RYERSON UNIVERSITY
FACULTY OF ENGINEERING, ARCHITECTURE AND SCIENCE

DEPARTMENT OF AEROSPACE ENGINEERING

Development of a 3D Holographic Flight Situational Awareness System

Jafer Mujtaba Kamoonpuri

AER 870 Aerospace Engineering Thesis

Faculty Advisor: Dr. Joon Chung
Date: February 28th, 2020

Abstract

Recent inventions of Augmented Reality (AR) Head-Mounted-Device (HMD) devices such as Microsoft's HoloLens have allowed certain innovations that up till now were only able to exist in Science Fiction. The ability to project holograms within a space have been used in the Aerospace industry since 2016, when the HoloLens was first released. However, the aviation industry has yet to harness the capability that such a device can allow. The conversion of a traditional 2D Primary Flight Display (PFD) to a Volumetric 3D representation of the PFD was explored. The 3D representation of the PFD was created in Unity 3D, and by means of the Holographic Remoting Tool the graphics were displayed onto the HoloLens. The symbology on the PFD was driven by live flight data from a flight simulator. For this project two different 3D PFD models were created one for a fixed-winged based aircraft, and another for a quadcopter. Two different flight simulators were used for the two different PFDs. For the fixed-winged PFD the Digital Combat Simulator (DCS) World by Eagle Dynamics was used, and for the quadcopter PFD the AirSim plugin by Microsoft was ran using Unreal Engine 4 (UE4). Through testing it was found that both the PFD models assist the pilots to safely keep their aircraft in the air and also perform an emergency landing by only using the 3D PFD. Another conclusion made was that in its current state the 3D PFD is ideal for Unmanned Arial Vehicle (UAV) pilots as a holographic Ground Control Station (GCS)

Acknowledgments

The author would like to thank his advisor, Dr. Joon Chung, for guidance and support in the last two years of his Undergraduate program, and for providing the space and environment to learn, innovate, and gain experience in the Aerospace industry. He would also like to thank his colleagues at the Mixed-Reality Immersive Simulation (MIMS) Laboratory for their additional support and guidance. Especially thanks to , Vamshi Chittaluri, Pratik Pradhan, Aditya Venkatesh, Lukasz Holownia, and Ricardo Ferreira Da Silva for volunteering as test subjects for this project.

Table of Contents

List of Figures

List Of Tables

Nomenclature

2D	2-Dimension
3D	3-Dimension
ADI	Attitude Direction Indicator
AR	Augmented Reality
BVLOS	Beyond Visual Line Of Sight
DCS	Digital Combat Simulator
FAA	Federal Aviation Administration
FAR	Federal Aviation Regulation
FOV	Field-Of-View
FPV	First-Person-View
GCS	Ground Control Station
HMD	Head-Mounted-Display
HMI	Human-Machine-Interface
IMU	Inertial Measurement Unit
MIMS	Mixed-Reality Immersive Motion Simulation
OOP	Object Oriented Programming
PC	Personal Computer
UAV	Unmanned Arial Vehicle
UE4	Unreal Engine 4
VR	Virtual Reality

1.0 Introduction

1.1 General

The objective of this thesis is to propose a new way of displaying flight information to pilots using a holographic Head-Mounted-Display (HMD). Current methods of displaying flight information to pilots is by projecting the information onto 2D screens or a 2D Head-Up-Display (HUD) panel in front of the pilot. Some methods that use 3D visuals are also projected onto 2D screens which result in loss of spatial information. Using HMDs such as Microsoft's HoloLens it is possible to display full 3D visuals without the loss of information as you would on traditional 2D screens. The loss of information mentioned here is further explained in section 1.6. The symbology created for this project is an example of how the traditional 2D symbology can be converted to 3D symbology to give a better understanding of an aircrafts attitude in flight.

For this thesis, the flight information is provided by Digital Combat Simulator (DCS) and a custom quadcopter simulator built on Unreal Engine 4 (UE4) based on Microsoft's open-source AirSim simulator. However, the flight information can be obtained from any flight simulator that allows the export of live flight data at runtime, any aircraft can also be used. The reason for using two vastly different flight models is to show how the same system can be used to assist both types of pilots. Using the two different types of aircrafts assisted in highlighting different use cases for a holographic based flight instrumentation panel. The pros and cons for both these systems are tested and discussed in section 4.0.

To complete the development of the holographic 3D HUD, many software, programming languages and computing hardware were involved. As mentioned earlier Microsoft's HoloLens was used as the HMD to display the holograms, but the main computational task is done on a separate PC. The PC is used to run the flight simulator and the software running the holographic instrument panel. There are several software available at the Mixed-Reality Immersive Motion Simulation (MIMS) Lab to create content that is viewable

on the HoloLens. Namely, Unity 3D, Unreal Engine 4, Microsoft Visual Studio, and BinariesLid's BuildWagon. For this thesis Unity 3D was chosen as the engine to create the 3D HUD

1.2 Head-Mounted-Display (HMD)

Head-Mounted-Displays (HMDs) are personal information-viewing devices that can provide information in a way that no other display can [2]. HMDs come in a variety of form factors to aid in different tasks. From viewing information, to augmenting the users view of the world, HMDs can have several ways of displaying information. Traditionally, HMD use in the aviation industry has been by fighter pilots. The HMDs are used not only to display flight information, but mission critical information as well. Outside aviation HMDs have been used in Medicine, Engineering, Law Enforcement, and many other professions. The use however, in all these professions have been the same. To display information to the user hands-free, and in a way that is always accessible. The information can be displayed as videos, texts, maps, or graphics [1].

1.3 Human-Machine Interface (HMI)

Human-Machine Interface (HMI) is a component of certain devices that are capable of handling human-machine interactions [3]. There are two types of HMI interactions, human-to-machine, and machine-to-human interactions. Both interactions are used in the completion of this project. The human-to-machine interactions occur when the user tries to pilot the aircraft in the flight simulator. The machine-to-human interactions occur when the HMD displays the instrument data to the user, so they can safely pilot the aircraft in the flight simulator. HMIs have two major components, an input component and an output component. For the interactions, the input goes into the output. In the case of human-to-machine interactions the human inputs information to the machine and vice-versa for machine-to-human interactions.

The HoloLens can receive inputs by many different methods. In the form of strings from a keyboard, or gestures provided by the user's hand for example. For the purpose of this project the main

method of interactions with the HoloLens are voice commands and gesture inputs. These inputs are for controlling what the user sees through the display of the HoloLens. For example, if the user wants to change the screen being displayed in front of the instrument panel (Figure 1), they would point the gaze cursor towards the holographic 'Change Screen' button and use the standard 'Air-Tap' hand gesture to click the button to change the screen. So, here the input is the hand gesture making the 'Air-Tap' motion, and the output is selecting the button in focus, namely the 'Change Screen' button to ultimately change the screen being displayed.

For the machine-to-human interaction between the HoloLens and the user an example are the buttons shown in Figure 1, and subtle lighting queues displayed in the holographic scene. When gazing at an object that is interactable the user will see a faint a white glow around the cursor indicating that the object can be interacted with. The input here is the faint glow, and the output of the user is the understanding that this object can be interacted with.

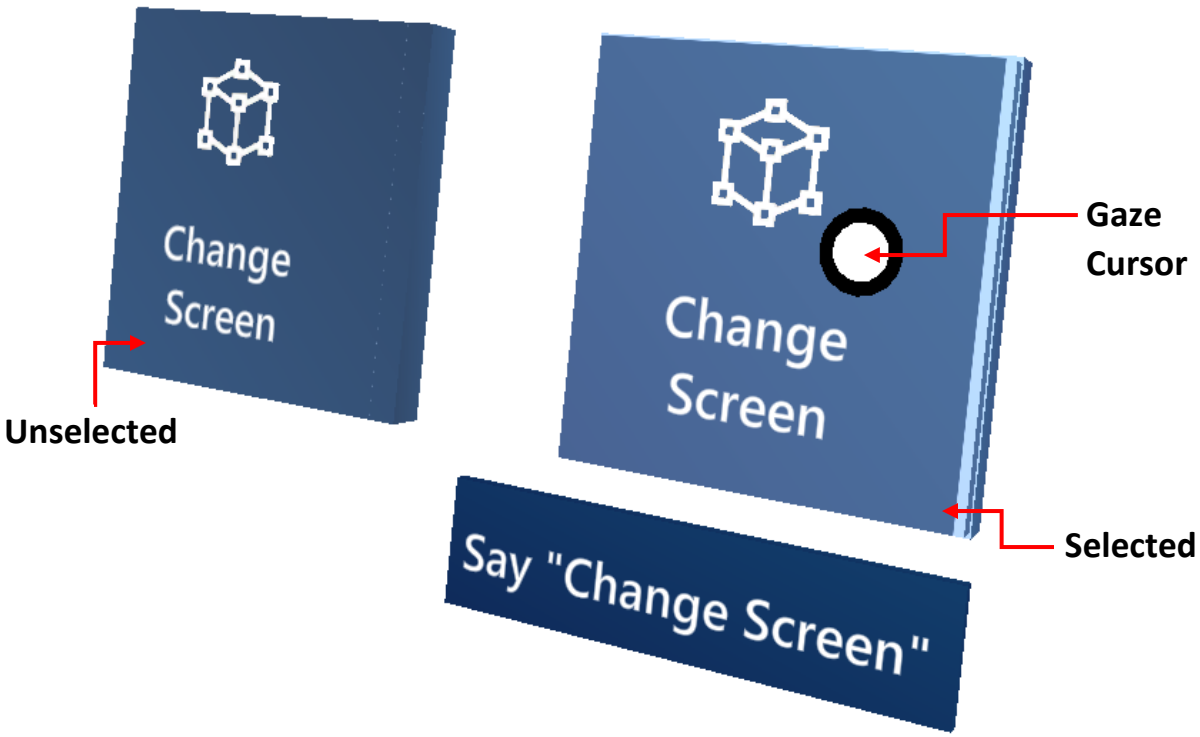


Figure 1. An example of holographic buttons used in the project

Between the flight simulator and the user, the human-to-machine interaction occurs when the user controls the aircraft using either mouse and keyboard or joystick. The user creates input using the joystick to create movement of the aircraft they are trying to control in way that makes sense to them. The flight simulator takes these inputs and interprets them to commands it can understand and changes the attitude of the aircraft accordingly. The machine-to-human interaction in this case is the flight information being fed to the user via the holographic instrument representation. The input is the data being presented to the user through the holographic instrument representation, and the output is the user's understanding of the aircrafts attitude in flight.

1.4 Human Factors Consideration

Human Factors refers to a body of knowledge about human limitations, human abilities, and other human characteristics, such as behaviour and motivation, that shall be considered in product design [4]. In the case of this project, the human factors consideration will be related to the what the user sees and experiences from the software side. The human factors on the hardware like the HoloLens, and joystick were conducted by their respective companies. There are number of ways the HoloLens has been made to be comfortable for the user when wearing it. For example, the head strap helps alleviate pressure from the forehead and the back of the head, which after prolonged use has been reported to be quite uncomfortable.

For this project the human factors considerations were made regarding how much information we can fit within the limited Field-Of-View (FOV) of the HoloLens. There are several criteria that had to be met in order to ensure that the user was provided with information in an efficient, helpful, and in a way that reduced the amount of load they can handle while flying an aircraft. These criteria were:

- Fit instruments related to the attitude and navigation outlined in the FAR 25.1321 guideline within the available FOV

- If the user must look away from the main instrument representation then have a secondary method of displaying the flight data that will provide the necessary information to maintain safe flight, but also not so obtrusive as to block their vision
- Use colors that are easy to view and read for the user
- Ensure all texts and symbology are crisp and easy to read and identify
- Minimize interactions that will require the user to use their hands for anything other than controlling the aircraft

There are also several factors that are specifically involved when dealing with AR/VR HMDs to ensure the user is comfortable. Motion sickness or spatial disorientation are usually the first things that are discussed when talking about VR. This is because of the disconnect the user experiences between what they see and what they feel. This disconnect is prevalent when the motion of the VR user and the virtual scene do not match. This is bit different when it comes to AR, however. The disconnect in wearable AR devices occurs when the user's eyes must keep adjusting focus on near and far objects. If the user's eyes must constantly change focus in and out of the AR scene, this could cause headaches. In order to mitigate this issue, the AR objects must be placed at a comfortable distance away from the user's eyes, or about the same distance as other objects in the user's environment. This would reduce the amount of adjusting the user's eyes have to do, thus reducing the chances of getting a headache.

