Imperial College London
Department of Computing

Augmented Reality using the Oculus Rift with Stereopsis and Real-time Occlusion Handling

David Avedissian

June 2016

Supervised by Ben Glocker

Submitted in part fulfilment of the requirements for the degree of
Bachelor of Engineering in Computing of Imperial College London

Abstract

Recently, development of Virtual Reality and Augmented Reality technologies, as well as research in the domain of Computer Vision has motivated a number of new and exciting applications. In this report, we investigate a variety of hardware and software choices to devise a prototype of an Augmented Reality system built using a head-mounted display and a stereo camera, and touch upon some well known issues with such a system. We then discuss an implementation using an Oculus Rift Development Kit 2 as a head-mounted display, with a Stereolabs ZED camera to provide stereo vision and an Intel Realsense F200 to measure depth.

For evaluation, we investigate the performance of the prototype to understand any bottlenecks, and determine the effectiveness of how we achieved presence using a stereo camera. We later conclude with a summary of the limitations of the prototype, discuss future work that can be taken with newer hardware released later on, and potential methods to overcome some of the limitations. Finally, we briefly touch upon some further applications of an Augmented Reality system similar as ours.

I would like to thank my supervisor, Dr Ben Glocker, for his continuous support and enthusiasm to help successfully complete the project. Additionally, I would like to thank my parents and girlfriend Dilyana for their unconditional love and support.

Lastly, I'd like to also acknowledge the Department of Computing Society for replacing some faulty hardware when time was short, allowing me to complete the project successfully.

Contents

1 Introduction

1.1 Motivation

In recent years, there has been a huge wave of excitement in the development of Virtual Reality technologies. From the development of consumer level head-mounted display devices with the Oculus Rift and Samsung Galaxy VR to exciting research into how these devices can be used to achieve *presence* - total immersion, VR technology has been widely regarded as the "next big thing" after the explosive popularity of smart-phones.

Beyond the domain of gaming, there has been a rise in interest in how this technology can be applied in fields such as communication, simulation and medicine. For example, virtual reality can potentially enable people to communicate and share experiences together in real time, despite being thousands of kilometres apart from each other. Additionally, an iPhone fitted inside a Google Cardboard frame was used to help plan heart surgery and save the life of a baby [1].

Another field with much overlap with Virtual Reality is known as Augmented Reality. It is concerned with taking a live view of a physical, real-world environment, and *augmenting* it by adding additional computer generated stimuli such as sound or computer graphics. Augmented Reality technology is commonly used in the military and in aviation. For example, in Figure 1.1, a display inside a NASA X38, an experimental re-entry vehicle designed by NASA for the International Space Station, is shown displaying additional information such as points of interest on top of an existing video stream.

Augmented Reality has seen more widespread use in recent years, with the advent of increasingly powerful mobile hardware in smart-phones. You can now download and install applications which take the video stream from the phone's camera, and project some 3D game onto a table. Furthermore, Microsoft has introduced the **HoloLens**, a device designed to integrate with Windows 10 to incorporate informational tiles as holograms into a users environment and allow interaction with them, designed as an evolution from using a computer monitor.

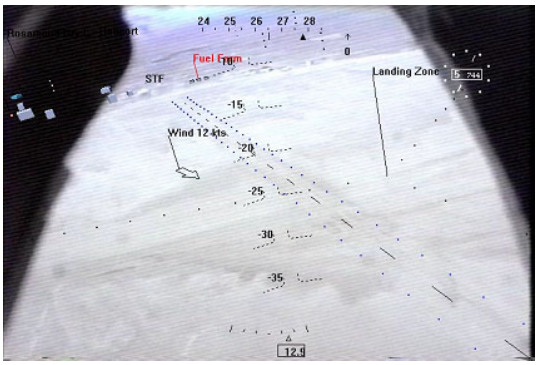

Figure 1.1: NASA X38 display showing overlays on a video stream. Adapted from "Augmented reality" [2].

Compared to the HoloLens style of implementing Augmented Reality by displaying imagery on specialised transparent lenses, a head-mounted display with an attached stereo camera provides more direct control of what the wearer sees. If a stereo camera is mounted onto a head-mounted display, which is set up to view precisely what the camera sees, we can then perform a variety of operations to the images visible to the user. For example, we can render 3D geometry and integrate it directly with the world, or modify the video stream by passing it through an edge detection filter or switching to an infrared camera. If we have a depth camera, we can additionally render objects which are *occluded* by objects in the real world.

1.2 Objective

In this report, we implement and evaluate a prototypical system to implement an Augmented Reality system, making use of a head-mounted display with a stereo and depth camera attached. This system is used to project medical imagery onto a 3D printed polystyrene head model, and allow the wearer to visualise the internal structures of a real patients head such as the skull, the brain or its blood vessels.

Forwarding a pair of stereo images into a users vision field presents a host of problems. In addition to implementing accurate occlusion handling to enable full integration with reality, our prototype needs to be able to automatically determine the focal point and adjust the stereo images accordingly to achieve stereopsis.

2 Background

2.1 Computer Vision

Computer Vision is a subset of Computer Science which incorporates techniques for processing and analysing images. It aims to extract numerical and semantic information in order to make informed decisions on real objects based on captured images. Some applications of Computer Vision include a robot navigating a set of obstacles, or a computer scanning the iris of a user for authentication. Computer Vision techniques are generally higher level compared to image processing, such as Facial Recognition [3]. However, numerous Computer Vision techniques rely on pre and/or post image processing stages, such as the way the Sobel Filter is used in Edge Detection [4].

2.1.1 Homogeneous Coordinates

Homogeneous coordinates, also known as Projective coordinates, is a coordinate system used in projective geometry. Homogeneous coordinates capture the concept of infinity, which does not exist in the Euclidean coordinate system. Let a and w be two numbers. If x is fixed and w is decreasing, then it can be said that x/w increases. Additionally, note that as w approaches 0, x/w approaches infinity. If we let a coordinate $p = \begin{bmatrix} x, w \end{bmatrix}^T$, we can represent all finite numbers by $v = x/w$ for any non-zero value of w, but this representation allows us to represent a point at infinity $p_\infty = \begin{bmatrix} x, 0 \end{bmatrix}^T$ for any non-zero value of x. Notice that multiple pairs of x and w map to the same value of v, this is usually called the *set of homogeneous coordinates* for the point v [5].

Extending this idea to Euclidean points, for each point $\mathbf{p} = \begin{bmatrix} x, y, z \end{bmatrix}^T$, there exists a set of homogeneous coordinates in the form $\begin{bmatrix} xw, yw, zw, w \end{bmatrix}^T$ for any non-zero value of w, which all represent the same point $\mathbf{p}$ in Euclidean Space [6]. Homogeneous coordinates are useful because they can be transformed by transformation matrices in the Euclidean group $\mathbb{SE}_3$. Transformation matrices in $\mathbb{SE}_3$, unlike 3-by-3 matrices, are able to transform a point by a translation as well as a rotation, encoded by a 4-by-4 matrix:

$$T = \begin{bmatrix} R & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{2.1}$$

where $R \in \mathbb{SO}_3$ and $\mathbf{t} \in \mathbb{R}^3$.

