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 (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(Leadem, 2016). Leadem (2016) reports numerous domains where virtual reality is successful being used, including healthcare (e.g surgery), military, architecture/construction, art, fashion, entertainment (games, films etc), education, business, telecommunications, sports and rehabilitation.

Despite this early success, there are still a lot challenges associated with virtual reality technology. One of these challenges is related to human-computer interactions and will be expanded upon later in this chapter. This chapter will first discuss the virtual reality field and how gesture recognition technology can be very relevant for it, before defining the problem definition, limitations and outline for the rest of this thesis.

1.2 Virtual Technology

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). This virtual environment is perceived through a virtual reality headset, which is a stereoscopic head-mounted display (HMD) that provide separate images for each eye (Kuchera, 2016). In addition to separate eye displays a HMD typically also contains head motion sensors such as gyroscopes, accelerometers and other sensors to track the user's head movements(Kelly, 2016). A person using a virtual reality head-mounted display should thus perceive a virtual world with realistic depth vision and be able to "look around" by turning his or her head.

Figure 1.1: The Oculus Rift Development Kit 1, released by Oculus VR in 2012.

The development of virtual reality head-mounted displays was in many ways fueled by the development of smart phones as many of the components are similar (e.g. gyroscopes), and these components also became more affordable by the prominence of smart phones. This led to the prototype HMD "Oculus Rift Development Kit 1", released by Oculus VR in 2012, being the first independently developed and sold virtual reality headset(Kelly, 2016).

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 communicating with our 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 whats include in the application and what isn't.

1.5 Outline

This thesis is organized as follows: In chapter 2 we will review the history and theoretical foundations for virtual reality, as well as discuss its performance demands and the various challenges it presents. In chapter 3 we will discuss gesture recognition technology, its concepts and the different technology enabling this technology. Chapter 4 will review the Leap Motion Controller, a vision-based gesture recognition device, both in terms of its hardware and software properties. Especial emphasis will be put on its API (application program interface), which significantly simplifies the creation of programs utilizing gesture recognition. Chapter 5 we will review the design of the design review application, its user stories (i.e function requirements) and what gestures it makes use of. Chapter 6 will go more into the technical details of how the application is implemented in the Unity game engine and review its architecture along with some central Unity concepts. In chapter 7 the user evaluation sessions will be covered, and the responses discussed and analyzed. Chapter 8 will conclude this thesis with a summary of the findings and some thoughts about future work.

Chapter 2

Virtual Reality Technology

2.1 The virtual reality ecosystem

As described in the previous chapter, a virtual reality head-mounted device (HMD) is in simple terms a device that is fastened to the user's head and, when fastened, covers the user's entire field of vision. Each eye has its own display, and both of these are positioned about 2-3 centimeters from the eyes. In addition to this several head motion tracking sensors are built into the headset to detect any movement (Kuchera, 2016). This usually includes a gyroscope, which is responsible for measuring the orientation of the HMD, and some times 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 headset also usually utilizes 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 are usually placed in opposite corners of the room, and uses photosensors and structured light lasers to obtain the users position and rotation Buckley (2015). It is worth noting that both of these virtual reality headset also is sold with their own controllers, which uses similar technology as the HMD, but as previously stated this thesis will investigate gesture recognition systems as the primary interaction method.

2.2 Virtual reality performance demands

Virtual reality places some strict demands on performance and software design to avoid discomfort for the user. This is in many ways connected to how virtual reality "tricks" the user's brain into thinking the virtual experiences are actually real, thus giving it its "reality feel". Failing to meet these demands can quickly result in significant discomfort for the user, often called ***virtual reality sickness***, a kind of motion sickness (Orland, 2013). Some of these demands are described below:

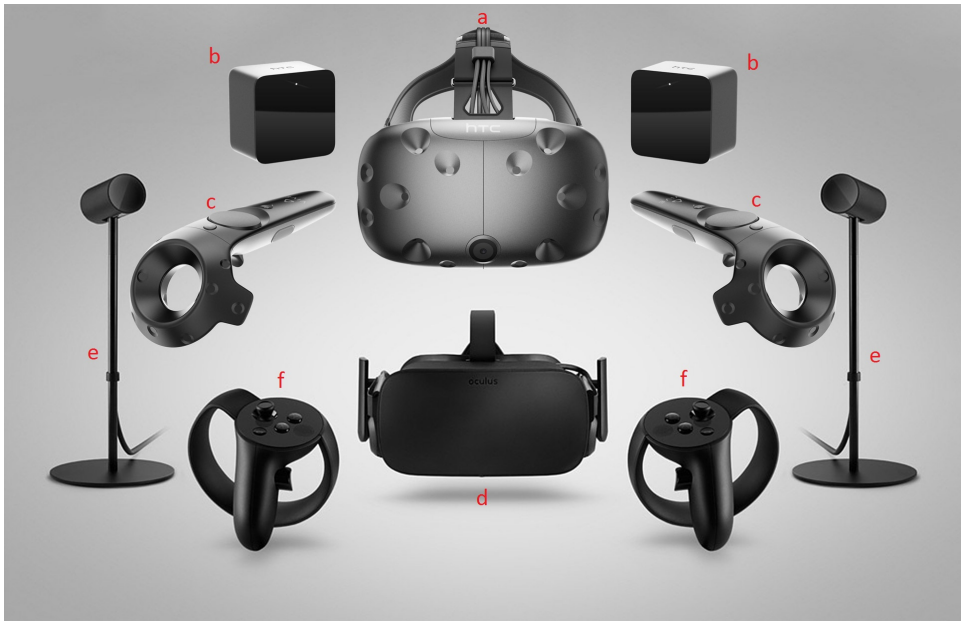

Figure 2.1: The HTC Vive and Oculus Rift Hardware. a) The HTC Vive headset (HMD). b) The HTC Vive Lighthouse Stations. c) The HTC Vive Controllers. d) The Oculus Rift headset (HMD). e) The Oculus Rift Constellation Sensors. f) The Oculus Rift Touch Controllers. Picture from Bye (2016)

