-Y-RWY-08` has been obtained from the airport approach charts has been written in a CSV file which is then loaded by an HoloAssist App. Each of these points is treated as the center of a rectangle which forms a vertical cross-sectional slice of what will become the trajectory tunnel. The correct indices are then added to connect every corner of every cross-sectional slice to the respective corner of the previous and next cross-sectional slice, yielding the tunnel-like appearance shown in Figure 3.12.

Although this technique is simple to implement, it is not perfect, as the approach trajectory is just piece-wise linearly interpolated, leading to sudden changes of direction in proximity of the points defining the tunnel cross-sectional slices. On a real aircraft, its Flight Management System (FMS) would compute the ideal trajectory using the GPS data of the aircraft, its speed and its aerodynamic performance. However, due to the complexity of these algorithms, displaying the ideal approach trajectory was deemed to be beyond the scope of the current version of this HoloAssist App. Nevertheless, it was decided to try using a spline to smooth the piece-wise-linear interpolation.

To do this, the set of points loaded from the CSV file are transformed to the ECEF Euclidean CRS by using PyProj and are then interpolated with a cubic spline via Scipy. This spline is then evaluated at around 60 different equally-

[2]Of course, this is a simplified description of what a required navigation performace procedure is and only focuses on the aspects relevant for this project.

spaced points, each of which is used as the center for rectangular cross-sectional slice of the approach tunnel. Such rectangle is oriented (by using the local rotation component of the `GeoFixedVertex` data structure) to be perpendicular to the ground and with its normal pointing towards the spline tangent at that point (and therefore "following" the spline). By connecting every rectangle with the previous and following one the result shown in Figure 3.13 is achieved.

The combination of these three HoloAssist Apps (shown in Figure 3.14) leads to the result which will be shown to a few pilots to gather their feedback on this initial example of AR techniques applied to an aircraft cockpit.

Figure 3.13.: The visual result from of the second iteration of the HoloAssist App displaying the approach tunnel for the Innsbruck airport, using a tunnel based on a computed spline.

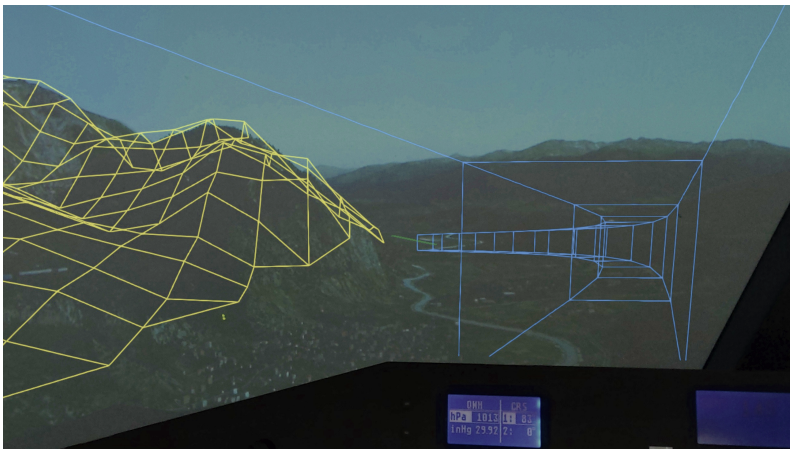


Figure 3.14.: The three HoloAssist Apps developed for the Innsbruck airport.

4. Evaluation

The HoloAssist Apps presented in chapter 3 already satisfy the main aim of this thesis project, which is to develop tools and workflows to easily create AR experiences in a fixed-platform simulator (see section 1.2). HoloAssist and its API were able to address all the use cases encountered while developing those applications, and the resulting Python code was fairly concise and straightforward to write. Moreover, the process to draw a new mesh in Blender and to use the digital elevation model as a visual reference proved to be effective, and produced artifacts that were easy to integrate in a HoloAssist App. Although a formal evaluation of the usability of the API and these workflows was not conducted, informal discussion with other researchers confirmed a general appreciation of the chosen design.

As part of this evaluation process for the HoloAssist platform, a suite of augmentations for enhancing the landing experience at the Innsbruck airport has been developed. Besides being used to evaluate the API flexibility, these augmentations have also been presented to some professional pilots, in order to gather their feedback on this initial attempt at an AR integration within an aircraft simulator. Receiving a generally positive feedback would hint at AR actually being useful in the cockpit, even if the currently developed augmentations fall short of the desired target. On the other side, receiving a more negative feedback could prompt a revision of the core assumptions of this project.

4.1. Evaluating augmentations

In order to evaluate the augmentations developed for the Innsbruck airport in a consistent way, it was decided to follow a partially standardized procedure.

It began with an introductory phase, in which the pilot was allowed to familiarize themselves with the Hololens and its main interaction paradigms, like using its hand detection feature to manipulate the position and scale of a virtual object. They also had to fill in the first part of a questionnaire, which focused on their flying experience and on their relationship with AR/VR.

After this introduction, the pilot was handed the `RNP-Y-RWY-08` approach chart for the Innsbruck airport (see Figure 3.7), which showed the details of

the maneuver that they would have to perform as part of the test. They were explicitly told that strictly following the procedure was not necessary and that they should consider it just as a general indication for the suggested approach, because the main focus of the test was on the user interface presented through the augmentations and not on the actual flight performance.

The pilot was then asked to land the aircraft at the Innsbruck airport by roughly following the given approach procedure and by using standard cockpit instrumentation. All pilots started from the same initial conditions. In particular, the visibility was reduced to 15km, in order to increase the difficulty of the approach while still keeping it feasible to execute without using instrumental flight.

A few moments before touchdown, the simulation was paused and reset to initial conditions, in order to mitigate the stress that could be caused from a sub-optimal landing. After the reset the pilot was asked to wear the Hololens, which had already been configured with HoloAssist running and the three HoloAssist Apps designed for the Innsbruck airport shown. The pilot then had to fly the same approach again and land at the Innsbruck airport. While wearing the Hololens, pilots were explicitly told that they were free to rely on the augmentations that were shown, on the cockpit instrumentation or both. They were also asked to ignore hardware problems caused by the fact that the Hololens is still a research device, like its the weight, limited field of view and occasionally intense color aberration, as these problem will likely be solved by future versions of the HMD.

The simulation was again interrupted a few moments before touchdown and the pilot was asked to fill in the second part of the questionnaire, which contained a structured comparison between the approach with and without the Hololens, and a few questions on the general applicability of AR in aviation. In conclusion, a short open discussion in which the pilot was free to express any opinion they had on what they saw closed the test procedure, allowing for a more general and free-form kind of feedback to be expressed.

4.2. Questionnaire results

The HoloAssist Apps developed for the Innsbruck airport have been shown to a small sample ($N = 5$) of pilots, which experienced them following the evaluation procedure described in section 4.1. The sample was composed mostly of amateur pilots with an amount of flight experience ranging from five to fifteen years and mostly focused on single engine planes and ultra-light aircrafts. Most of the sample was already familiar with VR technologies, but had no previous

experience with AR. No symptoms of simulator sickness or of VR/AR sickness have been reported during or after the evaluation procedure.

The structured comparison between the approach with and without the Hololens consisted in a set of questions which had to be answered twice, once for each approach. The answer had to be given on a five-point Likert scale, ranging from "very easy" to "very difficult". The results of this comparison are reported in tabular form in Table 4.1 and as a stacked bar chart in Figure 4.1.