To understand why homogeneous coordinates are required for translation, consider a matrix M_t that represents an arbitrary translation where $M_t \in \mathbb{SE}_3$, and use this to transform an arbitrary point $\mathbf{p}_t$ in Cartesian coordinates. If we try to do this using matrix multiplication, we would end up with a dimension error:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2.2}$$

However, if we add an extra w element to $\mathbf{p}_t$ and set it to 1, turning it into a homogeneous coordinate, then the dimensions are correct and the translation happens as expected after the transformation:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix} \tag{2.3}$$

2.1.2 Pinhole Camera Model

In order to implement the majority of algorithms in Computer Vision, it is essential to have a precise understanding of the geometry behind how the 2D pixels relate to the 3D world which they capture. When the human eye views a scene, objects which are further away appear smaller than other objects close by - this effect of size being inversely proportional to distance is known as *perspective*. The **Pinhole Camera Model** is a perspective camera model which describes the mathematical relationship between the coordinates of 3D points and their projection onto the image plane of an *ideal* camera.

The geometry of the pinhole camera is illustrated in Figure 2.1. The centre point at **C** is the location of the camera's *aperture*. The image plane is a plane in front of the camera which is where the virtual image is located after projection. It is fixed at a distance of f from

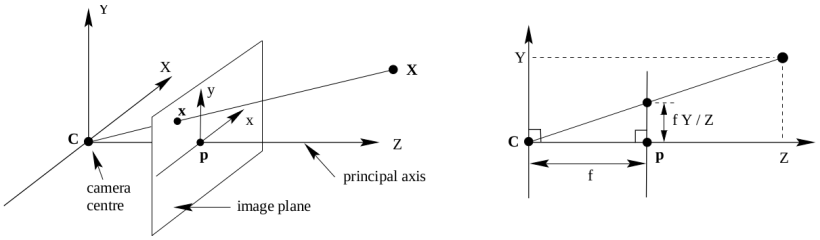

Figure 2.1: Geometry of the Pinhole Camera. Adapted from "Multiple View Geometry in Computer Vision" [6].

the camera centre, this is known as the focal length. **p** is known as the principal point, which is the location where the principal axis passes through the image plane, as some cameras can have an uneven number of pixels on either side of the aperture. Given a point $\mathbf{X} = \begin{bmatrix} x, y, z \end{bmatrix}^T$ in the scene, the *image point* can be defined to be $\mathbf{x} = \begin{bmatrix} xf/z, yf/z, f \end{bmatrix}^T$, which is the location of the intersection of the ray between **X** and **C** and the image plane.

This relation between scene and image points can be expressed as a matrix operation with homogeneous coordinates. If we let **y** be a projected pixel position in 2D homogeneous coordinates (a 3 element vector) and let **x** be a scene coordinate in 3D homogeneous coordinates (a 4 element vector), there exists a 3-by-4 *Camera Matrix* **P** [7] where:

$$\mathbf{y} \sim \mathbf{P}\mathbf{x} \tag{2.4}$$

In this equation, $\sim$ is a relation which says that the two operands are equal up to a non-zero scalar multiplication. In other words, we end up with a homogeneous coordinate with a value of w which may not be equal to 1, therefore to get the final 2D projected point, we need to divide the first two elements by the value of w.

2.1.3 Decomposing the Camera Matrix

So on its own, the Camera Matrix **P** performs a large number of operations, however it is quite difficult to reason about in this form. For example, we cannot discover the camera's current *pose*, or identify any of the camera's internal parameters such as its focal point or principal axis. However, a useful decomposition is to split the Camera Matrix into a *Calibration Matrix*, also known as the Intrinsic Matrix, and an Extrinsic Matrix, which maps a position from the world frame to the camera's own coordinate system. This can be written in the following way:

$$\mathbf{P} = K \begin{bmatrix} R | -RC \end{bmatrix} \tag{2.5}$$

Here, K is a 3-by-3 upper triangular intrinsic matrix which contains the camera's internal parameters, R is a 3-by-3 rotation matrix whose columns are the basis of the camera coordinate system, and C is the camera apertures position relative to the world frame. It is possible to recover K and the extrinsic matrix by taking advantage of the property that K is an upper-triangular matrix and make use of the RQ-decomposition technique on P, but usually both the intrinsic and extrinsic parameters are available [8].

The Extrinsic Matrix

Focusing our attention to the extrinsic matrix, it can be thought as a 3-by-4 transformation matrix which includes a rotation and a translation. This transformation is counter-intuitively used to describe how the world is transformed to be relative to the camera, rather than the other way around. The extrinsic matrix can be thought of as an $\mathbb{SE}_3$ transformation matrix which is multiplied by a 3-by-4 matrix which converts the homogeneous coordinate to a cartesian coordinate by stripping the vector of its w coordinate (which will always be 1):

$$\Big[R|-RC\Big] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & -RC \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{2.6}$$

The decomposition in (2.6) is useful because it allows us to reason about the world to camera transformation like it was any other 4-by-4 matrix, and it is also identical to the *View* matrix from OpenGL, which will be revisited in more detail in Chapter 2.2.

It is convenient to build this world to camera transformation matrix from the camera's current rotation and translation as if it was an object in the scene relative to the world origin. If we let T be the camera's transformation matrix as if it was an object in the scene, R_C be the camera's current rotation where $R_C \in \mathbb{SO}_3$, and C be the camera centre in world coordinates, then we can build the world to camera transformation by simply inverting T as follows [8]:

$$
\begin{aligned}
\begin{bmatrix} R_C & C \\ \mathbf{0}^T & 1 \end{bmatrix}^{-1} &= \left(\begin{bmatrix} I & C \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} R_C & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \right)^{-1} \\
&= \begin{bmatrix} R_C & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}^{-1} \begin{bmatrix} I & C \\ \mathbf{0}^T & 1 \end{bmatrix}^{-1} \\
&= \begin{bmatrix} R & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} I & -C \\ \mathbf{0}^T & 1 \end{bmatrix} \\
&= \begin{bmatrix} R & -RC \\ \mathbf{0}^T & 1 \end{bmatrix}
\end{aligned}
\tag{2.7}
$$

The Intrinsic Matrix

The Intrinsic matrix is a 3-by-3 matrix which is responsible for taking a 3D point in camera space, and projecting it into 2D homogeneous image coordinates. Hartley and Zisserman [6] parametrised the Intrinsic matrix in the following way:

$$
K = \begin{bmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.8}
$$

Here, f_x and f_y are known as the focal lengths for the x and y axis' respectively, measured in pixels. This is counter-intuitive but required as a true pinhole camera is perfectly square (and thus $f_x = f_y$), but in reality, cameras tend to produce an image which has a non-square aspect ratio. It is useful to imagine that the focal lengths for each axis is simply the true focal length of the camera in metres, multiplied by the dimensions of the image.

Next, we take note of the **Principal point offset** which is the vector $\begin{bmatrix} x_0, y_0 \end{bmatrix}^T$. It is incorrect to assume that the origin of the coordinates of the image plane will coincide with the principal point, so an offset is required to align the origin of the image plane with the principal point.

Additionally, there is a skewness parameter s which is sometimes required for more unusual camera set-ups. However, for a normal camera, it is sufficient to assume that this parameter will be 0.

2.2 Computer Graphics

2.2.1 3D Rendering Preliminaries

In 3D computer graphics, the objects displayed on the computer monitor are assembled from an array of n vertices $v = \{x_1, x_2, ..., x_n\}$ where $x_i \in \mathbb{R}^3$. Groups of vertices can be combined together to form different *primitives* such as triangles and then multiple primitives can be combined together to create any shape imaginable.