2.2.1 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 feel "sluggish", and user's actions and visual feedback might feel disjoint, which often lead to virtual reality sickness. According to one of the engineers behind the HTC Vive, the ideal latency is between 7 and 15 milliseconds (Orland, 2013). One important component of this 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. As an example 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), 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). Refresh rate and frame rate are thus highly codependent, where performance is only as good as the weaker of the two, and the target computer should thus have a 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 makes 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.

2.2.2 Display resolution and quality

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)***, which is when the lines separating the display pixel or subpixels is 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 shortcoming quickly came to be known as "the screen door effect")"

2.3 Virtual reality sickness

As mentioned in the previous sections, virtual reality sickness, a condition similar to ***simulator sickness***, can be a consequence of the usage of virtual reality headsets, and is considered a major barrier to using virtual reality. Like simulator sickness, virtual reality sickness causes symptoms that are similar to those of motion sickness, and can include symptoms like headache, stomach awareness, nausea, vomiting, pallor, sweating, fatigue, drowsiness and disorientation (Kolasinski, 1995).

Contrary to "regular" motion sickness, where the user visually perceived to be still while in actual motion, virtual reality sickness turns this around: The user visually perceive to be in motion which he or she is still. Virtual reality sickness can thus in many ways be considered as "a reverse

Figure 2.2: An example of the screen-door effect.

motion sickness". Both these condition can thus be caused by ***sensory conflict***, i.e that there exist a discrepancy between the information given by the senses ("the human sensors"). The susceptibility for this condition vary widely among users. Some user 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). The following two subsections 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 application design.

2.3.1 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.

Age

Research suggest that users between the ages of 2 and 12 are the most susceptible to virtual reality sickness (Kolasinski, 1995). The susceptibility then decreases rapidly until an age of about 21, before it start decreasing more slowly until and age of 50, where the susceptibility increases again (Brooks et al., 2010).

Gender

Women have proven more susceptible to 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 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).

2.3.2 Virtual reality design factors

This section identifies some of the most common contributers to virtual reality sickness that can be lessened or mitigated completely by the VR

application design.

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).

Focus distance

To avoid discomfort and fatigue it is important to place content the user 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).

Latency and lag

As mentioned earlier in this chapter, latency and lag can have a major impact VR 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).

Mouse and keyboard usage

While a user is wearing a virtual reality headset, interaction with external input devices such as a keyboard, might be inconvenient or difficult. Put simply, this is because the user can't see his or her hands and thus can't get the visual hand positional feedback they could get without the virtual reality HMD. Because of this, many virtual reality applications makes use of a gamepad controller instead.

Chapter 3

Gesture Recognition Technology

3.1 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 3.2).

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 3.3).

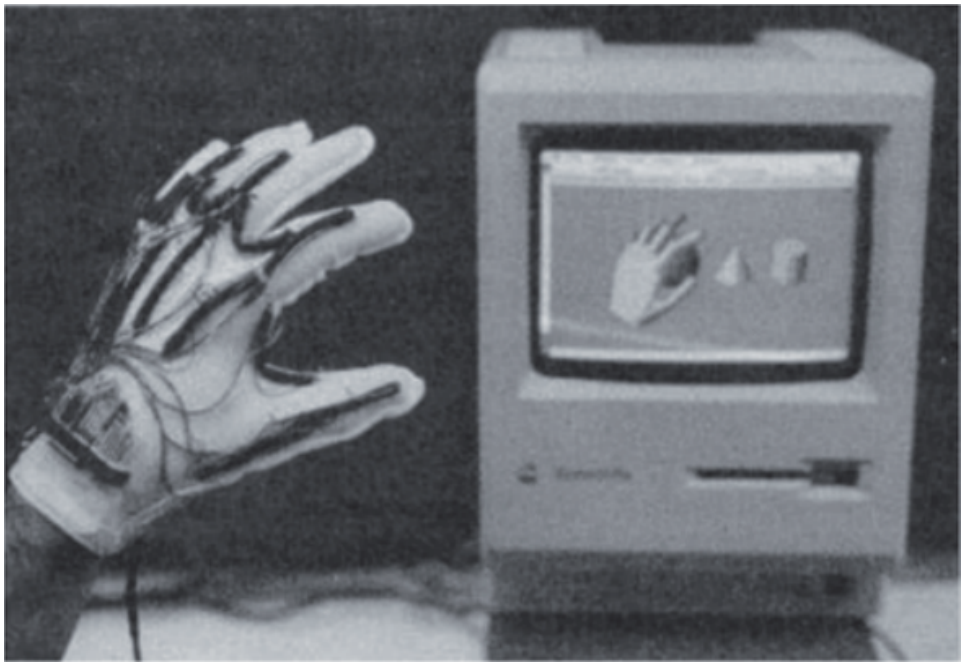

Figure 3.1: The Z Glove, developed by Zimmerman in 1982. Picture from Premaratne (2014)