1.5 Augmented Reality (AR) / Holograms

We define Augmented reality (AR) as a real-time direct or indirect view of a physical real-world environment that has been enhanced/augmented by adding virtual computer-generated information to it [5]. Another way to describe AR is to add computer-generated information to a user's vision or hearing to change reality of the user. In this report, when holograms are mentioned what is essential meant is this 'computer-generated information' being displayed through the HoloLens in the form of spatially anchored 3D and 2D object, numbers and texts. For this project the information being displayed is flight information in both 2D and 3D format. The information is anchored spatially in whatever space it being viewed. This

means that the holograms that are being displayed can be walked around and inspected. The holograms of the 3D flight instruments are created in Unity 3D and driven by real-time flight data from DCS World.

1.6 2-Dimensional vs Volumetric 3-Dimensional Holograms

There are two types of holograms that can be displayed on the HoloLens. 2D holograms are 3D object represented on single plane. Although these types of holograms can be anchored in space and walked around, they do not have volume. These types of holograms a good for displaying text, marks, and symbols. This is the same as displaying a 3D object on a 2D screen. In Figure 3 the 3D synthetic vision of the terrain is displayed on a 2D screen. This causes the loss of information that is not visible outside the Field-Of-View (FOV). Being able to view 3D objects in a 3D environment is one of the fundamental reasons devices like the HoloLens are ideal for this project. On the HoloLens the synthetic vision terrain model could be rendered around the user. In order to view the terrain behind or beside the the device, all the user has to do is turn their head.

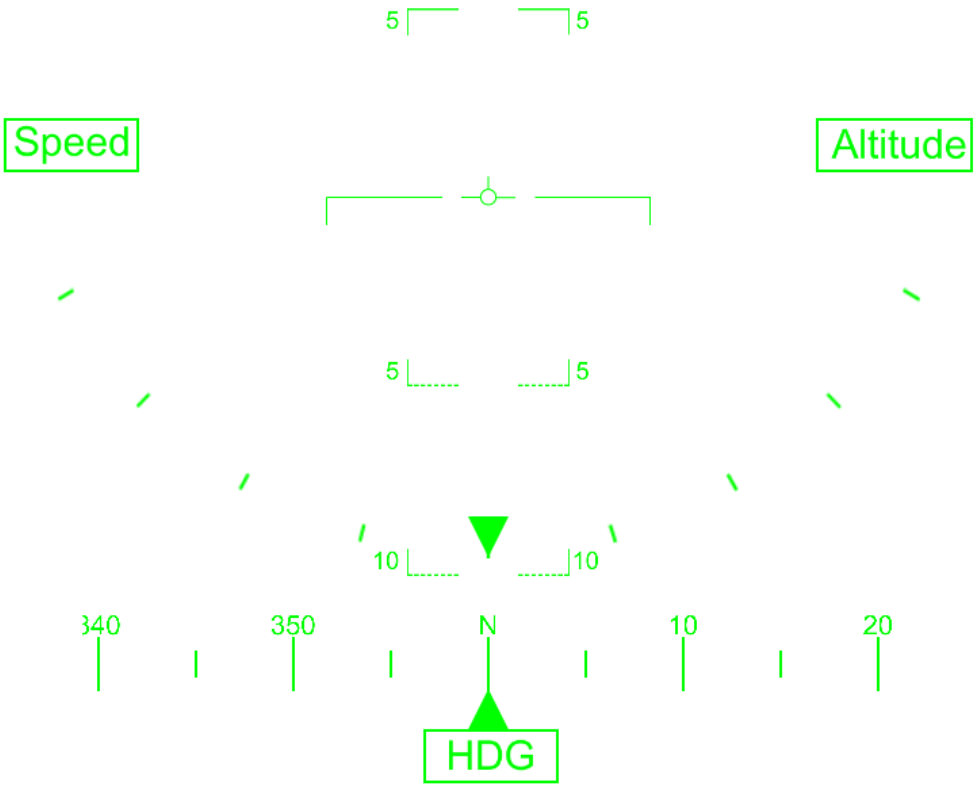


Figure 2. Example of a 2D flight symbology

Figure 3. Synthetic vision display bye GRT Avionics on the Horizon EX EFIS (Source: http://grtavionics.com/media/sport-ex-sv-fd.png)

Volumetric 3D holograms are 3D objects with volume. These 3D object not only have length and width but also depth. These types of holograms are ideal for representing physical objects in the virtual environment. The physical representation of the aircraft helps show the orientation of the aircraft. Using just one object displays both the pitch and roll of the aircraft in a more intuitive manner, similar to the ADI.

1.7 FAR 25.1321

FAR 25.1321 is a part of a set of regulations set up by the Federal Aviation Administration (FAA). FAR 25.1321 specifically deals with regulations regarding the arrangement and visibility of instruments used by a pilot. For this study a portion of the rules outlined in the FAR 25.1321 are followed to ensure the 3D instrumentation is easy for pilots to get accustomed to. The specific rules being focused on for this study are:

1. The instrument that most effectively indicates attitude must be on the panel in the top center position [6]

2. The instrument that most effectively indicates airspeed must be adjacent to and directly to the left of the Instrument in the top center position [6]
3. The instrument that most effectively indicates altitude must be adjacent to and directly to the right of the instrument in the top center position [6]
4. The instrument that most effectively indicates direction of flight must be adjacent to and directly below the instrument in the top center position [6]

1.8 Mission Objectives

The purpose of this project is to show how the 3D holographic environment created by AR devices such as the HoloLens can be leveraged to improve the current methods of displaying flight data to a pilot. What this report will present are examples of how traditional 2D flight information can be represented in a 3D environment. This report will also discuss the benefits and shortcomings of a 3D instrument representation. The following objectives will be completed for this project:

1. Conversion of 2D instrument symbology to volumetric 3D symbology
2. Testing the new 3D symbology with flight data being fed by a readily available flight simulator
3. Compare different 3D software capabilities in their ease-of-use, and compatibility with current and next-gen software
4. Conform the instrumentation with current instrumentation regulations (FAR 25.1321)
5. Comment on how current instrumentation could be adopted for 3D instrument representation

1.9 Scope of the Research

The scope of this report includes the conceptualization, design, creation, and testing of a volumetric 3D flight instrumentation representation. The report will discuss the reasoning behind the idea of creating a 3D instrument representation, the pros and cons of such a concept, how it conforms to current instrumentation related regulations, and areas it can be improved in future studies. This report will explain from the ground up how the system works, and how it can be used in real world scenarios.

2.0 Resources

2.1 Hardware

There are three main hardware components that were required for the completion of the project. These were:

1. Microsoft's HoloLens
2. A Personal Computer (PC)
3. An internet Router

Although the HoloLens is a standalone PC device, in the case of this project it was only used a display device via the Holographic Remoting Tool. This is discussed in further detail later in the report.

2.1.1 Microsoft HoloLens

The HoloLens headset enables users to see and manipulate holograms embedded in their environment [7]. The HoloLens, as mentioned earlier, is a wireless standalone holographic computer. HoloLens combines optics and sensors to provide holographic object that are anchored to the real world []. The HoloLens uses its combination of multiple image sensors and an Inertial Measurement Unit (IMU) sensor (Figure 4) to continuously spatially map the users surrounding environment. These sensors are also used to determine the position and orientation of the headset in space. The combination of the spatially mapped mesh and positional data of the headset are what help give the illusion of anchored holograms.

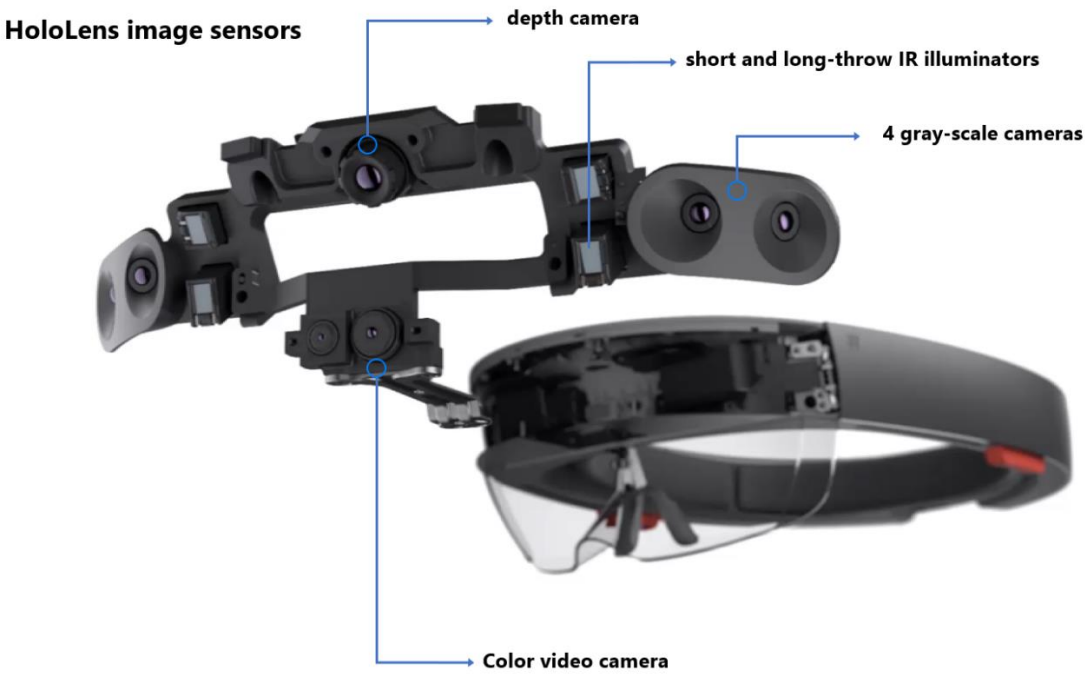


Figure 4. Labeled image of optical sensors used in the HoloLens (Source: https://wevolver-project-images.s3.amazonaws.com/0.6e98efojf2iHololens-image1.png)

The ability to anchor holograms is a key feature that was used in the completion of this project. Anchoring the instrument panel allows the user to walk around the instrument panel. For Unmanned Arial Vehicle (UAV) pilots this can be a way for them to take a closer look at the Point-Of-View (POV) screen. The ability to track the headset is also used in the completion of this project. The headset positional data is used to determine what the pilot is looking at. This will be discussed further in later sections of this report.

2.1.2 PC

Due to the different programs that are required to run simultaneously in order to achieve the objectives of this project, the PC used required certain minimum specifications. Running computationally intensive applications such as DCS World, Unreal Engine 4, and Unity each require their own minimum specifications. The most intensive of these applications is DCS World, which was used to set the base of the minimum specs required to complete this project. The minimum, required, and actual system specifications can be seen in Table 1. However, these specifications are required to run this single

application smoothly, but in order to complete this project we must run other programs at the same time. This requires the PC to have slightly better components than listed by the creators of DCS World.

	Minimum Specs	Recommended Specs	Actual Specs
Operating System 64-bit	Windows 7/8/10	Windows 8/10	Windows 10 Pro
Central Processing Unit (CPU)	Intel Cor i3 at 2.8 GHz / AMD FX	Intel Core i5+ at 3+ GHz / AMD FX or AMD Ryzen	AMD Ryzen 9 3900X at 3.79GHz
Random Access Memory (RAM)	8Gb (16Gb for heavy missions)	16Gb (32Gb for heavy missions)	64Gb
Storage Space	60Gb (HDD)	120Gb (SSD)	500Gb (SSD)
Graphics Processing Unit (GPU)	NVIDIA GeForce GTX 760 / AMD R9 280X	NVIDIA GeForce GTX 1070 / AMD Radeon RX Vega 56 with 8Gb VRAM	2x NVIDIA GeForce RTX 2080s in SLI with 8Gb VRAM
Peripherals	None	Joystick	Joystick

Table 1. Table comparing PC specifications required and used in the project

2.1.3 Internet Router

The communication between the PC and HoloLens occurs due to a built-in feature of the HoloLens called the Holographic Remoting Tool. For this tool to function, a stable internet connection is required. And in order to have a stable internet connection a decent internet router is required. How capable of an internet router needed is determined by how much graphical information need to be being sent from the PC to the HoloLens. Simple 2D texts, and images can be sent comfortably over a phone's hotspot. However, sending an entire scene with multiple 3D objects anchored in space requires a much higher bandwidth, which requires a more powerful internet router. For this project an average home router was used. This is due to accessibility as well as the graphical requirements. Certain steps were taken graphically to ensure the application ran smooth and stable.

