Virtual Reality Flight Simulator with hand-tracking technology and haptic feedback.

Panagiotis Drakopoulos

Πολυτεχνείο Κρήτης

0.1 Abstract

This thesis aims to take advantage of the latest Virtual Reality technology and technical know-how available, to create a highly realistic and interactive flight simulation experience ,by putting the user in a replica cockpit of a modern passenger jet.

First and foremost, we need to highlight the fact that VR applications require a different approach and design than the standard applications and games for 2D computer monitors, due to the nature of VR itself. Thus, the challenge of this work, was to combine different types of software and hardware technologies, in order to create a realistic 3Dflight simulation, in which the user-pilot can interact in with his own hands, without requiring previous experience with the technology being used or pilot skills. Providing a user-friendly , realistic and enjoyable experience is crucial for the success of this and other similar projects. This is because VR is still making its first steps in the consumer market, meaning that the vast majority of people have not yet experienced VR, but are used to interacting with traditional types of hardware, such as a 2D computer screen, keyboard, mouse and so on.

0.2 Acknowledgements

- I would like to thank my supervisor, Assoc. Prof. Katerina Mania for providing me the opportunity to work on this thesis, as well as the tools , environment and guidance needed throughout the process.
- I would like to thank professors Georgios Halkiadakis and Antonios Deligiannakis for their time reading and reviewing this thesis.
- I would like to thank my friends for their moral support.
- I would like to thank my colleagues Vasilis Dimakopoulos and Vasilis Giovanoglou for the technical support over the haptic feedback implementation
- Finally, I would like to express my sincere gratitude to my family for their endless encouragement and support in many aspects.

Contents

Chapter 1

Introduction

1.1 Concept

In the recent years, Virtual Reality(VR)is becoming more and more advanced, and is slowly but steadily gaining entry in the consumer market. This has made VR technology available to more people than ever before, in high, but affordable prices, which are expected to drop further as years pass. This effort was also boosted by famous companies such as Google, HTC, Oculus by investing in the design and production of Head Mounted Displays (HMDs). As we will explain later, an HMD is the main device that gives users access to virtual worlds.

As a result, VR is getting increasing attention by scientists , software and game developers which are putting big effort to integrate VR in current applications, create new ones and in general make it more attractive and usable to a larger portion of people.

As it was mentioned before, due to the nature of the VR technology, VR applications require developers to alter their approach while developing common non-VR apps. We will provide 3 short examples to prove why this is:

- Traditional input methods, such as the keyboard or mouse , are next to useless, because they are hard to manipulate when a user is wearing a Head Mounted Display. As a result, developers have to come up with more suitable input and output methods. These days, the most common input methods for VR apps are hand-tracking hardware, or looking at a specific direction (while wearing an HMD).

- As for the output, it is recommended that developers dont require users to make uncomfortable head movements in order to achieve their goal. Furthermore, any shown text or information should not be very small, otherwise it is running the risk of being indistinguishable, because of pixelation visible on the HMD screen (also known as screendoor effect more about this on the following chapter).
- VR developers must also take into account the fact that sudden or extreme motion in the virtual world is most likely to cause unpleasant effects on users, ranging from dizziness to nausea, headache, disorientation and even vomiting.Studies show that the cause for these unfortunate health issues related to VR usage, is that the human eye suggests that the body is moving, while the inner ear disagrees, thus flooding the brain with conflicting information. [1]

In this thesis, we attempted to overcome all of the above challenges, by creating an 3d VR Flight Simulator application with priority to user experience and comfortable interaction techniques. This was achieved by taking advantage of real-time **hand-tracking** technology and **haptic feedback**.

The **concept** of this application, was to develop a VR flight Simulator, complying with all of the above , in which the user-pilot can perform basic tasks and operations in a 3d replica of a real cockpit, by simply using his own hands and a joystick. The user should also be able to start-up the airplane, taxi to the runway, takeoff, fly (manually or by Auto Pilot) and land. To make this possible, a 3D model of the entire Eleftherios Venizelos Athens International Airport was constructed from scratch, including details such as runways, taxiways, terminal buildings, ground service vehicles, parked aircraft and more.

Additionally, a large part of photoreal terrain surrounds the airport, covering a large region of Attica, whichin combination with the sky and clouds make the Flight Simulator look even more realistic. Finally, a Flight Simulator wouldnt be a Simulator without the appropriate physics. A big effort was put to make the airplane feel realistic, by simulating all the forces acting on it in real-time: lift, drag, thrust and gravity. These of course will be analyzed and explained in the next chapters.

1.2 Thesis Outline

In the following chapters the framework that was followed is thoroughly explained. The first three chapters have an introductory and educative purpose. The rest of the chapters focus on the implementation process along with the software and hardware used to accomplish the final result.

The second chapter focuses on the theoretical background and basic terminology regarding the field of Virtual Reality. Additionally, the technology behind virtual reality headsets is extensively presented, explained and reviewed.

The third chapter provides an insightful review of technologies used in 3D Graphics, from the creation to rendering of 3D surfaces and objects. Moreover, a review and comparison of the most powerful free game engines is provided. The chapter also explains which technologies of the above are used in this project and why.

The fourth chapter describes the software tools used in the development process. Unity3D's architecture and basic user interface elements are presented. Finally the chapter presents the 3D modelling software used for creating the 3D environment.

The fifth and most extensive chapter is dedicated to presenting fundamental parts of this application. The first section describes the creation process of the 3D environment along with the used techniques and software. The following section provides the aerodynamic and physics theories regarding aircraft flight, including explanation and examples of how these are simulated in the game engine, Unity3D. The next section depicts the aircraft avionics, the electrical systems of the cockpit, and how they were implemented. The remaining sections outline the application's Audio, Menu in conjuction with its basic structure.

The sixth chapter portrays the process of making the user able to interact with the cockpit with his hands. It begins with a small review of the used hand-tracking hardware. Then the integration process to Unity3D is presented step by step, from installation to scripting so that hand gestures are recognized.

The seventh chapter is devoted to haptic feedback. It initially explains what haptics are and what purpose do they serve. The next section present the entire implementation process, from connecting, configuring and programming the hardware, to enabling communication with software.

The eighth chapter presents the evaluation result of this project. The final version of the application was assessed by a number of users, in terms of usability and overall quality, noting their comments, reactions and suggestions.

In the ninth chapter the conclusion of the whole development process is presented, along with thoughts and suggestions about future extensions.

Chapter 2

Background and Terminology

2.1 Virtual Reality

Virtual reality is a computer technology that simulates a user's physical presence in a software-generated replica of a real environment or an imaginary setting, enabling the user to interact with this space. A person using virtual reality equipment (ex. an HMD) is typically able to "look around" the artificial world, move about in it and interact with features or items that are depicted. Virtual realities artificially create sensory experiences, which can include sight, touch, hearing, and, less commonly, smell. Todays most popular VR devices are the so called HMDs, which we explain next. [2]

2.2 Frame Rate

Frame Rate, commonly measured and referred to as frames-per-second (FPS), is the frequency at which a hardware device is able todraw or capture consecutive images , called frames. The term is usually used when describing technical specifications of film and video cameras, computer graphics and motion capture systems.It is also a measure of performance of games and 3d applications. In video, film computer graphics and even more in Virtual Reality, Frame Rate output is critical, as it decides whether consecutive frames will be perceived by the brain as separate images or not, thus giving the desired illusion of motion. Although the human visual system can theoretically process 1000 different images a second, studies show that untrained eyes can hardly tell any difference above 60fps or 100fps, depending on the display device and usage. For example virtual reality devices usually require a higher than usual fps rate to create the pleasant illusion of smooth motion.Finally, it is essential to note that achieving high frame rates is usually not an easy task , as it requires hardware with substantial processing power and thoughtful application programming.

2.3 Head Mounted Displays (HMDs)

Figure 2.1: HMDs by Oculus,Sony and htc.

An HMD is generally a form of head-mounted goggles that has a display in front of one (monocular HMD) or both eyes (binocular HMD). These displays allow the user to see the virtual environment. Many current generation HMDs come with features such as positional-tracking, head-tracking, hand-tracking and even eye-tracking. The more sophisticated the HMD, the more of the above technologies are incorporated, providing users remarkable simulation experiences.

2.3.1 Positional tracking

This technology allows the HMD and/or the computer to have knowledge of real-time positioning of a user in real 3D space. This can be very useful for certain applications or games, as someone could move in the virtual world by simply walking or moving around in reality. This is achieved by constantly measuring distance and angles between fitted sensors inside the HMD in respect to one or more base stations located in a room. Then , mathematical calculations translate the measurements to the exact position of the user in the room.

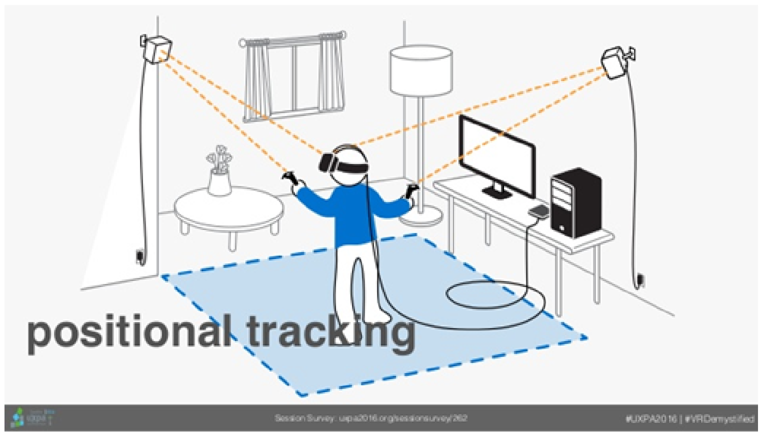

positional tracking
Session Survey: uxpa2016.org/sessionsurvey/262
#UXPA2016 | #VRDemystified

Figure 2.2: positional tracking

2.3.2 Head tracking

It is one of the most important features of an HMD. This is what allows the user to look-around a virtual world by simply moving/tilting the head as we normally do in our lives, without the need of separate controllers, thus giving the illusion of presence in the virtual environment. A head-tracking system typically consists of components such as accelerometers, gyroscopes and magnetometers,which are also used in nowadays smartphones.An external stationary sensor may also be used.

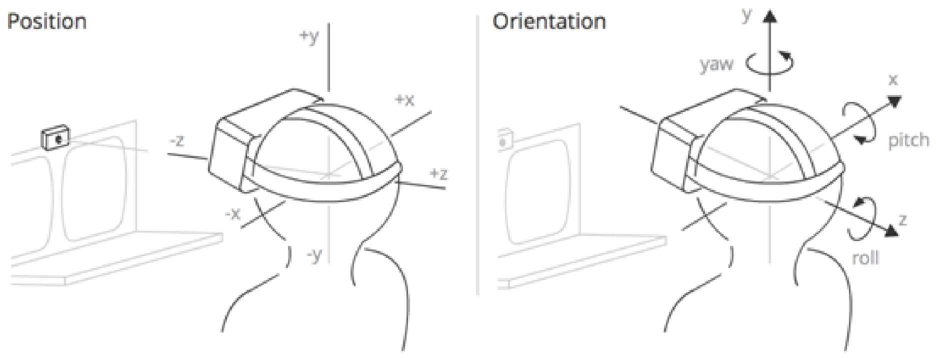


Figure 2.3: positional tracking. *Source : https://developer.mozilla.org/en-US/docs/Web/API/WebVR_API/WebVR_concepts*

2.3.3 Eye tracking

It is the process of measuring either the point of gaze (where one is looking) or eye motion. Eye trackers have a wide range of applications, which includes medical research, marketing, product design, simulators, human activity recognition, and recently VR. For example, in medical research eye-tracking can be used to study a subjects response to different kinds of stimuli, helping in the diagnosis of diseases and disorders, such as Attention Deficit Hyperactivity Disorder (ADHD), Autism Spectrum Disorder (ASD), Obsessive Compulsive Disorder (OCD). In marketing and product design, eye-tracking can be used to detect whether potential customers focus their attention on the desired elements of a product. In the Virtual Reality field, eye-tracking can be helpful in interacting with the environment by providing the point of gaze as input to the computer or bringing specific objects and scenery in/out of focus, depending on where someone is looking. [5]

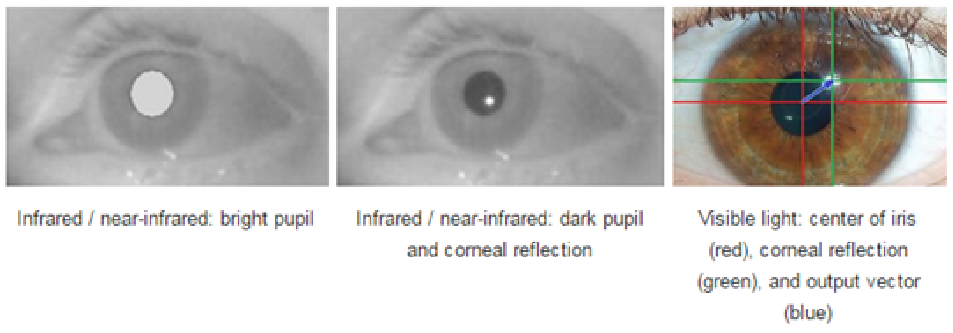

Infrared / near-infrared: bright pupil
Infrared / near-infrared: dark pupil and corneal reflection
Visible light: center of iris (red), corneal reflection (green), and output vector (blue)

Figure 2.4: Eye tracking. Source: `en.wikipedia.org/wiki/Eye_tracking`

The most widely used designs are based on video-based eye trackers. A camera focuses on the eye recording several frames per second (fps) which can vary from 30 to 1000 or more fps, depending on the application. Each frame is processed, and an output vector between the center of the pupil and the corneal reflection is calculated. This vector is then used to locate the point of gaze, after an initial calibration procedure.

2.3.4 Hand tracking

Much like head and positional tracking, hand-tracking describes the process of constantly capturing a users hands position and movements in real 3d space. Although this technology is still making its first steps in the Virtual Reality field , HMD manufacturers are starting to adopt it, as it opens new horizons and possibilities for VR users and developers. It is very obvious that being able to interact with a virtual environment by simply using your hands (as in real life), will bring the realism and interactivity of virtual reality to a whole new level. It usually works is by allowing the user to see a 3d computer generated replica of his hands in the virtual world, according to their current position and rotation in real 3d space. For example, a user could point his (real) finger to touch a virtual button , thrust his hand forward to punch a virtual enemy or make a gesture to trigger an event.

There are two main techniques that implement hand-tracking:

The first technique, requires a user to constantly hold special controllers with his hands, which act as positional-tracking devices for each individual hand. A similar process to positional-tracking is followed measurements are taken between the controllers and base stations in a room. Then, the position of each hand can be calculated in relativity to the head and room and be projected in the virtual world.

The second technique involves a design similar to eye-tracking which includes image capturing and image processing. One or more camera sensorslocated on the HMD point towards the users hands , capturing their movement in several frames per second. Then advanced mathematical and image processing algorithms analyze each image/frame, producing a 3d representation of how the device sees the hands.

As we will see in the Interactivity chapter, in this project the hand-tracking hardware that was chosen (Leap Motion) utilizes the second technique. This choice was made upon the fact that this technique is very efficient and user-friendly, even more for this specific application. [6]

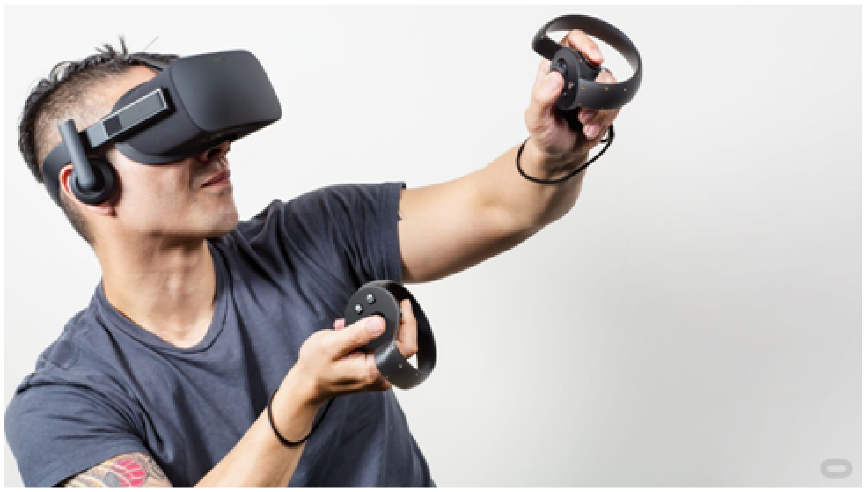


Figure 2.5: wireless controllers with hand-tracking capabilities.

2.3.5 HMD specifications

There are various quality factors that distinct HMDs from one another. Every current or older generation HMD has its advantages and disadvantages, depending on the application it is intended for. The most common HMD specifications are the following [7][8][9]:

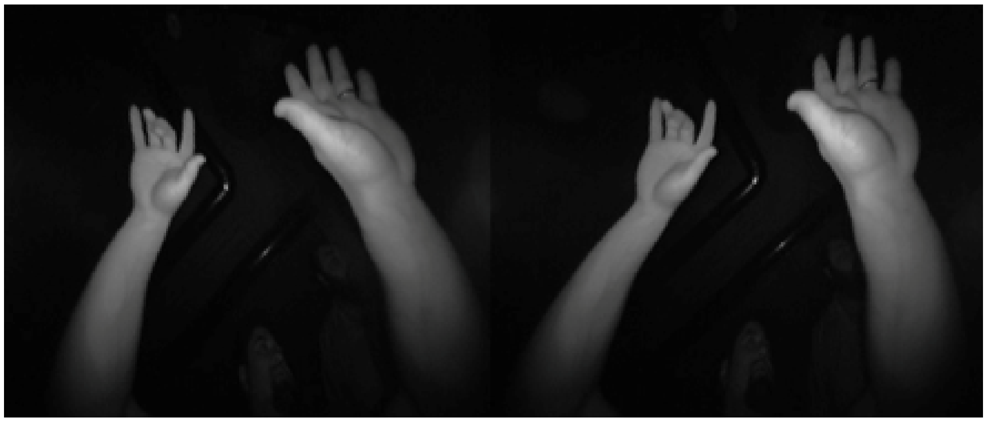


Figure 2.6: hand-tracking by image capture and image processing.

Display Resolution

It is one of the most important factors that plays a big role in visual quality. Due to the fact that the displays of the HMD sit very close to the eye, they must be of a minimum resolution of higher, in other words , have a high pixel density. If this is not the case, lines separating the pixels will become visible to users, which is known as the screen-door effect, distracting from the overall experience and realism. This is one of the biggest issues with current virtual reality headsets which manufacturers are trying to overcome by increasing the display resolution.

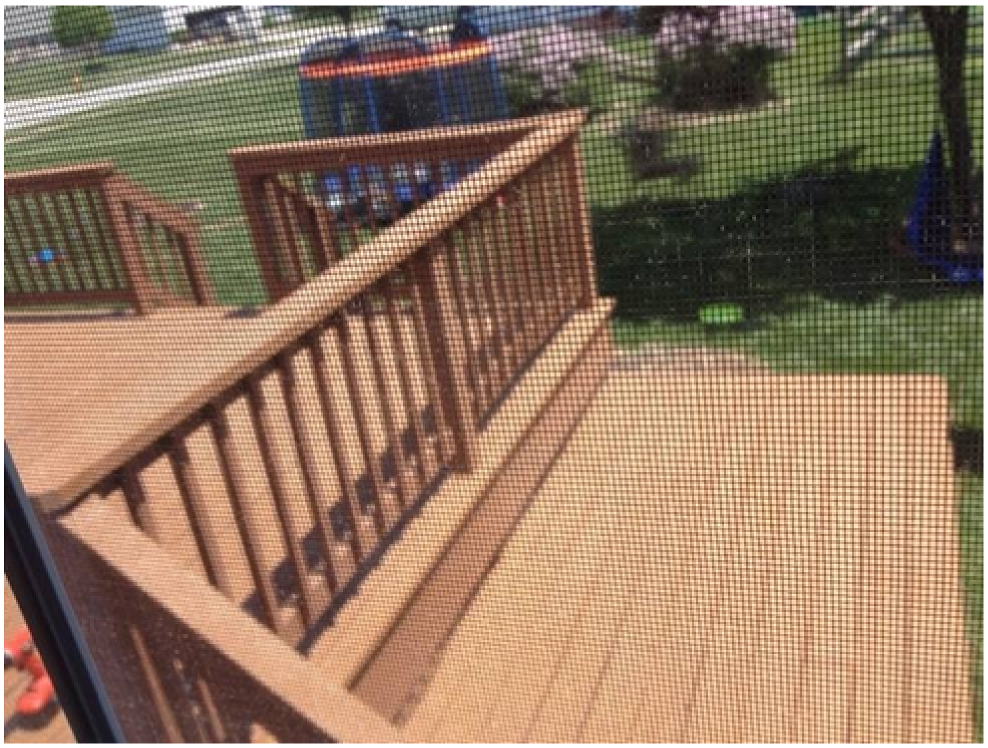


Figure 2.7: Screendoor effect .

Field of View (FoV)

Field of View describes a users viewable area of the world while wearing an HMD, in angles. Wider FOV means a larger area is observable at any given time , consequently giving a higher level of immersion. It is an important aspect of VR experience. A small FOV can harm the feeling of presence in a virtual environment, because users will feel as if they are looking through a small window. To put things into perspective, humans have a total FOV of 200 to 220 degrees (114 of that is binocular) while HMDs are still struggling to pass the 100 barrier.

