Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Living Within an Augmented Reality World

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Daniel Hernán Hidalgo Aliena

Tutor: Dr. Ramón Mollá Vayá

Tutor externo: Dr. Nabil Aouf

2016/17

Resumen

Con los rápidos avances en Realidad Virtual y Realidad Aumentada en los últimos años, y el enfoque de las grandes compañías en estas tecnologías, es crucial que su potencial y sus nuevas aplicaciones se evalúen mientras todavía están creciendo. Este trabajo resume la evolución de las tecnologías de Realidad Virtual y Realidad Aumentada y sus aplicaciones a lo largo de los años, y explora aplicaciones potenciales que no se han discutido ampliamente.

Proponemos un sistema que combina las tecnologías de Realidad Virtual y Realidad Aumentada, en una experiencia de Virtualidad Aumentada. Se pilotará un vehículo volador en un entorno controlado, que podría estar capturando video con una cámara montada. Mientras tanto, un usuario está inmerso en una copia virtual del entorno real gracias a la Realidad Virtual. Las tecnologías de realidad aumentada pueden usar la transmisión de video desde el entorno real para aumentar el entorno virtual en tiempo real con objetos predeterminados.

El sistema se divide en tres módulos: un módulo de simulación creado en Unity3D, que se encarga del entorno virtual y la realidad virtual. Un módulo de seguimiento de drones, que rastrea un dron usando la captura de movimiento basada en el marcador OptiTrack. Finalmente, un módulo de seguimiento de objetos, que rastrea objetos de un video con una técnica de seguimiento sin marcadores con modelo, basada en ORB y PnP.

La integración de los diferentes módulos fue exitosa, y la aplicación final puede aumentar el entorno de Unity en tiempo real al rastrear el dron y colocarlo en una versión virtual del entorno, y ubicar los objetos rastreados correctamente en el entorno. Se diseñó un esquema de controles intuitivo, un sistema extensible y un entorno de trabajo personalizable. El resultado es una interfaz intuitiva para la navegación espacial, que podría ampliarse para aceptar múltiples fuentes, información térmica / IR o incluso generar el entorno dinámicamente con SLAM.

Palabras clave: Realidad Virtual, Realidad Aumentada, Virtualidad Aumentada, Telepresencia, Navegación, Tracking por marcadores, Tracking sin marcadores, Estimación de pose 3D

Abstract

With the rapid advancements in Virtual Reality and Augmented Reality in the recent years, and the focus of big companies on these technologies, it is crucial that their potential and new applications are assessed while they are still growing. This thesis summarises the evolution of Virtual Reality and Augmented Reality technologies and their applications throughout the years, and explores potential applications that have not been extensively discussed.

We propose a system that mixes Virtual Reality and Augmented Reality technologies, into an Augmented Virtuality experience. A flying vehicle is being piloted in a controlled environment, which could be capturing a video feed with a mounted camera. Meanwhile, a user is immersed into a virtual copy of the real environment thanks to Virtual Reality. Augmented Reality technologies can use the video feed from the real environment to augment the virtual environment in real time with predetermined objects.

The system is divided in three modules: a simulation module made in Unity3D, which supports the virtual environment and Virtual Reality. A drone tracking module, which tracks a drone using OptiTrack marker-based motion capture. Finally, an object tracking module, which tracks objects from a simulated video feed with model-based markerless tracking technique based on ORB and PnP.

Integration of the different modules was successful, and the final application is able to augment the Unity environment in real time by tracking the drone and placing it in a virtual version of the environment, and placing tracked objects correctly in the environment. An intuitive control scheme, extensible system and customizable framework were built. The result is an intuitive interface for spatial navigation, which could be extended to accept multiple sources, thermal/IR information, or even generating the environment dynamically with SLAM.

Keywords : Virtual Reality, Augmented Reality, Augmented Virtuality, Telepresence, Navigation, Marker-based tracking, Markerless tracking, 3D pose estimation

Tabla de contenidos

1. Introduction

In recent years, there has been a surge in advancements in the fields of Augmented Reality (AR) and more notably, Virtual Reality (VR). Such are the improvements that in 2016, head-mounted displays (HMD) and VR set-ups have become commercially available worldwide at affordable prices, for the first time. Leading companies are shifting their focus towards integrating their technologies into the world that surrounds us, and exploiting the physical world to create interactions never seen before.

With these advancements also come new opportunities to exploit them, and to find how these technologies fit into our current procedures and workspaces. For many years, the aviation industry has been using virtual reality and CAVE set-ups to train pilots in flight tasks. More recently, the medical industry has started to adopt augmented reality and virtual reality technologies for educational purposes, in order to teach interns and resident doctors.

Figure 1 - CAVE and Augmented Reality in aerospace and medicine

Since AR and VR have now become the center of attention of many drivers of development, it is critical that we explore the potential and applications of these technologies while they are growing.

This thesis will focus on exploring and bringing forward applications of the mix of Virtual Reality and Augmented Reality technologies, in the form of what is called Augmented Virtuality (AV). We will explore the potential of an interface based on these technologies, build and analyse a system that exploits them, and discuss the extensions that can be made into different fields of research.

Originally, the motivation of this thesis was born out of a similar project by Louis-Pierre Bergé at Cranfield University, which involved using VR to examine point cloud data. Building on this idea, and its effectiveness, this thesis aims to show the potential of these technologies in applications tied to the real world.

We will first have a look at the advancements and applications of Virtual Reality, Augmented Reality, and surrounding technologies during the years. Then, a definition of a system exploiting these technologies will be given, with a set of constraints and targets. Afterwards, the methodology for implementing the system will be presented, with a detailed run-down of the different components. Following the methodology, we will present the results given by the system, discuss whether the established goals were met, and accentuate the weaknesses of the system in place. Finally, we will give a short discussion about the possible extensions and applications of the presented system, and the alternative technologies that could help improve it.

2. Literature Review

In this section, we will describe the background surrounding this thesis, and the research behind the key components of the developed application. The different approaches to the technologies used, as well as their applications are described in the following segment.

Virtual Reality

Virtual Reality (VR) is an increasingly used medium for telepresence, with many applications ranging from the academic to entertainment [1]. The term itself was first coined in 1989 by Jaron Lanier of VPL Research, Inc., a pioneer in the commercialisation of VR technologies [2]. A general definition of VR was given by Jonathan Steuer in 1992, where he states: *"A virtual reality is defined as a real or simulated environment in which a perceiver experiences telepresence."* [3]. Telepresence can be understood as *"enabl[ing] people physically located in [a] host location to behave and receive stimuli as though at a remote site"* [4]. However, this is not exactly what VR is trying to achieve. A more appropriate term would be "virtual presence", which is similar to telepresence, but states that the remote site is exclusively virtual [5]. Thus, we understand VR as *"a … simulated environment in which a perceiver experiences [virtual presence]".*

Virtual presence can be achieved by providing different kinds of sensory information, but the main focus of research around VR devices throughout the years has been visual information. The most common and understood form of VR is with the use of Head-mounted displays (HMD), which track the user's head to display a scene at different angles through lenses placed in front of the user's eyes, providing stereo vision. However, virtual presence also requires the user to be able to interact with the simulated environment in order to create a strong sense of presence, and to provide useful applications. The main interaction channels used by humans are locomotion and manipulation, and thus, it is key to read these user inputs in some way [1]. Through the use of body-suits which capture the user's movements, holding tracked devices on each hand, or head-mounted computer vision, this input is nowadays readily available.

The concept of Virtual Reality as we know it goes back at least to 1935, with Stanley G. Weinbaum's short story "Pygmalion's Spectacles" [6]. This was one of the first references to HMD-based VR, showing partial sensory immersion into a virtual world. Ever since, there have been many attempts and reformulations of VR, such as the Heilig's Sensorama [7] and Sutherland's "Ultimate Display" [8], the latter being considered the first VR HMD.

While some of these approaches were conceived with entertainment purposes in mind, Sutherland was also one of the first in describing the potential of VR in an academic background in 1965 [9]. In this short paper he describes how VR can be used to analyse and gain familiarity with new concepts and behaviours. However, before this "Ultimate

Display", Philco had also experimented with HMDs based on magnetic tracking [10], as opposed to Sutherland's mechanical approach. This system was used to monitor another room with a live video feed from a camera, which would move according to the user user's head. Although this technology cannot be called VR by the previous definition; which distinguishes telepresence and virtual presence; it is one of the first to show that HMD-based technologies offer potential beyond entertainment. It is our task to see just how far this potential can extend.

Nowadays, commercial devices like the Oculus Rift [11] and HTC Vive [12] offer HMD-based VR with visual and auditory immersion, as well as locomotive interaction. As the production costs of the technology are decreasing and the hardware is becoming more accessible, it is now possible to quickly find and test new applications for the technology. We can find applications related to training personnel in a controlled environment. In the aviation field, flight simulations are carried out to train pilots without any danger with high degrees of fidelity [13]. In the medical field, VR is also being widely used for anatomy education, due to its interactivity, intuitiveness and non-intrusiveness [14]. Similar to pilot training, surgical training in VR also offers a wide variety of advantages, like greater skill transfer than standard training [15] and box-training [16], and non-intrusiveness following the Minimally-Intrusive Surgery (MIS) philosophy. There are further applications outside training personnel, like acting as a better visualisation interface for 3D animation and modelling programs [17], or improved visual feedback from real-world robots requiring teleoperation [18]. Using VR not only as a platform to lead detailed simulations, but as a tool to offer more natural spatial understanding, has been heavily discussed and is one of the main potentials for applications of VR as of now.

Augmented Reality

With the steady increase in computational power of everyday devices, and the improvements on computer vision techniques, Augmented Reality (AR) has become much more accessible in the recent years. AR can be defined as

"[A]ny system that has the following three characteristics:

1. *Combines real and virtual*
2. *Is interactive in real time*
3. *Is registered in three dimensions"* [19].

In general, visual-based Augmented Reality systems understand real-world geometry and overlay geometrically dependent information on it; whether it is a 3D model of an object or text information; and allows the user to examine that environment in real time. However, this is not restricted to sight only, and just like VR, it can apply to other senses like hearing or smell. It is seen as capable of *"enhanc[ing] a user's perception of and interaction with the real world."* [19]. VR and AR have grown together in the last years, however AR has gotten a head-start due to the affordability of the required hardware.

Ivan Sutherland's "Ultimate Display" [8] is not only one of the first advances in VR and HMDs, but also the first in AR prototypes due to its see-through nature. A survey by D.W.F. van Krevelen and R. Poelman from 2010 describes the different AR supports and technologies available, as well as giving a brief history of the AR advances since Sutherland's first prototypes [20]. In the same paper, the numerous applications of AR that have already been proven to be of interest are described. Such applications involve personal assistance and advertising, spatial navigation, industrial design, maintenance, simulation, medical applications, games, education and training.

Rabbi and Ullah [21] classify current challenges of AR in five categories: performance, alignment, interaction, mobility and visualization. Performance challenges refer to the processing time of the system, with metrics such as frame rate and frame time, delay, CPU load, memory usage. Many factors are involved in this such as 3D model complexity and tracking methods used, as discussed by Wagner and Schmalstieg in 2009 on mobile phones as a potential platform [22]. Alignment challenges refer to proper placement of virtual objects with respect to the real world. If errors happen during the alignment phase, information will be incorrectly rendered, which can be dangerous in critical applications like the medical field [21]. Because of this, calibration and correct registration must be ensured. Interaction challenges look at the different interaction methods between the user and the augmented environment. While in this research we will not be concerned with this since interaction will be made through VR equipment, they are problems that still need to be taken into account nowadays. Mobility challenges involve the portability of the AR system, which we will also not be concerned with ourselves too in depth. We will however look at the expandability of the system and show possible extensions. Lastly, visualization challenges refer to the problems arising when trying to display the visual information to the user. We are not only talking about contrast, resolution and other characteristics intrinsic to the images, but also problems like occlusion in particular [23].

Markerless Tracking

To bring Augmented Reality to life, and to solve some of the usability challenges presented by it, markerless tracking is a very active field of research as of now. Markerless tracking involves the correct alignment of 3D objects on the real environment without the need of any markers placed on the real scene, as opposed to marker-based tracking, which uses easy to detect markers to infer geometry from the real world [24]. Markerless tracking can deal with scenes never visited before, and does not require physically altering the scene, or the object to be tracked with some kind of special marker. Generally, markerless tracking imposes many less restrictions than its counterpart. However, this comes at the cost of increasing the complexity of the problem and its computational cost. We will see the theory behind markerless tracking and show different approaches in literature to solve the problem.

Mathematical background

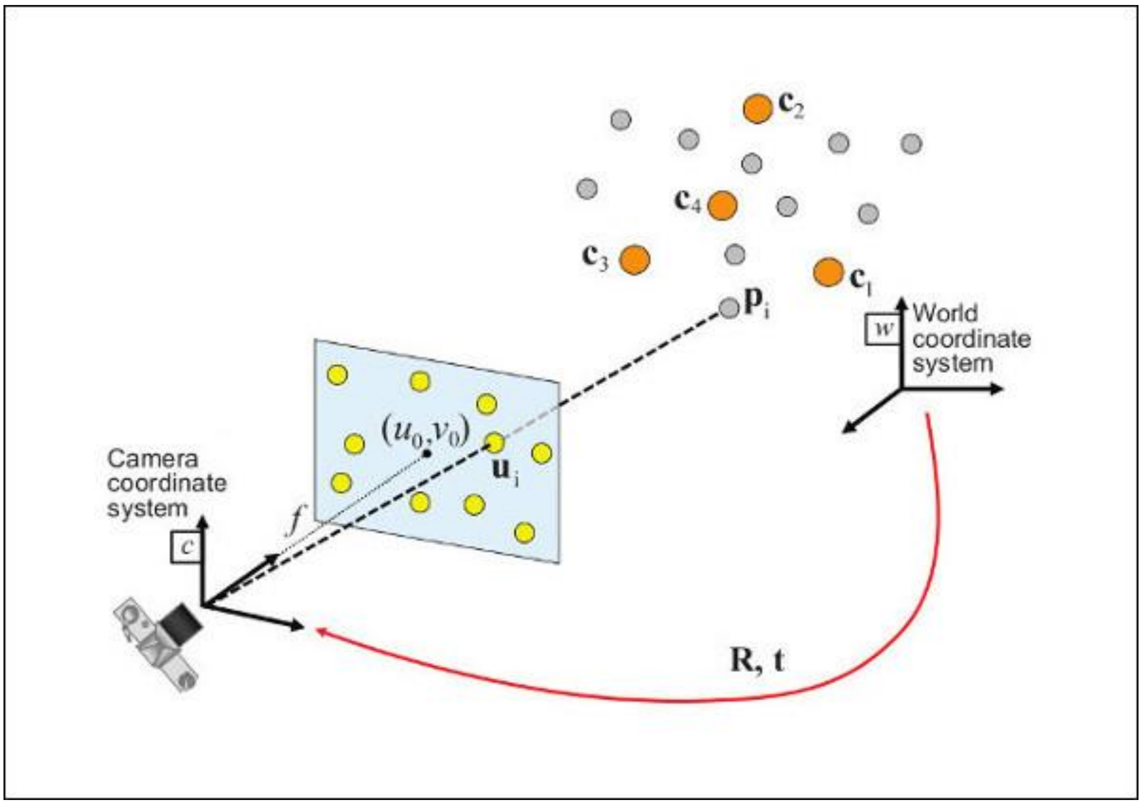


Figure 2 - Perspective camera model, from the OpenCV documentation

To better understand the requirements of tracking a 3-dimensional object from a 2-dimensional image, we have to assume a model for the camera to use. Generally we assume a perspective camera model, as shown in Figure 2.

We first distinguish between the world coordinate system (w_x, w_y, w_z), and the camera coordinate system (c_x, c_y, c_z), which are corresponding after a rotation and translation transform. Such transforms are represented in the rotation $R_{3\times3}$ and translation $t_{3\times1}$ matrices, and the transformation equation from w to c can be seen in equation ***(0-1)***.

$$\begin{bmatrix} c_x \\ c_y \\ c_z \end{bmatrix} = \underbrace{\begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix}}_{R_{3\times3}} \begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix} + \underbrace{\begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}}_{t_{3\times1}} \tag{0-1}$$

By composition we describe this operation from w to c with the $[R|t]_{3\times4}$ matrix, which reduces the equation to the following expression:

$$\begin{bmatrix} c_x \\ c_y \\ c_z \end{bmatrix} = \underbrace{\begin{bmatrix} R_{11} & R_{12} & R_{13} & t_x \\ R_{21} & R_{22} & R_{23} & t_y \\ R_{31} & R_{32} & R_{33} & t_z \end{bmatrix}}_{[R|t]_{3\times4}} \begin{bmatrix} w_x \\ w_y \\ w_z \\ 1 \end{bmatrix} \tag{0-2}$$

With this transformation, we can represent points from the world coordinate system in the camera coordinate system before projecting them onto the camera plane.

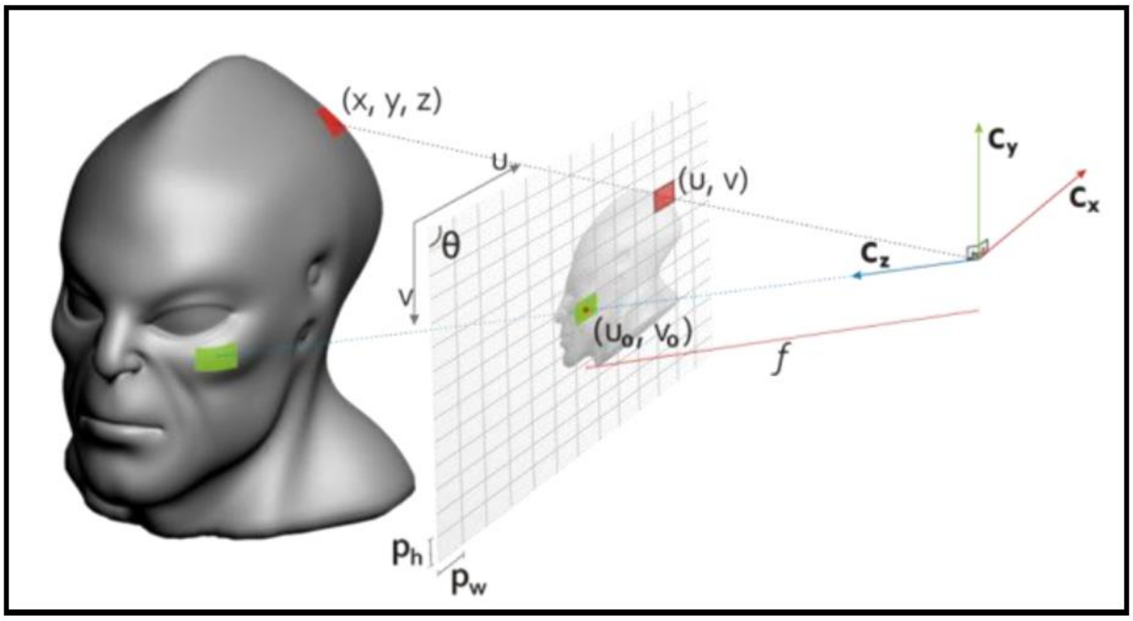


Figure 3 - Projection from 3D coordinates to camera plane (Lima et al. [25])

A point in the camera coordinate system (x, y, z) corresponds to a point (u, v) in the camera plane. The camera plane is described by the following values:

- f – focal length of the camera
- (u_0, v_0, f) – camera plane center in camera coordinate system
- (p_w, p_h) – pixel width and height

This is shown in Figure 3, from which we can also infer the perspective projection conditions:

$$u = \frac{x}{z} f + u_0 \tag{0-3}$$

$$v = \frac{y}{z} f + v_0 \tag{0-4}$$

However, knowing that pixels are not points but well-defined rectangles with pixel width p_w and pixel height p_h, we must take them into account:

$$u = \frac{x}{z}\frac{f}{p_w} + u_0 \tag{0-5}$$

$$v = \frac{y}{z}\frac{f}{p_h} + v_0 \tag{0-6}$$

These projection constraints can be represented into the single projection equation:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f/p_w & 0 & u_0 \\ 0 & f/p_h & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x/z \\ y/z \\ 1 \end{bmatrix} \tag{0-7}$$

Note that equation *(0-7)* assumes $p_w = p_h$. If that is not the case, the ratio between p_w and p_h must be described by an angle θ and factored in:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f/p_w & -\cot(\theta)/p_h & u_0 \\ 0 & f * cosec(\theta)/\, p_h & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x/z \\ y/z \\ 1 \end{bmatrix} \tag{0-8}$$

We call the resulting matrix in equation *(0-8)* the intrinsic parameters of the camera, since they depend on physical properties of the camera and the lens. It is also sometimes called k for calibration. We can now merge equations *(0-2)* and *(0-8)* to obtain the full projection formula:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f/p_w & -\cot(\theta)\,/p_h & u_0 \\ 0 & f * cosec(\theta)/p_h & v_0 \\ 0 & 0 & 1 \end{bmatrix} z^{-1} \begin{bmatrix} R_{11} & R_{12} & R_{13} & t_x \\ R_{21} & R_{22} & R_{23} & t_y \\ R_{31} & R_{32} & R_{33} & t_z \end{bmatrix} \begin{bmatrix} w_x \\ w_y \\ w_z \\ 1 \end{bmatrix} \tag{0-9}$$

With the full projection formula as a basis for 2D-3D projections, we can now define our criteria for estimating the pose of a 3D object from a 2D image.

Using equation *(0-9)*, we can determine the extrinsic parameters of the camera if we have corresponding 2D and 3D points, since we assume the intrinsic parameters are known. Knowing that the extrinsic parameters of the camera relate the position of the camera to the world coordinates, we can know the rotation and translation of an object if we solve the problem.

We call this problem Perspective-n-point (PnP), which involves using n 2D-3D correspondences to estimate the $[R|t]$ matrix. PnP solutions exploit the fact that the intrinsic parameters are constant, thus by knowing 2D and 3D points, $[R|t]$ remains the only unknown. There are different solutions to the PnP problem, such as Direct Linear Transformation (DLT) [26] which solves a linear system of equations made by the correspondences. However, it requires at least 15 of them, which shows its weakness compared to other methods which require less correspondences. While DLT also solves an algebraic error, other methods rely on the minimization of a geometric error, which is more desirable [25].

Although it is technically possible to solve the PnP problem with only 3 points, solving the P3P problem does not yield a unique solution. Using $n \geq 6$ consistently gives unique solutions in most cases.

Many solutions have been proposed to the PnP problem, such as an iterative solution in 2000 by Lu et al. [27], and EPnP, a non-iterative solution in 2007 by Moreno-Noguer et al. [28], which gave an efficient $O(n)$ solution for $n \geq 4$. More recent solutions include RPnP in 2012 by Shiqi et al. [29], OPnP in 2013 by Zheng et al. [30], and more recently REPPnP in 2014 by Ferraz et al. [31], based on early outlier rejection, capable of dealing with over 1,000 points in 5ms. Solutions to PnP strive to achieve high efficiency when dealing with a high number of points, and a high degree of accuracy when dealing with a low number of points. Since PnP solutions tend to show spurious errors, they are usually paired with a RANSAC (Random Sample Consensus) algorithm to minimize the effect of outliers.

Model-based markerless tracking

A great many tools are available to solve the PnP problem, but obtaining the required 2D-3D correspondences is not trivial. We will now see some of the proposed methods in literature.

As described by Teichrieb et al. [32], there are two branches of markerless tracking: Model-based tracking and Structure-from-Motion (SfM). The former involves previous knowledge of the object to be tracked by the system, while the SfM infers information from camera movements. A survey by Lima et al. [25] discusses in detail the different approaches to model-based tracking in particular, which we will focus on rather than SfM.

Inside the model-based tracking branch, there exist in turn two types of tracking methods: recursive tracking and tracking by detection. Recursive tracking gets its name from its time dependence. To track an object in a certain frame, it relies on the tracking results of the previous frames, and merges current information with past information. Tracking by detection, however, does not rely on past information and infers everything from the current frame.

Category	Method	Detection	Processing	Accuracy	Limitations
Recursive tracking	Edge based	No	Low	Jitter	• Fast camera • Cluttered background
	Optical flow based	No	Low	Cumulative errors	• Fast camera • Lighting changes
	Template maching	No	Low	Very Accurate	• Fast camera • Lighting changes • Occlusion
	Interest point based	No	High	Accurate	Fast camera movement
Tracking by detection	View based	Yes	High	Accurate	Restricted range of poses
	Keypoint based	Yes	High	Jitter and drift	None

Table 1 - Model-based tracking methods survey, by Lima et al. [25]

Table 1 shows the results of the survey made by Lima et al. comparing the main tracking techniques that have been studied. They particularly examine an edge based technique based on Wuest et al. [33], an interest point based technique based on Vaccheti et al. [34], and a keypoint based technique based on Skrypnyk and Lowe [35]. We will highlight the main concepts of the three methods, in the modelling phase and the tracking phase:

Method	Model	Tracking
Point Sampling (Edge based)	- 3D model of the object - Control points along edges	- Detect visible edges (Edge-ID) - Match image and model points using past result (ME* algorithm) - Pose estimation with 2D-3D

		correspondences
Interest point based	- 2D keyframes from known camera pose - 2D features (interest points) and corresponding 3D point on model - Normals at each 3D point	- Select keyframe closest to current frame using past results (Mahalanobis or histograms) - Generate intermediary image from keyframe to better match frame (homography) - Match features between frame and intermediary image - Pose estimation with 2D-3D correspondences
Keypoint based	- 3D model of the object - 2D image of the object - Project invariant 2D features onto model for corresponding 3D points	- Extract invariant 2D features - Match 2D features with knowledge-base - Pose estimation with 2D-3D correspondences

* ME = Moving Edges algorithm [36]

Table 2 - Main features of PS, IPB and KPB tracking methods

Note that for the implementation of these methods in Table 2, the pose estimation step with 2D and 3D correspondences was not made using the aforementioned PnP solutions. Instead, a minimisation of the reprojection error of the object from 3D space back to 2D space is made, using optimization algorithms with M-estimators for robustness against outliers. This choice is because PnP methods tend to be sensitive to noise.

