Implementation of a virtual reality design review application using vision-based gesture recognition technology

A Master's Thesis

Andreas Oven Aalsaunet

Thesis submitted for the degree of
Master in Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2017

Implementation of a virtual reality design review application using vision-based gesture recognition technology

A Master's Thesis

Andreas Oven Aalsaunet

Implementation of a virtual reality design review application using vision-based gesture recognition technology

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

The field of virtual reality (VR) technology has seen an exciting development in recent years, with the release of the first commercial virtual reality headsets, such as Oculus Rift CV1 and HTC Vive, taking place in 2016. The application area for these virtual reality headset have exceeded the expectations of many, with virtual reality technology being present in domains ranging from entertainment to educational training.

Despite this early success, there are still a lot challenges associated with virtual reality technology. This thesis will discuss several of these challenges, with especial attention to human-computer interaction, and more specifically to vision-based gesture recognition as an input method used in combination with virtual reality technology. This thesis is also interested in virtual reality's applicability to business and engineering, and will also review an associated implementation of a virtual reality-based design review application made for the company DNV GL. In addition to discussing the design and implementation details of this implementation, the thesis will also summarize findings made during user tests of the application.

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The field of virtual reality technology has seen an exciting development in recent years, with the release of the first commercially successful virtual reality headsets, such as the Oculus Rift CV1 and HTC Vive, taking place in 2016. The application area for these virtual reality headset have exceeded the expectations of many, with virtual reality technology already being present in several different domains, ranging from engineering to entertainment (Leadem, 2016). Leadem (2016), among others, reports numerous domains where virtual reality is successful being used, including healthcare (e.g in surgery), military, architecture/construction, art, fashion, entertainment (games, films etc), education, business, telecommunications, sports and rehabilitation.

With the success of virtual reality technology, and increased attention towards technologies that complement it, several businesses and institutions are interested in making use of the new possibilities virtual reality offer (Lubell, 2016). One major business area for virtual reality, besides entertainment, has been in the architecture, engineering and construction fields. Iris VR, a New York-based technology company building virtual reality applications, has reported that among their 15 000 customers, 75% are from these industry segments. One possible reason for the impact virtual reality has had on these fields is their dependence on big and complex 3D models. As virtual reality head mounted devices (HMD) are stereoscopic, i.e provide separate images for each eye, they are able to deliver a feeling of depth and scale that is unrivaled by regular two dimensional displays (Kuchera, 2016). This point was highlighted in an interview of a senior designer at an architect firm, conducted by Lubell (2016). He stated that "Practically nobody can understand architectural drawings, and even 3D visualizations are a stretch for most. But everybody gets VR instinctively. You can get to the point very quickly. It either sells or kills the project right away."

Virtual reality applications that allow its users to inspect models in 3D also has many additional possibilities. One example is to be able to work (e.g. to edit, annotate or comment) on the model while being "inside" it (e.g. when wearing a virtual reality HMD) and to use the virtual reality

application as a design and collaboration tool to exchange ideas about the model. DNV GL, the world's largest maritime classification society, is looking into exactly this, and view this as a potential big improvement over their current "paper-based" work flow. More specifically, DNV GL is interested in a virtual reality application for design reviews, a classification process where DNV GL employees review clients' design models and comments on various aspects of the models that need to be improved to meet the classification requirements.

This thesis will address this vision and utilize several state-of-the-art frameworks and technologies to design and implement such an application. This presents several challenges, some of which will be mentioned in the next section and reviewed more throughout the thesis, while DNV GL, their work flow, visions and motivations will be discussed further in chapter 2.

1.2 The Challenges of Virtual Reality

Despite the early success the field of virtual reality technology has seen, there are still a lot challenges associated with it. These challenges include prevention of virtual reality sickness (a kind of induced motion sickness), strict performance demands on target hardware and having more suitable input methods when using virtual reality HMDs. Addressing these challenges, in both design and implementation, is an important step when building virtual reality software (Dean Beeler and Pedriana, 2016), and each of these challenges, among others, will be discussed more in the following chapters.

As virtual reality technology enables users to experience virtual worlds in a new way, human-computer interaction (HCI) is also a highly relevant topic. This field has in many ways seen a resurgence as virtual technology gives new possibilities, but also set new constrains. One of these constraints is limiting the user's field of vision exclusively to that projected by the lenses, which may make interaction with traditional input devices, such as mouse and keyboard, more challenging. Because of this, alternate methods of interacting with the computer is a relevant topic. One of these methods is the use of gestures, which have long been considered an interaction technique that can potentially deliver more natural, creative and intuitive methods for interacting with computers (Rautaray and Agrawal, 2015). To enable the use of gestures as a viable input method to a computer, responsive and reliable gesture recognition techniques are needed.

1.3 Problem definition

This thesis will evaluate the consequences of utilizing virtual reality technology in combination with vision based gesture recognition technology, and discuss the benefits it might bring, as well and the challenges it presents. The thesis will also review the design and implementation of a design review application, which is developed as part of this thesis with the

aforementioned goal in mind. The design review application is also a prototype developed for the major international classification company DNV GL to evaluate how the use of virtual reality and gesture recognition technology might benefit their design review process. As such, the application requirements has been created in cooperation with DNV GL and represents common 3D object manipulating and navigation tasks. After discussing the design and implementation choices of this application, the user evaluation session will be discussed. The user evaluation sessions were performed in cooperation with DNV GL employees, the potential end users, and contained invaluable feedback relevant to the use of virtual reality and gesture recognition technology in a professional setting.

1.4 Limitations

The initial list of application features had to be shortened significantly to focus more on the most relevant parts for this thesis. As such the design review application is more a prototype or proof-of-concept than a finished product. Section 5.2.1 outlines the application features and will explain more of what's included in the application and what isn't.

1.5 Outline

This thesis is organized as follows: In chapter 2 DNV GL and their business domains and processes will be introduced, together with a general discussion regarding the role of classification societies. In this chapter we also define some of the scope for the application, a topic which will be revisited in chapter 5. In chapter 3 we will review the history, concepts and demands of virtual reality, as well as discuss the issue of virtual reality sickness and other challenges. The implication these challenges have for the design and implementation stages, which are covered in chapter 5 and chapter 7, are also discussed. Chapter 4 reviews gesture recognition technology and its exciting possibilities for virtual reality. This chapter also discusses the different technologies that makes up the field of gesture recognition technology, and how it functions. In chapter 5 we will review the design of the design review application, its various use cases and function requirements, how gesture recognition technologies can be used, and what frameworks and technologies are utilized. Chapter 6 will review the software development libraries, APIs and frameworks which were outlined in chapter 5 and is utilized in the implementation. Special emphasis will be put important concepts of the Unity Engine, its programming model and on the Leap Motion library. In chapter 7 we will document how the application is implemented and expand upon some of the Unity and Leap Motion concepts that were used. In chapter 8 the user evaluation sessions will be covered, and the responses discussed and analyzed. Chapter 9 will conclude this thesis with a summary of the findings and some thoughts about future work.

Chapter 2

Classification Societies in a Modern World

2.1 The Roles of Classification Societies

A classification society provides classification, statutory services and assistance to the maritime industry based on its accumulated knowledge of fields like maritime, engineering, construction and technology (Hormann, 2006). The International Association of Classification Societies (IACS) defines a classification society as an organization which publishes its own classification rules and technical requirements in relation to the design, construction or survey of ships. The organization should have capacity to apply, maintain and update these rules and requirements with own resources on a regular basis, and should be impartial, meaning that it should not be controller by ship-owners, shipbuilders or be otherwise commercially engaged in the manufacture, equipping, repair or operation of ships. In addition, the classification society should verify compliance with these rules and requirements during construction and periodically during a classed ship's service life.

Classification societies dates back to the second half of the 18th century, where marine insurers developed a system for independent technical assessment of the ships presented to them for insurance cover. These insurers were based out of Lloyd's Coffee House, a popular establishment for sailors, merchants and shipowners, and led to the establishment of the insurance market Lloyd's of London, Lloyd's Register and several related shipping and insurance businesses (Marcus, 1975). This also led to a committee being formed in London in 1760 being the first recorded classification society committee (Hormann, 2006). At this time, various aspect of a ship, such as its hull and equipment, were assigned "grades" for their condition, ranging between G, M and B (good, middling or bad), or simply 1, 2 and 3. As the classification profession evolved, these different classifications were mostly replaced by a more discrete classification system, meaning that a ship either meets the relevant class society's rules or it does not.

Classification serves as a certification process where the candidate has

to fulfill a number of requirements in order to "pass" the classification process. Hormann (2006) describes the objective of ship classification as verifying "the structural strength and integrity of essential parts of the ship's hull and its appendages, and the reliability and function of the propulsion and steering systems, power generation and those other features and auxiliary systems which have been built into the ship in order to maintain essential services on board. Classification Societies aim to achieve this objective through the development and application of their own rules and by verifying compliance with international and/or national statutory regulations on behalf of flag Administrations" This is a thorough and continuous evaluation process that has several phases (Hormann, 2006). These phases include:

- A technical review of the design plans and related documents for a new vessel to verify compliance with the applicable rules and requirements.
- Attendance at the construction site of the vessel by a classification society surveyor to verify that the vessel is constructed in accordance with the approved design plans and classification rules.
- Attendance at relevant production facilities that provide key components such as the steel, engine, generators and castings to verify that the components conforms to the rules and requirements.
- Attendance at the sea trials and other trials relating to the vessel and its equipment.
- Upon satisfactory completion of the above, the builder's/shipowner's request for the issuance of a class certificate will be considered by the relevant classification society and, if deemed satisfactory, the assignment of class may be approved and a certificate of classification issued.
- Once in service, the owner must submit the vessel to a clearly specified program of periodical surveys, carried out on board the vessel, to verify that the ship continues to meet the relevant rules and requirements for its class.

The first phase, i.e the technical review of the design plans, is of special interest to this thesis as it is one that stands to gain a lot from new ways of utilizing computer technology. The use of customized high quality software in this phase can potentially improve the entire work flow and enable useful features such as e.g. keeping a history of decision, changes and discussions, in addition to organize the information in an intuitive manner. As we will see in the next sections, this phase already commonly makes use of high fidelity 3D models, but often in a much more narrow fashion than what could be possible.

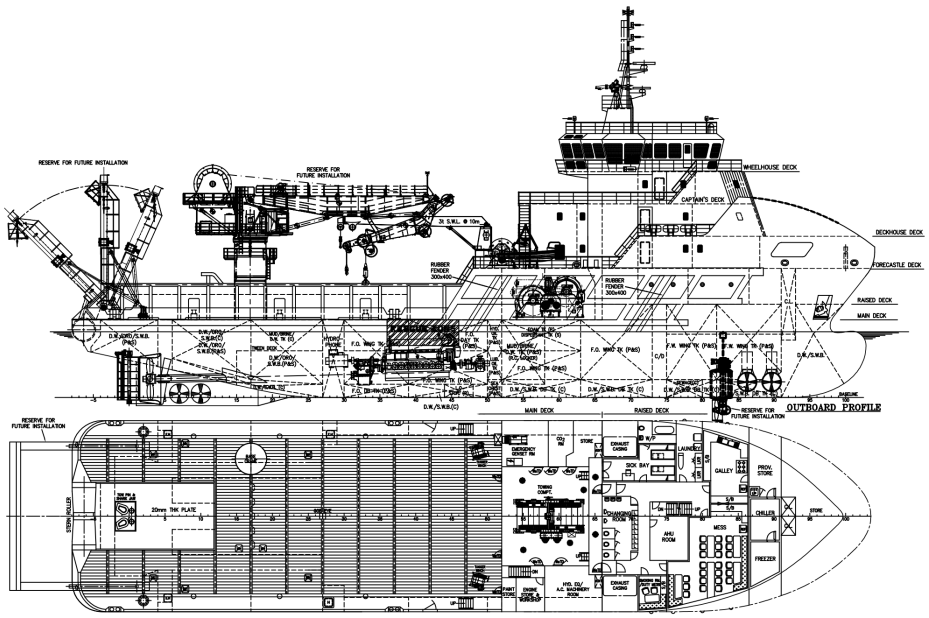

Figure 2.1: A typical paper based design sketch Marino Consulting (2017)

2.2 DNV GL's Digital Vision

DNV GL is the result of a merger, taking place in 2013, between two leading classification societies, Det Norske Veritas (Norwegian) and Germanischer Lloyd (Germany), and it is the world's largest classification society with about 15,000 employees and 350 offices operating in more than 100 countries. DNV GL provides services for more than 13 000 vessels and mobile offshore units, which represents a global market share of 21% (Jeffery, 2015). It is the world's largest technical consultancy to onshore and offshore wind, wave, tidal, and solar industries, as well as the global oil & gas industry – 65% of the world's offshore pipelines are designed and installed to DNV GL technical standards (Paschoa, 2013).

As a classification society, DNV GL operates in all the phases outlines above and, as mentioned in chapter 1 on page 1, they are currently investigating the idea of a virtual reality application for technical design reviews. Currently their technical design review process can be summarized by the following steps:

1. The designer sends the model to DNV GL for evaluation.
2. One or several *Approval Engineeers* from DNV GL inspect the model and notes down (usually on a document) aspects that don't meet DNV GL requirements.
3. The designer receives the remarks and has to make the necessary changes to the model to continue to process towards getting the classification.
4. The process is repeated until the design is approved.

This process usually results in a lot of papers being sent back and forth, and because of a lack of application support the process can, according to one DNV GL employee, feel very disconnected and "ad hoc".

Although digital 3D models usually are utilized in this phase, the work flow is reportedly still mostly based on design document and drawings. It is said that the model is usually just a reference, and is static (i.e receives no changes) throughout the process. After the model is "completed", and DNV GL starts its design review process, most comments, annotations and discussions are handled separate from the model, e.g on paper, while almost all communication is performed by either emails and phone calls. This might in part be because of limitations in the existing computer-aided design (CAD) software solutions, and in part because of companies' established practices.

DNV GL is thus intrigued by the prospects of digitalizing this process more, and make it more interactive and efficient by e.g utilizing an application that allows for *virtual* design review meetings in the 3D models. In these virtual design review meetings, the designer and reviewer could interact, survey the model together, and annotate it instead of model-printouts (on paper). This should also be possible without loosing the accountability that comes from today's paper trail. It is thus not necessarily the 3D models themselves that need to evolve, but rather the application that interfaces with them and what functionality they allow for.

As the sense of scale is important in a 3D model review, virtual reality technology is deemed promising as it gives a unique sense of scale and a depth, which is hard to match by regular "2D screens". DNV GL is also interested in alternate interaction methods, as mouse and keyboard can have some limitation when working in 3D environments (Rautaray and Agrawal, 2015). Gesture Recognition Technology has been of special interest as this can potentially offer unique approaches to working with 3D models.

2.3 Initial Design Ideas

The core functionality of a virtual reality design review application, such as the one outlined above, should be to navigate the 3D model from a first-person view and "annotate" it (i.e creating and placing remarks tied to a the model), primarily by using the advantages of virtual reality and gesture recognition technology. The users should also be able to create "sessions" that enable several users to be virtually present in the same instance of a 3D model (as opposed to different copies of it), and to interact with it using gestures. During these sessions a user should then be able to create annotations, which can be interacted with (e.g. edited or deleted) and are tied to the 3D model and the session.

Actions done during the 3D model session (such as annotating an object) should continuously be stored in a database or a distributed file system. If a user wants to re-enter the session at a later time, this database is read, and the actions done in previous sessions are loaded into the model.

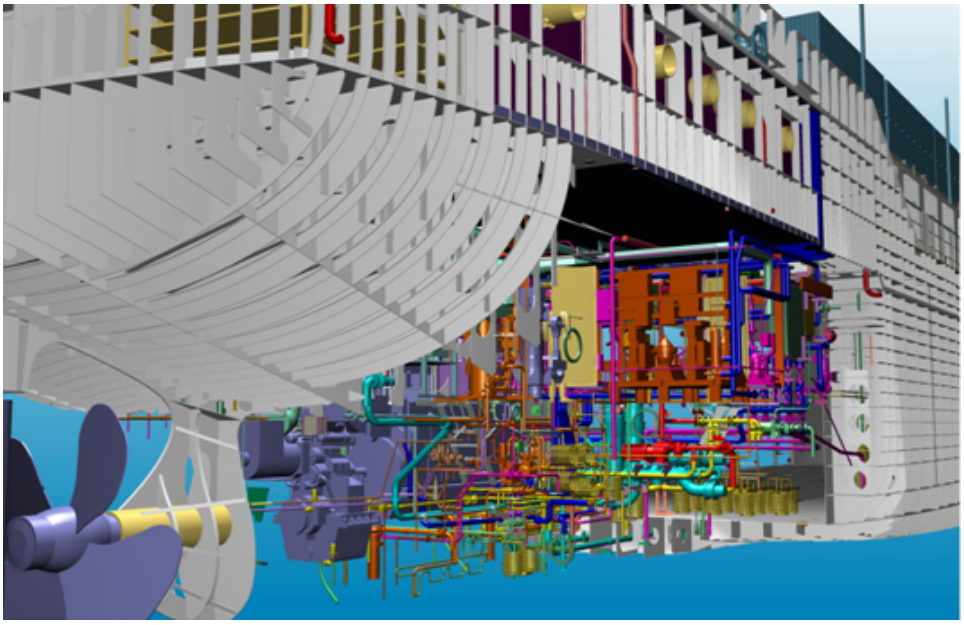

Figure 2.2: A ship model created using conventional Computer-aided Design (CAD) software. International Maritime Organization (2017)

By utilizing a database to store annotations in this way, the original model file (or files) could also remain unchanged throughout sessions, thereby avoiding conflicting or outdated versions of the model once it is submitted for a design review. This is opposed to the idea of saving the annotations as part of the model file(s), thus creating a new "version" of the model for each "save". The application could treat the storage of annotations as a revision control/version control system (VCS) would: The model itself could be the base (like a "first commit") and each time an annotation were added, edited and remove a "commit"" would happen were the "diff" (i.e the difference between the previous commit and this commit) would be stored. This would also allow for an annotation log, similar to a commit log in a VCSs, where "HEAD" (i.e the latest commit) would be the sum of the first commit and all following commits. If a session with a previous history were opened the application could thus load in all annotations by sequentially iterating through this annotation history (the "annotation commit log") and applying the diffs.

Another upside with utilizing a database is that it enables exposure of the actions done in the sessions to other platforms, such as web applications. This can enable annotation and comments done in a virtual design review session to become "issues" or "remarks" in more traditional collaboration tools such as Atlassian's Jira or Confluence. This might in its own way be a key approach to designing a virtual reality application that requires some sort of textual input. As a virtual reality headset might make it harder to utilize a keyboard, the user will then be able to first create annotations while using a virtual reality headset, and later input the text for those annotations through a web application tool. This would also allow

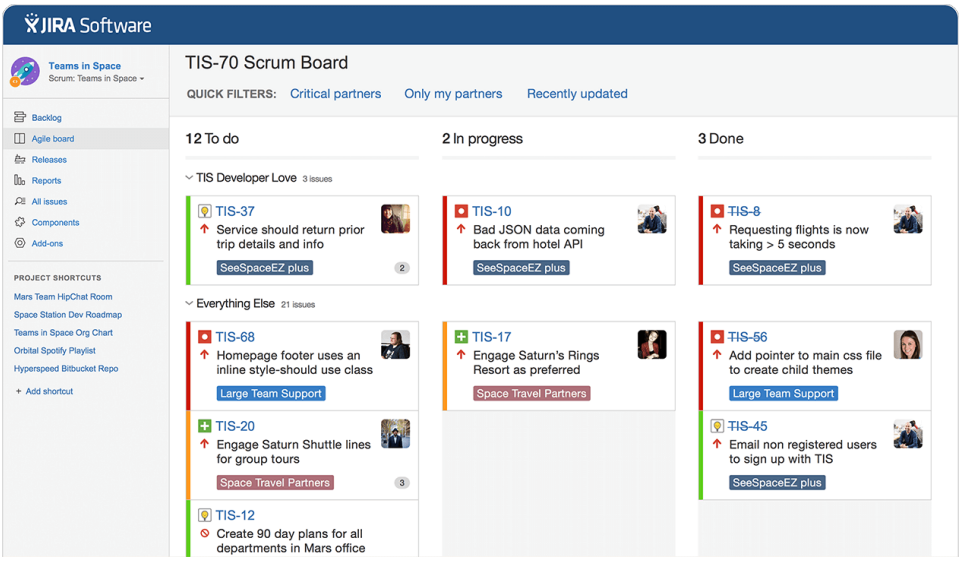

Figure 2.3: Annotation and comments created in a virtual design review session can become "issues" or "tasks" in more conventional web application collaboration tools, such as Atlassian's Jira. Both the web application and the design review application would be utilizing the same database, thus making changes performed in one of them also present in the other. Picture from Atlassian (2017).

the interested parties to access the annotations without necessarily having to enter the model again.

To approach these design ideas this thesis will first review the fields of virtual reality technology and gesture recognition technology, before revisiting the design in light of these reviews. The design is concretized and scoped in chapter 5 on page 33, where we select the focus and core functionality and write these as "use cases" (i.e informal, natural language descriptions of features, commonly used in software projects). Here we will also go more into the design issues that has to be addressed before the implementation starts, and review our technology choices.

Chapter 3

Virtual Reality Technology

This chapter will review what virtual reality is, how it is enabled through the use of virtual reality headsets and outline how virtual reality images are generated. After this, we will discuss some of the performance demands of virtual reality headsets and important considerations when designing and implementing virtual reality applications. We will also review the conditions *simulator sickness* and *virtual reality sickness*, which can be consequences of not adhering to the outlined considerations, and review ways to prevent this. More specifically we will discuss which factors that can be addressed in a virtual reality application's design and implementation phases, and which factors that are mostly due to individual differences and thus more outside the developers control. The lessons learned in this chapter will help guide the design and implementation consideration of the virtual reality design review application, which are discussed in chapter 5 and chapter 7.

3.1 The Basics of a Virtual Reality System

Virtual reality can be defined as a realistic and immersive simulation of a three-dimensional 360 degree environment, created using interactive software and hardware, and experienced or controlled by movement of the body (Leadem, 2016).

One of the most common ways to experience virtual reality is through virtual reality headsets, which are stereoscopic head-mounted displays (HMD) that provide separate images for each eye (Kuchera, 2016). These head-mounted displays are fastened to the user's head using straps - similar to those employed by headlamps - and, once firmly in position, should cover the user's entire field of vision. Virtual reality headsets contain one display per eye, often referred to as a *lenses*. These are positioned about 2-3 centimeters from their respective eye and have their own associated camera in the virtual world, giving each eye its individual video feed. These cameras are offset by the same length as the distance between the user's eyes, which enable depth vision and a true 3-dimensional experience (Abrash, 2012).

In addition to this, most virtual reality headsets also contain several

Figure 3.1: The Oculus Rift Development Kit 1, released by Oculus VR in 2012.

head motion tracking sensors that are built into the headset. These detect any movement, and either moves or rotates the cameras in the virtual environment in unison with the user's head movement, thus enabling the user to turn his or her head to "look around" in the virtual world (Kelly, 2016). This usually includes a gyroscope, which is responsible for measuring the orientation of the HMD, and sometimes an accelerometer to measure the proper acceleration of the HMD (Robertson, 2016). In addition, or instead of this, the first consumer versions of virtual reality headsets also usually utilize some other sensors or cameras outside the HMD. As an example the Oculus Rift CV1 utilizes constellation sensors (Feltham, 2015), which are usually positioned on a table, while the HTC Vive utilizes two Lighthouse Stations, which uses photosensors and structured light lasers to obtain the users position and rotation, and are usually placed in opposite corners of the room (Buckley, 2015). Several virtual reality headset vendors also offer controllers that are either included or sold separately. These are usually wireless and utilize similar sensor technology as the head mounted devices.

3.1.1 Generating Virtual Reality Images

There are a number of steps a virtual reality system has to perform from the moment a user performs an action to the moment that action is reflected visually on the displays. The first step of this is head tracking. The HMD sensors and the tracking software have to determine the exact position

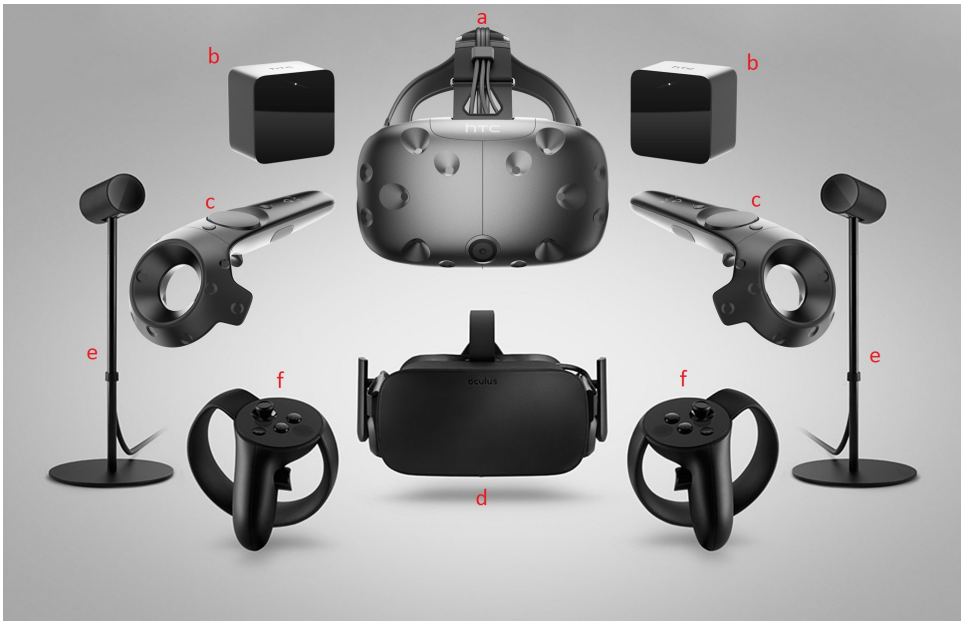

Figure 3.2: The HTC Vive and Oculus Rift Hardware. a) The HTC Vive headset (HMD). b) The HTC Vive Lighthouse Stations. c) The HTC Vive Controllers. d) The Oculus Rift headset (HMD). e) The Oculus Rift Constellation Sensors. f) The Oculus Rift Touch Controllers. Picture from Bye (2016)

and orientation of the HMD in the real world. Next, the application has to render the scene in stereo, i.e for both the cameras (as mentioned in section 3.1 on page 11), as it would look from that point of view. As pixel density usually is low for HMDs with a wide field of view (more on that later), this step should usually also include anti-aliasing to avoid jagged edges and pixelation, and to ensure "smoother" images. When the application has rendered the frames/images (one per eye) the frames need to be transfered to the HMD's displays by the graphics hardware. This is usually referred to as a scan-out and involves reading sequentially through the frame buffer, from top to bottom and moving right to left within each scan line, and streaming the pixel data for the scene over a link (e.g. a HDMI or Display Port cabel) to the displays (Abrash, 2012). When the displays receive the pixel data they have to start emitting photons for each pixel, and, at some point, they have to stop emitting the photons to prepare for the next frames to be displayed.