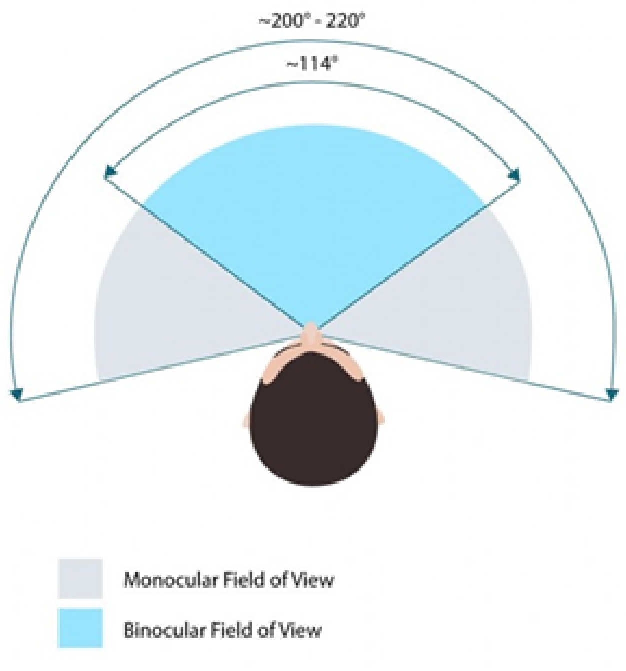


Figure 2.8: Human Field of View

Refresh rate

The refresh rate is the number of times a second that a display hardware completely updates its image. It is sometimes confused with the Frame Rate. The **refresh rate is constant and independent** of the rate a video source can feed the display device with new frames (frame rate). For example, if a computer feeds a 60Hz screen with a frame rate of 30fps, the screen will still update its content 60 times per second, drawing identical frames as needed.As we previously described, the human eye requires a high frequency of image updates in order to perceive separate frames as smooth motion. Therefore, HMD display screens must feature a high enough refresh rate.

Optics - Lenses

Good optics are also required for an immersive VR experience. It is one of the most important factors for image clarity and FoV. The lenses need a short focal length to magnify the displays for best possible FoV, while preserving image quality and minimizing their impact on the ergonomics and design of an HMD. Todays leading headset manufacturers, such as HTC and Oculus are equipping their HMDs with **Fresnel** lenses.

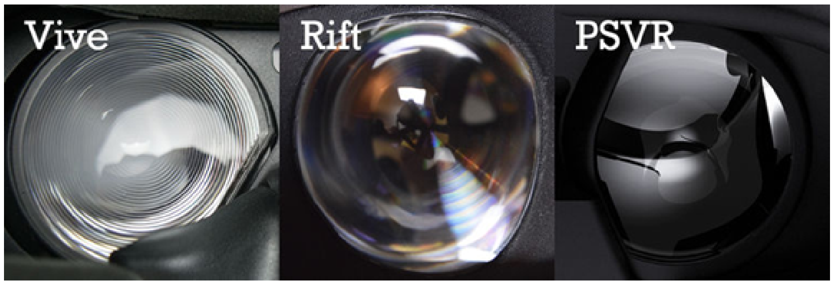


Figure 2.9: Lenses in current-gen HMDs.

This type of lens was originally developed many decades ago for use in lighthouses. It allows for a **short focal length** and **large aperture** while being thin and light, making it an attractive choice for headsets. Fresnel lenses also produce optical diffusion, which helps reduce the screen-door effect to some extent.

Other quality factors

There also more factors that contribute to the final user's experience. These include adjustable IPD(InterPupilary Distance), screen type, latency, weight and more.

2.4 Stereoscopy

Stereoscopy is a technique for creating or enhancing the illusion of depth in atwo-dimensional image. It is usually achieved by presenting two offset im-

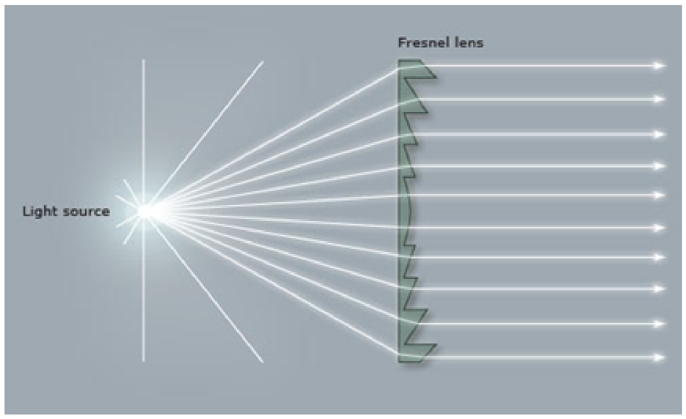
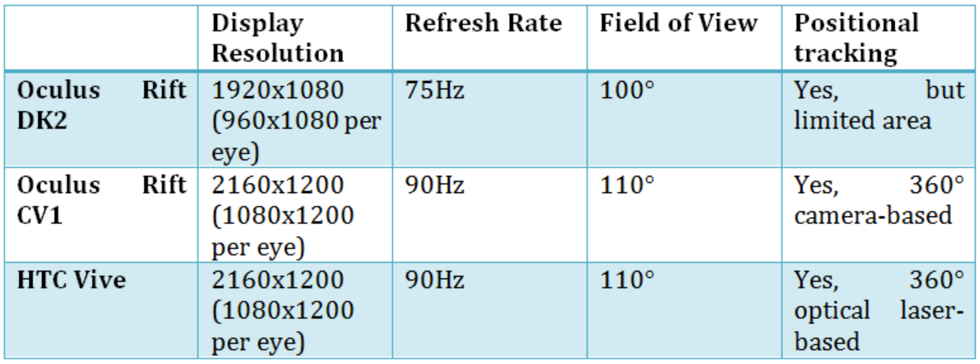


Figure 2.10: Light passing through Fresnel lens.



	Display Resolution	Refresh Rate	Field of View	Positional tracking
Oculus Rift DK2	1920x1080 (960x1080 per eye)	75Hz	100°	Yes, but limited area
Oculus Rift CV1	2160x1200 (1080x1200 per eye)	90Hz	110°	Yes, 360° camera-based
HTC Vive	2160x1200 (1080x1200 per eye)	90Hz	110°	Yes, 360° optical laser-based

Figure 2.11: Comparison of HMDs currently available

ages separately to the left and right eye of the viewer. These two-dimensional images are then combined in the brain to give the perception of 3D depth. Because it is relatively easy to implement , it is commonly used in todays 3D cinemas, TVs and Virtual Reality to produce the 3D depth effect. The viewer has to wear glasses which interact with an appropriate display (via polarization, active-shutter and other methods), providing each eye with a slightly different image. [10]

2.4.1 Stereoscopy in VR and its problems

In virtual reality, stereoscopy creates the illusion of 3D depth, however the effect is somewhat problematic, incomplete and feels unnatural to the hu-

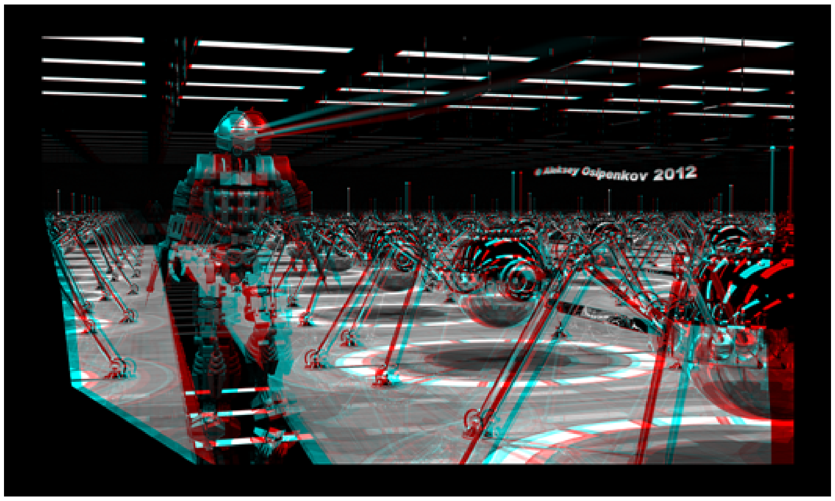


Figure 2.12: two offset stereoscopic images superimposed into one.

man eye. The main reason is that all points and objects of an image are positioned exactly in the same distance from the eye. What this means is, when a viewer tries to focus on far away objects, he is keeping a fixed focal distance but changing the "vergence" angle ofhis eyesin essence, going a little cross-eyed for a moment. That can lead to visual discomfort and fatigue, eye strain, diplopic vision, headaches, nausea, compromised image quality, and even more severe pathologies.
Researches are seeking possible solutions to this problem by experimenting with multiple screen layers to alleviate some of the issues. A second reason with easier solution, is that there might be a smaller or bigger offset than the ideal between the 2D stereoscopic images, and some hardware or software calibration may be required. [11]

Summary
In this chapter we covered the basic technological background of Virtual Reality. We provided the necessary terminology along with current software and hardware technologies used in HMDs.

Chapter 3

3D Computer Graphics

The field of computer graphics is concerned with generating and displaying three-dimensional geometric data in a two-dimensional space (e.g., computer monitor, screen). Whereas a single point in a two dimensional graphic has the properties of position, color, and brightness, a 3D point adds a depth property that indicates where the point lies on an imaginary Z-axis. The creation of 3D graphics begins with the modeling process, followed by the layout and interactivity of the modeled objects in a scene. The process ends with the 3D rendering, which refers to the computer calculations required to display the graphic on a screen

3.1 3D Modeling

3D modeling refers to the process of creating the three dimensional surface of an object. The product of this process, the 3D model, is formed by a collection of points in 3D space connected by geometric entities such as lines, triangles, and curved surfaces. The 3D model can originate manually, created by a user of a 3D modeling tool, algorithmically or it can be scanned into a computer from real world objects (via methods that are mentioned in the previous sections).

The 3D models are separated in two categories. One are the solid models, which define the volume of the object they represent. These are more realistic, but more difficult to build. Solid models are mostly used for non visual simulations such as medical and engineering simulations. The second are the shell or boundary models. These models represent the surface, the boundary of the object, not its volume. Theseare easier to work with than solid models. Because the appearance of an object depends largely on the exterior of the object, boundary models are the ones mainly used in computer graphics. Almost all visual models used in the entertainment industry are shell models.

A 3D model consists of points called **vertices** that define the shape and form **polygons**. A polygon is an area formed from at least three vertices, a triangle , or four vertices , which is called a quad. The line that joins one vertex with another is called edge. There are several modeling methods. The most commonly used ones are:

Polygonal modeling: is a modeling process that represents the surface of an object using polygons. The vertices are connected by line segments to form a polygonal mesh. The vast majority of 3D models today are built as textured polygonal models, because they are flexible and rendered quickly. However, polygons are planar and can only approximate curved surfaces using many polygons.

Curve modeling: Surfaces are defined by curves, which are determined by weighted control points. The curve follows but does not necessarily interpolate the points. Increasing the weight for a point will pull the curve closer to that point.

Digital sculpting: Still a fairly new method of modeling, 3D sculpting has become very popular in the last years. It refers to the use of software that offers tools to push, pull, smooth, or otherwise manipulate a digital object as if it were made of a real-life substance such as clay. It can introduce details to surfaces that would otherwise have been difficult or impossible to create using traditional 3D modeling techniques. The downside is that in order to achieve detail with sculpting the models must have a high number of polygons.

3.2 3D Rendering

3.2.1 Basics

Converting information about 3D objects into a graphics image that can be displayed is known as rendering. It usually **requires considerable memory and processing power**. It is the process of adding realism to computer graphics by adding three-dimensional qualities such as **light**, **shadows** and variations in color and shade. This process is usually performed using 3D computer graphics software. There are many rendering methods that have been developed, each one appropriate for specific applications. There is the non photorealistic rendering, which gives the effect of painting, drawing or cartoons, and the rendering methods aiming to achieve high photorealism.

Another categorization is suitability for real time rendering and non real time rendering. Non real time rendering is implemented in non interactive media such as films and video. In this case, the rendering process can take huge amounts of time. That is because non real time rendering has the advantage of very high quality even with limited processing power due to the absence of real time response, which makes the time for the rendering process not considerable. A method suitable for non real time rendering is ray tracing, which simulates the path of a single light ray as it would be absorbed or reflected by various objects in the scene. Real time rendering is implemented in interactive media such as games and simulations. The calculations and the display are happening in real time. The primary goal is to achieve an as high as possible degree of photorealism at an acceptable rendering speed. This is 24 frames per second, as that is the minimum the human eye needs to see to successfully create the illusion of movement.

3.2.2 Global Illumination (GI)

Global illumination (shortened as GI) or indirect illumination is a general name for a group of **algorithms used in 3D computer graphics** that are meant to add more realistic lighting to 3D scenes. Such algorithms take into account not only the light which comes directly from a light source (direct illumination), but also subsequent cases in which light rays from the same source are reflected by other surfaces in the scene, whether reflective or not (indirect illumination).

Images rendered using global illumination algorithms often appear more photorealistic than images rendered using only direct illumination algorithms. However, such images are **computationally more expensive** and consequently much slower to generate. One common approach is to compute the global illumination of a scene and store that information with the geometry, e.g., radiosity. That stored data can then be used to generate images from different viewpoints for generating walkthroughs of a scene without having to go through expensive lighting calculations repeatedly.

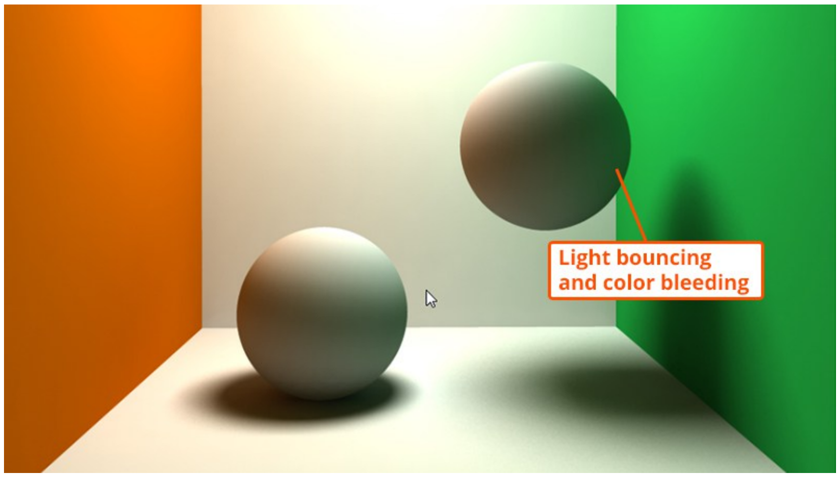


Figure 3.1: a scene rendered with GI. Notice that the green color of the wall is being cast onto the sphere on the right side of the image. It is the result of indirect lightning. The green wall is reflecting some of the light coming from a light source. *Source: http://blog.digitaltutors.com/understanding-global-illumination/.*

Radiosity, ray tracing, beam tracing, cone tracing, path tracing, Metropolis light transport, ambient occlusion, photon mapping, and image based lighting are examples of algorithms used in global illumination, some of which may be used together to yield results that are not fast, but accurate.

These algorithms model diffuse inter-reflection which is a very important part of global illumination; however most of these (excluding radiosity) also model specular reflection, which makes them more accurate algorithms to solve the lighting equation and provide a more realistically illuminated scene.

3.3 Definitions

3.3.1 Polygon Mesh

A polygon mesh consists of vertices, edges and faces that define the shape of a 3D object. The vertices are points which represent positions along with other information such as color and texture coordinates, an edge is a connection between two vertices and a face is a closed set of edges. The faces can be represented by triangles (three edges), quads (four edges) or convex polygons (five or more edges). A vertex can be shared by many adjacent faces, and an edge can be shared by no more than two faces. The faces share edges and form the three dimensional surface of an object like tiles.

A polygon mesh is fairly easy to render, however most rendering hardware supports only triangle or four sided faces (quads). It should be noted that most computer graphics software require that objects be composed entirely of triangles or quads. [24]

3.3.2 Texture mapping

There are two ways of adding detail to a surface. One method is to add details via modeling. That means that extra polygons need to be created in order to form the details. This results in increasing the scene complexity and consequently the rendering speed. In addition there are some fine details that are very hard to model. The other method, which is a more popular approach is to map a texture to the surface. Texture mapping is making possible to simulate photorealism in real time, by vastly reducing the number of polygons and lighting calculations needed to create a realistic 3D scene.

A texture map is a bitmap image that is applied to the surface of a polygon. Every vertex in a polygon is assigned a texture coordinate, which is known as a UV coordinate. Image sampling locations are then interpolated acrossthe face of the polygon to produce a visual result that seems to have more details than could otherwise be achieved with a limited number of polygons. Texture mapping was originally a method of simply mapping pixels from a texture to a 3D surface. Nowadays many complex mapping methods have been developed in order to add photorealism and detail to a flat surface. Some of the most used mapping methods are presented below. [25]

Diffuse mapping: A diffuse map is a texture you use to define a surface's main color. It is the most frequently used texture mapping method. It wraps a bitmap image onto the surface while displaying its original pixel color. Any bitmap image, such as images captured by digital camera, can be used as diffuse map. It sets the tint and intensity of diffuse light reflectance by the surface.

Bump mapping: is a texturing technique for simulating small displacements on the surface of an object, while the surface geometry is not modified.

Instead only the surface normal is modified as if the surface had been displaced. This is achieved by perturbing the surface normals (the imaginary vectors perpendicular to the surface that are used in lighting or shading calculations) of the object and employing the perturbed normal during lighting calculations. It uses the values of a grayscale image to simulate abnormalities on the surface. Black areas recede and white areas protrude.

Normal mapping: is considered a newer and better variation of bump mapping. The detail they create is also fake no additional resolution added to the geometry. A common use of this technique is to greatly enhance the appearance and details of a low polygon model by generating a normal map from a high polygon model or a height map. (a height map is a raster image used to store values such as surface elevation data, for display in 3D computer graphics). Normal maps consist of red, green, and blue. These RGB values translate to x, y, and z coordinates, allowing a 2D image to represent depth. The 2D image is applied to the surface. This way, a surface simulates the lighting and color associated with 3D coordinates. In the normal map, each pixel's color value represents the angle of the surface normal.

Displacement mapping: In contrast to bump and normal mapping, displacement mapping adds real 3D detail to a surface, based on a displacement map. Like a bump map, a displacement map consists of grayscale values. They can either be baked from a high resolution model or painted by hand. Due to the fact that real geometry is added in real-time, the mesh must be subdivided or tessellated and render times are increased.

Other types of mapping include specular mapping, which allows parts of an object to have a specular effect, reflection mapping which is a technique for approximating the appearance of a reflective surface by means of a pre computed texture image and opacity mapping which determines which partsof the object should be transparent and how much.

Nowadays, two or more of these techniques are used simultaneously on most surfaces to simulate real environments, while increasing the scene complexity by a little to no amount.

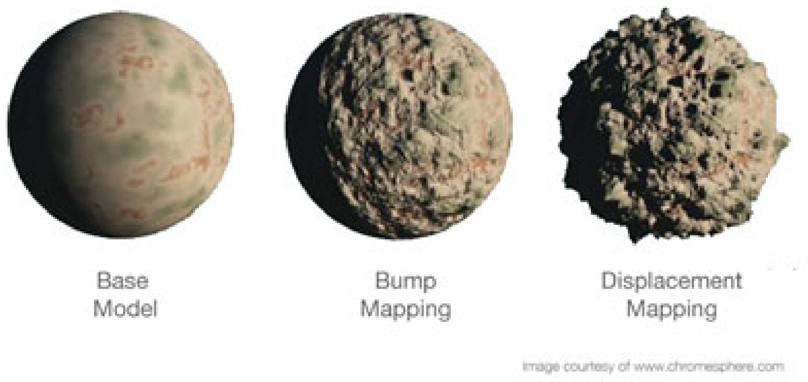


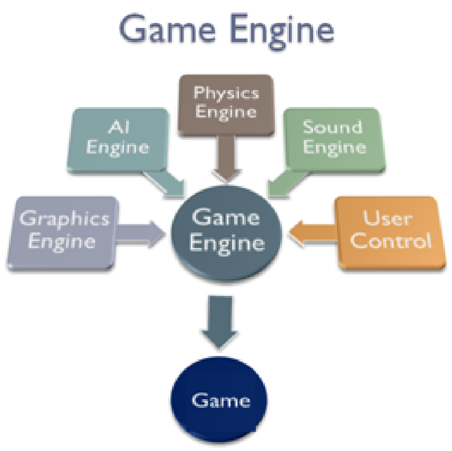
Figure 3.2: Applying texture map to a simple model

3.3.3 Shader

A shader is considered a program that is executed during the rendering procedure and it refers to the process of altering the color of a surface in the 3D scene, based on its angle to lights and its distance from lights to create a photorealistic effect. Shading is also dependent on the lighting itself that is used in the scene. Shaders are most commonly used to handle a scenes lighting and shadow effects. Beyond that more complex uses include altering the hue, saturation, brightness or contrast of an image, producing blur, normal mapping, light bloom, volumetric lightning, distortion, and a wide range of other.

3.4 Game Engines

3.4.1 Introduction to Game Engines



A game engine is a software framework designed for the creation and development of video games. Developers use them to create games for consoles, mobile devices and personal computers. The core functionality typically provided by a game engine includes a rendering engine (renderer) for 2D or 3D graphics, a physics engine, a collision detection (and collision response) system, sound, scripting, animation, artificial intelligence, networking, streaming, memory management, threading, localization support, scene graph, and may include video support for cinematics. The process of game development is often economized, in large part, by reusing/adapting the same game engine to create different games, or to make it easier to "port" games to multiple platforms.

T he beauty and power of game engines, is that they speed-up the development process, by providing a suite of visual development tools,reusable software components and simplification of frequently used tools, elements and processes. Game Engines are usually built upon one or multiple rendering application programming interfaces (APIs), such as Direct3D or OpenGL which provide a software abstraction of the graphics processing unit (GPU). These APIs are commonly used to interact and communicate with the GPU, to achieve hardware-accelerated rendering.