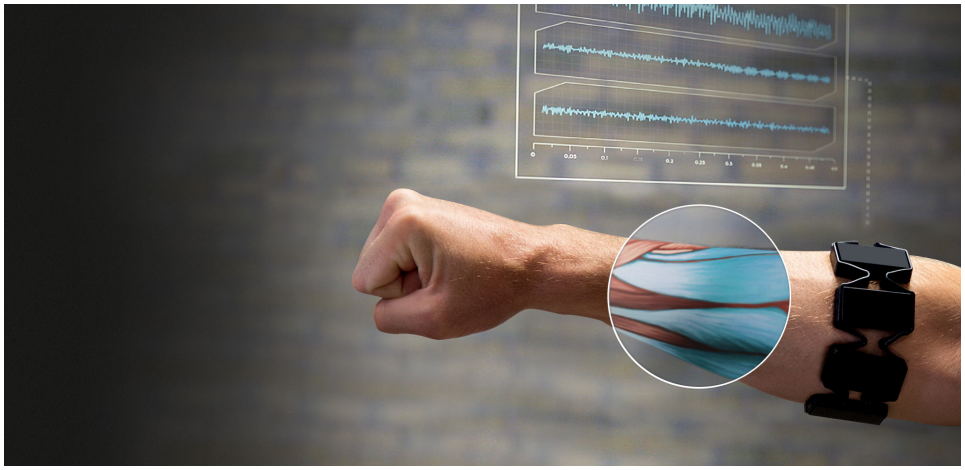

Figure 3.2: The Myo armband is a gesture recognition device worn on the forearm and manufactured by Thalmic Labs. The Myo enables the user to control technology wirelessly using various hand motions. It uses a set of electromyographic (EMG) sensors that sense electrical activity in the forearm muscles, combined with a gyroscope, accelerometer and magnetometer to recognize gestures (Silver, 2015).

Figure 3.3: 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 that of 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.

3.1.1 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 3.4: 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 pattern 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.

3.2 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.

3.2.1 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 3.5 for an hierarchical overview.

3.2.2 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

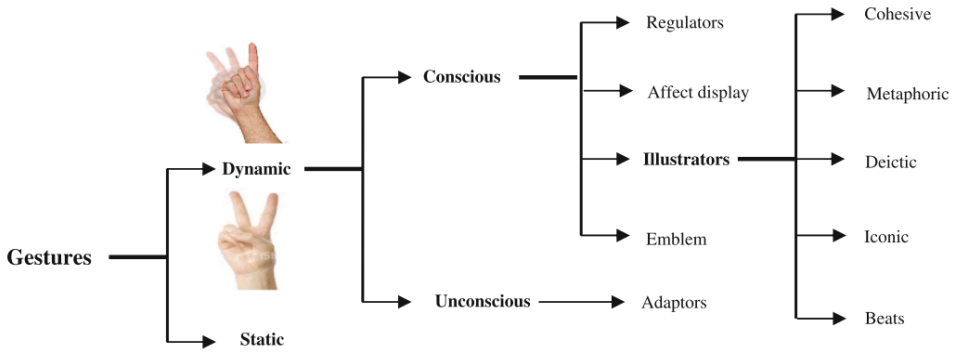

Figure 3.5: The vision-based hand gesture categories (Kaaniche, 2009).

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.

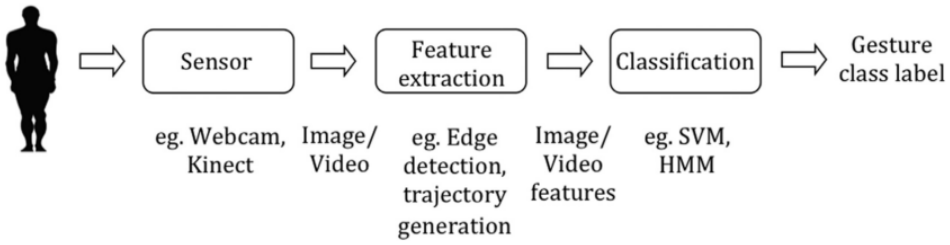

Figure 3.6: A typical gesture recognition pipeline (Pisharady and Saerbeck, 2015)

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.

3.2.3 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).

3.2.4 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).

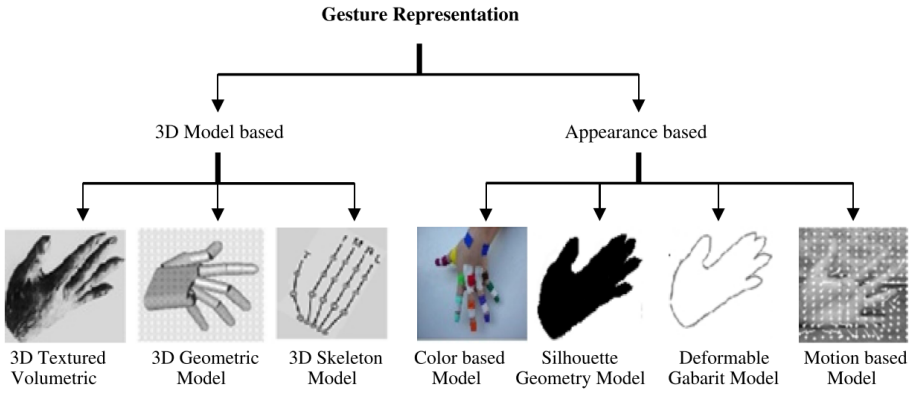

Figure 3.7: Vision-based hand gesture representations (Bourke et al., 2007)

Chapter 4

A review of the Leap Motion Controller

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.