It can be observed that, overall, the approach with the Hololens was easier for the pilots, in particular regarding the estimation of the optimal trajectory. If only the questions regarding the correct approach path, turn radius and descent slope are considered, adding the Hololens yielded an improvement of around one level of the Likert scale: this means that the approach tunnel augmentation is fairly effective, and its appreciation was confirmed during the unstructured discussion. On the other side, the augmentation for the terrain surrounding the airport did not yield a significant benefit, and the unstructured discussion explained why: the visibility was still good enough to have a clear understanding of the terrain geometry, and augmenting it was superfluous. However, a quick test performed by one of the pilots suggests that in a situation with significantly worse visibility (around 100m instead of 15km), the terrain augmentation would instead be really useful.

The runway augmentation caused mixed reactions. Pilots agreed that from far away[1] the augmentation was useful to get a rough positioning of the runway, especially given that at the beginning of the approach it was occluded by a mountain and without the augmentation pilots could only guess its location. However, as the airplane moved closer to it, the misalignment between the real runway and the augmented one caused significant confusion and degraded the experience. This misalignment was caused[2] by the difficulty of correctly locating the PEP for the projection setup and by the instability of the position estimate made by the Hololens API for the QR codes. Besides completely fixing the alignment issue (which has already been significantly improved), another suggested solution was to have the runway augmentation disappear once the airplane is closer than a certain distance threshold (dependent on the current visibility conditions).

The last part of the questionnaire was designed to gauge the pilot's opinion on the applicability of AR to aviation in general, and its results are shown in Figure 4.2. These questions show that in conditions of lower visibility pilots

[1] One pilot put the threshold at 3NM for the situation presented in this test.

[2] Unfortunately, the root cause for this misalignment has been identified and fixed only after gathering the pilots feedback. The version of HoloAssist described in this document already includes these fixes.

would definitely appreciate the help offered by the Hololens, but they would not use it if the sky is clear. They would like to have AR capabilities integrated in the aircraft models they usually fly and in general think that AR has the potential to improve the instrumentation commonly available in commercial airplanes. However, the unstructured discussion highlighted that, although the Hololens could be a desirable addition to the standard instrumentation, it does not seem sufficient to entirely replace it.

4.3. Further remarks

Besides the results reported above, a few more elements worth mentioning emerged during the various unstructured discussions.

One of the pilots included in the sample happened to also be a flight instructor and pointed out how relevant the usage of the Hololens to display an approach trajectory could be for new pilots during training. Further investigation in this space is definitely warranted. Another pilot pointed out that sometimes, especially for flights under visual metereological conditions (VMC), the approach trajectory is not always fixed and standardized: this means that the way in which the approach tunnel has been computed for this test might not be applicable to all situations.

Almost all of the interviewed pilots expressed the desire to include some elements of the primary flight display in the Hololens itself, in particular as far as speed, vertical speed, altitude and flight path marker are concerned. Since this was an almost universal request, its implementation should definitely be prioritized in future works.

Pilots were also asked for their opinion about the integration of Hololens-like systems in eVTOLs, and the replies were mixed. Although there seems to be potential in such kind of integration, there is also the risk of overwhelming inexperienced pilots with too much information and with the complexity of the new interface. Just copying elements like speed, altitude and flight path marker from the standard primary flight display to an AR HMD would not be sufficient, and a more thoughtful analysis that takes into account also the capabilities of the flight controller is required. Besides approach path tunnels, other suggested visualizations for eVTOLs include the highlighting of obstacles (which would be particularly useful in an urban environment) and visualizing the trajectory that the eVTOL will follow during flight configuration transitions (e.g. when transitioning from vertical flight to horizontal flight). In particular, this second aspect is quite relevant, as visually determining the point in space at which such a transition is complete is not trivial. A final suggestion, useful in the case of

How difficult was it to...		Pilot A	Pilot B	Pilot C	Pilot D	Pilot E	avg.
...estimate the distance of the terrain?	w.	1	1	2	3	1	
	w/o.	2	1	1	3	1	
	imp.	1	0	-1	0	0	0
...estimate your altitude?	w.	2	1	2	2	1	
	w/o.	2	1	1	4	1	
	imp.	0	0	-1	2	0	0.2
...estimate the position of the runway?	w.	2	1	-	2	2	
	w/o.	1	2	-	3	4	
	imp.	-1	1	-	1	2	0.75
...estimate the correct approach path?	w.	1	1	1	1	1	
	w/o.	2	2	3	2	3	
	imp.	1	1	2	1	2	1.4
...estimate the turn radius of the last curve?	w.	1	1	1	1	1	
	w/o.	3	1	2	3	1	
	imp.	2	0	1	2	0	1
...estimate the correct descent slope?	w.	1	1	2	1	1	
	w/o.	1	2	3	3	1	
	imp.	0	1	1	2	0	0.8
...maintain the correct speed and altitude?	w.	1	1	2	3	1	
	w/o.	3	1	3	3	1	
	imp.	2	0	1	0	0	0.6
average improvement		0.71	0.43	0.50	1.14	0.57	

Table 4.1.: Results of the structured comparison between the approach with and without the Hololens. The table reports the reply that each pilot gave on a five-point Likert scale ranging from "1 (very easy)" to "5 (very difficult)". The table also reports the improvement that each pilot perceived after wearing the Hololens, computed as the difference between the score reported without ("w/o.") the HMD and with it ("w."): a negative improvement means that wearing the Hololens worsened the pilot experience. The last column shows the average improvement across all pilots for a given question, whereas the last row shows the average improvement across all questions for a given pilot.