Modern game engines are some of the most complex applications written, which is the result of years and years of improvements , experience and development. Nowadays they often feature dozens of finely tuned systems interacting to ensure a precisely controlled user experience. The continued evolution of game engines has created a strong separation between rendering, scripting, artwork, and level design. It is now common, for example, for a typical game development team to have several times as many artists as actual programmers. [12]

Furthermore, due to the constant growth of the smartphone application market and increasing competition, popular high-end Game Engines are proving to be a precious tool for developers worldwide, to bring their ideas and games to life, in as many platforms as possible.

3.4.2 Popular Game Engines

In this section, we will take a brief look at 4 of the most popular free game engines currently available, and explain which is more suitable for this project and why.

Unreal Engine

Unreal Engine(UE),initially released on 1998, is a complete suite of game development tools, powering hundreds of games,simulationsandvisualizations. It is one of the most advanced engines to date, delivering top quality visuals while providing users with a large variety of tools to work with everything they need. Due to its capabilities ,efficient design and ease of use it is well-appreciated engine from hobbyists to development studios. It is also available for free. Developers can also port their projects to mobile devices, both iOS and Android. Unreal Engine also works with Virtual Reality. Finally, UE also gives access to its users with to a marketplace , to buy re-usable content and add to their project, speeding the development process. [13]

Unity3D

Unity 3D, initially released on 2005, is a flexible and powerful development platform for creating high quality 2D and 3D games. Emphasizing on portability, Unity currently supports over 20 platforms, including PCs, consoles, mobile devices (iOS and Android)and websites. Additionally, many settings can be configured for each platform. As a result, Unity can detect the best variant of graphic settings for the hardware or platform the game is running, thus optimizing performance and sacrificing visual quality if necessary. Apart from its next-generation graphical capabilities, Unity also comes with an integrated physics engine (nVidias PhysX). Much like Unreal Engine, Unity offers developers an Asset Store to buy re-usable content and assets for use in their project. To sum up, due to its ability to efficiently target multiple platform at once and user-friendly environment, this game engine is an ideal choice for a large portion of developers.

CryEngine

CryEngine is a game engine developed by game developer Crytek , which has been used in all of their titles. It is known for its ability to produce stunning, eye-catcing graphics and visuals, featuring advanced shader and lightning systems. Because of this, CryEngine clearly targets only powerful PCs and high-end consoles. It comes with VR support and a large amount of advanced visual features, tools, audio/physics systems and character and animation systems.CryEngine can be downloaded and used for free. [14]

Amazon Lumberyard

Amazon Lumberyard is a free game engine developed by Amazon and based on the architecture of CryEngine. Lumberyard has similar capabilities to CryEngine and can be used for production of high quality games targeting high-end platforms. It is remarkable that the entire source code can be viewed and changed by the developers to suit their needs. This engine focuses on a fee-based managed system for cloud building and hosting, intended to allow developers to easily develop games that attract ”large and vibrant communities of fans, as stated by the company. [15]

3.4.3 Comparison/Choosing the right engine

Unreal Engine and Unity are currently ahead of the competition as the two most popular game engines available to the public. This is due to the fact that they both succeed in providing high-end graphics, a large variety of usable tools, great support for platforms and devices, without compromising usability and efficiency. It is important to note that these 2 engines offer a large community support, which is also something that has to be considered when choosing an engine. CryEngine and Lumberyard are also great and powerful engines with remarkable capabilities, however their complicated structure and smaller community excluded them from our consideration.

In conclusion, taking into account the advantages and disadvantages of each engine, Unity proved to be the ideal choice for this project, mainly due to its efficiency, large community support and ease of use.

Summary

In this chapter we provided an insightful review of technologies used in 3D Graphics, from the creation of 3D models, to the rendering of photorealistic scenes. Moreover, we reviewed and compared today's most powerful free game engines available to the public.

Chapter 4

Technological Background

4.1 Unity3D

4.1.1 What is Unity3D?

Unity 3D is a powerful cross-platform 3D game engine with a user friendly development environment. Unity 3D helps developers create games and applications for mobile, desktop, the web, and consoles. Its 3D environment is suitable for laying out levels, creating menus, doing animation, writing scripts, and organizing projects. Unitys primary goal may be the development of 3D video games, however, it is also suitable to create other kinds of interactive content, such as animations, simulations or 3D visualizations.

Unity is a fully integrated development engine that provides functionality to create interactive 3D content. With Unity the developer can assemble assets into scenes and environments, add lighting, audio, special effects, physics and animation, simultaneously play, test and edit the application, and when ready, publish to chosen platforms, such as Mac, PC and Linux desktop computers, Windows, the Web, iOS, Android, Windows Phone 8, Blackberry 10, Wii U, PS3 and Xbox 360. Unitys complete toolset, intuitive workspace and rapid, productive workflows help users to drastically reduce the time, effort and cost of making interactive content.

4.1.2 Project structure in Unity3D

Unity is defined by its component based architecture. Its workflow builds around the structure of components. Each component has its own specific job, and can generally accomplish its task or purpose without the help of any outside sources.

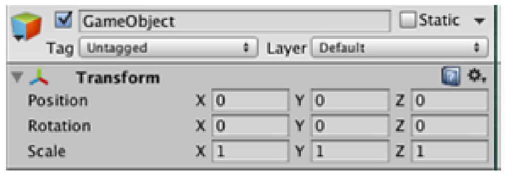
Each game or application created in Unity is called a **project**. Each project consists of one or more **scenes**. Scenes contain the objects of the game. They can be used to create a main menu, individual levels, and anything else. Every scene is considered as a unique level. In each scene, the user can position the 3D models, construct the environment and essentially design most of the functionality. Every object placed in a scene is considered a **GameObject**. GameObject sconsist of one or more components. **Components** are Unitys fundamental elements, which are used to define properties, behavior and characteristics of a GameObject. The user can add a wide variety of components in a GameObject to achieve the desired functionality.

4.1.3 Components

Some of the most commonly used components in Unity are presented next.

Transform Component

Every GameObject contains a Transform Component. When creating a GameObject a transform component is added automatically. It is impossible to create a GameObject without one or remove it. The Transform Component is one of the most important and most frequently accessed Component. It defines a GameObject's position, rotation, and scale in the game world based on the x,y,z coordinate system.



GameObject Static

Tag Untagged Layer Default

Transform

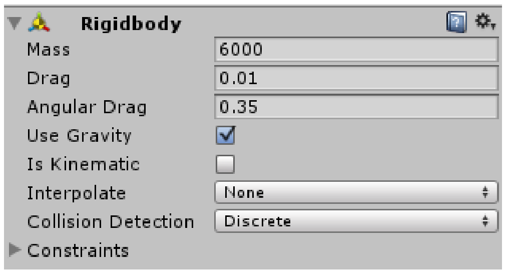
Position	X	0	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

These parameters are initialized by hand and/or can modified in runtime by script to make objects move, rotate and more. It is important to note that when scripting functionality such as movement, Unity considers the Z axis as forward/backwards , Y axis as up/down and X axis as left/right.

Mesh Component

3D meshes are the main graphic object primitive of Unity. Various components exist in Unity to render meshes. The most commonly used components are the mesh renderer and the mesh filter. They are used in collaboration in order to display an object. The mesh filter takes a mesh from your assets and passes it to the mesh renderer for rendering on the screen. The mesh renderer takes the geometry from the mesh filter and renders it at the position defined by the object's transform component. When importing mesh assets, Unity automatically creates a mesh filter along with a mesh renderer. Another component is the text mesh. It generates 3D geometry that displays text strings.

Physics Component


Rigidbody

Mass	6000
Drag	0.01
Angular Drag	0.35
Use Gravity	✓
Is Kinematic	
Interpolate	None
Collision Detection	Discrete
Constraints	

Physics components allow the user to give objects realistic motion and reaction to collisions by simulating physics laws. Unity has NVIDIA PhysX physics engine built-in. A physics engine is computer software that provides an approximate simulation of physical systems. This allows for unique realistic behavior and has many useful features. A **rigidbody** component makes the object that is attached to be affected by gravity or linear and angular forces and collide with other objects. There is also a variety of collider components (mesh, box, sphere, wheel collider) which surround the shape of an object for the purposes of detecting physical collisions. In this project, due to its simulation nature, a physics component was attached on the Aircraft game object.

Rendering Component

These are the components that have to do with rendering in-game and user interface elements, as well as lighting and special effects. The camera component is essential as it is used to capture and display the world to the player. It can be customized and manipulated to fulfill the requirements of the users application. The GUI Texture and GUI Text components are made especially for user interface elements, buttons, or decorations as well as displaying text on screen. Another important rendering component is the light component. It brings a sense of realism. Lights can be used to illuminate the scenes and objects, to simulate the sun, flashlights, or explosions just to name a few.

Audio Component

These components are used to implement sound. The most important component here, is the **Audio Source** component, which as the name suggests, plays a sound file at the location of the game object it is attached to. The developer can set parameters such as sound volume, pitch and change the sound file to be played at any time. These parameters can also be changed by script during run-time.

Script Component

The script component is used to attach a script onto a game object. Scripts are often attached to objects, to define their behavior and trigger effects upon specified conditions. More about scripts in the following section.

Materials and Shaders

Materials and shaders are crucial components that are categorized in the asset component group. There is a close relationship between materials and shaders. Materials are used in conjunction with mesh renderers and other rendering components used in Unity. They play an essential part in defining how the object is displayed. The properties that a materials inspector displays are determined by the shader that the material uses. A shader is a specialized kind of graphical program that determines how texture and lighting information are combined to generate the pixels of the rendered object onscreen. In other words, it tells the graphics hardware how to render surfaces.The user can select which shader each material will use. Specifically, a material defines which texture and color to use for rendering, whereas the shader defines the method to render an object.

4.1.4 Scripting

Scripting is an essential part of Unity as it defines the entire behavior of the game or application. Even the simplest game needs a script to respond to

input. Scripts can be used to create graphical effects, control physical behavior of objects or characters , trigger effects upon specified conditions and generally bring a game to life. Unity supports two programming languages: C sharp which is similar to C++ or Java and javascript. The scripts can be written and edited in MonoDevelop, which is an integrated development environment (IDE) within Unity. An IDE combines a text editor with additional features for debugging, auto-complete and other project management tasks.
Scripting is linked with the component based architecture Unity uses. As it was mentioned above, the behavior of GameObjects is controlled by the components that are attached to them. Thus, components can be accessed or modified by script at any time to achieve desired behavior and functionality. Each script makes its connection with the internal workings of Unity by implementing a built-in class called MonoBehaviour. This class refers to the component that can be enabled or disabled. Javascript automatically derives from the class without the need to be declared, whereas the other two languages have to explicitly declare the class.

When a script is created, there are two functions automatically declared in it, **Start**() and **Update**(). Start is called when a script is enabled and it is called exactly once in its lifetime.

- The **Start** function is the place where initialization occurs. It is used to initialize an objects position, state and properties or load other scripts and gameobjects for later use.
- **Update** is the function that implements game behavior. It is called in every frame and is crucial for checking and modifying the state of various parts of the application. From a programming standpoint each game or application runs in loop, which allows it to run smoothly regardless of a user's input or lack thereof. For a change to occur an event must be activated. The update function checks every frame for such events, which can be changes to position, state and behavior and determines the outcome. The start function is called by Unity before the Update function is called for the first time.

There are other kinds of event functions that can trigger a change. There are the input event functions, that track the mouse movement and input.

These functions allow a script to react to user actions with the mouse. Some examples of those functions are OnMouseDown, which is called when the user has pressed the mouse button, OnMouseUp which is called when the user has released the mouse button, OnMouseEnter which is called when the mouse enters a GUI element, etc.

Another important function is the OnGUI function. Unity has a system for rendering Graphical User Interface (GUI) controls over the main action in the scene and responding to clicks on these controls. The code handling those events is treated somewhat differently from the normal frame update and is placed in the OnGUI function, which is called multiple times per frame update.
Apart from the functions provided by Unity, the developer can create his/her own functions in order to control or determine the behavior of a GameObject, change the properties of a component or altering the overall state of the application. In order for these custom functions to be executed, they have to be called inside a Unity event function, like Update.
The most commonly used functions were presented briefly above, as well as the concept of how they are used. The basic notion of the Unity scripting is that the scripts are components that can control the GameObject. Each component property corresponds to a script variable and the scripts can access not only the components of the GameObjects they are attached to , but also other GameObjects.

4.1.5 Interface

Unity has a relatively simple and user-friendly interface, which can be also customized to meet each developers needs. The basic tabs and buttons are shown and explained below.

1. Tools to move, scale, rotate and resize GameObjects in the game world

2. Hierarchy: Contains every GameObject in the current scene

3. Project Brower: Contains the entire directory of assets and files that belong to the specific project. They can be accessed and placed to the scene at any time. Users can also drag and drop assets from the project brower directly to the Hierarchy(2) or the Scene(6) tab.

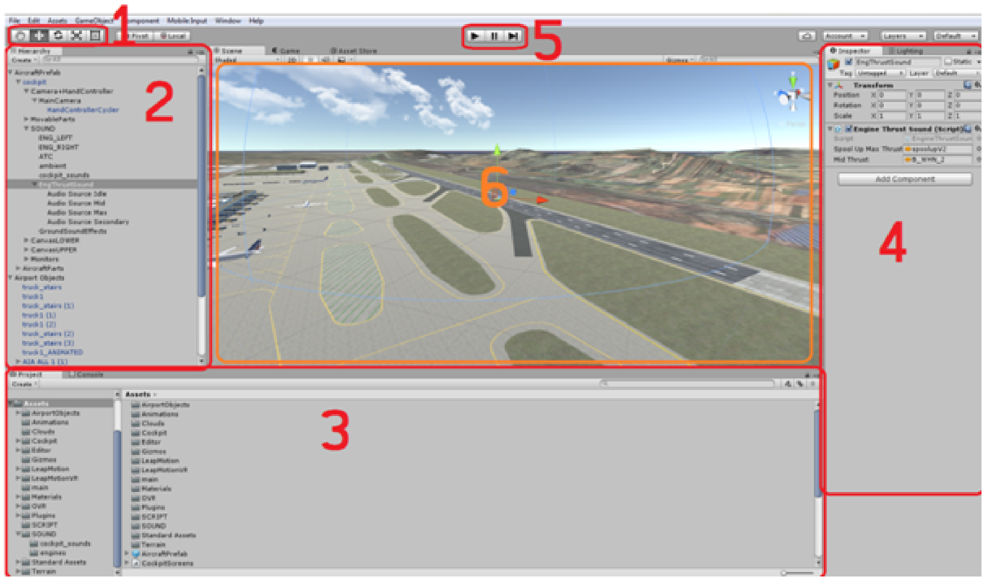


Figure 4.1: Unity's user interface.

4. Inspector: This panel displays information and properties of a currently selected gameobject, including its attached components.

5. Developers can press the play button to instantly run their project within the development environment to debug and test functionality. They can also pause at any time to perform any changes and see the outcome live.

6. Scene/Game panel: 3d viewer and editor of the current scene. Developers can move around anywhere they want and zoom in and out of objects.

4.2 3D Modeling Tools

4.2.1 Basics

In the process of creating games and 3d applications in general, one or more tools are used in order to create - model the 3d objects that exist in a 3d environment. These tools are referred to as 3d modeling software, and dozens of them exist. Each tool is different in its way. Some of them are simpler, others require experience and expertise. Some are free and some are quite expensive. Some tools focus on realistic visualizations and architecture, while others do a better job for 3d games modeling or animations. It is up to the developer to choose the tool or tools that satisfies his demands best. AutoCAD, 3D Studio Max, SketchUp, Blender and ZBrush are some of the most popular tools to date. In this thesis, the 3d environment , including the cockpit, airport and other objects ,was modeled in SketchUp.

4.2.2 SketchUp

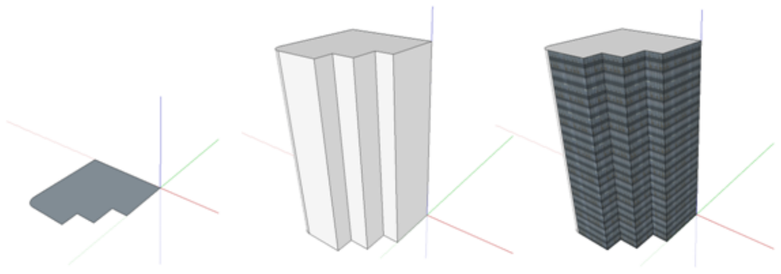


Figure 4.2: Example of creating a simple 3D office building in SketchUp.

SketchUp is a 3d modeling program. It is particularly known for its ease of use, even by users with little or no experience over 3d modeling, which also is what makes it popular and stand apart from its competitors. SketchUps main characteristic is that it allows the creation of 3d models by actually drawing and drag-and-dropping on a 3d space. Users start the modeling process of their scene by using 2D primitives such as lines, circles and planes. Then depth can be added by simply pulling a 2D surface in/out. For example, a building can be easily created by drawing its perimeter as a 2d surface, then pulling it out to add depth (in this example height) and thats it. Of course more advanced tools are provided for more details.

Like most modeling software, SketchUp has a built-in texturing system. Although rather simple and not as complete as others programs, users can import their textures to paint their objects by clicking on the desired surface. It also gives the ability to alter the position , scale and rotation of a texture on a surface to achieve the desired result.
The main disasdvantage of SketchUp is that due to its simple interface and

limited set of tools, it lacks the professionalism of other modeling tools. This means that some advanced and complex operations on a 3d model can not be performed.
Finally, SketchUp comes with a variety of plugins that allow 3d models to be exported in many formats. So when a 3d object/environment has been modeled, we can export it as a format Unity understands.

In this project we chose SketchUp as our 3D modelling software, as it is free, easy to use and adequate for the type of models we need in this project. We created the 3D models in SketchUp and applied basic diffuse textures. For more details, such as normal maps, emission maps, reflections and more, we used Unity's build-in material system, which is advanced enough to produce realistic graphics.

Summary

In this chapter we examined and presented the two main software tools used in this project, Unity3D engine and SketchUp. We covered the fundamentals of Unity and its component-based architecture. We also presented the basic structure of scripts and user interface. As for the 3D modeling tools, we explained their purpose and we justified our choice.

Chapter 5

Implementation

5.1 3D Environment

5.1.1 Cockpit

As it was mentioned in the introductory chapter, the cockpit presented in this Flight Simulator is closely based upon the real Airbus 320 passenger jet cockpit. Each part was carefully modeled based on its real dimensions, size, shape and material. During the process, real photographs of various angles were used and imported into our 3d modeling software, in this case SketchUp, as our guide to create the 3d replica.

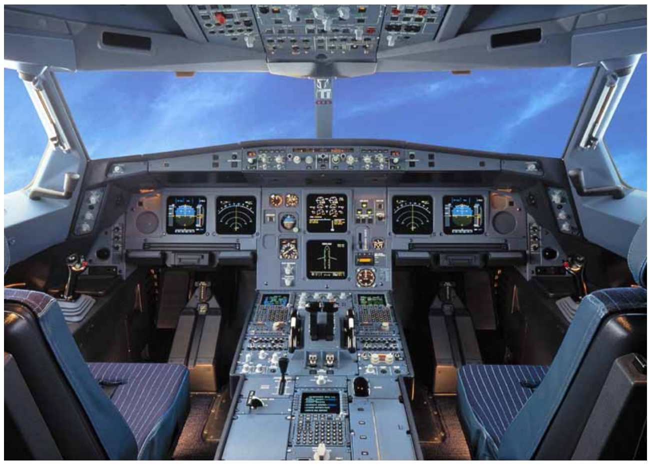


Figure 5.1: Airbus 320 cockpit

We modeled most of the cockpit details by importing real photographs in SketchUp. This way we could accurately replicate the parts of a real cockpit, one by one. Then, we stitched them together precisely, textured them, and added smaller details, such as buttons and switches.

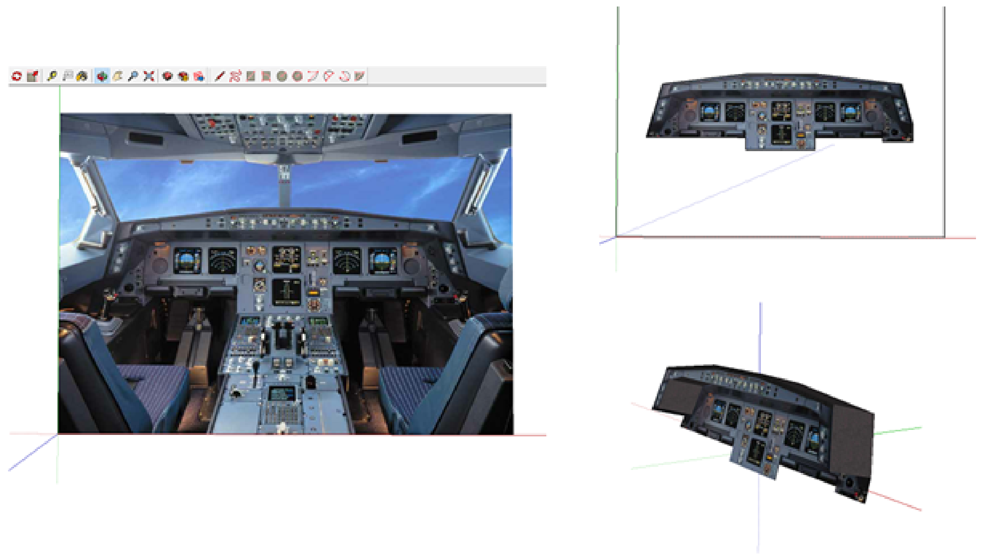


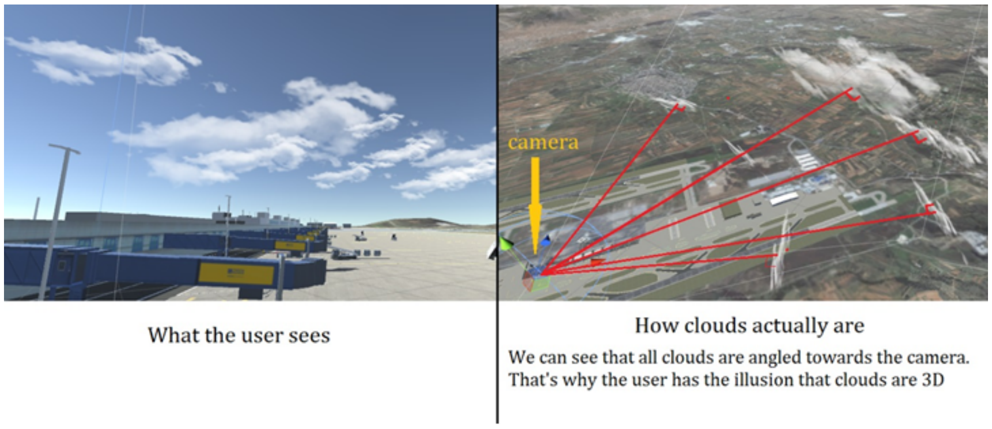
Figure 5.2: Making the 3D model based on a photograph in SketchUp

5.1.2 Clouds

The clouds in this simulation look realistic and appear to be 3d, but in fact they are not. Clouds are basically 2D textures with transparency that are stretched on 2D rotating planes, which are randomly positioned in the sky. To be more specific, we apply a simple but effective technique. The texture of the cloud constantly rotates in a way that the normal (perpendicular) vector of its surface is always pointing towards the users eyes, the game camera to be more precise. As a result, the user, no matter where he is positioned, always sees the front surface of the cloud, thus giving the illusion that the cloud has depth. The figure below illustrates the above technique:

Unity Implementation

Firstly we created 2D planes and positioned them in the sky at random po-


camera
What the user sees
How clouds actually are
We can see that all clouds are angled towards the camera.
That's why the user has the illusion that clouds are 3D

sitions. Then we edited the plane scale to match with a real clouds size. Secondly we created a new material to add to the plane, to turn its white surface into a realistic cloud. We change the default shader to *Particles/Alpha Blended*. We do this because our .png texture files has transparency, and because a cloud can be considered a form of particles.
Finally, we need to make a script to apply the rotation technique explained above. The main part of the code that implements it is :

```
void Update() {
        transform.LookAt (MainCamera.transform);
}
```

Where **transform** refers to the transform of the cloud the script is attached to, and **MainCamera.transform** refers to the transform of the camera the users eyes. This function is called in every frame update. The **LookAt**() function rotates the transform so the forward vector points at target's current position.