3.2 Virtual Reality Performance Demands

Virtual reality places some strict demands on performance and software design to avoid discomfort for the user. In many ways this is connected to how VR, as opposed to non-VR, applications "tricks" the user's brain into thinking the virtual experiences are actually real (giving it its "reality feel"). As a consequence of this, the user's brain tend to perceive VR-

applications differently from non-VR application, e.g by noticing anomalies more. One example of such an anomaly includes displacements of objects when the user's head rotates (i.e the objects are in the wrong position relative to the user's head) (Abrash, 2012). Failing to meet the performance demands outlined below can quickly result in significant discomfort for the user and give symptoms like headache, nausea or disorientation. Many of these symptoms are also common in the closely related conditions *simulator sickness* and *virtual reality sickness*, which will be reviewed in section 3.3 on page 18. Before this we will discuss what generally makes virtual reality applications more demanding in terms of execution, design and implementation than non-VR applications.

3.2.1 Latency Requirements

Virtual reality headsets have a much stricter requirements for latency, i.e the time required for an input to have a visible effect, than with use of regular displays (Lang, 2013). If this demand isn't met the system might often feel "sluggish" and the user will usually be more susceptible to virtual reality sickness. If e.g. too much time elapses between the time the user starts to turn his or her head, and the time the image is redrawn to account for the new head orientation, the visual image will feel disjoint for the user's action (Abrash, 2012).

The virtual reality system should thus have as low latency as possible. Abrash (2012), an engineer behind the HTC Vive and currently a Chief Scientist at Oculus VR, wrote that "when it comes to VR and AR, latency is fundamental – if you don't have low enough latency, it's impossible to deliver good experiences, by which I mean virtual objects that your eyes and brain accept as real" According to Abrash (2012) more than 20 milliseconds (ms) of latency is too much to be usable for virtual- and augmented reality, and a latency of 15 ms should be the absolute maximum.

One important component of latency is the *refresh rates* of the displays, i.e how often the display hardware updates its buffers and thus "draws" a new image on the displays. Both the Oculus Rift CV1 and the HTC Vive has a refresh rate of 90 Hz (i.e the display updates 90 times per second) for this reason, as opposed to the 60 Hz which is more common in commodity displays. In addition to refresh rate, the *frame rate*, i.e how often the graphics processing unit (GPU) renders new frames/images, is also important. To ensure that the displays don't "redraw" an identical frame on a buffer update the frame rate should thus ideally be the same or higher than the refresh rate (e.g 90 frames per second for the Oculus Rift CV1 or HTC Vive). Frame rate also determines the rendering latency, i.e the time is takes before an updated image, reflecting the user's latest actions, is produced. With 60 frames per second the rendering latency is on avaerage about 16.7 ms, while at 90 fps it is 11 ms on average, and thus under the 15 ms maximum threshold (not accounting for other phases). Refresh rate and frame rate are thus highly codependent, where latency is only as low as the weaker of the two allow. The target computer should thus have a CPU and GPU strong enough to meet a frame rate equal or above to the

HMD's refresh rate.

Asynchronous Reprojection

To reduce the perceived latency, or to compensate for a frame rate that is too low, several virtual reality HMDs make use of *asynchronous reprojection* (equivalent to what Oculus VR refer to as "asynchronous time warp") (S., 2016). This is a technique in which the virtual reality system generates intermediate frames in situations where the software (e.g a game) can't maintained the required frame rate (which is typically 90 fps with 90 Hz). In simple terms asynchronous reprojection produces "in-between frames", which is a manipulated version of an older rendered frame. This is done my morphing the frame according to the most recent head tracking data just before the frame is presented on the displays (S., 2016). By doing this, software that runs at e.g 45 FPS (frames per seconds) natively can be transformed into 90 FPS by applying asynchronous reprojection to each rendered frame. Every other frame is thus actually a manipulated version of the former frame. It should be noted that frames produced by asynchronous reprojection should only be regarded as "pseudo-frames" that compensation for lacking system performance in a rather performance-cost efficient manner, thereby giving lower-end system (such as Sony's Playstation 4) access to VR. Also note that these pseudo-frames, produced by asynchronous reprojection, are still more susceptible to unfortunate side effects, such as *positional judder*, than application rendered frames (the "real frames"). Positional judder is one of the most obvious visual artifacts using this approach and can make objects near the user seem "blurry" or unfocused (see figure 3.2.1 on the following page) (Antonov, 2015). Asynchronous reprojection should thus only be regarded as a technique to compensate for a lacking frame rate, as its side effects are still considered better than the negative effects (such as latency) a low frame rate has for the virtual reality experience (Dean Beeler and Pedriana, 2016).

3.2.2 Display Resolution and Pixel Density

Virtual reality headsets also have strict demands in respect to display resolution and quality. As the eyes of the user is closer to the displays than with a regular monitor, and the displays have to "wrap around" the user's whole field of view, flaws and shortcomings in the display technology become more apparent. One such example is the *screen-door effect (SDE)* (see figure 3.4 on page 17), which is when the lines separating the display pixel or subpixels are visible in the displayed image (Kumparak, 2016). To illustrate this issue Kumparak (2016) had the following remark about the Oculus Rift DK1 (released in 2013 with a resolution of 640×800 per eye): "Its low resolution screen (combined with magnification lenses that helped wrap the image around your view) made even the most beautifully rendered 3D environment look dated. It was like you were sitting too close to an old TV, or staring at the display through a screen door (aptly, this

Figure 3.3: Positional judder can make objects near the user seem "blurry" or unfocused. In the image the objects near the user are more blurry than than those that are farther away. This is because these object "move" more, relative to the user, from frame to frame as the user's head moves. Picture from Antonov (2015)

shortcoming quickly came to be known as "the screen door effect")" With the release of commercial and more high-end virtual reality headset, the screen-door effect have become less apparent. On the time of writing, two of the most sold virtual reality headsets, the Oculus Rift CV1 and the HTC Vive, both have a resolution of 1200 × 1080 per eye (a combined resolution of 2160 × 1200) with a pixel density of about 2450 ppi (pixel per inch), which is about ten times denser than the DK1's 215 ppi. With the improved pixel density, combined with the use of Fresnel lenses to create optical diffusion (i.e spreading out light to make it "softer"), the screen-door effect is severely minimized in the latest high-end virtual reality headsets (Davies, 2016).

Just as having low latency, and thus a high amount of frames rendered per second, demands much from the computer, so does the high resolution. As each lens has a resolution similar to a commodity computer display (1200 x 1080), the application must effectively render twice as many frames as it would when only using such a display with the same frame- and refresh rate. With the 90 Hz refresh rate of the Rift and Vive, this effectively means that 180 frames with a 1200 x 1080 resolution should ideally be produced per second.

3.2.3 Rendering Techniques

In addition to the considerable hardware demands a virtual reality system places on a computer, it also impacts how virtual reality applications should be designed and implemented. This is specially apparent with rendering techniques, i.e the process of generating images from 2D or 3D models. As this is a demanding process with high fidelity graphics, as

Figure 3.4: An example of the screen-door effect.

commonly found in modern computer games and other 3D applications, the rendering engine often employs several techniques and "tricks" to enhance it's performance, or at least make it seem so to the end user. Several of these optimization techniques' value can be diminished in a virtual reality setting, thus not yielding the the same performance benefit as they would when applied to non-VR applications.

Because of the latest virtual reality headset's high resolution and wide field of view (usually about 110 degrees), more of the virtual environment is shown to the user in a frame than is usual in non-VR applications (Ohannessian, 2015). This greatly affects several *culling* techniques, such as *frustum culling*[1] and *occlusion culling*[2], which are commonly used 3D rendering techniques for removing objects that don't contribute to the final image from the rendering pipeline (Johnson, 2013). Simply put, the rendering engine tries to only render what's actually visible to the user, while other objects are ignored to diminish the workload. As, on average, more objects are visible to a user wearing a HMD than a user using a regular display more object have to be rendered, which implies more work for the rendering engine (Ohannessian, 2015).

Another commonly employed technique in game development is using flat 2D images for certain parts of the virtual environment instead of 3D objects (Ohannessian, 2015). Examples of this are e.g. present in game "Super Mario 64", released for the Nintendo 64 in 1996, and one of the first commercially successful video games to utilize 3D. In this game one can often spot objects, such as balls, trees and fences, that tries to appear as 3D objects, but in reality are 2D images that are rotated to always face the camera. Similar techniques, although usually more subtly applied, are still commonly used today, but not for virtual reality applications. This is

[1] Frustum culling is concerned with only rendering objects that is within the field of view (the frustum pyramid) of the user.

[2] Occlusion culling consists of filtering out the objects that are entirely hidden behind other opaque objects (Pérez Fernández and Alonso, 2015).

simply because the "2D deception" becomes much more obvious with the depth perception the stereo cameras enable (Ohannessian, 2015).

3.3 Virtual Reality- and Simulator Sickness

Virtual reality sickness, also referred to as *cybersickness*, can occur with exposure to virtual environment when using virtual reality technology, and causes symptoms that are similar to those of motion sickness (LaViola, 2000). This condition also has many similarities with *simulator sickness*, which typically is experienced by pilots undergoing training in flight simulators, but as explained by Stanney et al. (1997) these two conditions are different. Simulator sickness, not using virtual reality, tends to be characterized by *oculomotor disturbances*, whereas virtual reality sickness, using virtual reality, tends to be characterized by disorientation (Stanney et al., 1997). Symptoms that can occur due to virtual reality sickness include headache, eye strain, nausea, sweating, disorientation (e.g through Vertigo [3]) and temporary loss of muscle coordinates (e.g through Ataxia [4]) (LaViola, 2000).

Contrary to motion sickness, where the user visually perceived to be still while in actual motion, virtual reality sickness often turns this around: The user visually perceive to be in motion while he or she in reality is stationary. Virtual reality sickness can thus in many ways be considered as "a reverse motion sickness". There are several theories on why virtual reality sickness occurs, with three of the more popular being the *sensory conflict theory*, *the poison theory* and the *postural instability theory* (LaViola, 2000).

The sensory conflict theory is the oldest and most supported of the three, and thus the one we will focus on, and claims that virtual reality sickness is caused by the conflict or discrepancies between the senses that provide information about the body's orientation, motion and acceleration. In a virtual reality setting this typically means that the visual sense perceive the body being in movement, while the *vestibular sense* [5] perceive that the body is stationary, thus causing a sensory conflict (LaViola, 2000).

The susceptibility for virtual reality sickness vary widely among users. Some users might experience it shortly after putting on the headset, while others may never experience it (Stanney et al., 2003). The causes for virtual reality sickness can vary, and while some are less under the VR application designer's control than others, they should still be understood by the VR designer (Stanney et al., 2003). To ensure an optimal virtual reality experience when using the design review application, outlined in section 2.3 on page 8, the design and implementation should address these potential consequences. The following subsections will thus review what's known about virtual reality sickness and what can be done from a design and implementation standpoint. The following two subsections

[3] A state where the user's surroundings appear to swirl dizzily (LaViola, 2000)

[4] A lack of voluntary muscle coordination

[5] a sensory system that partially provides the sense of balance and spatial orientation.

will review factors that contribute to virtual reality sickness, and make a distinction by what are mostly determined by individual differences and whats mostly determined by the virtual reality hardware and application design.

3.3.1 Individual Differences in Susceptibility

Research has identified some individual differences that correlate with the individual's susceptibility for experiencing virtual reality sickness. One observation is that the susceptibility for virtual reality sickness correlates heavily with motion sickness susceptibility, and factors that influence motion sickness susceptibility also usually influence virtual reality sickness susceptibility (Stanney et al., 2003). Below are some theories of the major contributing factors that are based on individual differences, and which are difficult to account for during the design of a virtual reality application. Note that some of these finding were originally for simulator sickness, but have proven to hold for virtual reality sickness as well.

Age

Research suggest that users between the ages of 2 and 12 are the most susceptible to virtual reality sickness, with a rapid decrease in susceptibility until an age of about 21 (Kolasinski, 1995). With regards to older users (e.g 50 years of age) the research findings seems to differ for virtual reality sickness and simulator sickness. Brooks et al. (2010) reported that older participants had a greater likelihood of simulator sickness than younger participants, while LaViola (2000) writes about how virtual reality sickness is almost nonexistent in participants of 50 years of age.

Gender

Women have proven more susceptible to simulator- and virtual reality sickness than men (Kennedy, 1985). The most common theories to explain this difference point out the genders' differences in hormonal composition, field of view (some research suggest that women has a wider field of view than men) and differences in depth cue recognition (Limited, 2012). Women are also most susceptible to virtual reality sickness during ovulation (Clemes and Howarth, 2005).

Ethnicity

Some ethnicities seem to be more susceptible to virtual reality sickness than others, suggesting a genetic component. Several studies indicate that asians tends to be more susceptible to visually-induced motion sickness, with the Chinese being more susceptible that European-Americans and African-Americans on measures to motion sickness induced by a circular vection drum, and with Tibetans and Northeast Indians having greater susceptibility than Caucasian races (Barrett, 2004).

Health

Symptoms of virtual reality sickness are more prevalent in people who are fatigued, sleep deprived, are nauseated or have an upper respiratory illness, ear trouble or influenza (Kolasinski, 1995).

Postural Stability

Users with a postural instability has been found to be more susceptible to visually-induced motion sickness, such as virtual reality sickness, and to experience stronger symptoms of nausea and disorientation (Kolasinski, 1995).

Experience with the Application

More exposure to virtual environments can train the brain to be less sensitive to their effects (Stanney et al., 2003). Users tend to become less likely to experience virtual reality sickness as they become more familiar with the virtual reality application. This adaption may occur with only a few seconds of exposure to the application (Kennedy, 1985).

In addition to this, people with a low threshold for detecting flicker and low mental rotation ability are more susceptible to virtual reality sickness Kolasinski (1995).

3.3.2 Virtual Reality Hardware and Design Factors

This section identifies some of the most common contributers to virtual reality sickness, which can be diminished or mitigated completely by either the virtual reality hardware or the virtual reality application design and implementation.

Flicker

Flicker is a contributing factor to virtual reality sickness, in addition to also being distracting for the user and a cause of eye fatigue (LaViola, 2000). There are two specially interesting observation done about flickering in research: First, the degree of perceived flicker is subjective as individuals have different *flicker fusion frequency thresholds*, i.e points at which flicker becomes visually perceivable (Pausch et al., 1992). Second, the likelihood that flickers are perceived increases as the field of view increases, as the peripheral visual system is more sensitive to flicker than the fovea vision system (i.e the sharp vision around the area the user focuses). This is again an important argument for the importance of a high enough refresh rate in virtual reality displays (discussed in section 3.2.1 on page 14).

Acceleration

As mentioned earlier sensory conflict during a virtual reality session might occur. This is especially noticeable during acceleration that is conveyed visually, but not to the vestibular organs (inner ear organs that responds to acceleration). The speed of movement does not seem to contribute to virtual reality sickness in the same scale as the vestibular organs do not respond to constant velocity.

Camera Control

Some theories indicates that the ability to anticipate and control the motion the user experiences plays a significant role in staving off motion- and virtual reality sickness (Rolnick and Lubow, 1991). Unexpected movement of the camera should thus be avoided in the virtual reality application. If the camera control is taken away from the user it is considered good practice to cue the impending camera movement to help the user to anticipate and prepare for the visual motion (Lin et al., 2004).

Field of View

The term "field of view" (FOV) can refer both to *display FOV* and *camera FOV*, which are similar, but still distinct concepts that can both have an effect on the user's proneness to virtual reality sickness.

Display FOV refers to the area of the visual field subtended by the display. As motion perception is more sensitive in the periphery view a wide display FOV can contribute to VR sickness by providing the visual system with more visual input, i.e more "area" in the periphery, than a smaller display FOV. This can lead to more sensory conflict as more of the visual view suggest that the user is moving, which he or she might be standing or sitting still. Reducing display FOV can reduce the changes of VR sickness (Draper et al., 2001), but can also reduce the level of immersion and awareness, and require the user to turn his or her head more than with a higher display FOV.

Camera FOV refers the area of the virtual environment that the graphics engine draws to the display. If the camera FOV is setup wrong, movement of the user head can lead to unnatural movement in the virtual environment (e.g a 15° rotation of the head can lead to a 25° rotation of the camera in the virtual environment). In addition to begin highly discomforting, this can lead to a temporary impairment in the vestibulo-ocular reflex, which is a reflex to stabilize images on the retinas during head movement (Stanney, 2002).

Latency and Lag

As mentioned earlier in this chapter, latency and lag can have a major impact on virtual reality sickness and the usability of the virtual reality application as a whole. Although designers and developers have no control over many aspects of a system's performance, it's important to make

sure the target virtual reality application doesn't drop frames or lag on a minimum technical specifications system (Dean Beeler and Pedriana, 2016). While some dropped frames or occasional jitter can be a minor annoyance in conventional applications or video games, it can have a much more discomforting effect on the user of a virtual reality application.

Some research indicates that a fixed, and thus predictable, latency creates about the same degree of VR sickness whether it's as short as 48 milliseconds or as long as 300 milliseconds, and that big and predictable latency or lag are more comfortable for VR users than smaller, but more unpredictable, latency or lag (Draper et al., 2001).

Focus Distance

Although not directly related to virtual reality sickness, it is still important to avoid discomfort and fatigue for the user by placing content s/he will be focusing on for extended amounts of time in an optimal range. As an example Oculus VR recommends such content to be placed a distance in the range of 0.75 to 3.5 Unity units/meters away from the camera (Dean Beeler and Pedriana, 2016).

3.4 Considerations for the Design Review Application

Throughout this chapter we have reviewed various aspects related to virtual reality, its performance demands and some potential contributers to virtual reality sickness. For the design- and implementation phase of our design review application, there are some aspect to be more concerned with than others. As will be discussed in chapter 5, 6 and 7, some of these potential issues, like incorrect field of view settings, are usually taken care of by the virtual reality headset's runtime environment, libraries or by other frameworks, and thus usually work correctly "out of the box". Other potential issues - like latency, lag and flicker - are usually avoided simply by utilizing sufficiently powerful hardware, with perhaps the HMD, CPU and GPU being the most important. An exception from this is with extremely performance demanding applications, in which case software optimization should be considered. Of topics that can be addressed at design- and implementation time acceleration, camera control and focus distance are included.

As the initial design ideas of the design review application revolves around the user virtually moving around a 3D model, while in actuality being stationary at his or her desk, some degree of sensory conflict is inevitable. There are however measures we can take to lessen the impact of such a sensation and combat virtual reality sickness. With regard to acceleration we can ensure that the acceleration always feels gradual instead of going from 0 m/s (meter per second) to e.g. 20 m/s in just a couple of frames (of which there should be about 90 per second). This gradual and slower increase in speed should not trigger the vestibular

organs to the same extent as higher acceleration. As gestures are more continuous in nature than buttons, which usually are discrete (either a button is pressed or not), they might be a natural fit for such a gradual acceleration. If acceleration is still too much of an issue, one can also limit the user peripheral vision and field of view by "framing" the frames and thus not have any movement appearing in the user's outer peripheral vision (see section 3.3.2 on page 21 for a quick discussion of this phenomenon). For the user this might seem like looking like unmagnified binoculars with the edges of his or her peripheral vision being obscured.

To combat virtual reality sickness we can also ensure that the user always has direct control of the camera. This is perhaps easier to ensure in our application, were there are few reasons to take the camera control away from the user, compared to e.g a game (which usually are much more event-driven). It does however become relevant and non-intuitive in certain scenarios. One such scenario, which is discussed in chapter 5 and 7, is the annotation form, which is an input schema the user can fill in when editing an annotation.

In a non-VR application this could typically be handled by showing the form as an overlay, temporary covering the model, while disabling any camera movement as long as the form was open. Although this might also seem like an intuitive thing to do in a virtual reality application, it does take camera control away from the user, which - as described in section 3.3.2 on page 21 - should be avoided. Because of this, the user should still be able to control the camera while the annotation form is open by, and thus be able to "look away" from the form.

There is also another consideration to take with regards to the annotation form, which is the focus distance (covered in section 3.3.2 on the facing page). Should the annotation form be an overlay (i.e be drawn directly on the screen space), as would be intuitive in a non-VR application, it would be too low a focus distance. This would be analogous to holding a sheet of paper directly in front of the user eyes and asking them to read it. The annotation form should thus be at a comfortable focus distance from the user, i.e 0.75 to 3.5 Unity units/meters away from the camera, and thus present in the world space instead of screen space, which is reviewed more in section 7.4 on page 67.

There are thus several potential issues to be mindful of when developing a virtual reality application. Just as this chapter has covered the landscape of virtual reality, the next chapter will cover the landscape of gesture recognition technology, a technology which can extend the possibilities of virtual reality and for our virtual reality design review application.

Chapter 4

Gesture Recognition Technology

4.1 Gesture recognition devices

Gesture recognition technology is a field that has gained much attention with the growth of the virtual reality field, and it's a very diverse one with roots in sensor technology, image processing and computer vision (Vafadar and Behrad, 2014). The first attempts at a commercial hand gesture recognition system were typically glove-based control interfaces, often called ***data gloves*** and were gloves with sensors attached to it. As the image processing and computer vision technology wasn't mature yet, these ***contact-based devices*** remained the primary gesture recognition technology, until the image processing-reliant ***vision-based devices*** began to see some success in the 2000s (Premaratne, 2014). Another factor which made data gloves ideal was a very limited requirement for processing power, as any pre-processing were rarely done, and thus the systems could run optimally on the commodity 1980s and 1990s computers (Premaratne, 2014).

Today, both contact-based and vision-based devices are utilized for gesture recognition purposes.

Contact-based devices are usually wearable objects, such as gloves or armbands, which register the user's kinetic movement through sensors and attempt to mirror it in the virtual world. Some notable products making use of this technology include the Nintendo Wii remote controller and the Myo armband (see figure 4.2 on the following page).

Vision-based devices usually make use of either depth-aware cameras or stereo cameras to approximate a 3D representation of what's output by the cameras, which in many ways are similar to how the human eyes work. Products making use of this technology include the Microsoft's Kinect and the Leap Motion controller (see figure 4.3 on page 27).

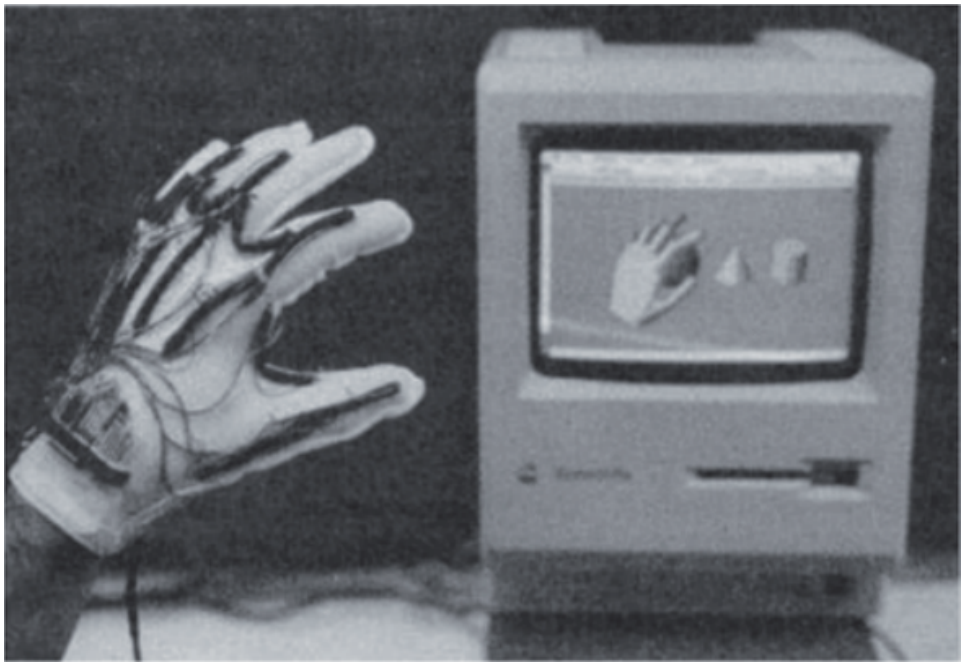

Figure 4.1: The Z Glove, developed by Zimmerman in 1982. Picture from Premaratne (2014)

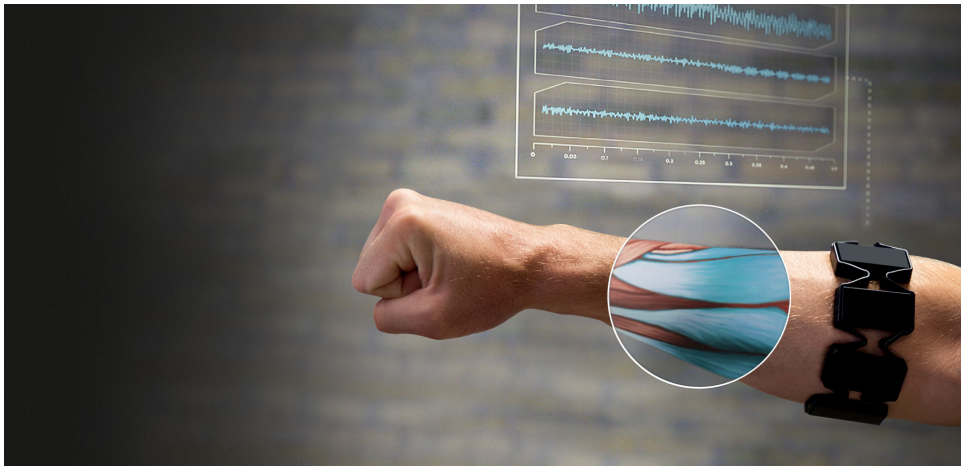

Figure 4.2: The Myo armband is a gesture recognition device worn on the forearm and manufactured by Thalmic Labs. The Myo enables the user to control technology wirelessly using various hand motions. It uses a set of electromyographic (EMG) sensors that sense electrical activity in the forearm muscles, combined with a gyroscope, accelerometer and magnetometer to recognize gestures (Silver, 2015).