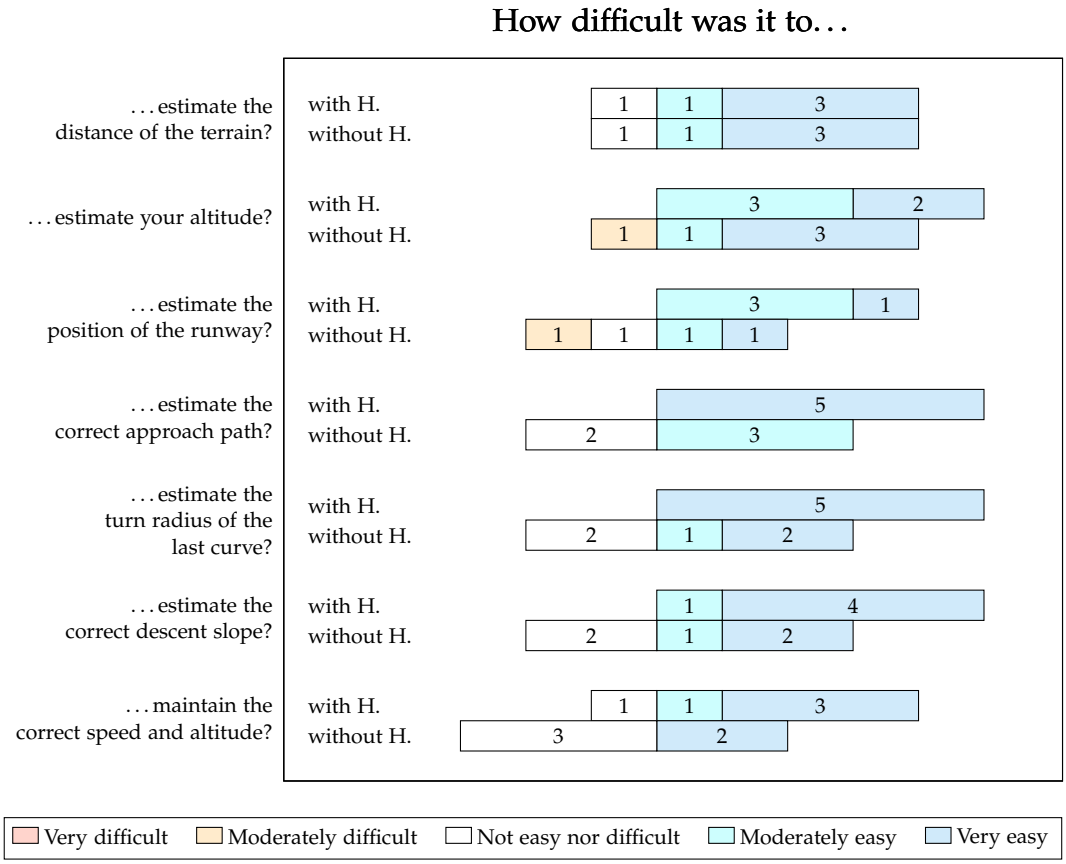


Figure 4.1.: Results of the structured comparison between the approach with and without the Hololens. The plot shows how many pilots replied in a certain way to a certain question. The sample size is too small for this plot to be really effective, but it still shows that on average replies to questions regarding the approach without the Hololens tend to be slightly more shifted towards the "very difficult" side than replies to questions regarding the approach with the Hololens.

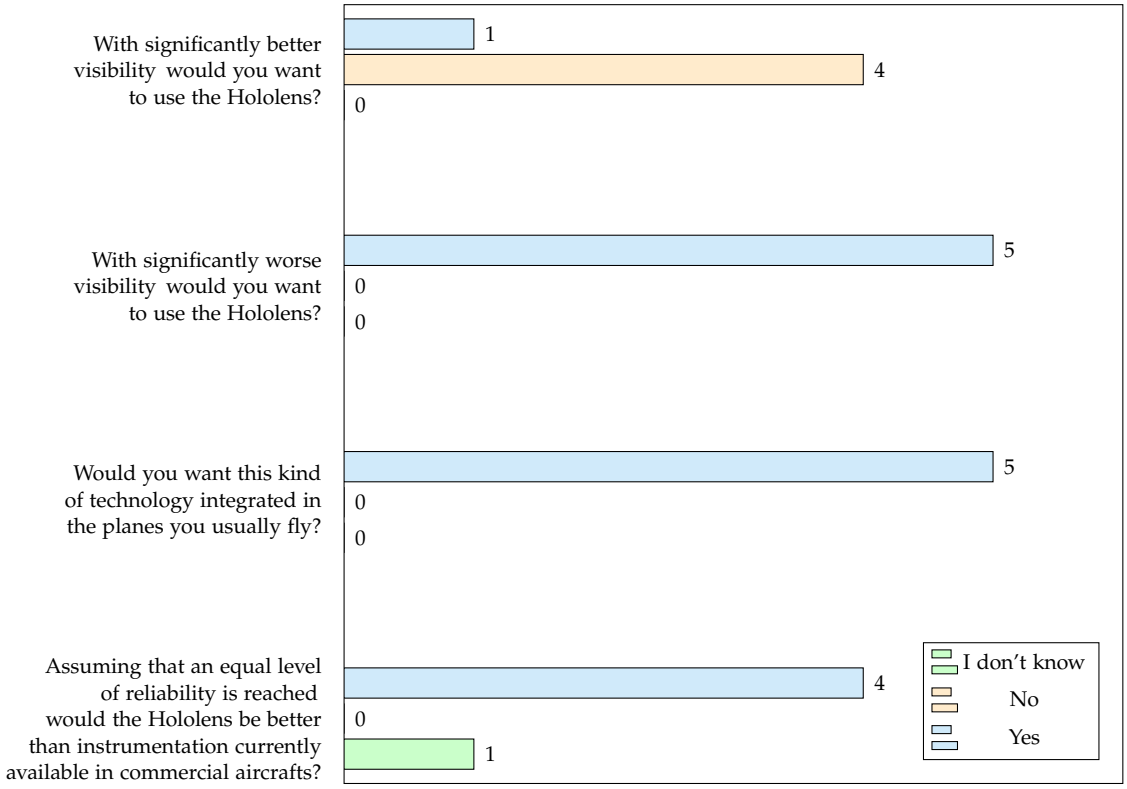


Figure 4.2.: Results of the general questions asked to pilots regarding their opinion on the more general applicability of AR to aviation.

vertical landing, would be to have an augmentation showing the landing area at the current altitude of the eVTOL: in this case, the pilot would not have to align the aircraft with a landing pad of the ground, which is probably difficult to look at, but could just look around at its current height level.

In conclusion, pilots were asked if they had ideas for other augmentations that could be shown during the Innsbruck approach. In no particular order, they suggested to:

1. Add an augmentation that shows the distance to the current target (e.g. to the runway). Such an augmentation could look like quest markers in video-games, appearing above the current target and displaying the distance from it in an appropriately scaled font;

2. Add an augmentation that integrates traffic collision avoidance system (TCAS) information to highlight the position of other planes directly in the pilots field of view;

3. Extend the terrain augmentation by using color to signal which part of the terrain cannot be out-climbed with the current aircraft performance. For example, if the airplane is too close to a specific terrain feature to out-climb it before reaching it, that part of the terrain would be colored in red;

4. Add an augmentation that extends the center line of the runway, so that the pilot would more clearly see how they should align the airplane for the landing. This augmentation should also interact with the terrain surrounding the airport, for example by being properly occluded by the mountains;

5. Add an augmentation that uses arrows to signal wind direction and intensity in the area surrounding the current airplane position.

5. Conclusions and future work

In conclusion, it can be said that the problem statement reported at the beginning of this document has been adequately investigated. A rapid prototyping platform for developing AR experiences inside a fixed-platform flight simulator has been successfully developed and the results obtained with it for the Innsbruck airport augmentations are encouraging. This platform proved to be flexible and reasonably simple to use and pilots reported a generally positive feedback for the usage of AR in a fixed-wing aircraft.

Nevertheless, the questionnaires seem to suggest only a small (or at best moderate) improvement, and a legitimate question is whether it is sufficient to justify the costs associated with adding AR solutions to an airplane cockpit. Moreover, the Hololens by itself is not reliable enough for usage in an actual aircraft, and integrating it into one would require significant additional work, rising the associated costs even more.

5.1. Future works

Although the achieved result is satisfactory, there are a number of improvements that could be implemented in following versions of HoloAssist.

The single biggest one is to remove the limitation on line meshes: this should be possible, as the only thing preventing it is the visibility issue after projecting all the vertices to the cylinder surface and a very similar problem already has a solution in computer graphics[48]. A possible approach could be to map a texture to the cylinder surface and use this texture as a depth buffer to decide the final color of each pixel. This would also allow the terrain augmentation for the Innsbruck airport to have proper occlusion and to only show the side of the mountains that is currently visible from the pilot's perspective.