5.1.3 Terrain

The terrain used for this simulation is a collection of real satellite images, imported and stitched together in Unity. The featured terrain covers a large portion of the Attica region, Greece, including Athens International Airport. The covered surface area is equal to approximately 1500 km^2.

The process was split into 3 parts. Firstly we downloaded the required satellite images. Secondly we created the terrain objects and imported the

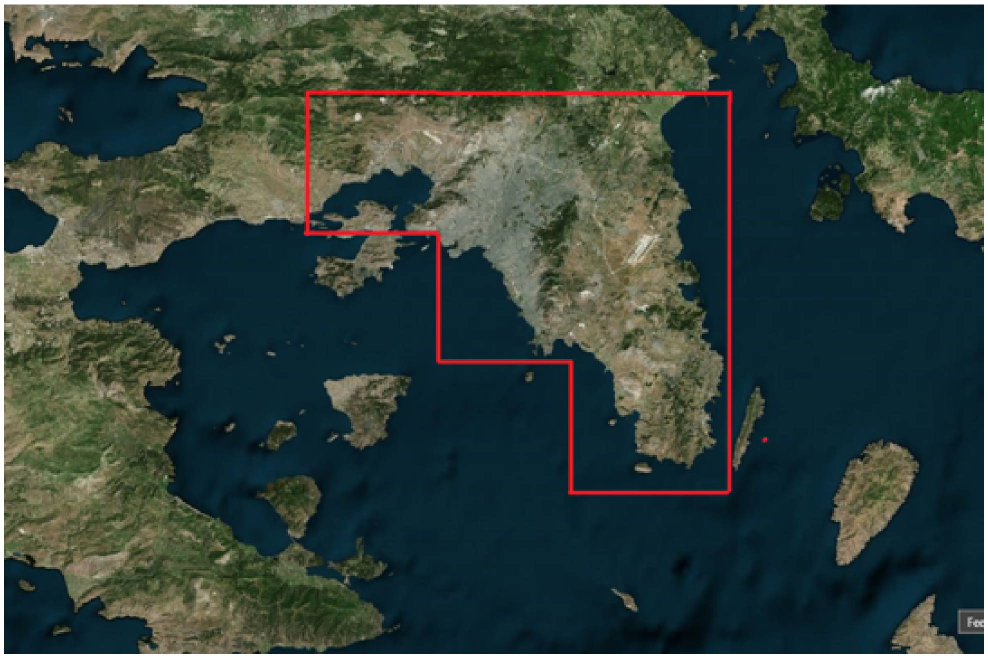
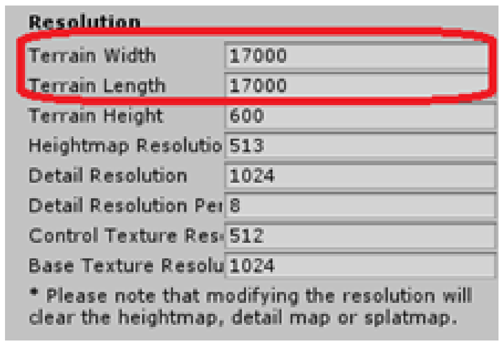


Figure 5.3: Illustration of the area covered. Source: BingMaps

satellite images to Unity. Thirdly we added elevation to the flat terrain.

Step 1 : Acquiring satellite terrain images.
We used *MapPuzzle* , a small software solution that allows automated satellite imagery download. The reason we used this tool, is that it automatically combines all downloaded images, into one very high resolution image. Otherwise we would have to stitch together hundreds of different images, needless to say, a painful and time-consuming process.

Step 2 : Creating, positioning and texturing terrain in Unity.

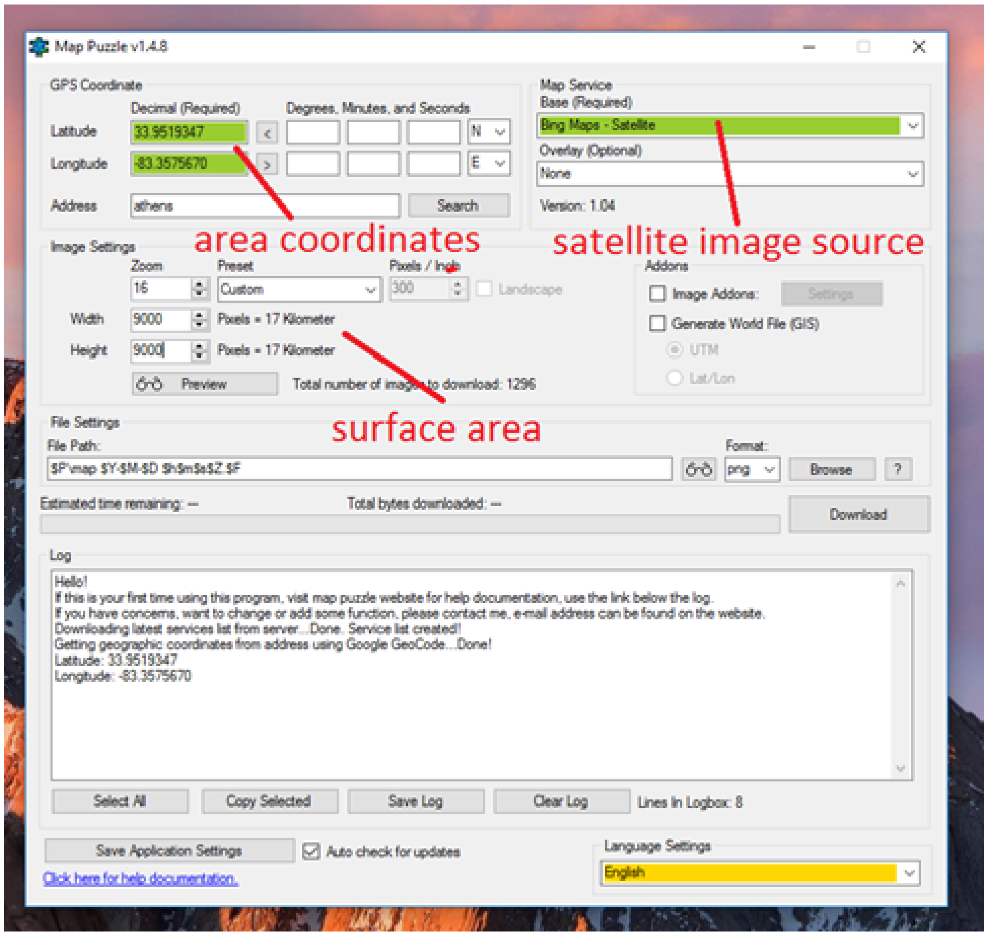


Resolution

Terrain Width	17000
Terrain Length	17000
Terrain Height	600
Heightmap Resolutio	513
Detail Resolution	1024
Detail Resolution Pe	8
Control Texture Res	512
Base Texture Resolu	1024

* Please note that modifying the resolution will clear the heightmap, detail map or splatmap.

We import the high resolution terrain images into Unity, as downloaded by the Map Puzzle application. Then we need to create the terrain game objects. Each terrain object will be textured by one high res-



GPS Coordinate
Decimal (Required) — Degrees, Minutes, and Seconds
Latitude: 33.9519347 — N
Longitude: -83.3575670 — E
Address: athens

Map Service
Base (Required): Bing Maps - Satellite
Overlay (Optional): None
Version: 1.04

Image Settings
Zoom: 16
Preset: Custom
Pixels / Inch: 300
Landscape
Width: 9000 Pixels = 17 Kilometer
Height: 9000 Pixels = 17 Kilometer
Total number of images to download: 1296

Addons
Image Addons:
Generate World File (GIS)
UTM
Lat/Lon

File Settings
File Path: $P\map $Y-$M-$D $h$m$s$Z.$F
Format: png

Estimated time remaining: --
Total bytes downloaded: --

Log

```
Hello!
If this is your first time using this program, visit map puzzle website for help documentation, use the link below the log.
If you have concerns, want to change or add some function, please contact me, e-mail address can be found on the website.
Downloading latest services list from server...Done. Service list created!
Getting geographic coordinates from address using Google GeoCode...Done!
Latitude: 33.9519347
Longitude: -83.3575670
```

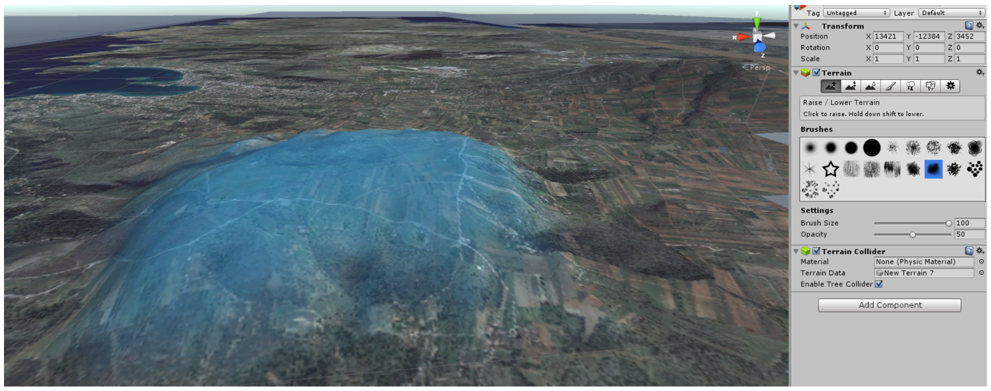
Lines In Logbox: 8
[✓ Auto check for updates]
Language Settings: English

Figure 5.4: Map Puzzle v.1.4.8 main window

olution terrain image. Each terrain image is 9000x9000 pixels, which corresponds to 17km width x 17 km height = $289km^2$ surface area. Furthermore, it is important to note that 1 distance unit in Unity3D equals 1 meter.

Step 3 : Adding elevation.

At the moment, our terrain is correctly positioned and textured, but is a completely flat surface without any elevation. Thus we can use one of Unitys paint brushes to raise/lower the terrain. This is feasible by simply selecting a brush type (different brushes produce different effects and surface anomalies) and clicking on the desired terrain area.



```
Tag Untagged          Layer Default
Transform
Position   X 13421  Y -12384  Z 3452
Rotation   X 0      Y 0       Z 0
Scale      X 1      Y 1       Z 1
Terrain
Raise / Lower Terrain
Click to raise. Hold down shift to lower.
Brushes
Settings
Brush Size   100
Opacity      50
Terrain Collider
Material             None (Physic Material)
Terrain Data         New Terrain 7
Enable Tree Collider ✓
```

Figure 5.5: Adding elevation to the terrain in Unity

5.1.4 Airport

Like most models, the airport was modeled and textured in SketchUp. The featured airport is highly based upon Athens International Airport, Greece. It features the main terminal buildings, two main runways for take-off and landing, taxiways and other objects that add realism and detail, such as aircraft jetways and parking areas, ground service vehicles and more.

When starting the simulation, the user can choose between 2 locations to start. The first option is at a gate at the main terminal buildings, and the second option is at the start of runway 03R.

5.2 Physics

5.2.1 How do airplanes fly?

Airplanes are constructed such that the airflow pattern around them generates lift, thereby enabling them to fly. The airflow is produced by the forward motion of the plane relative to the air. This forward motion is produced by engine thrust, delivered by way of propeller engines or air-breathing engines

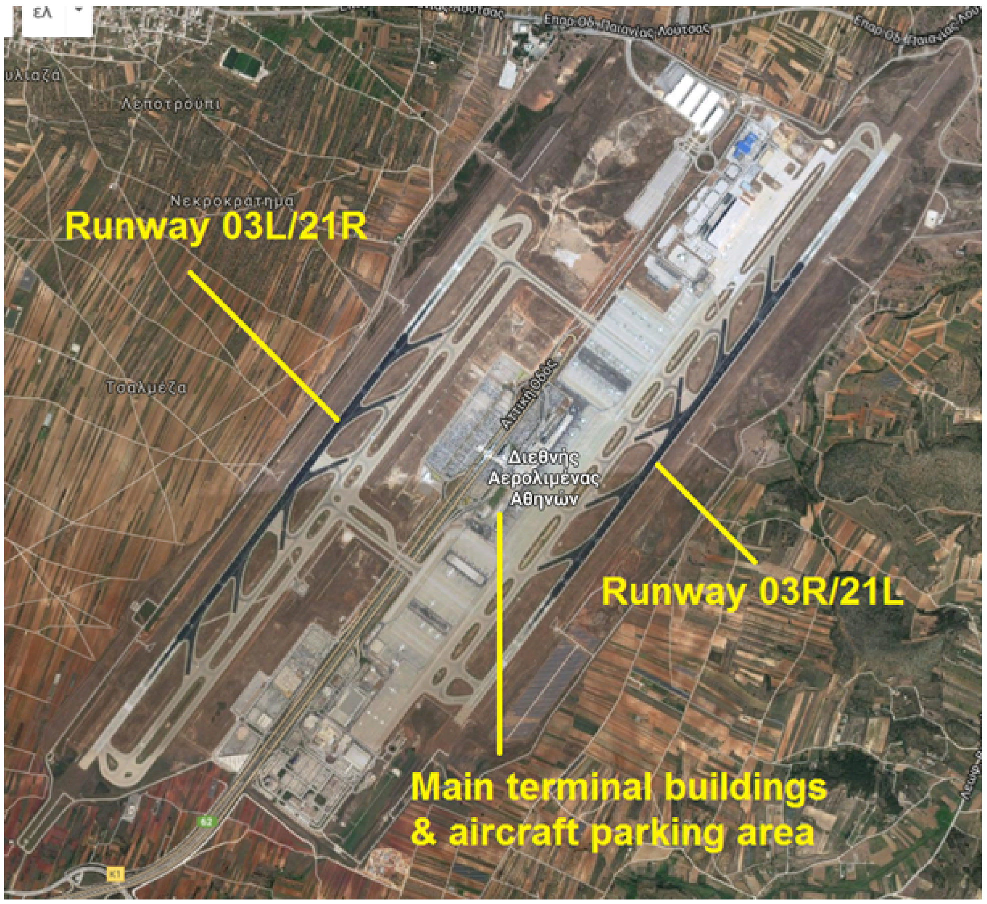


Figure 5.6: Athens International Airport from above.

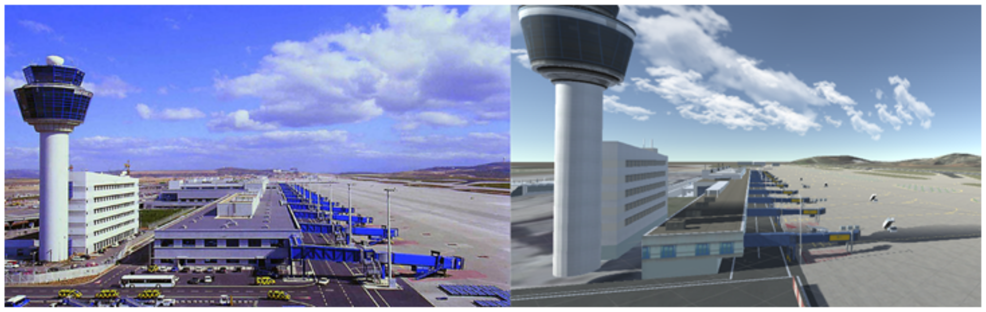


Figure 5.7: LEFT: real airport photo. RIGHT: airport render in Unity3D

(turbines). [16][17]

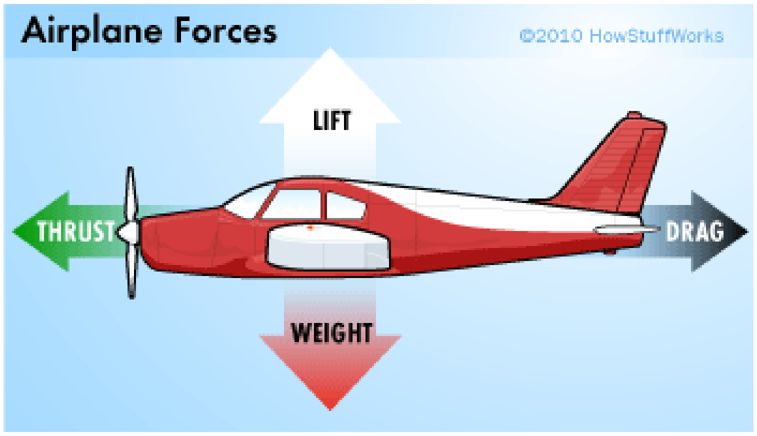


Figure 5.8: Airplane forces

All the forces (and moments) acting on an airplane are a result of pressure forces (normal to the airplane surfaces) and shear forces (along the airplane surfaces), both created by the airflow pattern over the airplane body. The moments (torques) are a result of the forces and, in addition to forces, are part of rigid body motion analysis. Airplanes can generally be treated as rigid bodies when analyzing the dynamics of their motion.

5.2.2 Linear forces

Lift

The lift force (L) "holding" a plane up is generated by airflow over the wings. This airflow is only possible if the plane moves relative to the air, hence lift is only possible if the plane moves relative to the air. The relative air speed must be large enough for sufficient lift to be generated.

In order to explain how lift force is produced, we have to take a brief look at the shape of a wing from an aerodynamics perspective. In aerodynamics, airplane wings are called airfoils. They have a cambered shape which enables them to produce lift, even for **angles of attack** (α) equal to zero.

The presence of air friction (viscosity) is what allows an airfoil to generate lift. The reasoning behind this is complicated and involves rather complex

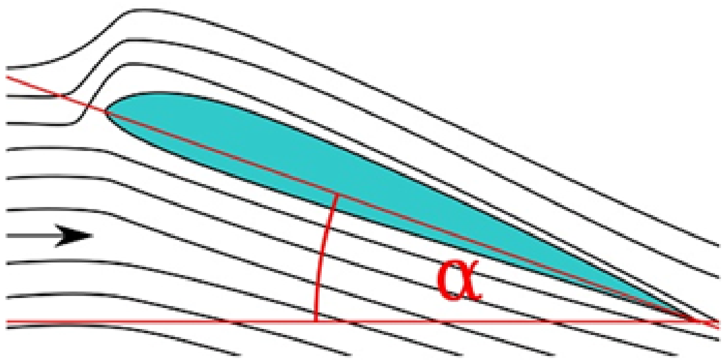


Figure 5.9: airflow over a wing (airfoil). Source: Wikipedia via Theresa knott.

mathematics. But basically, the air flow pattern around an airfoil results in the lower half of the airfoil experiencing greater pressure force than the top half of the airfoil. As a result, a lift force is generated. This is perhaps the least understood, and most asked, aspect of how airplanes fly.

The amount of lift produced is not only determined by an aircraft's velocity or thrust, but also by other factors. Lift depends on the air density, the square of the velocity, the air's viscosity and compressibility, the surface area over which the air flows, the shape of the body, and the body's inclination to the flow. In general, the dependence on body shape, inclination, air viscosity, and compressibility is very complex.

One way to deal with complex dependencies is to characterize the dependence by a single variable. For lift, this variable is called the **lift coefficient**, designated C_l This allows us to collect all the effects, simple and complex, into a single equation. The lift equation states that lift L is equal to the lift coefficient C_l times the density r times half of the velocity V squared times the wing area A. [18]

However, due to the fact that C_l mostly depends on the aircrafts current angle of attack (AoA or (α)), for the purpose of this thesis we can make a simplification and exclude other less significant parameters.