To display an object on the screen given a set of vertices, we can make use of transformation matrices. There exists a *Model-View-Projection* matrix M_{mvp} which maps a 3D homogeneous coordinate $\mathbf{p}$ to another 3D homogeneous coordinate $\mathbf{s}$:

$$\mathbf{s} = M_{mvp}\mathbf{p} \tag{2.9}$$

The Model-View-Projection matrix is similar to the Camera Matrix from Section 2.1.2, however depth is preserved. This is required to sort rendered objects correctly. It is built from three different matrices, the *Model* matrix, the *View* matrix and the *Projection* matrix:

$$M_{mvp} = M_p M_v M_m \tag{2.10}$$

- The **model** matrix is a transformation matrix in $\mathbb{SE}_3$ which combines translation, rotation and scale transformations for a single object to be rendered.
- The **view** matrix is another transformation matrix in $\mathbb{SE}_3$ which transforms a coordinate from *world space* to *camera space* (relative to the camera). This is identical to the 4-by-4 matrix which is part of the Extrinsic matrix from Section 2.1.3.
- The **projection** matrix is responsible for taking the objects vertices in *camera space* and transforming them into *clip space*. Once vertices are in clip space, they are then converted into *Normalised Device Coordinates* (NDC) by converting from 4D homogeneous coordinates to 3D Cartesian coordinates. This step is known as *perspective division*. Conventions for NDC space differ, but for OpenGL: vertices in NDC coordinates range from $[1, 1]^T$ at the top-right of the screen to $[-1, -1]^T$ at the bottom-left of the screen. The projection matrix also preserves depth, where the nearest depth value is -1 and the furthest depth value is 1 in NDC space.

2.3 Tracking and Mapping

In order to build any kind of Augmented Reality system, it is extremely important for the hardware device (known from this point onwards as the *Agent*) attached to the user to be able to understand its location in the world with respect to some fixed coordinate system, as well as building a model of the scene it is in. Without any understanding of the agent's own position and orientation, it isn't possible to seamlessly integrate a virtual scene with a physical scene.

2.3.1 Simultaneous Localisation and Mapping

Simultaneous Localisation and Mapping (SLAM) are a class of problems in robotics where a moving agent can simultaneously construct a representation of its environment as well as figuring out its current pose and location, from a continuously expanding set of data and with no pre-computation [9].

MonoSLAM is an algorithm which is an implementation of the Simultaneous Location and Mapping methodology to be able to recover the trajectory and motion of an uncontrolled camera in a completely unknown environment, but is efficient enough to be run in real time at around 30Hz [10]. An alternative algorithm which isn't specially a SLAM algorithm but is related is called Parallel Tracking and Mapping (PTAM) [11]. PTAM is designed for Augmented Reality applications in a small workspace, and is also more efficient than MonoSLAM as it splits the *tracking* and *mapping* tasks which allows them to be allocated into two different parallel threads, which is useful for modern computers which usually have two or more concurrent threads of execution.

2.3.2 KinectFusion

Both MonoSLAM and PTAM are able to solve the problem of locating the agent in an arbitrary environment without any prior knowledge given a single RGB video stream, as well as identifying feature points in a scene (See Figure 2.2). However, to discover 3D objects in the scene, we need an algorithm which can recover much more detail from the RGB video stream. Unfortunately, the resulting scene representation in MonoSLAM has a very low granularity, which is slightly improved in PTAM. However, it is still not possible to do accurate object to mesh registration to allow a convincing augmented reality experience without further increasing the detail recovered from the scene.

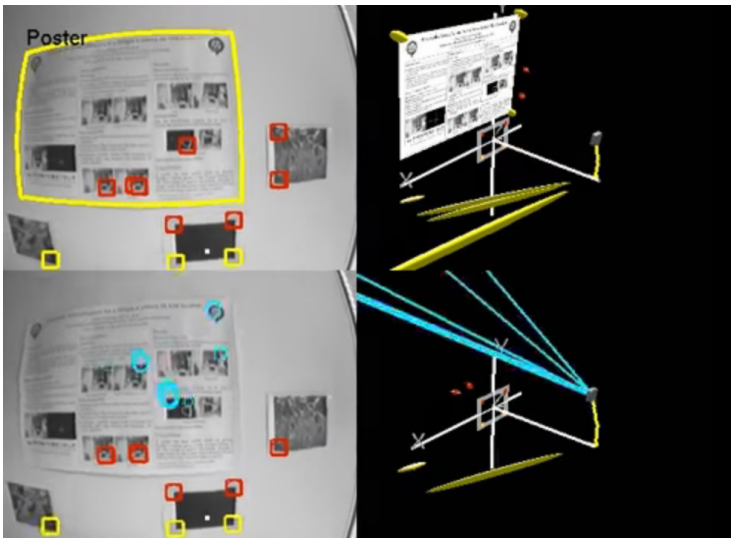

Figure 2.2: Tracking points from an implementation of the MonoSLAM algorithm. Adapted from "Object tracking with a single camera SLAM (ICRA 2007)" - URL: `https://www.youtube.com/watch?v=KhX_rLqhiaU`

KinectFusion is an algorithm which - making use of inexpensive hardware with depth perception, such as Microsoft's Kinect Camera - can build a dense 3D map of surfaces and simultaneously track the location of the camera in real time. The algorithm works by taking a noisy input depth map, performs real time dense SLAM by tracking the new position of the camera based on the previous cameras pose, and incrementally produces a 3D model of the scene in the form of a *Truncated Signed Distance Function* (TSDF) [12]. The TSDF can be thought of as function $D : \mathbb{R}^3 \mapsto \mathbb{R}$ that returns the distance to the nearest surface. This approach to SLAM, unlike MonoSLAM and PTAM, is more suitable because it generates a very detailed model of the scene in real-time when executed on the GPU on desktop class hardware.

KinectFusion's biggest weakness is its susceptibility to losing the current tracking state. The tracking part of the algorithm is designed with the assumption that subsequent frames will only have a relatively small amount of movement. If the camera is moved too fast between frames, then KinectFusion will no longer be able to calculate a new pose for the camera. At this point, tracking is now "lost" and the algorithm can take many more frames before it is able to recover tracking capabilities again.

2.3.3 Iterative Closest Point

In order to project additional information in 3D onto an object in a scene, it is necessary to not only know the precise location of the target object in the scene relative to the current location of the camera, but also the camera's pose relative to where the system was initialised. One approach to approach this would be to register a 3D mesh with a reconstruction of the scene from an algorithm such as KinectFusion in Section 2.3.2. **Iterative Closest Point (ICP)** is an algorithm which is designed to minimise the difference between two different point clouds in an effort to align them. One point cloud, named the *reference*, is fixed in place whilst a second point cloud, named the *source*, is transformed in an attempt to fit the reference [13].

In Marker Free Augmented Reality, this algorithm is particularly useful to match a surface mesh reconstruction from a given RGB-D SLAM system such as KinectFusion (the *reference*) against a previously created 3D model of this object (the *source*). If the ICP algorithm manages to discover a 4-by-4 transformation matrix T which maps from the *source* to the *reference*, and we initialise the source with a transform relative to the camera, named T_s, then the true transformation of the reference T_r is as follows:

$$T_r = T_c T_s T \tag{2.11}$$

2.4 Stereo Vision

Stereo Vision, also known as *Stereopsis*, is an effect where separate images from two eyes are combined into a single 3D image in the brain to give a sense of depth. Our eyes are separated 6.5cm apart and capture a view of the world from two slightly different perspectives. When we observe an object, both eyes converge to the position of this object, which reduces the disparity of the object in both views and allows the brain to construct a 3D image. This effect is seen in Figure 2.3. Additionally, each eye adjusts the focal point of its lens to match the depth of the object, bring it in focus.