Figure 4.3: The Leap Motion Controller is a small USB peripheral device which is designed to be placed on a physical desktop, facing upward. Using two monochromatic IR cameras and three infrared LEDs, the device observes a roughly hemispherical area, to a distance of about 1 meter, and generates almost 200 frames per second of reflected data (Colgan, 2016).

Both approaches have their advantages and disadvantages (see Rautaray and Agrawal (2015) for a deeper discussion of these). Contact-based devices generally have a higher accuracy of recognition and a lower complexity of implementation than vision-based ones. Vision-based devices are on the other hand seen as more user friendly as they require no physical contact with the user.

The main disadvantage of contact-based devices is the potential health hazards, which may be caused by some of its components (Maureen Schultz, 2003). Research has suggested that mechanical sensor materials may raise symptoms of allergy and magnetic component may raise the risk of cancer (Nishikawa et al., 2003). Even though vision-based devices have the initial challenge of complex configuration and implementations, they are still considered more user friendly and hence more suited for usage in long run. Because of the reasons outlined above this thesis will primarily be oriented towards vision-based gesture recognition technologies.

4.1.1 The primary Vision-based Technologies

Today, there are three primary vision-based technologies that can acquire 3D images: Stereoscopic vision, structured light pattern and time of flight (TOF) (Ko and Agarwal, 2012). These all make use of one or several cameras and lights to capture and recognize certain movements or poses from the user, and transform it to a certain action on the computer (e.g. a recognized finger tap might be the equivalent to left mouse button click).

Stereoscopic vision is the most common 3D acquisition method and uses two cameras to obtain a left and right stereo image. These images are

	Stereoscopic vision	Structured light	Time of flight (TOF)
Software complexity	High	High	Low
Material cost	Low	High/Middle	Middle
Response time	Middle	Slow	Fast
Low light	Weak	Light source dep (IR or visible)	Good (IR, laser)
Outdoor	Good	Weak	Fair
Depth ("z") accuracy	cm	μm ~ cm	mm ~ cm
Range	Mid range	Very short range (cm) to mid range (4–6 m)	Short range (<1 m) to long range (~ 40 m)
Applications			
Device control			✔
3D movie	✔		
3D scanning		✔	

Figure 4.4: Comparison of Vision-based sensor technologies (Ko and Agarwal, 2012).

slightly offset on the same axis as the human eyes. As the computer compares the two images, it develops a disparity image that relates the displacement of objects in the images.

Structured light measure or scan 3D objects through illumination. Light patterns are created using either a projection of lasers or LED light interference or a series of projected images. By replacing one of the sensors of a stereoscopic vision system with a light source, structured-light-based technology basically exploits the same triangulation as a stereoscopic system does to acquire the 3D coordinates of the object. Single 2D camera systems with an IR- or RGB-based sensor can be used to measure the displacement of any single stripe of visible or IR light, and then the coordinates can be obtained through software analysis.

Time of flight is a relatively new technique among depth information systems and is a type of light detection and ranging (LIDAR) system that transmits a light pulse from an emitter to an object. A receiver determines the distance of the measured object by calculating the travel time of the light pulse from the emitter to the object and back to the receiver in a pixel format.

Of these technologies stereoscopic vision is perhaps the most promising one for the consumer market as it has the lowest material cost (Ko and Agarwal, 2012), and has proved more reliable in variable light conditions than its counterparts. One of the latest consumer-oriented devices of this kind is the Leap Motion Controller, which distinguishes itself for having a higher localization precision than other depth vision-based devices (Weichert et al., 2013), and also for capturing depth data related to palm direction, fingertips positions, palm center position, and other

relevant points (Lu et al., 2016). The Leap Motion Controller will be reviewed more in-depth in the next chapter.

4.2 Gesture Recognition Principles

A gesture can be defined as a physical movement of the hands, arms, face and body with the intent to convey information or meaning (Mitra and Acharya, 2007), Even though the use of keyboard and mouse is a prominent interaction method, there are situations in which these devices are impractical for human-computer interaction (HCI). This is particularly the case for interaction with 3D objects (Rautaray and Agrawal, 2015).

To be able to convey semantically meaningful commands through the use of gestures one must rely on a gesture recognition system, which is responsible for capturing and interpreting gestures from the user and, if applicable, carry out the desired action. Often this process is seen as a sum of three fundamental phases: Detection, tracking and recognition (Rautaray and Agrawal, 2015). This section will describe what makes up a gesture recogntion system, with special emphasis on hand gesture recognition, and summarize some common challenges with vision-based gesture recognition methods.

4.2.1 Static and dynamic gestures

In the gesture recognition field it is common to define a gesture as either a static or dynamic. ***Static gestures*** can in simple terms be defined as gestures without any movement. The hand and its fingers and joints simply maintain a certain position or orientation and it is recognized as a gesture. One example of this gesture category is the "V sign" (or the "peace sign"), where the index and middle fingers are raised and parted while the other fingers are clenched.

Dynamic gestures, on the other hand, are gesture that involves or requires movement for the gesture to have meaning. One example of this might be to wave goodbye to someone or to twist a straight hand back and forth to indicate uncertainty. One can classify dynamic gestures into several subclasses, such as conscious gestures, which are done intentionally for communication purposes, or unconscious gestures, which are carried out unconsciously. See 4.5 on the following page for an hierarchical overview.

4.2.2 Detection

The first step in a typical gesture recognition system is to detect the relevant parts of the captured image and segment them from the rest. This segmentation is crucial because it isolates the relevant parts of the image from the background to ensure that only the relevant part is processed by the subsequent tracking and recognition stages (Cote et al., 2006). A gesture recognition system will typically be interested in hand gestures, head- and arm movements and body poses, and thus only these factors should

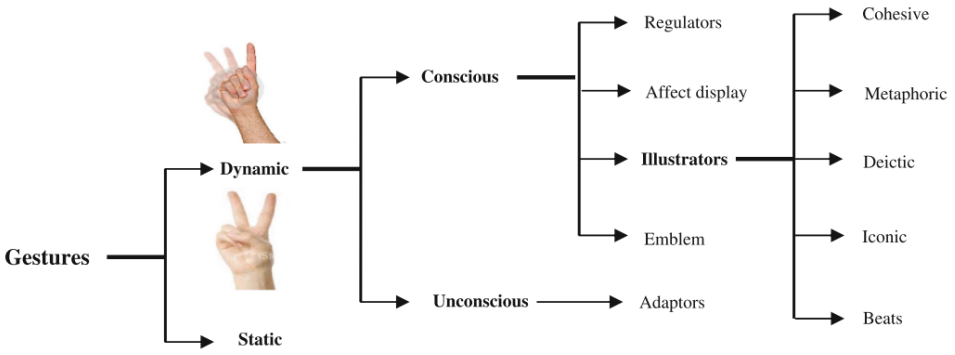

Figure 4.5: The vision-based hand gesture categories (Kaaniche, 2009).

be observed by the system. A gesture recognition system interested in detecting e.g. hand gestures should thus only consider hands as a relevant segment, and thus only observe these.

Many different detection methods have been proposed by research, each using different visual features to detect relevant segments. Example of such visual features include skin color, shape, motion and anatomical models of the hands (Cote et al., 2006).

Color detection is a method of detecting the relevant segment (e.g. hands) by its color. When employing this method one important decision is what color space to use, though color spaces efficiently separating the chromaticity from the luminance components of color are typically the preferred ones. These are favored as they have some degree of robustness to illumination variability, which is a weakness of this detection method. In addition to this skin color detection also have performance problems when the background contains objects that have a color distribution similar to human skin, although this can be combated by *background subtraction*, and with variability in human skin tones (Rautaray and Agrawal, 2015).

Shape detection is a method of detecting the relevant segment by its shape, and usually tries to extract the contours of objects to judge whether those objects are relevant or not. An advantage with this method over color detection is that it's not directly dependent on skin color or illumination, although these are still a factor (Rautaray and Agrawal, 2015). However, a major disadvantage with this methods relates to occlusion and viewpoint problems, which might cause a hand to not be recognized as one because of the camera angle and/or the hands orientation and configuration. One way to prevent this might be to use several cameras with different viewpoints. Shadows can also cause a problem as shadows of a hand often will be detected as hands themselves. Because of these disadvantages it is more common to use this method in combination with other ones rather than on its own.

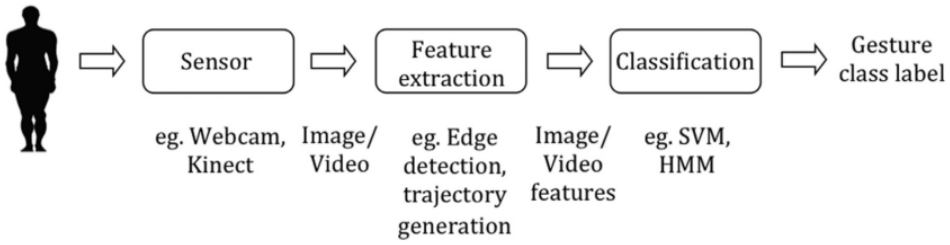

Figure 4.6: A typical gesture recognition pipeline (Pisharady and Saerbeck, 2015)

Motion detection is a method of detecting the relevant segment though motion, and assumes that all moving object are relevant. When used as a gesture recognition scheme it requires a very controlled setup as it assumes that the only motion in the image is caused by hand movement. This method is also more common to use in combination with other methods.

4.2.3 Tracking

The second step in a gesture recognition system is to track the movements of the relevant segments of the frames, e.g. the hands. Tracking can be described as the frame-to-frame correspondence of the segmented hand regions and aims to understand the observed hand movements. This is often a difficult task as hands can move very fast and their appearance can change vastly within a few frames, especially when light condition is a big factor (Wang and Li, 2010). One additional note is that if the detection method used is fast enough to operate at image acquisition frame rate, it can also be used for tracking (Rautaray and Agrawal, 2015).

4.2.4 Recognition

The last step of a gesture recognition system is to detect when a gesture occurs. This often implies checking against a predefined set of gestures, each entailing a specific action. To detect static gestures (i.e postures involving no movement) a general classifier or template-matcher can be used, but with dynamic gestures (which involves movement) other methods, which keep the temporal aspect, such as a Hidden Markov Model (HMM), are often required (Benton, 1995). The recognition technology often makes uses of several methods from the field of machine learning, including supervised, unsupervised and reinforced learning.

When a gesture recognition system detects a relevant segment, it is thus tracked and represented in some way in the system. For hand gesture representations, which is the most relevant for this thesis, there are two major categories of hand gesture representations: 3D model-based methods and appearance-based methods (Rautaray and Agrawal, 2015).

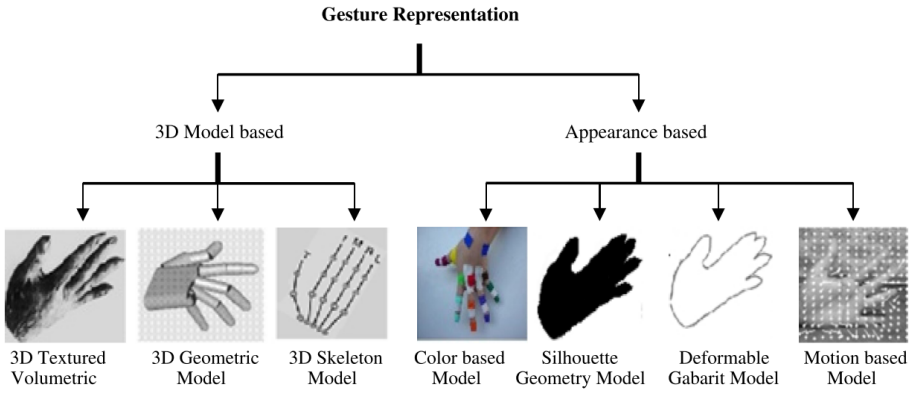

Figure 4.7: Vision-based hand gesture representations (Bourke et al., 2007)

Chapter 5

Application Design

5.1 The core functionality

In section 2.3 on page 8 we discussed the fundamental design ideas for a virtual reality design review application. To ensure that such an application could change the work flow of DNV GL's design reviews, multiple design aspects should be met and a satisfactory infrastructure would need to be set up. As this thesis' scope is limited to virtual reality and gesture recognition technology's role in such an application, several of the components necessary for a satisfactory product will not be implemented to focus more one these aspects. The resulting thesis implementation will thus be more a prototype or proof-of-concept of how virtual reality and gesture recognition technology can be used to interact and work with 3D models.

5.1.1 Application use cases

This section gives an overview of the application functionality, which will be implemented in this thesis. Initially the design also contained specification for a "Launcher", i.e a program that gave the boot arguments to the program itself. In this Launcher program a user could either "host" a design review session by selecting one or multiple 3D models(s) and invite other users, or "join" a session by either accepting an invite or browse available sessions. Because of time concerns, and to give more time to prioritize the more thesis-relevant use cases, this functionality was cut and the application instead boots up to a specific tanker model.

Once the user is loaded into this model, he or she should ble able to do the following:

Choose between Virtual Reality Mode and Desktop Mode. Virtual reality mode is meant to be used with a virtual reality headset and sets up the correct settings (e.g the field of view). Desktop mode is meant to be used without a virtual reality headset and instead used a regular display. This modes sets up the best setting for regular display usage, and if a

virtual reality headset is attached the input from it will be ignored (e.g. To avoid its orientation affecting the camera in the application).

Look around. By looking around the camera should rotate to the desired direction, but the player model should keep its orientation (e.g. "forward" is the same directing independent of where the user is looking). Looking around can only be achieved by the user turning his or her head while wearing a virtual reality HMD and having the application run in VR mode.

Rotate (i.e change orientation). When rotating the camera and the player model should rotate in the desired direction (e.g. "forward" is where the player model is facing after the rotation). Rotation should allow pitching and yawing (rotation along the Y and Z axis), but not rolling (rotation along the X axis) as this might cause the user discomfort, especially when using a virtual reality headset, and has little to no practical implication. See **??** on page ?? for an illustration of this. Rotation should be possible either by using a gesture or by moving the mouse.

Move (i.e change position). The user should be able to move freely along the X, Y and Z axis, thus moving both in the horizontal and vertical plane. This movement should happen without regard for any external forces, such as gravity or collision. The user should be able to this movement by using the keyboard or by using gestures. On the keyboard six different keys should be used (forward, backwards, left, right, up and down), while the same should be accomplished by either three distinct gestures (forward/backward, left/right, up/down) or one combined gesture (forward/backwards/left/right/up/down).

Annotate a point. The user should be able to create and attach an annotation, i.e a unit of information related to an aspect of the 3D model, to a point on a surface in the 3D model. These annotations can visually be represented as a sphere or orb in the model (to make it uniformly visible from all angles). This should be accomplished by either clicking the mouse or using a gesture.

Annotate an object. The user should be able to annotate a whole object in the 3D model, as opposed to only annotating a point on it. When an object, such as a wall, pipe or gear, is annotated in this fashion it should be highlighted or marked in some distinct manner. This should be accomplished by either clicking the mouse or using a gesture.

Edit an annotation. The user should be able edit an annotation, either tied to a point or an object, by clicking on the annotation. This should bring up a form, that should at least offer the following functionality:

- Textual input through a text box.

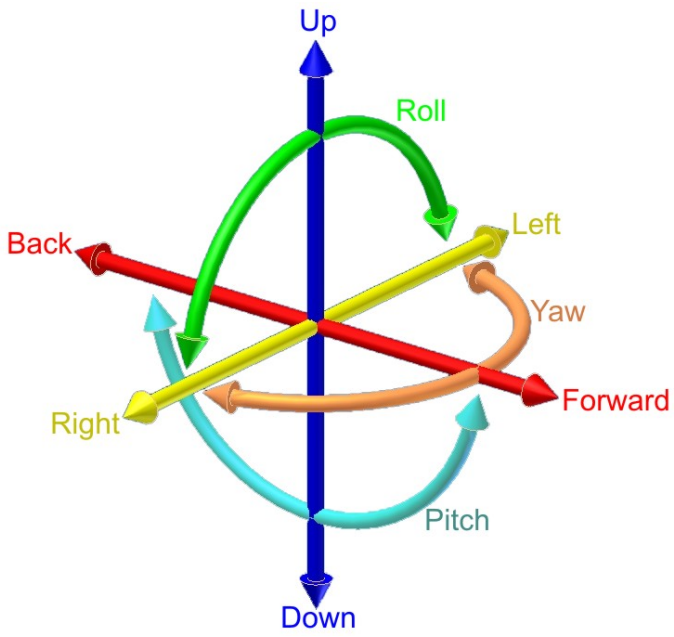

Figure 5.1: The six degrees of movement in a three-dimensional space. A rigid body in this space can change **position** along the X axis (left/right), Y axis (up/down)and Z axis (forward/backward), or change **rotation/orientation** along the X axis (rolling), Y axis (pitching) and Z axis (yawing). Picture from Horia Ionescu (2010)

- A submit-button to save the current annotation state and close the annotation form.
- A cancel-button to close the annotation form without saving any changes.
- A delete-button to delete the annotation, i.e removing the annotation sphere or highlighting and all it's associated information.
- Choosing between several annotation categories, labels or states by clicking on one of several associated buttons. These should function as radio-buttons, i.e when one is selected the others are always deselected. These categories could refer to the progress status of the task the annotation represents, e.g. "unresolved", "work in progress" and "approved", or they could represent the nature of the annotation itself, e.g. "information", "warning" and "error".
- A virtual keyboard that can be used instead of the physical keyboard to input text. This is primarily included so the user can input text using gesture recognition technology.

Access a menu. The user should be able to access a menu that offers different options related to the usage of the application. The menu should allow the the user to:

- Go back to the origin position, e.g move and rotate the player model to the same position and orientation as when the application was started.

- Choose whether the annotation spheres should be globally visible (e.g. visible through walls), only visible with line-of-sight or invisible. The first of these options is there to ensure that the user easily can see every annotation, regardless of where the user is in the model, while the other options are there for preference. The default should be global visibility.

- Toggle between (i.e turn off or turn on) gesture recognition based on whether it's already turn on or off. This is to enable the user to use his or her hands without it having effect on the application.

- Toggle between having X, Y, and Z axis movement as three separate gestures (forward/backward, left/right, up/down) or one (forward/backwards/left/right/up/down).

5.2 The gestures

As mention in the use cases for the application, all of the application's functionality should be accessible by using gestures. The user should thus be able to do every task only by using gestures (except the "look around case", which only be done by rotating the HMD). To support this a gesture scheme of seven (or eight depending on perspective) individual gestures were created. The gestures can all be considered static gesture, meaning that they don't require movement for the gesture to be detected, except for the movement required to form the gesture. Even though the gestures can be considered static in this aspect, the user is still often required to move his or her hand while holding the gesture to get the desired effect. The application should also give visual feedback when a gesture is recognized. This could be by altering the hand model in some way (e.g color it differently by switching materials) or by having some user interface showing it.

The gestures are individually described below on a functional level and will be covered in more technical detail during the next chapter. Both the left- and right hand should be able to execute all these gestures independently, so scenarios where both hands do the same gestures, or different gestures, should work. The only exception from this is the menu gesture, where one hand is assigned to be "the menu hand" (the left hand by default) and one is assigned to be "the selector hand" (right by default). The gestures are design to be as distinguishable from each other as possible (i.e so the gesture recognition system doesn't mistake one gesture for another), and to work with the cameras (assuming a vision-based system) positioned in several different positions (i.e also distinguishable from different angles).

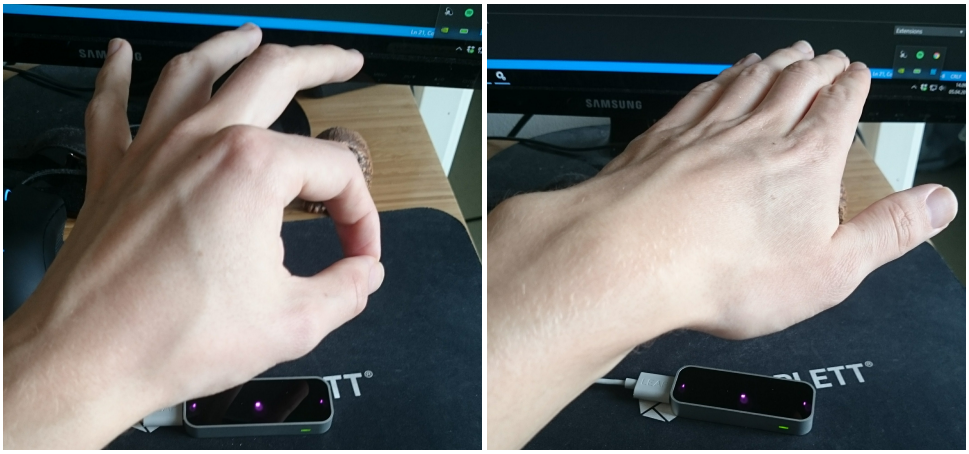

Figure 5.2: The pinch gesture (left) is used to rotate the camera along the y- and z-axis. The palm-down gesture (right) is used to move the user up and down along the y-axis.

5.2.1 The pinch gesture

The pinch gesture will cover the "rotate user case", specified in the previous section, and thus enable the user to rotate the camera by the Y and Z axis. The pinch gesture is accomplished by squeezing the tip of the thumb and index fingers together while, preferably, keeping the rest of the fingers erect and the palm facing somewhere between the table top and the displays (see 5.2 for an illustration). Once this gesture is done by the user, the system should indicate that the gesture was recognized as a pinch gesture. Once the system has recognized the pinch gesture it sets the x, y and z coordinates where the gesture was detected as an origin point and starts rotating the camera with the offset value of this origin point. This means that when the user does a pinch gesture without moving the hand, the pinch gesture should be detected and be "active", but the camera should not be moved. If the user then moves his or her hand to the right, while still keeping the pinch gesture, the camera should start rotating to the right also. If the user moves his or her hand further to the right the camera should start rotate at a faster rate than previously. The primary idea behind this origin-offset scheme, which also are used in other gestures, is to prevent user fatigue by allowing the user to execute the gesture in the position that feels most comfortable, as long as this position is captured by the vision-based gesture recognition system. In addition this scheme also prevents the user from having to move his or her hands as much as some other schemes would (e.g. dragging motions).

5.2.2 The palm-down gesture

The palm-down gesture, alternatively called the Y-gesture, fulfills the up-and-down functionality specified in the "move-user case", and enables the user to move the player model along the y-axis, relative to its orientation.

The palm-down gesture is accomplished simply by having all fingers extended, with all of them pointing in the direction of the display with the palm facing downwards towards the table top (see 5.2 on the preceding page for an illustration). This gesture, along with the rest of the "movement gestures", uses the same origin-offset scheme as the pinch gesture, but the offset is in this gesture only measured on the y-axis, so moving the hand to the right, as mention in the pinch gesture section, will cause no movement when the palm-down gesture is the active gesture. Instead the user can move his or her hands up and down on the y-axis, so the distance to the table top varies.

5.2.3 The palm-side gesture

The palm-side gesture, alternatively called the X-gesture, fulfills the left-and-right functionality specified in the "move-user case", and enables the user to move the player model along the x-axis, relative to its orientation. The palm-side gesture is accomplished simply by having all fingers extended, with all of them pointing in the direction of the display with the palm perpendicular (i.e at a 90° or 270° angle) to the table top (see 5.3 on the next page for an illustration). As one of the movement gesture, this gesture also uses the origin-offset scheme, but only with the x-axis monitored.

5.2.4 The fist gesture

The fist gesture, alternatively called the Z-gesture, fulfills the forward-and-backwards functionality specified in the "move-user case", and enables the user to move the player model along the z-axis, relative to its orientation. The fist gesture gesture is accomplished by forming a fist (i.e with no fingers extended) and is used by extending and retracting the fist.

5.2.5 The combined-movement gesture

The combined-movement gesture, alternatively called the XYZ-gesture, is a special gesture that's only enabled if the "use combined gesture" option is selected in the menu. When this gesture is enable the other movement gestures, i.e the palm-down-, the palm-side- and the fist gesture, are disabled. If the "use combined gesture" is disabled, by clicking "distinguish movement gestures" in the menu, the other movement gestures are once again enabled. This gesture is done in the same manner as the palm-down gesture, i.e by having all fingers extended, with all of them pointing in the direction of the display with the palm facing downwards towards the table top. However, instead of now only being responsible for navigation along the y-axis, i.e up and down, this same gesture is now responsible for movement along the x-, y- and z-axis. This gesture also used the origin-offset scheme, but now all the three dimensions are monitored.