Another set of improvements regards the representation that was chosen for HoloAssist's API. It currently has two main issues that could be addressed:

1. The local rotation of a `GeoFixedVertex` can only be represented as Euler angles. Since quaternions are a common representation for rotations (and offer a number of advantages), it is possible to run into the situation in

which an HoloAssist App computes a rotation quaternion, converts it to Euler angle and sends it via the API just for HoloAssist to immediately convert it back to a quaternion. This could be avoided by allowing to represent the local rotation with a quaternion also in the API. However, since quaternions tend to be less known outside the field of computer graphics, the possibility of using Euler angles should not be removed.

2. A lot of information is currently repeated. In many cases, most vertices share either the same topocentric origin or the same local rotation, but currently every vertex has to write this information explicitly. This could be improved by allowing users to omit some of these fields on some of the vertices: the missing field would either default to a zero value or could use the same value as the last vertex that specified them.

Another improvement regarding the API could be to move away from its current transport layer, UDP. It was chosen due to its simplicity and to the fact that it is already commonly used in flight simulators. However, this results in disadvantages like the necessity of implementing a custom wire protocol and the difficulty in relaying back to the caller a response on whether the API invocation was successful or produced some errors. Switching from UDP to another protocol, like TCP, and evaluating higher-level protocols like HTTP could significantly improve quality of the API (although at the cost of additional complexity for users with limited software development experience).

A final update would be to enable HoloAssist to display a third type of augmentations, head-fixed augmentations. These would allow to display information like speed and altitude, which have been requested by most of the pilots that evaluated the augmentations for the Innsbruck approach and would therefore likely prove to be very useful.

Besides the improvement for HoloAssist, the suggestions reported in section 4.3 offer a number of ideas for additional HoloAssist Apps that could be developed and tested in future projects.

Appendices

A. The HoloAssist API

All the previous chapters focus on the inner workings of HoloAssist, but gloss over the exact technicalities of how its API works, as they can be seen as implementation details. These details are available here, and this section constitutes the documentation for the currently available API.

HoloAssist uses an UDP-based API: each packet is an individual API command which, when received, triggers a certain action. Each packet contains a string that consists of a single, serialized JSON[65] object. This JSON object always contains a `type` field, which identifies what action it will trigger in HoloAssist. The other fields differ across different types of commands. The commands can be split in two big groups: those that act on geo-fixed augmentations and those that instead target plane-fixed augmentations. These groups are described in the following sections.

The last part of HoloAssist's API is the way in which simulator status updates are dispatched to it, which however has already been described in details in subsection 2.3.3 (in particular Table 2.1).

A.1. Commands for geo-fixed augmentation

Displaying a geo-fixed augmentation entails creating a GF-ELM and creating its vertices and indices. These changes are not applied immediately to what is shown to the user, but need to be committed first with a dedicated command: this allows to partially/slowly update the mesh without disrupting the user experience and then, when all the updates are done, immediately show the new desired result. Creating a new GF-ELM can be done with the JSON message shown in Figure A.1. After creation, vertices can be added or modified with the JSON message shown in Figure A.2. Indices can be handled in a similar way, shown in Figure A.3. There currently is no way of deleting already submitted vertices or indices, as that would make the whole API significantly more complicated (e.g. what should happen to the index list if some of the referenced vertices are deleted?). When the list of vertices and indices is complete, the changes can be committed with the command shown in Figure A.4: only at this point, the mesh shown to the user is actually updated. Before being able to see something, a

```
{
    "type": "CREATE_MESH",
    "id": "example_mesh_id",
    "interpolateOnCommit": true,
    "interpolatedSegmentMaxLengthMeters": 50
}
```

Figure A.1.: Creates a new GF-ELM. The `id` field is used to identify the created mesh in subsequent commands and the other two fields allow control over the automatic interpolation mesh interpolation described in subsection 2.3.5.

last step is needed: the mesh needs to be activated with the command shown in Figure A.5. The activation status of a GF-ELM can be toggled to temporarily hide it from the user's sight without deleting it, and by default a newly created mesh is deactivated. Finally, when a mesh is not needed anymore, it can be deleted with the command shown in Figure A.6.

A.2. Commands for plane-fixed augmentation

Commands for plane-fixed augmentations are very similar to those for geo-fixed augmentations, with just a few differences:

1. The `type` of each command is prefixed with `PF_`, to clearly distinguish which kind of meshes are being manipulated. For example, `CREATE_MESH` becomes `PF_CREATE_MESH`. Except for what is listed below, the remaining part of the JSON message remains unchanged.

2. When creating a mesh, the parameters regarding the interpolation are replaced with fields specifying the pose of the plane-fixed augmentation in the airplane mesh coordinate system, as shown in Figure A.7.

3. The `vertices` field of a `PF_SET_MESH_VERTICES` command is a list of objects like the one shown in Figure A.8, and not of `GeoFixedVertices`.

Moreover, an additional command to change the position and rotation of the mesh is available, as shown in Figure A.9.

```
{
    "type": "SET_MESH_VERTICES",
    "id": "example_mesh_id",
    "startIndex": null,
    "vertices": [{
        "originWgs": {
            "latitudeRadians": 0.844094,
            "longitudeRadians": 0.205382,
            "altitudeMeters": 453
        },
        "color": [0.0, 1.0, 0.0, 1.0],
        "localPositionMeters": [0,0,0],
        "localRotationRadians": [0,0,0]
    }]
}
```

Figure A.2.: Adds or modifies a list of `GeoFixedVertex`. The `id` field specifies which mesh will be modified by this command. `startIndex` can be either `null`, in which case the vertices will be added at the end of the current list, or an integer indicating a zero-based index *i* inside the current list of vertices, in which case this command will replace the *already existing* vertices of the mesh starting from *i* until all the vertices contained in the command are processed. In this second case, no new vertices will be created. The field `vertices` represents a list of `GeoFixedVertex`: the details of such structure are available in subsection 2.3.2. `originWgs` describes the topocentric origin of the vertex.

```
{
    "type": "SET_MESH_INDICES",
    "id": "example_mesh_id",
    "startIndex": null,
    "indices": [1,2,2,3]
}
```

Figure A.3.: Extends or modifies the list of indices of an GF-ELM. The `id` field specifies which mesh will be modified by this command. `startIndex` can be either `null`, in which case the indices will be added at the end of the current list, or and integer indicating a zero-based index *i* inside the current list of indices, in which case this command will replace the *already existing* indices of the mesh starting from *i* until all the indices contained in the command are processed. In this second case, no new indices will be created. The field `indices` consists of a list of integers, each of which will become an element in the list of indices and is the index of a vertex in this mesh's vertex list. The list of indices must contain an even amount of indices, as each pair denotes a line.

```
{
    "type": "COMMIT_MESH_CHANGES",
    "id": "example_mesh_id"
}
```

Figure A.4.: Commits the changes and displays them to the user. The `id` field is used to identify the mesh whose changes should be committed.

```
{
    "type": "SET_MESH_ACTIVE",
    "id": "example_mesh_id",
    "active": true
}
```

Figure A.5.: Changes the activation status of a GF-ELM. The `id` field is used to identify which mesh should be edited and the `active` field specifies whether the mesh should be activated or deactivated.

```
{
    "type": "DELETE_MESH",
    "id": "example_mesh_id"
}
```