2.2 Software

There are multiple software involved in the completion of this project. Two different flight information software were used to demonstrate two different use cases of the holographic flight display program. DCS World was used to demonstrate the program with fixed winged aircraft, and Unreal Engine

4 with Microsoft's AirSim plugin was used to demonstrate the same program but with a quadcopter. The Unity game engine was used to create the 3D instrument representation that is displayed on the HoloLens via the Holographic Remoting Tool.

2.2.1 Unity Game Engine

The Unity game engine is a relatively new 3D game engine initially released in 2005. Unity has since seen rapid and continuing growth in the gaming industry. Unity is generally seen as good entry level game editing software for new and small game developers. It's C# based scripting backend can be considered difficult for those who do not have a programming background. However, many plugins and assets sold in the Unity Asset Store help mitigate this issue, as it provides developers with polished features than can be added to their projects with a simple drag-and-drop. For individuals with a programming background Unity is a great starting point for game development as C# is one of the more easier programming languages to learn. C#'s Object-Oriented-Programming (OOP) model is an intuitive fit when scripting certain behaviours in-game.

Although Unity has predominantly been used as a game engine, in recent years many non-game uses have also been successful. Architects for example have used Unity to showcase their designs using Unity's real-time rendering capabilities [8]. At the MIMS lab itself Unity has been used to create non-game AR/VR experiences showcasing how these technologies can be used in the Aerospace Industry. The ease of use and extensive documentation have been key to the success of Unity.

For this project, Unity is used to create the 3D holographic flight instrumentation. The graphics created in the Unity Editor is piped through the Holographic Remoting Tool. The Holographic Remoting Tool is part of a toolkit created by Microsoft called the Mixed Reality Toolkit (MRTK) updated to the latest version 2. Other than providing the Holographic Remoting Tool, the MRTK v2 provides pre-built buttons and other such interactive elements that are compatible directly with the HoloLens.

2.2.2 Holographic Remoting Tool

As mentioned in section 2.2.1 the Holographic Remoting Tool is part of the MRTK v2. The main purpose of this tool is to provide developers a way to test their applications efficiently. The remoting tool has multiple emulation modes, simulate in editor and remote to device. Simulate in editor mode allows developers to test their application out, in either prebuilt rooms or in rooms that have been previously scanned using the HoloLens, within the editor itself. The remote to device mode enables Unity to send the graphics generated within the editor to the HoloLens over Wi-Fi. The communication over Wi-Fi does not happen one way. While Unity send the graphics information to HoloLens, HoloLens sends spatial and positional data back to Unity. This process essentially allows Unity to map the environment the HoloLens is in and anchor the holograms accordingly. This second mode (remote to device) is what is used to test the instrumentation design in this project.

2.2.3 Unreal Engine 4

Unreal Engine 4 (UE4) is another popular game engine used by large AAA game developers. This engine was first launched in 1998 and is currently in its 4th iteration as the name suggests. Unreal Engine is renowned for its realistic graphics and real-time rendering. The Unreal Engine like Unity is currently being used for more than just game development. The base scripting language of Unreal Engine is C++. Epic Games, the Corporation that own the IP of Unreal Engine, recognized that not all game developers and users of the engine are proficient in C++. This prompted them to create a visual node-based scripting model which is a lot easier for designers to start using. Epic Games calls this visual scripting model Blueprints. Developers can choose to use either the C++ base or Blueprint base or even a hybrid combination of both bases to create their program. Like Unity, UE4 has an asset store called Marketplace where developers can purchase user created content and plugins. These contents help developers save time and effort when creating their applications. For this project UE4 is predominantly used as a base software to run the AirSim plugin. Custom C++ and blueprint code were added to export real-time flight data from the quadcopter included in the AirSim plugin.

2.2.4 Digital Combat Simulator World

Digital Combat Simulator (DCS) World is a ‘combat flight simulator’. One of the main reasons for using DCS World is that it is free-to-play, with three free aircraft models. Additional aircraft models must be purchased. However, for this project the actual aircraft was inconsequential. Another reason DCS World was used is the ability to add extra capabilities, such as exporting live flight data. This is done by adding a script called Export.lua to the simulators, project folder. Any additional capabilities that are added to or removed from DCS World are written in the Lua programming language.

2.2.5 Microsoft’s AirSim

AirSim is an open-source, cross platform simulation software created in UE4 by Microsoft. Being open-source and having been built in UE4 allows developers the freedom to use and edit the simulator for whatever purpose they see fit. AirSim also includes several ways of including Machine Learning (ML). There are two simulation models included in the AirSim plugin, these are a car simulator and a quadcopter simulator. The Quadcopter simulation model is used for this project. The only control inputs accepted by the quadcopter simulator are gaming controllers, Pixhawk 4 (PX4) flight controller or a DJI controller [9]. For this project a Play Station 3 controller was used to fly the quadcopter. This was purely due to convenience as access to a PX4 controller was not feasible at the time of testing.

2.3 Pilots

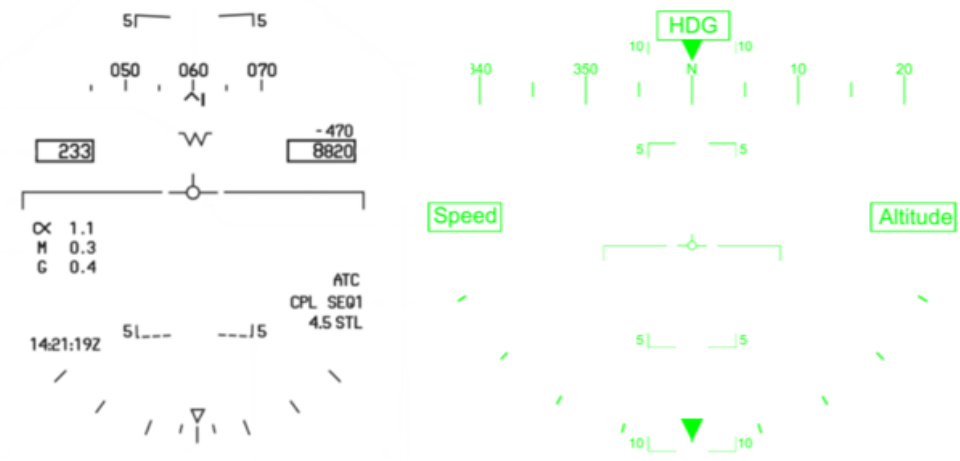
Whenever pilots are mentioned in the testing process in this report, it refers to student volunteers that work in the MIMS lab or have visited the MIMS lab. Flight experience of the student volunteers who tried the 3D HUD have flight experience ranging from having piloting licences to never having flown even on flight simulators. This allowed for feedback from a wide range of perspectives.

3.0 Design Concept

The idea behind this study came from the need to leverage the holographic capabilities of the HoloLens in the Aerospace industry. After exploring various use cases of the HoloLens in Aerospace, it

was clear that a device such as the HoloLens can provide a method of displaying information in way that has never been capable before its invention. The ability to display 3D objects around a user and being able to interact with them opens a world of possibilities. Since the HoloLens' release back in 2016, several industries have tried to adopt the new technology. Some have been successful, while others were not seen as viable uses. Within the Aerospace Industry itself the HoloLens has been adopted in numerous ways. Majority of the uses in Aerospace have been on the factory and maintenance floor or design rooms. Not many of the use cases have focused on the aviation side. Therefore, the focus of this research has been to create a way that can leverage a device such as the HoloLens to assist pilots in flight.

The initial prototypes were created for fixed-winged aircraft pilots, specifically air force pilots. The reasoning for this was that air force pilots already use HMDs in flight, so the road to adoption is slightly easier when compared to civilian pilots who might view wearing HMDs as a hassle. For this reason, some of the design is based on the Heads-Up-Display (HUD) similar to those used on the McDonnell Douglas F/A-18 Hornet (F-18). The comparison between the F-18 and early designs on the holographic HUD can be seen in Figure 4. The initial designs were meant to mimic what a fighter pilot would see in their HMD. This is to ensure the final design does not stray too far from the type of information the pilots would expect to see.



(a) (b)

Figure 5. Comparison between the (a) F-18 HUD symbology (Source: https://forums.vrsimulations.com/support/images/thumb/6/63/HUD.png/700px-HUD.png) and (b) 2D base symbology used as a reference to convert to 3D symbology

After testing the system, seen in Figure 4, it was clear that more could be done to utilize the 3-Dimensional capabilities of the HoloLens. So, as an initial step, and for this report, the 2D Attitude Direction Indicator (ADI) symbology was converted to a 3D symbology. The roll and pitch of the aircraft was now shown as a moving 3D representation of the aircraft in the center of the display. After more testing it was clear that this type of symbology would benefit drone pilots as well.

There is no guideline or procedure when it comes to converting 2D flight symbology to 3D flight symbology, other than the one created for this study shown by the flowchart in Figure 6. In order to do such a conversion requires some creativity and testing. Reaching the final design for this project required numerous iterations and testing. The goal was to create a 3D symbology that can assist a pilot in operating an aircraft without the need of additional displays and without the out-of-window view. This was to test if the information conveyed by the new symbology was enough for pilots to keep their aircraft in the air. Once the design of the HUD for fixed-winged aircrafts was completed, it was then tested with a quadcopter.

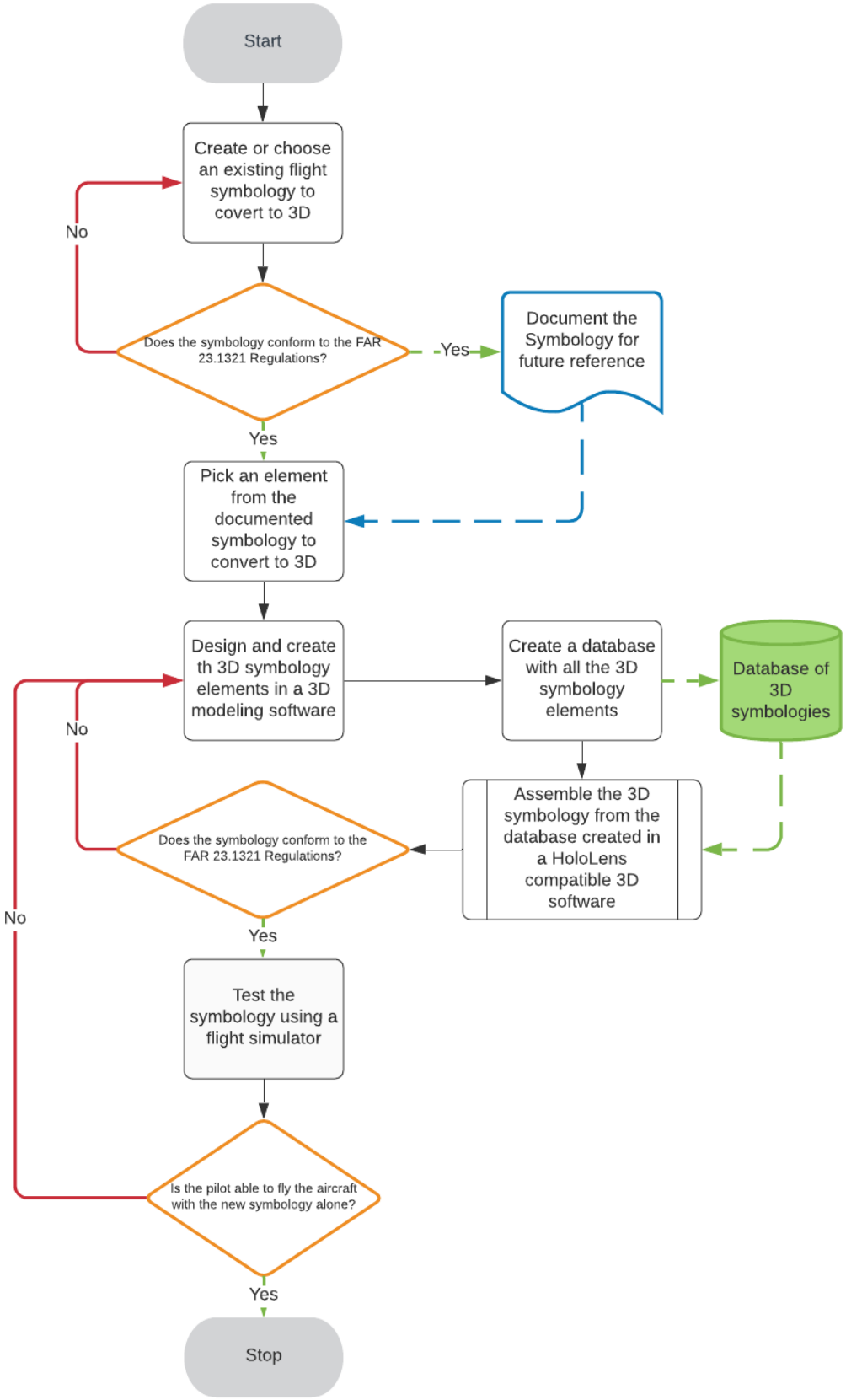


Figure 6. Methodology flowchart of converting 2D flight symbology to a 3D flight symbology

3.1 Design Methodology

Figure 6 shows a high-level overview of the method used to covert 2D flight symbology to a 3D flight symbology. In this section a more detailed explanation will be given on how the specific 3D symbology was created for this study. As mentioned in section 3.0, the 2D symbology being mimicked is from the F-18 Hornet fighter jet. Following the outline seen in Figure 6, the 2D symbology was checked to see how well it conformed with the selected parts of the FAR 25.1321 regulations discussed in section 1.7. The heading ladder was found to be placed on the opposite side than what is required by the FAR 25.1321 regulation. To fix the issue the heading ladder was brought down below the roll indicator as shown in Figure 7.