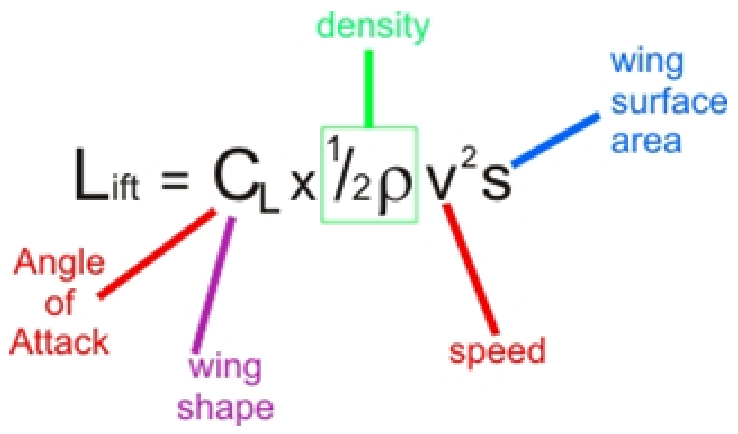


Figure 5.10: lift equation.

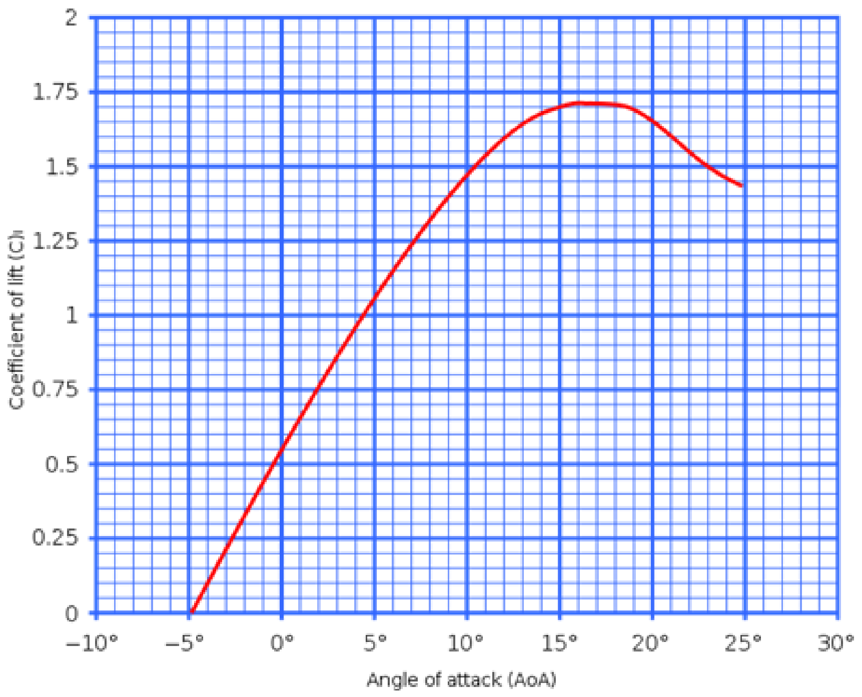


Figure 5.11: lift coefficient (Cl) vs. angle-of-attack ((α))

Weight

As one would expect, the weight (W) of the plane points straight down in the direction of gravity. Now, $W = mg$, where m is the mass of the plane and g is the acceleration due to gravity, where $g = 9.8m/s^2$.

Drag

The drag force (D) is opposite to the velocity (V) force, and is caused by air resistance generated on aircraft surfaces. Drag is a mechanical force, meaning that it is not generated by a force field (for example gravitational or electromagnetic field), but by the difference in velocity between the solid object (aircraft) and the fluid (air).
Drag can be manipulated to an extent by pilots, depending on the flight phase. For example , pilots may want to increase drag to slow down the plane during descent, by extending movable surfaces such as flaps and speed brakes.

Thrust

Thrust is a mechanical force which moves an aircraft through the air. Thrust serves two main purposes. Firstly, it is used to overcome the drag of an airplane. Secondly, as mentioned earlier, lift can only be generated when a plane moves relative to the air at sufficient speeds. Thrust is produced by accelerating the airflow in the rearward direction. This backwards acceleration of the airflow exerts a "push" force on the airplane in the opposite direction, by Newton's third law (For every action, there is an equal and opposite reaction), causing the airplane to move forward.

When the airplane is moving at constant velocity it is experiencing zero acceleration, and the forces must balance. This means that the lift force (L) generated by the airplane wings must equal the airplane weight (W), and the thrust force (T) generated by the airplane engines must equal the drag force (D) caused by air resistance. However, when for example the pilot wants to gain altitude (climb), lift must overcome the weight caused by gravity, thus $L > W$.

- $L = W$: aircraft is flying at steady altitude.
- $L > W$: aircraft climbs

- $L < W$: aircraft descents

It is important to mention that lift(L) is always perpendicular to the airplane body, or the velocity vector(V), to be more specific. This means that lift force is only opposite to the Weight force only when the airplane is flying wings-level (not turning).

5.2.3 Torque forces

Torque forces are responsible for maneuvering and navigation of the aircraft during flight. Pilots control the aircraft by adjusting physical elements on the outside of the airplane, elements which modify the airflow pattern around the plane, causing the plane to adjust its attitude and flight path. These physical elements are called control surfaces and consist of ailerons, elevators, rudders, spoilers, flaps, and slats. Adjusting a plane's flight path always involves either **pitching**, **rolling**, or **yawing**, or a combination of these. The figure below illustrates what these are.

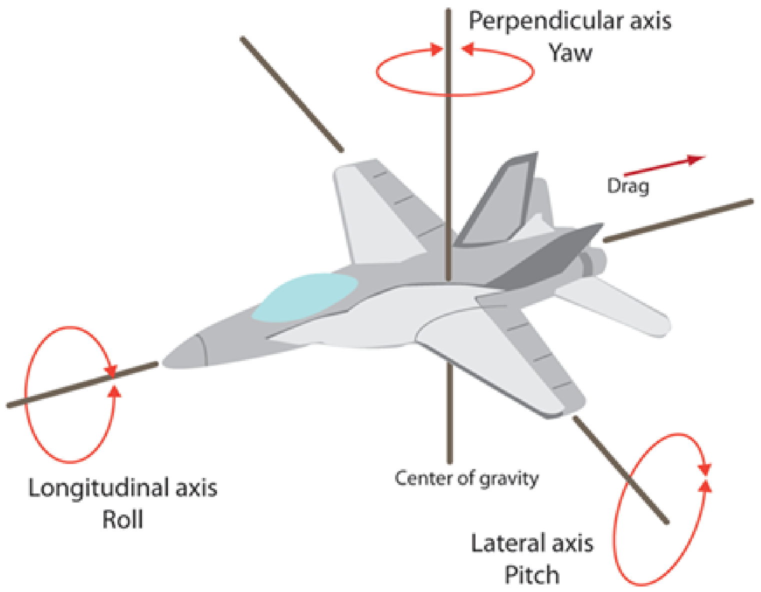


Figure 5.12: Rotation of an aircraft about the 3 principal axes.

- Roll: A roll is the rotation of the aircraft about the Unity Z axis, which causes the tilt to the left or right side. The angular displacement about this axis is called bank.The pilot changes bank angle by increasing the lift on one wing and decreasing it on the other. A positive roll angle lifts the left wing and lowers the right wing. The ailerons are the primary control of bank.

- Pitch: Pitch is the rotation of the aircraft about the Unity X axis, whichcauses the aircraft to nose-up (climb) or nose-down (descent). The primary control of pitch are the elevators, located in the tail.
- Yaw: Yaw is the rotation of the aircraft about the Unity Y axis, which causes the aircraft nose to move from side to side. The rudderis the primary control of yaw.

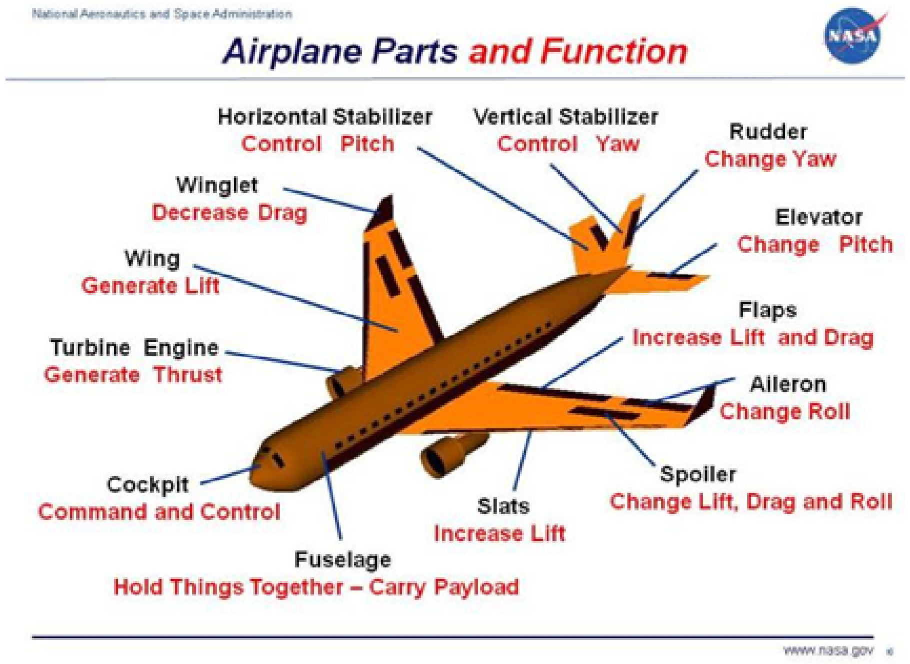


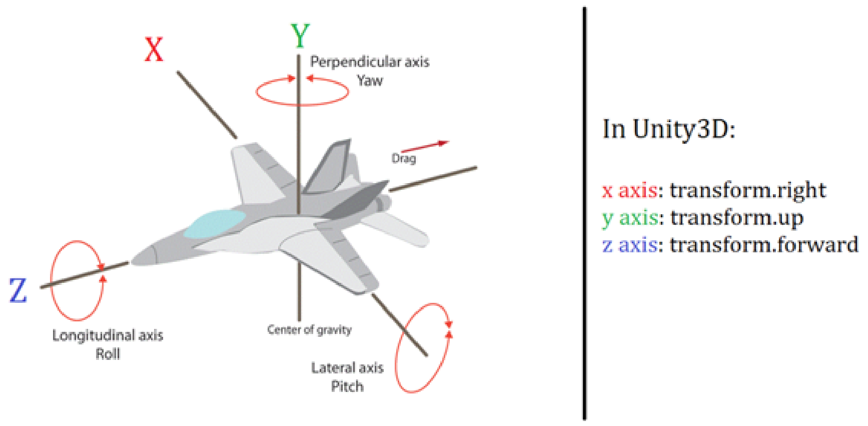
Figure 5.13: Aircraft parts and their purpose. Source: NASA

5.2.4 Simulating physics in Unity3D

Physics are implemented in our aerodynamics.cs script file.

Firstly, in order to enable physics simulation on any object in Unity, we need to attach the **Rigidbody** component to it.Adding a Rigidbody component to an object will put its motion under the control of Unity's physics engine. Even without adding any code, a Rigidbody object will be pulled downward by gravity and will react to collisions with incoming objects if the right Collider component is also present.

In this project, simulation and calculation of all forces acting on the airplane is provided by twodedicated functions, which update and re-calculate every parameter in every frame.



Our first function is **CalculateLinearForces()**. This function simulates every force that makes the plane move up/down or backward/forward. To add a linear force in Unity, we need a vector along which the force will act, and a variable to describe the specific amount of force we want to apply.

For example, adding a thrust force requires a vector corresponding to the aircraft's z axis , because thrust is supposed to make the plane go forward, multiplied by the desired force amount. So, if we want to add a thrust force equal to 10000 Newton, we do it as following:

```
airplaneRb.AddForce(10000 * transform.forward);
```

where airplaneRb is the object with the attached RigidBody component and transform.forward is a vector pointing forward (object z axis). Simulating the Lift force was a more complex procedure, because it depends on many parameters.

The **vector along lift will** act changes as the plane tilts and turns. However Lift is always perpendicular to the z axis (velocity/forward) and the x axis (vector pointing to the right). In other words, lift direction is equal to the cross product between the aircrafts z and x axis. (Given two linearly independent vectors a and b, the cross product, a b, is a vector that is perpendicular to both a and b)

As for the **lift amount**, we use the lift equation as shown previously , in combination with the lift coefficient, which is defined as shown in figure X. The lift amount is calculated in a separate function, CalculateLift().

The code that implements the above is:

```
liftDirection = Vector3.Cross(airplane.velocity, transform.right).normalized;
liftAmount = CalculateLift();
Airplane_rb.AddForce(liftAmount * liftDirection);
```

For the simulation of **drag** and **gravity**, we didnt write any extra code because Unity already implements both on every active RigidBody. Acceleration of gravity is set to 9.83 m/s by default. As for drag forces, Unity allows us to manipulate two parameters, drag and angular drag.

- Drag: the higher its value, the more an object resists to movement.
- Angular drag: the higher its value, the more an object resists to rotation.

The above values were experimentally set by testing which caused the plane to fly and behave realistically.

The second function is **CalculateTorque()**. This function simulates every force that makes the plane rotates about its 3 axes. This includes pitch, roll and yaw, which were previously described. To add a torque force in Unity, we need a vector representing the rotational axis, and a value/variable representing torque amount.
When an aircraft performs a **roll**, it rotates about its z axis, which is described in Unity by the transform.forward vector. During a **pitch**, the aircraft rotates about its X axis, which is described by the transform.right

vector. Finally, during a **yaw**, the aircraft rotates about its Y axis, which corresponds to the transform.up vector.
Concluding, code implementation of torques is similar to that of linear forces. For example, a torque making the plane climb or descend (pitch) can be applied as shown:

```
airplane_rb.AddTorque(PitchInput * transform.right);
```

In the above code example, *PitchInput* represents the amount of pitch and is controlled by the pilot. Positive value results in nose-up and negative value in nose-down. The higher the absolute value, the greater the torque , thus the more aggressive the change of pitch, and the faster the rate of climb/descent.
Yaw and Roll are implemented in exactly the same way, the only difference being the rotational axis.

Furthermore, when a plane needs to turn, the wings must roll to a banked position so that they are angled towards towards the desired direction of the turn. When the turn has been completed the aircraft must roll back to the wings-level position in order to resume straight flight. In the case of an aircraft making a turn, the horizontal component of lift acting on the aircraft causes **centripetal acceleration**, thus causing the aircraft to accelerate inward and execute the turn. [19]

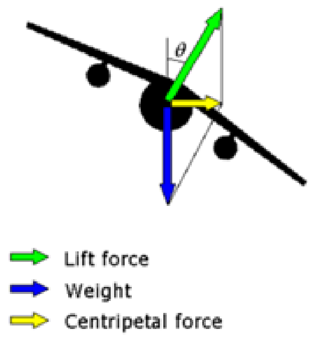


Figure 5.14: Centripetal force acting during a banked turn. Source: Wikipedia via Deeday-UK

By analyzing the forces and applying some simple mathematics, we can

produce a formula depicting the relation between the radius of the turn (R) , the velocity (v) and the angle of bank (ϑ).
While the aircraft performs a banked turn without changing altitude, the vertical component of lift balances out the weight, thus:

$$L \cos \vartheta = W = mg \tag{5.1}$$

$$L \sin \vartheta = mv^2/R \tag{5.2}$$

By dividing 5.1 and 5.2 we get:

$$R = \frac{v^2}{g \tan \vartheta} \tag{5.3}$$

This formula shows that the radius of turn is proportional to the square of the aircrafts true airspeed. With a higher airspeed the radius of turn is larger, and with a lower airspeed the radius is smaller.

This formula also shows that the radius of turn decreases with the angle of bank. With a higher angle of bank the radius of turn is smaller, and with a lower angle of bank the radius is greater.

5.3 Avionics

Avionics are the electronic systems used on aircraft, artificial satellites, and spacecraft. Avionic systems include communications, navigation, the display and management of multiple systems, and the hundreds of systems that are fitted to aircraft to perform individual functions.

5.3.1 PFD and ND

These monitors make the most important instruments of the airplane, and pilots consult them through all phases of a flight. The PFD (left monitor) provides current airspeed, altitude, heading, and an artificial horizon. It also displays other important information regardingflight status. The ND

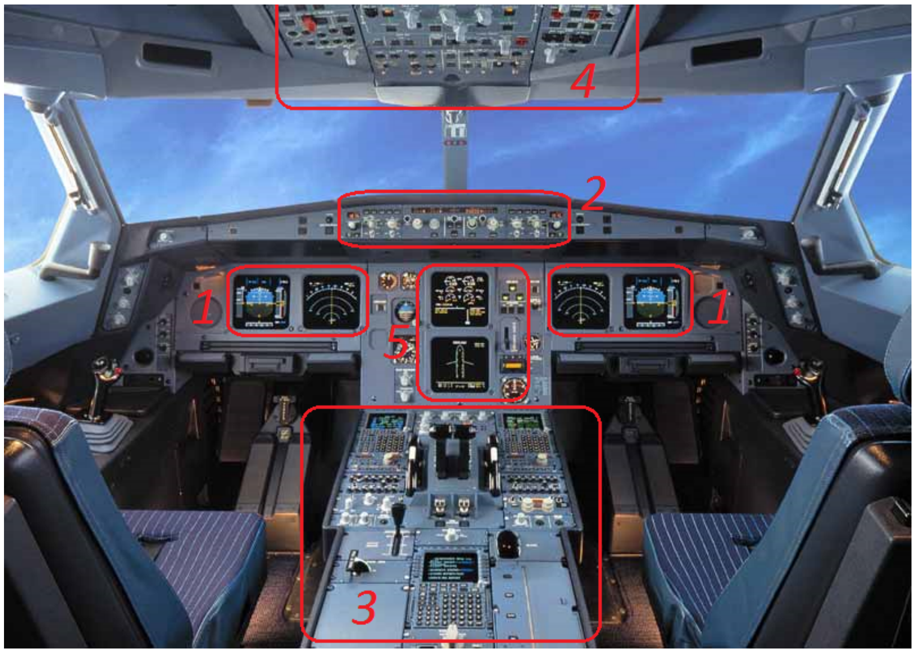


Figure 5.15: 1. PFD and ND 2. Autopilot 3. Pedestal 4. overhead panel 5. landing gear lever and EWD

(right monitor) , as its name suggests, provides information related to the navigation of the airplane. It shows current heading, speeds, planned route, nearby traffic, airports and waypoints.

5.3.2 Autopilot

An autopilot is a system used to control the trajectory of a vehicle without constant 'hands-on' control by a human operator being required. *Autopilots do not replace a human operator, but assist them in controlling the vehicle*, allowing them to focus on broader aspects of operation, such as monitoring the trajectory, weather and systems. Autopilots are used in aircraft, boats (known as self-steering gear), spacecraft, missiles, and others. Autopilots have evolved significantly over time, from early autopilots that merely held an attitude to modern autopilots capable of performing automated landings under the supervision of a pilot. Modern autopilots on complex aircraft use computer software. *The main function is to allow pilots to set a desired altitude, speed, and heading to be maintained.* Nowadays autopilots can perform more complex operations, such as follow a predefined flight path, maintain a steady rate of climb/descent or execute an entire landing procedure, without requiring any action from the pilots.

In our project, the Autopilot is handled by a separate script, autopilot.cs. The user can set a desired heading, speed and altitude.

We treat the autopilot as a **closed-loop system**. A Closed-loop Control System, also known as a feedback control system is a control system which has one or more feedback loops (hence its name) or paths between its output and its input. The reference to feedback, simply means that some portion of the output is returned back to the input to form part of the systems excitation. [21]
Closed-loop systems are designed to automatically achieve and maintain the desired output condition by comparing it with the actual condition. It does this by generating an error signal which is the difference between the output and the reference input. In other words, a closed-loop system is a fully automatic control system in which its control action being dependent on the output in some way.

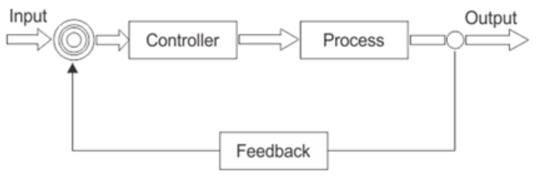


Figure 5.16: a typical closed-loop system

To explain how autopilot is implemented in our Unity project, we will give an example on how the aircraft maintains a specified altitude. In this case, **Input is the desired altitude** set by the pilot, and **Output is the current altitude**. If the aircraft flies in a different altitude, the comparison will produce an **error**. The bigger the difference between the actual and the target altitude, the bigger the error. The bigger the error, the greater the rate of climb or descent. As the plane tries to reach the target altitude, the error will get smaller and smaller and eventually become equal to zero. At that point , the aircraft has successfully reached the target altitude, thus the rate of climb/descent is set to zero. This functionality is illustrated in Figure 5.17 as a closed-loop system. In other words, the rate of climb/descent, depends on the size of the error (up to a limit)

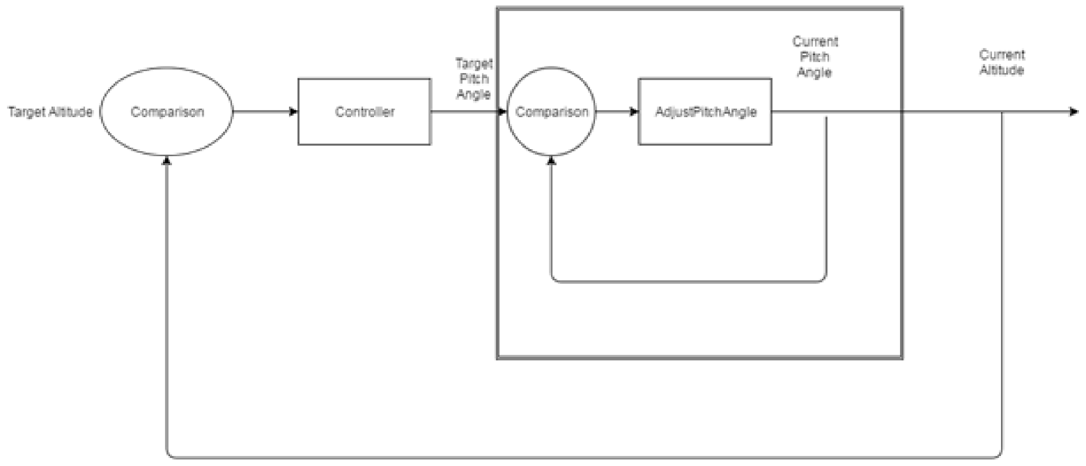


Figure 5.17: our closed-loop system that implements the Autopilot system.