Figure A.6.: Completely deletes a GF-ELM. The `id` field is used to identify which mesh should be deleted.

```
{
    "type": "PF_CREATE_MESH",
    "id": "example_mesh_id",
    "originPositionMeters": [0.0, 0.0, 0.0],
    "originRotationRadians": [0.0, 0.0, 0.0]
}
```

Figure A.7.: Creates a new plane-fixed mesh. The `id` field is used to identify the created mesh in subsequent commands and the other two fields will be used as position and orientation of the Unity object that backs this mesh. The coordinate of the origin position are with respect to the airplane mesh's coordinate system.

```
{
    "position": [0.0, 0.0, 0.0],
    "color": [0.0, 0.0, 0.0]
}
```

Figure A.8.: An example of a vertex for a plane-fixed mesh, to be used as part of the `vertices` field of a `PF_SET_MESH_VERTICES`.

```
{
    "type": "PF_UPDATE_MESH_ORIGIN",
    "id": "example_mesh_id",
    "originPositionMeters": [0.0, 0.0, 0.0],
    "originRotationRadians": [0.0, 0.0, 0.0]
}
```

Figure A.9.: Command to change the origin and rotation of the plane-fixed mesh.

B. Implementation issues

Developing a Hololens app in Unity entails programming in a weird environment: Unity's code is written in C#, which is then compiled to CIL (Common Intermediate Language), an intermediate language that is then usually executed by the .NET virtual machine. However, a game engine trying to run on a platform as constrained as the Hololens cannot afford the overhead imposed by this kind of virtual machines, and is therefore forced to develop alternative solutions. Unity Technologies created IL2CPP[66], which takes as input the generated CIL bytecode and converts it to C++, which is then compiled to native code for the desired platform, in this case the ARM processor of the Hololens.

All these different compilation steps (besides causing lengthy compilation times) have the concrete result of seriously degrading the quality of the logs that are produced when running a Unity application on the Hololens, as shown in Figure B.1 and Figure B.2. This is particularly true when compiling in "Release" mode, which however is often necessary because the performance of "Debug" build makes them unusable.

Another source of frustration has been the difference in the kernel interface that is available in the Unity editor (while developing and debugging the application) and on the Hololens. The HMD only exposes the Windows Universal Platform (UWP) API, originally developed for Windows Mobile, and that should (very theoretically) replace all the other Windows API. Besides trivial nuisances like its capabilities system, this forces developers to deal with two completely different network APIs (`UdpClient` for "normal Win-

```
Stacktrace:
Stacktrace is not supported on this platform._CRT_ASSERT caught:
'''C:\holo\Build\Main\Il2CppOutputProject\IL2CPP\libil2cpp\vm\Class.
    ↪ cpp(641) : Assertion failed: 0 && "Class::IsAssignableFrom"
```

Figure B.1.: An example of logs produced by running HoloAssist compiled in release mode on the Hololens. An assertion deep in the internals of IL2CPP is triggered, with no clear connection to the C# code originally written for the application.

```
Exception thrown at 0x770794AB (KernelBase.dll) in holo-assist.exe:
    ↪ WinRT originate error - 0x80072726 : 'An invalid argument was
    ↪  supplied.'.
Exception thrown at 0x770794AB (KernelBase.dll) in holo-assist.exe:
    ↪ WinRT originate error - 0x80072726 : 'An invalid argument was
    ↪  supplied.'.
Exception thrown at 0x770794AB (KernelBase.dll) in holo-assist.exe:
    ↪ WinRT originate error - 0x80072726 : 'An invalid argument was
    ↪  supplied.'.
```

Figure B.2.: An another example of logs produced by running HoloAssist compiled in release mode on the Hololens.

dows" and `Windows.Networking.Sockets.DatagramSocket` for UWP), one of which can only be tested on device and has to be debugged with poor logs and long feedback loops (due to the lengthy compilation process). This is further complicated by the fact that the two different APIs use different concurrency methods (`System.AsyncCallback` vs `System.Threading.Tasks`), which have to be integrated with Unity's own way of dealing with concurrency, which by itself is not straightforward. A more detailed description is available in the comments embedded in HoloAssist's code, an excerpt of which is shown in Figure B.3. All of this, combined with weird known bugs like the one shown in Figure B.4, resulted in spending more than three days of work to be able to receive and send UDP packets on device, a task which takes less than half an hour on virtually every other development stack.

A couple of additional honorable mentions are worth mentioning:

1. Due to the path length limitation in Windows and the absurdly long filesystem names that Unity generates for its installed packages, WLT requires HoloAssist's project folder to have a short name. Its absolute path must be shorter than *eleven characters*[67].

2. Microsoft's QR code detection library has a known bug which causes it to break when used on an UWP platform and the camera permission has not been already granted to the app. WLT's developers (members of the same company) encountered the same problem an developed a workaround[68], which has been also been added to HoloAssist.

3. Locale setting affects the way in which float variables are printed. In particular, in some locale the dot is used as a decimal separator (`1.2`) and in other the comma is used (`1,2`). The Wavefront OBJ file format

requires the dot as decimal separator. In HoloAssist's case, the Unity editor runs on a computer with English locale, which uses the dot as decimal separator: during testing, therefore, the code that was written to serialize OBJ files worked flawlessly. The Hololens on which HoloAssist was tested, however, has a German locale, which uses the comma as a decimal separator, resulting in corrupted OBJ files. This was particularly difficult to spot, because software like Meshlab did not flat out reject the file and produced instead a "wrong" visual result.

4. At some point, somewhere, some compiler introduces a bug that causes a segmentation fault. This happens when data is passed between the thread that does the QR code detection and the main Unity thread that processes such results. Explicitly copying the data seems to solve the problem, leading to the implementation shown in Figure B.5. Concurrency is hard.

Nevertheless, the Hololens remains a very capable platform, and despite these problems, its integration with Unity noticeably simplifies and speeds up the development of AR experiences.

```
void Update()
{
    /*
        This is a bit "weird". In summary, Unity has its own
        main thread, where it runs all the Start() and
        Update() of the various components for each
        GameObject. The ‘DatagramSocket‘ that is used to
        receive UDP packet on UWP (= on the Hololens) uses
        C# async, which basically means that ‘Task‘s can
        technically be executed in additional worker
        threads, different from the main one from Unity.
        Now. I am not sure that this is the precise cause,
        but getting this UDP receiver to work on the
        Hololens was a pain, and a certain point during
        debugging I choose to be extra-safe with thread
        safety, which leads to the current implementation.

        The idea is that, whenever a packet is received,
        it is enqueued on the _ExecuteOnMainThreadQueue,
        which, being a ‘ConcurrentQueue‘ should work
        regardless of threads. Then, Unity executes
        UDPReceiver::Update (on its main thread): here
        the enqueued packet are dequeued, processed,
        and the state of the Unity application updated.
     */

    while (!_ExecuteOnMainThreadQueue.IsEmpty)
    {
        if (_ExecuteOnMainThreadQueue.TryDequeue(out Action a))
        {
            a.Invoke();
        }
    }
}
```

Figure B.3.: An extract from the HoloAssist's code that deals with the network API.