A stereoscopic image pair is defined as a pair of images captured by two horizontally displaced cameras. By presenting the corresponding image to each eye, it allows the user to experience Stereopsis. When viewing a stereoscopic image pair, the users eyes are focused on the display, but can converge anywhere on the Z axis. this can cause the eyes to turn excessively inwards, or diverge depending on the disparity of an object in both images. Therefore, it is necessary to apply some post processing to the image pair to minimise the perceived

disparity and allow the eyes to converge correctly to the depth of the display.

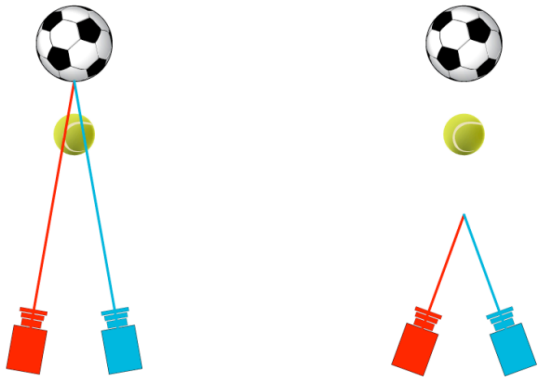

Figure 2.3: An illustration of the points of convergence for different convergence angles. Adapted from "Basic Principles of Stereoscopic 3D" [14]

2.4.1 Horizontal Image Translation

In order to reduce this breakdown between the focus point of each eye and the convergence angle, we can perform **Horizontal Image Translation** as a post process [15]. Disparity is defined as the distance, in pixels, between two corresponding points in a stereoscopic image pair.

There are three possible situations for the disparity of an image point: A positive disparity indicates an object beyond the screen surface, a zero disparity indicates a point on the screen surface, and a negative disparity indicates an object closer than the screen surface, as shown in Figure 2.4. Positive and negative disparity can cause the eyes to unnaturally converge or diverge respectively and create an uncomfortable experience for the user.

Let s be the distance between the stereo camera pair and the screen surface, l be the position of a point on the left image, r be the corresponding position of the same point on the right image, and e be the inter-ocular distance (the distance between both cameras). We can express the relationship between the on-screen disparity between the two points $d = r - l$ and its perceived range z as:

$$z = s\left(\frac{e}{e-d}\right) \tag{2.12}$$

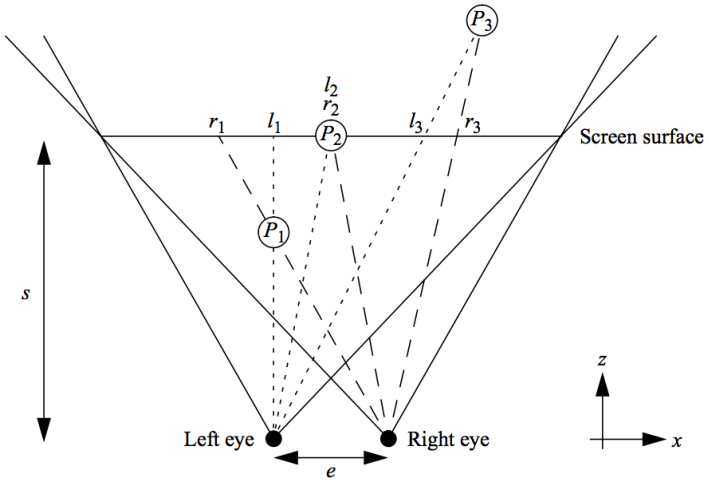

Figure 2.4: An illustration of disparity for objects at different distances from each eye. P_1 has a negative disparity, P_2 has zero disparity and P_3 has positive disparity. Adapted from "Algorithm for automated eye strain reduction in real stereoscopic images and sequences" [15].

Note that d is measured in the physical distance unit. To reduce the disparity of this particular point, we can perform Horizontal Image Translation by shifting the left image to the right by $d/2$ and shifting the right image to the left by the same amount:

$$d' = \left(r - \frac{r-l}{2}\right) - \left(l + \frac{r-l}{2}\right) = 0 \tag{2.13}$$

3 Building the Prototype

3.1 Hardware

To build a fully integrated marker-free augmented reality system with no pre-computations or assumptions of the scene it is located in, we need to choose an adequate set of hardware which meets the objective stated in Section 1.2. This section compares different implementations of a head mounted display, and an array of cameras suitable for stereoscopic vision and depth sensing.

3.1.1 Head Mounted Displays

Oculus Rift

The Oculus Rift is a virtual reality headset developed by Oculus VR, which was originally pitched as part of a Kickstarter campaign which raised $2.5 million towards the development of the device [16]. The Rift is marketed as a consumer level VR headset built using "off-the-shelf" components. Compared to earlier prototypes of Virtual Reality devices, they were either not designed for *presence*, an immersive experience designed to fool a user's brain by tricking them into believing that they're in a simulated world, such as the Vuzix iWear VR920 [17], or the resolution or frame rate of other competing devices were just too low.

The Oculus Rift has gone through various prototypes, released as Developer Kits since the Kickstarter campaign. The first developer kit was released to the public in March 2013 after shipping to Kickstarter backers who pledged $300 or more, and to pre-order customers on Oculus' website for the same price. The DK1 has a display resolution of 640-by-800 per eye, and a horizontal field of view of 90° [18]. Due to the low resolution of the headset, some percentage of wearers would get motion sickness due to the refresh rate being limited to 75Hz, as well as experiencing the *Screen Door Effect*, where the individual pixels are discernible to the wearer.

As of November 2015, the latest developer kit was the Developer Kit 2 (DK2), which began shipping in March 2014. The DK2 has a resolution of 960-by-1080 per eye, on a single 1920-by-1080 OLED display, with a 100° field of view and a refresh rate of 75Hz. Inertial tracking is done through an array of three sensors, a Gyroscope, Accelerometer and Magnetometer, with a polling rate set to 1000 Hz.

The DK2 additionally comes with a specially designed near infrared CMOS sensor [19]. The headset contains an array of LEDs built into the shell which is outside the visual range of humans but can be detected by the infrared sensor. This is designed to optionally provide accurate location tracking for the Oculus Rift to complement the information already obtained from gyroscopes and accelerometers which already exist on the headset. Nevertheless, the field of view of the infrared sensor is not very wide, which requires the wearer to stay in front of the sensor in order to gain any beneficial effects.

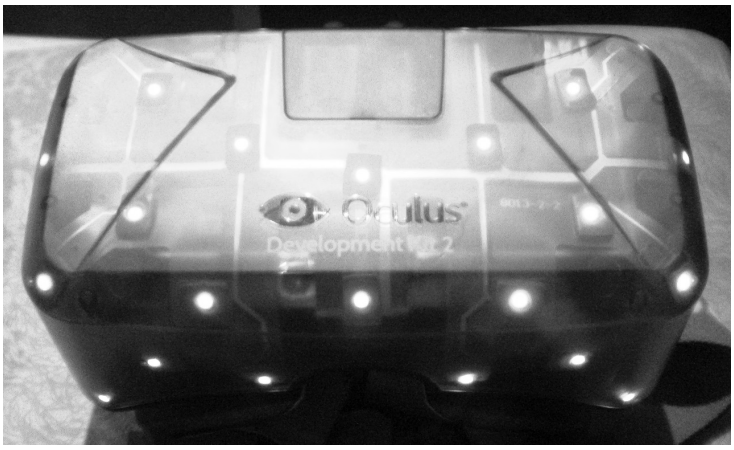

Figure 3.1: Infrared markers on the Oculus Rift. Adapted from "First Impressions from the New Oculus Rift DK2 3D VR HMD" [20].