Another key point for the success of these methods is the type of 2D feature extraction to be used. It is important for the registration phase to be as efficient as possible, for both the extraction and the matching of 2D features, in order to obtain an acceptable frame rate. Aside from being fast, we also need the features to be scale, rotation, and lighting invariant, since tracked objects can come at any angle and distance. Some of the possible feature descriptors to use are SIFT [37], SURF [38], ORB, an alternative to both presented by Rublee et al. in 2011 [39], or the more recent state-of-the-art LIFT, proposed in 2016 by Yi et al. [40].

Conclusion

While there are several tools available for the construction of VR applications and AR systems, and a great many solutions to most of their technical requirements and roadblocks, there is still much to explore in their applications and how they fit in our current world. In particular, while this has been explored in the past, there is great potential in testing the fusion of these two technologies. The immersiveness and intuitiveness of VR interfaces, and the information extraction of AR technologies, jointly open the door to new possibilities which we hope to show with this thesis.

3. Problem Description

With the potential of VR and AR technologies, we would like to merge these two and extend them into an Augmented Virtuality experience. Augmented virtuality lies in the reality-virtuality continuum, proposed by Paul Milgram in 1994 [41].

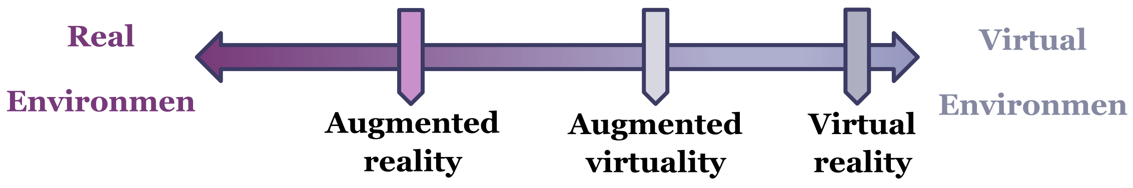


Figure 4 - Reality-virtuality continuum

Augmented virtuality is the counterpart to augmented reality, in a sense. While AR consists in augmenting the real environment with completely virtual elements by integrating them in it, AV consists in augmenting a virtual environment by integrating real objects in it. This can be achieved in various ways, from a simple live video feed from a real camera, to the virtualisation of real objects in 3D.

On another hand, exploration of 3D environments using 2D interfaces like video cameras and sensors presents some inconveniences. Thalmann discussed this in their previously mentioned article [17], which highlights this contradiction of trying to understand a 3D object through 2D interfaces. They are by nature non-intuitive visualisation tools, which require careful user input to perceive details in the way the user requires. Teleoperation with robots in real environments can as well present a problem if the visualisation medium consists of a live video feed from the robot [18]. Visual obstacles like fire or smoke can pose a problem to operators on the other end, which can be ignored by virtual reality.

We will propose an augmented virtuality system that attempts to solve these problems, by defining its requirements, and explain some of its potential applications.

Proposed system

With the help of virtual reality as a natural visualisation interface, and augmented reality solutions as a way to integrate real-world elements into a virtual world, we propose a surveying system by virtual presence.

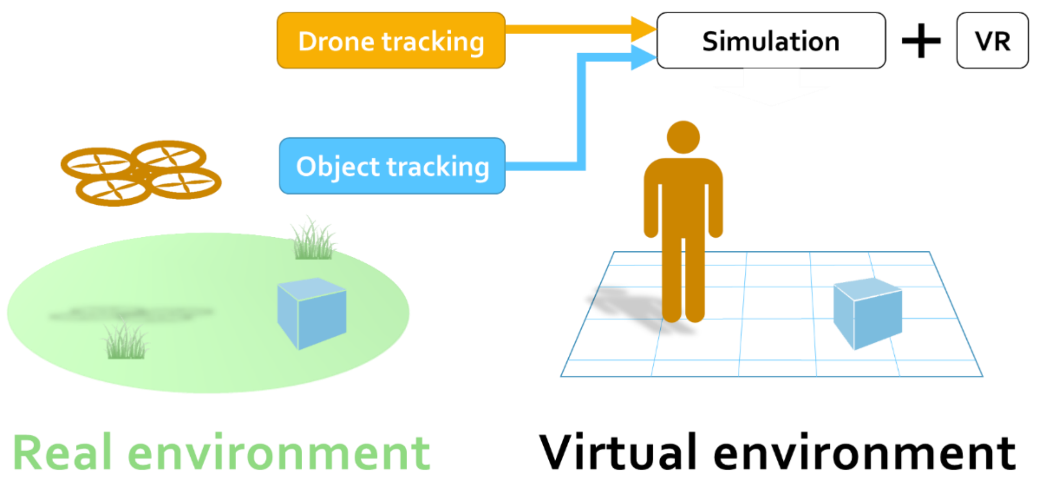


Figure 5 - Surveying system by virtual presence, structure diagram

Figure 5 shows the basic structure of the surveying system, which makes use of a real set-up and a virtual set-up. In the real environment, a vehicle is moving and exploring its surroundings, possibly with the help of an operator controlling it remotely. Meanwhile, the user is placed in a virtual environment which is a recreation of the real environment. The vehicle is equipped with sensors that extract information from the environment, and augment the virtual environment by placing objects or other information in it. Because of the dependence of the virtual objects on the real world, and the display environment being completely virtual, this is an AV application.

With the help of the VR equipment, the user can look around as if they were in the virtual environment. While they can be placed to follow the vehicle's point of view as it is moving, the user can also move around freely to explore the environment at ease, while the vehicle is augmenting the environment in real time. In our case, we will work with a remote-controlled commercial drone, and the way we will augment the environment is by tracking the position of surrounding objects with a monocular camera.

Because of this, the application should consist of three main modules:

- A **drone tracking** module, responsible of keeping track of the vehicle's position in real time, and broadcasting the information to other applications.
- An **object tracking** module, in charge of tracking specific objects using a live video feed from a monocular camera, and broadcasting the information to other applications.
- A **simulation** module, which manages the virtual environment that can be augmented with the information sent by the drone and object tracking modules.

Functional Requirements

- A drone should fly in a closed, controlled environment
- The simulation should show a simplified version of the environment
- The user should be able to observe the virtual environment with VR

- Drone movements in the real environment should be mimicked by moving in the virtual environment
- The user should be able to detach from the drone and explore the environment freely
- An object tracking module should track objects and their position with a monocular camera
- The application should be able to receive data from tracked objects and display it in the virtual environment
- The application should be expandable for future applications
- The application should offer customization options and some freedom to both the user and the programmer
- Modules should have a low degree of coupling upon implementation
- Total failure of the drone and object tracking modules should not affect the simulation module

Non-Functional Requirements

- A framerate of at least 60fps should be achieved for the drone tracking
- There should not be a delay of over 100ms for the drone's movements to be reflected in the simulation
- A framerate of at least 5fps should be achieved for the object tracking
- There should not be a delay of over 100ms for the object tracking module results to be reflected in the simulation
- A framerate of at least 60fps should be achieved for the simulation
- The user should not suffer from motion sickness from the application for short periods of usage
- Main landmarks of the real environment should be kept in its virtual representation
- The user should be able to discern the tracked objects from the environment immediately

With this definition of the system, we will work towards an augmented virtuality proof of concept, which can easily be generalized for other applications in different fields.

Advantages and Contribution

The main advantage of representing the environment in a virtual way, is that it offers a more **natural spatial understanding** of the vehicle's surroundings. In monitoring tasks and navigation applications, it is essential to have a clear understanding of the vehicle's surroundings. Certainly, one of the most natural ways of understanding the environment is to be in the environment and explore it oneself.

One similar application in the past has proven to be of interest, proposed by Ott et al. [42], and extended by Righetti et al. in 2007 [43]. In these papers, a monocular camera is mounted on a R/C blimp, which can be accessed through a HMD. The former focuses on providing a more natural interface to view the video feed provided by the camera, with the use of eye-tracking technology. The user wearing the HMD controls the camera's orientation with their gaze, thus giving a sense of presence under the blimp. In the publication by Ott et al. [42], the same benefits of their system are discussed, and the applicability of such a system is demonstrated by their experiments.

However, we believe that virtualising the environment and offering the possibility to explore it in 3D rather than from a 2-dimensional image in a HMD offers an even more natural way of interacting with surveillance tools.

Another advantage of this system is that traditional surveillance and monitoring systems cannot naturally merge information from different sources and capture devices. Traditional interfaces have to either show this information separately, or process it in a different way to be able to show them simultaneously. However, by offering a virtual platform with the simulation module**, multiple sources of information can augment the virtual environment** and cooperate to create a more complete image of the real environment. This could all be done in real time and concurrently, although with a centralised model of the system.

Stripping undesired information from the environment while keeping the essential geometry also allows for more efficient analysis tasks in order to understand the real environment. It gives a focused view of the world, where the desired **objects are brought immediately to the attention of the user**, and leaves more time for the actual decision-making process.

We believe that the potential of this kind of interface is worth exploring since it appears to offer a great deal of advantages, and provides an accessible decision-making tool for many applications.

Limitations

One of the main limitations of this platform is the assumption of the real environment geometry, and how it may not necessarily stay the same at the moment of capture. This can lead to inaccurate analysis by the user, as they may not assess the situation as they should considering the environment. With the current implementation that we propose, this is an important problem. However, it can be overcome by dynamically generating the environment around the drone with Simultaneous Localization and Mapping (SLAM) tools.

Another limitation regarding the simplification of the environment and the extraction of a few points of interest is that, while many undesired details are lost, some of the lost details could also be of importance to the user. This can happen in multiple situations, the most likely being a tracking failure from the object tracking module. When this happens, the object of interest will not be picked up at all, and the user will not know that it is there. However, this problem can also come from the development of the system itself, and the target objects. The developer must be careful to make sure they are tracking all necessary objects of interest, lest they assume tracking a certain object will not be necessary, but as it turns out it is crucial in understanding the environment.

Such objects could be things like cars or pedestrians. Any object that could be dynamic in the environment, but is not necessarily an object of interest, can and should still be tracked in order to further understand the environment.

These limitations mean that the user who is in the virtual environment, can absolutely not be the one operating the drone remotely. While they may have an accurate understanding of the real scene's geometry, and could pilot the drone not to crash against any of the basic surfaces, there are many elements that could be unaccounted for. These elements range from structures that were omitted when constructing the virtual environment for clarity, to dynamic elements that are not being taken into account in the virtual environment.

Because of this, in order to complement this AV interface, the real 2D feed from the mounted camera can still be used to support the understanding of the real environment. However, this 2D feed should be naturally integrated into the 3D space by means of a tool the user can look at, rather than a fixed display overlaid in front of their eyes. This is in order to follow the principle of diegetic or spatial interfaces in VR, an active topic of research in user interfaces [44]. A practical way of doing this would be having the 2D feed on a 3D plane attached to one of the controllers, so that the user may bring it closer or hide it when they do not need to look at it.

Applications

The proposed Augmented Virtuality system has several potential applications, the first being an interface for monitoring and surveillance systems. Many of the available surveillance techniques rely on non-cooperative 2D interfaces, while this system allows for the modelization of the monitored environment, as well as providing a manageable framework for the cooperation of multiple capture points, be them static sources or moving vehicles.

Modelization of the environment surrounding the vehicle and capture of key objects can also have applications in aerospace, where it might be of interest to place sensors around a vehicle and monitor the surroundings in 3D for extraneous or target objects. Depending on the capability of the sensors, the scale of the environment to be monitored can be as big as the technology allows.

We chose AR with monocular computer vision techniques when developing this project, but the augmentation of the virtual environment is not restricted to these technologies only. In fact, any sensor or combination of sensors which allows estimating the direction of a signal, as well as estimating the distance of said signal, can place information in the virtual environment. This ranges from infrared light to thermal sensors, as well as high level data inferred by other means, similar to the pose estimation of objects we will show. It is from the combination of all these technologies that powerful monitoring systems can be born, providing much more than just positional and visual information.

Generally speaking, this kind of system is highly malleable to accommodate multiple kinds of information. It can cover a wide range of applications that require some sense of spatial awareness for monitoring tasks, as well as providing a framework for cooperation to augment the same environment.

4. System Implementation

In this chapter, the structure and implementation of the proposed AV system will be presented, as well as any other details regarding the followed methodology.

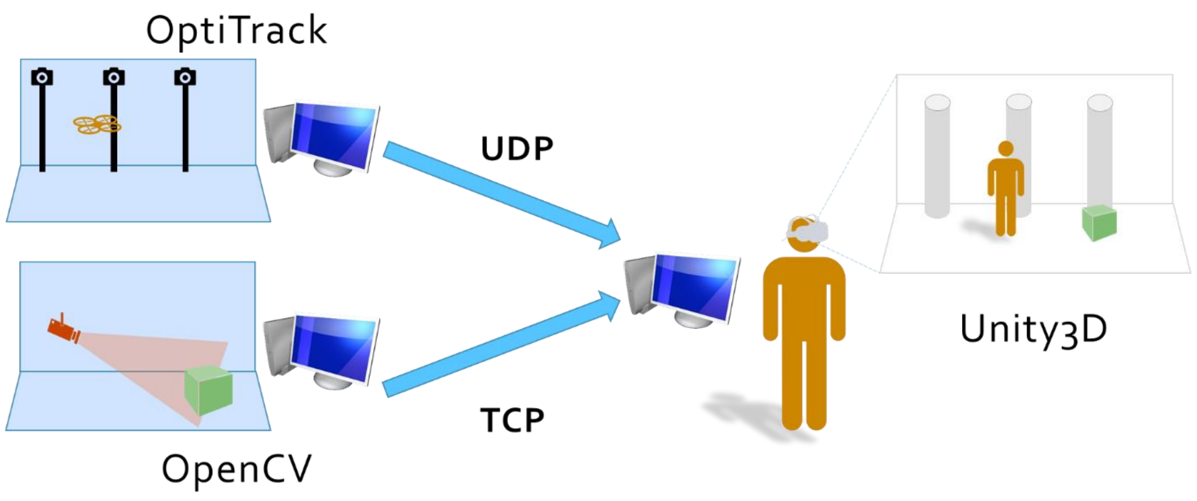


Figure 6 - Structure of the AV system

In order to keep the required modules from Section 0 independent, the system has been designed to be supported in up to three different machines. The machines can be connected via standard TCP/IP or UDP/IP protocols, either via a local area network (LAN), from a distance, or any combination of both. The drone tracking and object tracking modules communicate with the simulation module as seen in Figure 6, using standardised messages that we will see in their respective sections.

The first module to be built was the simulation module, using **Unity3D** and C# exclusively, and the **HTC Vive** head-mounted display. Afterwards, the drone tracking module was prepared using **OptiTrack** in a laboratory, tested, and connected to the simulation module. Finally, the object tracking module was implemented using **OpenCV** for C++ in either of the machines, tested and connected to the simulation module.

We will now have an overview of the technologies used and the implementation behind each of the modules.

Simulation module

First of all is the simulation module, which encompasses everything that the user will see in the final application. Everything was implemented using Unity3D and Visual Studio C# as the development environments for multiple reasons, the first of which being that Unity3D allows for very quick prototyping, correcting and testing.

Furthermore, Unity3D provides an easy to learn workflow which allows applications to scale nearly as much as the programmer wants, even with non-professional editions.

When building the simulation module, multiple goals were first defined:

- Integrate the VR interface
- Define a scheme for user movement
- Prepare the object tracking features
- Make all features generic and customizable
- Document objects and scripts and provide mouse-over descriptions

Note that we did aim to prepare the drone tracking features when first defining the goals of this module. This is because, as we will see in Section 0, there are factors to take into account that can only be seen when dealing with the real system.

In the following sections, we will see the different aspects of the simulation module implementation, and how the different goals were achieved during development.

Unity3D

Unity3D [45] is a powerful development framework for Windows and Mac OS X. It primarily offers tools for the creation of games, but can also serve in the development of educational applications and interactive simulations. Not only it includes an extensive API for this purpose, but applications built in Unity3D run on its proprietary engine. This means that the programmers only have to understand high-level concepts, and do not need to worry about the low-level intricacies of building an engine from the ground up if they do not wish to.

Layout overview

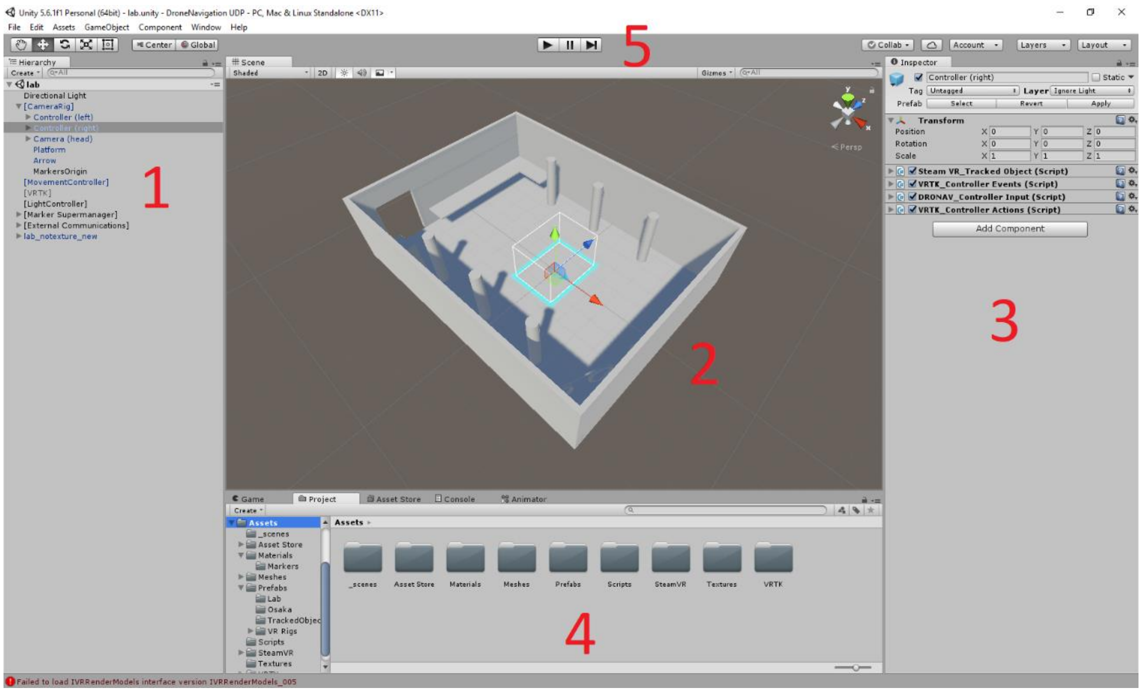
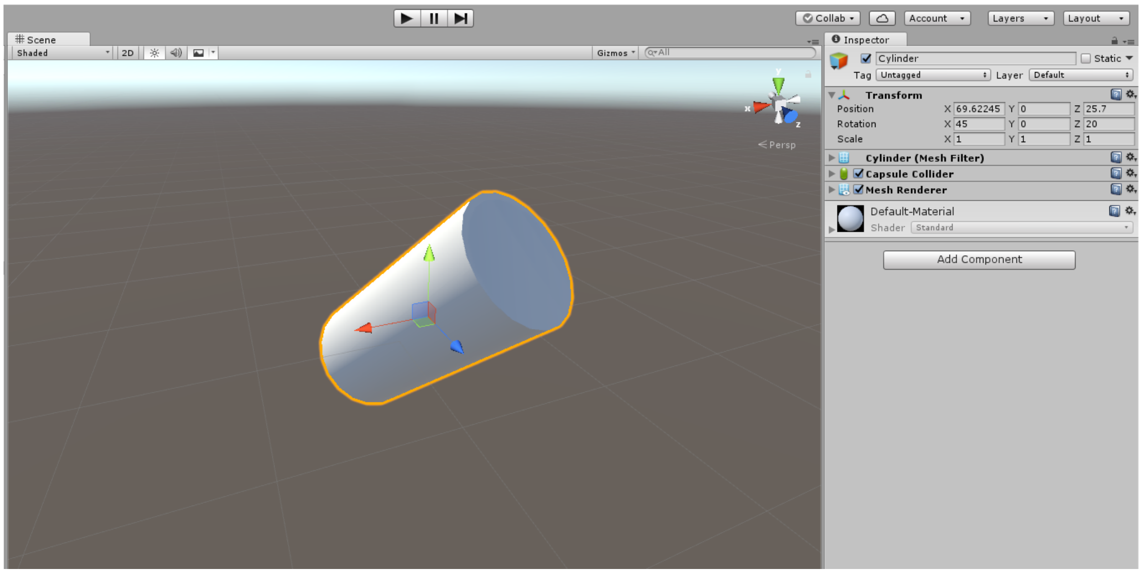


Figure 7 - Unity3D editor view. 1: Scene hierarchy, 2: Scene view, 3: Inspector view, 4: Project explorer, 5: Run/Pause tools

Figure 7 shows the basic view of Unity3D in editing mode. Every project is broken up in scenes, and each of these scenes can be edited individually on this screen. We can see the scene hierarchy (1), which lists the objects present in the scene and their relationship. The scene view (2) shows the 3D layout of the scene, and preview of the lighting conditions. Objects can be manipulated from here. The inspector view (3) shows the details of the selected object, and all of the components attached to it. The project explorer (4) allows the programmer to organize the project in the file system, by arranging all files usually in a standardised manner. Finally, the scene can be launched, paused and advanced step by step thanks to the run/pause tool on top (5).

GameObjects

Unity3D relies on **GameObjects** as building blocks to establish a scene. Each GameObject is an entity with a position in the world, which contains attached **Components** that define its behaviour.



Cylinder ☐ Static
Tag Untagged Layer Default

Transform

Position	X 69.62245	Y 0	Z 25.7
Rotation	X 45	Y 0	Z 20
Scale	X 1	Y 1	Z 1

Cylinder (Mesh Filter)
✓ Capsule Collider
✓ Mesh Renderer
Default-Material
Shader Standard

Figure 8 - GameObject in Scene view (left) and Inspector view (right)

The list of Components in a GameObject can be seen in the inspector view, as shown in Figure 8. Every existing GameObject has an associated **Transform** component, because they are physically present in the scene. The transform does not only hold positional, rotational and scale information about the object, but it also specifies where the object is in the scene hierarchy. Child objects and the parent object are all in the Transform. In fact, the scene hierarchy seen in Figure 7 is not a hierarchy of GameObjects, but a hierarchy of Transforms. Transform position / rotation / scale are shown relative to the parent Transform in the inspector view, or absolute if there is none.

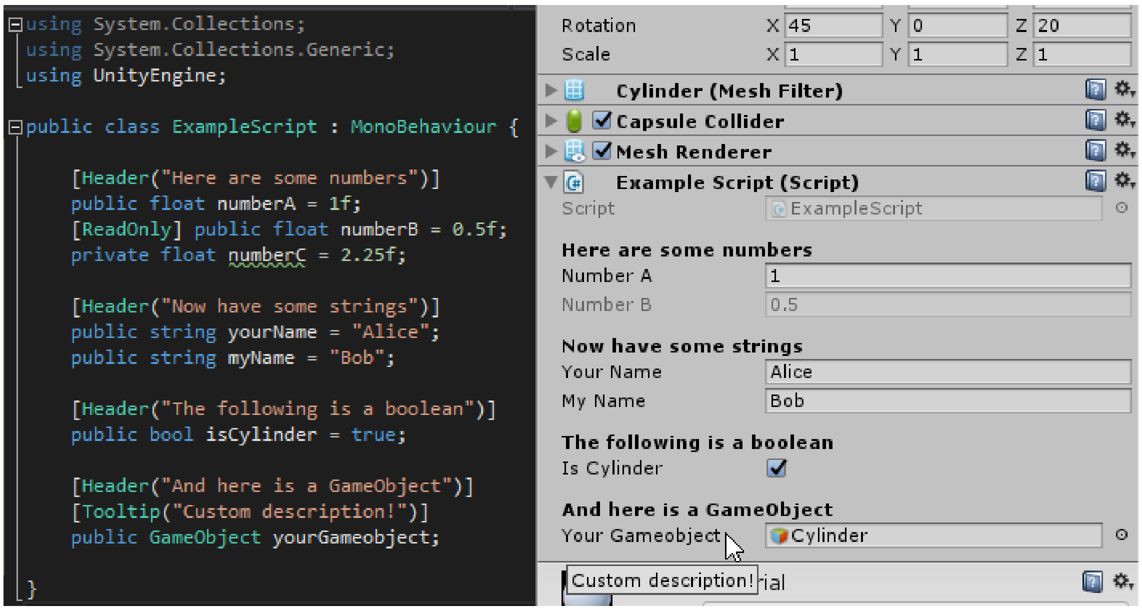
Here are some of the relevant Components that can be attached:

- **Transform** – Local position, rotation, scale and scene hierarchy
- **Mesh filter** – 3D mesh/model of the object in the world
- **Mesh renderer** – Visual properties of the model (texture, opacity…)
- **Collider** – Collision boundaries for interaction (physics, etc…)
- **Rigidbody** – Physical properties (mass, drag…) for physics calculations
- **Light** – Light emission from the center of the GameObject
- **Script** – Custom behaviour of the object, variables exposed in inspector

When a GameObject is going to be generated multiple times with the same properties or structure, or similar enough behaviour, it can be made into a **Prefab**. Prefabs can be created by simply dragging a GameObject to the project explorer, and act as a base object which can be instantiated into the scene multiple times. A good example of this would be bullets, birds or cars, even if they may present variations: these can be generated and changed at runtime. This is useful not only because it avoids re-building the same object multiple times, but also because they can be instantiated at any time, which means they can be used for any kind of task. This allows either a certain degree of genericity in GameObjects, or can be used as a simple shortcut when building scenes.