When running on the Hololens, there is a bug somewhere in the UDP stack that *requires you to send any data out before being able to receive messages*. I thought I was able to get this finding working with `UdpClient`, but only got a few reads out and it stopped - didn't explore further.

As discovered here & gist.

Code verbatim from above gist:

```
var portStr = "3109";
socket = new DatagramSocket();
socket.MessageReceived += _Socket_MessageReceived;
await _Socket.BindServiceNameAsync(portStr);

// Unclear the need for this
await Task.Delay(3000);

// send out a message, otherwise receiving does not work ?!
var outputStream = await _Socket.GetOutputStreamAsync(new HostName("255.255.255.255
DataWriter writer = new DataWriter(outputStream);
writer.WriteString("Hello World!");
await writer.StoreAsync();
```

Share Edit Follow Flag

edited Nov 13 '19 at 10:30

answered Nov 13 '19 at 9:55

fionbio

3,425 ● 1 ● 20 ● 35

Figure B.4.: A weird known bug[69]. UDP sockets on the Hololens require to send data before being able to receive them. The original bug was originally reported on the MSDN forum in 2017[70].

```
private void OnQRCodeUpdated(object sender, QRCodeUpdatedEventArgs
    ↪ args)
{
    var data = new QRCodeData();
    data.Id = new Guid(args.Code.Id.ToString());
    data.Data = new string(args.Code.Data.ToCharArray());
    data.SpatialGraphNodeId = new Guid(args.Code.SpatialGraphNodeId.
        ↪ ToString());
    data.PhysicalSideLength = args.Code.PhysicalSideLength;
    _executeOnMainThreadQueue.Enqueue((EventType.UPDATED, data));
}
```

Figure B.5.: This code is executed on the QR code recognition thread and dispatches the data acquired from the QR code detection to Unity's main thread. Removing the explicit copying from the `args` parameter results in a segmentation fault on the Hololens, which shouldn't be possible in a managed language like C#. This is likely caused by a bug in some step of the conversion from C# to native code, but given the complexity of this process it was decided to not investigate this bug further.

Glossary

East North Up coordinate reference system A commonly used topocentric coordinate reference system. See subsection 2.3.1 . 89

airplane ENU CRS An East North Up coordinate reference system centered at the current airplane's geographical position (as computed by the flight simulator) . 26–31, 35

EPSG:4978 A commonly used Earth-centered, Earth-fixed coordinate reference system. See subsection 2.3.1 . 20–23

EPSG:4979 A commonly used coordinate reference system based on latitude, longitude and altitude. See subsection 2.3.1 . 20–24, 27

geo-fixed augmentation One of the two types of augmentations displayed by HoloAssist. They consist in an arbitrary 3D mesh positioned at some geographical point. HoloAssist does all the computation required to shown them in the correct geographical position when looking from the flight simulator cockpit's window . 7, 9, 11, 13, 17, 19, 20, 24, 26, 30, 32, 33, 36–38, 41–43, 54, 75, 76, 87

HoloAssist A Hololens application developed in Unity that allows to easily create augmented reality experiences to be shown in a fixed-platform flight simulator. It exposes a high-level API that can be used to draw geo-fixed augmentation and plane-fixed augmentation . 5–9, 11–13, 15, 17, 19, 20, 22–24, 26, 27, 29, 30, 32, 33, 36, 38, 40, 43–46, 48, 49, 51–56, 60, 63–65, 71, 72, 75, 81–84, 87

HoloAssist App A script, written in an arbitrary programming language, that uses the HoloAssist API to create some augmented reality experience . 5, 7, 8, 25, 45, 52–54, 56, 58–64, 72

plane-fixed augmentation One of the two types of augmentations displayed by HoloAssist. They consist in an arbitrary 3D mesh positioned at some fixed point inside the flight simulator cockpit . 7, 13, 19, 43–45, 48, 54, 75, 76, 87

remote Unity editor A developer tool that allows to move and rotate a virtual 3D object in a Unity application running on the Hololens without having to recompile/redeploy the project. It also allows to retrieve the current pose of a `GameObject` . 17, 41, 46, 48

Acronyms

API application programming interface. 1, 2, 5–7, 9, 11–13, 23–27, 38, 39, 43–45, 51–56, 60, 63, 65, 71, 72, 75, 81, 82, 84

AR augmented reality. v, 1–6, 8, 10, 37, 38, 48, 51, 54, 62–66, 69, 71, 83

CAD computer aided design. 13, 19

CRS coordinate reference system. 20–23, 26–31, 35, 38, 39, 60, 61, 87

ECEF Earth-centered, Earth-fixed. 22, 23, 27, 38, 39, 42, 60, 61, 87

ENU East North Up. 21–24, 26–31, 35, 87

eVTOL electric vertical take-off and landing aircraft. 4, 66, 70

FOV field of view. 1

GF-ELM geo-fixed external line mesh. 24, 26, 27, 31, 36–38, 52, 75, 76, 78, 79

GIS geographic information system. 56

HMD head-mounted display. 1, 2, 4, 64, 66, 67, 81

HUD head-up display. 4

ICP iterative closest point. 17

IMU inertial measurement unit. 5

PEP projection eye point. 31, 33, 34, 36, 41, 48, 65

TCAS traffic collision avoidance system. 70

UDP user datagram protocol. 27, 28, 48, 49, 60, 72, 75, 82, 85

VR virtual reality. 1, 63–65

WLT World Locking Tools library. 9–12, 82

Bibliography

[1] D. Kocian. "A Visually-Coupled Airborne Systems Simulator (VCASS) - An Approach to Visual Simulation". In: *undefined* (1977). URL: https://www.semanticscholar.org/paper/A-Visually-Coupled-Airborne-Systems-Simulator-An-to-Kocian/7012fc9e83f931df281d6143f3758f1ca2f67956 (visited on 01/17/2022).

[2] Google LLC. *ARCore*. Build new augmented reality experiences that seamlessly blend the digital and physical worlds. URL: https://developers.google.com/ar (visited on 01/17/2022).

[3] Apple, Inc. *ARKit*. ARKit Overview - Augmented Reality. URL: https://developer.apple.com/augmented-reality/arkit/ (visited on 01/17/2022).

[4] Microsoft Corporation. *Microsoft HoloLens*. Microsoft HoloLens - Mixed Reality Technology for Business. URL: https://www.microsoft.com/en-us/hololens (visited on 01/17/2022).

[5] Meta Platforms, Inc. *Spark AR Studio*. Spark AR Studio - Create Augmented Reality Experiences | Spark AR Studio. URL: https://sparkar.facebook.com/ar-studio (visited on 01/17/2022).

[6] Niantic, Inc. *Pokémon GO*. URL: https://pokemongolive.com/ (visited on 01/05/2022).

[7] Unity Technologies. *Unity Real-Time Development Platform | 3D, 2D VR & AR Engine*. URL: https://unity.com/ (visited on 01/26/2022).

[8] World Wide Web Consortium. *WebXR Device API*. URL: https://www.w3.org/TR/webxr/ (visited on 01/26/2022).

[9] Magic Leap, Inc. *Magic Leap 1*. URL: https://www.magicleap.com/magic-leap-1 (visited on 01/17/2022).

[10] F. Viertler and M. Hajek. "Requirements and Design Challenges in Rotorcraft Flight Simulations for Research Applications". In: 2015. DOI: 10.2514/6.2015-1808.