4.1 Physical properties

The Leap Motion Controller (see fig. 3.3 and 4.1) contains two stereoscopic cameras, with a field of view of 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.

4.2 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

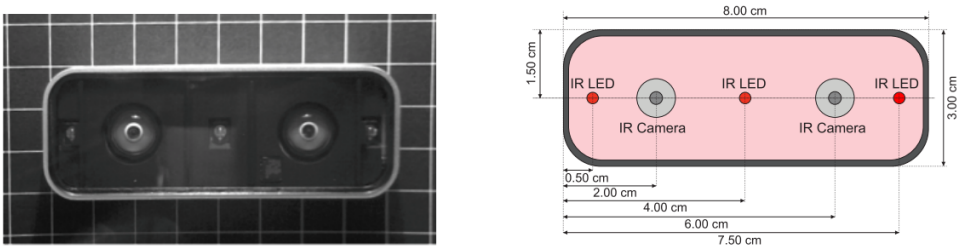

Figure 4.1: Visualization of a Leap Motion Controller, with Infrared Imaging (left) and a Schematic View (right) (Weichert et al., 2013).

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.

4.2.1 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.

4.2.2 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 proceeeding, 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;
            }
        }
    }
}
```

4.2.3 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 4.2) (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 4.2). 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).

4.2.4 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,

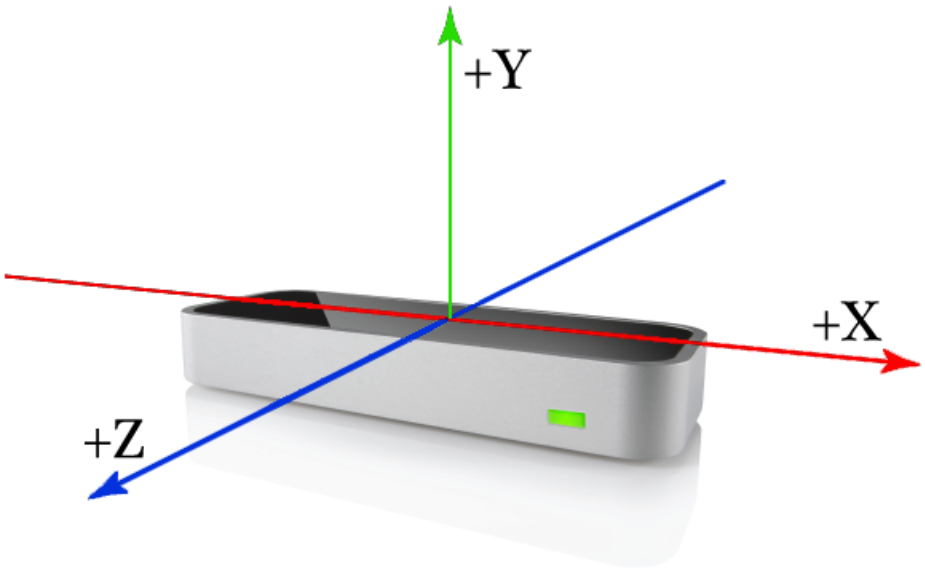

Figure 4.2: The Leap Motion Coordinate System has its origin between the two cameras.

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 5

Designing the virtual design review application

5.1 DNV GL and their motivations

DNV GL is the world's largest classification society with 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). A major part of DNV GL's work is evaluation and quality assurance of a client's product (e.g. a ship) , where a DNV GL "Approval Engineer" conducts a design review of the client's model of the proposed product. This process usually consists of the following steps:

1. The designer sends the model to DNV GL for evaluation.
2. The approval engineer inspects the model noting down aspects that doesn't meet DNV GL requirements.
3. The designer receives the remarks and makes the necessary changes to the model.
4. This process is repeated until both parties are satisfied.

DNV GL is looking into the possibilities of digitilizing this process, and making it more interactive and efficient by using virtual reality technology to conduct virtual design review meetings in the 3D models. 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 a 3D environment (Rautaray and Agrawal, 2015). As mentioned in the previous chapters this thesis will use the Leap Motion Controller, a vision-based device using stereoscopic cameras, as a primary input device to the application.