Oculus provides a C++ Software Development Kit (SDK) for Windows (and formerly Mac OS X and Linux), which allows programmers writing graphics programs using either Direct3D or OpenGL to interface directly with the DK1 and DK2 hardware, such as displaying the final output directly onto the Rifts display, or querying the device for its current pose or other intrinsic information such as the inter-pupillary distance. The SDK hooks onto the graphics API and distorts the image being displayed on the headset using *barrel distortion*, which is then corrected by the *pin cushion* effect by the lenses in each eye (see Figure 3.2).

(a) Barrel Distortion (b) Pincushion Distortion

Figure 3.2: Barrel and Pincushion Distortion visualised as a grid. Adapted from "Rendering to the Oculus Rift" [21]

Google Cardboard

Cardboard is a virtual reality platform developed by Google to be used with a head mount designed to hold a smartphone. It is designed to be a very low-cost system to encourage wider interest in VR applications, named after Google's own fold-out cardboard head mount design [22]. Tracking the change in orientation of the users head is done through the use of the phones already existing hardware, and interaction is done through the use of a detached magnet which influences the magnetometer of the phone and allows the platform to register a single *click* through the action of a user moving the magnet.

Google provides an SDK for multiple platforms which allows VR applications to leverage the phones existing hardware, as well as to barrel distort the images for each eye before they are displayed and corrected by the lenses in the head mount. This idea of "building your own headset" is interesting as it allows us to design our own head mount to integrate a camera easily and be as light as possible. However one huge limitation with this approach is that a smartphone is used as the display. The CPU and GPU in a smartphone is much slower than desktop class hardware, which can cause frame rate issues with some algorithms for localisation.

Samsung Gear VR

The Gear VR is a virtual reality headset developed by Samsung and Oculus, released in November 2015. It is designed to take a high resolution display from a Samsung Galaxy S-series phone and combine it with a pair of lenses and integrated sensors such as a gyroscope and accelerometer [23]. The latest Samsung Galaxy S-series phone is the S6, which comes with a display resolution of 1440-by-2560 pixels, which is 1280-by-1440 in each eye, and the lenses provided provide the user with a 96° field-of-view.

Figure 3.3: The Samsung Gear VR head mount with a Galaxy S6 phone visible. Adapted from "Gear VR — Samsung US" - URL: `http://www.samsung.com/us/explore/gear-vr/`

To interface with the Gear VR hardware, Oculus provides a C++ SDK called the *"Oculus Mobile SDK"*, which was released to developers in November 2014. The Mobile SDK works very similarly to the PC SDK used with the DK1 and DK2, but it is designed with the Gear VR hardware in mind.

3.1.2 Stereoscopic Cameras

Stereolabs ZED

The ZED is a hardware synced stereoscopic camera developed and manufactured by StereoLabs. It is a lightweight camera which is designed for depth sensing via the use of stereo vision techniques, but also functions as a high resolution stereoscopic camera for AR and autonomous navigation. It outputs a high resolution side-by-side video stream through USB 3.0 which can be configured to operate in a number of different configurations which are reproduced in Table 3.1 [24].

The ZED's RGB video stream can be accessed like a normal webcam, as it is a UVC-compatible camera that Windows, Mac OS X and Linux all provide built in drivers for. However, StereoLabs provides an SDK for developers to use, which provides access to the colour stream for each eye, but additionally implements a Stereo Vision algorithm on the GPU using CUDA to compute a detailed depth stream with a resolution that matches the currently configured resolution of the ZED. The effective range of this synthetic depth stream

Video Mode	Frames per second	Output Resolution
2.2K	15	4416 x 1242
1080p	30	3840 x 1080
720p	60	2560 x 720
VGA	120	1280 x 480

Table 3.1: A matrix of the different configurations of the output video stream from the ZED

begins at 1 metre, and it is accurate up to 15 metres. This depth sensing capability is fantastic for robotic vision which requires a depth stream with these specifications to accurately map and understand the environment it is in [25].

Duo MLX

The Duo MLX is a small and light stereoscopic camera designed by Code Laboratories Inc. It has a modest resolution of 640-by-480 by eye at a high framerate of 60 FPS, with an exceptionally large field of view of 170° [26]. The Duo comes with a C/C++ SDK which provides low level APIs to access the sensor, and a higher level API which implements a Stereo Vision algorithm to build a synthetic depth stream much like the ZED. The depth stream has a working range between 25cm and 2.5m [27].

3.1.3 RGB-D Cameras

Intel RealSense F200 Camera

Intel RealSense is a platform of technologies designed to implement gesture-based human-computer interaction (HCI) techniques. The platform consists of a variety of consumer-level 3D cameras - such as cameras mounted onto tablets such as the Dell Venue Pro 8000 - and software provided by Intel to interface with this class of hardware, having implement a simple perception library which can be used by application developers.

Creative Labs designed a standalone camera for the platform, named the Intel RealSense F200. Specifications include a full VGA depth resolution with a range between 0.2 and 1.2 meters, and an 1080p RGB camera. Compared to the ZED camera, the RealSense F200 is more useful for close quarters, because its minimum range is only 20cm compared to the ZED which has a minimum range of 1m. Of course, the F200 also has a much lower maximum range of only 1.2m compared to the ZEDs maximum range of 15m, but this is not an issue for an application which has a small workspace [28].

Intel provide a C++ SDK for interfacing with the Realsense platform, providing features such as generating a point cloud from depth, and higher level implementations of algorithms such as facial recognition. The SDK is bundled in a large installation file which includes lots of examples and utilities for making the most of the camera. However, Intel also provides a much lower level C++ library called "librealsense" which is solely designed to interface with the camera without having to download lots of extra unneeded software [29]. librealsense exists in GitHub which allows anybody to access and contribute to the source code, I even contributed a fix which dealt with a linker error when the library is used in more than one compilation unit.

Microsoft Kinect

The Kinect is a range of gesture based input devices developed by Microsoft originally for the Xbox 360 console, but later brought to the Windows and Xbox One platforms. The original Kinect hardware for the Xbox 360 works by using an IR projector to display a pattern of infra-red light into a room. This pattern is then read by an IR sensor which then analyses distortions in the infra-red patterns caused by the shapes of objects in the scene to build a depth map, which is then sent to the host device for further processing [30].

As the specifications of the Kinect hardware are not public, reverse engineering has determined that the its depth map has a frame rate between 9 and 30 frames per second. The depth stream has a VGA resolution (640-by-480 pixels) with 11 bits for depth, which allows 2048 different depth levels to be encoded [31]. The Kinect depth sensor has a range from 0.8m to 4m, which is designed for a typical size living room where the players will not be too close to the sensor. Alternatively, Kinect for Windows also can be switched to "Near Mode" which instead works at a range of 0.5m to 3m.

Unfortunately, the Kinect is a rather large and heavy device -weighing about 1.4kg - so its impractical to mount it onto a head-mounted display as it would make for a very uncomfortable experience. Additionally, the minimum range of the Kinect sensor is too great for an application with close quarters where objects are typically only tens of centimetres in front of the depth sensor. As a result, the Kinect range of devices are unsuitable for our prototype.

3.1.4 The Prototype

As of November 2015, the only head mounted displays (HMDs) worth considering that were on the market are the Oculus Rift DK1 and DK2, and the Samsung Gear VR. The Samsung Gear

VR and other phone based HMDs designs like the Google Cardboard are cheap, however they are extremely limited in processing power compared to traditional PCs with powerful PCI-E based GPUs. Because of this limitation, it would be difficult to make use of an algorithm like KinectFusion for localisation and tracking without taking a large hit on the frame rate. The DK1 is cheaper as it is an older device, however it suffers from a number of drawbacks including a relatively low display resolution and no motion tracking, only orientation tracking. The Oculus DK2 builds upon the DK1 design by making it lighter, increasing the display's resolution and extending the tracking capabilities of the device, so we chose to use the DK2 as the frame for the prototype.