[11] S. Andress, A. J. M.d, M. Unberath, A. F. Winkler, K. Yu, J. Fotouhi, S. W. M.d, G. M. O. M.d, and N. Navab. "On-the-fly augmented reality for orthopedic surgery using a multimodal fiducial". In: *Journal of Medical Imaging* 5.2 (Jan. 2018). Publisher: SPIE, p. 021209. ISSN: 2329-4302, 2329-4310. DOI: 10.1117/1.JMI.5.2.021209. (Visited on 01/17/2022).

[12] J. Fotouhi. "Interventional 3D Augmented Reality for Orthopedic and Trauma Surgery". In: 16th Annual Meeting of the International Society for Computer Assisted Orthopedic Surgery (CAOS). June 9, 2016.

[13] P. Fallavollita, A. Winkler, S. Habert, P. Wucherer, P. Stefan, R. Mansour, R. Ghotbi, and N. Navab. "Desired-View–controlled positioning of angiographic C-arms". In: *Medical image computing and computer-assisted intervention: MICCAI ... International Conference on Medical Image Computing and Computer-Assisted Intervention* 17 (Pt 2 2014), pp. 659–666. DOI: 10.1007/978-3-319-10470-6_82.

[14] G. Turini, S. Condino, P. D. Parchi, R. M. Viglialoro, N. Piolanti, M. Gesi, M. Ferrari, and V. Ferrari. "A Microsoft HoloLens Mixed Reality Surgical Simulator for Patient-Specific Hip Arthroplasty Training". In: *Augmented Reality, Virtual Reality, and Computer Graphics*. Ed. by L. T. De Paolis and P. Bourdot. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 201–210. ISBN: 978-3-319-95282-6. DOI: 10.1007/978-3-319-95282-6_15.

[15] Assistance Publique Hôpitaux de Paris. *World premiere for mixed reality surgery*. URL: https://healthcare-in-europe.com/en/news/world-premiere-for-mixed-reality-surgery.html (visited on 01/17/2022).

[16] Microsoft Corporation. *Remote Assist*. Remote Assist, Microsoft Dynamics 365. URL: https://dynamics.microsoft.com/en-us/mixed-reality/remote-assist/ (visited on 01/17/2022).

[17] Microsoft Corporation. *Apps, Services, and Solutions for HoloLens 2 | Microsoft HoloLens*. URL: https://www.microsoft.com/en-us/hololens/apps (visited on 01/17/2022).

[18] C. Walko and B. Schuchardt. "Increasing helicopter flight safety in maritime operations with a head-mounted display". In: *CEAS Aeronautical Journal* 12.1 (Jan. 1, 2021), pp. 29–41. ISSN: 1869-5590. DOI: 10.1007/s13272-020-00474-7.

[19] T. H. Tran, F. Behrend, N. Fünning, and A. Arango. "Single Pilot Operations with AR-Glasses using Microsoft HoloLens". In: *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*. 2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC). ISSN: 2155-7209. Sept. 2018, pp. 1–7. DOI: 10.1109/DASC.2018.8569261.

[20] F. Brooks. "No Silver Bullet Essence and Accidents of Software Engineering". In: *Computer* 20.4 (Apr. 1987). Conference Name: Computer, pp. 10–19. ISSN: 1558-0814. DOI: 10.1109/MC.1987.1663532.

[21] C. Walko and N. Peinecke. "Integration and use of an augmented reality display in a maritime helicopter simulator". In: *Optical Engineering* 59.4 (Apr. 28, 2020), p. 1. ISSN: 0091-3286. DOI: 10.1117/1.OE.59.4.043104.

[22] Microsoft Corporation. *HoloLens 2 Moving Platform Mode*. URL: https://docs.microsoft.com/en-us/hololens/hololens2-moving-platform (visited on 01/26/2022).

[23] A. Martin-Gomez, A. Winkler, K. Yu, D. Roth, U. Eck, and N. Navab. "Augmented Mirrors". In: *2020 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*. 2020 IEEE International Symposium on Mixed and Augmented Reality (ISMAR). ISSN: 1554-7868. Nov. 2020, pp. 217–226. DOI: 10.1109/ISMAR50242.2020.00045.

[24] PTC Inc. *Vuforia Enterprise Augmented Reality Software*. URL: https://www.ptc.com/en/products/vuforia (visited on 01/28/2022).

[25] Microsoft Corporation. *World Locking Tools*. URL: https://microsoft.github.io/MixedReality-WorldLockingTools-Unity/README.html (visited on 01/28/2022).

[26] D. G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision* 60.2 (Nov. 1, 2004), pp. 91–110. ISSN: 1573-1405. DOI: 10.1023/B:VISI.0000029664.99615.94. URL: https://doi.org/10.1023/B:VISI.0000029664.99615.94 (visited on 01/28/2022).

[27] P. Alcantarilla, J. Nuevo, and A. Bartoli. "Fast Explicit Diffusion for Accelerated Features in Nonlinear Scale Spaces". In: *BMVC*. 2013. DOI: 10.5244/C.27.13.

[28] K. Hata and S. Savarese. *CS231A Course Notes 1: Camera Models*. URL: http://web.stanford.edu/class/cs231a/course_notes/01-camera-models.pdf (visited on 02/08/2022).

[29] Z. Zhang. "A flexible new technique for camera calibration". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22.11 (Nov. 2000). Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 1330–1334. ISSN: 1939-3539. DOI: 10.1109/34.888718.

[30] P. Labatut, J.-P. Pons, and R. Keriven. "Robust and Efficient Surface Reconstruction From Range Data". In: *Computer Graphics Forum* 28.8 (2009). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2009.01530.x, pp. 2275–2290. ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2009.01530.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2009.01530.x (visited on 02/08/2022).

[31] M. Jancosek and T. Pajdla. "Multi-view reconstruction preserving weakly-supported surfaces". In: *CVPR 2011*. CVPR 2011. ISSN: 1063-6919. June 2011, pp. 3121–3128. DOI: 10.1109/CVPR.2011.5995693.

[32] AliceVision. *Meshroom - 3D Reconstruction Software*. URL: https://alicevision.org/#meshroom (visited on 01/28/2022).

[33] A. Knapitsch, J. Park, Q.-Y. Zhou, and V. Koltun. "Tanks and temples: benchmarking large-scale scene reconstruction". In: *ACM Transactions on Graphics* 36.4 (July 20, 2017), 78:1–78:13. ISSN: 0730-0301. DOI: 10.1145/3072959.3073599. URL: https://doi.org/10.1145/3072959.3073599 (visited on 01/28/2022).

[34] Apple, Inc. *iPad Pro 12.9 (5th generation)*. Apple. URL: https://www.apple.com/ipad-pro/specs/ (visited on 01/28/2022).

[35] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli, and G. Ranzuglia. "MeshLab: an Open-Source Mesh Processing Tool". In: *Eurographics Italian Chapter Conference* (2008). Artwork Size: 8 pages ISBN: 9783905673685 Publisher: The Eurographics Association, 8 pages. DOI: 10.2312/LOCALCHAPTEREVENTS/ITALCHAP/ITALIANCHAPCONF2008/129-136. URL: http://diglib.eg.org/handle/10.2312/LocalChapterEvents.ItalChap.ItalianChapConf2008.129-136 (visited on 01/28/2022).