Figure 5.3: The palm-side gesture (left) is used to move the user left and right along the x-axis. The fist gesture (right) is used to move the user forward and backward along the z-axis.

5.2.6 The single-point gesture

The single-point gesture is used to annotate a point or edit a point annotation, and is used by having the index finger extended and "pointing" at the display while the rest of the fingers are non-extended (the thumb can be either extended or not extended). When the user does the single-point gesture, a raycast (a kind of invisible beam) should be fired from the player model towards where the player model is facing. The player should thus be able to aim, e.g. by utilizing a crosshair in the middle of the players screen, by looking at a spot and use the single-point gesture to fire off the raycast. At the point the raycast collides with a part of the model a point annotation should be created. If the user use the single-point gesture again, while still aiming at the same spot (where an annotation now is), the annotation form should open up to supply input to the annotation.

5.2.7 The double-point gesture

The double-point gesture is used to annotate an object by highlighting it, or to edit a object annotation. The double-point gesture is invoked by pointing the index- and middle finger at the screen with a slight angle between them, while the rest of the fingers are non-extended (the thumb can be either extended or not extended). Apart from this the double-point gesture function very similar to the point gesture, with some few exception. Object annotations are edited by using the double-point gesture at them again, as opposed to using the single-point gesture, which created a point annotation on the annotated and highlighted object.

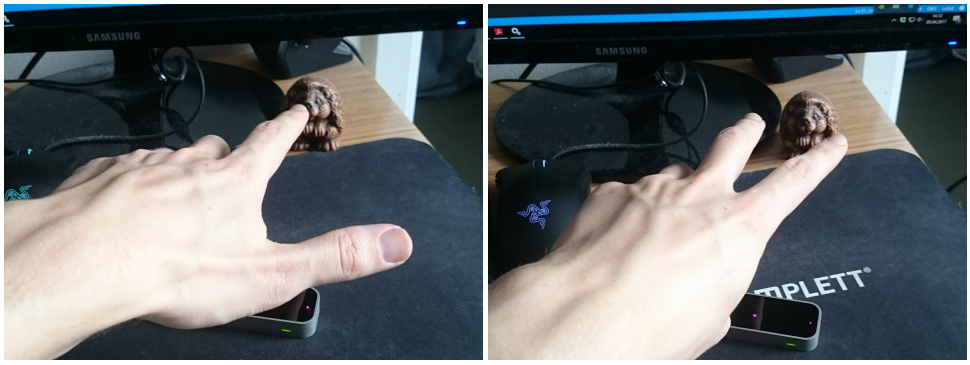

Figure 5.4: The single-point gesture (left) is used to create or edit a point annotation. The double-point gesture (right) is used to create or edit an object annotation.

5.2.8 The menu gesture

The menu gesture gives the user access to a menu especially design to use with gestures. The menu is invoked by extending all fingers on "the menu hand" (the left hand by default) and turn it so the palm faces the user. When this gesture is recognized by the system a menu should appear in the shape of a fan with its root in the palm of the user. The user can then use the index finger on the other hand, "the selector hand" (right hand by default), and click on one of the buttons by holder the tip of the index finger within the range of the button (i.e close enough to the button in terms of x-, y- and z coordinates). If the tip of the index finger is close enough to the button, the button will start to "fill up", indicating that it is in the process of being pressed. Once the button is "filled up" the selection is registered and the action the button represents is carried out. Not that this mechanism is in place to prevent miss-clicks from the user.

5.3 Technology Choices

There are several technology choices to make with regards to the implementation of the design review application. These span from programming framework, language, and what gesture recognition and virtual reality vendors to use, and these decisions also must address capability between the different technologies.

5.3.1 The Game Engine

A game engine is a software framework, usually designed for development of video games. The core functionality of a game engine typically includes a rendering engine, a physics engine (at least providing collision detection), sound, scripting, animation, networking, streaming, memory management, and threading (Gregory, 2014). As game engines are created to enable development of complex 3D environments and contain many of

the facilities necessary for the application use cases outlined above, they provide a good foundation for the implementation. Even though several game developers develops their own proprietary game engines, which are kept strictly private to the company, there are several commercially available ones as well. The biggest of these is the Unity and the Unreal engines, both with broad support from a number of third party vendors. This is a great benefit for the implementation as support "straight out of the box" for our choice of virtual reality and gesture recognition technology will ease the development process.

Of these two Unity were chosen as it was requested from DNV GL, as they have more experience with it from other projects. The Unity engine and its central concepts will be discussed in chapter 6 on page 43.

5.3.2 Why Leap Motion?

When deciding on what vision-based gesture recognition system the Leap Motion Controller was chosen. This is primarily because of vision-based gesture recognition systems that focus on hand gesture, this seemed like the most mature. The Leap Motion Controller also has a convenient API, with support for several high-level languages, and integration with Unity is well supported both through software libraries and documentation. The Leap Motion Controller and its API will be reviewed in chapter 6 on page 43.

5.3.3 Oculus Rift and HTC Vive

As mentioned in earlier sections, two of the most popular virtual reality HMDs are the Oculus Rift and the HTC Vive. These two HMDs both have good support for usage with Unity, but have separate SDKs (Software Development Kits) and runtime environments. Even though this is the case the implementation will support both of these HMDs. There are a couple of reasons for this: Unity 5 support both HMDs natively (i.e "out-of-the-box"), so adding the basic VR functionality is trivial and implementing support for both enable a comparison of their performance in the application.

Chapter 6

Technical Review

This chapter will give a brief introduction to central concepts of the software frameworks outlined in the previous design chapter. The sections covering the Unity game engine and the Leap Motion Controller are both based on their documentation pages, found at https://docs.unity3d.com and https://developer.leapmotion.com. Note that although these introductions are brief, more detailed information is mentioned as necessary in the implementation chapter.

6.1 Unity - The Cross-platform Game Engine

Unity is a cross-platform game engine developed by Unity Technologies, and is a popular engine both in a personal- and enterprise settings. Unity Personal is a free version for individuals and enterprises making less than 100 000$ a year off content created in Unity, and is the edition used in for the implementation phase. Unity Personal is full-featured and comes with all the necessary subsystems typically required in a game engine, like rendering, physics and scripting. Unity makes use of either C# or UnityScript (a dialect of JavaScript) as scripting language, where the former is the preferred by the community and exclusively used in the implementation. As its primary software framework Unity used *Mono*, an open source development platform based on the .NET framework. The main different between the .NET framework and the Mono framework is that mono aims to be platform independent, whereas .NET is Windows only. The major components of the Mono framework is a C# compiler, a runtime environment, the .NET class library and a Mono class library.

In Unity, all script which will be using the framework must be derived from the base class `MonoBehavior`. This class contains a lot of key functions in the Unity framework, such as `Start()` and `Update()`. The following are important concept in Unity:

6.1.1 Scene

A Scene is the top level "container" of objects in the application and is equivalent to a level in a game. An application can thus be divided into

several scenes, e.g one scene might be a main menu, while another might be a tutorial stage. In the design review application there is currently only one scene, which is on the tanker model.

6.1.2 Game Objects

A GameObject is perhaps the most important concept in the Unity editor and is essentially a generic container. Everything in the scene is, or belongs to, a GameObject. What defines a GameObject's function is its *components*, which essentially are properties of the GameObject. In Unity there are several built ones, e.g camera components, light components etc. The most important GameObject component is perhaps the `Transform` component, which is a mandatory component (i.e every GameObject has one) and defines a GameObject's position, rotation and scale in the scene. Scripts, i.e custom code, is also commonly used as a component of a GameObject.

A GameObject can be assigned a *Tag*, which serves as useful GameObject categories. This can be especially useful in a script logic setting when, e.g. a collision between two objects occur and one wants to find out what kind of objects were involves (e.g. maybe a "Player" GameObject collides with a "Coin" GameObject, which is to be collected when this occurs).

6.1.3 Prefabs

In Unity a Prefab is a type of GameObject template or blueprint, an is useful when the same assets is used multiple times (e.g. multiple copies are present in the scene). Just as a class can instantiate objects, multiple objects can be instantiated from one prefab. Any edits made to a prefab asset are thus immediately reflected in all instances produced from it, but any edits or overrides to the instances will be treated individually.

6.2 Leap Motion - A Vision-based Gesture Recognition Device

The latest technological breakthrough in gesture-sensing devices has come in the form of a Leap Motion Controller (Leap Motion, San Francisco, CA, United States). The controller, approximately the size of a box of matches, allows for the precise and fluid tracking of multiple hands, fingers, and small objects in free space with sub-millimeter accuracy (Guna et al., 2014). This chapter is based on the Leap Motion Controller documentation for the Orion software (i.e version 3.2 of the Leap motion software), and aims to highlight the important conceptual foundation for using the Leap Motion Controller in this thesis' design review application.

6.3 Physical properties

The Leap Motion Controller (see fig. 4.3 on page 27 and 6.1 on the next page) contains two stereoscopic cameras, with a field of view of

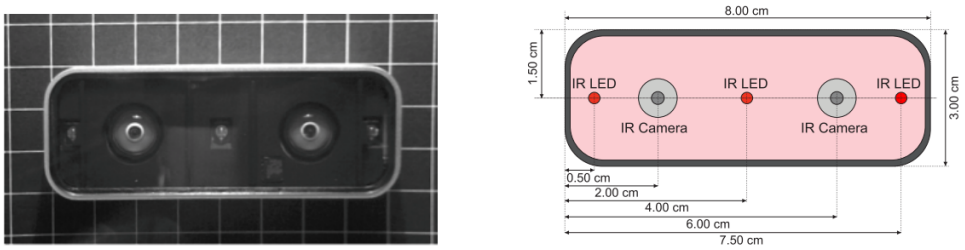

Figure 6.1: Visualization of a Leap Motion Controller, with Infrared Imaging (left) and a Schematic View (right) (Weichert et al., 2013).

about 150 degrees, in addition to three infrared LEDs. These infrared lights periodically emit light pulses with a wavelength of 850 nanometer, and thus outside the visible light spectrum. During the light pulses, which light up about eight cubic feet in front of the controller, grayscale stereo images are captured by the cameras and sent to the Leap Motion tracking software (Colgan, 2016). This image capturing has an effective range from approximately 25 to 600 millimeters above the device. In the software, the images are analyzed to reconstruct a 3D representation of what the device sees, compensating for static background objects and ambient environmental lighting. The Leap Motion software combines this sensor data with an internal model of the human hand to help cope with challenging tracking conditions.

6.4 The Leap API

The controller itself can be accessed and programmed through high level Application Programming Interfaces (APIs), with support for a variety of programming languages, including C++, C#, Objective-C, Java, JavaScript and Python. Although the API is programmed almost exclusively in C, access through a variety of other languages is achieved by virtue of various "wrapper libraries", which exposes and translates functions from their respective languages into the corresponding C function[cite]. In addition to this, the Leap Motion SDK also features integration with commercial game engines such as Unity and the Unreal Engine (Guna et al., 2014). This section will cover important concepts in the Leap API, which are thoroughly used in the thesis implementation.

6.4.1 Integration with the Unity editor

To use Leap Motion in a Unity project one simply import the Leap Motion Asset Package, which included plugin files, as well as hand prefabs, scripts and demo scenes, into the project and includes certain components to the scene (the most important being the LeapHandController and HandModels prefabs). LeapHandController is responsible for representing the Leap Motion device in the Unity scene, while the HandModels prefabs are model of the virtual hands that are to mimic what the user's hands are

doing.

6.4.2 The hand abstractions

The Leap Motion API offers many convenient abstractions for relevant properties when detecting and tracking hands. Hands are the main entity tracked by the Leap Motion controller, and it maintains an inner model of the human hand and validates the data from its sensors against this model. This allows the controller to track finger positions even when some fingers are not visible from the Leap Motion Controllers point of view.

The Hand class represents a physical hand detected by the Leap, and is perhaps one of the most central abstractions in the Leap Motion API. A Hand object provides access to lists of its pointables as well as attributes describing the hand position, orientation, and movement. Each hand-object have object-representations for its fingers, palm etc, each with its own data. One common way to access the hands are through the Frame object, which is an object-oriented representation of the last captured frame of the device. Each frame will contain a list called "hands", which will contain a hand-object per detected and tracked hand. These hands have their own characteristics, which are handily available to the developer. Some examples (from Colgan (2016)) of what variables the hand objects contain include:

- isRight, isLeft — Whether the hand is a left or a right hand.
- Palm Position — The center of the palm measured in millimeters from the Leap Motion origin.
- Palm Velocity — The speed and movement direction of the palm in millimeters per second.
- Palm Normal — A vector perpendicular to the plane formed by the palm of the hand. The vector points downward out of the palm.
- Direction — A vector pointing from the center of the palm toward the fingers.
- grabStrength, pinchStrength — Describe the posture of the hand.
- Motion factors — Provide relative scale, rotation, and translation factors for movement between two frames.

Below is an example derived from the MovementController.cs class in the Design Review implementation (in C#). This examples highlights how hand-object can be acquired from the frame-object, how we can e.g. make sure its a left-hand before proceeding, and how we can calculate a new player model position based on the hand position offset from the gesture origin. Note that this code is incomplete and only meant as a somewhat compact example:

```
//Update() runs every frame (typically between 30 - 120 times per second)
void Update()
{
    Frame frame = LeapBehavior.getLastFrame();
    iBox = frame.InteractionBox; //Used for normalization
    for (int i = 0; i < frame.Hands.Count; i++)
    {
        Hand hand = frame.Hands[i];

        if (hand.IsLeft && leftHand.getGestureType() != HandState.NONE)
        {
            //Measure hand position from palm position
            Vector leapPoint = hand.StabilizedPalmPosition;

            //Converting from right hand to left hand coordinate convention
            leapPoint.z *= -1.0f;

            //Normalizing the point
            Vector normPoint = iBox.NormalizePoint(leapPoint, false);

            if (gestureHand.getGestureType() == HandState.PALM_DOWN)
            {
                //PALM_DOWN is the gesture to navigate up and down the y-axis
                //The y-axis hand offset from origin:
                float y_offset = normPoint.y -
                    gestureHand.GetGestureOriginPosition().y;

                //Calculate new player model position
                transform.position += transform.up * speed * y_offset *
                    Time.deltaTime;
            }
        }
    }
}
```

Table 6.1: Accessing the Leap Motion Frame objects

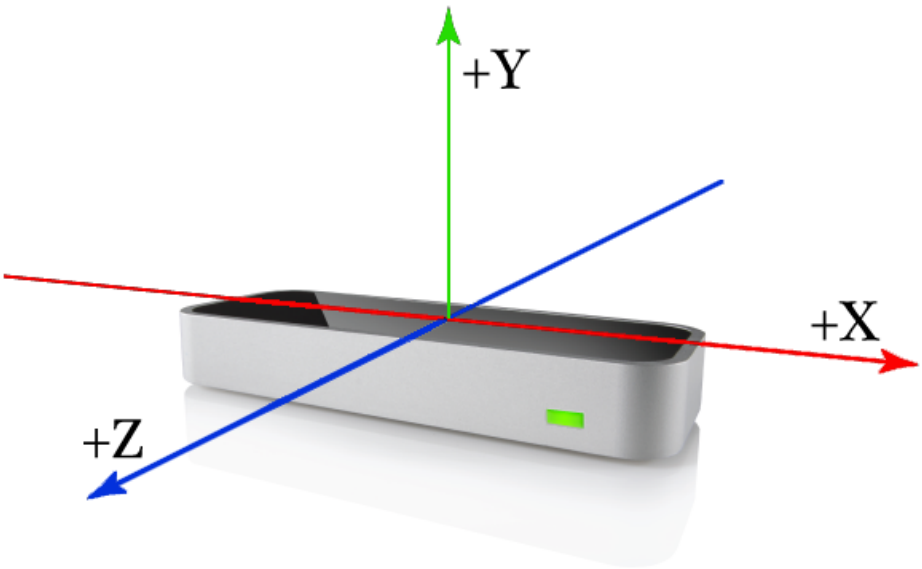

Figure 6.2: The Leap Motion Coordinate System has its origin between the two cameras.

6.4.3 The coordinate system

The Leap Motion API enables acquisition of the recognized object's position through Cartesian and spherical coordinate systems, which are used to describe positions in the controller's sensory space (Guna et al., 2014). The hand positions above the Leap Motion device are given as three dimensional vectors on the form {x, y, z}, with origin being in the center of the Leap Motion surface (see 6.2) (Colgan, 2016). Positional information, like the position of a hand, or the position of the tip of a finger, can be access in various ways. One way is to access the hands through the Frame-object, and then find the relevant hand, palm, finger or finger-joint.

The Leap Motion API uses a right-handed coordinate convention, meaning that when the user is positioned in front of the Leap Motion Controller the x-axis grows more positive towards the right, the y-axis grows more positive upwards and the z-axis grows more positive towards the user (see 6.2). As frameworks like that of Unity uses a left-handed convention for its coordinate system, i.e that the z-axis grows more positive away from the user instead of towards, the Leap Motion API also does an appropriate convention to adhere to its software environment. The Leap Motion API also adhere to differences in units, as e.g Unity uses a default unit of meters, while the Leap Motion uses millimeters (Colgan, 2016).

6.4.4 The detection utilities

To provide a common and high level interface to recognize gestures the Leap Motion API offers several detection utilities called ***detectors***. Detectors are scripts in the core asset package that serve as basic building blocks for hand action detections, and can e.g. detect whether a certain

finger is extended or not or which way the palm is facing (Colgan, 2016). New detectors can also be created by the developer by extending the Detector base class and implement logic that calls "Active" when the detector turns on and "Deactivate" when it turns off.

Several of these detector can be chained together using a ***Logic Gate*** to create more complex expressions. The Detector Logic Gate is itself a detector that logically combines two or more other detectors, using operations like AND, OR, NAND (not AND) and NOR (not OR), to determine its own state. If one thus were to make a thumb's up-gesture, one could use a logic gate with an AND-configuration together with a detector for detecting whether or not the thumb is extended and a detector for determining whether or not the thump if facing upward (Colgan, 2016).

The detectors also have some public variables, which can be adjusted for preference. These include "Period", which determines how often the detector checks the hand state, "Hand Model", which refers to which hand model is being observed and several "On and off values", which sets the thresholds for when the detector should be on (i.e the detector recognizes the property it's looking for) or off. The latter is can especially be the subject of repeated adjustment, as it is deemed crucial to find a good compromise between the two values (more on this in the implementation chapter). The detectors also has some public functions, with two of the most important ones being "OnActivate" and "OnDeactivate". OnActivate is called by its detector when the detector turns on (activates), while OnDeactivate is called when it turns off (deactivates).

This outlines the primary means of creating gestures and connecting them to actions using the Leap Motion API. One can create gesture expressions, like the thumb's up-gesture described above, using a Logic Gate with AND. This Logic Gate will only be active (on), while all of the detectors it references ("is hooked up to") are active, so in our example only when the thumb is extended AND facing upwards. By then assigning a custom created function, e.g. a function called "Accept", to the Logic Gate's OnActive-function we ensure that this function is called only when the thumb's up-gesture is done correctly.

Chapter 7

Implementation of the Application

7.1 External Assets

The Design Review application was implemented by using several premade assets. When the application is started user is positioned into a oiltank model, which was provided by DNV GL. This model is of high fidelity and was originally developed for the DNV GL Survey Simulator, an application to train surveyors.

The application also makes use of several "best practice" assets from Leap Motion, Oculus VR and SteamVR to ensure that these devices function as optimally as possible. From Leap Motion the `LeapHandController` is utilized, which is a prefab (gameObject), with several important scripts attached to it. Leap motion provided hand models are also being used, which was provided from the Leap Motion Hands-module. More specifically the `RiggedPepperCutHands` were used, but any other of the hand models could be used as easily.

From Oculus VR two prefabs is used. The first is `OVRCameraRig`, which is the recommended camera setup for using the Oculus Rift HMD. This prefab sets several important settings to ensure that both the head tracking and visual performance is as optimal as possible. The second prefab which is used is one called the `GazePointerRing`. This was showcased in a demo unity implementation by OculusVR and is essentially a cursor that exist in the game world a fixed length in front of the user. As regular crosshair (which are drawn directly on the screen space) isn't allowed in VR (more about this later), the `GazePointerRing` serves as a crosshair. From SteamVR the `[CameraRig]` prefab is used, which essentially does the same configurations as the `OVRCameraRig` does, but with the HTC Vive HMDs in mind.

The Design review application also makes use of Hover UI Kit, an open source project for creating VR/AR-enabled, customizable and dynamic user interfaces. This kit was vital in rapidly prototyping a gesture-enabled menu and a virtual keyboard to the annotation forms.

The Unity project has four top-level game objects: `EventSystem`,

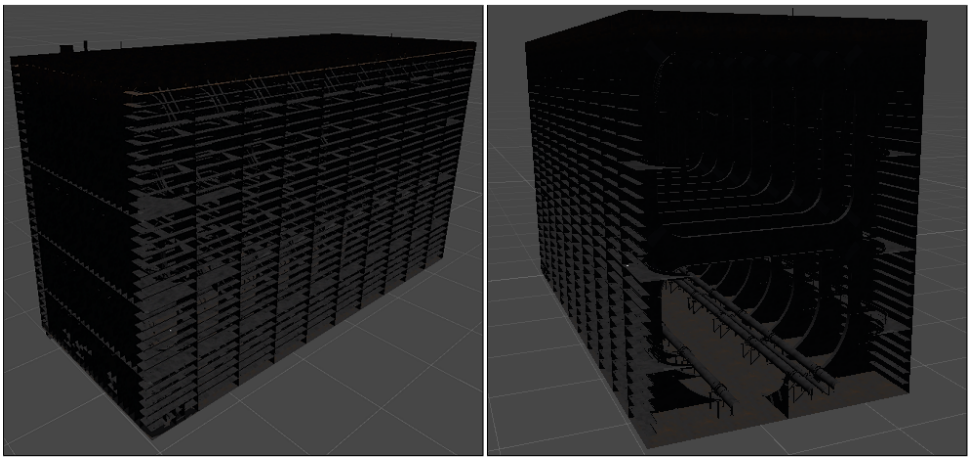

Figure 7.1: The Oil tank model from the outside

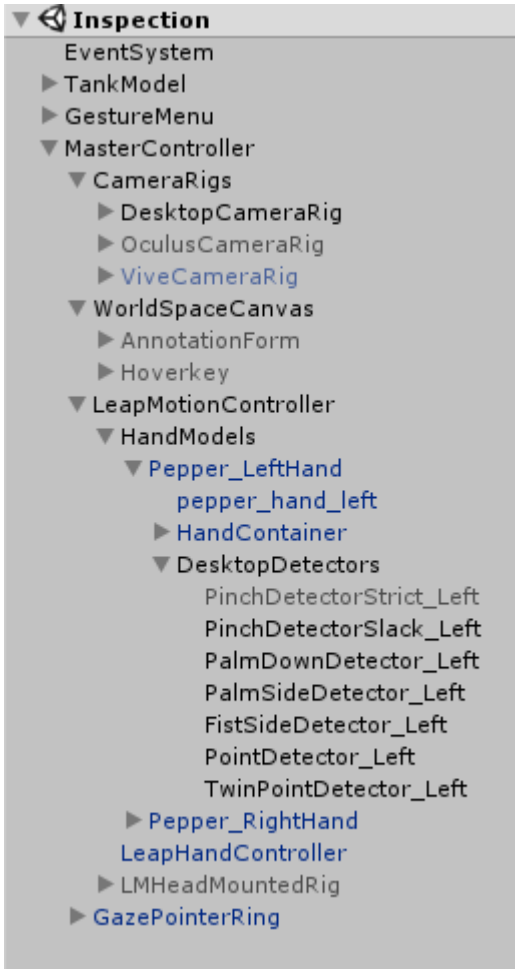

Figure 7.2: The Unity project hierarchy of the Design Review Application

TankModel, GestureMenu and MasterController.

The EventSystem game object is responsible for processing and handling events and input actions in the scene. In this implementation a standard unity event system is used (generated when creating a new systems), with some modifications done for virtual reality. Most of these modification are accomplished by the OVRInputModule script, which inspired by an Oculus VR sample project. Apart from this no other changes was done to EventSystem as it worked optimally "right out of the box".

The TankModel game object contains several child objects that together make up the oiltank-model, which this application is based on. Originally the design included the functionality to load different models into the application and starts sessions, but this was fell out of scope to both prioritize the gesture recognition and virtual reality aspect, and because of a low availability of similar models. The tank model has not received any changes during implementation and has been used as it was supplied.

The GestureMenu and MasterController together contain most of the key components of the application and will, with their relevant child objects, be the main focus for the rest of this chapter. GestureMenu represents the gesture menu, which by default is available by directing the left hand palm in the direction of the camera. The MasterController game object represents the player model and contains many of the most important game objects, in addition to holding many key scripts. The MasterController's transform, with its position, rotation and scale, represents the user's position and orientation, and every child object of MasterController will have a position, rotation and scale that is relative to its own. This ensures that e.g. the camera will always "follow" the user. Several of the important game objects that are covered in later sections are children of the master controller for this reason. First, however, we will cover the important components of MasterController.

7.2 The Master Controller Components

The master controller represents a collection of controllers, which all have the role of handling input from the user and translate it into the correct action. These controllers typically interact with the GestureHand class to check if a certain hand state-criterion is met, utilize its utility functions and check for changes every frame. The GestureHand will be covered more in depth in section 7.6 on page 70, but can in short be described as a class which is instantiated once per hand and keeps track of what potential gesture the hands are performing, the origin coordinates for the potential gestures and other useful hand-specific information.

Another important component in the controllers, with the AnnotationFormController as an exception, is how they all relies on the Update-function, which is called in a set order by the Unity runtime environment on every frame. During the update call these controllers checks for input relevant for its purposes. The following four sections will cover the controllers that the master controller contains.