5.2 The core design

The core functionality in the application should be to navigate the 3D model 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. The users should in later iteration also be able to create "sessions" that enable several users to be virtually present in the same instance of the 3D model, 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. Beyond this there is a lot of other functionality which should be in place for a complete product, but which will not be a priority for this thesis as the virtual reality and gesture recognition aspects are the focus.

In the final product the application should support a lot more functionality, some of which is described in the next sections.

5.2.1 Application use cases

This section gives an overview of the use cases which is intended for the finished application, some of which will be implemented in this thesis. This section separates the user stories into two subsection "The Launcher" and "The Inspector", where "The Inspector stories" was all implemented, while "The Launcher stories" where skipped because they aren't relevant for the thesis, and to give more time to prioritize "The Inspector stories".

The Launcher

Note that the Launcher was not implemented as part of the thesis. The Launcher should show up when the application is launched and in many ways function as a "server browser". Here the user can decide between hosting a session on a 3D model he or she has access to, or join an existing session from a list/browser. When the user wishes to initiate or join a session with a particular 3D model to be inspected, he/she should be able to:

When hosting a session, the user should be able to:

- Specify a 3D model from a standard file format to host the session on.
- Give the session a name
- Define a password, which will be required to enter the session.
- Choose between different visibility settings for the session (e.g whether it should show up in the session list)

When joining a session, the user should be able to:

- Choose a session from the session list and click the join-button to enter.

- Enter the name of a session in the search text field to search for a session by name. This should also enable to find sessions that are otherwise "hidden".

The Inspector