[36] J. Zhang, Y. Yao, and B. Deng. "Fast and Robust Iterative Closest Point". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021). Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 1–1. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2021.3054619.

[37] Blender Foundation. *The Blender Project - Free and Open 3D Creation Software*. URL: https://www.blender.org/ (visited on 01/28/2022).

[38] IOGP Geomatics Committee. *EPSG Geodetic Parameter Dataset*. URL: https://epsg.org/home.html (visited on 01/29/2022).

[39] *EPSG:4326*. URL: http://epsg.io/4326 (visited on 03/05/2022).

[40] M. Zintl. "Design and Implementation of a Modular Primary Flight Display for a Flight Simulator". Master's Thesis in Informatics. Munich: Technical University of Munich, July 15, 2020. 167 pp.

[41] *Local tangent plane coordinates*. In: *Wikipedia*. Page Version ID: 1063628292. Jan. 4, 2022. URL: https://en.wikipedia.org/w/index.php?title=Local_tangent_plane_coordinates&oldid=1063628292 (visited on 01/30/2022).

[42] *PyProj - Python interface to PROJ*. URL: https://pyproj4.github.io/pyproj/stable/ (visited on 01/30/2022).

[43] QGIS Development Team. *QGIS - A Free and Open Source Geographic Information System*. URL: https://qgis.org/en/site/ (visited on 01/30/2022).

[44] P. Karlton and M. Fowler. *TwoHardThings*. URL: https://martinfowler.com/bliki/TwoHardThings.html (visited on 01/30/2022).

[45] Microsoft Corporation. *Hologram stability - Mixed Reality*. URL: https://docs.microsoft.com/en-us/windows/mixed-reality/develop/advanced-concepts/hologram-stability (visited on 01/30/2022).

[46] *Quaternions and spatial rotation*. In: *Wikipedia*. Page Version ID: 1067865383. Jan. 25, 2022. URL: https://en.wikipedia.org/w/index.php?title=Quaternions_and_spatial_rotation&oldid=1067865383 (visited on 01/30/2022).

[47] *Distance from a point to a line*. In: *Wikipedia*. Page Version ID: 1061784220. Dec. 23, 2021. URL: https://en.wikipedia.org/w/index.php?title=Distance_from_a_point_to_a_line&oldid=1061784220 (visited on 01/30/2022).

[48] Khronos Group. *Depth Test - OpenGL Wiki*. URL: https://www.khronos.org/opengl/wiki/Depth_Test (visited on 01/31/2022).

[49] R. Gabriel. *The Rise of 'Worse is Better'*. 1991. URL: https://web.stanford.edu/class/cs240/old/sp2014/readings/worse-is-better.html (visited on 01/31/2022).

[50] Japan Industrial Standards. *MicroQR Code. JIS X 0510 standard.* URL: https://www.qrcode.com/en/codes/microqr.html (visited on 02/03/2022).

[51] D. Marsh. *The Paved Road at Netflix: At the junction of freedom and responsibility*. Austin, Texas. URL: https://www.oreilly.com/library/view/oscon-2017/9781491976227/video306724.html (visited on 02/03/2022).

[52] *The Python Programming Language*. URL: https://www.python.org/ (visited on 02/03/2022).

[53] *PyPI · The Python Package Index*. URL: https://pypi.org/ (visited on 02/03/2022).

[54] *NumPy - The fundamental package for array computing with Python*. Version 1.22.1. URL: https://www.numpy.org (visited on 02/03/2022).

[55] *SciPy - Scientific Library for Python*. Version 1.7.3. URL: https://www.scipy.org (visited on 02/03/2022).

[56] B. Rhodes. *Skyfield - Elegant astronomy for Python*. Version 1.41. URL: http://github.com/brandon-rhodes/python-skyfield/ (visited on 02/03/2022).

[57] *Matplotlib - Python plotting package*. Version 3.5.1. URL: https://matplotlib.org (visited on 02/03/2022).

[58] *PyTorch - A Python deep learning framework*. URL: https://pypi.org/project/torch/ (visited on 02/03/2022).

[59] *Stack Overflow Developer Survey 2021*. Stack Overflow. URL: https://insights.stackoverflow.com/survey/2021/#programming-scripting-and-markup-languages (visited on 02/03/2022).

[60] VACC Austria. *LOWI Charts*. URL: https://www.vacc-austria.org/index.php?page=content/chartlist&icao=lowi (visited on 02/07/2022).

[61] H. Butler, M. Daly, A. Doyle, S. Gillies, T. Schaub, and S. Hagen. *The GeoJSON Format*. Request for Comments RFC 7946. Num Pages: 28. Internet Engineering Task Force, Aug. 2016. DOI: 10.17487/RFC7946. URL: https://datatracker.ietf.org/doc/rfc7946 (visited on 02/07/2022).

[62] *geojson*. URL: https://pypi.org/project/geojson/ (visited on 02/07/2022).

[63] Land Kärnten. *Digitales 10m Geländemodell aus Airborne Laserscan Daten in der Projektion EPSG:31287 (Lambert)*. URL: https://www.data.gv.at/katalog/dataset/b5de6975-417b-4320-afdb-eb2a9e2a1dbf (visited on 02/07/2022).

[64] The Open Geospatial Consortium. *OGC GeoTIFF Standard*. 2019. URL: https://earthdata.nasa.gov/esdis/eso/standards-and-references/geotiff (visited on 02/07/2022).

[65] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. Request for Comments RFC 8259. Num Pages: 16. Internet Engineering Task Force, Dec. 2017. DOI: 10.17487/RFC8259. URL: https://datatracker.ietf.org/doc/rfc8259 (visited on 02/07/2022).

[66] Unity Technologies. *Unity Manual: IL2CPP Overview*. URL: https://docs.unity3d.com/Manual/IL2CPP.html (visited on 02/07/2022).

[67] Microsoft Corporation. *Initial setup of World Locking Tools*. URL: https://microsoft.github.io/MixedReality-WorldLockingTools-Unity/DocGen/Documentation/HowTos/InitialSetup.html#a-warning-note-on-installation-path-length (visited on 02/07/2022).

[68] M. Finch. *QR codes don't scan on first run*. GitHub. URL: https://github.com/microsoft/MixedReality-WorldLockingTools-Samples/issues/20 (visited on 02/07/2022).

[69] *Sockets - Hololens UDP Server never receives message*. Stack Overflow. URL: https://stackoverflow.com/questions/44034118/hololens-udp-server-never-receives-message (visited on 02/07/2022).

[70] M. Osthege. *DatagramSocket.MessageReceived does not trigger on incoming broadcast messages*. 2017. URL: https://social.msdn.microsoft.com/Forums/en-US/41fa63c9-c19f-4fa3-8349-fad8e465af3c/uwp-datagramsocketmessagereceived-does-not-trigger-on-incoming-broadcast-messages?forum=wpdevelop (visited on 02/07/2022).

Lo duca e io per quel cammino ascoso
intrammo a ritornar nel chiaro mondo;
e sanza cura aver d'alcun riposo,

salimmo sù, el primo e io secondo,
tanto ch'i' vidi de le cose belle
che porta 'l ciel, per un pertugio tondo.

E quindi uscimmo a riveder le stelle.

– Dante Alighieri,
on finishing a master degree at TUM