For video pass through, the best choice of a Stereoscopic camera was the Stereolabs ZED camera due to its huge resolution and high frame rate. However, its depth sensing is limited to a minimum range which is too large for the demo environment. Therefore, we chose to add the Intel Realsense F200 and make use of its very accurate depth stream as its range was far more suitable for close quarters AR. The entire system is held together using a 3D printed bracket provided by my supervisor, which fits snugly into a concave part of the headset. Additionally, to reduce the weight of the prototype, the casing and cooling for the Realsense F200 has been stripped off, producing the final prototype in Figure 3.4. When testing the augmented reality features of the prototype, a 3D-printed polystyrene head model has been provided, with a corresponding 3D model file for the head itself, and various internal structures of the brain.

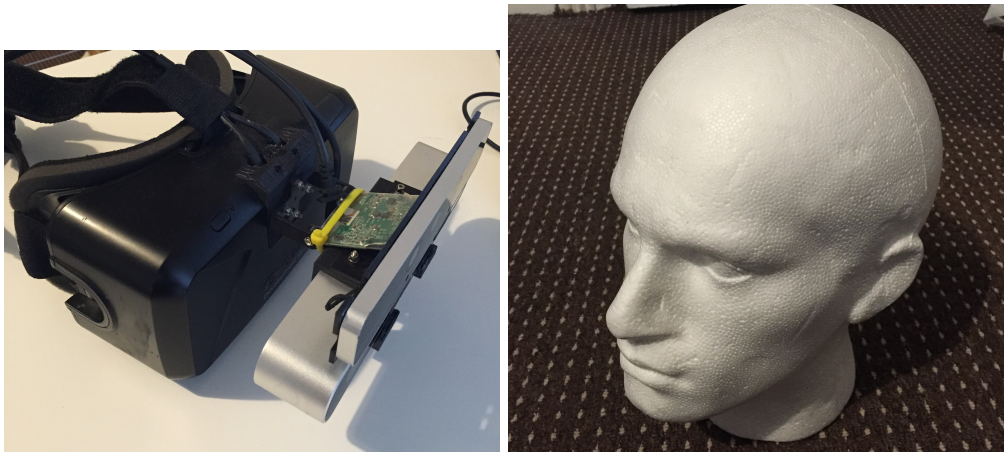

Figure 3.4: The fully built prototype with a Stereolabs ZED and Intel Realsense F200 (with cooling stripped) attached (left) and the polystyrene head used for testing (right).

3.2 Software

3.2.1 OpenGL

OpenGL is a low level C API designed by the Khronos Group for graphics programming across many platforms. It is usually implemented by the GPU vendor for each operating system as part of the graphics driver (with the exception of Mac OS X where the OpenGL driver is implemented by Apple), but software implementations also exist for certain platforms. We decided to make use of OpenGL as it is easy to use and is very portable which allows the projects source code to be useful for more system configurations. It is by all means not the only existing graphics API, there are a few others to note:

- **Direct3D** - Direct3D is another low-level API for graphics programming which is part of the DirectX collection of APIs designed by Microsoft for Windows. As of March 2016, the latest version is Direct3D 12 which comes with Windows 10, which has seen a convergence to almost zero driver overhead (known as *AZDO*), which allows a lower level abstraction of the GPU hardware and allows the application to reduce CPU utilisation, rather than providing a one-size-fits-all high level API [32].

- **Vulkan** - Vulkan is a new low-level C API designed to supersede OpenGL by the Khronos Group which aims to implement the AZDO idea to allow the application to reduce CPU utilisation and address some long term faults with OpenGL. Vulkan is based upon and extends the Mantle API, originally developed by AMD to reduce CPU overhead in graphics APIs. Vulkan works across many platforms including Windows, Android and Linux, with the notable exception of Mac OS X and iOS, where support has not been announced yet as of March 2016. As Vulkan is a much lower level of abstraction compared to OpenGL, it is sometimes not feasible to use Vulkan to build quick prototypes, as the application is responsible for managing the GPU memory and other tasks that were previously assigned to the graphics driver when using OpenGL.

- **Metal** - Metal is yet another low-level API for graphics and compute programming designed by Apple initially for iOS and later Mac OS X to reduce CPU utilisation and power consumption on mobile devices and make better use of the hardware. Metal improves upon OpenGL by allowing shader programs to be precompiled, unlike OpenGL which compiles shader programs when they are loaded into the application. It also implements the AZDO idea much like Direct3D 12 and Vulkan, by allowing the application fine grained control over GPU memory and synchronisation, rather than allowing the driver to guess and deal with every possible scenario [33].

OpenGL is a graphics programming library, however it is not responsible for window management and relies on an already existing window for the target platform created using Win32, Carbon, Cocoa or X11. Initialising an OpenGL context also requires context bridge such as WGL on Windows, GLX on Linux and EGL to ensure that the rendered content is displayed. To abstract over the complexity of all these combinations of libraries, libraries exist to make this easier for us. We decided to make use of GLFW for this task, as it provides a very simple and easy to use API to create a window and initialise an OpenGL context without having to worry about the details of the target platform [34]. Alternative libraries that exist are SDL2, GLUT and many others.

3.2.2 The Graphics Pipeline

Here, we give a quick introduction to the pipeline used by OpenGL 3.3 onwards and how it is programmed in the various stages of the pipeline using the GL Shader Language (known as GLSL) [35]. We have omitted a few optional stages as they are beyond the scope of the prototype.

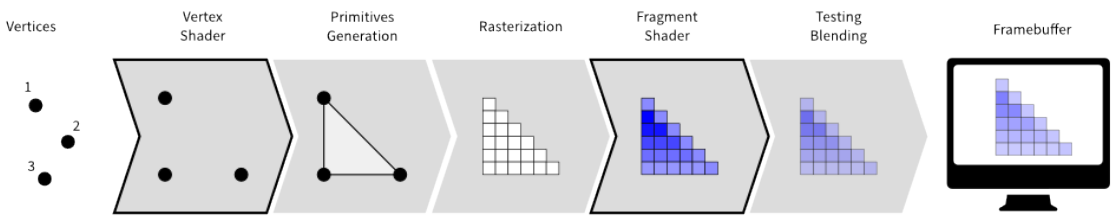

Figure 3.5: The OpenGL graphics pipeline. Adapted from "Modern OpenGL" [36]

Vertex Specification

The first stage of the pipeline is where vertex and optionally element data are specified. We would create a *Vertex Array Object* (VAO) which acts as a "container" for the data required to render the object. To provide the actual vertex data such as positions, normals, etc. we need to create an object called a *Vertex Buffer Object* (VBO) and set the offsets for accessing each element in each vertex. Optionally, an *Element Buffer Object* (EBO) can also be created to reuse vertex data in the bound VBO, such as creating a rectangle out of 2 triangles using only 4 vertices, rather than 6. Once all this set up has been completed, the object can be rendered to the screen by only binding the VAO which then implicitly binds the VBO, EBO

if it exists and any vertex attribute offsets set up previously.

Vertex Shader

The vertex data encapsulated in the VBO is then passed to the vertex processing stage. Each vertex is processed using a **Vertex Shader** which is a small chunk of code run on the GPU which takes a vertex as input, and a transformed vertex as output. Usually, the Vertex Shader stage is used to transform each vertex by the *Model-View-Projection* matrix, but due to the programmable nature of shaders, many other effects are possible such as displacement based on the normals, or calculating texture coordinates. One limitation of the vertex shader stage is that it can only modify existing vertex data, it cannot add or remove any vertices.