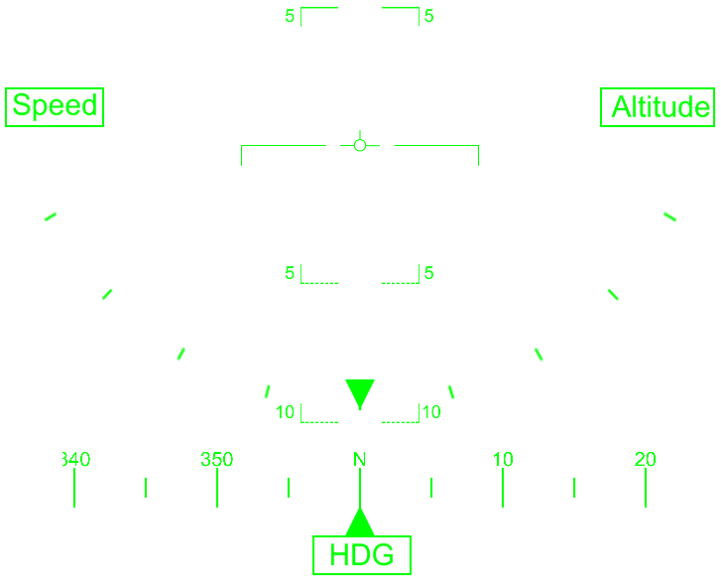


Figure 7. Updated 2D flight symbology to conform with part of the FAR 25.1321 regulation

Following the steps outlined in Figure 6 the ADI was chosen as the 2D symbology that will be converted to a 3D symbology. As a first step in the conversion process a simple representation of a fixed wing aircraft was created. In this case the Pro Builder add-on in Unity was used to create the model shown in Figure 8. The idea is that the motion of this 3D model will be tied to the motion of the actual aircraft. This way the pilot will be able to see a one-to-one representation of the attitude of the aircraft they are piloting.

Figure 8. Simple aircraft model used to represent the aircraft's attitude in flight for a fixed-winged 3D ADI symbology

The next step was to connect the Unity project to the desired flight simulator, in this case the flight simulator was DCS World. The steps to connect the two programs is outlined in section 3.2. This section will focus on the design of the 3D Instrument representation. Once the connection was established and the 3D symbology was assembled, the testing began. The assembled 3D Primary Flight Display (PFD) is shown in Figure 9. Right away it was apparent that having an aircraft with no means of determining when the aircraft was in level flight didn't allow for a safe flight. The lone aircraft model floating in space gave a general idea of what the actual aircraft was doing in space. However, this did not convey enough information for the pilot to make fine-tuned adjustments to their flight.

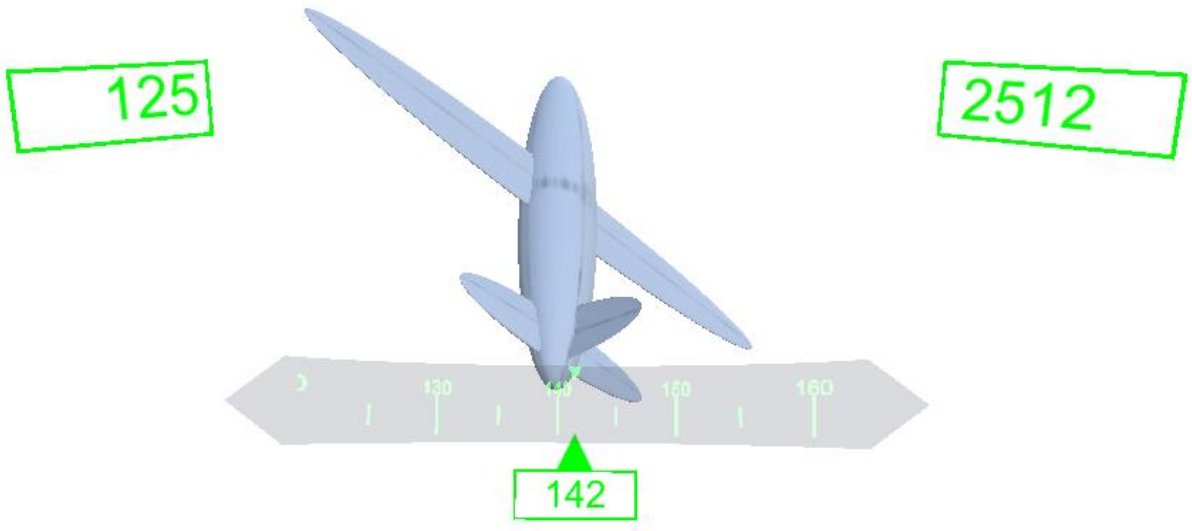


Figure 9. Early representation of the 3D ADI symbology

Once again consulting the steps outlined in Figure 6 it was determined that the pilot was not able to fly safely with the iteration shown in Figure 9, so a change in design was explored. For the next iteration, markers were placed on four sides around the model aircraft. The two markers near the wings and two more near the nose and tail section of the aircraft model indicated where the respective points on the aircraft should line up in level flight as seen in Figure 10. After testing this version of the 3D symbology, it was determined that it was difficult to tell how much offset the aircraft was experiencing from level flight due to parallax. So, again the design was iterated on.

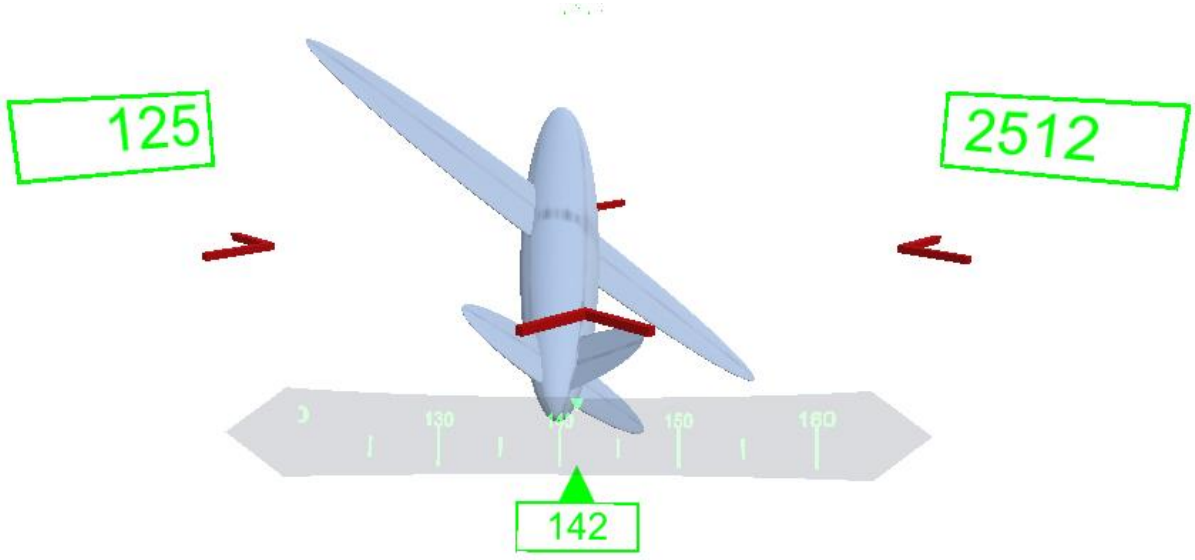


Figure 10. 3D ADI symbology with red indicators showing where the wings, nose and tail of the model aircraft should line up to maintain level flight

The next iteration saw the addition of a translucent plane that bisects the aircraft through the center in level flight (Figure 11 (a)). This addition of the plane provides a much clearer understanding of the attitude of the aircraft in flight. The next addition was a plane that represents the ground relative to the aircraft model. This ground plane moves closer and further to the ADI symbology depending what the altitude of the aircraft is. Another feature of the ground plane is a color change based on the altitude:

- Over 700m the ground plane is green (Figure 11 (b))
- Between 700m and 100m the ground plane is yellow (Figure 11 (c))
- Less than 100m the ground plane is red (Figure 11 (d))

This color change provides a visual cue to the pilot indicating the approximate altitude of the aircraft.

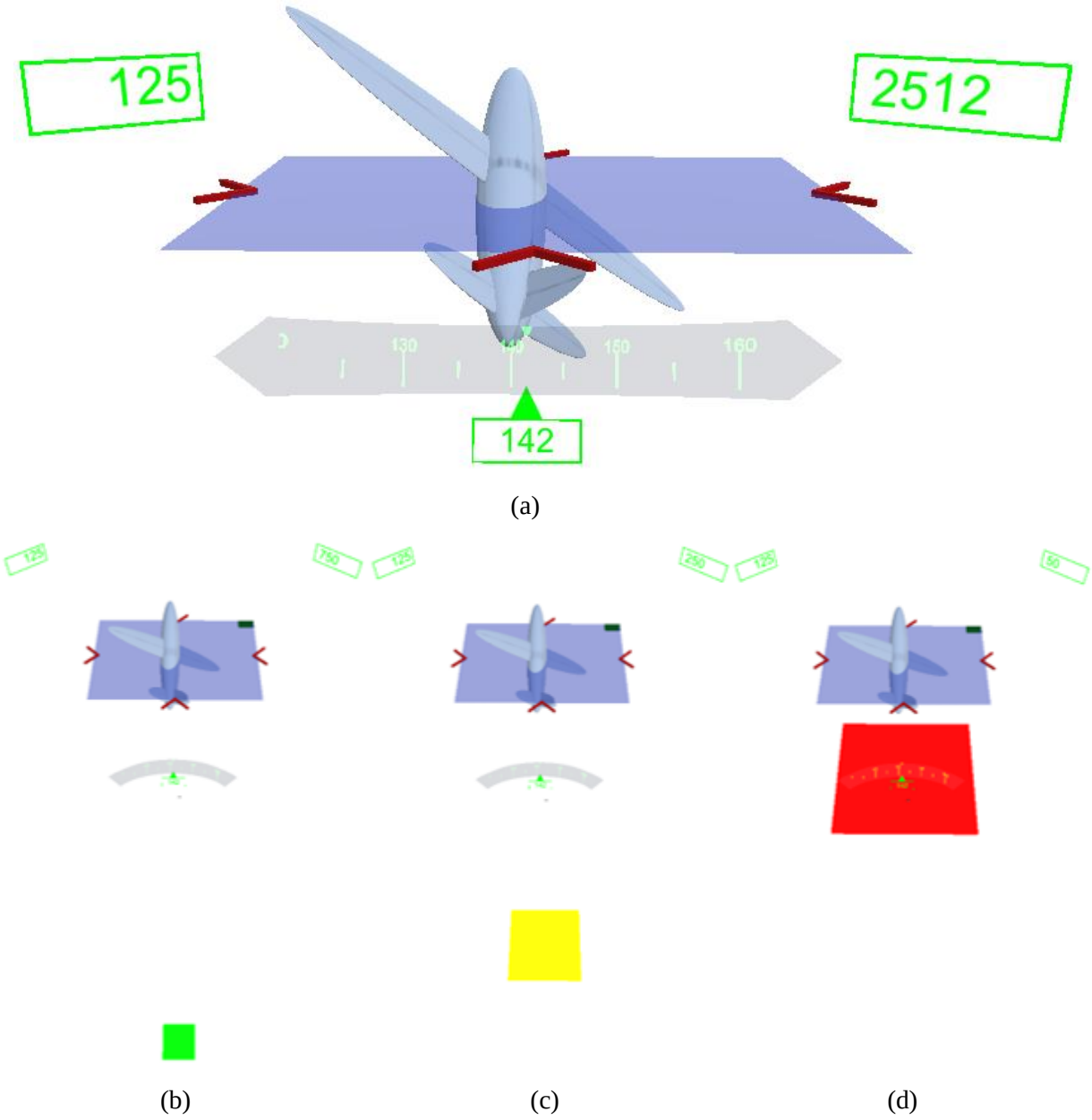


Figure 11. (a) Shows the addition of a plane for pilots to reference to maintain level flight, (b) shows ground plane at 750m altitude, (c) shows ground plane at 250m altitude. (d) shows ground plane at 50m altitude

The final test conducted with the setup shown in Figure 11 not only allowed the pilots to keep their aircrafts in the air, but also allowed the pilots to conduct an emergency landing with no external cues other than the 3D flight symbology. The results of the tests do not mean the iterative process is over, however they do indicate that the steps being taken are headed in the right direction.