C# scripts

One of the most important parts when building an application for Unity3D is the script Components attached to the GameObjects, since these define the behaviours that will fit our application. Unity3D supports both the C# and JavaScript scripting languages; during the research project only C# was used.



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ExampleScript : MonoBehaviour {

    [Header("Here are some numbers")]
    public float numberA = 1f;
    [ReadOnly] public float numberB = 0.5f;
    private float numberC = 2.25f;

    [Header("Now have some strings")]
    public string yourName = "Alice";
    public string myName = "Bob";

    [Header("The following is a boolean")]
    public bool isCylinder = true;

    [Header("And here is a GameObject")]
    [Tooltip("Custom description!")]
    public GameObject yourGameobject;

}
```

```
Rotation        X 45    Y 0    Z 20
Scale           X 1     Y 1    Z 1
▶ Cylinder (Mesh Filter)
▶ ✓ Capsule Collider
▶ ✓ Mesh Renderer
▼ Example Script (Script)
Script          ExampleScript

Here are some numbers
Number A        1
Number B        0.5

Now have some strings
Your Name       Alice
My Name         Bob

The following is a boolean
Is Cylinder     ✓

And here is a GameObject
Your Gameobject Cylinder
Custom description!ial
```

Figure 9 - Script in Visual Studio (left), Script component in Inspector view (right)

Public variables in a script will automatically be shown in the Inspector view, as shown in Figure 9; and private variables will be hidden, although they can still be exposed by other means. These variables can be modified at any point from the Inspector view, before or during execution, although changes made during execution will not be permanent. The values provided inside the script are just the default values that will be used when attaching the script to an object.

Every script has a set of functions defined by the Unity API that will help bring their behaviours to life. They are as follows: [46]

- **Awake()** – Called *once* when the object is first created
- **Start()** – Called *once* when the object is first enabled
- **Update()** – Called once *every frame* (framerate dependent)
- **FixedUpdate()** – Called at a *fixed rate*, possibly multiple times every frame (not framerate dependent)
- **LateUpdate()** – Called *after every other Update()* has finished

If any of these functions is specified in the script, they will be called by the Unity3D engine at runtime accordingly. Most of the time code will be in the **Update()** function, but since the delay between each Update() call is not fixed, this can be fatal for physics and collision calculations, where objects might have different behaviours at different framerates. Some internal calculations should be done in **FixedUpdate()** in that case, which should not suffer from those framerate issues.

User input by polling should also be done in the correct function: if the user input is received in FixedUpdate(), it might affect the scene in-engine, but frames are not generated and the visual feedback is not accurate. Because of this, it is worth considering putting user input in Update(), where the user input matches visual feedback. However, for time sensitive user input, FixedUpdate() should still be used.

Finally, another special type of function is essential for some features: **Coroutines** [47]. Coroutines are functions defined as `IEnumerator` that can be interrupted with the `yield return` statement, to be resumed later from the same point. They are executed serially with the rest of the code but allow using intermediate results, and can simulate parallel execution like a thread.

Project layout

Now that we have covered the basis of Unity3D, this section will present the layout and hierarchy that was used to bring the simulation module to life.

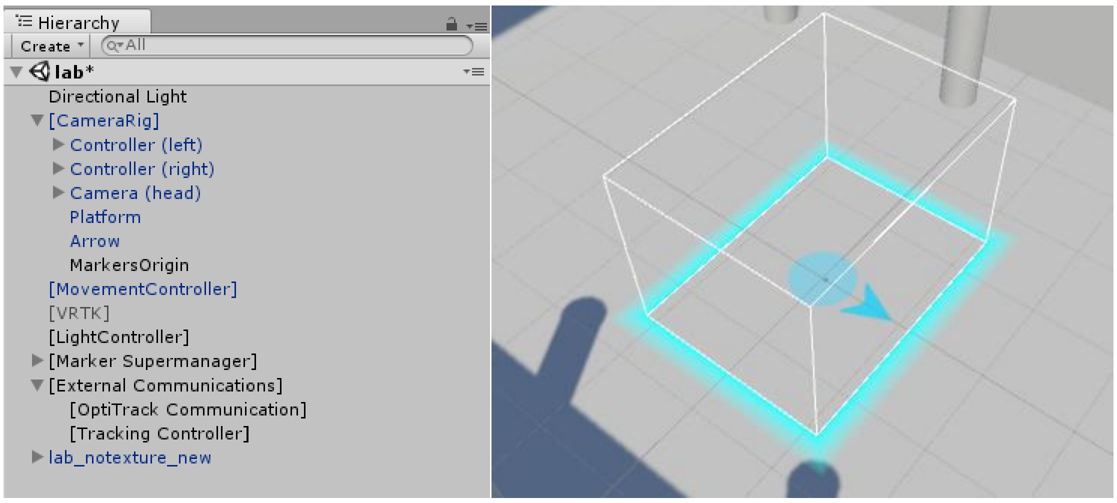


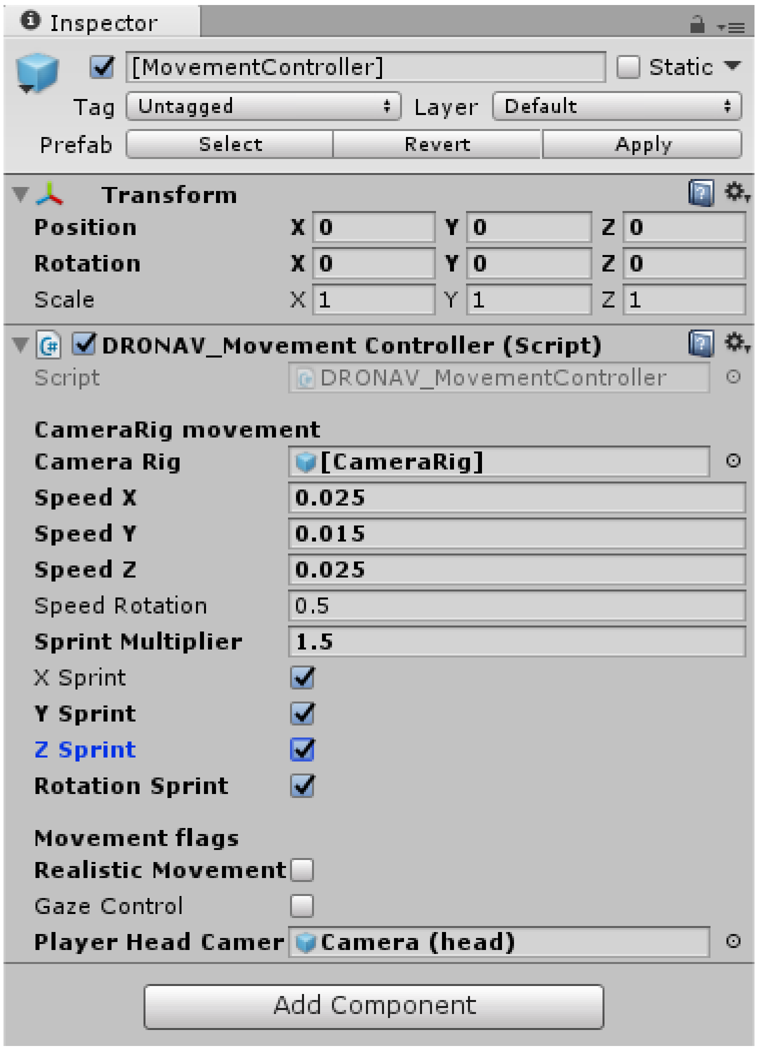
Figure 10 - Scene hierarchy in Unity3D (left), [CameraRig] in the scene (right)

As seen in Figure 10, we make use of several different controllers to manage all aspects of the simulation. We will go through all GameObjects shown and briefly explain them:

- **Directional Light** – Default light illuminating the scene
- **[CameraRig]** – Area where the user will stand, default VRTK prefab
- **Controllers** – Models and input scripts for the controllers
- **Camera (head)** – Camera following the user’s head with HTC Vive
- **Platform + Arrow** – Indicate where to stand when moving
- **MarkersOrigin** – Used for object tracking, see Section 0
- **[MovementController]** – Manages [CameraRig] movement
- **[VRTK]** – VRTK prefab to connect VR kit to Unity
- **[LightController]** – Manages changes in lighting for display features

- **[Marker Supermanager]** – Used for object tracking, see Section 0
- **[External Communications]** – Used for communicating with other modules, see Section 0 and Section 0.
- **lab_notexture_new** – simplified 3D mesh of the laboratory

Since the controllers related to the other two modules will be explained with their respective sections, we will look at the movement and input controllers, and their scripts.


Inspector

[✓] [MovementController] Static
Tag Untagged Layer Default
Prefab Select Revert Apply

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

[✓] DRONAV_Movement Controller (Script)

Script	DRONAV_MovementController
CameraRig movement	
Camera Rig	[CameraRig]
Speed X	0.025
Speed Y	0.015
Speed Z	0.025
Speed Rotation	0.5
Sprint Multiplier	1.5
X Sprint	✓
Y Sprint	✓
Z Sprint	✓
Rotation Sprint	✓
Movement flags	
Realistic Movement	☐
Gaze Control	☐
Player Head Camer	Camera (head)

Add Component

Figure 11 - [MovementController] Inspector view

The most important controller in this regard is the **[MovementController]** GameObject, whose inspector view can be seen in Figure 11. The script attached to it serves as a bridge between any event that wants to move the [CameraRig], and the [CameraRig] itself. Appendix A.1 show the parameters that can be specified to control the [CameraRig] movement, and the functions that serve as an interface for the other controllers.

With this [MovementController] acting as an interface between user input and user movement, a control scheme must be defined to allow intuitive movement. The **Controller (left)** and **Controller (right)** GameObjects have a “Controller Input” script attached to them, which reads user input via Update() and events, and calls the appropriate [MovementController] (or other controller) functions.

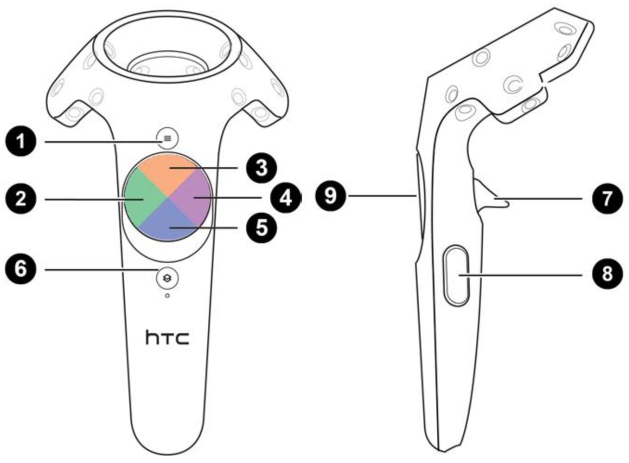


Figure 12 - HTC Vive controller buttons

#	Button name	Left controller function	Right controller function
1	Menu	Toggle day/night mode	Toggle gaze control
2	Trackpad left	Rotate left	Move left
3	Trackpad up	Fly up	
4	Trackpad right	Rotate right	Move right
5	Trackpad down	Fly down	
6	System	Display HTC Vive menu	
7	Trigger	Move back	Move forward
8	Grip	Toggle drone attachment	Sprint (while holding)
9	Trackpad press	*Unassigned*	

Table 3 - HTC Vive button control scheme

Figure 12 shows all the controller buttons available for non-locomotion user input in HTC Vive. To make use of all movement features, we distinguish between the two controllers provided, the left controller and right controller in-application, as shown in Figure 13. Red was assigned to the right controller, and blue was assigned to the left controller.

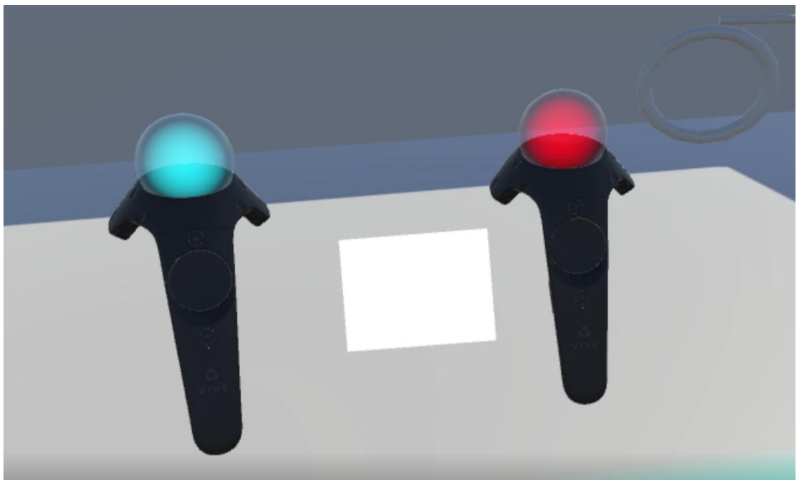


Figure 13 - Left (blue) and right (red) HTC Vive controllers in-application

The control scheme presented in Table 3 was used. Moving forward was assigned to the right controller for being the most used button, and most people being right handed. Rotation is more frequently used than lateral movement, thus rotation was kept in the opposite controller to accommodate usage with both hands. Regardless of these assignments, the controllers can simply be swapped if the user finds it more comfortable.

A control scheme with the same functions was used for the keyboard and is shown in **¡Error! No se encuentra el origen de la referencia.**. It is read in the Update() function of the [MovementController] and [LightController] scripts.

Key	Function
W	Move forward
S	Move backward
A	Move left
D	Move right
UP arrow	Fly up
DOWN arrow	Fly down
LEFT arrow	Rotate left
RIGHT arrow	Rotate right
Shift	Sprint (while holding)
N	Toggle drone attachment
M	Toggle day/night mode

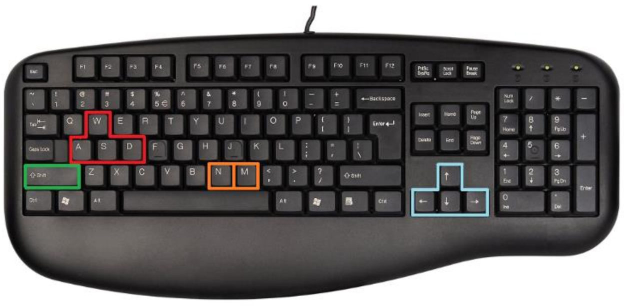


Table 4 - Keyboard control scheme

Figure 14 - Keyboard control layout

As seen in **¡Error! No se encuentra el origen de la referencia.**, this keyboard is easy to use since it follows the common control patterns used in 3D editing applications and games. This layout can be used if the user does not feel comfortable with the HTC Vive controls, or for simple debugging tasks that require moving around the environment.

One limitation of this control scheme is that it does not allow rotations in the X-Z axis (pitch, roll). This slightly limits the freedom in movement of the user. However, it is advised that the user is standing upright in the virtual environment anyway, since they are standing upright in the real world. If the user rotates around the X-Z axis, they might become disoriented and stumble as their visual information does not match that of their body's.

Finally, the **[CameraRig]** itself depicted in Figure 10, serves as a visual point of reference, which indicates where the user should stand and where to look at.

Object tracking

In order to accept the positional information of predetermined real-world objects, an internal infrastructure for the management and display of the objects was built. When building the system, the goal was to have highlighted markers imitating the shape of the real objects, moving in the virtual environment according to the real object's movement. The different objects would be predetermined before the execution of the application, and this internal module would be completely self-managed.

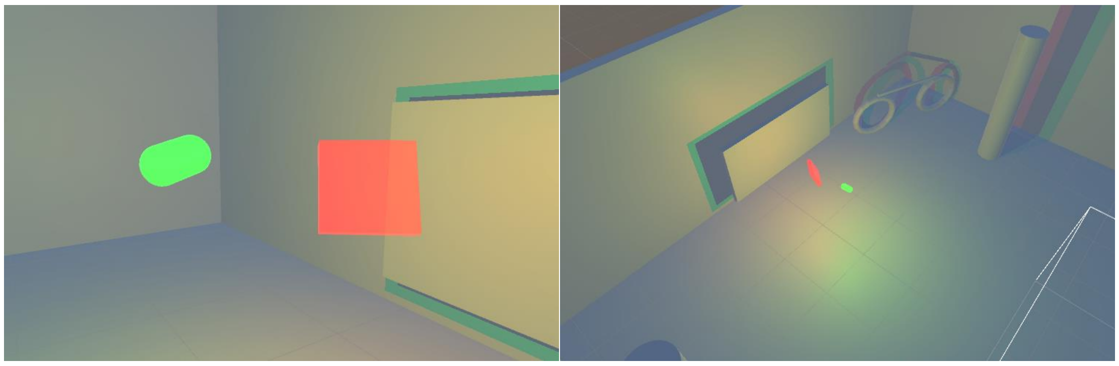


Figure 14 - Two example tracked objects inside Unity3D, in night mode

Figure 14 depicts this main set of features. The following is a full list of features that were finally implemented in the object tracking module inside Unity:

- ➢ Genericity, and scalability: takes in multiple kinds of objects
- ➢ Customizable model colours and light colours
- ➢ Night mode, further highlighting the tracked markers
- ➢ Formatted external input for refreshing marker positions
- ➢ Time-out and fade-out animations for unrefreshed markers

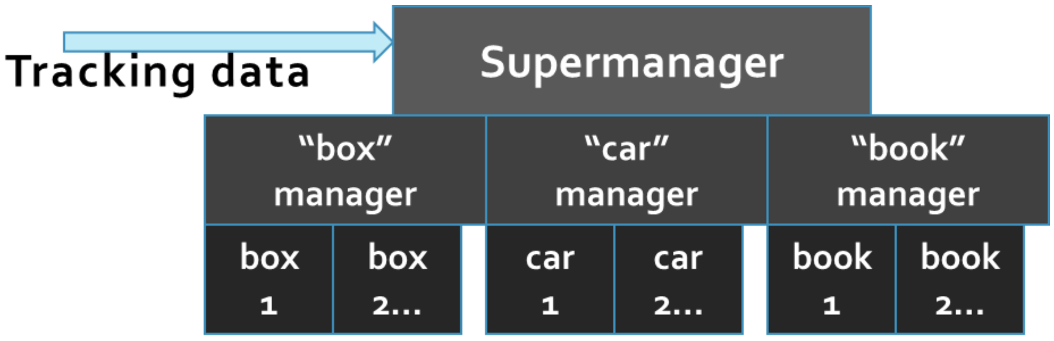


Figure 15 - Basic structure of the marker system in the simulation module

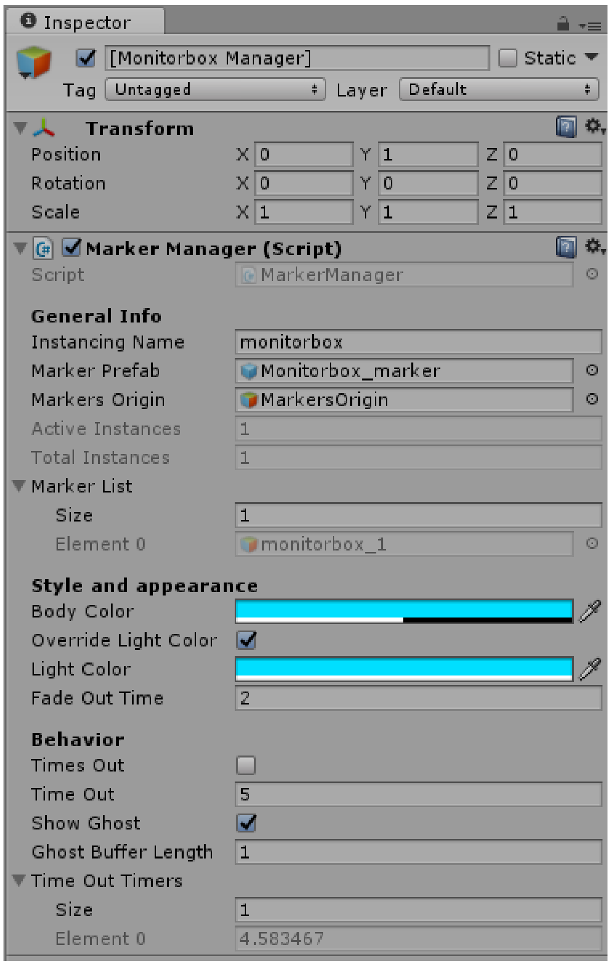
In order to implement these features, the layered structure shown in Figure 15 was established. This layered structure follows a hierarchy, where higher levels encapsulate and manage lower levels. The three layers are:

- ➢ **Marker** – Represents a real-world object with a highlighted model
- ➢ **Manager** – Manages all markers of a certain type of object
- ➢ **Supermanager** – Data entry point, sends commands to all managers

Each **Marker** GameObject contains a model of an object, a light source that ignores the model, and an attached script. It is defined by a name and a numbered ID; the name and ID are generated by a Manager depending on the type of object. The first "taxi" object would be called "taxi_1" with ID 1.

A marker is not capable of moving by itself, and its only purpose is to depict an object in a static position. It is only capable of controlling its light and colour, and animate changes by fading over time.

Managers, as previously mentioned, refer to one and only one specific type of object. They decide the name of the object they want to handle, the Prefab to instantiate when generating markers (thus, the model), and the colour of the generated markers. They keep track of the Markers under their control and update their time-out timers with the help of internal lists, as evidenced by Figure 16.



Inspector

[Monitorbox Manager] Static

Tag Untagged Layer Default

Transform

Position	X	0	Y	1	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

Marker Manager (Script)

Script	MarkerManager
General Info	
Instancing Name	monitorbox
Marker Prefab	Monitorbox_marker
Markers Origin	MarkersOrigin
Active Instances	1
Total Instances	1
Marker List	
Size	1
Element 0	monitorbox_1
Style and appearance	
Body Color	
Override Light Color	✓
Light Color	
Fade Out Time	2
Behavior	
Times Out	☐
Time Out	5
Show Ghost	✓
Ghost Buffer Length	1
Time Out Timers	
Size	1
Element 0	4.583467

Figure 16 - [Manager] Inspector view

They can be given the order to move a marker with a certain ID, and will either move the existing marker to the target position, or generate a new one with the predefined properties. The time-out timer for said marker will also be reset. Since these specific cases are handled internally by the Manager, higher layers only need to call this method on the appropriate manager:

```
manager.MoveMarkerByID(id, position, rotation);
```

Additionally, since the tracking data comes from a real camera with a position in the real world, we must specify where the camera is in our virtual world so that the Manager can place the Markers relative to it. This is done with the empty MarkersOrigin GameObject, which is shown in Figure 10.

Finally, the **Supermanager** has no specific parameters, and serves exclusively as an entry point for any parsed tracking data. It keeps track of all the existing Managers and their names, and there is only one instance at a time. It can be given the order to move a marker of a specific object name, with a certain ID, with the following function:

```
supermanager.MoveMarker(name, id, position, rotation);
```

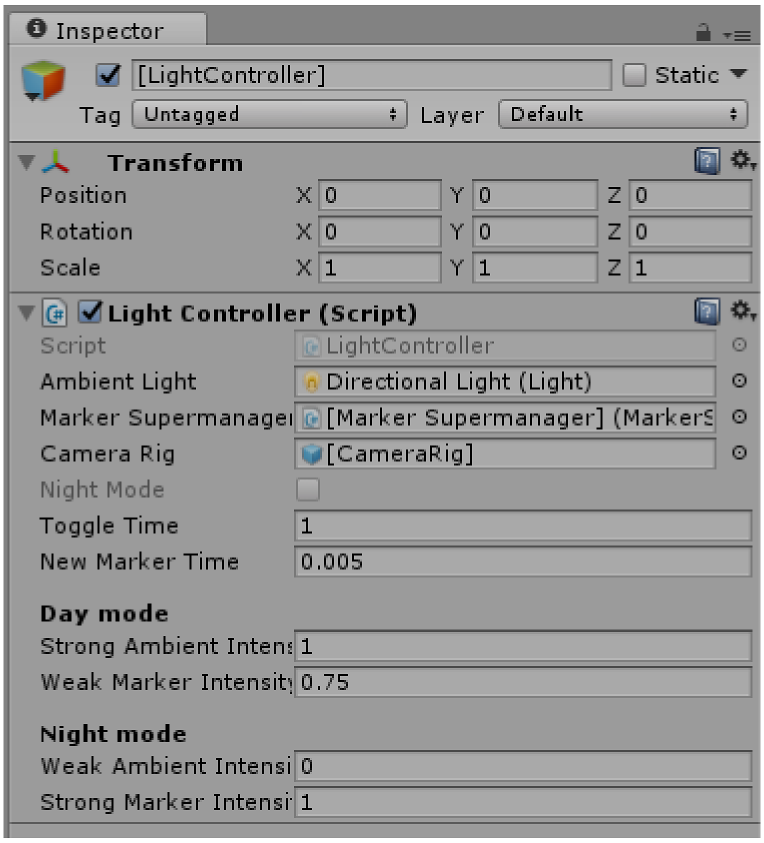
If a Manager with the corresponding name exists, it will call its `MoveMarkerByID` method and relay the ID, position and rotation information for the marker. Should a Manager not exist, it will simply return an error message instead of creating a Manager, like Managers create Markers. It means the simulation was not prepared for that kind of object, and it would have to guess what 3D model and parameters we want to use for it.

Visually, the two main features of this infrastructure are the day/night mode, and the automatic fade-out of markers. We will briefly discuss how these were achieved.

The **automatic fade-out** of unrefreshed markers is handled by Managers. For every Marker, they also keep track of a timer. At each Update(), all timers are decreased and then checked, and whenever a Marker is moved its timer is reset to its original value. However once the timer hits zero, the Manager forces the Marker to fade out to transparency over time with the help of dedicated coroutines. After the Marker has become invisible, it lingers for some seconds and is finally destroyed completely from the scene. It can still be re-instantiated if new tracking data is received.

Finally, the **day/night mode** is achieved thanks to the **[LightController]** seen in Figure 10. When night mode is enabled, the [LightController] fades the Directional Light in the scene and all the existing Markers to different values (preferably decreasing the scene light and increasing the Marker light). It calls the following function from the Supermanager:

```
Supermanager.FadeLightAll(lightIntensity, time)
```



Inspector

[✓] [LightController] [ ] Static

Tag: Untagged — Layer: Default

Transform

	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

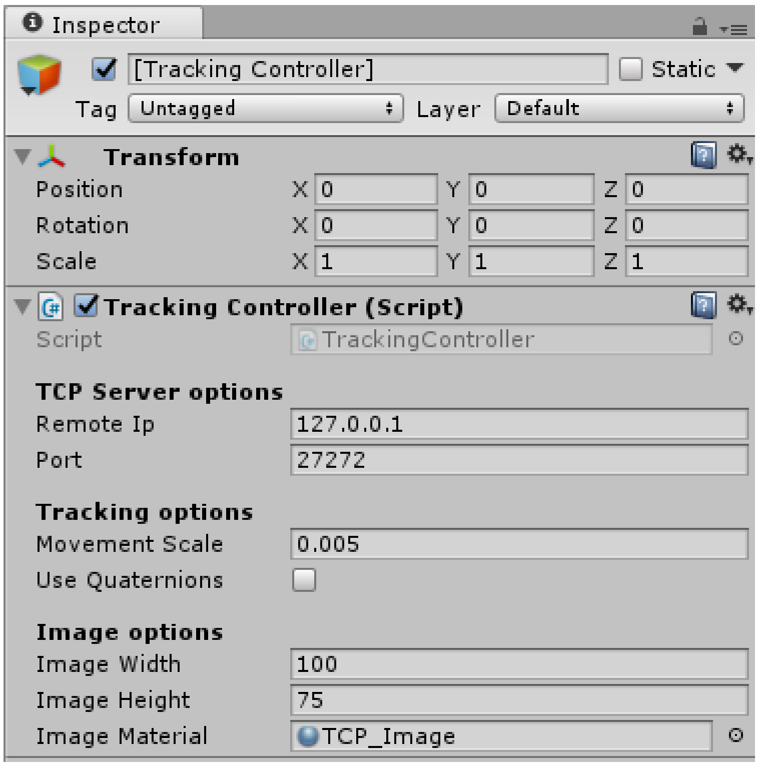
[✓] Light Controller (Script)

Script	LightController
Ambient Light	Directional Light (Light)
Marker Supermanagei	[Marker Supermanager] (MarkerS
Camera Rig	[CameraRig]
Night Mode	☐
Toggle Time	1
New Marker Time	0.005
Day mode	
Strong Ambient Intens	1
Weak Marker Intensity	0.75
Night mode	
Weak Ambient Intensi	0
Strong Marker Intensi	1

Figure 17 - [LightController] Inspector view

¡Error! No se encuentra el origen de la referencia. shows the different parameters that can be customized in the [LightController]. Depending on how much the programmer wants to highlight the markers, they can adjust these numbers in order to obtain better results. This can be used for other effects, such as for completely turning off the lights emitted by the markers, and similar applications.

While this entire infrastructure works on its own, it needs to receive external data in order to actually generate the markers. The **[Tracking Controller]** takes care of this task, by being the entry point of the entire application for object tracking data.


Inspector

☑ [Tracking Controller] ☐ Static

Tag: Untagged — Layer: Default

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

☑ **Tracking Controller (Script)**

Script: TrackingController

TCP Server options

Remote Ip: 127.0.0.1

Port: 27272

Tracking options

Movement Scale: 0.005

Use Quaternions: ☐

Image options

Image Width: 100

Image Height: 75

Image Material: TCP_Image

Figure 18 - [Tracking Controller] inspector view

As shown in Figure 18, the Tracking Controller only needs to know the IP and TCP port to listen to. The Tracking Controller will accept an incoming client connection, and will act as a server receiving data asynchronously. We decided to use a TCP connection instead of a UDP connection since the tracking framerate requirements were relatively low. Thus, we can afford to sacrifice throughput for increased reliability.

The data received by the Tracking Controller should be a string of characters, specifically formatted for the application. Incoming strings use an XML-like structure to describe the state of the tracking system, and mimic that of the structure of the system described earlier in this subsection. Each individual marker is described as:

<ID=_><x=_><y=_><z=_><qx=_><qy=_><qz=_></ ID>

Here, x y and z define the position, and qx qy and qz define the rotation of the marker. The rotation can also be represented in quaternions, as seen in Figure 18. These markers will be encapsulated in the following structure:

<Markername="__"> … … … </ Markername>

Each of these elements will describe each type of tracked object, and contain all the markers referring to that type. Finally, all *<Markername>* elements will be encapsulated by a final level:

<Markers> … … … </ Markers>

An example of this structure in the real application would be:

<Markers>

```
    <Markername="monitorbox">
        <ID=1><x=-15.9756><y=16.512><z=214.234>
        <qx=-149.817><qy=4.90648><qz=-28.5223></ID>
    </Markername>
</Markers>
```

After receiving a message, it is parsed inside the [Tracking Controller] and the `supermanager.MoveMarker(name, id, position, rotation)` function is called for each *<ID>* element in the message. However, a scaling parameter is used for the position information, which depends on two things: the scale of the object in the tracking application, and the scale of our world in the simulation *(NB: In Unity, 1 unit = 1 meter)*. These should be taken into account and factored in to obtain valid results.

This standardised format allows communication with any kind of tracking application, as long as said application tailors its output before sending it to the simulation module in Unity3D. With this, the object tracking section of Unity is ready to accept information, update it, and manage it by itself.

Building the virtual lab

Before proceeding to tracking the drone's position in the real laboratory, it is necessary to set up the virtual environment that will be equivalent to the real environment. A 3D model of the real environment has to be built, imported to Unity, and verified against the real environment.

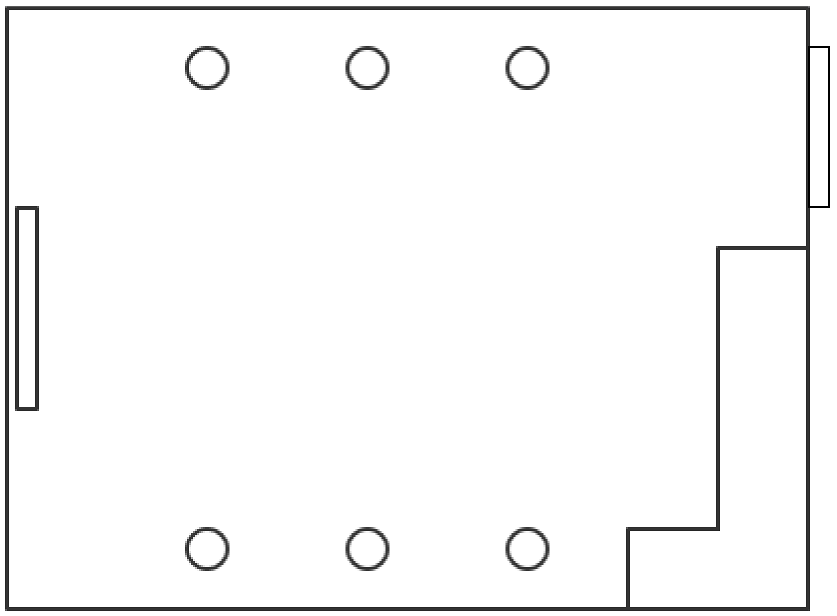


Figure 19 - Rough map of the laboratory static features

As shown in Figure 19, the map has a set of static features that must be taken into account when building its 3D model: 6 OptiTrack cameras, a TV monitor, a workbench in the corner, and the main entrance. Less reliable static features were added in the final model, like a bicycle and a stack of boxes, which never changed throughout this thesis. The room has 3:4 proportions in length and width, and the ceiling was not taken into account for the purposes of the application.

With this information, we can use any 3D modelling program to build the 3D model. In our case, we used **Blender**, an easy to use application for 3D modelling, rendering and animation. The obtained model is shown in Figure 20. Note that some of the shapes in the Unity3D version are moved (cameras farther apart, cameras closer to workbench, boxes closer to the door). This is because the .blend file retains the different shapes when imported in Unity, and allows to move them inside the model.

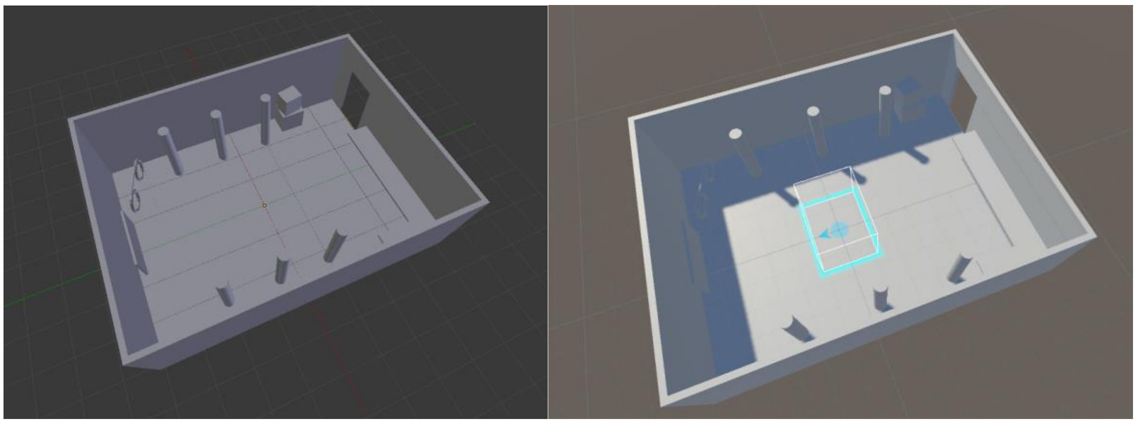


Figure 20 - Laboratory model, in Blender (left) and Unity3D (right)

When first testing the drone tracking, the elements of the virtual environment were adjusted so that the environment would be correctly calibrated, and its geometry matched that of the real one. This was made by checking where the tracked object would collide with the main features by bringing it closer, and adjusting the position accordingly in the Unity project.

The simulation module is now ready to move the [CameraRig] in the virtual environment according to the real environment. In the following section, more details will be given on the [OptiTrack Communication] GameObject, which receives external tracking data from the drone and applies it.

Drone tracking module

Going back to our proposed system in Section 0, the objectives related to the drone tracking module are:

- A drone should fly in a closed, controlled environment
- Drone movements in the real environment should be mimicked by moving in the virtual environment
- The user should be able to detach from the drone and explore the environment freely

From these requirements, we decided to use the OptiTrack technology available in our laboratory to carry out this task. In this section, we will see the principle behind OptiTrack and how the tracking module was handled.

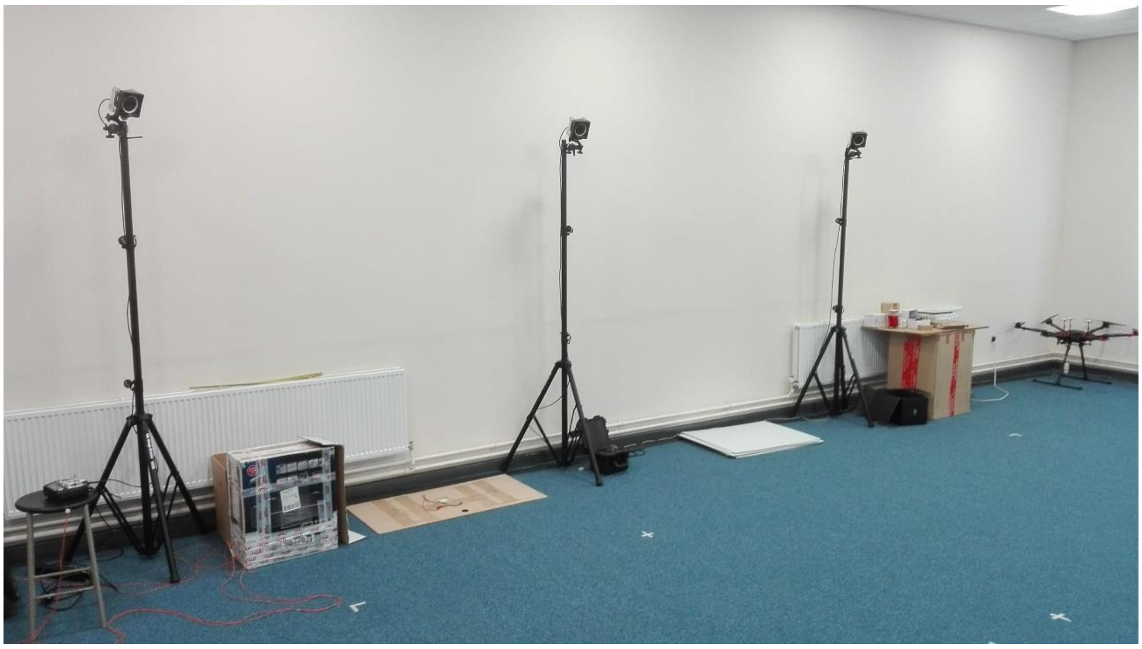


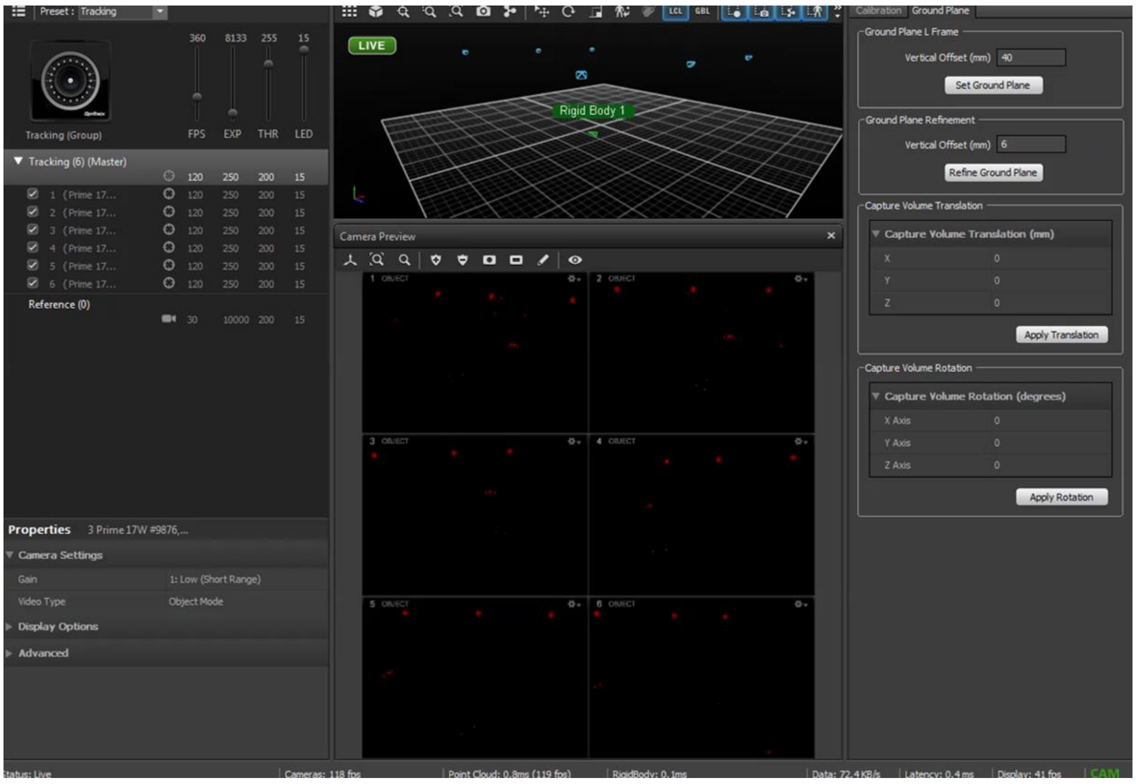
Figure 21 - View of the laboratory's OptiTrack environment

OptiTrack

OptiTrack is a widespread commercial motion capture solution, with an extensive list of applications as of today [48]. It offers 6 DoF tracking (3D position and 3D rotation). It consistently offers sub-millimetre precision, and negligible rotational error in most conditions. Furthermore, it works at a high framerate, being capable to capture up to 360 fps with our available set-up.

Due to its accuracy and efficiency, OptiTrack is used as the ground truth for the validation of many applications, such as AR tracking and trajectory estimation. It is also the reason this technology was selected for this project.

OptiTrack operates with a set of IR cameras that cover different angles and regions of the same space. Thanks to a set of reflective markers, they are able to estimate the position and rotation of an object to which the markers are attached.



Preset : Tracking

Tracking (Group)	FPS	EXP	THR	LED
	360	8133	255	15
Tracking (6) (Master)	120	250	200	15
1 (Prime 17...	120	250	200	15
2 (Prime 17...	120	250	200	15
3 (Prime 17...	120	250	200	15
4 (Prime 17...	120	250	200	15
5 (Prime 17...	120	250	200	15
6 (Prime 17...	120	250	200	15
Reference (0)	30	10000	200	15

Properties 3 Prime 17W #9876,...

Camera Settings

Gain	1: Low (Short Range)
Video Type	Object Mode

Ground Plane L Frame

Vertical Offset (mm) 40

Ground Plane Refinement

Vertical Offset (mm) 6

Capture Volume Translation

Capture Volume Translation (mm)	
X	0
Y	0
Z	0

Capture Volume Rotation

Capture Volume Rotation (degrees)	
X Axis	0
Y Axis	0
Z Axis	0

Status: Live | Cameras: 118 fps | Point Cloud: 0.8ms (119 fps) | RigidBody: 0.1ms | Data: 72.4 KB/s | Latency: 0.4 ms | Display: 41 fps | CAM

Figure 22 – Motive main interface, using 6 OptiTrack cameras, tracking 1 object

In order to exploit the potential of this technology, we used **Motive**, an application that serves as an interface for OptiTrack's hardware. After calibration of the cameras by "wanding" the OptiTrack wand around the area, and defining the X-Y-Z axis by using a special piece with three markers, OptiTrack can start tracking objects. Each object that we want to track has a representation in Motive called "Rigid Body". A Rigid Body is defined by a set of 3 or more markers, and its principle is that the markers cannot move relative to one another. This is a promise that the structure of the markers is "rigid", hence the name. Failure to ensure this will lead to tracking errors.

Tracking

In order to track the drone and send its positional information in Unity, we must follow a specific pipeline. First, the drone has to be registered by Motive. Then, Motive should stream the information captured in each frame. For that, the gap between Motive and Unity must be bridged. Finally, Unity should use the frame information to move the drone in the AV. We will now see the details in each step.

Once the drone is ready to be tracked, a set of at least there reflective markers must be placed on it. All of the markers must be visible before tracking, and they must be selected in the 3D view of Motive. Once a Rigid Body is created out of them, Motive will be tracking the Rigid Body at the selected framerate; for the purposes of our application, we selected 120fps as the target framerate. Figure 22 – Motive main interface, using 6 OptiTrack cameras, tracking 1 objectshows for example a Rigid Body tracked with only three markers.

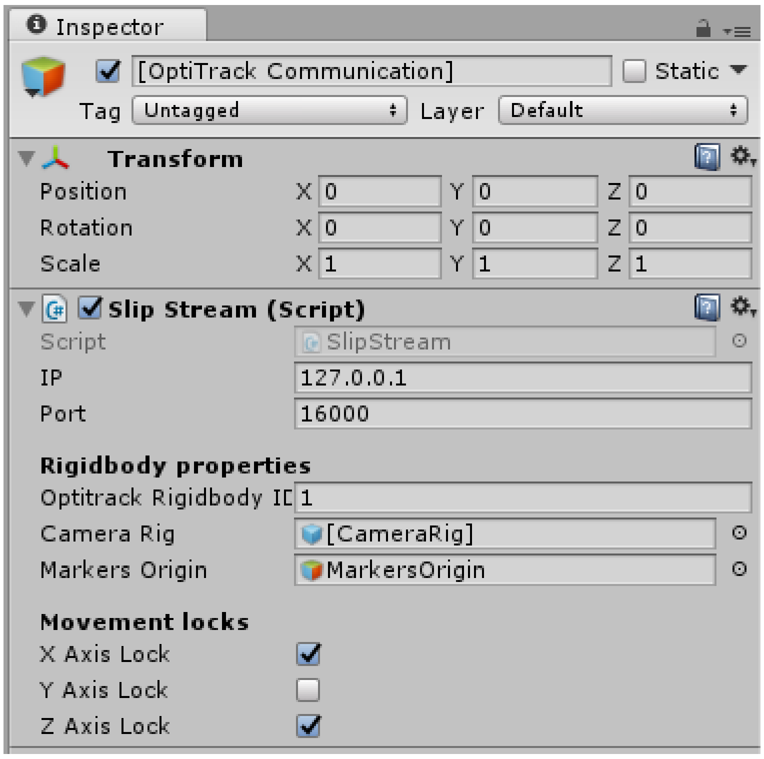
The next step is to stream the data from the Data Streaming tab. From here, the information to be streamed can be selected. In our case, we only want to know the pose of the drone's Rigid Body, so we disable streaming Markers and Skeletons. In order to send the data to Unity, a C++ executable bridges the gap between the two applications. The basis for this executable is provided in the **NatNetSDK**, the networking SDK used for Motive. This executable unwraps the data package send by Motive, and wraps it in an XML format, similar to that of Section 0. The produced format is the following:

```
<Rigidbodies>
        <Rigidbody ID=1
        x="1" y="1" z="-3"
        qx="0" qy="1" qz="0" qw="2" />