Once a user has either created or joined a session and is loaded into the model, he or she should be 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 **??** 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

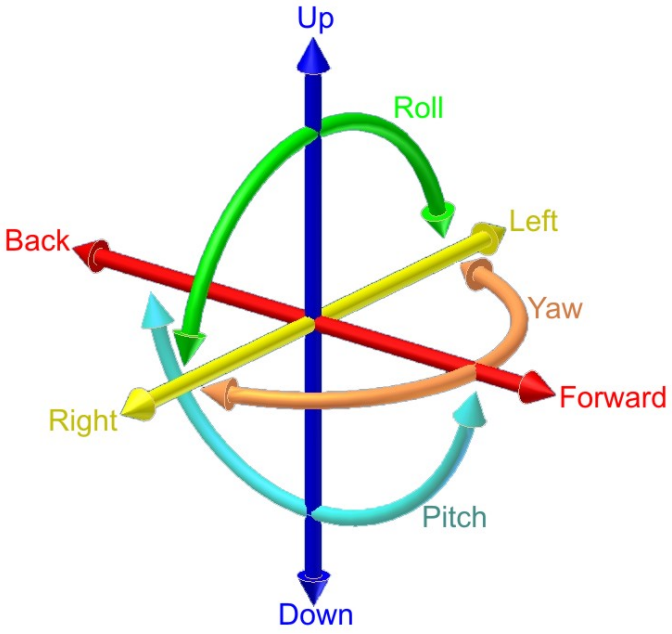

Figure 5.1: The six degrees of movement in a three-dimensional space. A rigid body in this space can change **position** along the X axis (left/right), Y axis (up/down)and Z axis (forward/backward), or change **rotation/orientation** along the X axis (rolling), Y axis (pitching) and Z axis (yawing).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

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.
- 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).

Actions done during the 3D model session (such as annotating an object) should continuously be stored in a database. 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. By utilizing a database in this way the model files themselves can also remain unedited throughout a session, as opposed to saving annotations into the model files itself, which could be more inefficient and create model versioning issues. 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 on the 3D model to become "issues" or "remarks" in more traditional collaboration tools such as Atlassian's Jira or Confluence, although this will not be a focus point for the thesis.

5.3 Functionality limitations

5.3.1 Handling textual input with gestures

5.3.2 User gesture calibration

5.3.3 Saving annotation to a database

5.3.4 Exposing annotation to web servers

5.3.5 Annotation time-lines

5.4 The gestures

As mention in the user stories 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 story", 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 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).

5.4.1 The pinch gesture

The pinch gesture will cover the "rotate user story", 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

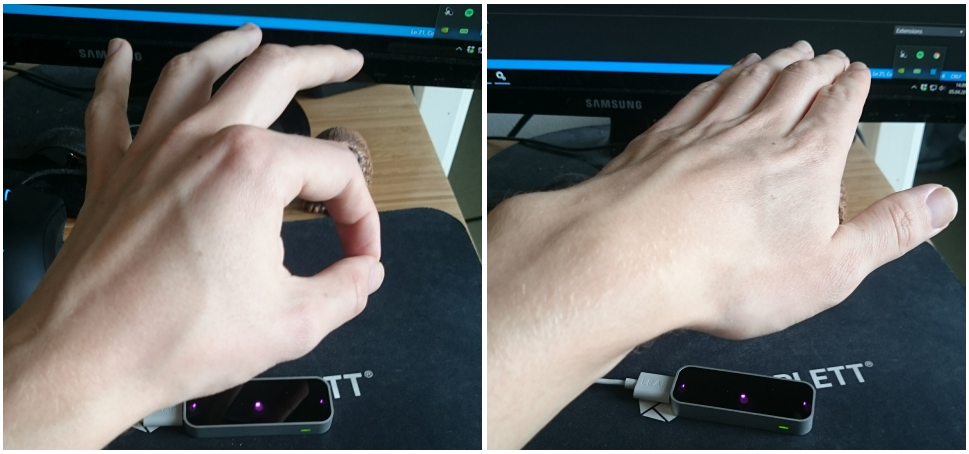

Figure 5.2: The pinch gesture (left) is used to rotate the camera along the y- and z-axis. The palm-down gesture (right) is used to move the user up and down along the y-axis.

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).

5.4.2 The palm-down gesture

The palm-down gesture, alternatively called the Y-gesture, fulfills the up-and-down functionality specified in the "move-user story", 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 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.