Since the results from testing the fixed-winged based flight symbology seemed promising, the next step was to test the same concept with a quadrotor aircraft. In order to give the pilots a better understanding of the aircraft being flown a new aircraft model was created to resemble a quadcopter. This time the model was created in SketchUp by Trimble Inc. SketchUp was only used due to convenience; any 3D modeling software could have been used. The quadcopter model created can be seen in Figure 12.

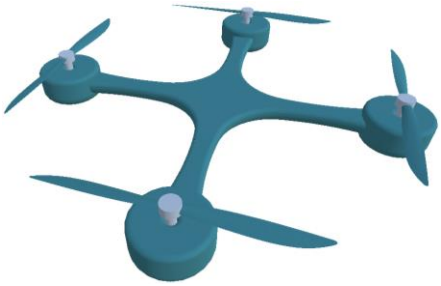


Figure 12. Simple quadcopter model used to represent the aircrafts attitude in flight for a quadcopter 3D ADI symbology

Due to the way the 3D HUD symbology was created in Unity, it only took a few steps to replace the aircraft model with the quadcopter model. Once the model was replaced the motion linked to the imported flight data. Another key step was to change the flight simulator from a fixed-wing simulator to a quadcopter specific simulator. For this project, Microsoft's AirSim simulator based in UE4 was used. After a few changes made in UE4 to allow for the exportation of live flight data to Unity, the testing began. Figure 13 shows the 3D HUD setup for a quadcopter pilot.

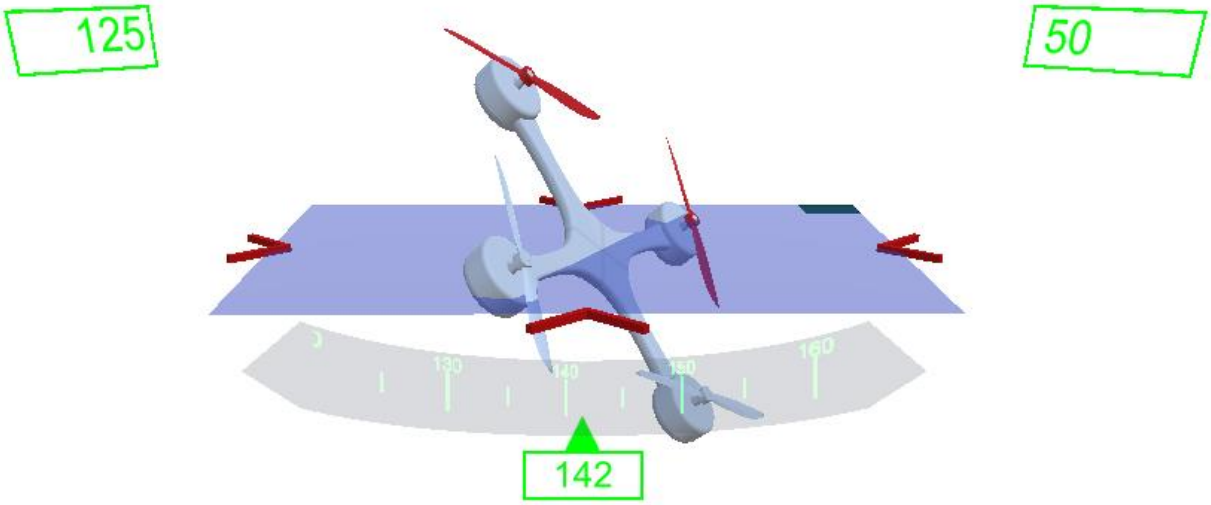


Figure 13. 3D HUD symbology for a quadcopter pilot

So far, only the design process has been discussed. In the next few sections a more technical description will be given on how the flight symbology in Figure 11 and 13 was achieved, and how the testing was conducted.

3.2 Communication between Programs

The following section will explain how the communication between the flight simulators and Unity were accomplished

3.2.1 Exporting flight information from DCS Word

In order to export data from DCS World a script called ‘Export.lua’ was created. This script was written in the Lua programming language and located in DCS World project’s ‘Script’ folder. This is because DCS is built to recognize that file name, in that specific folder to allow the export of data. Within the ‘Export.lua’ file, the aircraft’s airspeed, altitude, heading, roll, pitch, and vertical velocity were queried every frame. Once these flight parameters were saved as variable, they were then saved into individual text files. The text files are updated every frame of the simulation, this way the data is kept up to date. These text files can then be accessed from other programs to use as they please.

Five main functions are used in the Export.lua file to export data from DCS World at runtime, these were:

1. LuaExportStart():
    - This function is called once at the start of the simulation
2. LuaExportBeforeNextFrame():
    - This function is called before every frame
    - This function is ideal for issuing commands to the DCS simulation at runtime from an external source.
3. LuaExportActivityNextFrame(t):
    - This function is called during the current frame

- This is the function that is used to query for all the flight data and where the data is saved to text files.

4. LuaExportNextFrame():
    - This function is called after every frame
    - This function is ideal for closing any processes started in the LuaExportBeforeNextFrame() function
5. LuaExportStop():
    - This function is called once when the simulation has been stopped

3.2.2 Exporting flight information from UE4

In order to export live flight data from the UE4 based AirSim simulation a custom blueprint node was created using the C++ language. The node is set up to accepts 3 data inputs in string form:

1. The first input is called ‘Save Text’:
    - This input expects to receive the flight information in string form
2. The second input is called ‘File Path’:
    - This input expects to receive the file that the export text file is to be saved to
3. The third input is called ‘File Name’:
    - This input expects to receive the file name ending in .txt

Figure 14 shows an image of the node. The output, ‘Return Value’ is a Boolean value that gives a value of ‘True’ or ‘False’ depending on whether saving the file was successful or not. The output was used to ensure the data was saved successfully.

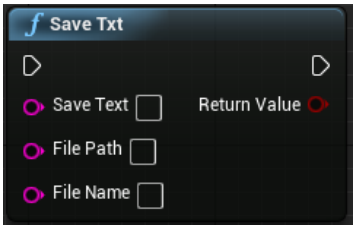


Figure 14. Custom Blueprint node used in UE4 to save data to a text file

3.2.3 Importing flight information into Unity

The final step in completing the communication between the flight simulator and Unity is importing the data that was exported into the Unity project. This is done by creating a component script in Unity's C# backend. Importing the data is done quite simply by opening the text file and saving the data within as a variable, and then closing the file back up. Closing the file ensures that the simulator will be able to open the file again to update the data. For this project the script that imports the data also handles the 3D HUD elements. This is done for simplicity and to ensure the least amount of latency between the data transfer. Figure 16 shows a flowchart of how the data is handled once imported into unity. Figure 15 is the UI created by the C# component script.

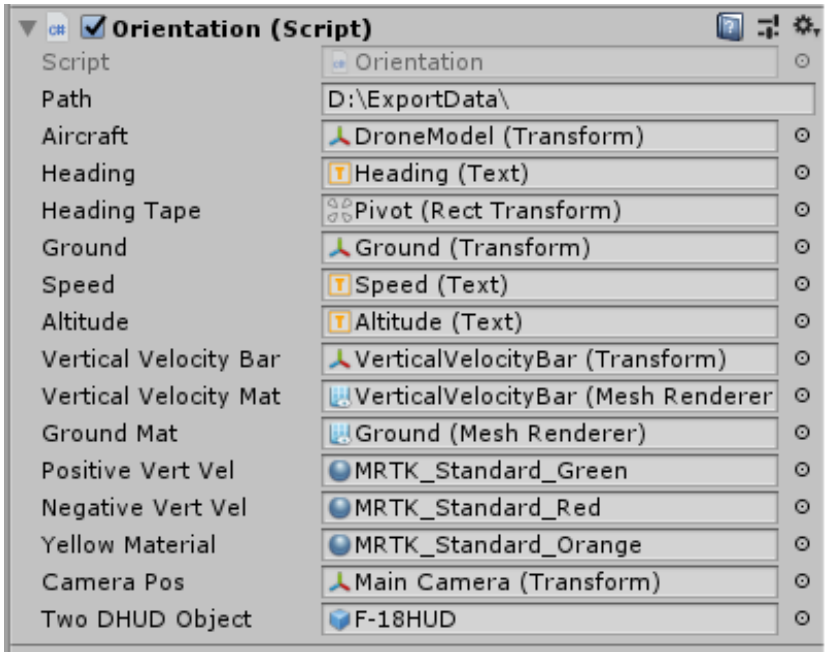


Figure 15. Unity GameObject Orientation component layout

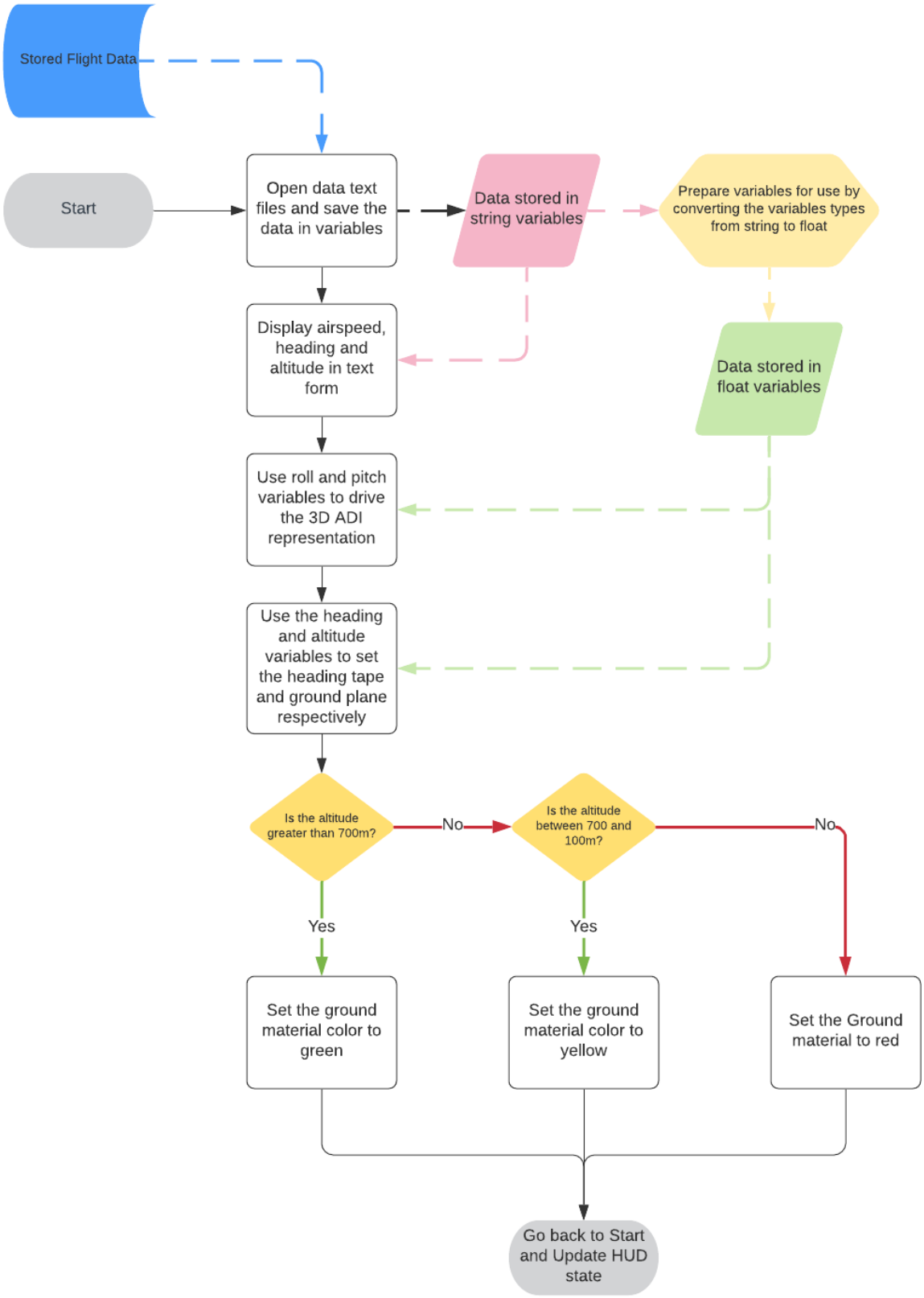


Figure 16. Methodology flowchart showing how the imported flight data is handled inside Unity to drive the 3D HUD Symbology