        <Rigidbody ID=2
        x="3" y="0" z="-2"
        qx="0" qy="1" qz="0" qw="2" />
...
</Rigidbodies>
```

Rotations are expressed in quaternions, rather than Euler angles. This is helpful since while Unity might not display rotations as quaternions, internal operations and assignments are all managed in quaternions.

Each of these messages is sent to Unity through a UDP port. In this case, we use UDP since the throughput is very important, as this data will potentially move the player around. If the framerate is too low, the user's movement will be irregular and they will become motion sick. These messages are captured by the [OptiTrack Communication] GameObject.


Inspector

[✓] [OptiTrack Communication] [ ] Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

[✓] Slip Stream (Script)
Script SlipStream
IP 127.0.0.1
Port 16000

Rigidbody properties
Optitrack Rigidbody IC 1
Camera Rig [CameraRig]
Markers Origin MarkersOrigin

Movement locks
X Axis Lock [✓]
Y Axis Lock [ ]
Z Axis Lock [✓]

Figure 23 - [OptiTrack Communication] Inspector view

Similarly to the [Tracking Controller] in Figure 18 - [Tracking Controller] inspector view the [Optitrack Communication] GameObject requires an IP address and a Port to listen to. As seen in Figure 23, it requires additional information. First, it requires the ID of the Rigid Body to be treated as the drone in the XML. Then, it requires to know what the [CameraRig] is, and where the MarkersOrigin GameObject is.

The basis for the connectivity on Unity's side is also provided in the NatNetSDK. After receiving the message, the correct Rigid Body is parsed from the XML. If the user is currently attached to the drone, then the script will simply move the [CameraRig]. However, if the user is not attached to the drone and is moving freely, only the MarkersOrigin which represents the real camera's position will respond to the tracking.

Rotation around the X/Y/Z axis can also be independently restricted. It is useful to restrict the pitch and roll motions for the same reasons they are restricted in user movement, to avoid user disorientation. Furthermore, it can also simulate stability even when the real drone is flying unstably. However, this must be taken into account when spawning objects from the object tracking module, as their positions or rotations might not be accurate.

The tracking module is now able to provide the simulation module with accurate and high framerate tracking for the drone, and the simulation module has the tools to manage the incoming data.

Object tracking module

Our goal is to track objects with a monocular camera by estimating their pose, and sending the information to the simulation module. During the research project, we decided that the OpenCV [49] library fit our requirements, in terms of flexibility of the code. OpenCV is an open-source C++ and Python library for computer vision, with frameworks for detection, AR, machine learning, and many more applications.

For the purpose of this application, a single C++ program was developed to take care of the object tracking. In this section, we will see the underlying architecture, and the algorithm behind the tracking.

Software architecture

While it is not necessary to make it so, the architecture for the object tracking module mirrors that of the marker system in Unity. The same layered architecture of Object → Manager → Supermanager is followed, as shown in Figure 24.

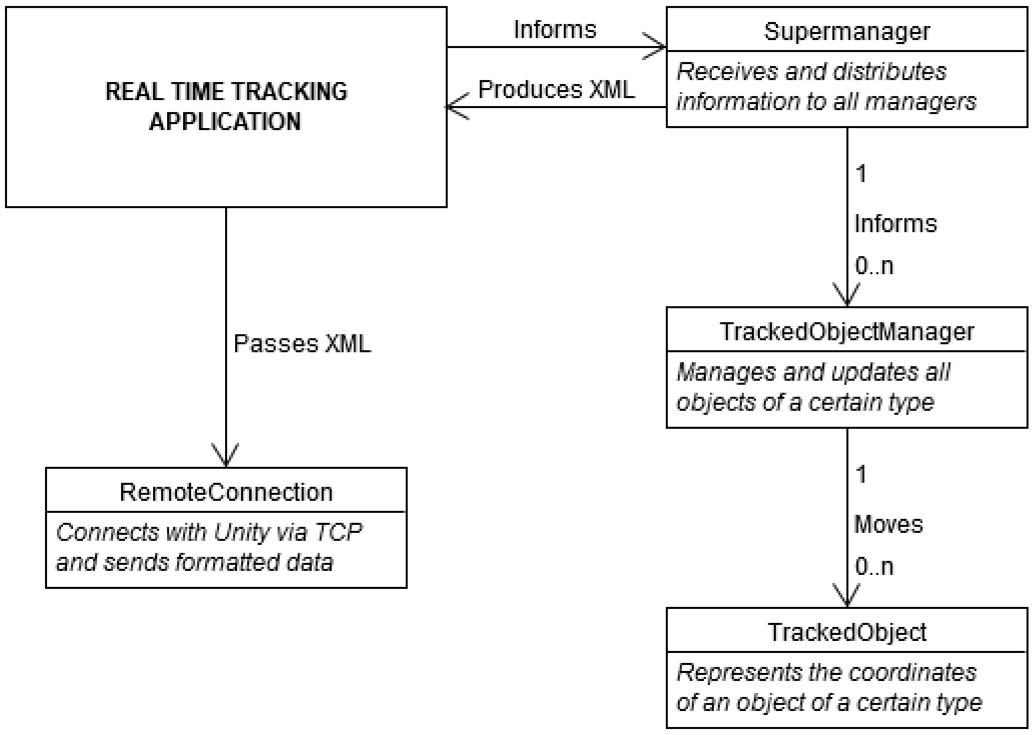

REAL TIME TRACKING APPLICATION
Informs
Produces XML
Supermanager
Receives and distributes information to all managers
1
Informs
0..n
TrackedObjectManager
Manages and updates all objects of a certain type
1
Moves
0..n
TrackedObject
Represents the coordinates of an object of a certain type
Passes XML
RemoteConnection
Connects with Unity via TCP and sends formatted data

Figure 24 - Basic UML diagram of the object tracking module

`TrackedObject` represents a single object of a given type, with a 3D position and a 3D rotation. It can only change its position and rotation, and nothing else. `TrackedObjectManager` keeps track of `TrackedObjects` in an ordered double-ended queue (`deque`). It can retrieve them by index or by ID, and place them back efficiently in the `deque` when an object is being tracked again. This is checked and done automatically when calling the following function:

```
bool MoveObjectByID(int id, Vector3 position, Vector3 rotation);
```

The `Supermanager` is the final layer of this structure, which has access to all `TrackedObjectManagers`. It is the entry point for the tracking data obtained from the other half of the application, and can move an object calling:

```
bool MoveObject(std::string type, int id, Vector3 position, Vector3 rotation);
```

Unlike the Unity version of this structure, this `Supermanager` is able to create `TrackedObjectManagers` if the “type” does not match any existing one. This is because in this case `TrackedObjectManagers` do not store any knowledge of the object aside from their name, so there are no assumptions to be made.

Additionally, the `Supermanager` can produce an XML string that can be output by the application, following the format specified in Section 0. Each layer creates a wrapper, and asks the next layer to fill it with information. First, the `Supermanager` creates the following wrapper:

<Markers> … … … < / Markers>

`TrackedObjectManagers` add their own wrapper one by one:

<Markername=”__”> … … … < / Markername>

Finally, for each `TrackedObjectManager`, each `TrackedObject` adds:

<ID=_><x=_><y=_><z=_><qx=_><qy=_><qz=_>< / ID>

After the XML has been generated with the `std::string GetXml()` method, the tracking application can use the `RemoteConnection` class to connect to Unity and send the resulting string through a TCP port.

However, this architecture is only a way of representing the tracking data; it does not help in the actual tracking of real objects. The other half of the C++ program follows a certain pipeline to track objects from a video feed. We will now see in detail this real-time tracking pipeline.

Tracking algorithm

Going back to Table 1 by Lima et al. we can see the different methods at our disposal for our task. We must take into account that our video feeds will have rough camera movements because of the drone, that there might be strong lighting changes, and that backgrounds could be cluttered. Because of this, it seems that a **keypoint-based** approach would be appropriate. For this application, we will follow the suggested workflow in the OpenCV documentation for a keypoint-based approach [50].

In our case, the priority was to provide a proof of concept for the applications of AV in navigation. More complex shapes and 3D meshes could be used; however we decided that tracking textured objects with 8 vertices like boxes, books, and so on, would serve our application well. This is also one of the reasons the suggestion from the OpenCV documentation was used.

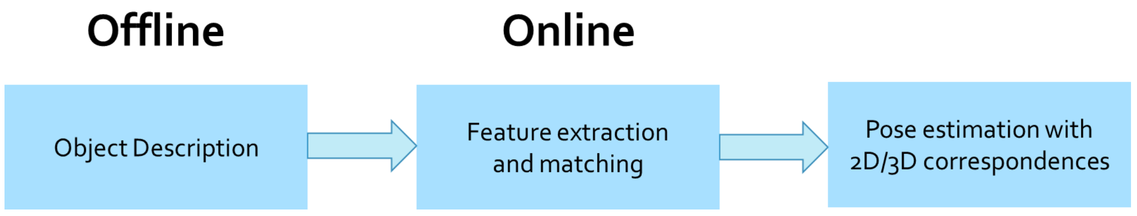

Offline
Online
Object Description
Feature extraction and matching
Pose estimation with 2D/3D correspondences

Figure 25 - Tracking workflow diagram

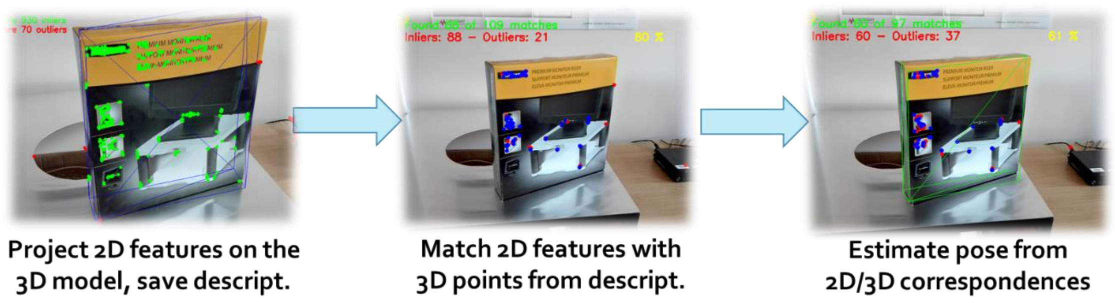

Project 2D features on the 3D model, save descript.
Match 2D features with 3D points from descript.
Estimate pose from 2D/3D correspondences
Inliers: 88 – Outliers: 21
Inliers: 60 – Outliers: 37

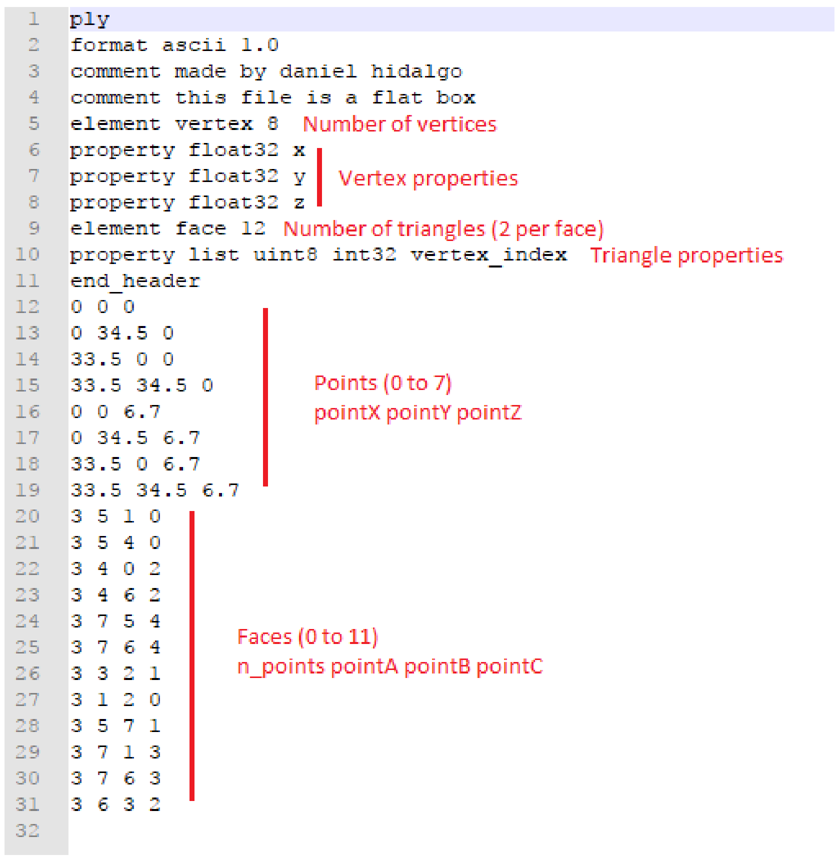
Figure 26 - Tracking workflow with images

Figure 25 and Figure 26 show the basic idea of the workflow. It relies on a simple offline phase for description, and an online phase for the detection and tracking of the object. We will first see the offline description phase, and then the online registration phase.

Offline description

A separate executable was made for the offline description of an object to be tracked. It requires two things: a 3D mesh of the object, and a picture of the unoccluded object, containing the features that we expect to see in the video feed. The keypoint-based approach relies on the detection of 2D features from the object, and associating them to 3D points. In that way, each 2D feature effectively has a 3D point that can be used to estimate the pose of the object when detected in an image.

The format used for the 3D mesh is .ply, a human-readable type of file that can easily be interpreted and edited. It has the structure shown in Figure 27.



```
ply
format ascii 1.0
comment made by daniel hidalgo
comment this file is a flat box
element vertex 8
property float32 x
property float32 y
property float32 z
element face 12
property list uint8 int32 vertex_index
end_header
0 0 0
0 34.5 0
33.5 0 0
33.5 34.5 0
0 0 6.7
0 34.5 6.7
33.5 0 6.7
33.5 34.5 6.7
3 5 1 0
3 5 4 0
3 4 0 2
3 4 6 2
3 7 5 4
3 7 6 4
3 3 2 1
3 1 2 0
3 5 7 1
3 7 1 3
3 7 6 3
3 6 3 2
```

Figure 27 - .ply format structure, 6-sided box example

With this 3D mesh and the 2D picture, we can project the 2D features from the image onto the mesh to associate each feature with a 3D point. However, we must first place the 3D mesh in the camera coordinate system to be able to project those features. A common way of placing the 3D mesh in the camera coordinate system is to use pictures from which we know the pose of the object relative to the camera in the first place, like in the interest-point based method [34]. In our case, we opted for the more flexible solution proposed in the OpenCV documentation for simple-shaped objects. By going through all the points in the .ply and placing them one by one in the image, we are giving a set of 2D-3D correspondences; these can be used to estimate the pose of the mesh in the image by the use of PnP solvers (0). A pseudo-code of this process:

```
For every point P(x,y,z) in list_points
    Select corresponding point p(x,y) in image
    Generate pair c(p,P) with 2D and 3D point
    Add c to list_correspondences
Solve PnP for list_correspondences
Place 3D mesh in image
```

After the 3D mesh has been placed in the camera coordinate system, we can select the 2D features that we would like to describe the object with. During the online registration, these 2D features will be extracted once again by the registration algorithm, so the choice of feature descriptor is very important in the application. We must take into account the alternatives that are available, considering the restrictions of our system in the real world. The objects that we want to detect might not be facing the camera directly, might be rotated in an unexpected way, and might be closer or farther from the camera. This means that we need a descriptor that is both rotation invariant and scale invariant. Since OpenCV provides a good implementation, we decided to use the ORB feature descriptor [39]. It provides both of these properties, and is efficient enough for our target application.

Once the feature descriptor has been decided, we can proceed to extract these features from the provided image. The number of keypoints that the descriptor will extract is important, since it will have a direct impact on both the accuracy of the system, and its computational cost. Increasing the number of keypoints will increase both the accuracy and the computational cost of the online registration phase. This will have a direct effect on the framerate, thus the programmer should decide where to settle in this trade-off.

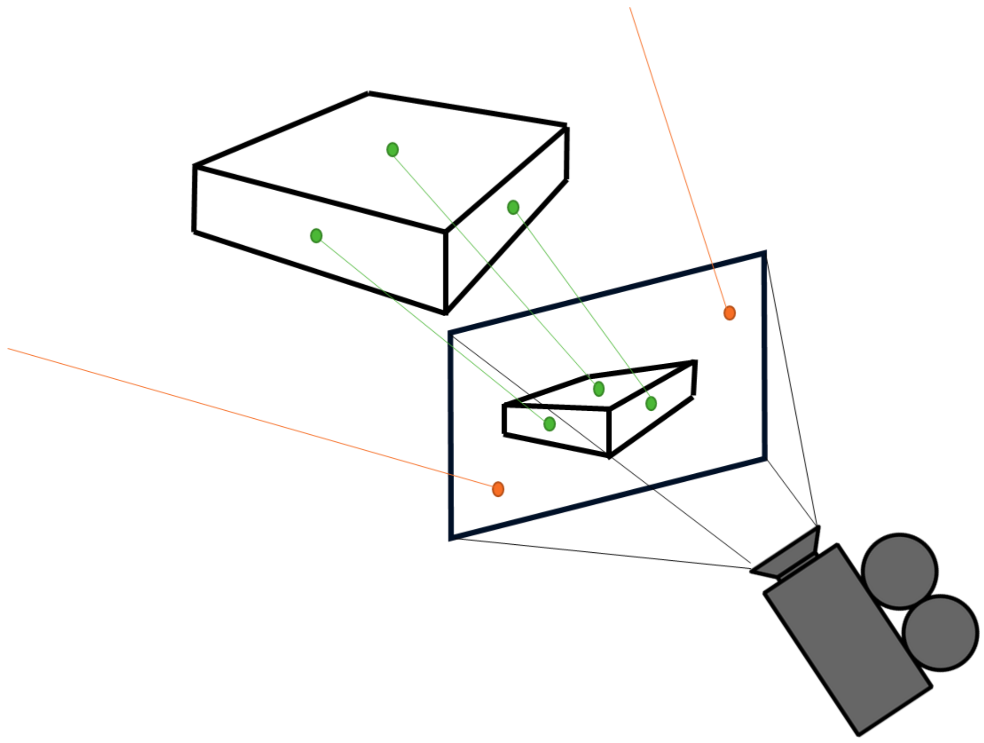


Figure 28 - Projection diagram of 2D features. Green: inliers, Orange: outliers

Once all the 2D features have been extracted from the image, they are projected into the 3D space, from the camera coordinate system to the world coordinate system. This is done by "ray-casting" through each of the 2D points, creating a 3D line that might intersect with surfaces in the 3D space. Every feature that intersects with the 3D mesh is considered an inlier, which means it contributes to the description of the object. However, features that do not intersect with the 3D mesh are considered outliers, which means they are features that describe the background of the image and have nothing to do with the object. This behaviour is shown in Figure 28. After obtaining all of the inliers that describe the object, the features are exported to a .yml file that lists all of the ORB features with their descriptor, and their corresponding 3D point.

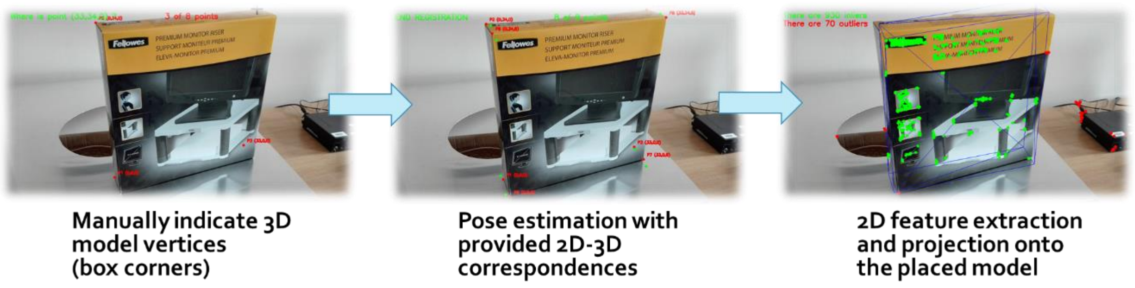

Manually indicate 3D model vertices (box corners)
Pose estimation with provided 2D-3D correspondences
2D feature extraction and projection onto the placed model

Figure 29 - Offline description workflow with images

Once the .yml file is produced, we now have a full description of the object in 3D based on 2D features. The entire workflow for the offline description can be seen in Figure 29, where images at each step is shown. In the third image, inlier features projected onto the mesh are shown in green, and outlier features which did not find a projection are shown in red. A pseudo-code of the entire process would be:

```
For every point P(x,y,z) in list_points
      Select corresponding point p(x,y) in image
      Generate pair c(p,P)
      Add c to list_correspondences
Estimate pose A(R,t) of 3D model for list_correspondences
Place 3D mesh in image with pose A(R,t)
Extract features from image with ORB onto list_features
For every feature F(p(x,y),descriptor) in list_features
      Project p(x,y) with ray-cast R(x,y)
      If R hits 3D mesh with A(R,t) at point P(x,y,z) in WCS
            Add F to list_inliers
            Add P(x,y,z) to list_inlierpoints
      Else
            Add F to list_outliers
Save list_inliers and list_inlierpoints to .yml
```

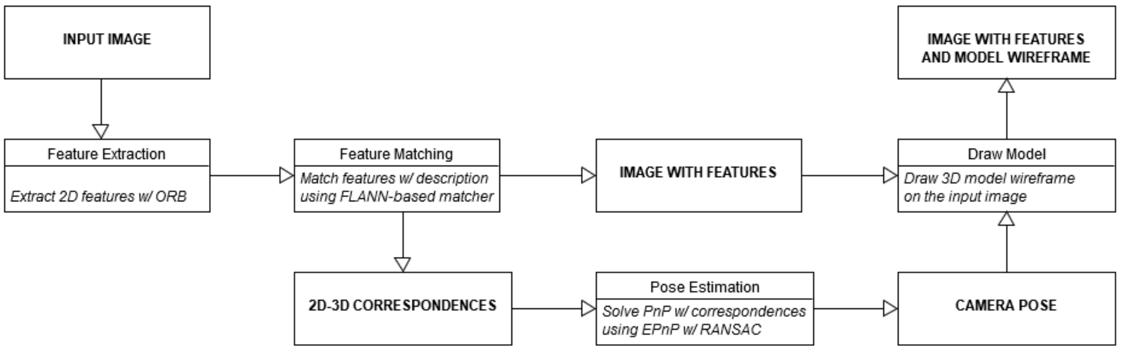
To ensure tracking will perform correctly in the online registration phase, the programmer must take into account two key aspects of this process: first, as previously mentioned, the features should be robust and tolerate the conditions that will be imposed in the registration phase. If this is not the case, the feature extractor will not be able to find these features in the real images, because the features extracted will not match the features from the object description. Furthermore, another aspect to be taken into account is the pose estimation phase of this process. If the pose estimation is done incorrectly or is too inaccurate, the mesh will be incorrectly placed in the camera coordinate system, and the projection of features will be done incorrectly; two things might happen in this case. On one hand, features that should be inliers will not be projected at all on the object, and will be lost as false outliers. This will result in fewer features with which to track the object in the registration phase. On the other hand, the inlier features that will be projected onto the mesh will be projected as different 3D points than the real 3D point they represent. When estimating the pose of the object with these points during the registration phase, the object will be translated and rotated with the same error as the pose estimation error made during the offline description phase.

This workflow can be applied to more complex objects, as long as the user is able to place the 3D points describing the mesh onto the 2D image. However, this workflow is strongly oriented to simple objects whose points can be clearly seen in the provided images. Nevertheless, if a strong pose estimation algorithm is used, the user would be able to roughly place the 3D points in the image, and the pose estimation should still find an accurate pose for the object.

Online registration

Thanks to the offline description phase, we have the resources necessary to detect and track objects in real time. The online registration part of the code corresponds to the "Real-time tracking application" element in Figure 24, and functions as its own module, similar to that of the layered architecture. The online registration and the Supermanager infrastructure are part of the same execution, and the former contributes to the latter frame by frame.

First of all, multiple resources are needed for the registration. The most essential is the .yml description generated by the user during the offline description phase. Additionally the online registration makes use once again of the same 3D mesh of the object, in .ply format. Of course the registration needs video input, which can either come from a pre-recorded video, or directly from a camera in real time. For testing purposes, we worked on pre-recorded footage.



INPUT IMAGE
Feature Extraction
Extract 2D features w/ ORB
Feature Matching
Match features w/ description using FLANN-based matcher
IMAGE WITH FEATURES
Draw Model
Draw 3D model wireframe on the input image
IMAGE WITH FEATURES AND MODEL WIREFRAME
2D-3D CORRESPONDENCES
Pose Estimation
Solve PnP w/ correspondences using EPnP w/ RANSAC
CAMERA POSE

Figure 30 - Online registration pipeline diagram

Figure 30 shows the pipeline of the registration, and the techniques used to solve each step. We will go through each step, and finally provide a pseudo-code implementation of the whole process.

- **Import object data** – First of all, the .yml description and the .ply mesh must be loaded beforehand. Afterward, the whole process can be executed frame by frame, with each execution being independent from one another. Additionally, the intrinsic parameters of the camera to be used should be specified in the code.
- **Feature Extraction** – For each input image, extract the 2D features by using the same feature extractor as the offline description process. It is important that the number of features to extract is greater or equal than the number of inliers found during the offline registration. As stated in Section 0, we will be using ORB since OpenCV provides a robust implementation and it fits our requirements adequately.

- **Feature Matching** – From the list of 2D features obtained after the ORB feature extraction, perform matching against the set of features from the object description. There are multiple ways to perform this matching step, but the FLANN-based robust matcher (Fast Approximate Nearest Neighbour) proposed by the OpenCV documentation is a sensible choice considering our requirements. It is fast, efficient, and easy to use for our proof of concept. By attempting to match each of the 2D features from the list, we obtain a list of outliers which did not find a match in the description; and a list of inliers which successfully found themselves in the object description, and to which we can now associate a 3D point.

- **Pose Estimation** – We now have a list of inlier features, which means we have a set of 2D-3D correspondences from 2D coordinates to 3D points in the world coordinate system (see Section 0). PnP solutions only need to know 2D and 3D points to estimate the rotation and translation of the camera, since the intrinsic parameter matrix is constant. OpenCV provides different implementations of PnP solutions, which only need to be given a list of 2D points and a list of 3D points, as well as additional accuracy thresholds. PnP solutions will try to minimize the reprojection error, and will give an $[R|t]$ matrix, with the help of RANSAC. In this case, we are using EPnP, since it is more efficient than traditional implementations of iterative PnP.

- **Output and Display** – Once the pose has been estimated, a wireframe of the object can be placed on the original image, and now the pose of the object for the last frame can be retrieved by other parts of the application.

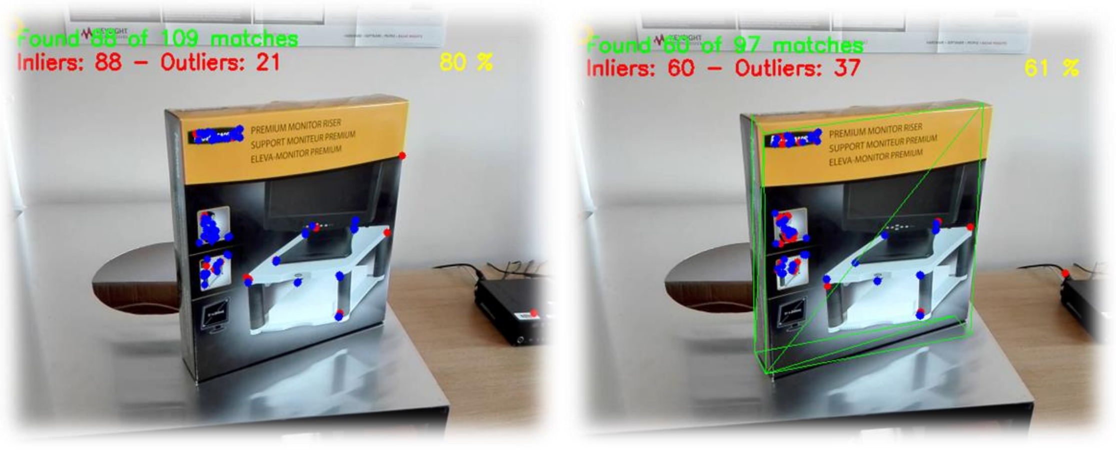


Figure 31 - Feature matching (left) and Pose estimation (right) examples

Figure 31 - Feature matching (left) and Pose estimation (right) shows an example of feature matching and pose estimation achieved with this workflow. Being based on the OpenCV documentation implementation of this pose estimation task, a great part of the pose estimation module relies on the open source code provided. However, the code has extensively been rewritten and re-structured for multiple reasons, the first of which being improved clarity, genericity and maintainability of the code. On the other hand, most of the utility code used by this workflow has been optimized at the lowest level, especially for recurring operations such as projections and rotations, by restructuring the data and eliminating overhead. The following is a pseudo-code implementation of the pipeline:

```
Extract features from image with ORB onto list_features
For every feature F(p(x,y), descriptor) in list_features
    Try matching F to object_description with FLANN
    If good match with feature f(P(x,y,z), descriptor)
        Add m(p(x,y),P(x,y,z)) to list_matches
        Draw F on image as inlier
    Else
        Add F to list_outliers
        Draw F on image as outlier
Estimate pose A(R,t) of 3D mesh for list_matches with EPnP
Draw 3D mesh in image with pose A(R,t)
```

N.B: In the actual implementation, the feature extraction from the image is incorporated in the same function that performs the feature matching.

After one frame has been processed, the rotation matrix and translation vector are retrieved from the pipeline. The rotation matrix is converted into an Euler angle rotation vector, and the information is passed to the `Supermanager` by calling its function:

```
bool MoveObject(std::string type, int id, Vector3 position, Vector3 rotation);
```

After the `Supermanager` has finished updating the state of the system, it can generate the XML message to send to Unity for that frame.

It is important to note that in the event that the processing speed of this pipeline is lower than the fps of the incoming video signal, the tracking will lag behind the live feed. The problem lies in the buffering of frames, in which frames queue before being processed by the OpenCV code. To counter this effect, an optional **frame-skip** feature was added. Upon retrieving a frame, the system will ignore all the frames that were lost during the previous computation cycle, and will use the latest frame to arrive in the buffer. In this way, the tracking system is synchronized to the live video feed.

With this system in place, Unity now receives tracking data from a video feed, describing a system of tracked objects with labels and positions/rotations.

5. Results

Due to this research project having been led at the Defence Academy of the United Kingdom, several restrictions have made testing the system in its entirety, a fairly difficult task. Fortunately, all three modules are independent from one another, meaning that they can be run in separate machines, or share machines in any possible combination. The main features which could not be tested together were the drone tracking and the HMD, due to both physical and network constraints. However, they are separate layers handled by Unity that do not affect one another, and will merge seamlessly when all components are properly integrated.

In this section, we will see the different tests that were carried out, and the viability of the components that make up the system.

Validation

Validation of the system was carried out continuously throughout the research project in two different machine: the **VR machine** and the **Lab machine**. The following are specifications of both machines:

VR machine specifications

- Windows 10 Entreprise 64-bit (Build 15063)
- LENOVO 30A6S3CV00
- Processor: Intel® Xeon® CPU E5-1630 v3 @ 3.70GHz (8 CPUs), ~3.7GHz
- Memory: 16,384 MB RAM
- DirectX 12
- GPU: NVIDIA GeForce GTX 960 (2,016 MB VRAM) (10,157 MB Total)

Lab machine specifications

- Windows 7 Entreprise 64-bit
- LENOVO 10FCS0DA00
- Processor: Intel® Core™ i5-6500 CPU @ 3.20GHz (4 CPUs), ~3.2GHz
- Memory: 8,192 MB RAM
- DirectX 11
- GPU: Intel® HD Graphics 530 (No dedicated memory) (1,824 MB Total)

Note that the Lab machine is less powerful than the VR machine in most aspects. This must be taken into account when evaluating the modules of the application.

The following parts of the application were tested on the VR machine:

- Simulation module (Unity3D) – *main platform*
- Head-mounted display (HTC Vive) – *only platform*

- Object tracking module (OpenCV) – *main platform*

On the other hand, the Lab machine ran the following parts:

- Simulation module (Unity3D)
- Drone tracking module (OptiTrack) – *only platform*
- Object tracking module (OpenCV)

To evaluate the success of the application, we can go back to our original set of requirements, and verify that each of the functional requirements is met. Since it is difficult to quantify the success of these features, we will discuss the user experience, restrictions and other thoughts on each aspect of the final application.

- **A drone should fly in a closed, controlled environment**
- **Drone movements in the real environment should be mimicked by moving in the virtual environment**

Thanks to the drone tracking module built from the OptiTrack technology, the standardisation of the communication and the UDP connection to the simulation module, we have successfully been able to mimic the drone's movements in the virtual environment. During the final weeks of the research project, multiple tests were made in the laboratory with real drones.

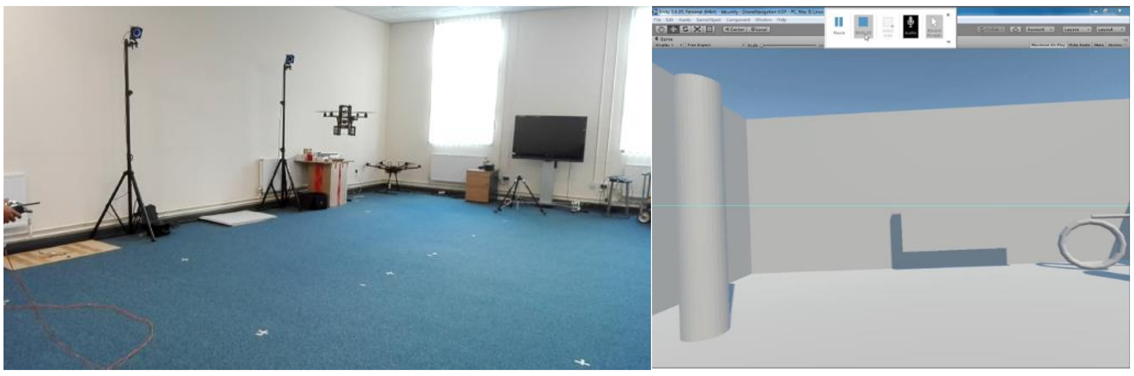


Figure 32 - Drone tracking module, flight test (left), Unity view (right)

As seen in Figure 32, the tests were successful. The connection between Motive and Unity3D is successfully established via the NatNetSDK executable, and the drone's position is being correctly mimicked in Unity3D. However, the module had previously been tested repeatedly with objects other than drones, for accessibility and safety measures. This allowed to test extreme angles and fast movements indoors, without any possible dangers and safety concerns.

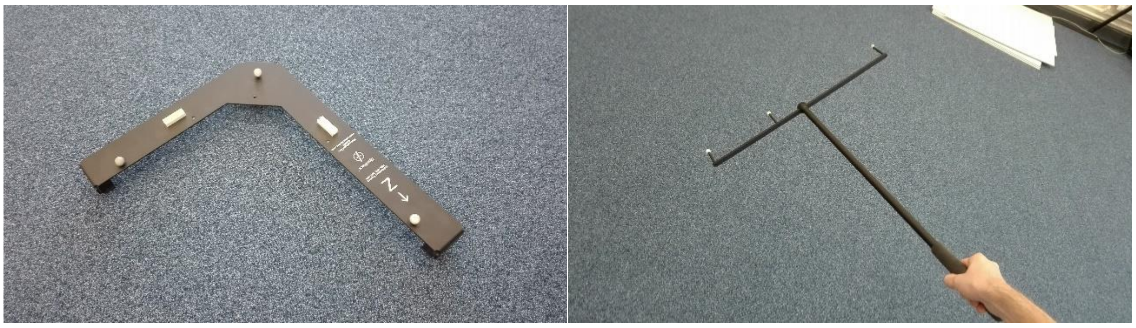


Figure 33 - (a) OptiTrack axis (left) - (b) OptiTrack wand (right)

Figure 33 shows one of the objects that were used to test the connectivity between OptiTrack and Unity3D, which we will refer to as pseudo-drones. The software responds correctly to the movement of the tracked object to replace the drone, and the functionalities such as the rotation locks and the Rigid Body selection (if multiple Rigid Bodies are being tracked) as well.

One of the noticeable restrictions of our set-up is that the OptiTrack cameras do not allow the drone to increase its altitude a lot. Horizontally, the cameras roughly cover a 7m x 6m rectangle in which tracked objects are not lost. However, the cameras lose track of the drone once it reaches roughly 2m of altitude, which is the height that can be seen in Figure 32. This is because the markers on the drone were all on the top, making it difficult for the cameras to capture the drone once it is elevated too much. However, for the purposes of our proof of concept, this limitation did not hinder our testing, and is tied directly to our set-up. Other tracking methods or camera configurations should not necessarily show this problem.

- **The simulation should show a simplified version of the environment**
- **The user should be able to observe the virtual environment with VR**

While some adjustments had to be made to the 3D model of the virtual environment, as stated in Section 0, the virtual representation of the real environment was also a success, as shown in Figure 32. Both when wearing the HMD and testing the drone tracking module, the room is up to scale and the distinguishing features are in place. This was verified by moving the pseudo-drone against the objects, and walking to key points of the scene, making sure that Unity shows the correspondence correctly.

Manual orientation of the environment was required when building the correspondence. When calibrating the OptiTrack environment, it was necessary to use the instrument shown in Figure 33 to define the X, Y and Z axis in Motive. When doing this, it was important to always use the same orientation for the X and Z axis between tests. If this was not done, upon sending the information to Unity, the entire environment would appear to be rotated, and require the model of the laboratory to be rotated manually in Unity, or redefining the axis in OptiTrack. Surely enough, the latter is easy to do and takes less than 30 seconds, but it is crucial to make sure that under any circumstance, the 3D model of the virtual environment is correctly oriented at any time. The biggest limitation at the moment is that this has to be done manually.

Despite this, integrating the VR into the project is as easy as using the tools provided by the Virtual Reality Toolkit (VRTK), including the [CameraRig] prefab, input listener scripts and many other functionalities that simply layer on top of the existing project seamlessly. Despite the environment lacking many features and being obviously virtual, all testers of the system have been successfully immersed in the environment, also recognizing the laboratory without being told. This shows the success of the virtual presence, once again, thanks to the VRTK , and also the calibration of the virtual environment model.

- **An object tracking module should track objects and their position with a monocular camera**

- **The application should be able to receive data from tracked objects and display it in the virtual environment**

During the research project, the object tracking module has been tested multiple times in both the VR machine and the Lab machine. One critical aspect of our testing is that due to time constraints, we were unable to test a real camera mounted on the drone. Instead, different sets of footage was capture in-house, and tested directly on the different parts of the software. This acts as a simulated feed, and can be treated as feed coming from the drone for testing purposes. Nevertheless, validation on real conditions still remains an issue.

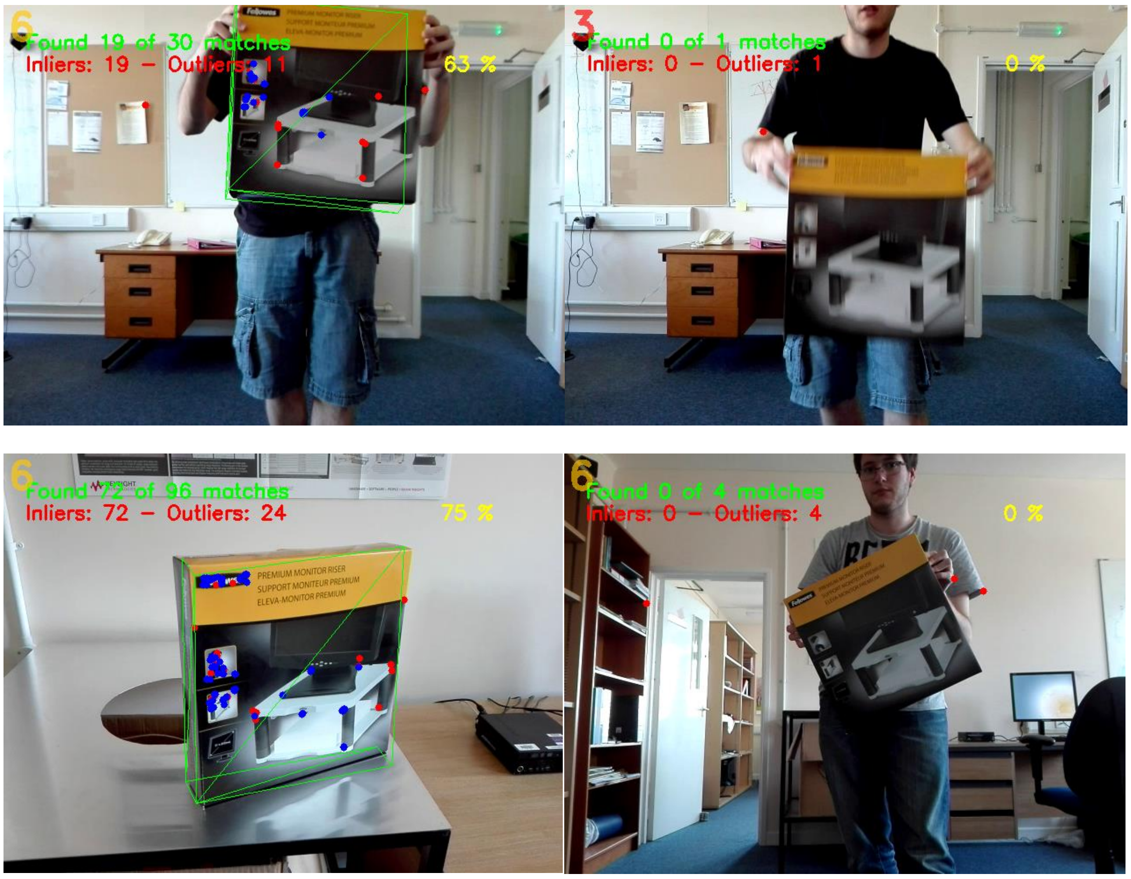


Figure 34 - Tracking examples with "monitorbox", success (left) & failure (right)

The software was tested with various objects, however tests for validating the connectivity and the tracking functionalities in Unity were made using the single “monitorbox” object. The tests shown in Figure 34 were made using 500 keypoints, and illustrate the success and the constraints of the system.

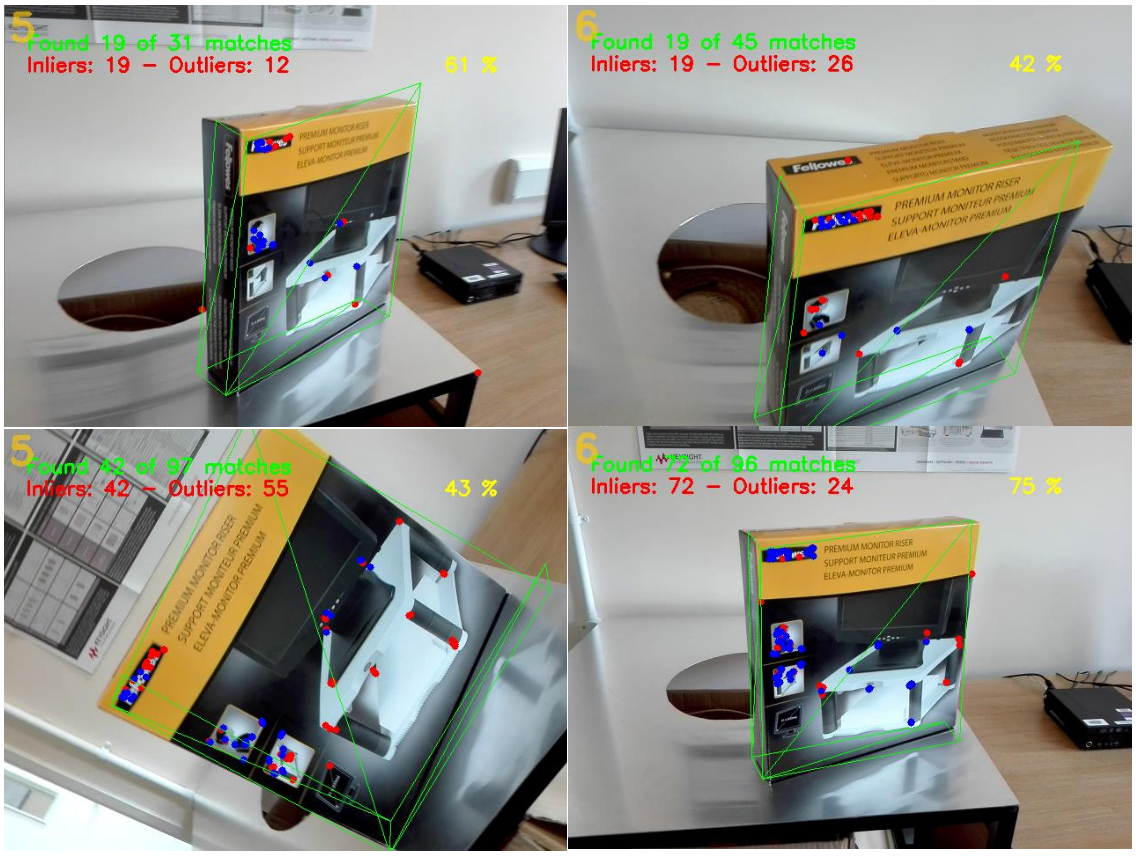


Figure 35 - Tracking examples with "monitorbox3.mp4" at various angles

First of all, the system successfully identifies and matches the features, even when shown at drastically different angles and distances, as shown in Figure 35. The rotation-invariant and scale-invariant properties of the feature descriptor that was chosen are indeed validated by our data set. The pose estimation is done correctly, as the wireframe of the model is being overlaid on top of the real object from the video feed.

However, there are some constraints that must be taken into account with the current system. The test footage was recorded on a 640x480 smartphone camera at 29fps rather than using high-speed and high-resolution cameras, both for flexibility and to potentially expose the weaknesses of the system. Two key points appear on the right half of Figure 34.

First of all, motion blur (top-right) is a very important issue for our application. Considering the real camera will be mounted on a moving drone, a considerable amount of motion blur will appear on our images if the camera is not able to handle high speeds. When this occurs, feature extraction and matching becomes a nigh impossible task. It is important to verify whether the process is able to handle fast moving objects, and if this is not the case, either an alternative solution must be found, or this must be compensated either via pre-processing methods, or hardware.

Another problem appears when the camera does not have a high dynamic range (HDR) and cannot offer enough contrast in dark areas or bright areas (bottom-right). If this is the case, our system will lose features to track when the object becomes too bright or too dark, whether because the camera changes its opening or because the object is simply too dark or bright for the camera. If these details are lost on a hardware level, feature extraction and matching becomes once again a difficult task. With these two constraints in mind, the properties of the camera being used a crucial for the success of the system.

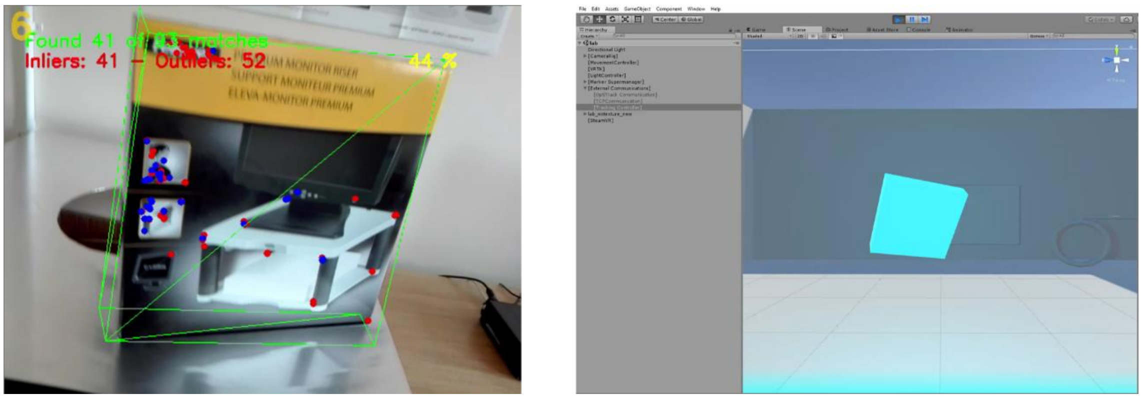


Figure 36 - Tracked object (left) and marker in Unity (right)

On the other hand, the connectivity and proper display of the information in Unity was successful, the information being correctly relayed between one application and the other. The time-out of the markers when no more information is received, fading animations, and correct placement by following the pseudo-drone in the scene all functioned as expected, and contributed to making a coherent and descriptive scene.

- **The user should be able to detach from the drone and explore the environment freely**

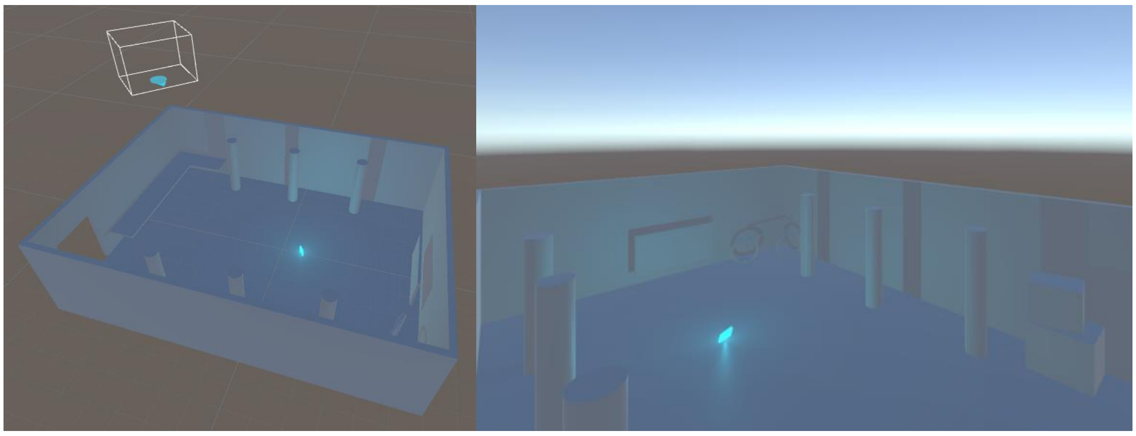


Figure 37 - User at vantage point, while drone tracking is performed elsewhere