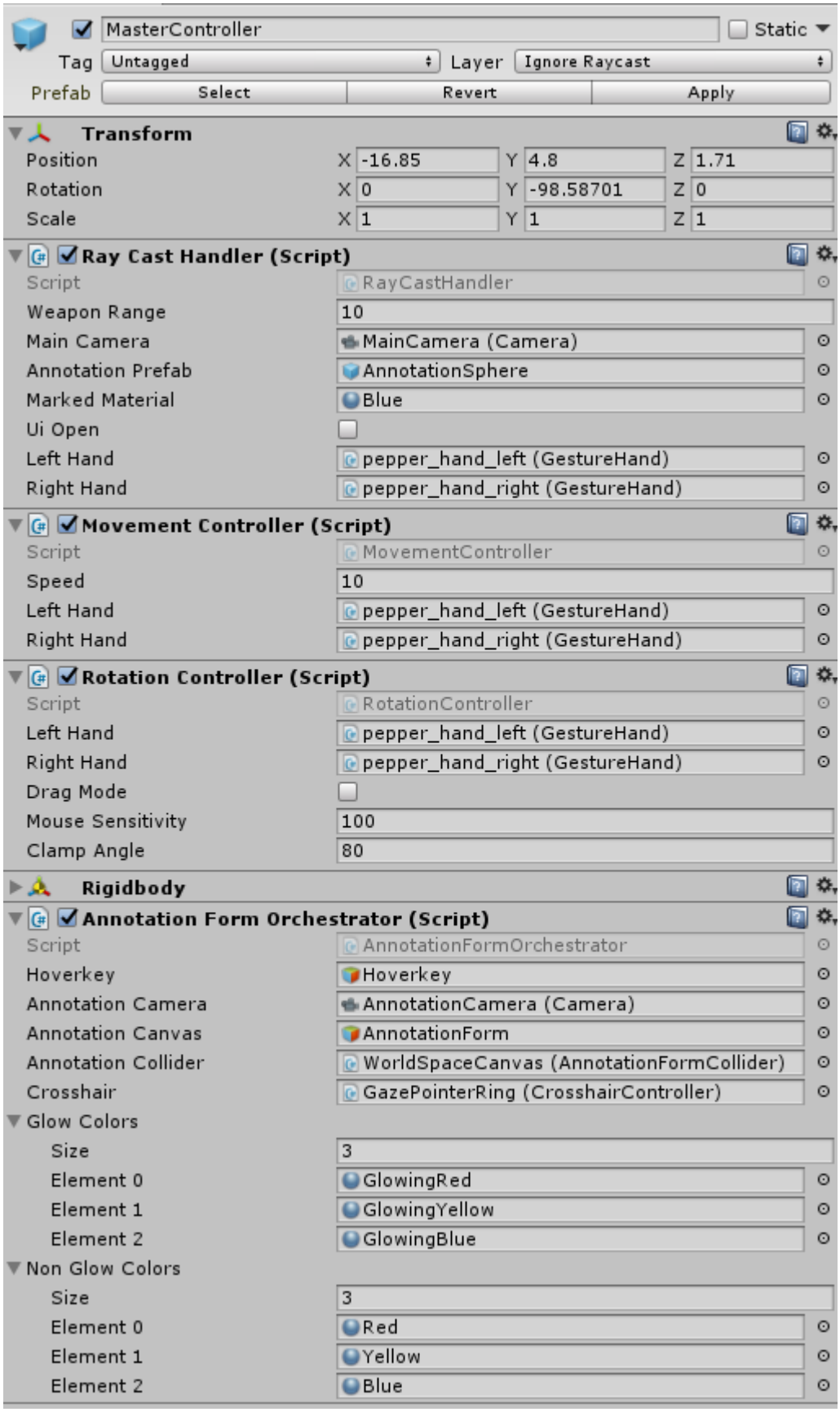

Figure 7.3: The `MasterController` components seen in the Unity Inspector view.

7.2.1 The Rotation Controller

The `RotationController` is a script component of the `MasterController` and its primary function to handle user input related to rotation. `RotationController` contains a number of instance variables, which will be described below. Some of these variables have a public access modifier, as this allows their values to be seen and edited from the Unity Inspector view (see figure 7.3 on the preceding page for an example). Variables that does not have this requirement, and is which should not be accessed from other parts of the application, are given a private access modifier.

- `public GestureHand leftHand, rightHand` - Stores the `GestureHand` instances that represents the left and right hand.
- `public float sensitivity` - A float-point multiplier that determines the sensitivity of the rotational actions. All statements that rotates the camera is multiplied by this variable's value, which by default is 100.0.
- `public float clampAngle` - An absolute value in degrees for the maximum rotation allowed along in y-axis. Its default is 90.0 (degrees), meaning that the user can rotate from looking straight ahead (0.0 degrees on the y-axis) to straight up (90.0 degrees on the y-axis) and straight down (-90.0 degrees on the y-axis). Note that this rotational limitation along the y-axis is in place to prevent the user from rotating the cameras (and the enire `MasterController`) "upside-down".
- `private float rotX, rotY` - The floatpoint variables rotX and rotY are intermediate values that stores the rotation of the `MasterController` (and thus the cameras) and is used to calculate the new rotation quaternion that is applied to the `MasterController`'s transform every frame. Note that `RotationController` only allows rotation along the x-axis (left-and-right) and y-axis (up-and-down), and not along the z-axis (a "barrel roll" rotation).
- `private InteractionBox iBox` - The `InteractionBox` class is a Leap Motion abstraction for the area (i.e the "box") above/in front of the Leap Motion device that is interactable (i.e where the device can detect and track the hands), and is used for normalizing purposes. This variable holds a reference to the latest InteractionBox-object, which is updated on every Leap Motion-frame by retrieving it from the Leap Motion `Frame` object (see table 7.8.2 on page 81 for an example).

During its update function `RotationController` both checks for mouse movements and whether the pinch gesture is performed by either hand.

Mouse movements are retrieved by calling the unity-function `Input.GetAxis(string axisName)` once per axis (i.e twice to get the x- and y axis). The return value of this function is in the range -1 and 1 for each axis, which

```
public float sensitivity = 100.0f;
private float rotX, rotY = 0.0f;

void Update() { // Update is called every frame
    trackMouse();
    [...]

    // Create new rotation quaternion and replace the rotation of the
    // MasterController's transform.
    Quaternion localRotation = Quaternion.Euler(rotX, rotY, 0.0f);
    transform.rotation = localRotation;
}

private void trackMouse() {
    float mouseX = Input.GetAxis("Mouse X"); // Get x-axis mouse movement
    float mouseY = -Input.GetAxis("Mouse Y");// Get y-axis mouse movement

    rotY += mouseX * sensitivity * Time.deltaTime; // Transform mouse
        movements
    rotX += mouseY * sensitivity * Time.deltaTime; // to rotations
}
```

Table 7.1: How mouse movement is captured and transformed to rotations.

signifies in what direction the mouse is moving and at what speed. If `Input.GetAxis("Mouse X")` e.g. returns "-0.01" the mouse is moving very slowly to the left, while "1.0" would mean that is moves as fast as possible to the right. If calling GetAxis with the x-axis and the y-axis both yield 0, then the mouse is not moving and no rotation is performed.

Before applying the captured mouse movements to the rotation it is multiplied by the sensitivity variable outlined above, and by `Time.deltaTime`. `Time.deltaTime` is another function in the Unity framework and returns the time in seconds it took to complete the last frame. By applying this to the equation we make the calculation frame rate independent and essentially expresses that we want to rotate X amount per second, instead of X amount per frame.

The pinch gesture, which is used for rotation, is captured in a similar fashion. The update function checks for the pinch state, and if its detected it tracks the hand position within the interaction box (i.e the area above/in front of the Leap Motion device). Just as the variables `mouseX` and `mouseY` is captured in table 7.2.1 the variables `handX` and `handY` are captured, but somewhat differently. These two variables are calculated by obtaining the hand's palm position within the interaction box and subtracting the origin point, i.e the point were the current pinch gesture started (see table 7.2.1 on the facing page). After this is done they are, like with the mouse movement, multiplied by the `sensitivity` variable and `Time.deltaTime`.

```
public GestureHand leftHand, righthand;
public float sensitivity = 100.0f;
private float rotX, rotY = 0.0f;

void Update() {
    trackMouse();

    bool leftPinch = false, rightPinch = false;

    if (leftHand.getGestureType() == GestureHand.HandState.PINCH)
        leftPinch = true;
    if (rightHand.getGestureType() == GestureHand.HandState.PINCH)
        rightPinch = true;

    if (leftPinch || rightPinch) {
        Frame frame = LeapBehavior.getLastFrame();
        iBox = frame.InteractionBox;
        for (int i = 0; i < frame.Hands.Count; i++) {
            Hand hand = frame.Hands[i];

            if (hand.IsLeft && leftPinch)
                TrackPinch(hand, leftHand.GetGestureOriginPosition());

            if (!hand.IsLeft && rightPinch)
                TrackPinch(hand, rightHand.GetGestureOriginPosition());
        }
    }

    // Create new rotation quaternion and replace the rotation of the
    // MasterController's transform.
    Quaternion localRotation = Quaternion.Euler(rotX, rotY, 0.0f);
    transform.rotation = localRotation;
}

private void TrackPinch(Hand hand, Leap.Vector originCoordinates)
{
    // Get position, convert from right to left hand coordinates, normalize
    Leap.Vector leapPoint = hand.StabilizedPalmPosition * -1.0f;
    Leap.Vector normalizedPoint = iBox.NormalizePoint(leapPoint, false);

    float handX = normalizedPoint.x - originCoordinates.x; // Find x-offset
    float handY = normalizedPoint.y - originCoordinates.y; // Find y-offset

    // Transform hand movement to rotation
    rotX += -handY * sensitivity * Time.deltaTime;
    rotY += handX * sensitivity * Time.deltaTime;
}
```

Table 7.2: How the pinch gesture is captured and transformed to rotations. Note that the implementation code looks somewhat different (e.g. Some shorter variable names in this table.)

7.2.2 The Movement Controller

The `MovementController` is another script component of the `MasterController`, and relates to the movement of the user. This controller has many similarities with the `RotationController`, covered in the previous section, as it has very similar code for hand detection in the update method. If one of the hands have a different state than NONE, the `TrackHandMovement` function (see table 7.2.2 on the next page) is called. In this function the hand position is obtained and a number of conditions are checked for:

1. Is the combined gesture scheme used AND is the palm-down gesture active?
2. Is the combined gesture scheme NOT used AND is the palm-down gesture active?
3. Is the combined gesture scheme NOT used AND is the palm-side gesture active?
4. Is the combined gesture scheme NOT used AND is the fist gesture active?

If any of these criteria are met the appropriate action is taken. For the last criterion this would mean going forward by X, where X is positive or negative float point number, and is the result of (current coordinates - origin coordinates). Also note that although only one of these cases can trigger for each hand, the user can perform different gestures simultaneously using both hands, and thus potentially meet more of these criteria at the same time.

As `MovementController` also support keyboard usage some keys are also tracked, primarily by using the two different unity built-in functions `Input.GetAxis` and `Input.GetButton`. The former of these is used for movement along the x- and z-axis (left/right and forward/backward) and returns a continuous number in the range <-1, 1>, while the latter is used for the rest of the actions and returns a discrete number. `Input.GetAxis` is used in a similar fashion as mouse movement was captured, but instead using the arguments "Vertical" for the x-axis and "Horizontal" for the z-axis. The movement calculations are similar to those found in `TrackHandMovement` (see table 7.2.2 on the facing page), only substituting or removing the hand position calculations (e.g. `normalizedPoint.x - gestureHand.GetGestureOriginPosition().x`).

What key each movement corresponds to is configured in the Unity Input Manager, which is a build specific configuration that allows the user to customize the controls at runtime. This also gives an abstraction layer for the developer, where a virtual "input name", e.g. "down", is specified instead of which physical key that needs to be used (e.g. `KeyCode.DownArrow`). If the latter approach is preferred, Unity also offers the function `Input.GetKey(Keycode)`. With this function the developer must specify a "physical" and "actual" key instead of a virtual one. This means that the code `Input.GetKey(Keycode.Q)` always listen for the

```
private void TrackHandMovement(Hand hand, GestureHand gestureHand)
{
    // Get position, convert from right to left hand coordinates, normalize
    Leap.Vector leapPoint = hand.StabilizedPalmPosition * -1.0f;
    Leap.Vector normalizedPoint = iBox.NormalizePoint(leapPoint, false);

    // True if movement along the x-, y- and z-axis are handle by one
        gesture instead of three, false otherwise.
    if (gestureHand.getCombineGestures())
    {
        if (gestureHand.getGestureType() == HandState.PALM_DOWN)
        {
            transform.position += transform.up * speed * (normalizedPoint.y
                - gestureHand.GetGestureOriginPosition().y) * Time.deltaTime;
            transform.position += transform.right * speed *
                (normalizedPoint.x -
                gestureHand.GetGestureOriginPosition().x) * Time.deltaTime;
            transform.position += transform.forward * speed *
                (normalizedPoint.z -
                gestureHand.GetGestureOriginPosition().z) * Time.deltaTime;
        }
    }

    else if (gestureHand.getGestureType() == HandState.PALM_DOWN)
        transform.position += transform.up * speed * (normalizedPoint.y -
            gestureHand.GetGestureOriginPosition().y) * Time.deltaTime;

    else if (gestureHand.getGestureType() == HandState.PALM_SIDE)
        transform.position += transform.right * speed * (normalizedPoint.x -
            gestureHand.GetGestureOriginPosition().x) * Time.deltaTime;

    else if (gestureHand.getGestureType() == HandState.FIST)
        transform.position += transform.forward * speed * (normalizedPoint.z
            - gestureHand.GetGestureOriginPosition().z) * Time.deltaTime;
}
```

Table 7.3: How movement gestures are detected and handling in `MovementController`

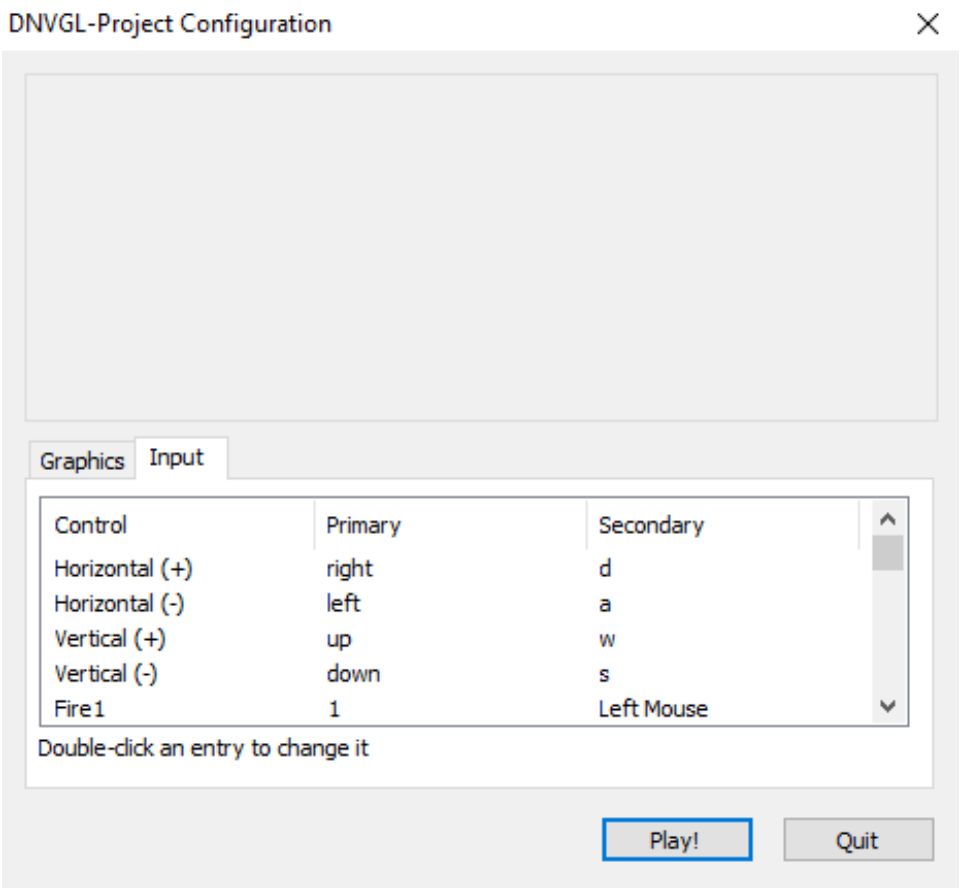

Figure 7.4: The Unity Input Manager enables the user to specify which input keys to use for certain actions when starting up the application

Q-button on the keyboard and never any other. Because of the flexibility of using the Input Manager this implementation uses `Input.GetAxis` and `Input.GetButton` for all keyboard input.

In this implementation the default keys for actions handled by the `MovementController` class are as follows: "Left" or "A" for left (-horizontal), "right" or "D" for right (+horizontal), "up" or "W" for going forward (+vertical), "down" or "S" for going backward (-vertical), "Q" for going up (+altitude), "E" for going down (-altitude), "Left Shift" for increasing movement speed and "Left Ctrl" for decreasing movement speed. The movement speed is a multiplier that is applied to every movement (i.e the ones mention in this section), and it is increased or decreased in increments of 0.5.

7.2.3 The Raycast Controller

The `RaycastController` is another script component of the `MasterController`, and is responsible for detecting certain input from the user, create raycast-beams and handle when a raycast hits an object. When activated a raycast beam should be fired from the position of the `MasterController` and to-

```
if singlePointGesture OR leftMouseClick is active

    objectWhichWasHit = createRayCastBeam();
    if objectWhichWasHit is a point annotation
        editTheHitPointAnnotation();
    else
        // Place a point annotation on the object,
        // even if its on top of an object annotation
        CreatePointAnnotation();

else if doublePointGesture OR right-click-mouse is active

    objectWhichWasHit = createRayCastBeam();
    if objectWhichWasHit is an object annotation
        editTheHitObjectAnnotation();

    else if (objectWhichWasHit is NOT a point annotation
        // Object annotation of a point annotation is not allowed.
        CreateObjectAnnotation();

else
    Go check for other stuff!
```

Table 7.4: Pseudo code for the raycast scenarios

wards where the `MasterController` is oriented ("toward were the camera is located"). To enable the user to aim the raycast properly a crosshair or cursor (depending on the active camera rig) is present in the middle of the users field of view, just like in a first-person shooter game. If a raycast is created and collides with a part of the model (i.e an object with a collider that is within range), one of the following cases occur:

1. A point annotation is created at the collision point (where the raycast hit the object).
2. The object the raycast hit gets an annotation attached to it and have its material changed to indicate that it is annotated.
3. The annotation form because active and edits the target point annotation.
4. The annotation form because active and edits the target object annotation.

These scenarios and which one of them is triggered can be summarized with the following pseudo code:

Note that a raycast beam can be created by a single-point gesture or "Fire1" (a input name in the Input Manager that defaults to the left mouse

button), or by a double-point gesture or "Fire2" (right mouse button by default). The logic for these to scenarios are handled independently as seen in the pseudo code in table 7.2.3 on the previous page.

The `RayCastController` contains the following instance variables:

- `public float raycastRange` - The raycasts maximum range, meaning the user has to be within N distance from an object for the raycast to hit it. Default is 50.0f.
- `public Camera mainCamera` - A reference to the main camera of the active camera rig.
- `public GameObject annotationPrefab` - A reference to the point annotation prefab. This serves as a blue print for new point annotations.
- `public Material markedMaterial` - The initial material to use for marking object annotations.
- `public bool uiOpen` - This variables tells whether or not the annotation form is open (i.e active and visible to the user).
- `public GestureHand leftHand, rightHand` - References to the left and right hand's `GestureHand` instances.
- `private int annotationCounter` - A count of active point annotation in the scene. This variable is used to assign annotations IDs upon their creation.
- `private bool annotationPlacedDuringGesture` - Whether or not a point annotation or an object annotation was placed during the current active gestures.
- `private AnnotationFormController orchestrator` - A reference to a singleton `AnnotationFormController`. This controller is described more in the next section.

The update function checks for button clicks ("Fire1" and "Fire2") and gestures ("SINGLE_POINT" and "DOUBLE_POINT"), each with its own additional conditions. If a button is clicked the value of `uiOpen` needs to be false. This is because we don't want to fire a raycast when the user interface (i.e the annotation form) is open (i.e when `uiOpen = true`). Gestures work a little differently as the detectors are disabled when the annotation form is open, and we thus don't have to check for the value of `uiOpen`. Instead, the gesture checks whether the value of `annotationPlacedDuringGesture` is false before proceeding. This is to prevent an issue where the user performs a single- or double point gesture and the annotation is both created and opened for editing instantaniously (which is again the desing). An annotation could be placed by using one of the point-gestures, and on the next frame, were the same gesture is presumably still present, a new raycast would have been fired off,

```
private void CreateRaycastBeam(bool isObjectAnnotation)
{
    // Create a vector at the center of our camera's viewport
    Vector3 rayOrigin = mainCamera.ViewportToWorldPoint(new Vector3(0.5f,
        0.5f, 0.0f));

    // RaycastHit stores information about what our raycast has hit
    RaycastHit hit;

    // Check if our raycast has hit anything
    if (Physics.Raycast(rayOrigin, transform.forward, out hit,
        raycastRange))
    {
        string tag = hit.collider.gameObject.tag;
        if (tag.Equals("PointAnnotation") || tag.Equals("ObjectAnnotation"))
            {
            if (tag.Equals("PointAnnotation") && !isObjectAnnotation)
                EditAnnotation(hit, isObjectAnnotation);
            else if (tag.Equals("ObjectAnnotation") && isObjectAnnotation)
                EditAnnotation(hit, isObjectAnnotation);
            else if (!isObjectAnnotation)
                CreatePointAnnotation(hit);
        }
        else if (isObjectAnnotation)
            CreateObjectAnnotation(hit);
        else // Else create a new annotation
            CreatePointAnnotation(hit);
        annotationPlacedDuringGesture = true;
    }
}
```

Table 7.5: How the `CreateRaycastBeam` function of the `RaycastController` works.

htting the annotation placed one frame ago and opening the annotation form to edit it. As `annotationPlacedDuringGesture` is set to true when an annotation is placed, and the gesture (detector) becomes inactive, it circumvents this issue.

All the actions mentioned above calls the function `CreateRaycastBeam(bool isObjectAnnotation)`, which creates the raycast and applies the logic outlines by the pseudo code in table 7.2.3 on page 61. "Fire1" and "SINGLE_POINT" calls this with the argument `false` and "Fire2" and "DOUBLE_POINT" calls it with `true`.

The code in table 7.2.3 on the next page also shows how point- and object annotation are created (editing of these is handled by the `AnnotationFormController`, which is reviewed in the next section). In the function `CreatePointAnnotation` and `CreateObjectAnnotation` the game object which was hit, and the point where the hit occured (i.e where the raycast collided with the object), is accessable through the `RaycastHit` object, and creating annotations then simply becomes a matter of either instantiating a prefab or attaching an annotation, in to "tagging" them as annotation.

```
private void CreatePointAnnotation(RaycastHit hit)
{
    //Create an annotation object at hit's coordinates.
    GameObject newAnnotation = (GameObject) Instantiate(annotationPrefab,
        hit.point, Quaternion.identity);
    newAnnotation.tag = "SphereAnnotation";
    newAnnotation.name = "SphereAnnotation_" + annotationCounter++;
    AnnotationInformation info =
        newAnnotation.GetComponent<AnnotationInformation>();
    info.initializeMaterials();
    info.text = "Please enter your notes";
    info.annotationSphere = newAnnotation;
}

private void CreateObjectAnnotation(RaycastHit hit)
{
    GameObject targetObject = hit.collider.gameObject;
    targetObject.tag = "ObjectAnnotation";
    AnnotationInformation info =
        targetObject.AddComponent<AnnotationInformation>();
    info.initializeMaterials();
    info.changeMaterial(markedMaterial);
}
```

Table 7.6: How the `CreatePointAnnotation` and `CreateObjectAnnotation` functions in the `RaycastController` works. Note that annotation editing is handled by the `AnnotationFormController`, which is reviewed in the next section.

7.2.4 The Annotation Form Controller

The `AnnotationFormController` performs various task when entering or exiting the annotation form, in addition to hosting many of its button listeners. The `AnnotationFormController` contains the following instance variables:

- `public GameObject hoverkey` - A reference to the hoverkey (the virtual keyboard) game object.
- `public Camera annotationCamera` - A reference to the annotation camera.
- `public GameObject annotationCanvas` - A reference to the canvas the annotation form is drawn on.
- `public CanvasCollider annotationCollider` - A reference to the collider-script that is used for the world-space canvas.
- `public CrosshairController crosshair` - A reference to the crosshair/cursor. This is disabled when the annotation form is active.
- `public Material[] glowColors` - An array for the different annotation priority material. 0 = error, 1 = warning, 2 = info.
- `public Material[] nonGlowColors` - A similar array, with a non-glowing variant of the materials in case glow is disabled. Follows the same order as the previous array.
- `private RayCastController raycastHandler` - A reference to the raycast controller singleton which were covered in the previous section.
- `private AnnotationInformation annotationInstance` - A reference to the current annotation being edited.

The annotation form controller functions are invoked by the raycast controller when the user wishes to edit an annotation. The raycast controller then sequentially calls `EnterAnnotationForm`, `AttachActionButtonListeners` and `AttachPriorityButtonListeners` in the `AnnotationFormController` class. The `EnterAnnotationForm` function does the neccessary setup, while the latter two attach button listeners to the buttons. When the user clicks either the "Submit", "Cancel" or "Delete" buttons the `ExitAnnotationForm` function is called, which essentially does the inverse of what `EnterAnnotationForm` does.