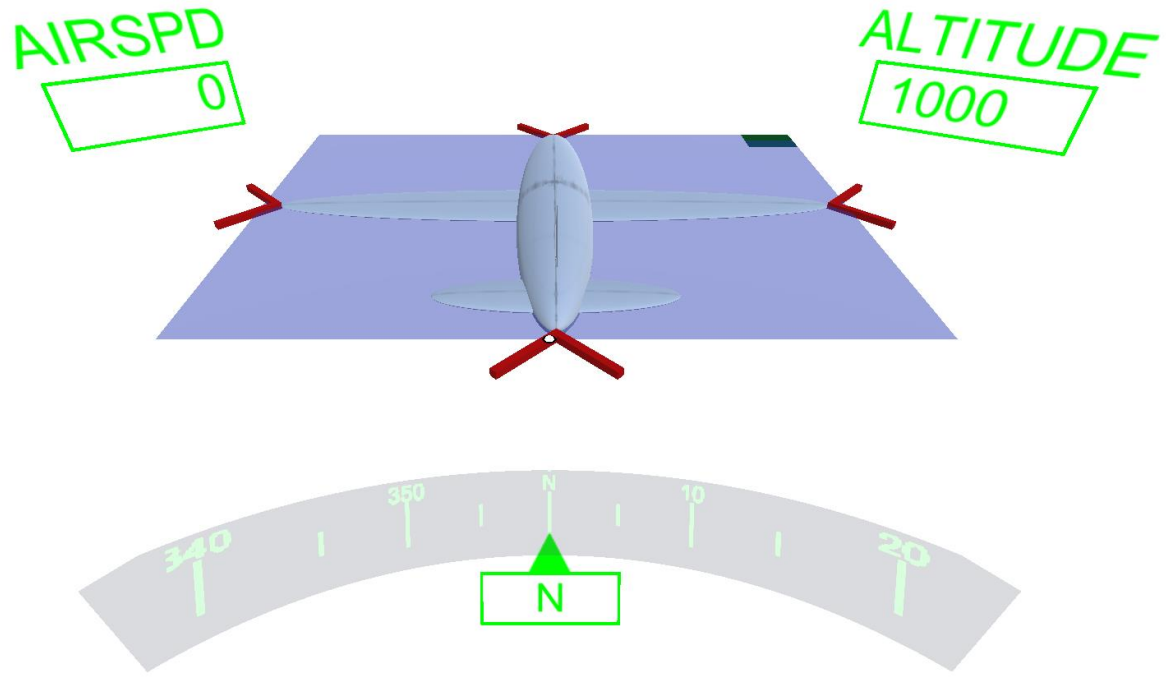
3.3 Additional Capabilities

A few additional capabilities that were added to the Unity project were screen[s] that mirror the monitor[s] attached to the PC, a button to change the view in DCS world, and two buttons to change what monitor is displayed on the center screen. These features were added as quality of life features for the development process. The tests were conducted with these features removed. However, the screen adds a way to provide the First-Person-View (FPV) most drone pilots are accustomed to flying with.

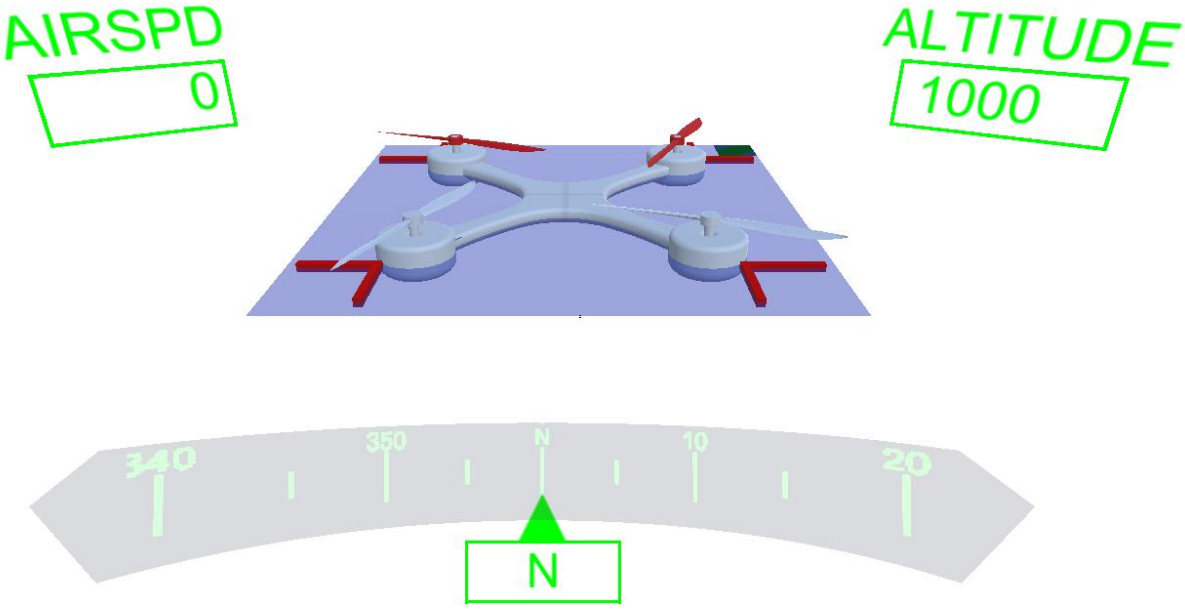
4.0 Results and Discussion

4.1 Results

After going through the testing and iterating discussed throughout section 4.1 the final layouts are shown in Figure 17. Labels were added to the airspeed and altitude indicators to add clarity in what the numbers indicate. For the heading indicator the label now shows ‘N’, ‘NE’, ‘E’, ‘SE’, ‘S’, ‘SW’, ‘W’, and ‘NW’ indicating North, Northeast, East, Southeast, South, Southwest, West, and Northwest, when the aircraft is facing 0°, 45°, 90°, 135°, 180°, 225°, 270°, and 315° heading respectively. It was found that showing these specific directions helped pilots better orient themselves in flight. A notable change was made to the 3D drone HUD where the red roll and pitch indicating chevrons were moved to make contact with the four motor housings of the quadcopter. This gave a better understanding of how to line up the aircraft for level flight.



(a)



(b)

Figure 17. (a) Final 3D HUD design for a fixed-winged based aircraft, (b) Final 3D HUD design for a quadcopter-based aircraft

4.2 Discussion

As mentioned in the mission objective, the main goal of this study was to create a 3D flight symbology that can utilize the HoloLens' holographic capabilities. This goal was achieved by the end of the research term. As shown in Figure 16 the standard 2D Primary Flight Display (PFD) was successfully converted top a 3D flight display. This was tested in a limited capacity by student volunteers at the MIMS lab. The test involved the pilots to fly an aircraft, keep it in the air for a limited time, and then perform an emergency landing by only using the 3D HUD. All seven of the pilots successfully completed this test. The same test was also conducted with the 2D HUD shown in Figure 6. Although some pilots with actual flight experience were able to fly and land the aircraft, the pilots with no flight experience found it difficult to land. This indicated that although for experienced pilots the symbology does not affect their flight in a drastic way. For newer inexperienced pilots, the 3D HUD provides a better understanding of their aircrafts behaviour, which results in a safer flight.

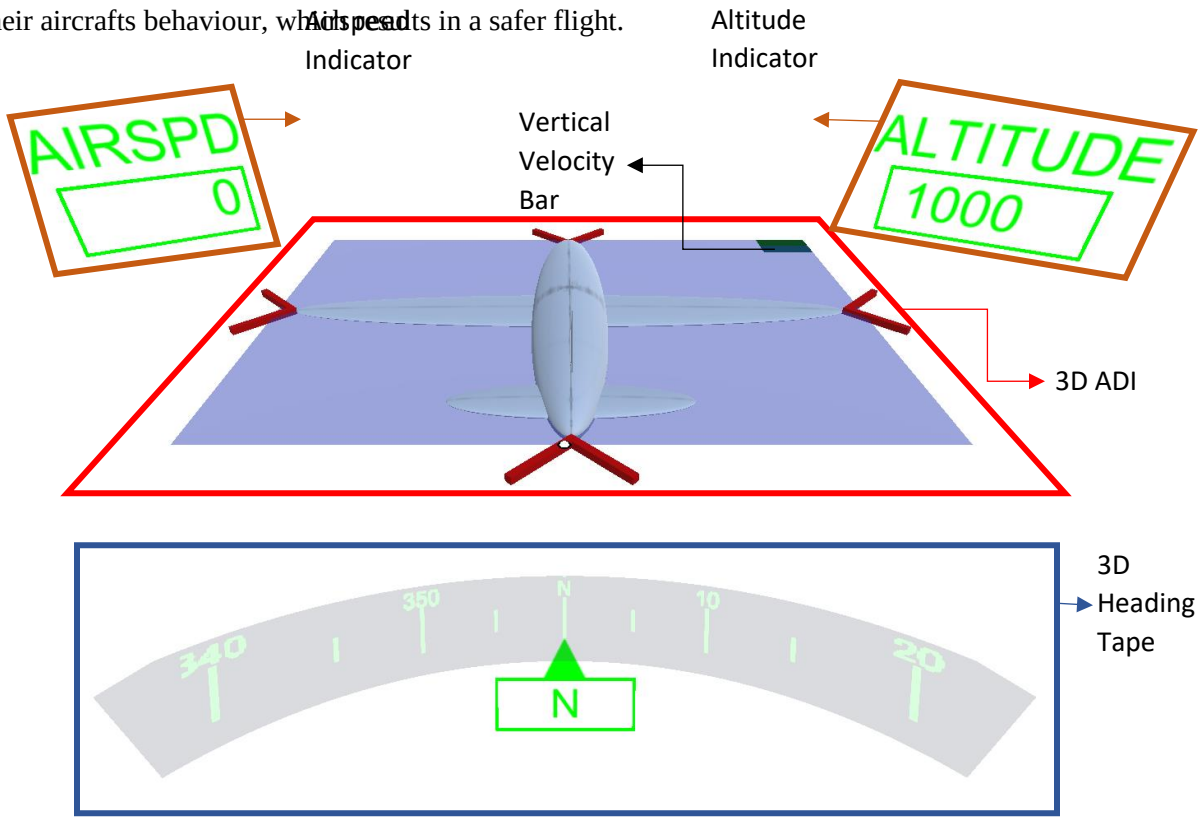


Figure 18. Breakdown of the 3D HUD symbology

Breaking down Figure 17 shows how even though the symbology has been converted from the traditional 2D display to a 3D on, the requirements for regulations such as FAR 25.1321 can still be followed.