Primitive Generation and Rasterisation

Once the vertices have been passed through the vertex shader, further post processing is done, such as the *perspective division* from Section 2.2.1. Vertices are assembled into primitives such as points, lines or triangles depending on the application preferences, and optionally a **Geometry Shader** can be written to take primitives as input, and output zero or more primitives. The primitives are then passed to the rasteriser, which then creates rows of fragments that is covered by the primitives, and interpolates the parameters produced by the vertex shader.

At this point, if the *depth buffer* is enabled, and the fragment shader doesn't write any depth information, depth testing occurs at this point before the fragment shader is executed. This is known as **Early Z Rejection**. A depth test is a condition configured by the application which decides whether a fragment should be written to a frame buffer by comparing the new depth value with an existing depth value from a previously rendered object. The default depth test is *less than*, which means that only fragments with a depth value less than the existing depth value are then written to the backbuffer.

Fragment Shader and Blending

Once the fragments are generated by the pipeline, they are each passed through a **Fragment Shader**. The Fragment Shader takes the interpolated parameters from the Vertex Shader as input, and it is commonly used for lighting calculations and looking up image data from textures. The Fragment Shader then generates a final colour value which is written to one or more *frame buffers*. Fragments generated by the Fragment Shader are then blended together

with the existing fragments in the target frame buffers using the configured blending mode e.g. add the values of the existing and new fragments together, or combine using alpha mapping. If the fragment shader writes any depth information by itself, depth testing is performed here instead.

3.2.3 Resource Binding

Resources are exposed to OpenGL as either buffers or textures. Buffers exist as a sort of general data storage which are specialised for different uses. The most common buffer types are **Vertex Buffers** for storing vertex information, and **Element Buffers** for storing indices into Vertex Buffers. Another common buffer type is a **Uniform Buffer** which is used to encapsulate *uniform blocks*. A uniform block is a block of uniform parameters which are common to a large chunk of shaders. For example, a View-Projection matrix can be stored in a uniform block and the uniform block can be reused for every object visible from a single camera, without having to pass the matrix data into each shader [37].

Another common resource in OpenGL are textures. A texture is an OpenGL object which contains one or more images which can be used either as a source of a texture lookup from a shader, or as a render target known as a *framebuffer object*. To load an image resource in OpenGL, we would create a texture object and copy the pixel data in using the relevant API. From that point onwards, binding the texture object would allow shader programs to access the texture information. Additionally, if we would like to implement a kind of post processing effect, the scene can be rendered to a framebuffer which can then be taken as input to another fragment shader [38].

3.2.4 CUDA

CUDA is a parallel processing framework created by Nvidia for use with its GPU product ranges for general purpose computing. In CUDA terms, the CPU and RAM is known as the *host*, and the GPU being controlled to is called the *device*. To implement a parallel algorithm, we write code using a special dialect of C/C++ called CUDA C/C++, which is compiled using Nvidia's own `ncc` compiler. `ncc` works by looking for device functions called *kernels*, which are prefixed with `__device__`. It then takes kernel functions and separates those from C/C++ code meant for the host, and compiles them separately. The host code is then sent to the compiler for the target platform, such as `cl.exe` on Windows or `gcc` (by default) on Linux. CUDA organises kernel launches into many threads which all execute the same sequential program in parallel. Many threads are grouped together into *blocks*, which are executed on

a single Streaming Multiprocessor on the GPU. Threads in a single block can synchronise with each other and exchange data, but communication between blocks is expensive. Kernel launches are asynchronous, meaning that control returns to the host application immediately, but many consecutive kernel launches are streamed, meaning that they will run in the order that they are launched. Other CUDA operations such as memcpy are also streamed and will block until previous kernel launches have been completed.

There are at least two other routes for general purpose GPU programming that we can use. OpenGL has this capability already built-in, in the form of **Compute Shaders**. A compute shader is similar to other shader types, but operate independently of the graphics pipeline. Invocation of the shader are split into 3-dimensional *work groups*, and the compute shader itself has a *local size* which defines how many times the compute shader is executed in parallel in each work group. This is more limited compared to CUDA and OpenCL as communication between the host and the device is constrained to the OpenGL interface [39].

Open Computing Language, also known as OpenCL, is an open standard for GPGPU programming developed by the Khronos Group for implementation on a wide variety of hardware. OpenCL is very similar to CUDA where compute programs (OpenCLs analogue to CUDA kernels) are written using a variant of C++ [40]. As CUDA has been around for much longer than OpenCL, it has more mature tools, including a memory check tool similar to Valgrind, and a debugger.

3.2.5 OpenCV

Open Source Computer Vision (OpenCV) is "an open source computer vision and machine learning software library" [41]. The library is split into a number of modules which are used for various purposes. Our prototype makes use of **core**, the module containing the core functionality of the library; **calib3d**, a module which implements mono/stereo camera calibration and object pose estimation and **highgui**, a module designed to provide a High-level GUI interface for rapid prototyping. We make heavy use of **calib3d** in the calibration step, and **highgui** when visualising debugging information for KFusion. Outside the calibration procedure, we make use of the `Mat` class for convenient access to image data.

3.3 High Level Architecture

This report focuses mainly on the methods that we used to implement the prototype successfully, but as a reader it is useful to understand the structure behind the software implementation powering the prototype. We created two libraries named **Framework** and **Camera**, and used these to build two applications: **StereoCalibrate** and **RiftAR**. In this section, we give a brief overview of these different components and what they provide.

Framework

OpenGL is a low-level graphics API, and because of this there are a lot of details that need to be addressed by the application. For example, the application needs to load OpenGL function pointers from the graphics driver, create a window, initialise an OpenGL context and set up a main loop. Furthermore, this excludes any setup required to display a single rectangle on the screen, or load 3D models. To deal with these complexities, we implemented a library called **Framework** that is designed to abstract all this boilerplate into an easy to use suite of classes.

To build a new OpenGL application, Framework provides an interface named `App` that provides hooks for initialisation, updating and input events. Rendering a 3D model and a 2D rectangle is handled by the `Model` and `Rectangle2D` classes respectively. OpenGL shader programs are wrapped into a convenient class named `Shader` which deals with compilation, linking, and passing uniform parameters in a type-safe manner. Lastly, a `Model` and `Shader` is bound together by a class named `Entity` which abstracts the involved render process into simply loading a model, positioning it somewhere, and rendering it.

Camera

Interfacing with various cameras requires making use of a number of different external libraries. To prevent cluttering the application with various calls to these other libraries, it is useful to abstract them into some common interface. We wrote a second library called **Camera** specifically for this purpose.

The library exposes an interface named `CameraSource` which contains methods to capture frame data, copy frames into an OpenCV matrix, and access the intrinsic and extrinsics of different streams in a uniform way. Intrinsics are provided as a data type named `CameraIntrinsics`, which conveniently provides a helper method to generate an OpenGL

projection matrix from its parameters. Extrinsics are provided as a 4-by-4 transformation matrix, which can be used as-is to transform homogeneous coordinates.

We provide an implementation of `CameraSource` using the ZED SDK to access the ZED camera and query the device for its properties such as the baseline or convergence angle. As the ZED SDK stores frame data on the GPU, the implementation uses a `cudaGraphicsResource` to copy data directly to an OpenGL texture without having to copy frame data to the host, and back to the device. We also provide an implementation using the `librealsense` library to access the Realsense camera.