In order to make the aircraft maintain the desired heading and altitude appearing on the LCD, **the user must activate the Autopilot** system by pressing the **Autopilot switch**, shown in figure 5.18. For **as long as autopilot is activated, the Autopilot is in full control of the aircraft**.

5.3.3 Pedestal

The pedestal is the group of instruments located in the middle. It provides access to fundamental controls of the aircraft, such as the **thrust lever** (to manually control the amount of thrust) **,engine switches** (to start or shut down engines) and other levers to set the **flaps**, **speed brake** and **parking brake**.

5.3.4 Overhead panel

The overhead panel is located in the center area just above the pilots. In reality the overhead panel contains many switches and buttons to control

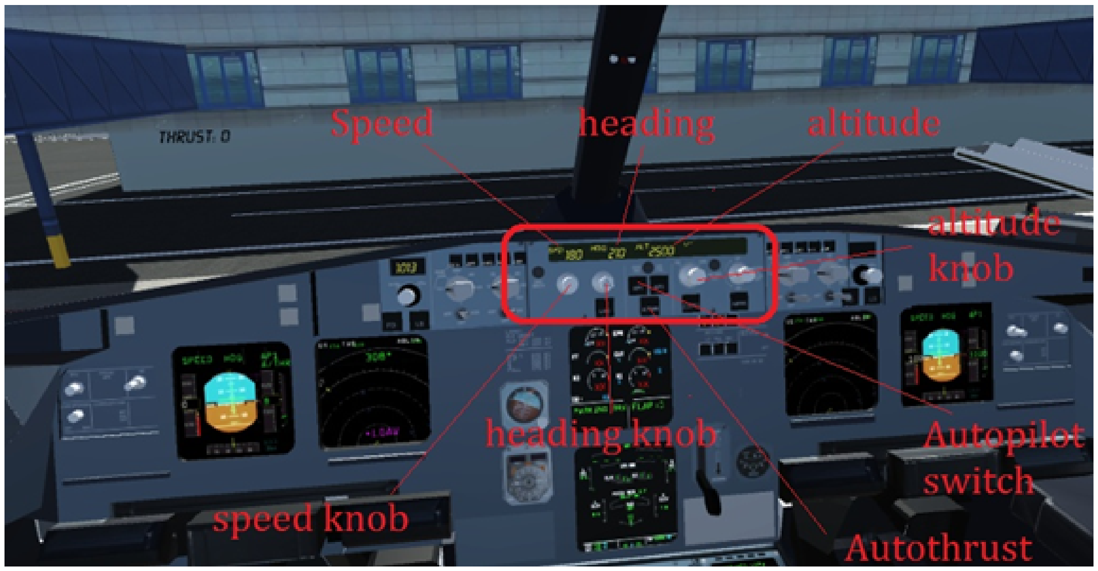


Figure 5.18: Autopilot: LCD displaying current target airspeed, heading and altitude, and rotary knobs

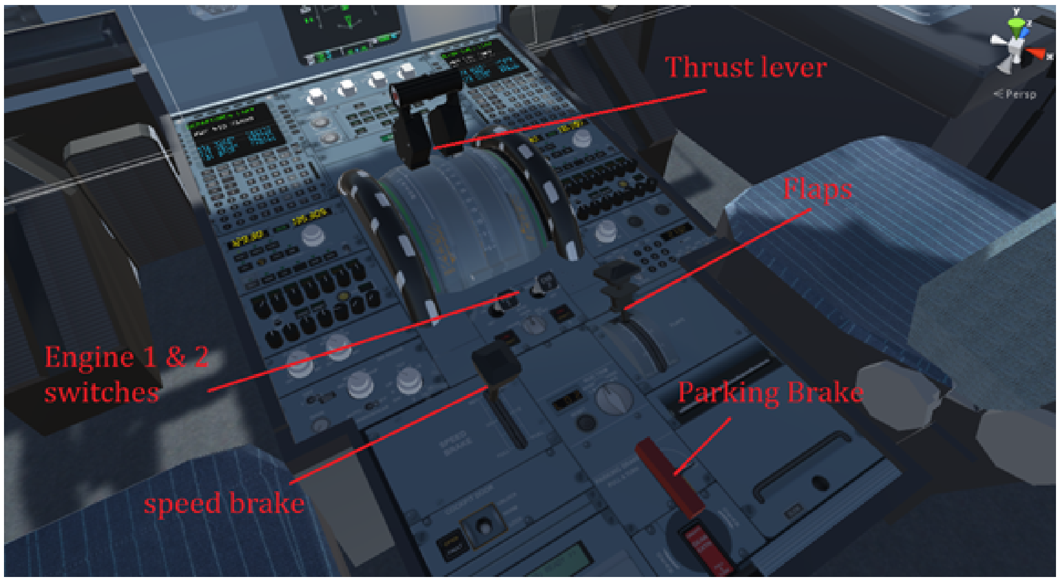


Figure 5.19: Pedestal.

aircraft systems, electronics, hydraulics and more. However for the purpose of this thesis, providing such functionality would make the simulation very complex, thus we only included some very basic functionality.

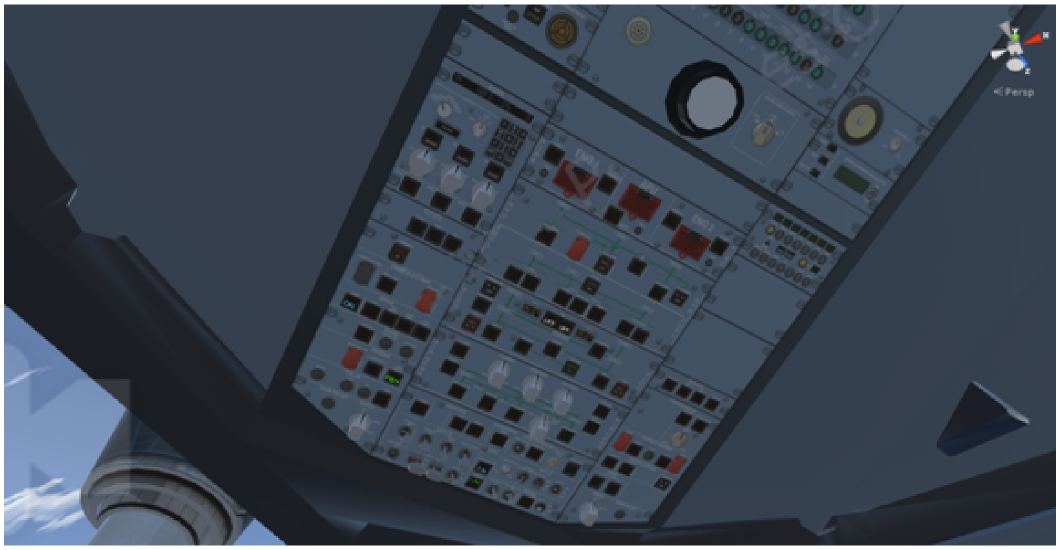


Figure 5.20: Overhead panel.

5.3.5 Implementing Avionics in Unity3D

The implementation of instrument displays in Unity was based upon the build-in UI(user interface)components. Unity UI features a variety of components such as **Canvas**, **Image**, **Text**, **Button** and more. In order for UI components to be rendered in a scene, they must be children of a GameObject with a Canvas component (recall that components that provide properties and functionality are attached to GameObjects). In this thesis we use more than 1 Canvas, containing UI text and raw image components, which work as cockpit displays and instruments [22]

Canvas

The Canvas is the area that all UI elements should be inside. The Canvas is a Game Object with a Canvas component on it, and all UI elements must be children of such a Canvas.

Creating a new UI element, such as an Image using the menu GameObject → UI → Image, automatically creates a Canvas, if there isnt already a Canvas in the scene. The UI element is created as a child to this Canvas. The Canvas area is shown as a rectangle in the Scene View. This makes it easy to position UI elements without needing to have the Game View visible at all times.
The Canvas can be set to different render modes to get different effects. For example Screen Space Overlay mode is used to place UI elements on top of the scene. If the screen is resized or changes resolution, the Canvas will automatically change size to match this. It is commonly used for menus.
The most frequently used render mode in this project is "World Space" In this render mode, the Canvas will behave as any other object in the scene. The size of the Canvas can be set manually, and UI elements will render in front of or behind other objects in the scene based on 3D placement. This is useful for UIs that are meant to be a part of the world. This is also known as a diegetic interface. Needless to say, all cockpit instruments and displays are in World Space render mode.

Creating the PFD and ND in Unity3D

Before stepping in to the creation process of these important instrument displays, which are **crucial for aircraft navigation**, lets explain their very basic functionality.

The **Primary Flight Display (PFD)**, consists of 4 basic elements. The first two are the **airspeed** and **altitude** indicators. The third element, is the **artificial horizon**.It is used in an aircraft to inform the pilot of the orientation of the aircraft relative to Earth's horizon. It indicates *pitch* (nose up/down) and *bank* (side to side tilt) and is a primary instrument for flight in instrument meteorological conditions. The fourth and final part of the PFD displays information such as if autopilot or autothrust are activated.

The **Navigation Display (ND)**'s main purpose is to display the aircraft's current **heading** in degrees, as well as its relative distance to airports and other aircraft, by using a 360 degree rotating circle , similar to a compass.

Now that we have briefly presented the basics of these important instru-

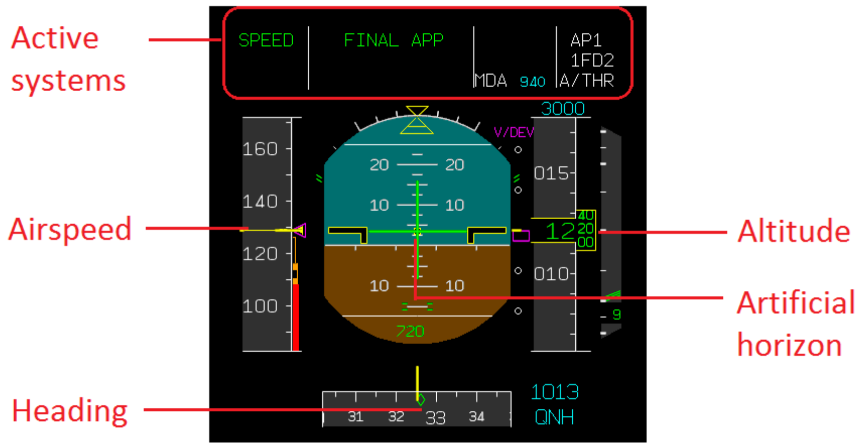


Figure 5.21: Primary Flight Display (PFD)

ment displays, we can proceed to the implementation process in Unity3D. First of all, these instruments take advantage of Unity's UI system. This means that we will need a **Canvas** which will render the UI elements. The Canvas needs to be set in World Space mode, because the instrument displays' position is fixed and independent of the camera position. Secondly, we need to think which parts will be static , and which will change, appear or disappear, move or rotate. Since we have decided, we can create the appropriate UI element, position it correctly and customize it. Note that changing or moving elements need to be scripted to do so.

As for the PFD, there is one static part, considered as the PFD background. Since it remains unchanged, we can import an image and attach it on a *UI* → *Raw Image* component. The image needs to be transparent, so that we can later place the artificial horizon image behind it. The next step is to create a **new material** that contains the transparent texture map and the emission map (it is a display so it has to emit some light) . The material's rendering mode should be changed from the default Opaque mode to Cutout mode, otherwise the transparency won't be rendered properly. After positioning and setting up the PFD's background, we can add the artificial horizon. The artificial horizon is basically an image that will rotate and move. It is positioned slightly behind the PFD background, and is visible through the transparent gap. We follow the same process as the PFD background. We create a new *UI* → *Raw Image* and set up the material.

Scripting the behaviour
Finally we have to script the artificial horizon so that it moves and rotates, in accordance to the aircraft. Hence, we need the script to retrieve the aircraft's current pitch and roll rotations, and rotate/move the artificial horizon's image accordingly. This process is executed every frame.

Initially, we need a reference to the aerodynamics script, because it is the script that will supply us with the aircraft's current pitch and tilt. Then we just apply these rotation values to the artificial horizon's transform. We use *transform.localRotation = Quaternion.Euler(xAngle, yAngle, zAngle)* to alter its rotation and *transform.localPosition = new Vector3(x, y, z)* to alter its position. We attach the script , and we have a fully functional artificial horizon.

For the rest of the displayed information, we use the *UI* $\rightarrow$ *Text* components. These components can display any text in a string format at a specific position in the 3d world. They are commonly used for on-screen information, text, words or numbers. They can display a pre-set string which can also be changed during run-time by scripts. These components were used in our project to display information such as current airspeed, altitude, heading , flap position and more.

The Navigation Display was created by following the same pattern , so there is no need for a separate section.

5.4 Audio

A game would be incomplete without some kind of audio, be it background music or sound effects. Unitys audio system is flexible and powerful. It can import most standard audio file formats and has sophisticated features for playing sounds in 3D space, optionally with effects like echo and filtering applied. Unity can also record audio from any available microphone on a users machine for use during gameplay or for storage and transmission.

Basic Theory
In real life, sounds are emitted by objects and heard by listeners. The way a sound is perceived depends on a number of factors. A listener can tell roughly which direction a sound is coming from and may also get some sense of its distance from its loudness and quality. A fast-moving sound source

(like a falling bomb or a passing police car) will change in pitch as it moves as a result of the Doppler Effect. Also, the surroundings will affect the way sound is reflected, so a voice inside a cave will have an echo but the same voice in the open air will not.

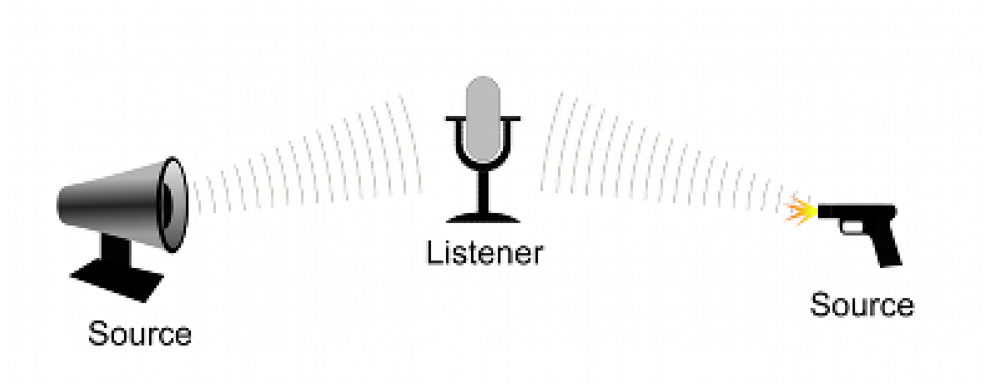


Figure 5.22: Audio Sources and Audio Listener

To simulate the effects of position, Unity requires sounds to originate from **Audio Sources** attached to objects. The sounds emitted are then picked up by an **Audio Listener** attached to another object, most often the main camera (user). Unity can then simulate the effects of a sources distance and position from the listener object and play them to the user accordingly. The relative speed of the source and listener objects can also be used to simulate the Doppler Effect for added realism.
Apart from Audio Source and Audio Listener, the next important object used to script audio, is **Audio Clip**. Audio Clips contain the audio data used by Audio Sources. An Audio Source object can be scripted to use more than one Audio Clip. However, an Audio Source can only play one Audio Clip at a time. For sounds that have to be played simultaneously, more Audio Sources are needed.

In this project it is common that many different sound effects are played at the same time. For example , the user can hear the engines running, the wheels make noise and the click of a button at the same time. Therefore, many Audio Sources are required. Even each engine has its own Audio Source. Concluding, these are the Audio Sources used:

- Left engine

- Right engine
- Cockpit sounds (clicks, switches, landing gear, flaps)
- External sounds (wheels)
- ATC (air traffic control)

Normally, each Audio Source has a script attached to it, which defines its behavior. The script defines the audio clips that will be used, which one will be played, when and how. Specifically, the programmer can write functions that can also be called externally , to play a desired audio clip and set its playback properties , like volume, pitch and other such as if the audio clip should play repeatedly (loop) or not.
For example, lets take a brief look at the script (*cockpitsounds.cs*) that handles cockpit sounds on how it works, supposing that it is already attached to an Audio Source game object:

The first lines define the Audio Source component and the Audio Clips to be used:

```
AudioSource sound;
public AudioClip AutoPilot;
public AudioClip CabinBell;
public AudioClip Click;
public AudioClip RotarySwitch;
```

Next, we use the GetComponent function to gain access to the Audio Source component:

```
void Start () {
        sound = GetComponent<AudioSource>();
}
```

Finally, we write the functions that will make the Audio Source play a specific Audio Clip, with specific parameters. Obviously, the functions are set as public, so that they can be called from other scripts:

```
public void playAutoPilotSound() {
        sound.clip = AutoPilot;
        sound.loop = false;
        sound.Play ();
}

public void playCabinBellSound() {
        sound.clip = CabinBell;
        sound.loop = false;
        sound.Play ();
}

public void playClickSound() {
        sound.clip = Click;
        sound.loop = false;
        sound.Play();
}
```

Additionally, we can accomplish more complicated functionality by scripting, such as setting audio clips play for a specific amount of time, or make two audio clips **crossfade**, which is useful when simulating engine sounds . Crossfade is the technique that makes a smooth transition between two audio files. It is achieved by fading out (slowly decreasing volume) one source file while fading in (slowly increasing volume) the other. This method creates a smooth transition because for a short period of time the listener hears both files playing simultaneously. This can be easily implemented by modifying an Audio Sources volume at run-time. Note that crossfading two clips requires manipulation of two Audio Sources.

5.5 Menu

This project features a menu, which appears on the screen on application start, or when the user pauses the simulation. It is a simple menu that provides some basic instructions and information and options. The menu was created by using Unity's UI elements (Canvas ,button, text) along with a script to execute the desired option.

The menu allows the user to start from two different positions by clicking the respective button. Clicking the "**Start at gate**" button, the simulation will start with the aircraft parked at an airport gate, and clicking the "**Start at runway**" button, the aircraft will be moved to the start of the runway for takeoff.

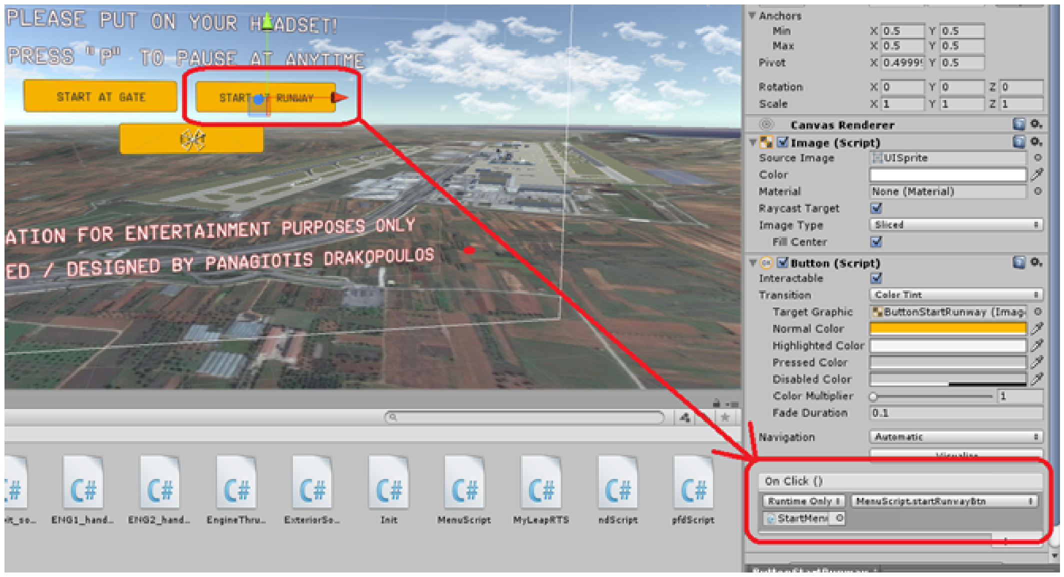
We want the menu to be on top of the screen, independent of any game object. Thus we set the canvas render mode to screen space-overlay. Then we add the text and buttons onto the canvas. Now we have created the visible part of the menu, but it isnt programmed to react to user input yet.

So the next step is to create a new script, supposing MenuScript.cs , to respond to user input and execute some operations. This script needs access to 3 different game objects and we will explain why. These objects are the camera, the aircraft and the cockpit. The reason is that when the user clicks on Start at gate or Start at runway, the following actions need to be performed:

- Hide the menu (because the user already clicked something)
  - Set the *canvas.enabled = false*; of the menu canvas
- Move the aircraft to the desired position (gate or runway)
  - Set *AircraftPrefab.transform.position*
  - Set *AircraftPrefab.transform.rotation*
- Make the camera a child of the cockpit gameobject.
  - *MainCamera.transform.parent = Aircraft.transform;*

The final step is to define which function of the above script will be executed when the **OnClick()** event of a button is triggered. Obviously, we need two different functions for two different buttons.

By now, we have a functional menu, however once the user starts the simulation, he can no longer pause or go back to the menu. To counter this, we allow the user to pause the simulation at any time, by pressing the P button. During a pause, the menu automatically appears on top of the screen. Needless to say, pressing the P button again will un-pause and the simulation will continue as normal.