1. The display that most effectively indicates the attitude of the aircraft, in this case the 3D ADI, has been placed in the top center position in front of the pilot.
2. The instrument that most effectively indicates the airspeed has been placed adjacent and directly to the left of the instrument in the top center position, in this case the 3D ADI
3. The instrument that most effectively indicates the altitude has been placed adjacent and directly to the right of the instrument in the top center position, in this case the 3D ADI
4. The instrument that most effectively indicates direction of flight, in this case the 3D Heading Tape, has been placed adjacent and directly below the instrument in the top center position, once again being the 3D ADI.

A few additional features were added in the final design that are not listed as priorities in the FAR 25.1321 regulation. These are the addition of the ground plane and a vertical velocity bar. The reasoning for the addition of the ground plane is to give the pilots a relative distance of the ground from the aircraft. Although the altitude indicator provides the necessary information to let the pilot know what altitude they are operating in, the ground plane assists in visualizing the distance. Visualization are on the key benefits of using HMDs such as the HoloLens, so it was decided that the ground plane was a useful enough addition to the flight symbology. It was also proven to be useful from testing. During the emergency landing sequence, pilots found the ground plane easier to use when compared to the actual altitude indicator. This is because the ground plane is connected to the 3D ADI, the pilots could focus on the attitude of their aircraft in keeping it level, while the ground plane moved in and could be seen in the pilot's peripheral vision. It was found difficult for the pilots to understand the altitude in their peripheral vision with only the altitude indicator, which resulted in them having to switch focus away from then 3D ADI to the altitude indicator.

It was a lot easier for the pilots to use the ground plane in their peripheral while maintaining focus on the attitude of their aircraft, allowing them to better control the aircraft.

The addition of the vertical velocity bar (Figure 18) helped pilots better understand how fast or how slow their aircraft was rising or dropping. When the vertical velocity of the aircraft is positive the bar raises above the blue level plane and is green in color. When the vertical velocity is negative, or when the altitude is dropping, the bar dips below the blue level plane, and is red in color. The size of the bar is relative to how fast the aircraft is rising or dropping. This assisted the pilots in better understanding the terrain they aircraft was operating in. For example, if the aircraft is flying level and the vertical velocity bar is not rising too far above or dipping too far below the blue level line, but the ground plane is rising and/or, it can be inferred that the terrain is not even. If the ground plane is rising, the aircraft is approaching a hill or a mountain, which lets the pilot know to increase altitude or maneuver away in order to avoid crashing. During testing pilots used this feature to avoid crashing into a hill or used to assist their landing. This also shows the holographic symbology's potential in assisting pilot in Degraded Visual Environment (DVENS) scenarios.

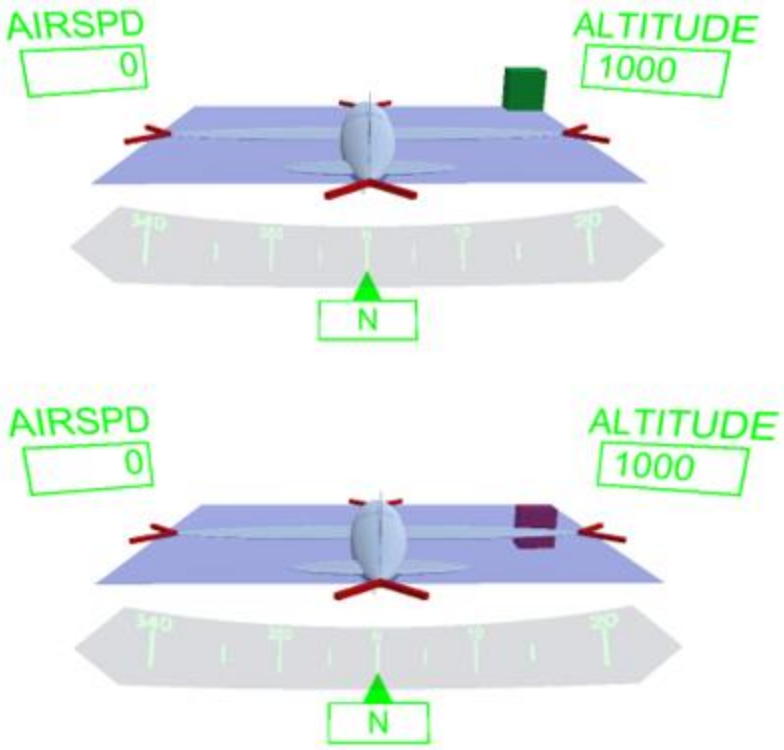


Figure 19. (Top) Vertical velocity at 10m/s, (Bottom), vertical velocity at -10m/s

After further testing of the 3D HUD it was clear that UAV pilots will benefit the most from such an application. Especially pilots operating in Beyond Visual Line Of Sight (BVLOS) scenarios. Being able to see the attitude of their aircraft in the air will allow the pilots to operate their aircraft more safely. Using the system built as a holographic GCS, UAV pilots will be able to setup a fully customizable GCS with minimum resources. Using the features such as the POV screen, UAV pilots can setup a GCS in remote areas. The holographic POV screen, shown in Figure 19, for example eliminates the need for a physical screen

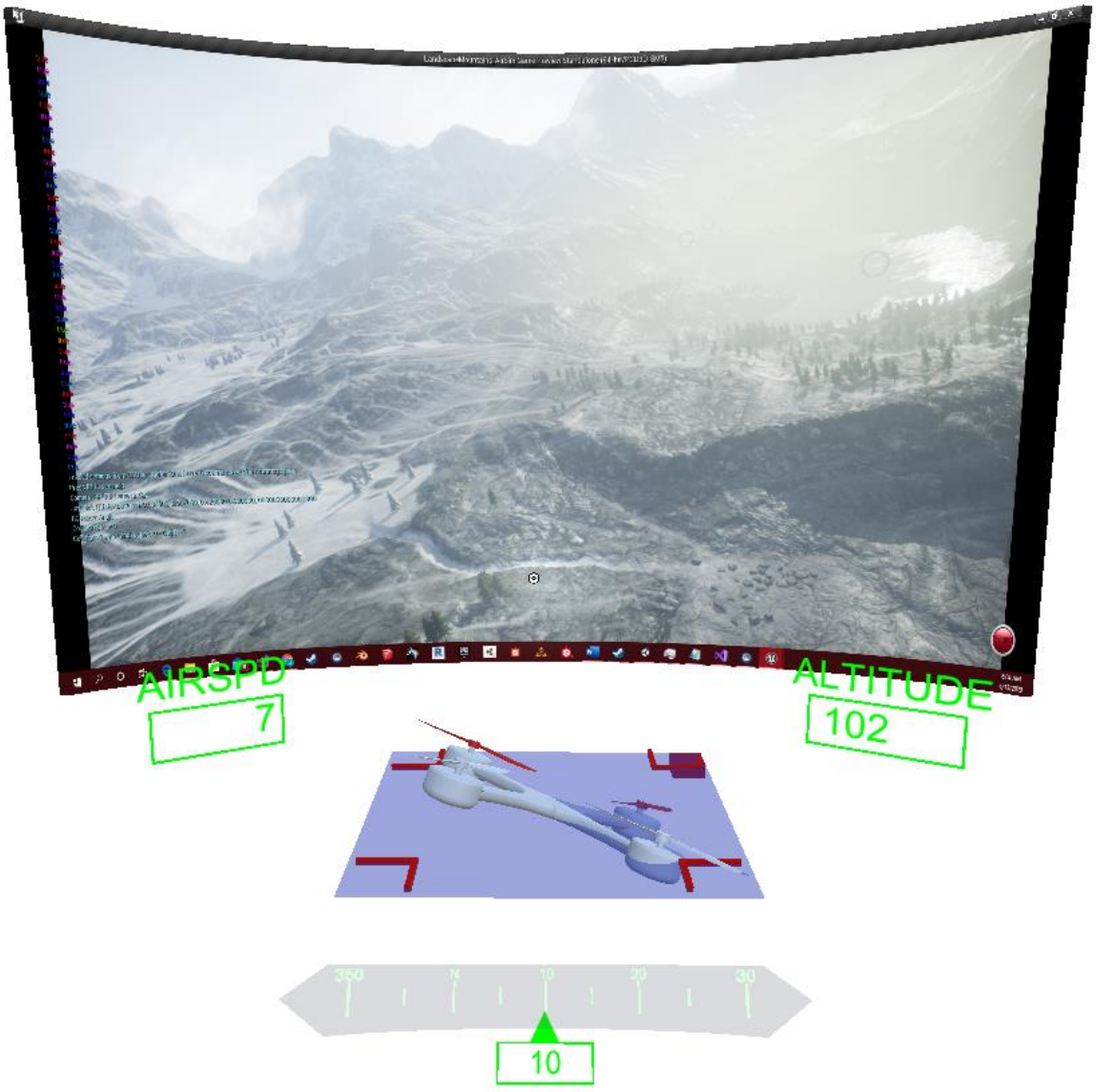


Figure 20. Implementation of a holographic POV screen with the 3D HUD

5.0 Conclusion

In the course of the study discussed in this report, research was conducted to devise a method to display traditional 2D flight symbology in a 3D holographic format. The 3D symbology was designed to be used on AR HMD devices such as the HoloLens. Once the 3D HUD was created it was tested and iterated upon several times to ensure that pilots of any skill level can safely operate an aircraft by only using the 3D HUD. This study started with the assumption that fighter pilots might benefit the most from a such a display, however in the course of testing it was found that UAV pilots can benefit just as much if not more from

such a system. With a device such as the HoloLens, UAV pilots will be able to setup a holographic Ground Control Stations that will provide more capabilities with less resources, such as a POV screen. The study described in this report is one of many ways that devices like the HoloLens can help drive innovation in the aviation industry.

6.0 Future Work

This study was just an introduction of how devices such as the HoloLens can be used to create holographic flight instruments and so there is a lot of room for improvements and additions. For this study only instruments found in a PFD have been converted to a 3D representation, however future studies can be done to incorporate other instruments into this system. Such as, creating a method of representing skid and slip using the same model used for the ADI. More research can also be done on the human factors side of such a system. Due to the limited time and resources (mainly caused by the COVID-19 pandemic), enough flight test and iterations could not be conducted. Due to how new the concept discussed in this report is, it requires a lot more research and experimentation.

Reference

[1] Pradhan, Pratik. (2017). Development of Holographic User Interface for UAV Ground Control using Microsoft HoloLens. 10.13140/RG.2.2.25901.33760.

[2] Spitzer, C. R., Ferrell, U., & Ferrell, T. (2017). *Digital avionics handbook*.

[3] Techopedia.com. (2016, May 9). What is Human-Machine Interface (HMI)? - Definition from Techopedia. Retrieved March 8, 2020, from https://www.techopedia.com/definition/12829/human-machine-interface-hmi

[4] Center for Devices and Radiological Health. (2017, December 23). Human Factors Considerations. Retrieved March 9, 2020, from https://www.fda.gov/medical-devices/human-factors-and-medical-devices/human-factors-considerations

[5] Furht, B. (2011). *Handbook of Augmented Reality*. New York, NY: Springer Science Business Media, LLC.

[6] Federal Aviation Administration. (2014). *Far / Federal Aviation Regulations/Aeronautical Information Manual*. New York, NY: Skyhorse Publishing.