5.4.3 The palm-side gesture

The palm-side gesture, alternatively called the X-gesture, fulfills the left-and-right functionality specified in the "move-user story", and enables the

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.

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 for an illustration). As one of the movement gesture, this gesture also uses the origin-offset scheme, but only with the x-axis monitored.

5.4.4 The fist gesture

The fist gesture, alternatively called the Z-gesture, fulfills the forward-and-backwards functionality specified in the "move-user story", 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.

5.4.5 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

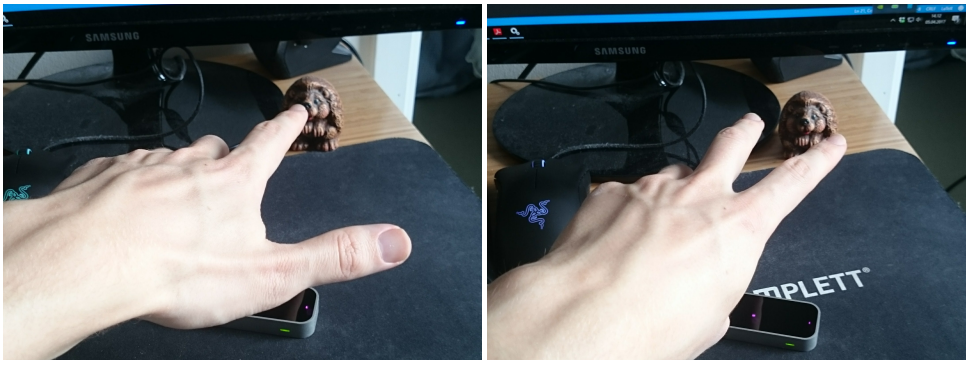

Figure 5.4: The single-point gesture (left) is used to create or edit a point annotation. The double-point gesture (right) is used to create or edit an object annotation.

movement along the x-, y- and z-axis. This gesture also used the origin-offset scheme, but now all the three dimensions are monitored.

5.4.6 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.

5.4.7 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.

5.4.8 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.

Chapter 6

The Implementation

6.1 External resources

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.

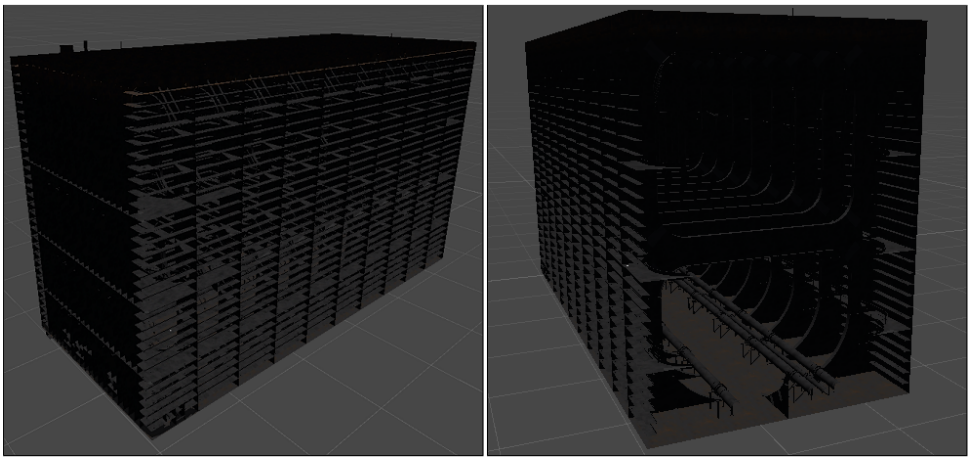

Figure 6.1: The Oil tank model from the outside

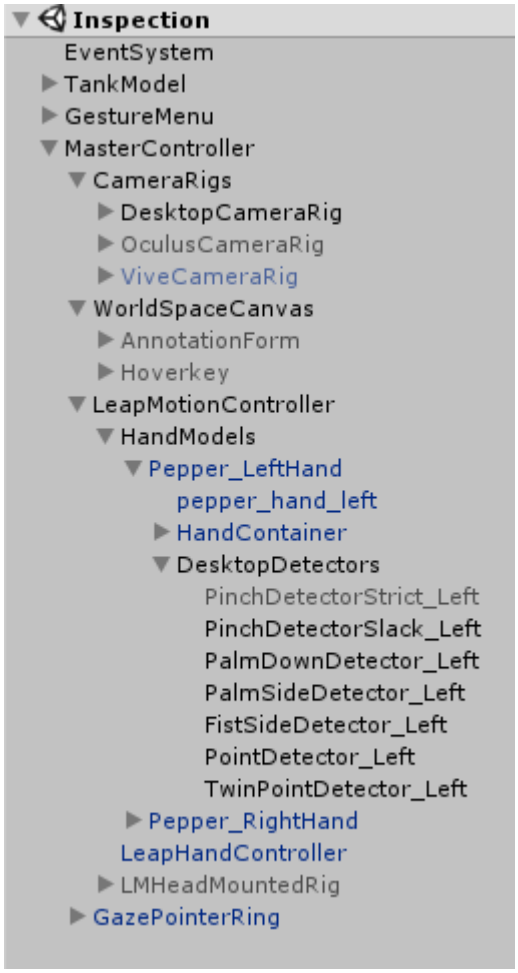

Figure 6.2: The Unity project hierarchy of the Design Review Application

6.2 The architecture

The Unity project has four top-level game objects, as visible in 6.2. These will be covered by their own sections, with subsections detailing their important child game objects where appropriate.

6.3 The master controller

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. Because this game object is so essential, we will cover several important components and child objects in the following section.

6.3.1 The camera rigs

The `CameraRigs` game objects 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. 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 wheter 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.

6.3.2 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

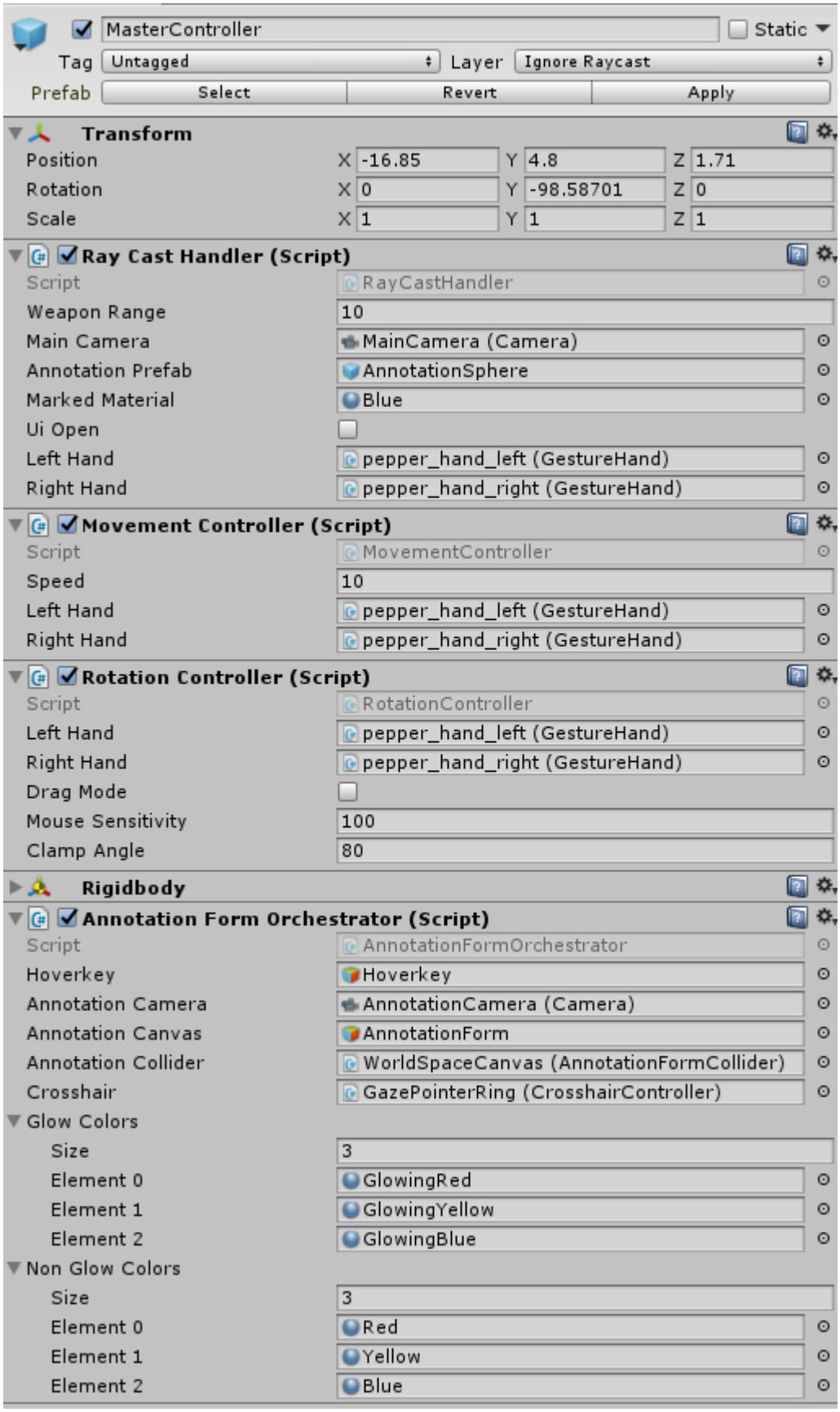

Figure 6.3: The `MasterController` components seen in the Unity Inspector view.