PLEASE PUT ON YOUR HEADSET!
PRESS "P" TO PAUSE AT ANYTIME
START AT GATE
START AT RUNWAY
ATION FOR ENTERTAINMENT PURPOSES ONLY
ED / DESIGNED BY PANAGIOTIS DRAKOPOULOS
ENG1_hand…
ENG2_hand…
EngineThru…
ExteriorSo…
Init
MenuScript
MyLeapRTS
ndScript
pfdScript

```
▼ Anchors
    Min              X 0.5      Y 0.5
    Max              X 0.5      Y 0.5
  Pivot              X 0.4999   Y 0.5

  Rotation           X 0        Y 0        Z 0
  Scale              X 1        Y 1        Z 1
     Canvas Renderer
▼ ☑ Image (Script)
  Source Image       UISprite
  Color
  Material           None (Material)
  Raycast Target     ☑
  Image Type         Sliced
    Fill Center      ☑
▼ ☑ Button (Script)
  Interactable       ☑
  Transition         Color Tint
    Target Graphic   ButtonStartRunway (Image)
    Normal Color
    Highlighted Color
    Pressed Color
    Disabled Color
    Color Multiplier 1
    Fade Duration    0.1

  Navigation         Automatic

  On Click ()
  Runtime Only       MenuScript.startRunwayBtn
  StartMenu
```

Figure 5.23: Creating the Menu in Unity. The red arrow illustrates that when a button is clicked, the OnClick() event is triggered, which in turn calls a script function to do something. The canvas of the menu is also visible.

The main method to implement a **pause** in Unity3D, is to stop the passage of time. There is a specific variable for this , which is *Time.timeScale*. Setting this equal to zero will basically pause the game, if all functions are frame rate independent. T**his means that FixedUpdate() functions will not be called at all** when timescale is set to zero (Update() functions will be normally called) .This is how a pause is achieved.

5.6 High level project structure

Up to this point, we have seen that this Flight Simulator project consists of many different sections and functions, that work seamlessly producing the end user experience. For this to happen, scripts and game objects have to constantly communicate with each other and exchange data. Each script serves its own purpose and is dependent upon one or more game objects or scripts. For example, the aerodynamics.cs script simulates physics, Avionics.cs handles the aircrafts systems and instrument displays, TouchRecogn.cs monitors users hand position and gestures, and so on.

Below is a high level diagram of the most important scripts and game objects that shows how information is exchanged between them.

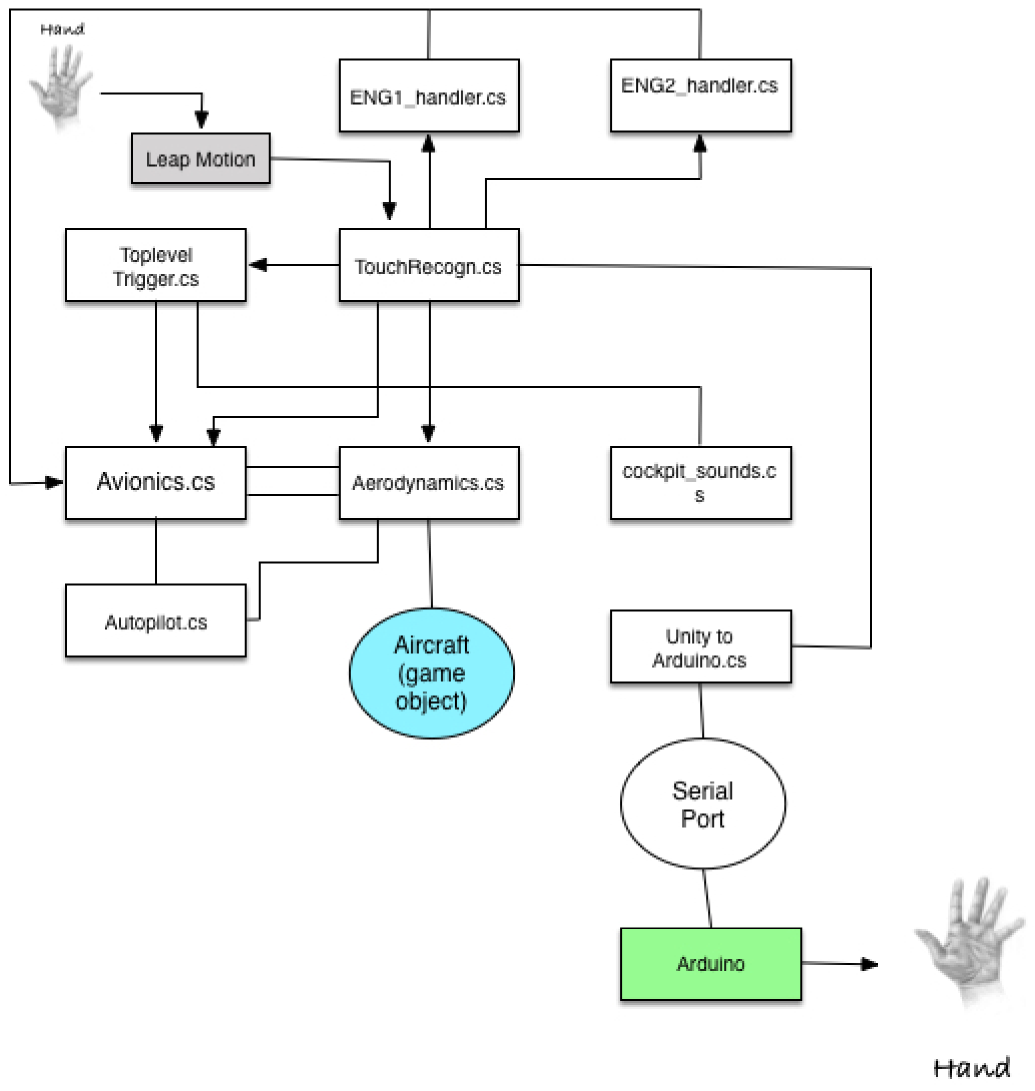


Figure 5.24: High level project diagram depicting the most important parts.

Lets see what happens when for example the user turns on the auto pilot, by touching the autopilot switch.

1. The TouchRecogn script recognizes that the auto pilot switch has been pressed.

2. TouchRecogn script instructs the UnityToArduino script to send vibration feedback to the user.

3. TouchRecogn script instructs the Avionics script that the auto pilot system is on, so it updates the instrument displays, and the auto pilot switch is lit , depicting that it is active.

4. It also activates the Autopilot script. The Autopilot.cs script is now in control of the aircraft, by communicating and instructing the Aerodynamics script how to fly the airplane.

5. The Autopilot script controls the trajectory of the aircraft by using the closed-loop system shown previously.

Summary

In this chapter we presented the entire process of utilizing real photographs, techniques , mathematics and physics in order to implement the fundamental elements of this Flight Simulator, from the ground up. We covered the making of the 3d environment, avionics, audio and aerodynamic physics. We also presented a high level project diagram that illustrates interaction and communication between the most important elements of this application.

Chapter 6

Interactivity

6.1 Introduction to Leap Motion's hand tracking

Leap Motion is a computer hardware sensor device that supports hand and finger motions as input, analogous to a mouse, but requires no hand contact or touching.

The Leap Motion controller is a small USB peripheral device which is designed to be placed on a physical desktop, facing upward. It can also be mounted onto a **virtual reality headset**. Using two monochromatic IR cameras and three infrared LEDs, thedevice observes a roughly hemispherical area, to a distance of about 1 meter. The cameras generate almost 200 frames per second of reflected data. This is then sent through a USB cable to the host computer, where it is analyzed by the Leap Motion software using "complex maths" in a way that has not been disclosed by the company, in some way synthesizing 3D position data by comparing the 2D frames generated by the two cameras. In a 2013 study, the overall average accuracy of the controller was shown to be 0.7 millimeters.[29] The smaller observation area and higher resolution of the device differentiates the product from the Kinect, which is more suitable for whole-body tracking in a space the size of a living room.

Leap Motion has significantly improved virtual reality experience overall by allowing users to manipulate and Dztouchdz virtual objects, something impossible before. As a result, the illusion of physical presence in virtual environments is stronger and more believable, thus making developers want

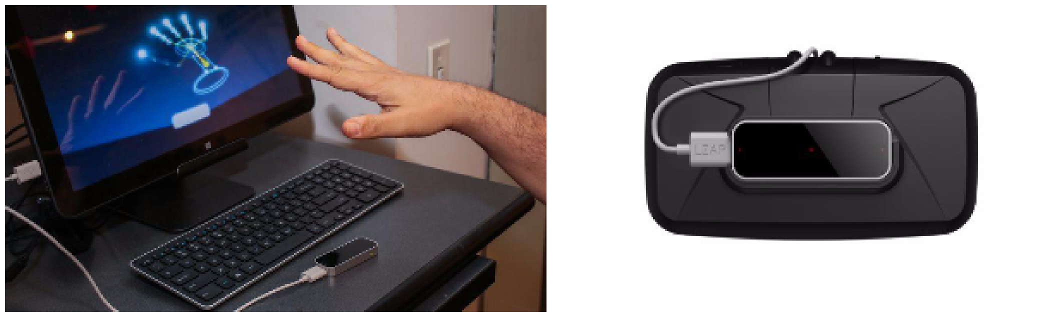


Figure 6.1: Leap Motion controller. **Left**: controller placed on desk facing upwards. **Right**: controller mounted on HMD facing forward.

to take advantage of this technology and integrate it into their applications.

In this project, interactivity is achieved by integrating the LeapMotions API (Unity Assets for Leap Motion) into Unity. The application is designed in such way that the controller must be mounted on the HMD to track hands properly.

6.2 Integrating Leap Motion in Unity

The first step code, is to download the *Unity Assets for LeapMotion* and import them to our Unity project. These assets allow Unity to communicate with Leap Motions software and hardware. They come with example scenes, prefabs and scripts for some basic functionality (such as gesture recognition). They also include the required libraries that allow scripts to access Leap Motions classes and functions.

The second step is to set the Leap Motion controller (HandController) as a child of the Main Camera object. The HandController object is a prefab that comes with Leap Motions unity assets. Camera is the camera object that is connecteddz with the virtual reality HMD. Thus when the user moves and looks around, his hands must follow his movement, giving the illusion that they are attached to the body as in real life. This is the reason why the controllers game object must be a child of the camera object.

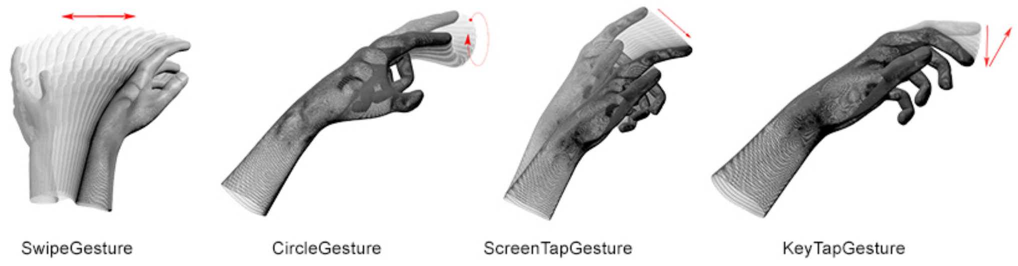
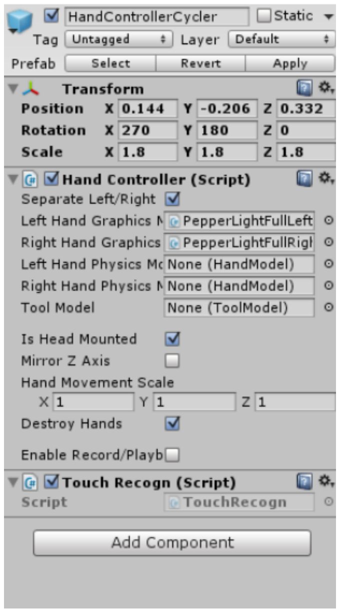


Figure 6.2: Leap Motion : recognized gestures



```
☑ HandControllerCycler          ☐ Static
Tag  Untagged      Layer  Default
Prefab  Select   Revert   Apply
▼ Transform
Position   X 0.144   Y -0.206   Z 0.332
Rotation   X 270     Y 180      Z 0
Scale      X 1.8     Y 1.8      Z 1.8
▼ ☑ Hand Controller (Script)
Separate Left/Right      ☑
Left Hand Graphics M     PepperLightFullLeft
Right Hand Graphics      PepperLightFullRigl
Left Hand Physics Mc     None (HandModel)
Right Hand Physics M     None (HandModel)
Tool Model               None (ToolModel)

Is Head Mounted          ☑
Mirror Z Axis            ☐
Hand Movement Scale
   X 1          Y 1          Z 1
Destroy Hands            ☑

Enable Record/Playb      ☐
▼ ☑ Touch Recogn (Script)
Script                   TouchRecogn
         Add Component
```

We can see that the *HandController* object has two scripts attached to it:

The **Hand Controller** script defines hands graphics and physics models to be used. Leap Motions assets come with "prefab" hand models, such as realistic human hands or robotic hands. Developers also need to specify whether the controller is mounted on an HMD or not for proper functionality.

The **Touch Recogn** script handles touch recognition, by constantly tracking the position of the fingertip, every frame. This script functions under a main principle: When the fingertip gets to close proximity to one of the cockpits touchable buttons and switches, the object is considered as being physically touched. The touch distance threshold is defined inside the script. It must be carefully set. A small threshold might make touching a button hard and require high precision, while a big threshold might accidentally trigger buttons and switches that were not meant to be touched. [23]

It is important to note that we had to implement a **debouncing** method for almost all buttons and switches. Debouncing is a technique that ensures that only one signal can be registered within the space of a given time (usually milliseconds), even when the hand is continuously in touch with an object.

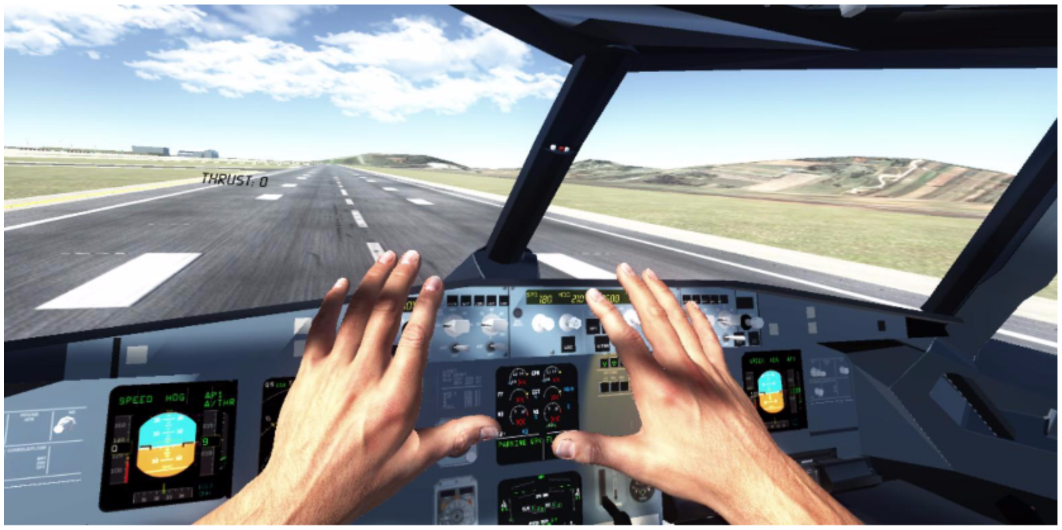


Figure 6.3: Leap Motion's realistic hand models in real-time gameplay

To give an example, without debouncing, it would be extremely hard to trigger a switch ON and OFF , because it would constantly trigger between the OFF and ON states for as long as the hand is touching it. To sum up, the application constantly measures the distance between the hands and each Dzclickable/touchabledz object inside the cockpit. When the distance to an object is smaller than a defined threshold, it is considered as physical contact.

6.3 Detecting touch and gestures in Unity3D

Lets see in more detail how the TouchRecogn script works - from hand movement tracking to converting finger coordinates from LeapMotion's coordinate system to our scene's system and how actions are triggered when a touch is detected.

The process that will be described in the following sections can be summarized as following:
Leap Motion tracks hands ⇒ captured data is stored in objects ⇒ needed variables are obtained ⇒ information converted to usable formats.

Firstly, we need to highlight that the method we use for touch detection, is based on measuring the finger's distance to every touchable object in the cockpit, every frame. If the measured distance to an object is lower than a specified threshold, the object is considered to be touched.

Consequently, the initial step is to load all the clickable/touchable objects in the cockpit and store them in a GameObject variable, so we can access their transform component. For example, we load the landing gear's GameObject by calling *GameObject.Find("LDG_GEAR_LEVER");*. During initialization, we also load external scripts, such as the *Avionics* and the *UnityToArduino* script. We need access to the Avionics script, because touching a button will trigger an event or a change in the aircraft's systems, and we also need access to the UnityToArduino script in order to enable vibration feedback upon touching an object.

Once done with initialization, we can proceed to the actual process of tracking hand movement, **fingertip position** to be more specific. However, before calculating distances from the fingertip to objects, we need to perform some necessary actions:

1. Every frame update, we need to retrieve the updated **frame** object. The **Frame** class represents a set of hand and finger tracking data detected in a single frame. The Leap Motion software **detects hands, fingers and tools within the tracking area**, reporting their **positions**, **orientations**, **gestures**, and motions in frames at the Leap Motion frame rate. We do this by calling: ***Frame*** *frame = controller.Frame();*

2. We need to utilize the retrieved **frame** object to gain access to information regarding the fingers. We store this information in a fingers object as such: ***FingerList*** *fingers = frame.Fingers;*
   The **FingerList** class represents a list of Finger objects.

3. Any coordinates, directions, and transformations reported by these classes are expressed *relative to the Leap Motion coordinate system*, not your Unity game world. To convert position vectors to Unity coordinates, we use the Vector class extension *ToUnityScaled()*. For example , in order to get the fingertips position relative to Unity's hierarchy, we

need to do: *Vector3 localTipPos = fingers[1].TipPosition.ToUnityScaled();* fingers[X] returns the finger object of the X finger. In this case, 1 represents the index finger.

4. Depending on the application and developer's approach, we can convert the above finger coordinates, from local space to **world space**. We do this by calling: *worldTipPos = transform.TransformPoint(localTipPos);* This step is only necessary when we need coordinates in respect to the world space, independent of any parent game object.

By now, we have the latest coordinates of a user's finger and all the coordinates of touchable objects. So all we have to do now is calculate distances to determine whether the fingertip is touching an object. For more readable code, we have separate functions that measure the distance to each object.

```
void Update() {

.......
Frame frame = controller.Frame();
FingerList fingers = frame.Fingers;
.......
Vector3 localTipPos = fingers[1].TipPosition.ToUnityScaled();
Vector3 worldTipPos = transform.TransformPoint(localTipPos);
.......
checkDistanceFromLDGGear();
checkDistanceFromThrottleLever();
checkDistanceFromFlapsLever();
checkDistanceFromSpeedBrakeLever();
checkDistanceFromEngineSelectors();
......
}
```

The above "checkDistanceFromXXX" functions are called in every frame, that's why they are placed in the Update() section of the script, together with the previous actions. These functions share the same structure, so we will take a look at one, for example the checkDistanceFromThrottleLever();

As its name suggests, this function determines whether or not the user is touching the throttle lever that controls the thrust of the engines. Lets analyze the procedure that is followed every single frame step by step:

1. The distance between the fingertip and the throttle is measured.
2. If the distance is lower than a specified threshold, the throttle is being touched, so:
   (a) A signal must be sent to Arduino to give vibration feedback to the user. We do this by setting a Boolean variable as true. (This variable is false when nothing is touched - no vibration feedback)

   (b) The throttles must follow the movement of the hand, while in touch. In Unity, the forward/back motion of objects is performed along its Z axis. Thus, the x and y coordinates remain always unchanged.

   (c) The new throttle Z coordinate must be equal to the fingertips z value. This way the user has the illusion that he is naturally moving the throttle with his hand.

   (d) To this point, the position of the throttle in the 3D space is only a set of x,y,z coordinates. We need to convert and normalize the coordinates, in order to get the current thrust amount in the 0%-100% range. So, supposing that the maximum Z value represents maximum thrust, and the minimum Z value represents idle thrust, we use the normalization formula:

   $$thrust\% = \frac{Z - Z_{min}}{Z_{max} - Z_{min}} * 100 \tag{6.1}$$

   (e) Concluding, we have successfully translated the throttle lever's current position to a usable thrust percentage in the 0-100 range, to set the current engine thrust. We can now send this value to the scripts that handle the aircraft systems.