The following setup is done by `EnterAnnotationForm` and `ExitAnnotationForm`:

1. Disable the annotation camera. This makes annotation invisible to the user.
2. Disable colliding object. This disabled (i.e temporary removes) all objects that would otherwise obstruct the world-space canvas.

3. Activate the world-space canvas so its visible to the user (the annotation form is now active).
4. Activate the hoverkey (the virtual keyboard) so its visible to the user.
5. Retrieve the annotation instance that was hit by the raycast beam.
6. Set the `uiOpen` boolean variable in the raycast controller to `true`.
7. Lock movement. This will disable the `RotationController` and `MovementController` classes so the user cant move or rotate when the annotation form is open.
8. Enable the mouse cursor, which is disabled when the annotation form isnt open. This is to allow the user to use the mouse to click on buttons.
9. Disable the crosshair so its no longer visible.
10. Disable the detectors for the left and right hand so no gestures are being recognized while the annotation form is open (using the hand to click buttons still work though).
11. Input the text that is already stored in the annotation to the annotation form text field. If the annotation is newly created this text value is blank.

As previously mentioned `ExitAnnotationForm` undoes most of these action when the user exits the annotation form.

7.3 The Camera Rigs

The `CameraRigs` game object is a direct child of the `MasterController` and holds three different game object, which each represents its own camera-setup: `DesktopCameraRig`, which is meant to be used without virtual reality, `OculusCameraRig`, meant to be used with Oculus Rift HMDs, and `ViveCameraRigs`, meant to be used with the HTC Vive. While the desktop rig uses one main camera, the virtual reality rigs (i.e the Oculus and Vive rigs) utilizes two main cameras (one per eye). These are slightly offset, by about the same length as the real-world distance between two eyes, and rendered separately. The camera rigs all utilize a separate camera for rendering only annotation objects in the scene, while the main camera(s) (one for desktop, two for VR), renders the rest. This is done by:

1. Placing the annotation objects in the scene on a different rendering layer than the rest of the objects (called the annotation-layer).
2. Have the main camera(s) render all layers, except the annotation-layer.
3. Have the annotation-camera only render objects on the annotation-layer.

4. "Combine" the output of the two cameras by drawing the output from the annotation camera on top of the output from the main camera. The main camera thus renders first and the annotation camera second. This is accomplished by giving the main camera `depth = 0` and the annotation camera `depth = 1`.

By rendering these two categories of layers independently we get some flexibility and options with regards to how to present the annotations. This will be discuss more in depth in the 7.9.2 on page 85 section.

For the application to run successfully one of these rigs should be enabled, while the other two should be disabled. This can be done by switching between the three rigs in the dropdown-menu named Rig, which is present on the `CameraRigs` game object itself and implemented in the `CameraRigSetup` script. In addition to ensuring that only the correct rig is enabled, the `CameraRigSetup` script also does several other operations. One of these is ensuring that the field of view is set to 60 degrees if the desktop rig is selected, as this can wrongfully be set to a HMD's value if a HMD is connected to the computer. When a virtual reality rig is used the field of view is set automatically by the HMD software. Another thing done by the script is to decide whether a two dimensional crosshair/cursor should be drawn on the screen space (in case of the desktop rig), or if a three dimensional crosshair/cursor (i.e the `GazePointerRing`) should be drawn in the world space.

7.4 The World Space Canvas

The `WorldSpaceCanvas` is a canvas object, which in Unity serves as a container for other user interface elements, such as buttons and input fields, and is rendered in world space. It is thus diegetic and exists there like other 3D objects.

In applications that don't utilize virtual reality, canvases and other UI elements are usually non-diegetic (i.e they don't exist within the game world), and in 2D and drawn directly to the screen space (as opposed to world space) using x- and y-coordinates. With this approach one can specify e.g. a position by its x- and y-coordinate, where {0, 0} usually represents the top-left of the display. This changes in virtual reality applications, as the user's eyes are unable to focus on the screen space. An analogy to this would be to ask the user to read a letter while holding it 2-3 centimeters from their eyes. Because of this, elements appearing on the screen space is not rendered in unity while running it with the virtual reality SDKs.

Another reason why the canvas is rendered in world space, and also the reason why this is the case in desktop mode, is because of our touch interaction. To enable the user to click on buttons using his or her hands, the user interface must also exist in world-space so a collision can occur between the desired button and the hand models (that mimic the users hand).

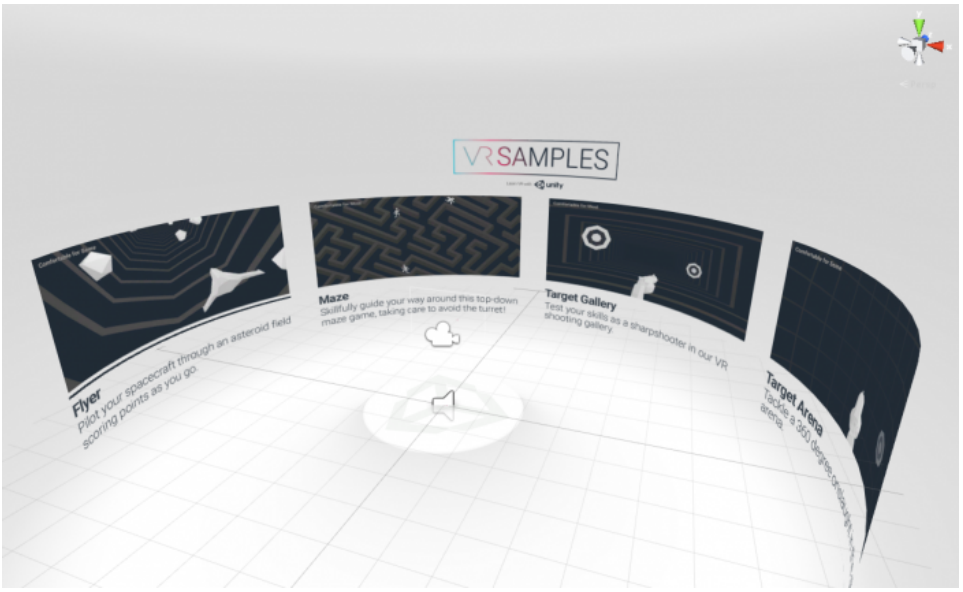

Figure 7.5: An example of world-space (diegetic) user interfaces (Unity, 2016).

`WorldSpaceCanvas` is thus rendered in the world space, and is always positioned 0.8 unity meters (i.e the virtual representation of a meter in unity) in front of the user. The game object is thus always in the center of the camera, but is only visible and enabled when the user is editing an annotation. One issue with this approach is "clipping", i.e that the annotation form visually collides with another object (e.g. the tank model or an annotation sphere) thus obstructing it from view. To combat this `WorldSpaceCanvas` has a box-shaped collider component, which covers the canvas as well as the area between the canvas and the camera, and a script called `CanvasCollider`, which keeps track of objects that's within the collider component. When the user wish to edit an annotation, and the `AnnotationForm` and `Hoverkey` becomes active, the objects within the canvas' collider is disabled, thus hiding objects that could potentially obstruct the whole, or parts of, the canvas. The objects are enabled again once the users is done editing the annotation (i.e when the user clicks "submit", "cancel" or "delete").

The `WorldSpaceCanvas` has two child game objects: `AnnotationForm` and `Hoverkey`. `AnnotationForm` currently only contains a inputfield-object and a background rectangle, but can in future iteration grow to contain other user interaction elements. The `Hoverkey` game object represents the touch keyboard and is part of the HoverUI-kit. In addition to the keyboard six other similar buttons are also present: Submit, Cancel, Delete, Error, Warning and Information (see 7.7 on the next page).

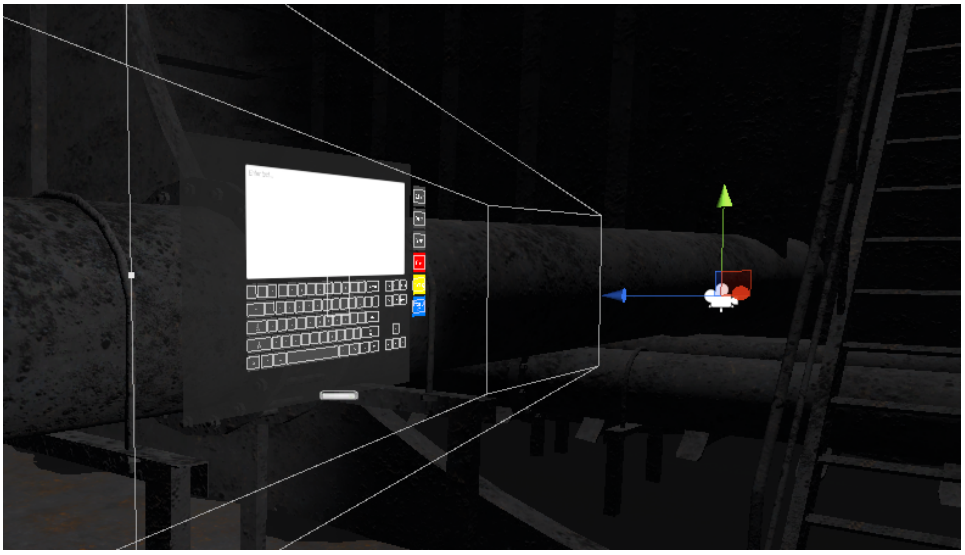

Figure 7.6: The `WorldSpaceCanvas` as seen in the Unity Scene View.

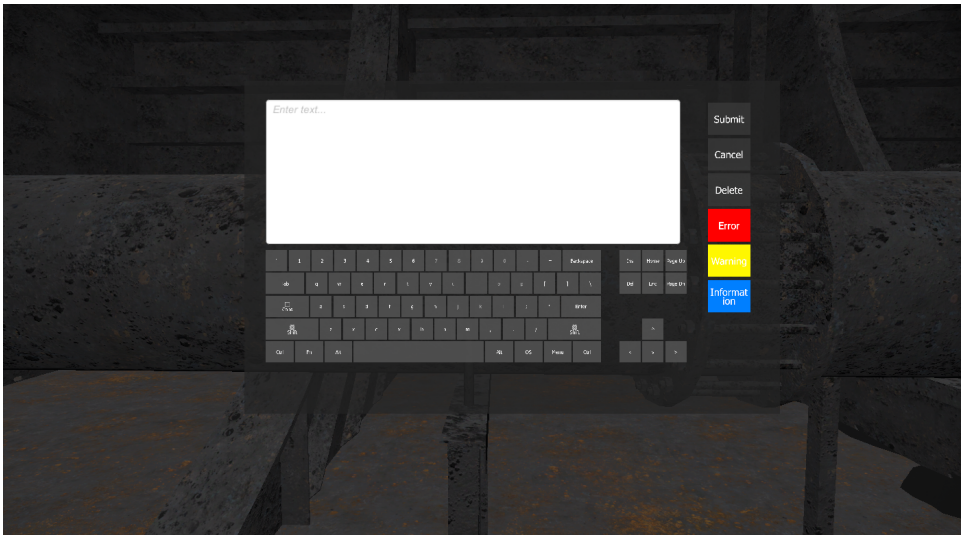

Figure 7.7: The `WorldSpaceCanvas` as seen in the Unity Game View.

7.5 The Leap Motion Controller

The `LeapMotionController` game object contains objects related to the Leap Motion device and gesture recognition and consist primarily of the hand models, necessary scrips and detectors. In the game object `HandModels` there is one object-representation of the left hand, called `Pepper_LeftHand`, and one for the right hand, called `Pepper_RightHand`. These objects have their own hand models (as there needs to be different models for the left and right hand) and their own detector objects (as a detector can/should only observe one hand). Each "hand" thus have its own list of detectors called `Detectors`. These are the definition and implementation of the gesture scheme that were discussed in 5.2 on page 36. In addition to detectors, each hand also have its own instance of the `GestureHand` class, which is an important component to keep track of hand states and resources, and it exposes utility function to other classes.

7.6 The Gesture Hand Class

The `GestureHand` class is assigned as a component to each hand object, and contain several important variables.

- `bool isLeftHand` - Keeps track of whether this instance belong to the left or the right hand.
- `GameObject handModel` - A reference to the unity game-object that contains the handModel this gesturehand instance is relevant for.
- `Material[] handMaterials` - An array of different material for the hand model. These are swapped between when e.g. a certain gesture is brecognized and tracked.
- `GestureHand otherHand` - A reference to the other GestureHand-instance. For the lefthand GestureHand-instance this variable thus point to the right hand GestureHand hand-instance.
- `GameObject detectors` - A reference to the gameobject that hold/is parent of this hands detectors.
- `bool combineGestures` - Keeps track of whether the user is using a combined XYZ axis gesture scheme or if these movement gesture are kept separate (see 5.2.5 on page 38).
- `Vector gestureOriginPosition` - Holds the x-, y- and z- coordinates of the palm when the current gesture was recognized.
- `HandState handState` - Holds one of several HandState enum values that represent the hand state. This variable has one of the following enum values: NONE = 0, PINCH = 1, PALM_DOWN = 2, PALM_SIDE = 3, FIST = 4, SINGLE_POINT = 5, DOUBLE_POINT = 6 or DISABLED = 7 (see 7.6 on the facing page).

ID	Variable name	Implication
0	NONE	No gesture is being performed.
1	PINCH	User can rotate by the y- and z-axis
2	PALM_DOWN	User can move along the y-axis.
3	PALM_SIDE	User can move along the x-axis.
4	FIST	User can move along the z-axis.
5	SINGLE_POINT	User is placing/has placed point-annotation.
6	DOUBLE_POINT	User is placing/has placed object-annotation.
7	DISABLED	The detectors are disabled and no gesture and hand state can be achieved before enabling them.

Table 7.7: The `GestureHand` class' hand states

The `GestureHand` class also contains several important functions that are called when a certain gesture is activated or deactivated. `void ActivateGesture(int gestureCode)` is called by a detector when it becomes active, i.e when its criteria are met and the gesture it represent is recognized. It then signals the `GestureHand` class with its code/signature so `GestureHand` knows which detectors called it. If a pinch gesture is detected by the left hand pinch detector the left hand `GestureHand` class' `ActivateGesture` is thus called with the argument "1". From a design standpoint one should be able to send the Handstate.PINCH enum as a value, but then this function wouldnt be exposed in the Unity Inspector interface (which only seem to accept primitive or built-in argument types).

Once this function is called it sets the hand state to the value of the gestureCode, sets the gesture origin position and switches the hand materials. Hand material switches is done by assigning the material at index `gestureCode` in the `handMaterials` list to the hand model, so if a pinch gesture is a performed the hand model is assigned the material at `handMaterials[1]`. The HandState enums and the hand material list thus follow the same sorting order.

The `GestureHand` class also has the function `void DeactivateGesture()`, which is called by a detector when it becomes deactivated. This function simply resets the hand state by setting it to HandState.NONE (NONE = 0) and assign the material at `handMaterials[(int) HandState.NONE]` (i.e 0) to the handmodel. `GestureHand` also contains the functions `enableDetectors()` and `disableDetectors()`, which enables or disables all the detectors that belongs to the same hand as the current `GestureHand` instance. These are used in two scenarios: When the user switches between

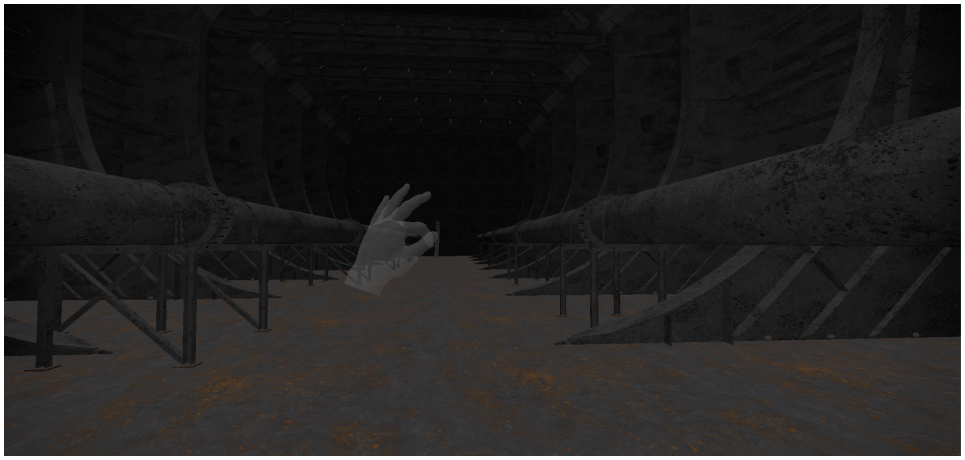

Figure 7.8: When gestures are disabled, i.e all the detectors are disabled, the hand models are transparent

having gestures enabled and disable by using the menu options "Enable Gestures" and "Disable Gestures" and when an annotation is edited. When an annotation is edited, and the annotation form is open, gestures that one can use otherwise (e.g the movement and rotation gestures) are disabled.

7.7 The Detectors

The detectors are represented as game objects and children of the `Detectors` game object under each hand game object. Each of these detector objects have the naming convention <gesture name> + "Detector" + <optional specifier> + "_" + <handedness: Left or Right>, e.g. `PinchDetectorStrict_Left` and `FistDetector_Right`. A certain detector of one hand is usual identical to the equivalent detector for the other hand with a few exception. These differences between hands are usually minor and will be mentioned when applicable. The detectors used in this implementation utilizes a combination of several Leap Motion provided detectors, as these both cover the functional needs and are considered best practice. The Leap Motion provided detectors can as such be regarded as "base detectors", while the detectors that represents gestures in this implementation can be regarded as "composite detectors". To differentiate between these two categories, the base detectors will be written in *italic* or plain text, while the composite detectors and game objects will be written in `monospace`. The Leap Motion provided detectors utilized in the implementation is *DetectorLogicGate*, *PinchDetector*, *ExtendedFingerDetector*, *PalmDirectionDetector* and *FingerDirectionDetector*.

The Leap motion base detectors provides some important parameters that can be set on a per detector instance basis, which often relates to thresholds values (e.g. on/off values) and discrete values (e.g extended or not extended). Finding the optimal values for a certain gesture can often be the subject of a lot of adjustment and tweaking, as vision-based

gesture recognition technology often or always will be someone unprecice compared to e.g. mouse and keyboard. The challenge is thus to find values that give a high amout of true positives (e.g. the user attemps a gesture and the gesture was recognized) and a low amount of false positives (e.g. the user did not attempt a gesture and a gesture was recognized). During the implementation phase these values were adjusted several times in pursuit of the optimal values, and during the evaluation phase this was a much discussed topic were users often has their own "gesture sensitivity preferences" (this will be review in the next chapter).

As was mentioned in the design, giving the user visual feedback when a gesture is recognized (i.e a detector is active) is probably a good idea. In this implementation this is performed by changing the material of the hand models. The materials for each hand is listed in the handMaterials variable in the `GestureHand` class, and follows the same order (or indecies) as the HandState enums specify (see 7.6 on page 71). Changing these materials is thus easy to do in Unity, but as default the following color-categories are used:

- Plain gold - Used for gestures that perform rotations (only the pinch gesture).
- Black - Used for gestures that perform movement. This includes the Palm-down gesture (either y-axis movement or xyz-axes movement), Palm-side gesutre (x-axis movement) and the fist gesture (z-axis movement).
- Glowing teal - Used for gestures that perform annotation interaction, i.e either places or edits annotations. This includes the single point-gesture and the double point-gesture.
- White - Used when no gesture is performed (but gestures are still enabled).
- Transparent - Used when gestures are disabled.

First the gesture implementations will be reviewed.

7.7.1 The PinchDetectorStrict and PinchDetectorSlack

The `PinchDetectorStrict` and `PinchDetectorSlack` were originally one detector called `PinchDetector`, but was split up to represent two different options for the user. Both detectors utilize the base *PinchDetector* script provided by the Leap Motion detection utilities, while the strict version also uses the *ExtendedFingerDetector*. The `PinchDetector` measures the distance between the tip of the thumb and index finger, and if these are below a certain threshold (i.e the activate distance) the detector is active and signals `gestureCode`. If the distance grow larger than a set deactivate distance the detector is deactivated. The activate distance used is 0.03 meter, while the deactivate distance is 0.06 meter.

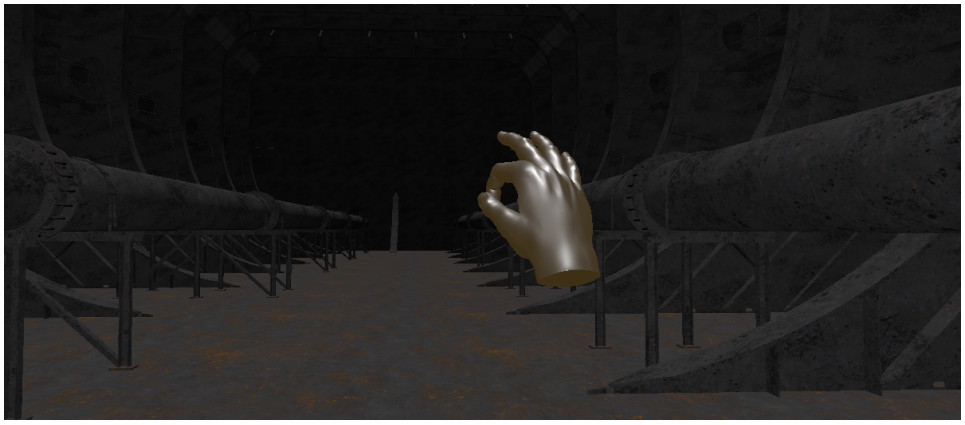

Figure 7.9: When the PinchDetector is active, i.e a pinch gesture is recognized, the hand is colored in a pale gold color.

One problem with only having distance between the two finger tips as a criterion for the pinch gesture is that there would sometimes be false positives (i.e unintentional pinch gestures could occur). This was especially the case when attemping to perform the fist gesture as the distance between the tip of the thumb and index finger is relatively small when the hand is a fist, and the application could thus sometimes perform a pinch gesture instead of a fist gesture. Because of this the stricter version `PinchDetectorStrict` was created. This uses the same criterion as `PinchDetectorSlack`, but it also requires that at least two fingers are extended. This is accompished by using the *ExtendedFingerDetector* and *DetectorLogicGate*. The finger states of the *ExtendedFingerDetector* for all individual fingers are set to "either", meaning that, individually, each finger can be either extended or not extended, but on the "minimum extended" parameter 2 is set, while "maximum extended" is set to 5, meaning that anything between two and five fingers can be extended.

7.7.2 The PalmDownDetector

The `PalmDownDetector` uses a PalmDirection detector and an ExtendedFinger detector, which is AND'ed by a DetectorLogicGate. The PalmDirectionDetector is configured to become active when the palm is pointing within 30 degrees of 0, -1, 0 (x = 0, y = -1, z = 0) direction, relative to the camera. The coordinate system used functions just as if the three-dimensional axis had been drawn on the screen, so e.g. the coordinates 0, 0, 1 would be directly forward, while 1, 0, 0 would be to the right.

The coordinates used for the PalmDirection Detector thus means that from the perspective of the camera, the palm should face directly downwords such that the palms are as parallell to the ground or table top. The PalmDirectionDetector is also configured with an "On Angel" of 30 degrees and an "Off Angle" of 50. The detector is thus activated as long as the palm points within 30 degrees of the desired direction, and is deactivated if the palm directions surpasses the threshold of 50 degree from

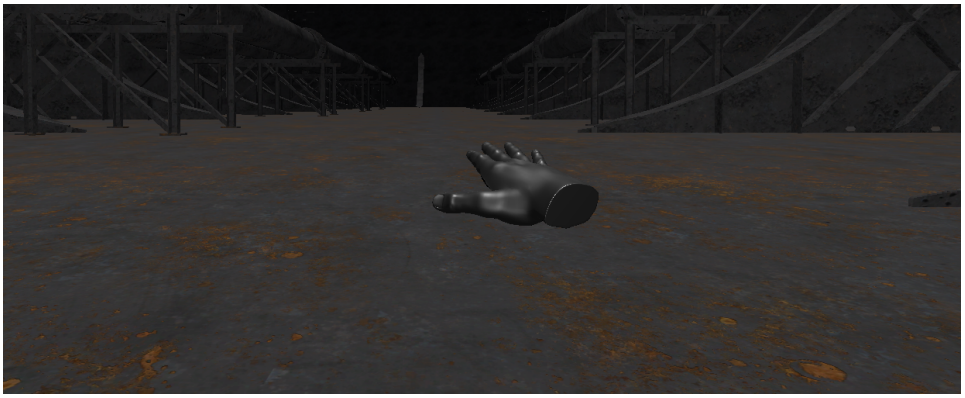

Figure 7.10: When the PalmDownDetector is active, i.e a palm-down gesture is recognized, the hand is colored in a black.

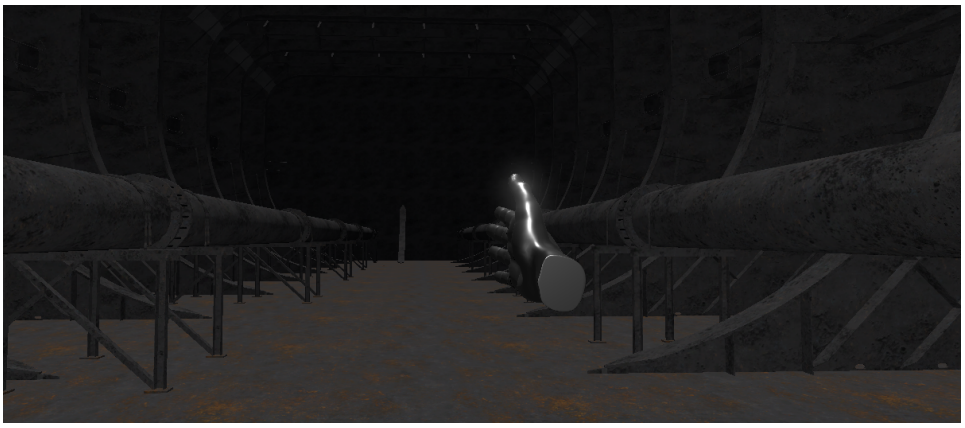

Figure 7.11: When the PalmSideDetector is active, i.e a palm-side gesture is recognized, the hand is colored in a black.

the 0, -1, 0 direction.