As Figure 37 shows, the user can move around freely, while the tracked objects will be correctly spawned from the last known camera position. The controls have been tested extensively and proved to be a success among new users, who would get accustomed to the control scheme in the first minute of usage.

The detached user movement has also been tested while moving the pseudo-drone in the scene, and both tasks proved to not affect one another. The tracked objects were placed correctly in accord to the pseudo-drone's simulated feed, and the user could freely detach and watch the pseudo-drone's tracking move around the scene, or instantly snap back to the pseudo-drone's position at any point.

- **Total failure of the drone and object tracking modules should not affect the simulation module**
- **Modules should have a low degree of coupling upon implementation**

During testing, modules were forcibly closed, connections inadvertently cut off, and the drone would be brought out of OptiTrack's range. However, none of these failures affected other modules, and the system would keep running as expected in the different machines. All modules are successfully self-managed.

- **The application should offer customization options and some freedom to both the user and the programmer**
- **The application should be expandable for future applications**

During the research project, numerous efforts have been made to provide future customization options for both the user and the programmer. Here is a list of options that are available in Unity:

Movement

X/Y/Z speed, Rotation speed, Sprint multiplier, X/Y/Z sprint, rotation sprint, Gaze control, Controller haptic feedback

Light Controller

Ambient light, Night/Mode switch time, Day mode ambient/marker intensities, Night mode ambient/marker intensities

Markers

Marker name and prefabs, Model colour, Emitted light colour, Fade out time, Time out enabling, Time out time

Drone tracking

IP/Port, Target Rigid Body, Target Markers Origin, X/Y/Z rotation locks

Object tracking

IP/Port, Movement scale, Use of Quaternions

The application has been extensively tested by altering this settings throughout the research project in order to create a better user experience (by reducing motion sickness, enabling controls that seemingly useful or intuitive, etc).

The application on Unity's side is also expandable, offering an infrastructure ready to track multiple objects simultaneously from the same drone. While this is not implemented in the code, the application can be fairly easily made to accept multiple drones / sources of tracking, track them and place them in the environment.

The biggest limitation in the system is that due to time constraints, the object tracking module is not able to track multiple objects *simultaneously*. The infrastructure of the C++ code is already able to generate multiple poses and update the system with multiple objects of different types. However, the part of the program shown in Section 0 at the moment only generates the pose for one object. Nevertheless, this algorithm inside the program can be extended to multiple objects, or replace with another structure able to generate multiple poses at the same time.

Efficiency

For performance evaluation, we take into account the characteristics of the machines shown in Section 0.

Unity Performance

Unity has showed exceptionally good performance both in the VR Machine and in the Lab machine, through the entirety of the research project. The following is data from testing sessions with functionalities implemented in their final state.

Testing conditions

Machine: VR Machine

Unity version: Unity 5.6.1f1 Personal (64bit)

GPU: NVIDIA GeForce GTX 960 (2,016 MB VRAM) (10,157 MB Total)

CPU: Intel® Xeon® CPU E5-1630 v3 @ 3.70GHz (8 CPUs), ~3.7GHz

RAM: 16,384 MB

Session durations: 120 seconds

Comments: Constant user-input, viewing scene from all angles, rendering tracked object

Results

Average FPS: **105fps**

Minimum FPS: 101fps

Maximum FPS: 112fps

Average CPU time: **9.5ms**

Minimum CPU time: 9.2ms

Maximum CPU time: 10.1ms

Our objective was to achieve a frame rate of at least 60fps for the simulation module, including the VR computation and the input from other modules. However, the frame rate obtained during testing is almost twice as high as our target frame rate. The FPS remained consistently above 100, which ensures VR will be properly displayed in the HMD without straining the user.

Total computation time was also fast, consistently staying under 10ms. Overall, the system that was built in Unity is not demanding, and fits our target performance for this application.

OptiTrack Performance

OptiTrack and Motive have not shown to be a bottleneck in the Lab machine, despite its characteristics being significantly lower than those of the VR machine.

The OptiTrack cameras used in the laboratory are the Prime 17W model [51]. Characteristics are:

Name: OptiTrack Prime 17W

Image sensor: 1664×1088, 5.5 µm×5.5 µm pixel size, 30–360 FPS, 2.8ms latency

LEDs: 20 ultra-high power LEDs, 850nm IR

Lens: 6mm F#1.6, 850nm band-pass filter

Lens FOV: 70° horizontal, 49° vertical

As described in Section 0, our laboratory set-up uses 6 OptiTrack cameras, arranged in the pattern seen in Figure 19. The drone tracking was performed at a rate of **120fps**, above the maximum frame rate that Unity has given us over the course of the research project. Since the used cameras can go up to 360fps, the tracking frame rate has been completely constant at 120fps throughout all tests carried out.

Thanks to the OptiTrack hardware, the Motive software and the efficiency of the NatNetSDK, the connectivity between OptiTrack and Unity showed no bottlenecks, and the simulation module was extremely responsive, even when faced with extreme movements and rotations from the pseudo-drones. The only limitations imposed by our drone tracking set-up are the ones stated in Section 0, where the effective tracking area was a 7m x 6m x 2m box, restricting our vertical movements of the drone.

Overall, both of our efficiency goals for the drone tracking module were met: we obtained a framerate higher than 60fps for the tracking, and the delay between the drone tracking module and the simulation module remained well under 100ms, being near-instant when tested in laboratory conditions.

Object tracking Performance

Finally, the performance of the object tracking module has been close to our goals, but has shown signs of being fragile. The following charts are performance results examples from the footage recorded at the academy.

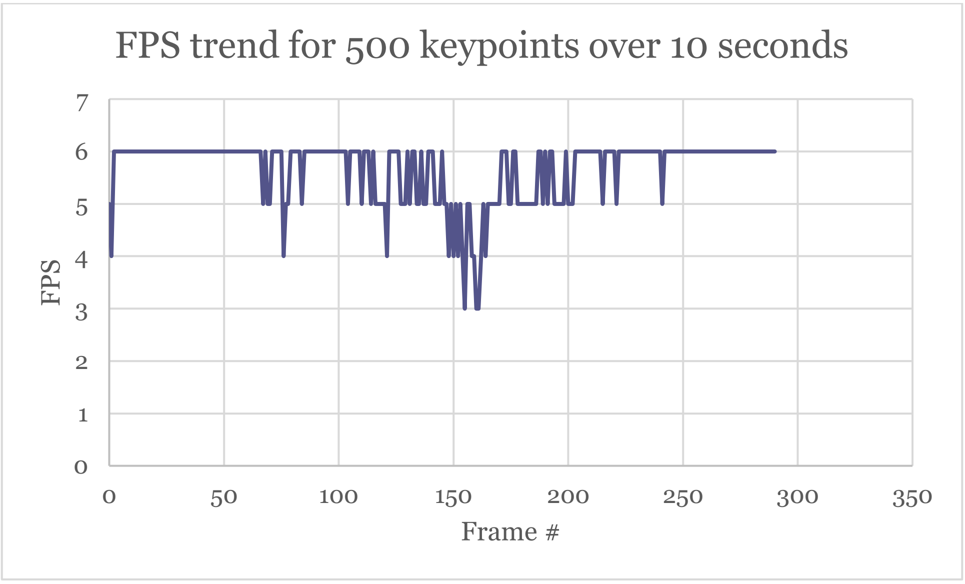


Figure 38 - FPS trend for "monitorbox3.mp4" with 500 keypoints

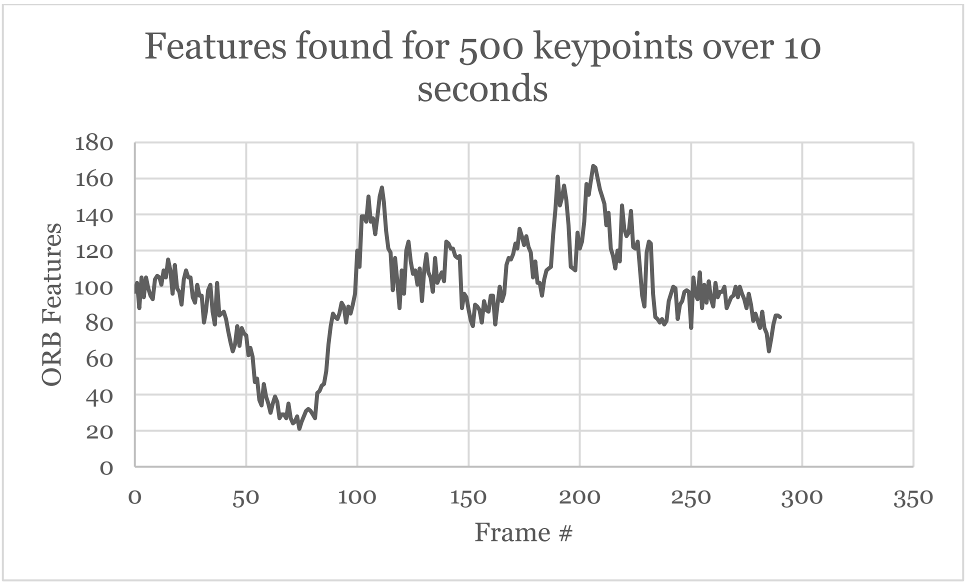


Figure 39 - Features trend for "monitorbox3.mp4" with 500 keypoints

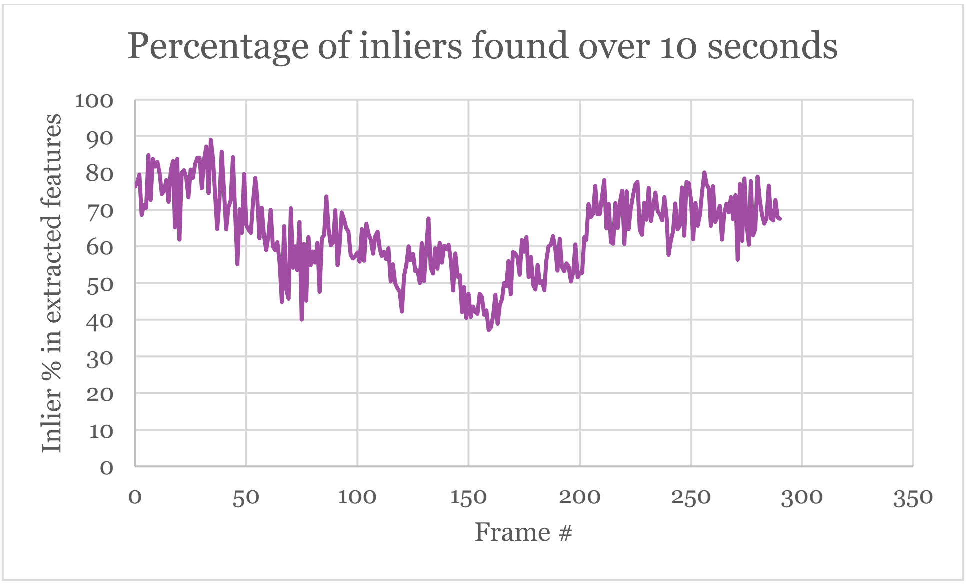


Figure 40 - Inlier % trend for "monitorbox3.mp4" with 500 keypoints

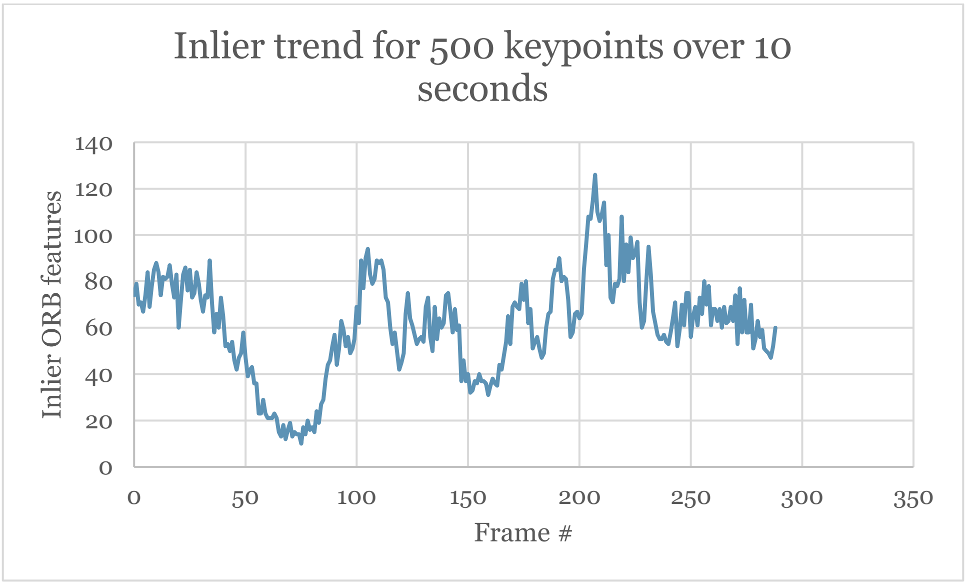


Figure 41 - Inlier trend for "monitorbox3.mp4" with 500 keypoints

Figure 38 - FPS trend for "monitorbox3.mp4" with 500 keypoints shows the frame rate of the entire object tracking module, over the span of 10 seconds for readability. In this case, we are dealing with the extraction of up to 500 keypoints. The behaviour of this trend is consistent over all our testing data. The maximum amount of fps that the system usually achieves with 500 keypoints is **6fps**, while averaging throughout the 1600+ frames of capture at **5fps** when the object is present. Generally, the system stays at our target fps of at least 5 fps, in the non-functional requirements.

However, at some points the frame rate dips under 4 fps. While this is acceptable in the overall context of our application, it risks not meeting our standards at certain moments. This behaviour is consistent with the percentage of inlier features found in the online registration phase, shown in Figure 40. Once the percentage of inliers drops under 45%, the pose estimation phase of the algorithm can take over 150ms in average, creating a bottleneck in the tracking pipeline. This could be avoided by using REPPnP, whose performance is stable with a high percentage of outliers, and seems to outperform every other alternative in that situation.

Computation time can be reduced by reducing the number of keypoints, the following charts are results after reducing the number of keypoints from 500 to 150:

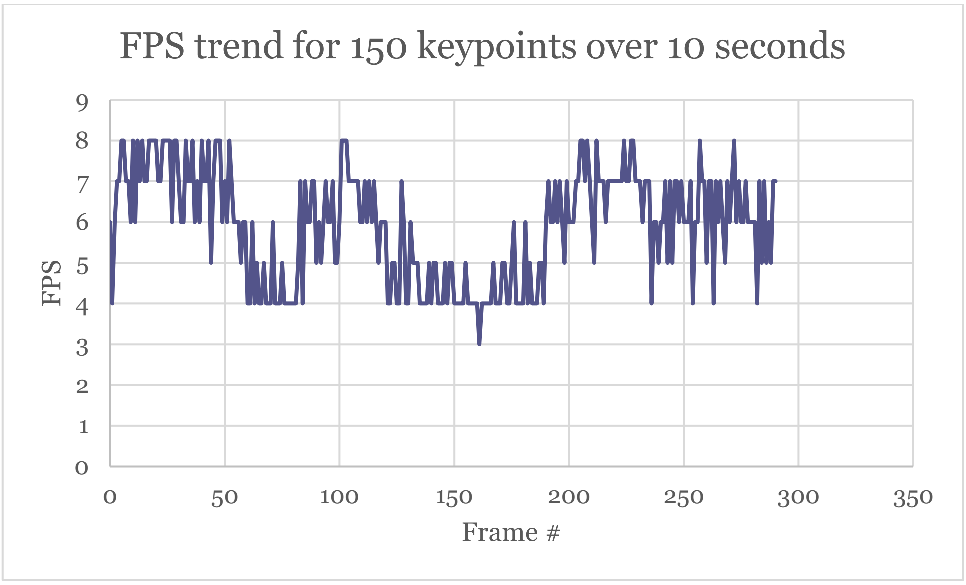


Figure 42 - FPS trend for "monitorbox3.mp4" with 150 keypoints

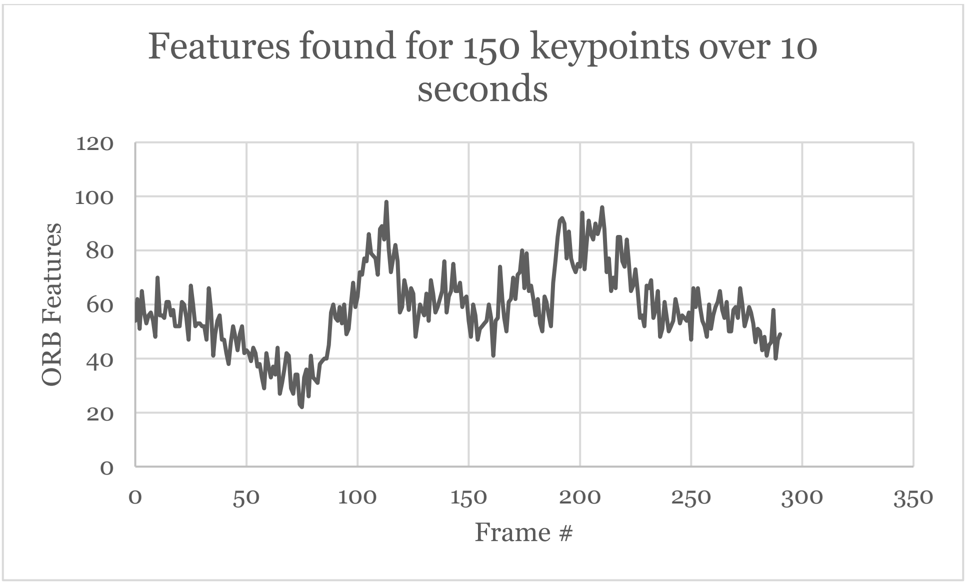


Figure 43 - Features trend for "monitorbox3.mp4" with 150 keypoints

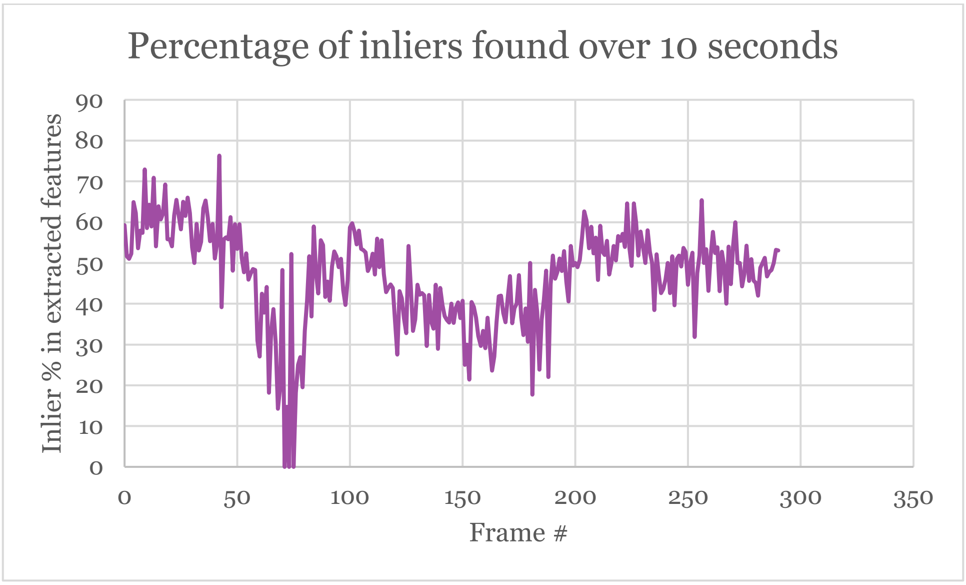


Figure 44 - Inlier % trend for "monitorbox3.mp4" with 150 keypoints

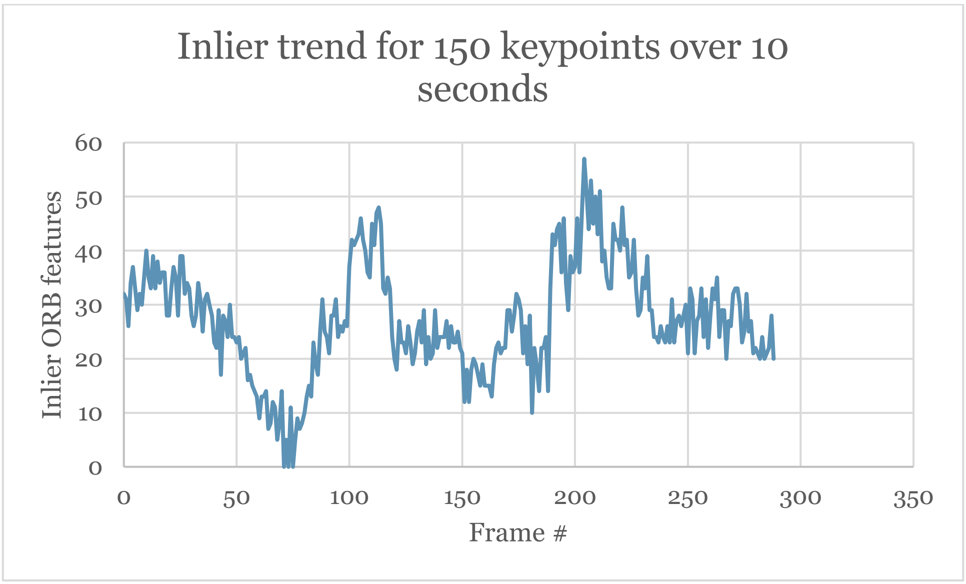


Figure 45 - Inlier trend for "monitorbox3.mp4" with 150 keypoints

When reducing the amount of keypoints to extract, match and use for pose estimation, performance is significantly increased as shown in Figure 42 in comparison to Figure 38. The application is now able to go up to **8fps**, and averages at **6fps**. However, this comes with an accuracy cost. Figure 44 shows that the percentage of inliers has drastically decreased. 500 keypoints showed an average of 52% inliers, whereas 150 keypoints only show a measly 38%. Because of this, the pose estimation with 150 keypoints is generally less refined, and is dangerously susceptible of losing track of the object completely (as seen around frame 70).

It is important to note that for some unknown reason, despite the characteristics of the VR machine, the object tracking module is not performing as expected. The object tracking module has been tested on multiple other devices, with a frame rate averaging well over 12fps despite the other devices having significantly lower characteristics. The

Lab machine seems to perform similarly to the VR machine. We are still not sure of what could be causing this effect, and this is something that needs to be investigated further.

As a side note, all of these results can be further improved by moving part of the code to the GPU. While this was not done due to time constraints, NVidia and OpenCV have CUDA tools at the programmer's disposal to this end.

6. Conclusion

Summary of work

The aim of this thesis was to present a system that exploits the advantages of Virtual Reality and Augmented Reality, and can be used as a framework for different navigation applications

A system based on three key modules has been built: a simulation module, which acts as the main environment and interface using Unity and VR; a drone tracking module, which allows a drone to be captured in real time and moved in the virtual environment thanks to OptiTrack; and an object tracking module based on AR technologies, which allows to augment the virtual environment in real time.

After continuous unit tests and integration tests, the fusion of these technologies not only has proven to offer a valid alternative to traditional interfaces, but it also provides a framework to exploit this AV environment for numerous other applications.

This AV system has multiple advantages. First and foremost, it allows the user to perform thorough spatial exploration tasks and sensible assessments of a situation, by being immersed in a simplified version of the environment they are monitoring. This mix of virtual presence and telepresence can serve as a decision-making tool, as it could allow a person to examine terrain they would not normally be able to examine.

Furthermore, the system has been built so that it can easily be exploited and extended to other applications. Not only because of the flexibility of the simulation module, but also thanks to the low coupling of the three modules. This allows the drone tracking or the object tracking module to be replaced with different technologies, depending on the application that is being developed.

We believe that this work can serve as the starting point for new fusions of technologies, and that this thesis has exposed new potentials of augmented virtuality, as well as offered the necessary tools in order to explore new areas of application for natural interfaces in spatial navigation.

Future work

One of the main lines of improvement for the research project is the improvement of the object tracking module. Further AR technologies can be explored in order to obtain the best results from the real environment. It is crucial to explore frameworks that allow tracking high amounts of objects in real time, considering the applications of our work.

Another improvement crucial to our intended application, is the extension of drone tracking from an indoors environment to an outdoors one. While our system was developed as a proof of concept, it is important to evaluate the implications of switching to outdoor technologies such as GPS or GLONASS. Meeting frame rate requirements is key, either through high refresh rates, or interpolation in Unity; and the impact of the accuracy when placing the vehicle in the virtual environment must also be assessed.