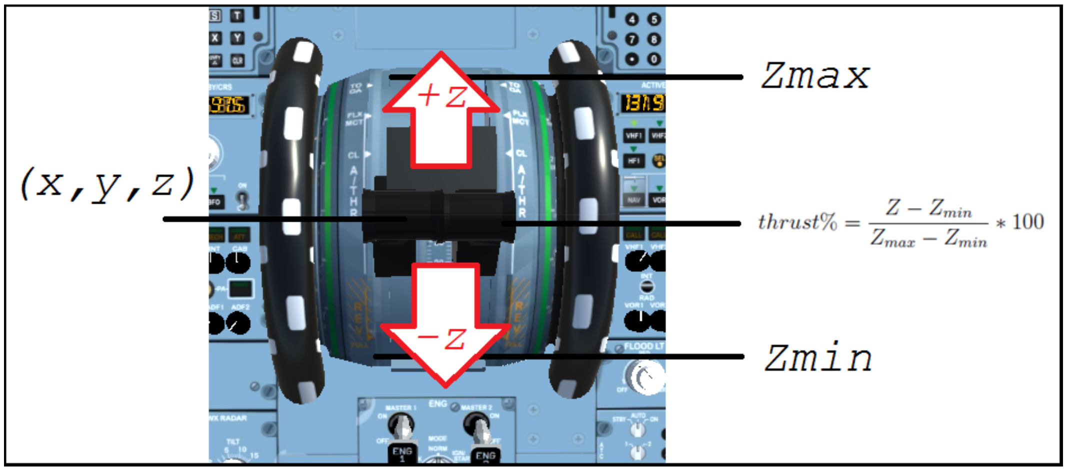
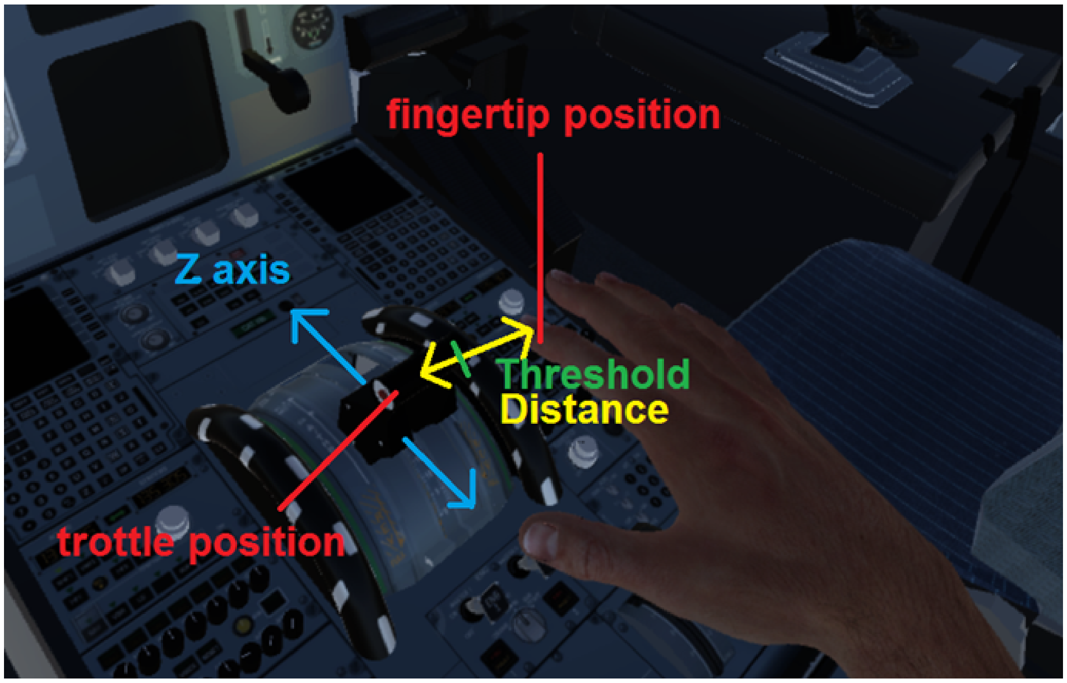


Figure 6.4: Converting throttle lever coordinates to thrust % amount.



6.4 Gestures

Recognizing gestures, such as making circles with a finger, requires a slightly different approach. The Leap Motion software recognizes certain movement patterns as **gestures** which could indicate a user intent or command. The Leap Motion software reports gestures observed in a **frame** the in the same way that it reports other motion tracking data like fingers and hands. For each gesture observed, the Leap Motion software adds a **Gesture object** to the frame. The gesture objects can be obtained from a **GesturesList**, same way as finger objects are obtained from a **FingerList**.

In this project, we took advantage of the **Circle** type of gesture. Circle gestures are continuous. Once the gesture starts, the Leap Motion software will update the progress until the gesture ends. A circle gesture ends when the circling finger or tool departs from the circle locus or moves too slow.

In order to convert a gesture into usable input for our application, we need to perform some actions:

1. Access the gestures objects stored in **GestureList** (if any).
2. Check what type of gesture it is. If it is **Circle** type, we proceed further, If not, the gesture is rejected.
3. We need to find out the whether the circle made was clockwise or counterclockwise. We do this by *(circle.Pointable.Direction.AngleTo(circle.Normal) <= Mathf.PI / 2)*
4. If the above condition is true, the circle is clockwise, else it's counter clockwise.

The circle gesture is recognized while rotating the index finger like in the image. We use clockwiseness to determine whether the value needs to increase or decrease.

Summary

In this chapter we thoroughly examined the process followed behind the scenes to accomplish interactivity by hand gestures. We learnt the basics of Leap Motion and how it is integrated into Unity. Then we presented how we utilize hand-tracking data stored in objects as usable input to trigger buttons, move levers to control the aircraft.

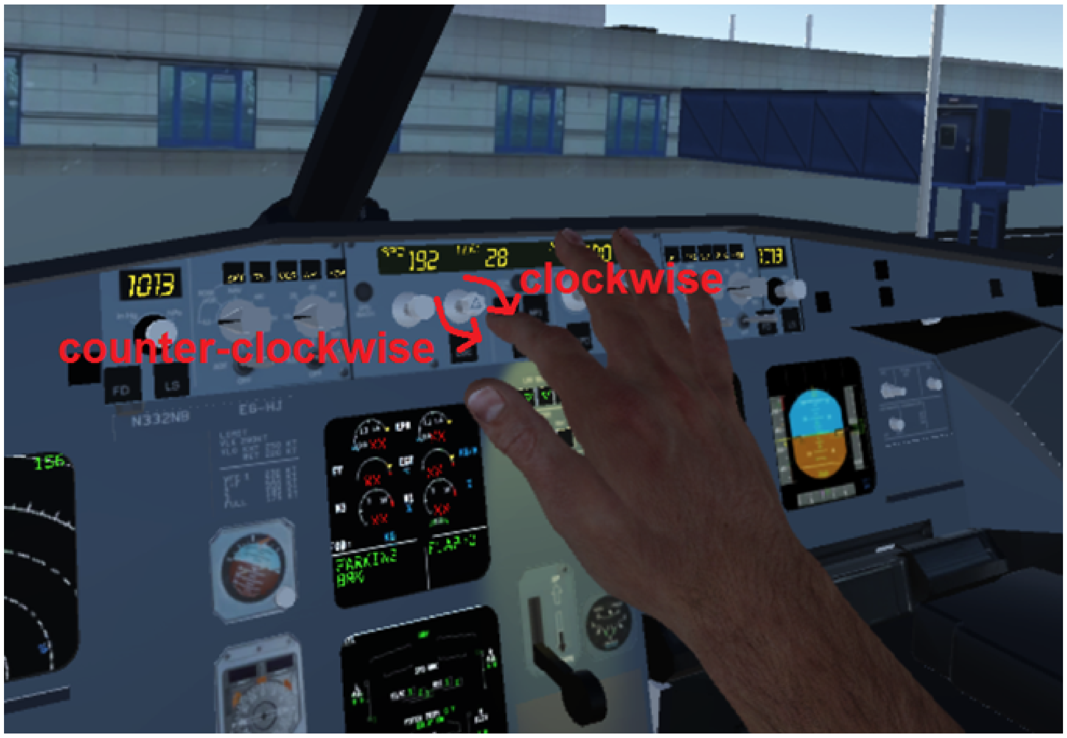


Figure 6.5: Making a circle with the index finger as shown in the image will be recognized as a gesture.

Chapter 7

Haptic feedback

7.1 Haptic feedback and why is it important

Haptic feedback, often referred to as simply "haptics", is the use of the sense of touch in a user interface design to provide information to an end user. When referring to mobile phones and similar devices, this generally means the use of vibrations from the device's vibration alarm to denote that a button has been pressed.[24]

The goal here is that a user gets some kind of haptic feedback, when his hand/finger is in contact with a switch or a lever. This is important, because apart from improving realism, it helps users understand whether their touch and gestures are recognized by the application or not. To give a better clue, hand-tracking technology and techniques are still imperfect and there is always a possibility of error. By providing instant haptic feedback to the user, he knows for sure whether his gestures were processed by the application.

We achieved a satisfactory result by using a vibration motor, connected to a digital pin in an Arduino microcontroller. When the application recognizes a gesture, the motor produces a vibration, simulating physical contact.

7.2 Introduction to Arduino

Arduino is an open-source project that created **microcontroller**-based kits for building digital devices and interactive objects that can **sense and con-**

trol physical devices. The project is based on microcontroller board designs, produced by several vendors, using various microcontrollers. These systems provide sets of digital and analog input/output (I/O) pins that can interface to various expansion boards (termed shields) and other circuits. The boards feature **serial communication** interfaces, including Universal Serial Bus (USB) on some models, for loading programs from personal computers. For programming the microcontrollers, the Arduino project provides an (IDE) based on a programming language named Processing, which also supports the languages C and C++. [25]

The first Arduino was introduced in 2005, aiming to provide a low cost, easy way for novices and professionals to create devices that interact with their environment using sensors and actuators. Common examples of such devices intended for beginner hobbyists include simple robots, thermostats, and motion detectors.
An Arduino's microcontroller is pre-programmed with a boot loader that simplifies uploading of programs to the on-chip flash memory, compared with other devices that typically need an external chip programmer. This makes using an Arduino more straightforward by allowing the use of an ordinary computer as the programmer.

7.3 Setting up and programming Arduino

The Arduino project provides an integrated development environment (IDE), which is a cross-platform application written in the programming language Java. Programmers can **compile the code and instantly upload it to the Arduinos hardware, usually by a serial connection over USB**. A typical Arduino script consist of two functions that are compiled and linked with a program stub main() into an executable cyclic executive program:

- **setup()**: a function that runs once at the start of a program and that can initialize settings.
- **loop():** a function called repeatedly until the board powers off.

In order to establish a communication between Arduino and a PC we need to write a simple sketch(script), compile it and upload it to our microcontroller, as explained previously. We basically want to program our Arduino to receive data from the PC (and Unity) over a serial connection, and

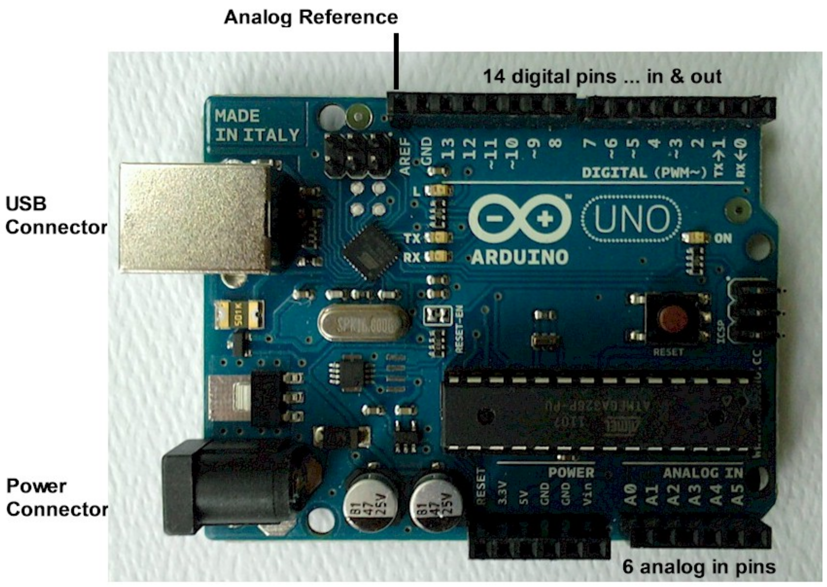


Figure 7.1: Arduino 'UNO' microcontroller

change its output according to the received input. We have two different inputs (characters) that Unity sends:

- 'y': Unity sends this character when a hand is in contact with a switch, thus the Arduino gives HIGH voltage to the vibration motor via the connected pin.
- 'n': Unity sends this character when a hand is not in contact with a button or a switch. Thus Arduino gives LOW voltage to the vibration motor, and the vibration stops.

```
int Vout = 10; // pin no. = 10
char myCol[2];

void setup()
{
        Serial.begin(9600);
        pinMode(Vout,OUTPUT);
        digitalWrite(Vout, LOW);
```

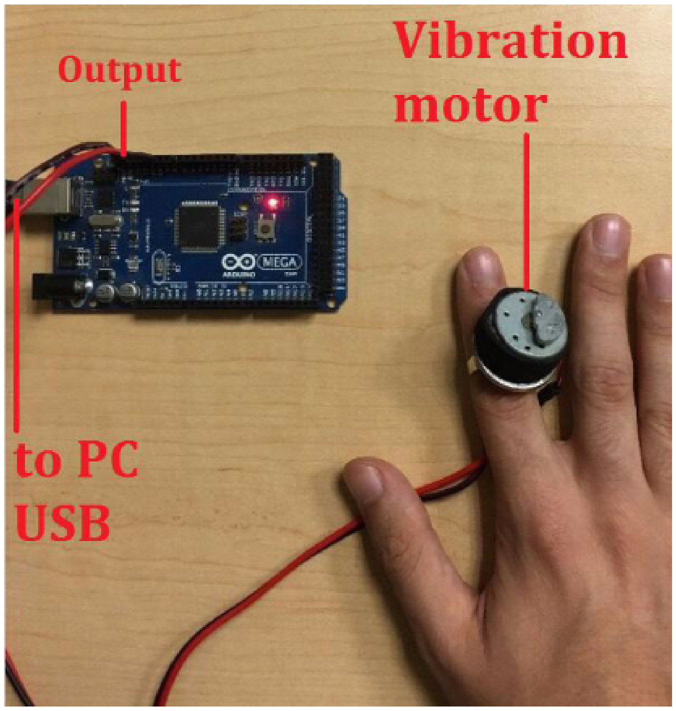


Figure 7.2: Vibration motor connected to Arduino.

```
}

void loop()
{
        int tc = 10;
        Serial.readBytesUntil(tc, myCol, 1);
        if (strcmp(myCol, "y") == 0) {
                digitalWrite(Vout, HIGH);
        }

        if (strcmp(myCol, "n") == 0) {
                digitalWrite(Vout, LOW);
        }
}
```

The above code is written in Arduino's IDE, compiled and uploaded to the micro controller.

- **Serial.begin**() : opens serial port and sets data rate to 9600 bps.
- **pinMode**(10, OUTPUT) : Configures the specified pin to behave either as an input or an output. In this case, as output, because we want Arduino to control the vibration motor.
- **digitalWrite**(): Write a HIGH or a LOW value to a digital pin. If the pin has been configured as an OUTPUT with pinMode(), its voltage will be set to the corresponding value: 5V (or 3.3V on 3.3V boards) for HIGH, 0V (ground) for LOW.
- **Serial.readBytesUntil**(tc, myCol, 1): reads characters from the serial buffer into an array. The function terminates if the terminator character is detected, the determined length has been read, or it times out.

Finally, we need to compare the character from the serial buffer to 'y' or 'n', by using the **strcmp**() function, which returns 0 when two strings are identical. If the input character equals to 'y', the vibration starts, because the pin is set to HIGH voltage. If the input character equals to 'n', the vibration must stop, thus the pin's voltage is set to LOW.

7.4 Unity to Arduino communication

The last step to achieve vibration feedback, is to program Unity to communicate and send data to Arduino. There is a dedicated script that handles Unity to Arduino communication, which is the UnityToArduino.cs script. The script executes the following processes:

1. Unity.IO.Ports library is imported to our script, to be able to open and use serial ports.
2. The serial ports parameters are defined, according to Arduino setup.
3. We open a new serial port, after checking that it isnt already open.

4. Send the appropriate byte to Arduino by Dzwritingdz a character to the serial port.
    - Writes 'y' if the users hand is touching a switch or a lever.
    - Writes 'n' if the users hand is not touching anything.

```
using UnityEngine;
using System.Collections;
using System.IO.Ports;
using System.Threading;

public class UnityToARDUINO : MonoBehaviour {
public static SerialPort sp = new SerialPort("COM5", 9600);
private int safety_time = 1; // 1 second input ingore to avoid ena
private bool ingnoreInputVib = false;  // Use this for initializati

void Start()      {
              OpenConnection();
}

 // Update is called once per frame
 void Update()
 {
 }


 public void OpenConnection()      {
 if (sp != null)          {
              if (sp.IsOpen)
              {
                      sp.Close();
                      print("Closing port because its already open
              }
              else
      {
              sp.Open();
              sp.ReadTimeout = 16;
              print("port opened!");
```

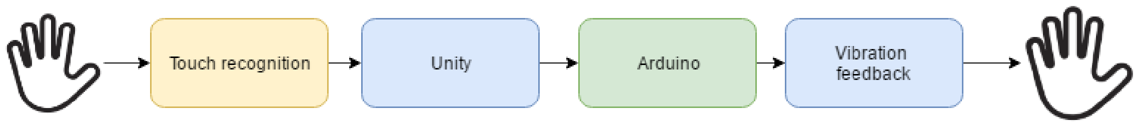
```
        }

     }
     else
     {
        if (sp.IsOpen)
                print("port already open");
        else print("port==null");
     }
    }


    public void ContinuousVibrate()
    {
        sp.Write("y");
    }

     public void stopVibration()
     {
        sp.Write("n");
     }
  }
```

The process we described in the above sections can be summarized by the process diagram below:



Chapter 8

Result, Evaluation and Future Work

8.1 Result and System Evaluation

From the first development phases, the application was constantly tested by users, and necessary changes were immediately made. The process of continuous assessment and refinement since early development resulted in a polished environment and straightforward gameplay with a few defects and flaws.

The development versions that were put to the test, gained mostly positive responses from the testing team of users, consisting of about **20 people**. The graphics felt significantly convincing. They stated that they were immediately submerged into the environment, believing that they are in a real cockpit, in a real airport. Physics and flying mechanisms gave the sense of flying to users on their seat, but without making them feel motion-sick, which is extremely important to us. Users also enjoyed interaction with the cockpit by their own hands.

As for the negative comments, users with no previous experience on VR environments, reported lack of instructions as well as difficulty controlling the aircraft and performing simple actions. However this is partially due to the nature of simulations, which requires some basic knowledge of the objective. As for the hand-tracking, users initially had trouble interacting with the cockpit , as for example they found themselves touching buttons and switches not intended to, while others felt that the system did not respond to their gestures. Fortunately these problems only lasted from seconds to a few minutes as users started to get used to the experience.

In our effort to make further improvements, we realised that most negative comments had to do with some **difficulty interacting** with the aircraft. As a result, some elements in the cockpit were resized or repositioned. Furthermore, the touch distance threshold that designates how far or close to objects the hand must be to be triggered was re-configured. Finally, some switches or levers are disabled during some stages of the flight, to prevent undesired behaviour by unintended hand motions. For example, users cannot retract the landing gear whilst on the ground or move the throttle lever while the autopilot is engaged.

Upon re-evaluation with the above changes implemented , with the same users, the overall experience was improved , with fewer negative comments.

Concluding, although there is always margin for improvement, a perfect result is impossible to achieve at this very moment. This is **mainly due to two reasons**. The **first** one is that hand-tracking sensors are not 100% accurate. Thus, simulating hand movements that require high precision is a challenge. The **second** reason, is Virtual Reality itself. Despite the improvements over the last years, there are still visual issues to be solved, such as image quality and sense of scale and distance.

8.2 Conclusion and Future Work

It seems that Virtual Reality is on a good track in getting into our lives, not only for entertainment , but also for other purposes, such as education, medicine and simulations. Computer graphics, hardware and current technology in general has finally reached the point where virtual environments feel more alive than ever before. Virtual reality is becoming more convincing, as hand-tracking or body-tracking is advancing along with HMD technology. Users can now be part of realistic and highly-detailed 3D environments and interact with them in many ways. The good news is that there is still a lot of room for improvement, regarding both hardware and software.

The **main technical challenge** of this work was the implementation of a realistic and detailed 3D environment along with simulated physics and extensive interactivity, without compromising:

- Usability
- Performance - even on older generation hardware
- Stability
- User experience (dizziness, nausea, motion sickness...)

However, this project can be extended and improved in many of its aspects. Graphics can be enriched. There could be more airports, larger and higher resolution terrain, realistic sea with real-time reflections and more. Interactivity can be also vastly improved - more buttons and switches and more accurate response to users gestures. The vibration feedback implemented, provides user feedback by giving a sense of touch through a light vibration, still feels incomplete and inferior to the real sense of touch and remains an area of future research. Finally, a mini-tutorial can be implemented to help novice users perform basic actions during flight.

Finally, in a more mature state of this project, **A.I. (Artificial Intelligence)** could be possibly utilized and integrated into certain functions. For example, A.I. airplanes that take-off , fly and land, A.I. Air Traffic Controllers giving instructions to pilots, or even some kind of pilot behaviour/reaction monitoring system.

References

1. http://www.wsj.com/articles/what-does-virtual-reality-do-to-your-body-and-mind-1451858778
2. https://en.wikipedia.org/wiki/Virtual_reality
3. http://gizmodo.com/this-is-how-valve-s-amazing-lighthouse-tracking-technol-1705356768
4. http://blog.natebeatty.com/2016/03/13/lighthouse-deconstructed/
5. https://imotions.com/blog/top-8-applications-eye-tracking-research/
6. http://blog.leapmotion.com/hardware-to-software-how-does-the-leap-motion-controller-work/
7. http://www.vrs.org.uk/virtual-reality-gear/head-mounted-displays/
8. http://www.tomshardware.co.uk/vive-rift-playstation-vr-comparison,review-33556-3.html
9. http://uploadvr.com/vr-hmd-specs/
10. https://en.wikipedia.org/wiki/Stereoscopy
11. http://arstechnica.com/gaming/2015/08/uncross-those-eyes-researchers-solve-vrs-depth-of-focus-headaches/
12. https://en.wikipedia.org/wiki/Game_engine
13. https://www.unrealengine.com/unreal-engine-4
14. https://www.cryengine.com/
15. https://aws.amazon.com/lumberyard/faq/
16. https://www.grc.nasa.gov/www/k-12/airplane/thrust1.html
17. http://www.real-world-physics-problems.com/how-airplanes-fly.html
18. https://en.wikipedia.org/wiki/Lift_coefficient
19. https://en.wikipedia.org/wiki/Banked_turn

20. https://www.grc.nasa.gov/www/k-12/airplane/

21. http://www.electronics-tutorials.ws/systems/closed-loop-system.html

22. https://docs.unity3d.com/Manual/UICanvas.html

23. https://developer.leapmotion.com/documentation/v2/csharp/devguide/Leap_Overview.html

24. http://www.mobileburn.com/definition.jsp?term=haptic+feedback

25. https://www.arduino.cc/