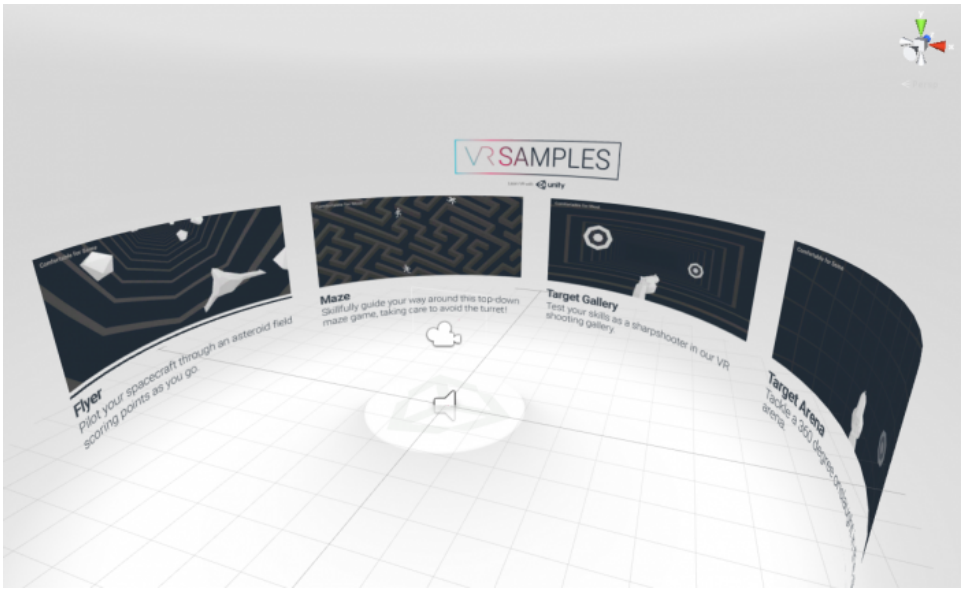

Figure 6.4: An example of world-space (diegetic) user interfaces (Unity, 2016).

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).

`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 thus always in the center of the camera, but is only visible and enabled when the user is editing an annotation.

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 6.6).

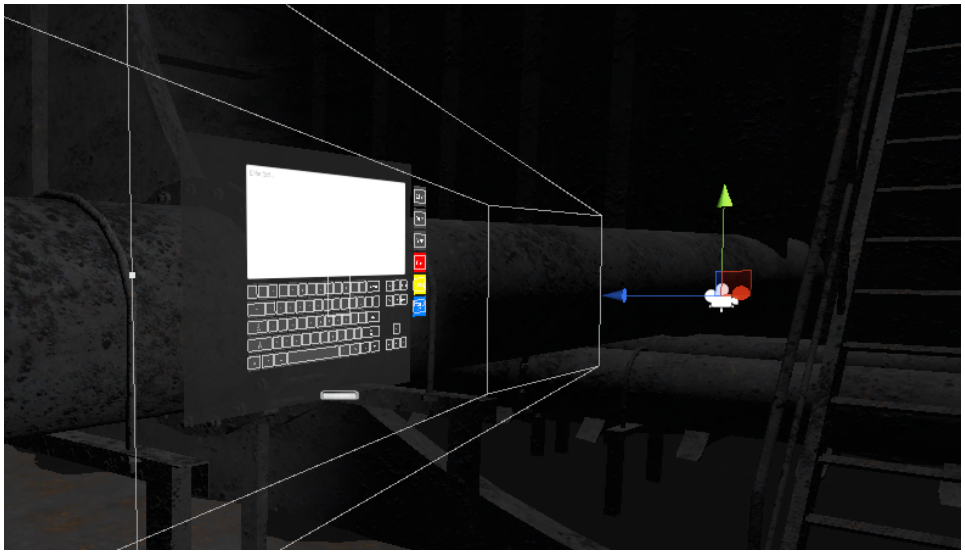

Figure 6.5: The `WorldSpaceCanvas` as seen in the Unity Scene View.

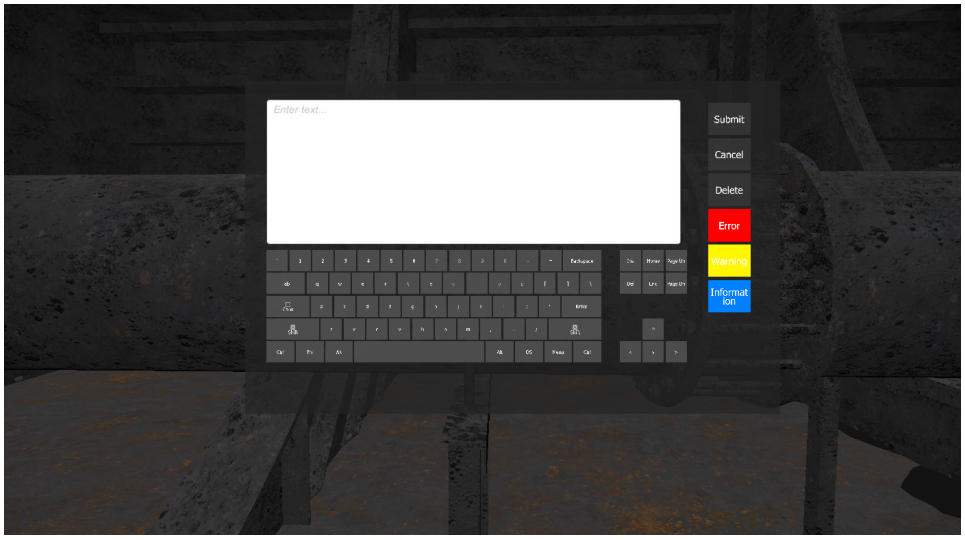

Figure 6.6: The `WorldSpaceCanvas` as seen in the Unity Game View.

6.3.3 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 5.4. As these detectors are a key component of this implementation, they will be reviewed in more detail.

Chapter 7

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 season were held the 24th March 2017 and involved one test person, while the second round were held at X and involved Y persons. The users 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 user 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 users 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).

7.1 The instructions

The users 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.

7.2 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 was it to place the cursor/crosshair where you wanted?
5. How difficult or impractical was it to use the annotation form?
6. How was using the application with a virtual reality head mount different from using it in "desktop mode"? Which one did you prefer?

Chapter 8

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

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.

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*.

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].

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

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*.

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*.

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.

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.

Orland, K. (2013). How fast does "virtual reality" have to be to look like "actual reality"? *Ars Technica*.

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

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.

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.