The `PalmDownDetector` also used an ExtendedFingerdetector, as was covered in the previous section. This one is configured to require that the index-, middle, ring and pinky finger have to be extended, which the thumb can be either.

7.7.3 The PalmSideDetector

The `PalmSideDetector` is quite similar to the `PalmDownDetector` and uses the same detectors, but with a different configuration. Although the settings used for the DetectorLogicGate and the ExtendedFingerDetector are very similar, the settings for the PalmDirectionDetector differs. Unlike the `PalmDownDetector`, where both hands are required to point in the direction 0, -1, 0 to perform the gesture, the `PalmSideDetector` makes a distinction here. This is because requiring the palm direction 1, 0, 0 (right) is reasonable for the left hand, as it is within its natural range of motion, but for the right hand this requires the whole arm to twist. The required

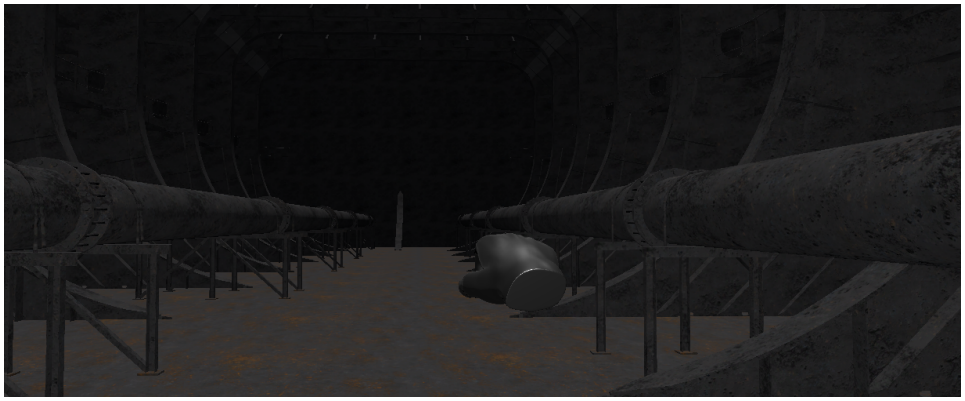

Figure 7.12: When the FistDetector is active, i.e a fist gesture is recognized, the hand is colored in a black.

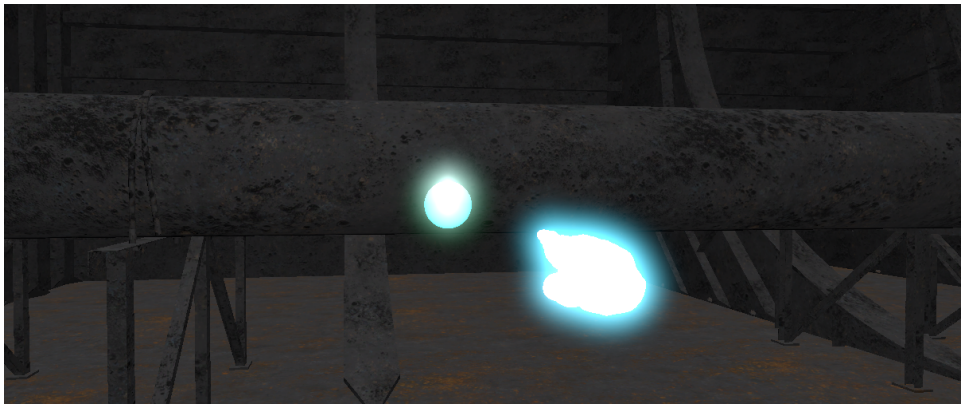

Figure 7.13: When the SinglePointDetector is active, i.e a single-point gesture is recognized, the hand is colored in a glowing teal color.

palm direction for the left hand is thus 1, 0, 0 (right), and -1, 0, 0 (left) for the right hand, both with an "On Angel" of 30 degrees and an "Off Angle" of 50.

7.7.4 The FistDetector

The `FistDetector` is prehaps the simplest of the detectors and only uses the ExtendedFingerDetector. The ExtendedFingerDetector is simply configured to require that no finger is extended. As such the minimum- and maximum amount of fingers extended are both set to 0, and all finger have their required state set to "Not Extended"

7.7.5 The SinglePointDetector

The `SinglePointDetector` uses two detectors, ExtendedFingerDetector and FingerDirectionDetector, bound together with an AND-DetectorLogicGate. The ExtendedFingerDetector here requires that the index finger is extended and that the middle-, ring- and pinky fingers are not extended (both thumb

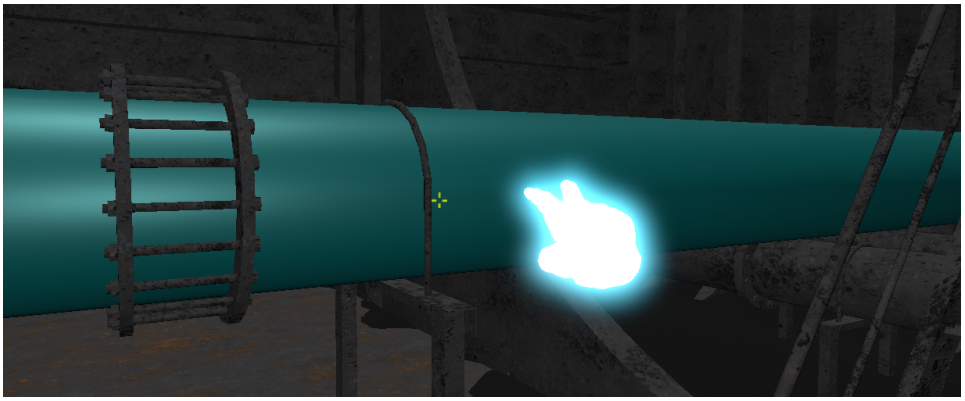

Figure 7.14: When the DoublePointDetector is active, i.e a double-point gesture is recognized, the hand is colored in a glowing teal color.

states are accepted). The FingerDirectionDetector is set to be active when the index finger points within 15 degrees of the direction 0, 0, 1, relative to the camera. Both the "On Angle" and "Off Angle" settings are set to 15 in this detector, as, unlike the previously mentioned detectors, this one is not of a "continous nature". Simply put, the previous gestures can last as long as the gesture is held, and while this is the case the application continously watches the active hands and responds to hand movements. The single- and double point detectors are techniqually also continous and can last as long as the user desires, but as soon as these gestures are activated their discrete action is performed. After this action has been performed no other action will be performed by the same and while the same gesture is kept. In the case of both the single- and double point detectors, an annotation is either placed or edited upon activation. If a user thus with to place an annotation and immediatly edit it, he or she has to use the approprate gesture, release the gesture and do the same gesture again.

7.7.6 The DoublePointDetector

The `DoublePointDetector` is similar to the `SinglePointDetector` and uses the same base detectors. The only implementational difference between this two is that the ExtendedFingerDetector is configured to require that both the index- and middle finger are extended, while the ring- and pinky finger are contracted (both thumb states are accepted).

7.8 The Menu

The menu is opened when the palm of the left hand is facing towards the camera, and can be interacted with using the index finger of the right hand. Unlike the other gestures-enabled commands the menu is implemented through the use of the open source project Hover UI Kit, created by Aesthetic Interactive (Kinstner, 2016). The Hover UI Kit project offer three different interface modules: Hovercast, Hoverkey and Hoverpanel, where

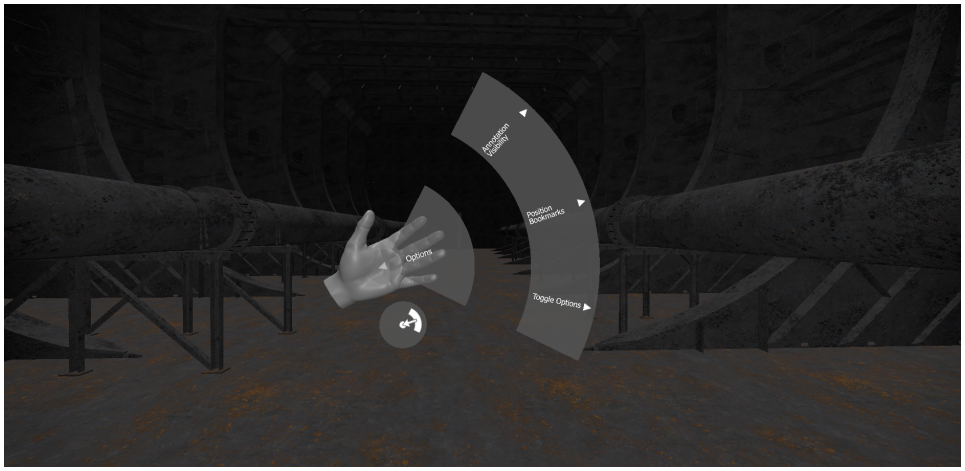

Figure 7.15: The menu is opened when the palm of the left hand is facing towards the camera, and can be interacted with using the index finger of the right hand.

Hovercast is the one the menu is based on. This means that several of the menu's interaction components, such as registering button clicks are handled by the Hover UI kit package.

The Hover UI kit main game object is present in the `GestureMenu` game object, which is a top-level game object in the project hierarchy. `GestureMenu` has four child game object: `Hovercast`, `CursorRenderer` (which is disabled by default), `MenuHandler` and `Hoverkit`. `Hovercast` represent the menu itself and contains the hierarchy of game objects that represents the menu buttons. `CursorRenderer` can optionally render a ring/cursor around the fingers that can be used as cursors. By default only the index finger of the right hand can be used to select elements in the menu. `MenuHandler` contains the scripts `ToggleOptions`, `Bookmarks` and `AnnotationVisibility`, which are all responsible for handling the different actions accessable from the menu. Lastly, the `Hoverkit` game objects contains some important "management scripts" (e.g. `HoverItemsManager` and `HoverInteractionSettings`) related to the different functionality of the Hovercast menu and hoverkey keyboard in the annotation form (more on that later). The `Hoverkit` game object also contains a Cursor game object, which contains a hierarchy with representations of the left- and right hand, and then each hand's respective finger objects. In this object hierarchy we can select which fingers that are cursors, which hand that can "host" or "spawn" the menu etc. As mentioned earlier the default setting is that the menu only can be created by the left hand palm and only interacted with using the right hand index finger.

7.8.1 The Menu Objects

The `Hovercast` game object has four game objects as direct children, which all represent a visible part of the hovercast menu: `OpenItem`, `TitleItem`, `BackItem` and `Rows`.

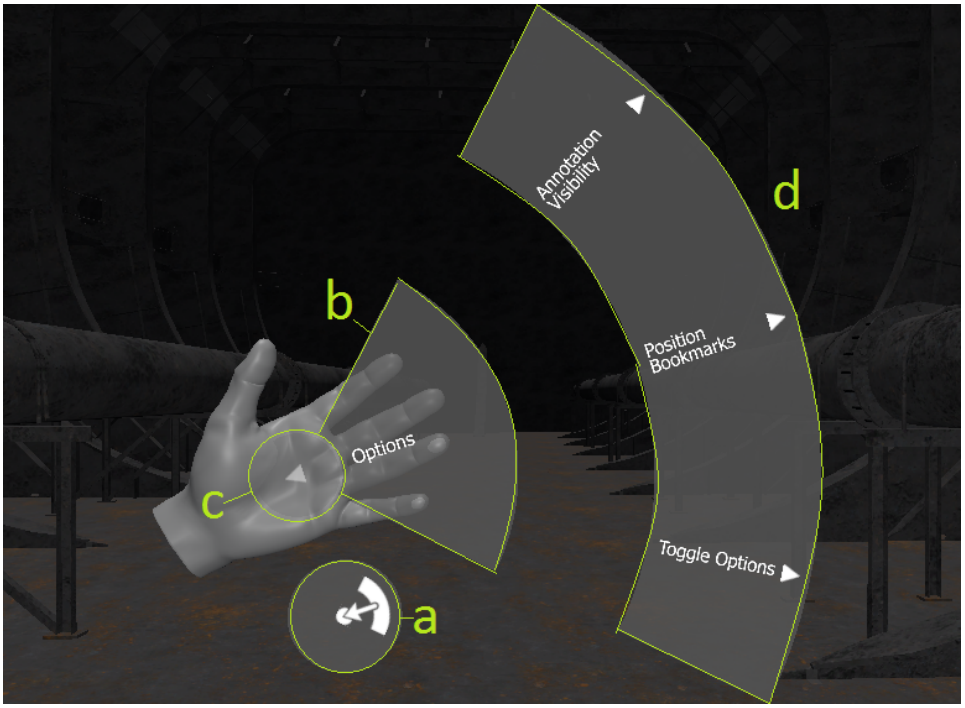

Figure 7.16: The main menu object is represented with four sub-objects: `OpenItem` (a), `TitleItem` (b), `BackItem` (c) and `Rows` (d).

While the `OpenItem`, `TitleItem` and `BackItem` uses functionality that is the default in Hover UI Kit, and is readily available in presets, `Rows` contains quite a bit of implementation specific objects. The term "row" will her refer to a set of one or more buttons, which are clickable and visible at together, in the "row section" of the menu (see 7.16 (d)). The `Rows` game object has four direct children: `Root`, `RowA`, `RowB` and `RowC`. `Root` is the root menu row, i.e the row that is visible when the menu is opened, and contains three buttons, each represented as child game objects of `Root`: `ItemA`, `ItemB` and `ItemC`. Each of these buttons leads to their own submenu, i.e their own row. `ItemA` represent the "Annotation Visibility" submenu, and brings up `RowA` as the "current row" instead of root when clicked. `ItemB` ("Position Bookmarks") and `ItemC` ("Toggle Options") follows the same logic and lead to `RowB` and `RowC` respectively (see 7.8.1 on the next page for the hierarchical overview. This transition is handled by the `HovercastRowSwitchingInfo` component that is attached to each button game object.

`RowA`, `RowB` and `RowC` each contains game objects that represents buttons in that submenu. In these button game objects `HoverItemDataSelector`, `HoverItemDataRadio` or `HoverItemDataCheckbox`, for "regular buttons", radio buttons and checkboxes respectively, are attached as components. These components are all either directly or indirectly subclasses of `HoverItemDataSelectable`, which contains important properties like "Label" (i.e the button text value) and a list of eventhandler. An example of this is `ItemCB` in `RowC`, which has the label "Disable Gestures" and has an entry in the "OnSelectedEvent" list. This entry is a reference to the `GestureOptions`

- Root menu
  - Annotation Visibility (submenu/row)
    - Always visible (radio button)
    - Visible with LOS (radio button)
    - Invisible (radio button)
    - Use Glow (checkbox)
    - Back (button)
  - Position Bookmarks (submenu/row)
    - Origin (button)
    - Back (button)
  - Toggle Options (submenu/row)
    - Toggle crosshair (button)
    - Enable gestures/Disable gestures (button)
    - Combined XYZ gestures/Distinguish XYZ gestures (button)
    - Back (button)

Table 7.8: The Menu Hierarchy

```
public void alwaysShow()
{
   //Include the SphereAnnotation layer to the culling mask and use "Depth
       only" as clear flag.
   annotationCamera.cullingMask |= 1 <<
       LayerMask.NameToLayer("SphereAnnotation");
   annotationCamera.clearFlags = CameraClearFlags.Depth;
}

public void showWithLOS()
{
   //Include the SphereAnnotation layer to the culling mask and dont clear
       anything.
   annotationCamera.cullingMask |= 1 <<
       LayerMask.NameToLayer("SphereAnnotation");
   annotationCamera.clearFlags = CameraClearFlags.Nothing;
}

public void hide()
{
   //Only render objects in the first layer (Default layer)
   annotationCamera.cullingMask &= ~(1 <<
       LayerMask.NameToLayer("SphereAnnotation"));
}
```

Table 7.9: Annotation visibility manipulation example in C# code. This code snippets makes use of bit shifts on the annotation camera's culling mask and clear-flags properties.

class' function `toggleGestures`.

7.8.2 The MenuHandler Scripts

The menu makes use of three scripts to handle all actions: `AnnotationVisibility`, `Bookmarks` and `GestureOptions`. These are all components of the `MenuHandler` game object and serve their seperate submenus (i.e Row A, B and C) and buttons.

The `AnnotationVisibility` script's main purpose is to interact with the active camera rig's annotation camera, and manipulate the way the annotations are presented to the user. This include choosing between three annotation presentation modes: "Always visible", "Visible with LOS" and "Invisible", and choosing whether to use a glow effect on annotation or not. This is done by manipulating the active annotation camera's culling mask and clear flags using bit shift:

The functional differences between the annotation presentation mode with or without glow are covered in 7.9.2 on page 85.

The `Bookmarks` script's main purpose is to handle what in the menu is refered to as "Positional bookmarks". The primary idea behind positional bookmarks is to allow the user to save specific points of interest in the model to be able to quickly return there, just like one could bookmark a web page in a web browser. In the current iteration of the application it isn't

possible to create new bookmarks, so the only available one is the "Origin" bookmark, which is the position and orientation the user has when first entering the application (useful to reset the position e.g. if the user should get lost in some way). In the script this functionality is accomplished by having a linked list of transforms (a unity game object component storing relevant information) and a referance to the `MasterController` game object. The `MasterController`'s position and rotation in the applications first rendered frame is stored as the first entry in this list and, should the user click the appropriate menu button "Origin", in the "Positional bookmarks" submenu, the function `GoToBookmark(int index)` should be called with index = 0 as index argument. The `GoToBookmark` function then looks up the transform stored at index *i*, and if successful sets the position and orientation coordinates of the `MasterController`'s transform. The user will experience this as the camera "teleporting" to the same spot as the player started on when entering the 3D model.

The `GestureOptions` script's main purpose is to handle options related to gestures, and handles two menu elements from the Toggle Options submenu: "Enable gestures" / "Disable gestures" (one button with a context dependent label) and "Combine XYZ gestures" / "Distingush XYZ gestures" (also context dependent label). This is primarily done by two of its functions `toggleGesturesActive` and `toggleGestureMode`, which both use the context dependent toggle principle and functions in the `GestureHand` class. In the implementation a toggle is simply to swap to the opposite value of the current, with two possible values available (i.e toggle + true = false and toggle + false = true).

`toggleGesturesActive` checks the value of the instance variable `GesturesEnabled` and either disables or enables gestures based on what is variable holds. If `GesturesEnabled = true`, i.e that gestures are enabled, the function disables gestures by calling each `GestureHand`'s `disableDetectors` function, swaps hand materials to the "disabled-material" and swaps the `GesturesEnabled` variable to its opposite value (i.e now `GesturesEnabled = false`). In the opposite case, i.e when the function is called and the `GesturesEnabled` variable has the value false, gestures are enabled by calling each `GestureHand`'s `enableDetectors` function, swapping hand materials to the "none-material" and swapping the `GesturesEnabled` variable to its opposite value (i.e now `GesturesEnabled = true`).

The `toggleGestureMode` functions in a very similar manner as `toggleGesturesActive` and uses a boolean value present in the `GestureHand` instances to toggle between a combined or separate state for movement gesture (i.e whether to have movement gestures in the x, y and z plans combined or separate.

7.9 The Annotations

As mentioned in earlier sections, an annotation can either be a point annotation or an object annotation. A point annotation is an instance of

```
public GestureHand[] hands;
private static bool gesturesEnabled;

public void toggleGesturesActive(HoverItemDataSelector selector)
{
   if (gesturesEnabled)
   {
      selector.Label = "Enable Gestures";
      foreach (GestureHand hand in hands)
      {
         hand.disableDetectors();
         hand.ToggleHandMaterial(hand.handMaterials[7]);
      }

   }
   else
   {
      selector.Label = "Disable Gestures";
      foreach (GestureHand hand in hands)
      {
         hand.enableDetectors();
         hand.ToggleHandMaterial(hand.handMaterials[0]);
      }
   }

   gesturesEnabled = !gesturesEnabled;
}
```

Table 7.10: The GestureOptions class can be called to enable or disable all detectors based on its current state.

a prefab called `SphereAnnotation` and is instantiated and added to the scene at the point where the raycast hits a collider (assuming the range threshold isn't surpassed). The `SphereAnnotation` prefab includes a sphere mesh, sphere collider, mesh renderer and shader, and effectively looks like a small shining orb. An object annotation functions a little differently as no new object is added to the scene to represent the annotation. Instead, the annotation is attached to the object itself and the object's material ("color") is changed to show that the object has an annotation attached. Although these two annotation types differs in this respect they both are based on the `AnnotationInformation` class, which is a component in both the `AnnotationSphere` prefab and the annotated objects.

In the current implementation `AnnotationInformation` is a pretty simplistic and limited class, but can easily be extended to include a lot of other functionality. Prehaps the most essential class member is the string variable `text`, which is simply the string value of the annotation. This value is read when the annotation form opens (i.e the user starts editting an annotation) and displayed in the annotation form text box. If the user exits the annotation form by clicking the submit button the text value is replaced/updated with what's currently in the text box. In addition to this, the `AnnotationInformation` class also contain a material-list called `annotationMaterials` and a material reference called `previousMaterial`. `annotationMaterials` is a list of the available materials to the annotation and is utilzed to look up which material to use for the annotation, e.g. when the user changes the annotation's priority (different priorities have different colors). `previousMaterial` holds a reference to the previous material the annotation had, and is most commonly used in object annotations to "remember" the original material (since annotating an object overrides its materials), in cases where the object annotation is deleted.

`AnnotationInformation` also contains a destroy function, called `Destroy`, which is used to delete annotations (i.e removing them from the scene).

7.9.1 Annotation Categories

As mentioned earlier an annotation can have one of the following labels or categories: "Information", "Warning" or "Error", selectable in the annotation form. These categories or labels were created to reflect the nature of an annotation in a design review settings. The information-category is meant for general remarks, the warning-category for potential issues and things that should be improved, and the error-category for design aspects that needs to be changed for the design to be approved. To make this annotation distinction clear to the user the annotation's material ("color") is changed depending on the category it belongs to. In the implementation the information-category is the default category, i.e the category an annotation belongs to upon creation, and has the color teal. The warning-category has the color yellow and the error-category has the color red (see figure 7.17 on the next page).

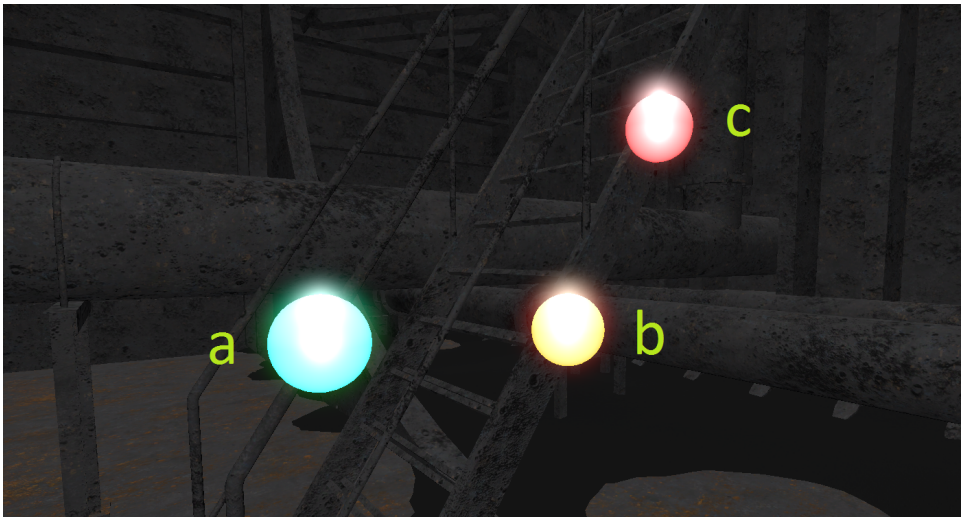

Figure 7.17: The Annotation Category Colors. a) Is an information-annotation and is colored in teal. b) Is a warning-annotation and is colored in yellow. c) Is an error-annotation and is colored in red.

7.9.2 Annotation Visibility Levels

Annotation visibiliy levels can be accessed through the menu, by clicking the "Annotation Visibility" button in the root menu, and decides how annotations appear in the virtual world. This includes choosing between three annotation presentation modes: "Always visible", "Visible with LOS" and "Invisible", with the "Always visible" setting being used by default. The user can from this menu also choose whether to use a glow effect on the annotation or not. As was mentioned in the camera rigs sections (7.3), this is accomplished by manipulating the annotation camera in the active rig, and more specifically its culling mask and clear flags. This is done in the `AnnotationVisibility` script, which is partly shown in table 7.8.2 on page 81.

Always Visible

As mentioned in 7.3 on page 66 the main camera's culling mask includes (and thus renders) every layer except the annotation layer (actually called "AnnotationSphere" in implementation), while the annotation camera, only renders the annotation layer (`CullingMask = {SphereAnnotation}`). The output of these two cameras is combined by rendering the main camera first, and then render the annotation camera and put its results "on top of" the main camera's output. To ensure that the annotation camera's output doesn't overwrite the entire output of the main camera, so only the annotation-cameras output would be shown, we have to use a different clear flag for the annotation camera than we would in a single-camera setup.