[7] wevolver.com. (2019). Knowledge shared by technology developers and scientists. Retrieved March 15, 2020, from https://www.wevolver.com/wevolver.staff/hololens.gen.1/

[8] Haas, J. K. (2014). A History of the Unity Game Engine. Retrieved from https://digitalcommons.wpi.edu/iqp-all/3207

[9] G0DCH. (2019, December 10). microsoft/AirSim/Remote Control. Retrieved March 23, 2020, from https://github.com/microsoft/AirSim/blob/master/docs/remote_control.md

Appendix A: Orientation.cs Script

```
using System;
using System.IO;
using UnityEngine;
using UnityEngine.UI;

public class Orientation : MonoBehaviour
{
    /*
     * Initialize the Variable
     * Adding the 'public' keyword indicates that these variables will be exposed in
Inspector
     * Adding the 'private' keyword indicates that these variables can only by used
within the script
    */

    public string path;
    public Transform Aircraft;
    public Text Heading;
    public RectTransform HeadingTape;
    public Transform Ground;
    public Text Speed;
    public Text Altitude;
    public Transform VerticalVelocityBar;
    public MeshRenderer VerticalVelocityMat;
    public MeshRenderer GroundMat;
    public Material PositiveVertVel;
    public Material NegativeVertVel;
    public Material YellowMaterial;
    public Transform CameraPos;
    public GameObject TwoDHUDObject;

    private StreamReader _roll;
    private StreamReader _gndSpeed;
    private StreamReader _altitude;
    private StreamReader _pitch;
    private StreamReader _heading;
    private StreamReader _verticalVelocity;

    private string[] _dataList;

    private string _check;
    private string _prevData = "0" + ',' + "0" + ',' + "0";
    private double tNext = 0.1f;
    private Vector3 _vertVelPos;
    private Color GroundColor;

    private Vector3 _prevOrientation;
    private string _prevsAltitude;
    private string _prevSpeed;
    private string _prevsHeading;

    string ReadString()
    {
```

```
        //Read the text from directly from the .txt files
        try
        {
            _check = "Roll";
            FileStream fRoll = File.Open(path + "roll.txt", FileMode.Open,
FileAccess.Read, FileShare.ReadWrite);
            _roll = new StreamReader(fRoll);
            string roll = _roll.ReadToEnd();
            _roll.Close();
            fRoll.Close();
            print("Roll Passed");

            _check = "Speed";
            FileStream fSpeed = File.Open(path + "gndspeed.txt", FileMode.Open,
FileAccess.Read, FileShare.ReadWrite);
            _gndSpeed = new StreamReader(fSpeed);
            string gndSpeed = _gndSpeed.ReadToEnd();
            _gndSpeed.Close();
            fSpeed.Close();
            print("Speed Passed");

            _check = "Altitude";
            FileStream fAlt = File.Open(path + "altitude.txt", FileMode.Open,
FileAccess.Read, FileShare.ReadWrite);
            _altitude = new StreamReader(fAlt);
            string altitide = _altitude.ReadToEnd();
            _altitude.Close();
            fAlt.Close();
            print("Altitude Passed");

            _check = "Pitch";
            FileStream fPitch = File.Open(path + "pitch.txt", FileMode.Open,
FileAccess.Read, FileShare.ReadWrite);
            _pitch = new StreamReader(fPitch);
            string pitch = _pitch.ReadToEnd();
            _pitch.Close();
            fPitch.Close();
            print("Pitch Passed");

            _check = "Heading";
            FileStream fHeading = File.Open(path + "heading.txt", FileMode.Open,
FileAccess.Read, FileShare.ReadWrite);
            _heading = new StreamReader(fHeading);
            string heading = _heading.ReadToEnd();
            _heading.Close();
            fHeading.Close();
            print("Heading Passed");

            _check = "Vertical Velocity";
            FileStream fvertVel = File.Open(path + "vertVel.txt", FileMode.Open,
FileAccess.Read, FileShare.ReadWrite);
            _verticalVelocity = new StreamReader(fvertVel);
            string vertVel = _verticalVelocity.ReadToEnd();
            _verticalVelocity.Close();
            fvertVel.Close();
            print("Vertical Velocity Passed");
```

```
            // Save all the data in a single string so it can be sent as a single
variable
            string data = roll + ',' + gndSpeed + ',' + altitide + ',' + pitch + ',' +
heading + ',' + vertVel;

            return data;
        }
        catch
        {
            print("Missed:" + _check);
            return _prevData;
        }

    }

    void TwoDHUD()
    {
        float CameraX = CameraPos.localEulerAngles.x;
        float CameraY = CameraPos.localEulerAngles.y;

        if (20 <= CameraX && CameraX <= 358 || 20 <= CameraY && CameraY <= 340)
        {
            TwoDHUDObject.SetActive(true);
        }
        else
        {
            TwoDHUDObject.SetActive(false);
        }
    }

    // Update is called once per frame
    void Update()
    {
        try
        {
            if (Time.time >= tNext)
            {

                TwoDHUD();

                string loadData = ReadString();
                _prevData = loadData;

                _dataList = loadData.Split(',');

                float roll = Convert.ToSingle(_dataList[0]);

                // Handles the airspeed indicator value
                string speed = _dataList[1];
                double intSpeed = Math.Round(Convert.ToDouble(speed), 0);
                speed = Convert.ToString(intSpeed);
                Speed.text = speed;

                // Handles the altitude indicator value and prepares the data to be used
to change ground plane distance
                string altitude = _dataList[2];
                float fAltitude = Convert.ToSingle(_dataList[2]);
                float aMultiplier = fAltitude / 100;
```

```
                double intAltitude = Math.Round(Convert.ToDouble(altitude), 0);
                altitude = Convert.ToString(intAltitude);
                Vector3 GroundPos = new Vector3(0.1152181f, -0.1649159f - aMultiplier, -
0.7435627f);
                Altitude.text = altitude;

                float pitch = Convert.ToSingle(_dataList[3]);

                // Handles the text in the heading indicator and prepares the data to be
used to move the heading tape
                float heading = Convert.ToSingle(_dataList[4]);
                string sHeading = Convert.ToString(Math.Round(heading, 0));
                sHeading = (sHeading == "0") ? "N" : sHeading;
                sHeading = (sHeading == "45") ? "NE" : sHeading;
                sHeading = (sHeading == "90") ? "E" : sHeading;
                sHeading = (sHeading == "135") ? "SE" : sHeading;
                sHeading = (sHeading == "180") ? "S" : sHeading;
                sHeading = (sHeading == "225") ? "SW" : sHeading;
                sHeading = (sHeading == "270") ? "W" : sHeading;
                sHeading = (sHeading == "315") ? "NW" : sHeading;
                float hMultiplier = heading / 10;
                Vector3 HeadingPos = new Vector3(855.5f - hMultiplier * 71.9f, -11, 0);

                // Handles the vertical velocity bar
                float vertVel = Convert.ToSingle(_dataList[5]);
                float vMultiplier = Math.Abs(vertVel);
                Vector3 vertVelScale = new Vector3(1, vMultiplier, 1);

                if (vertVel < 0)
                {
                    _vertVelPos = new Vector3(4.502f, -1 * vMultiplier/2, 4.48f);
                    VerticalVelocityMat.material = NegativeVertVel;
                }
                else
                {
                    _vertVelPos = new Vector3(4.502f, vMultiplier / 2, 4.491f);
                    VerticalVelocityMat.material = PositiveVertVel;
                }

                // Handles the movement and color of the Ground plane
                float altColorMultiplier = 0;

                if (fAltitude > 700)
                {
                    GroundColor = new Color(0, 255, 0);
                }

                if (300 <= fAltitude && fAltitude <= 700)
                {
                    altColorMultiplier = Convert.ToSingle(Math.Round(0.6375f * (700 -
fAltitude), 0));
                    GroundColor = new Color(altColorMultiplier, 255, 0);
                }

                if (100 <= fAltitude && fAltitude < 300)
                {
                    altColorMultiplier = Convert.ToSingle(Math.Round(1.275f * (fAltitude
- 100), 0));
```

```
                    GroundColor = new Color(255, altColorMultiplier, 0);
                }

                if (fAltitude < 100)
                {
                    GroundColor = new Color(255, 0, 0);
                }

                GroundMat.material = YellowMaterial;
                YellowMaterial.color = GroundColor;

                // Implements the motion of all the PFD elements
                HeadingTape.localPosition = HeadingPos;
                Ground.localPosition = GroundPos;
                Heading.text = sHeading;
                Vector3 Orientation = new Vector3(-1 * pitch, 0, -1 * roll);
                VerticalVelocityBar.localScale = vertVelScale;
                VerticalVelocityBar.localPosition = _vertVelPos;

                Aircraft.localEulerAngles = Orientation;


                tNext = Math.Round(Time.time, 2) + 0.1;
            }
        }
        catch
        {
            print("Failed To Update");
        }
    }
}
```

Appendix B.1: RWTxtFile.h Script

```
#pragma once

#include "CoreMinimal.h"
#include "Kismet/BlueprintFunctionLibrary.h"
#include "RWTxtFile.generated.h"

UCLASS()
class LANDSCAPEMOUNTAINS_AIRSIM_API URWTxtFile : public UBlueprintFunctionLibrary
{
	GENERATED_BODY() public:

		UFUNCTION(BlueprintPure, Category = "FileIO", meta = (Keywords =
"LoadTxt"))
			static bool LoadTxt(FString FileNameA, FString& SaveTextA);

		UFUNCTION(BlueprintCallable, Category = "FileIO", meta = (Keywords =
"SaveTxt"))
			static bool SaveTxt(FString SaveText, FString FilePath, FString
FileName);

};
```

Appendix B.2: RWTxtFile.cpp Script

```
#include "RWTxtFile.h"
#include "Misc/FileHelper.h"
#include "Misc/Paths.h"

bool URWTxtFile::LoadTxt(FString FileNameA, FString& SaveTextA)
{
    return FFileHelper::LoadFileToString(SaveTextA, *(FPaths::ProjectDir() +
FileNameA));
}

bool URWTxtFile::SaveTxt(FString SaveText, FString FilePath, FString FileName)
{
    return FFileHelper::SaveStringToFile(SaveText, *(FilePath + FileName));
}
```

Appendix D: Export.lua Script

```
local log_file = nil
local command_file = nil
local check = 0
local view_command = nil

MainPanel = GetDevice(0)

function LuaExportStart()
   log_file = io.open("C:/Users/jafer/Saved Games/DCS/Logs/Export.log", "w")
end

function LuaExportBeforeNextFrame()
       command_file = io.open("D:/ExportData/ViewCommand.txt", "r")

       if command_file then
              view_command = command_file:read()
              check = 1
       else
              check = 0
       end

       if view_command == "1" then
              LoSetCommand(8)
       end

       if view_command == "0" then
              LoSetCommand(7)
       end

end

function LuaExportAfterNextFrame()
       command_file:close()
       if check == 1 then
              os.remove("D:/ExportData/ViewCommand.txt")
       end
end

function LuaExportStop()

   if log_file then
         log_file:write("Closing log file.")
         log_file:close()
         log_file = nil

         altitude_file:close()
         altitude_file = nil
```

```
        gndspeed_file:close()
        gndspeed_file = nil

        heading_file:close()
        heading_file = nil

        pitch_file:close()
        pitch_file = nil

        roll_file:close()
        roll_file = nil

        vertVel_file:close()
        vertVel_file = nil
   end

end

function WriteToFile(alt, gnd, hdng, pch, rl, yw, vV, la, lo)

      if unexpected_condition then error() end

      altitude_file = io.open("D:/ExportData/altitude.txt", "w")
      gndspeed_file = io.open("D:/ExportData/gndspeed.txt", "w")
      heading_file = io.open("D:/ExportData/heading.txt", "w")
      pitch_file = io.open("D:/ExportData/pitch.txt", "w")
      roll_file = io.open("D:/ExportData/roll.txt", "w")
      vertVel_file = io.open("D:/ExportData/vertVel.txt", "w")

      altitude_file:write(alt)
      gndspeed_file:write(gnd)
      heading_file:write(57.2958 * yw)
      pitch_file:write(57.2958 * pch)
      roll_file:write(57.2958 * rl)
      vertVel_file:write(vV)

      altitude_file:flush()
      gndspeed_file:flush()
      heading_file:flush()
      pitch_file:flush()
      roll_file:flush()
      vertVel_file:flush()

      altitude_file:close()
      gndspeed_file:close()
      heading_file:close()
      pitch_file:close()
      roll_file:close()
      vertVel_file:close()
```

```
pitch = %.2f roll = %.2f\n", t, altitude_data, gndspeed_data, 57.2958 * Heading_data,
57.2958 * pitch_data, 57.2958 * roll_data))
end

function LuaExportActivityNextEvent(t)
	local tNext = t

	local altitude_data = LoGetAltitudeAboveGroundLevel()
	local gndspeed_data = LoGetTrueAirSpeed()
	local pitch_data, roll_data, yaw_data = LoGetADIPitchBankYaw()
	local vertVelocity_data = LoGetVerticalVelocity()

	if pcall(WriteToFile, altitude_data, gndspeed_data, Heading_data, pitch_data,
roll_data, yaw_data, vertVelocity_data) then

		WriteToFile(altitude_data, gndspeed_data, Heading_data, pitch_data,
roll_data, yaw_data, vertVelocity_data)

	else

		tNext = tNext + 0.1 -- data collected once every second / change this
according to preference
		return tNext

	end

	tNext = tNext + 0.1 -- data collected once every second / change this
according to preference

	return tNext
end
```