Simultaneous Localization and Mapping (SLAM) technologies can also be integrated into Unity, which is one of the interesting extensions of this project. The system would allow the exploration of uncharted environments directly in VR, to scale, and with simultaneous object tracking.

Furthermore, the object tracking module does not necessarily need to rely on AR. With the goal of augmenting the virtual environment to create an AV, the information acquired from the real environment could come from different types of sensors. Whether it is sound information, thermal information (for example projected onto the environment model), infrared signals; any kind of real information can be formatted and presented in the AV.

Finally, the most interesting extension of the project in our opinion, would be the ability to receive information from multiple tracking sources, and place them in the virtual environment. This would be a key advantage over traditional interfaces, since the simulation module could provide a shared environment which can be augmented simultaneously by different sources.

Other potential applications include for example exploring terrain around a spacecraft in 3D to plan routes; generation and visualisation of unmapped terrain with thermal information; or large scale monitoring in traffic security, urban areas, or terrain scouting.

REFERENCES

[1] E. Gobbetti and R. Scateni, “Virtual Reality: Past, Present and Future,” in *Virtual Environments in Clinical Psychology and Neuroscience: Methods and Techniques in Advanced Patient-Therapist Interaction*, 1998, pp. 3-20.

[2] The Economist, “The virtual curmudgeon,” *Technology Quarterly,* Q3 2010.

[3] J. Steuer, “Defining Virtual Reality:Dimensions Determining Telepresence,” *Journal of Communication,* vol. 42, no. 4, p. 73, Autumn 1992.

[4] Borko Fuhrt (Ed.), “Telepresence,” in *Encyclopedia of Multimedia*, Springer US, 2008, p. 849.

[5] Borko Fuhrt (Ed.), “Virtual Presence,” in *Encyclopedia of Multimedia*, Springer US, 2008, pp. 967-968.

[6] S. G. Weinbaum, “Pygmalion's Spectacles,” Project Gutemberg, June 1935. [Online]. Available: http://www.gutenberg.org/ebooks/22893. [Accessed July 2017].

[7] M. Heilig, “Sensorama”. Patent 3,050,870, August 1962.

[8] Wikipedia, “The Sword of Damocles (virtual reality),” [Online]. Available: https://en.wikipedia.org/wiki/The_Sword_of_Damocles_(virtual_reality) . [Accessed July 2017].

[9] I. E. Sutherland, “The Ultimate Display,” in *Proceedings of IFIP 65, vol 2*, 1965, pp. 506-508.

[10] R. Kalawski, The Science of Virtual Reality and Virtual Environments: A Technical, Scientific and Engineering Reference on Virtual Environments, Wokingham: Addison-Wesley, 1993.

[11] "Oculus Rift," Oculus VR, [Online]. Available: https://www.oculus.com/rift/. [Accessed July 2017].

[12] "HTC Vive," HTC Corporation, [Online]. Available: https://www.vive.com/. [Accessed July 2017].

[13] J. Vince, "Virtual Reality Techniques in Flight Simulation," in *Virtual Reality Systems*, Academic Press, 2014, pp. 135-141.

[14] J. Falah, S. F. Al-falah, T. Alfalah, S. Khan, W. Chan, D. K. Harrison and V. Charissis, "Virtual Reality medical training system for anatomy education," in *Science and Information Conference (SAI)*, London, 2014.

[15] W.D. Cannon, MD; W.E. Garrett Jr., MD, PhD; R.E. Hunter, MD; et al., "Improving Residency Training in Arthroscopic Knee Surgery with Use of a Virtual-Reality Simulator: A Randomized Blinded Study," *Journal of Bone & Joint Surgery - American Volume,* vol. 96, no. 21, p. 1798–1806, 2014.

[16] M. Nagendran, K. Gurusamy, R. Aggarwal, M. Loizidou and B. Davidson, "Virtual reality training for surgical trainees in laparoscopic surgery," Cochrane Database of Systematic Reviews 2013, Issue 8, 2013.

[17] D. Thalmann, "Using Virtual Reality techniques in the Animation Process," in *Virtual Reality Systems*, Academic Press, 2014, pp. 143-158.

[18] R. J. Stone, "Virtual Reality: A Tool for Telepresence and Human Factors Research," in *Virtual Reality Systems*, Academic Press, 2014, p. 181.

[19] R. T. Azuma, "A Survey of Augmented Reality," *Presence: Teleoperators and Virtual Environments,* vol. 6, no. 4, pp. 355-385, 1997.

[20] D. Van Krevelen and P. R, "A Survey of Augmented Reality," *The International Journal of Virtual Reality,* vol. 9, no. 2, pp. 1-20, 2010.

[21] I. Rabbi and S. Ullah, "A Survey on Augmented Reality Challenges and Tracking," *Acta Graphica znanstveni časopis za tiskarstvo i grafičke komunikacije,* vol. 24, no. 1-2, pp. 29-46, 2013.

[22] D. Wagner and D. Schmalstieg, “Making Augmented Reality Practical on Mobile Phones, Part 2,” *Computer Graphics and Aplications (IEEE),* no. 29, pp. 6-9, 2009.

[23] A. Fuhrmann, G. Hesina, F. Faure and M. Gervautz, “Occlusion in Collaborative Augmented Environments,” *Computers and Graphics,* vol. 23, no. 6, pp. 809-819, 1999.

[24] D. Stricker, G. Klinker and D. Reiners, “A fast and robust line-based optical tracker for augmented reality applications,” *IWAR '98,* pp. 31-46, 1998.

[25] J. Lima, F. Simões, L. Figueiredo and J. Kelner, “Model based markerless 3D tracking applied to augmented reality,” *Journal on 3D Interactive Systems,* vol. 1, 2010.

[26] I. Sutherland, “Three-dimensional data input by tablet,” *Proceedings of the IEEE,* vol. 62, no. 4, pp. 453-461, 1974.

[27] C. Lu, G. D. Hager and E. Mjolsness, “Fast and globally convergent pose estimation from video images,” *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 22, no. 6, pp. 610-622, 2000.

[28] F. Moreno Noguer, L. Vincent and P. Fua, “Accurate Non-Iterative O(n) Solution to the PnP Problem,” in *IEEE 11th International Conference on Computer Vision,* Rio de Janeiro, 2007.

[29] L. Shiqui, X. Chi and M. Ming, “A Robust O(n) Solution to the Perspective-n-Point Problem,” *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 34, no. 7, pp. 1444-1450, 2012.

[30] Y. Zheng, Y. Kuang, S. Sugimoto, K. Astrom and M. Okutomi, “Revisiting the PnP problem: A fast, general and optimal solution,” *Proceedings of the IEEE International Conference on Computer Vision,* pp. 2344-2351, 2013.

[31] L. Ferraz, X. Binefa and F. Moreno-Noguer, “Very fast solution to the PnP problem with algebraic outlier rejection,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition,* pp. 501-508,

2014.

[32] V. Teichrieb, J. Lima, E. Apolinário, et al., "A survey of online monocular markerless augmented reality," *International Journal of Modeling and Simulation for the Petroleum Industry,* vol. 1, no. 1, pp. 1-7, 2007.

[33] H. Wuest, F. Vial and D. Stricker, "Adaptive line tracking with multiple hypotheses for augmented reality," *Proceedings of the 4th International Symposium on Mixed and Augmented Reality,* pp. 62-69, 2005.

[34] L. Vaccheti, V. Lepetit and P. Fua, "Stable real-time 3d tracking using online and offline information," *IEEE transactions on pattern analysis and machine intelligence,* vol. 26, no. 10, pp. 1385-1391, 2004.

[35] I. Skrypnyk and D. Lowe, "Scene modelling, recognition and tracking with invariant image features," *Third IEEE and ACM International Symposium on Mixed and Augmented Reality,* pp. 110-119, 2004.

[36] P. Bouthemy, "A maximum likelihood framework for determining moving edges," *IEEE Transactions on pattern analysis and machine intelligence,* vol. 11, no. 5, pp. 499-511, 1989.

[37] D. G. Lowe, "Object recognition from local scale-invariant features," *The proceedings of the seventh IEEE niternational conference on Computer vision,* vol. 2, pp. 1150-1157, 1999.

[38] H. Bay, T. Tuytelaars and L. Van Gool, "Surf: Speeded up robust features," *Computer vision-ECCV 2006,* pp. 404-417, 2006.

[39] E. Rublee, V. Rabaud, K. Konolige and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," *IEEE international conference on Computer Vision,* pp. 2564-2571, 2011.

[40
] K. M. Yi, E. Trulls, V. Lepetit and P. Fua, "LIFT: Learned Invariant Feature Transform," *Europe Conference on Computer Vision 2016,* 2016.

[41] P. Milgram, H. Takemura, A. Utsumi and F. Kishino, "Augmented Reality:

A class of displays ont eh reality-virtuality continuum," *Proceedings of Telemanipulator and Telepresence Technologies,* vol. 2351, no. 11, pp. 282-292, 1994.

[42] R. Ott, M. Gutiérrez, D. Thalmann and F. Vexo, "Advanced virtual reality technologies for surveillance and security applications," *Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications,* pp. 163-170, 2006.

[43] X. Righetti, S. Cardin, D. Thalmann and F. Vexo, "Immersive flight for surveillance applications," in *IEEE Symposium on 3D User Interfaces*, 2007.

[44] P. Salomoni, C. Prandi, M. Roccetti, et al., "Diegetic user interfaces for virtual environments with HMDs: a user experience study with oculus rift," *Journal on Multimodal User Interfaces,* pp. 1-12, 2017.

[45] "Unity3D," Unity technologies, [Online]. Available: https://unity3d.com/. [Accessed August 2017].

[46] "Unity Documentation: Execution Order of Event Functions," Unity Technologies, [Online]. Available: https://docs.unity3d.com/Manual/ExecutionOrder.html.

[47] "Unity Documentation: Coroutines," Unity technologies, [Online]. Available: https://docs.unity3d.com/Manual/Coroutines.html.

[48] "Motion Capture Robotics," OptiTrack, [Online]. Available: https://optitrack.com/motion-capture-robotics/.

[49] "OpenCV library," OpenCV Team, [Online]. Available: http://opencv.org/. [Accessed August 2017].

[50] "Real time pose estimation of a textured objects," OpenCV Documentation, [Online]. Available: http://docs.opencv.org/3.1.0/dc/d2c/tutorial_real_time_pose.html. [Accessed August 2017].

[51] "OptiTrack Prime 17W," NaturalPoint, [Online]. Available: https://optitrack.com/products/prime-17w/. [Accessed August 2017].

APPENDICES

MovementController

Variable	Type	Description
Camera Rig	GameObject	[CameraRig] GameObject to move in the scene
Speed X/Y/Z	float	Speed factor for movement along each axis
Speed Rotation	float	Rotation speed factor around Y-axis
Sprint Multiplier	float	Speed multiplier when in “sprint” mode
X/Y/Z Sprint	bool	If true, sprint is enabled along each axis
Rotation Sprint	bool	If true, sprint is applied to rotation speed
Realistic Movement	bool	If true, movement uses Unity physics (not 100% convenient)
Gaze Control	bool	If true, forward = where the head camera is looking
Player Head Camera	GameObject	“Camera (head)” GameObject for gaze control

Table 0-1 [MovementController] variables

Function	Type	Description
moveRigBy(Vector3 distance)	void	Moves [CameraRig] in the desired distance
moveRigTo(Vector3 position)	void	Moves [CameraRig] to the desired position
rotateRigBy(Vector3 rotation)	void	Rotates [CameraRig] by the desired rotation in Euler angles
rotateRigTo(Vector3 coordinates)	void	Rotates [CameraRig] to the desired rotation in Euler angles
rotateRigTo(Quaternion coordinates)	void	Rotates [CameraRig] to the desired rotation in Quaternions
movePlayerForward/Backward()	void	Moves the [CameraRig] along the Z axis with the specified speed
movePlayerLeft/Right()	void	Moves the [CameraRig] along the X axis with the specified speed
movePlayerUp/Down()	void	Moves the [CameraRig] along the Y axis with the specified speed
changePlayerRotation(Vector3 rotationDelta)	void	Changes the [CameraRig] rotation by the specified Euler angles
sprintOn/Off()	void	Scales XYZ/r speed according to the Sprint Multiplier
ToggleGazeControl()	void	Toggles movement controlled by the “Camera (head)” orientation

Table 0-2 [MovementController] functions

Results of “monitorbox3.mp4” over 55 seconds, 500 keypoints

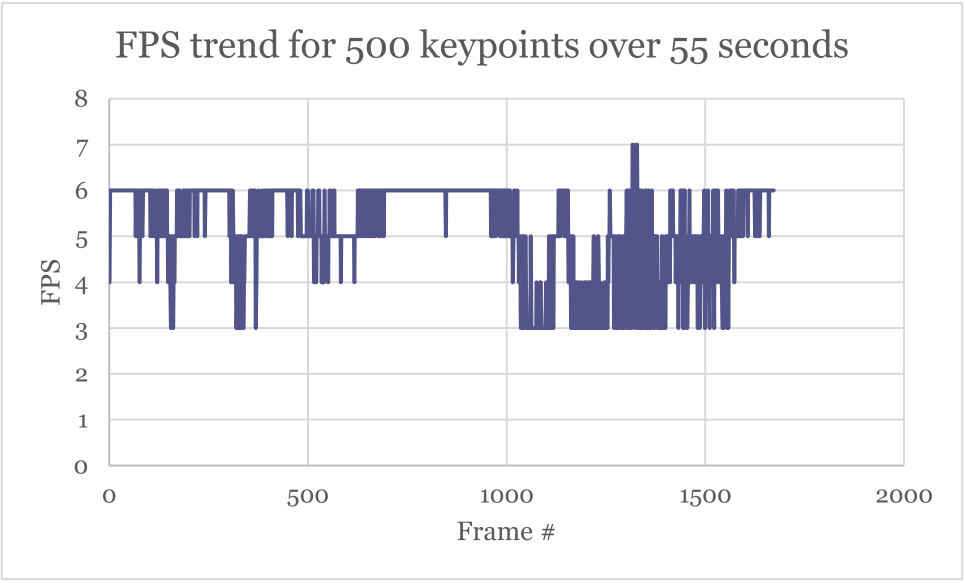


Figure 0-1 - FPS trend for "monitorbox3.mp4" over 55 seconds with 500 keypoints

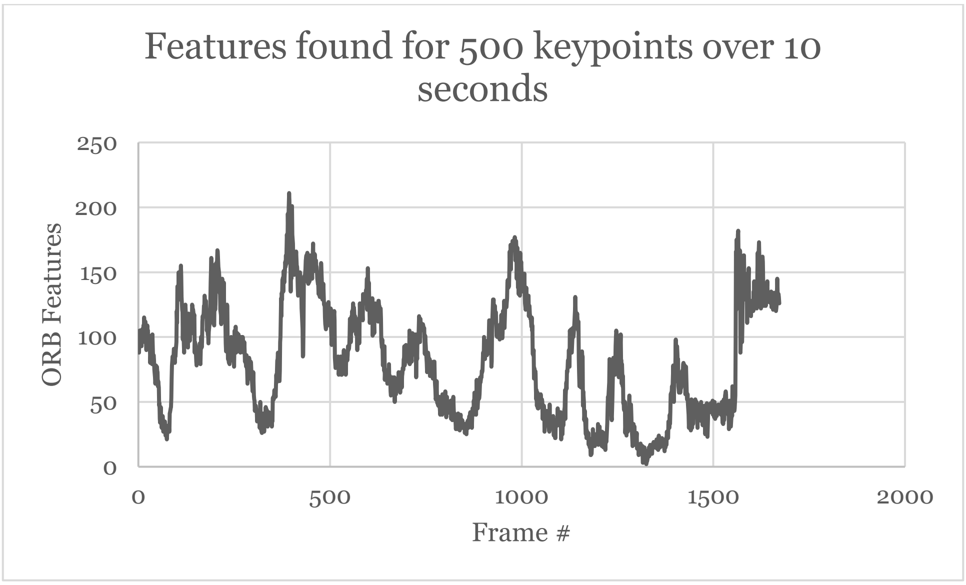


Figure 0-2 - Features trend for "monitorbox3.mp4" over 55 seconds with 500 keypoints

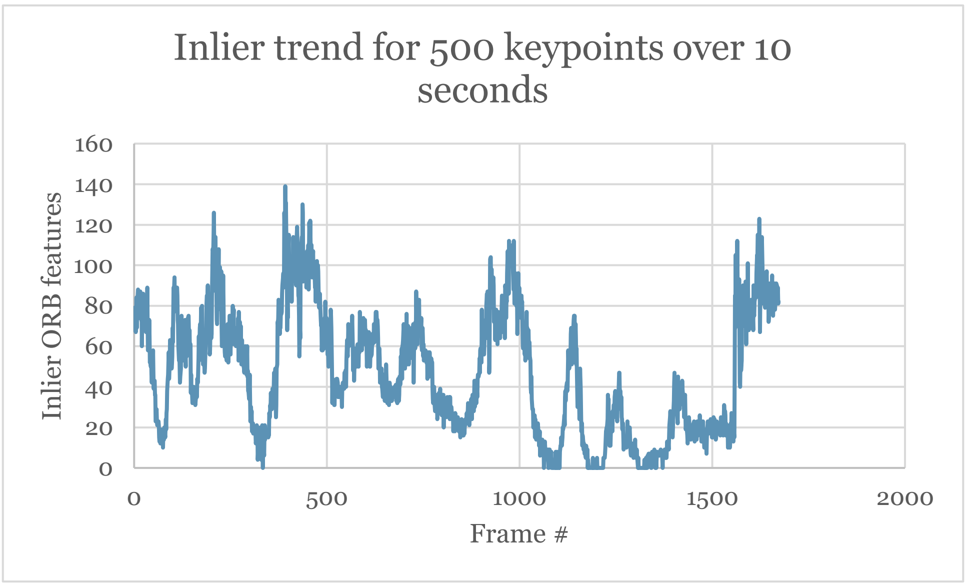


Figure 0-3 - Inlier trend for "monitorbox3.mp4" over 55 seconds with 500 keypoints

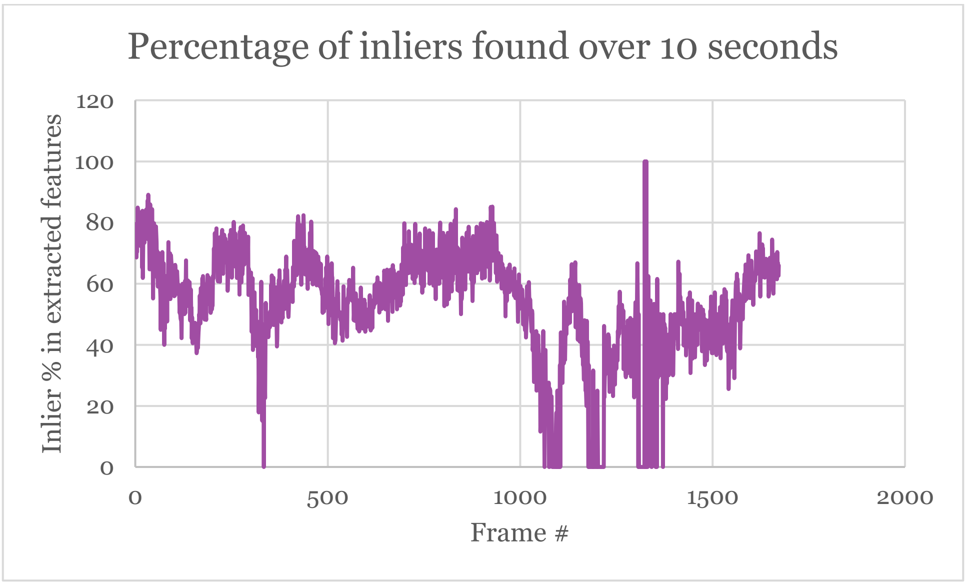


Figure 0-4 - Inlier % trend for "monitorbox3.mp4" over 55 seconds for 500 keypoints

Results of "monitorbox3.mp4" over 55 seconds, 150 keypoints

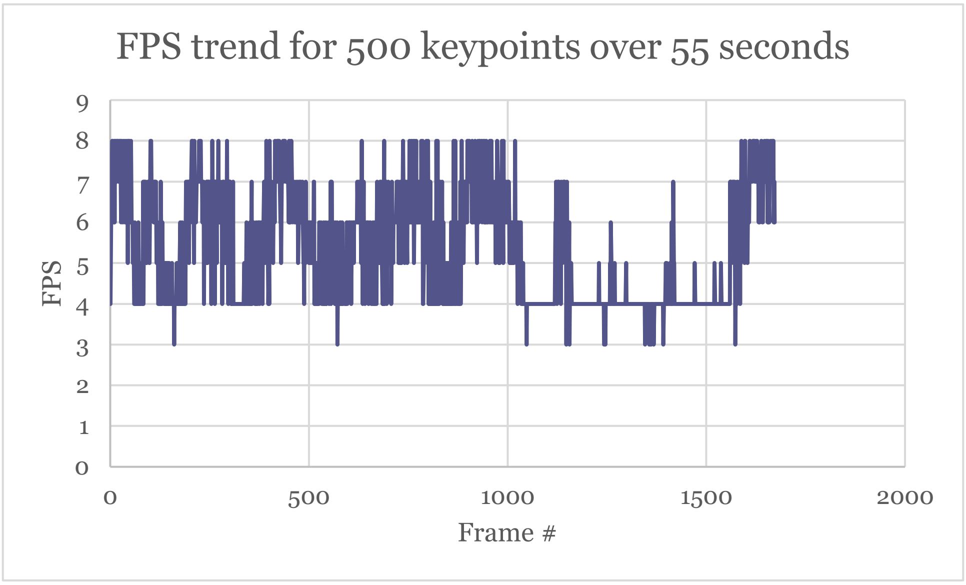


Figure 0-1 - FPS trend for "monitorbox3.mp4" over 55 seconds for 150 keypoints

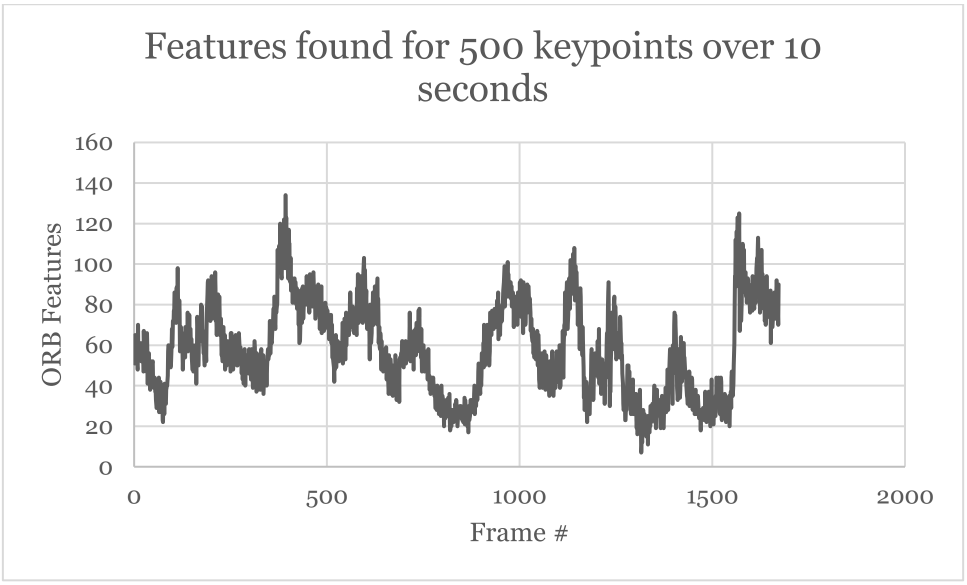


Figure 0-2 – Features trend for “monitorbox3.mp4” over 55 seconds for 150 keypoints

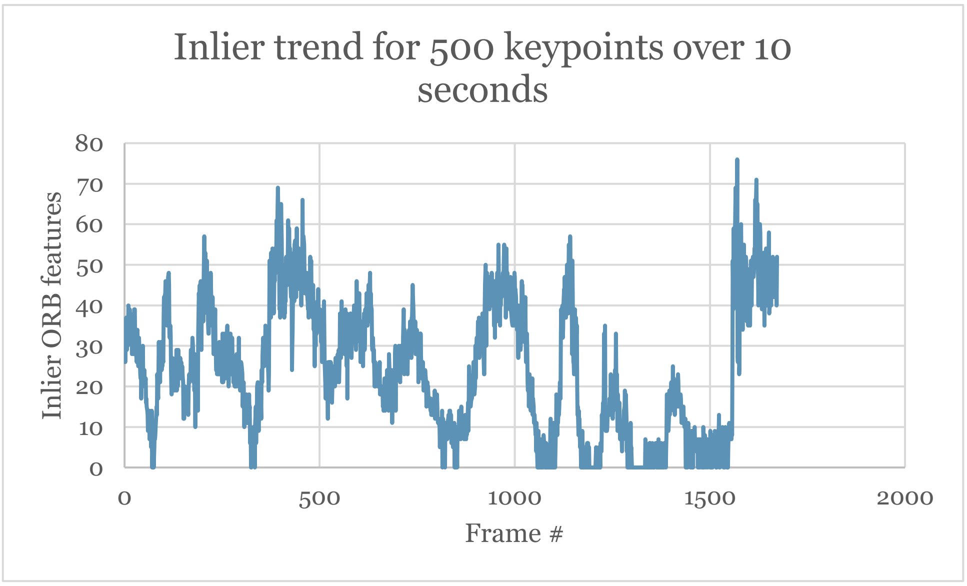


Figure 0-3 - Inlier trend for "monitorbox3.mp4" over 55 seconds for 150 keypoints

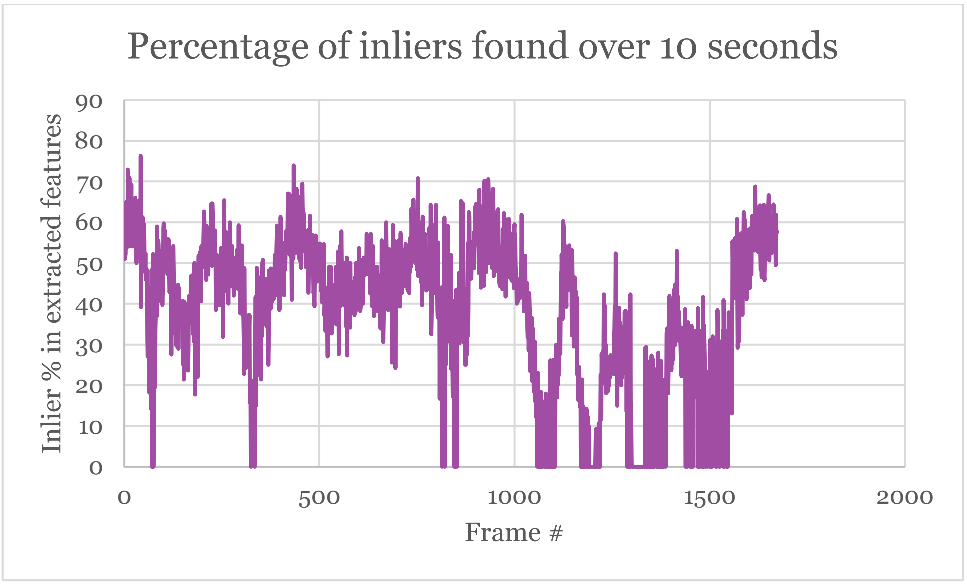


Figure 0-4 - Inlier % trend for "monitorbox3.mp4" over 55 seconds for 150 keypoints