The main camera uses a clear flag called "Solid Color", which effectivly clears any empty portions of the screen by displaying a certain background

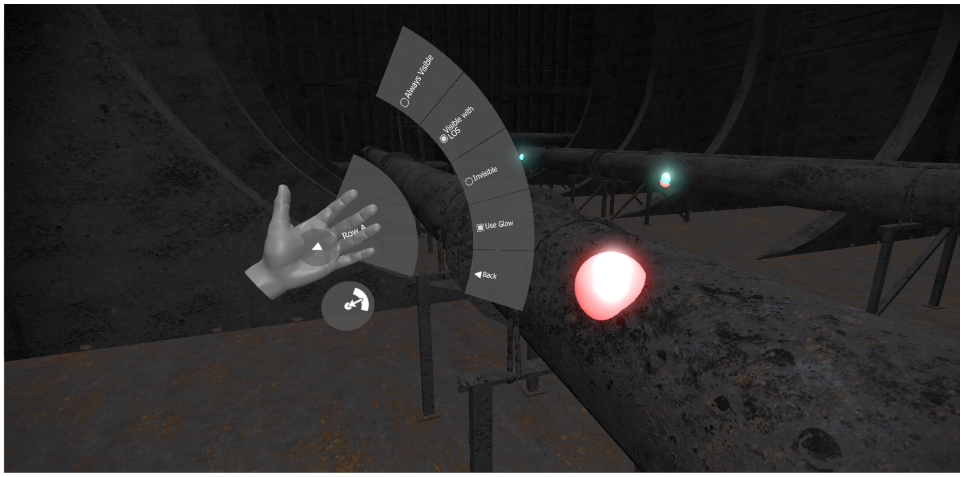

Figure 7.18: The user can decide three different annotation visibility settings in the Annotation Visibility Submenu.

color. If there is nothing for the camera to render (i.e no object in front of the camera on a layer which is included in the camera's culling mask) only this background color will be shown. If this clear flag were used on the annotation camera everything on the frame, except annotation models, would be this background color. Because of this another clear flag, called "Depth only", is used on the annotation camera. This clear flag only clears the depth buffer and not the color buffer as is done in with the "Solid Color"-flag. It also doesn't overwrite anything from the frames produced by other cameras unless it captures an object that is on a layer thats in the cameras culling mask. This effectivly means that annotations, picked up by the annotation camera, is effectly drawn over the pixels that were produced by the main camera, effectivly removing or obstructing the parts of the model that would otherwise obstruct the annotation from the user. This makes it so annotation are visible from all angles and distances with the "Always visible" setting active.

Visible with Line Of Sight

As mentioned earlier the "Always visible" option is the default in the application. This is meant to make it more easy to keep track of the different annotations present in the model. This seems like a good option with relatively few annotation, but can probably be distracting if the number or concentration of annotations grow beyond a certain size. Because of this the "Visible with LOS" (Visible with Line of Sight) option was developed to give the user the option to disable the "annotation are always visible"-functionality and instead only see annotation if they are directly in the line of sight.

The "Visible with LOS" option in the menu changes the behavior outlined in the previous section by some degree. It keeps the same culling mask configurations as in the "Always visible" setting, but changes the

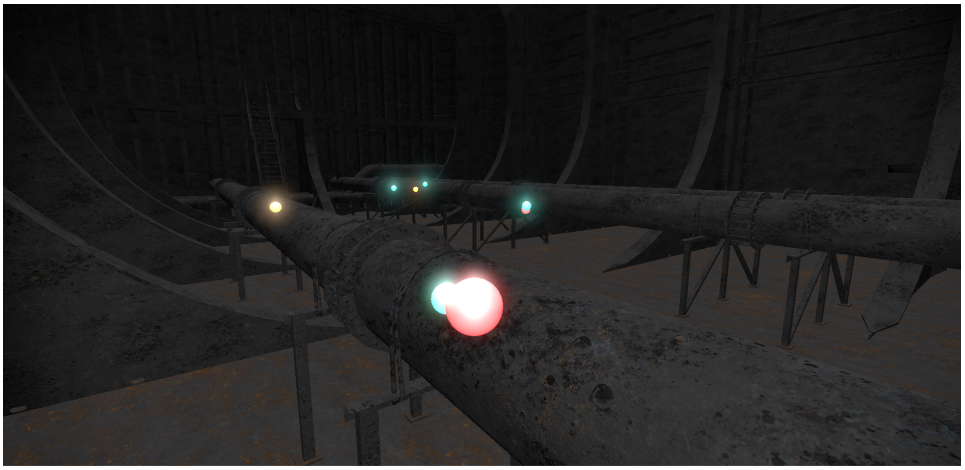

Figure 7.19: When selecting the "Always visible" option in the Annotation Visibility Submenu annotations are not occluded and thus always visible, even through other objects.

annotation camera's clear flag to the "Nothing flag". This flag will leave colors and depth buffer from the previous frame, which is produced by the main camera, and thus not clear anything. Because the depth buffer isn't cleared, as is the case with the "Solid Color" and "Depth Only" flags, objects rendered by the main camera can now obstruct objects rendered by the annotation camera, thus giving the "normal" line of sight requirement for seeing objects.

Invisible

To enable the user to hide annotation all together the "Annotation Visibility" submenu also offers an invisible-option. This option simply removed the annotion layer from the annotation camera's culling mask, thus leaving the annotation camera to render nothing. The clear flag is kept in its current state as neither the "Depth only" or "Nothing" flag interferes with the frames produced from the main camera when nothing is produced by the annotation camera.

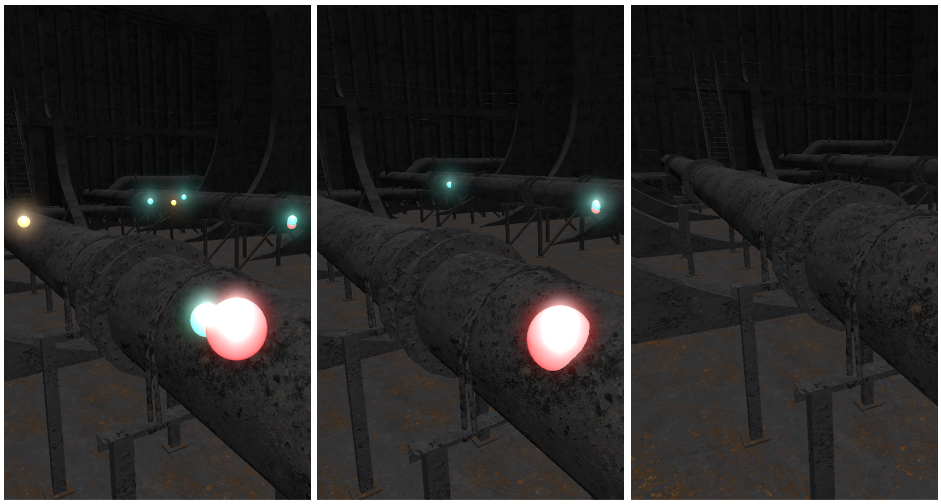

Figure 7.20: A side-by-side comparison of the annotation visibility levels. These three pictures are taken with the same camera position and orientation using different visibility settings. "Always visible" option (left): Annotations are not occluded and thus always visible, even through other objects. "Visible with LOS" option (middle): Annotations are only visible with line of sight. "Invisible" option (right): Annotations are not visible.

Chapter 8

Evaluation of the Implementation

To evaluate the application's ability to meet user requirements, two rounds of user testing seasons were conducted at the DNV GL headquarters in Høvik, Norway. The first of these session involved one test person, while the second session involved two. The participants were brought in individually and asked to take a seated position at an ordinary work stations with a mouse, keyboard and display, in addition to a leap motion controller positioned at the desk between the keyboard and the user. A HTC Vive head mount was also present for use during the experimentation phase.

The computer used for the testing had the following specifications (hardware and software):

- An Intel i7 as processor.
- 8 GB of RAM.
- A Nvidia Geforce GTX 1080 graphics card.
- A Windows 10 64-bit operating system (build 14393).
- Unity 5.5.2
- Leap Motion Control Panel version 3.2.0+45899
- Steam VR runtime (for use with the HTC Vive head mount)

After the participant was seated the test phases were conducted in the following order (including an estimated of alloted time):

1. 5 minutes of introduction. The users were informed about the purpose of the application, some of its long term goals and its limitations.
2. 10 minutes of demonstration. The users were shown each of the possible actions and the different gestures available to them.

3. 15 minutes of instructions. The users followed a series of instructions and oral explanations to teach them to use the program.

4. 20 minutes of experimentation. The users were asked to use the program freely without any instructions.

5. 10 minutes of questions. The users were interviewed with a series of questions related to the application and their experience using it.

With the exception of the experimentation phase, all the steps above were conducted without the use of a VR head mount. In the experimentation phase the participants were asked to divide their time equally between using the application in "desktop mode" (i.e using an regular display without a VR head mount) and "VR mode" (i.e using a VR head mount).

8.1 The Instructions

The participants were asked to perform the following tasks:

1. The pinch gesture is performed by pushing the thumb and index finger together, while keeping the palm directed against the table surface. Move the hand which holding the pinch gesture to rotate the camera along the X and Y axis.

2. The X gesture is performed by holding your hand straight with all fingers extended, pointing towards the screen and the palm facing downward towards the table surface. Lift and lower your hand to change move the camera along the Y axis.

3. The Y gesture is performed by holding your hand straight with all fingers extended, pointing towards the screen and the palm facing to the side, perpendicular to the table surface. Move it from side to side to move the camera along the X axis.

4. The Z gesture is performed by holding your hand curled up into a fist with no finger extended, pointing towards the screen and the palm facing downward towards the table surface. Move your fist closer or further from the screen to move the camera along the Z axis.

5. Maneuver from your current position around one of the pipes present in the 3D model and back to your original position, using one or both hands.

6. Hold your left hand straight and rotate it so the palm if facing towards you. A menu shaped like a fan should appear and follow the movements of your left hand as long as this gesture is held. Use the index finger of the right hand to select "Toggle Options" and then "Combine XYZ Gestures". To select a button hold the tip of the right index finger close enough (in terms of X, Y and Z axis) to the button for it to gradually highlight. When "Combine XYZ Gestures" has

been selected the X, Y and Z gestures are combined/replaced by a combined XYZ gesture, which is performed the same way as the Y gesture (hand straight and palm down). When now performing and holding this gesture the user can move along the X, Y, and Z axis in the virtual space by moving the hand correspondingly in the physical space.

7. Maneuver as in instruction #5, but this time by using in the combined XYZ gesture. After the user has completed this s/he might switch back to the other gesture scheme by bringing up the menu and select "Toggle Option" and "Distinguish XYZ Gestures", or keep the the combined XYZ gesture.

8. By utilizing the gestures introduced thus far, move the camera so the cursor/crosshair in the middle of the screen is positioned over a nearby object. Perform a pointing gesture by only having the index finger extended and point at the screen (away from you). If this is done correctly a blue sphere should occur, which is called an "Annotation Sphere". This is in short a unit of information related to the position it is attached to. Create two more Annotation Spheres by moving the cursor/crosshair over other nearby surfaces and point.

9. Now annotate/mark an entire object or surface by pointing two fingers ("double pointing") instead of one. These two fingers should ideally be held in a bit of an angle, like a scissor. When done correctly the entire surface or object the cursor/crosshair is indicating should be colored in a similar blueish color as the annotation spheres.

10. Now place the cursor/crosshair over an annotation sphere or an annotated object and either point (if an annotation sphere is selected) or double point (if an annotated object is selected). When done correctly a form containing a text field, a virtual keyboard and some buttons should be displayed.

11. Write "DNV GL" in the text field by utilizing the virtual keyboard. After this click on one of the colored buttons to set a color on there annotation (used to indicate a priority), and click submit to save the changes to the annotation.

12. Open the same annotation again by and delete it by pressing the delete button.

8.2 The Questions

At the end of the individual test session the users were asked the following questions:

1. Did you prefer to have distinct gestures for movement along the X, Y or Z axis or did you prefer having it combined in a single gesture?

2. How effective and responsive did you find:
   (a) The pinch gesture?
   (b) The X gesture?
   (c) The Y gesture?
   (d) The Z gesture?
   (e) The combined gesture?
   (f) The point gesture?
   (g) The double point gesture?
3. How easy to use was the menu?
4. How difficult or impractical was it to use the annotation form?
5. How difficult was it to place the cursor/crosshair where you wanted?
6. How was using the application with a virtual reality head mount different from using it in "desktop mode"? Which one did you prefer?
7. Did or do you feel any symptoms of motion or virtual reality sickness after using the application in virtual reality mode?

8.3 Responses

On question no. 1 two of the participants responded that they preferred having distinct gestures for movement along the x-, y- and z-axis, because it gives more precision and it's easier to avoid accidental movement, while the remaining participant preferred the combined gesture scheme as s/he found it more intuitive to use. Combined gestures were also favored by this participant because s/he felt that it was easier to perform the required tasks with only one hand, and s/he could thus switch between using the left and the right hand to combat fatigue (e.g. "gorilla arm syndrome").

When asking the participants about their impressions about the different gestures the answers were more unanimous in some aspect as all participants preferred the fist gesture the least and the pinch gesture the most. As we will see in the next section there might be an inverse relationship between the two. All the participants also reported that they didn't use the palm-down (y-axis movement) and palm-side (x-axis movement) that much and instead preferred to rotate to the direction they wanted to move and use the fist gesture (z-axis movement) to go there. All the participants also preferred the single-point gesture over the double-point gesture as the latter often was mistaken for the former. The participants also found the initial 25 degree on and off angle of the single- and double point detectors (see section 7.7.5 on page 76) to be to "generous", resulting in several annotation being placed by mistake. As a result of this the participants also were allowed to try with a stricter 15 degree on/off-angle, which all preferred (and thus is the application default).

The participants all seemed pretty satisfied with the menu, as they felt it was straight-forward to use. One participant felt that is was a little "too slow", by which s/he referred to how the user of the menu is required to hold a button for about 1-2 second before a click is registered (to avoid accidental clicks). Another participant also felt that is was to hard to read the menu text (i.e the labels), as he it was to small and pixelated. Both of these issues can quickly be adjusted in the implementation, by e.g increasing the font size and applying anti-aliasing to the text, but this wasn't done during the testing. With regard to using the annotation form the participants had similar remarks as to the menu.

When asking the participant about the differences between using the application with or without a virtual reality HMD, and what they prefer, they unanimously responded that they preferred to use the application in virtual reality mode. One of the reasons for this was that it was simply easier to position the crosshair/cursor at the desired location (i.e to aim) with the assistance of head movements. Although the participants felt that aiming with the crosshair/cursor was easiest in VR mode, they still didn't find it troublesome to do so in desktop mode. Another reason the participants preferred using the application with a VR HMD was because of the better depth vision they felt it provided. This was, according to one participant, especially beneficial with regards to using the menu and annotation form, as it was easier to click the buttons (i.e easier to see the relation between the hand models and the buttons). The added depth information was, according to another participant, also especially useful to understand the model better.

The participants were also asked if they felt any symptoms of motion or virtual reality sickness after using the application. One participant responded that s/he felt a little dizzy after using the application and that it felt like this was especially caused by unintentional movement (e.g. When intentionally performing a movement gesture). The other two participant felt no kind of symptoms or fatigue after using the application.

8.4 Observations

As mentioned in the previous section, the fist gesture was the least preferred gesture. This appeared to be because of two primary reason: Detector conflict and fatigue. When the participants attempted to do a fist gesture the gesture recognition system would often mistake it for a pinch gesture, possibly because the distance between the tip of the thumb and index finger is relatively small when forming a fist. As the Leap Motion base pinch detector only defines a pinch as a small enough thump–tip-to-index-tip distance, additional detectors was added to the pinch gesture to avoid this detector conflict. Because of this, there are two composite pinch detectors per hand in the implementation, one "slack" version only using the base pinch detector and one "strict" version using a combination of the base pinch detector and the base finger extended detector. The strict version requires a pinch gesture to be performed while having at least two

fingers extended at the same time. This makes it more distinguishable from the fist gesture, but also make it a bit harder to use (i.e more false negatives).

During the test session several of the participant also had interesting suggestions for the application. One participant said he appreciated not having collision on the `MasterController` in the application, thus being able to move through objects, but that he would prefer if the user "bumped into objects" before going through them. S/he thus felt that some sort of collision, while still having the ability to move through objects, would be helpful when positioning. Another user, upon discussing the fist-pinch detector conflict, suggested to use a new gesture to rotate the camera. More specifically, the participant suggested a fist gesture with the thumb extended to rotate and the regular fist gesture for Z-axis movement (thus using a gesture scheme without the pinch gesture).

8.5 Lessons Learned

Although the sample size of these test was too small to draw definitive conclusions, they do point towards some hypothesizes and theories. Generally it feels like users of a gesture recognition system preferred false negatives over false positives, i.e they would rather have it occasionally happen that a gesture they attempt aren't recognized than that the system detects a gesture which isn't being performed.

Chapter 9

Conclusion

This essay has given a brief summary of the virtual reality design review application that is going to be implemented for DNV GL as part of the master's thesis, and how virtual reality- and gesture recognition technology can be utilized to potentially improve the human-computer interaction experience beyond that of more conventional interaction methods.

Gesture recognition technology is often divided into the categories of vision-based and contact-based, where the former usually is the preferred one because of user-friendliness and the health concerns associated with the latter. Vision-based gesture recognition devices usually utilize either stereoscopic vision-, structured light pattern- or time of flight techniques, where stereoscopic vision-based devices have proved the most promising. One device of this kind is the Leap Motion Controller, which consists of two stereoscopic cameras and three infrared LEDs and periodically captures grayscale stereo images which are sent to the tracking software, where 3D representations are constructed.

The master's thesis aims to evaluate the performance and user experience of utilizing a Leap Motion Controller in combination with the Oculus Rift and HTC Vive virtual reality headsets during a virtual design review in a complex 3D model. The final application should thus be primarily focused on utilizing the most intuitive ways of interacting with complex 3D models in a collaborative virtual reality setting.

Bibliography

Abrash, M. (2012). Latency – the sine qua non of ar and vr. *The Verge*.

Antonov, M. (2015). Asynchronous timewarp examined. *Oculus Developer Blog*.

Atlassian (2017). The number one software development tool used by agile teams. [Online; accessed April 25, 2017].

Barrett, J. (2004). Side effects of virtual environments: A review of the literature. *Information Sciences*.

Benton, S. A. (1995). Visual Recognition of American Sign Language Using Hidden Markov Models Accepted by.

Bourke, A., O'Brien, J., and Lyons, G. (2007). Evaluation of a threshold-based tri-axial accelerometer fall detection algorithm. *Gait and Posture*, 26(2):194 – 199.

Brooks, J. O., Goodenough, R. R., Crisler, M. C., Klein, N. D., Alley, R. L., Koon, B. L., Jr., W. C. L., Ogle, J. H., Tyrrell, R. A., and Wills, R. F. (2010). Simulator sickness during driving simulation studies. *Accident Analysis and Prevention*, 42(3):788 – 796. Assessing Safety with Driving Simulators.

Buckley, S. (2015). This is how valve's amazing lighthouse tracking technology works. *Gizmodo*.

Bye, K. (2016). Comparing oculus touch and htc vive technology and ecosystems. *Road to VR*.

Clemes, S. A. and Howarth, P. A. (2005). The Menstrual Cycle and Susceptibility to Virtual Simulation Sickness. *Journal of Biological Rhythms*, 20(1):71–82.

Colgan, A. (2016). Leap motion controller sdk v3.2 documentations. *developer.leapmotion.com*.

Cote, M., Payeur, P., and Comeau, G. (2006). Comparative study of adaptive segmentation techniques for gesture analysis in unconstrained environments. pages 28–33.

Davies, A. (2016). Oculus rift vs. htc vive vs. playstation vr. *Tom's Hardware*.

Dean Beeler, E. H. and Pedriana, P. (2016). Asynchronous spacewarp. *Oculus Developer Blog*.

Draper, M. H., Viire, E. S., Furness, T. a., and Gawron, V. J. (2001). Effects of image scale and system time delay on simulator sickness within head-coupled virtual environments. *Human factors*, 43(1):129–146.

Feltham, J. (2015). Palmer luckey explains oculus rift's constellation tracking and fabric. *VR Focus*.

Gregory, J. (2014). *Game Engine Architecture*. CRC Press.

Guna, J., Jakus, G., Pogačnik, M., Tomažič, S., and Sodnik, J. (2014). An analysis of the precision and reliability of the leap motion sensor and its suitability for static and dynamic tracking. *Sensors (Switzerland)*, 14(2):3702–3720.

Horia Ionescu (2010). Six degrees of freedom. [Online; accessed April 04, 2017].

Hormann, H. (2006). Classification societies. *WMU Journal of Maritime Affairs*, 5(1):5–16.

International Maritime Organization (2017). Ship design and equipment. [Online; accessed April 22, 2017].

Jeffery, K. (2015). Dnv gl to unveil rules this year. *Tanker Operator*.

Johnson, A. (2013). Culling explained. *CryEngine Documentation*.

Kaaniche, M. (2009). Gesture recognition from video sequences.

Kelly, K. (2016). The untold story of magic leap, the world's most secretive startup. *Wired*.

Kennedy, R. S. ; Frank, L. H. (1985). A Review of Motion Sickness with Special Reference to Simulator Sickness. *Naval Training Equipment Center*.

Kinstner, Z. (2016). Hover ui kit. [Online; accessed January 10, 2017].

Ko, D.-i. and Agarwal, G. (2012). Gesture recognition : enabling natural interactions with electronics. page 13.

Kolasinski, E. M. (1995). United states army research institute for the behavioral and social sciences. *Tech Crunch*.

Kuchera, B. (2016). The complete guide to virtual reality in 2016 (so far). *Polygon*.

Kumparak, G. (2016). A brief history of oculus. *Tech Crunch*.

Lang, B. (2013). John carmack talks virtual reality latency mitigation strategies. *Road to VR*.

LaViola, J. J. (2000). A Discussion of Cybersickness in Virtual Environments. *SIGCHI Bulletin*, 32(1):47–56.

Leadem, R. (2016). Applications of virtual reality. *Virtual Reality Society*.

Limited, B. C. (2012). The eyes have it: Men and women do see things differently, study of brain's visual centers finds. *ScienceDaily*.

Lin, J. J. W., Abi-Rached, H., and Lahav, M. (2004). Virtual guiding avatar: An effective procedure to reduce simulator sickness in virtual environments. *Conference on Human Factors in Computing Systems - Proceedings*, 6(1):719–726.

Lu, W., Tong, Z., and Chu, J. (2016). Dynamic Hand Gesture Recognition With Leap Motion Controller. *IEEE Signal Processing Letters*, 23(9):1188–1192.

Lubell, S. (2016). Vr is totally changing how architects dream up buildings. *Wired*.

Marcus, G. J. G. J. (1975). *Heart of oak : a survey of British sea power in the Georgian era*. London ; New York : Oxford University Press. Includes index.

Marino Consulting (2017). Marine advanced technologies. [Online; accessed April 22, 2017].

Maureen Schultz, Janet Gill, S. Z. R. H. F. G. (2003). Bacterial contamination of computer keyboards in a teaching hospital. *Infection Control and Hospital Epidemiology*, 24(4):302–303.

Mitra, S. and Acharya, T. (2007). Gesture recognition: A survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 37(3):311–324.

Nishikawa, A., Hosoi, T., Koara, K., Negoro, D., Hikita, A., Asano, S., Kakutani, H., Miyazaki, F., Sekimoto, M., Yasui, M., Miyake, Y., Takiguchi, S., and Monden, M. (2003). Face mouse: A novel human-machine interface for controlling the position of a laparoscope. *IEEE Trans. Robotics and Automation*, 19(5):825–841.

Ohannessian, K. (2015). The technical challenges of virtual reality. *iQ by Intel*.

Paschoa, C. (2013). Jip collapse assessment of offshore pipelines with d/t < 15. *Marine Technology News*.

Pausch, R., Crea, T., and Conway, M. (1992). A literature survey for virtual environments: Military flight simulator visual systems and simulator sickness. *Presence: Teleoper. Virtual Environ.*, 1(3):344–363.

Pérez Fernández, R. and Alonso, V. (2015). Virtual Reality in a shipbuilding environment. *Advances in Engineering Software*, 81(C):30–40.

Pisharady, P. K. and Saerbeck, M. (2015). Recent methods and databases in vision-based hand gesture recognition: A review. *Computer Vision and Image Understanding*, 141:152–165.

Premaratne, P. (2014). *Human Computer Interaction Using Hand Gestures*.

Rautaray, S. S. and Agrawal, A. (2015). Vision based hand gesture recognition for human computer interaction: a survey. *Artificial Intelligence Review*, 43(1):1–54.

Robertson, A. (2016). The ultimate vr headset buyer's guide. *The Verge*.

Rolnick, A. and Lubow, R. E. (1991). Why is the driver rarely motion sick? the role of controllability in motion sickness. *Ergonomics*, 34(7):867–879. PMID: 1915252.

S., J. (2016). The tech behind playstation vr and how it delivers 120 hz on console. *Game Debate*.

Silver, C. (2015). Gift this, not that: Myo armband vs this toaster. *Forbes*.

Stanney, K. M. (2002). *Handbook of virtual environments: design, implementation, and applications*.

Stanney, K. M., Hale, K. S., Nahmens, I., and Kennedy, R. S. (2003). What to expect from immersive virtual environment exposure: Influences of gender, body mass index, and past experience. *Human Factors*, 45(3):504–520.

Stanney, K. M., Kennedy, R. S., and Drexler, J. M. (1997). Cybersickness is not simulator sickness. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 41(2):1138–1142.

Unity (2016). User interfaces for vr. [Online; accessed April 10, 2017].

Vafadar, M. and Behrad, A. (2014). A vision based system for communicating in virtual reality environments by recognizing human hand gestures. *Multimedia Tools and Applications*, 74(18):7515–7535.

Wang, X. and Li, X. (2010). The study of movingtarget tracking based on kalman-camshift in the video. pages 1–4.

Weichert, F., Bachmann, D., Rudak, B., and Fisseler, D. (2013). Analysis of the accuracy and robustness of the Leap Motion Controller. *Sensors (Switzerland)*, 13(5):6380–6393.