StereoCalibrate

StereoCalibrate is a small application which makes use of the Camera library to capture a number of frames from the left ZED camera and the Realsense colour sensor containing a checkerboard, which are subsequently converted to the OpenCV format and passed into the stereo calibration functions provided by OpenCV to generate the extrinsic parameters mentioned later in Section 4.1.3.

RiftAR

The RiftAR application is the main application which powers the prototype. It implements a number of classes to abstract away the different stages of its pipeline. There exists a `Renderer` class which is responsible for storing information related to the virtual scene such as the overlay, and other parameters required for rendering such as the current view and projection matrices. A class named `KFusionTracker` wraps the implementation of KinectFusion, and provides the methods to find a given object in the scene. Warping the depth from the Realsense camera to match the ZED camera described in Section 4.3.1 is implemented in another class named `RealsenseDepthAdjuster`. Finally, displaying a frame on the Oculus Rift is implemented in a class named `RiftOutput`, which encapsulates the initialisation and rendering steps required by the Oculus SDK.

4 Implementation

4.1 Calibration

To build the prototype, it is essential to have a thorough understanding of how the different camera types relate to each other. In Section 2.1.3, we discussed the ideal pinhole camera and how the cameras internal parameters and pose can be represented using the **Intrinsic** and **Extrinsic** matrices. Here, we discuss how calibration is done using OpenCV to acquire the intrinsics of all the cameras, and the extrinsics that connect them all together.

4.1.1 Finding the Checkerboard Pattern

To calculate the intrinsic and extrinsic parameters of a pair of cameras, it is required to have captured different poses of a predictable pattern in view of both cameras. OpenCV comes with two functions designed for this purpose, one to find chessboard corners and another to find centres in a symmetric or asymmetric grid of circles. We decided to use the `cvFindChessboardCorners` function which, given a greyscale input image, attempts to search for the corners of a chessboard pattern of a given size, and returns an array of points.

Optionally, this function first calls a function named `cvCheckChessboard` which attempts to quickly search for whether a chessboard pattern actually exists in the given image. It does this as an optimisation step, because `cvFindChessboardCorners` is especially slow in an image which contains no pattern at all. After verifying that a checkerboard pattern likely exists in the image, a threshold filter is applied to the input image:

$$g(x,y) = \begin{cases} 1 & \text{if } f(x,y) > T(x,y) \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

In this equation, $f(x,y)$ is the source pixel value at (x,y), $g(x,y)$ is the output pixel value, and $T(x,y)$ is a threshold value defined for that particular pixel. Depending on the

options provided to `cvFindChessboardCorners`, this can either be a fixed threshold where $T(x, y) = \bar{x}$, and $\bar{x}$ is the mean value of all pixels in the image, or an adaptive threshold where the threshold value $T(x, y)$ is the mean of a neighbourhood of pixels of a specified size [42].

Next, the algorithm performs a **Dilate** operation to split the corners apart. Dilation is a common type of a morphology operator, which is used for image processing. A morphological operation processes an image based on shapes. They work by applying a *structuring element* known as a kernel to an input image to generate an output image. A kernel can have any shape or size, and has a defined *anchor point*. As the kernel is scanned over an image, we compute the maximum value overlapped by the kernel, and replace the image pixel at the kernels anchor point with this value [43]. Erosion is another morphological operator similar to Dilation which instead replaces the pixel at the anchor point with the minimum value overlapped by the kernel. The effects of these two operators are seen in Figure 4.1.

(a) Original (b) Dilation (c) Erosion

Figure 4.1: The effects of dilation or erosion operators on a fingerprint image.

Once the corners have been separated, this function then calls an internal function named `icvGenerateQuads`, which is responsible for finding the squares in the image by returning the corners of each square in clockwise order. Next, the code goes through a series of checks to condense these quads into chessboard corners by calling auxillary functions including `icvFindConnectedQuads`, `icvCleanFoundConnectedQuads` to remove extra corners, and `icvCheckQuadGroup` [44].

4.1.2 Calculating Extrinsics

Once a selection of pairs of views of chessboard corner points have been captured by `cvFindChessboardCorners`, we are now able to estimate the intrinsic and extrinsic parameters. OpenCV provides another function named `cvStereoCalibrate` loosely based on [45] for this purpose which takes untransformed calibration pattern points, and a list of observations

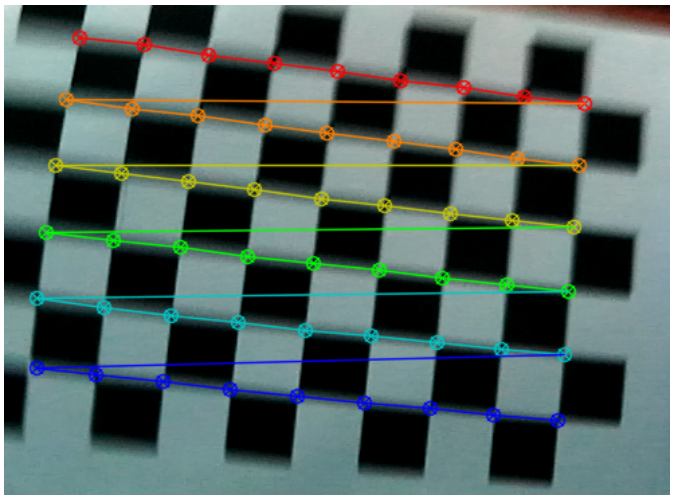

Figure 4.2: A checkerboard pattern recognised by the algorithm in 4.1.1.

of this pattern from both cameras in the stereo pair. It generates an intrinsic matrix for both cameras, distortion coefficients for both cameras, a rotation matrix R between the 1st and 2nd camera coordinate systems, and a translation vector T. Fortunately, `librealsense` and the ZED SDK both provide access to the internal parameters of the cameras, so we can specify `cvStereoCalibrate` to fix the intrinsic parameters and generate only R and T to a higher degree of accuracy. Otherwise, due to the high dimensionality of the many different parameters and any noise in the input data, this function can diverge from the correct solution.

Firstly, it estimates the pose of each pair of views from the 2D untransformed calibration pattern using a function named `cvFindExtrinsicCameraParams2`. This auxiliary function is an implementation of a *Perspective-n-Point* solver, which generates two pairs of extrinsic parameters R_1, T_1 and R_2, T_2 for both respective cameras in the stereo pair. Perspective-n-Point is the problem of estimating the pose of a camera given a set of correspondences between n 3D points and their 2D projections [46]. OpenCV uses an iterative method based on the *Levenberg-Marquardt optimisation*, which attempts to find a pose which minimises the re-projection error. Additionally, OpenCV also implements the P3P method [47] and the EPnP method [46] to find the camera pose, but these methods are not used here.

Using `cvFindExtrinsicCameraParams2`, `cvStereoCalibrate` estimates the extrinsics of each camera in the form of R_{i_1}, T_{i_1} and R_{i_2}, T_{i_2}. Afterwards, the relative pose between the two cameras is computed with the following equations for each pair i:

$$R_i = R_{i_2} {R_{i_1}}^{-1} \tag{4.2}$$

$$T_i = T_{i_2} - R_i T_{i_1} \tag{4.3}$$

Once this has been done for every pair of calibration patterns, the algorithm computes the median values of R_i and T_i from all n pairs. The median parameters R_M and T_M is then used as an initialisation for a *Levenberg-Marquardt optimiser*, which then attempts to find the extrinsic parameters R and T that minimises the re-projection error even further [48].

4.1.3 Our Setup

The `librealsense` library provided by Intel for the Realsense F200 helpfully provides all the internal parameters of the RGB camera, the depth camera and the infra-red sensor, where an intrinsic matrix for each camera can easily be generated. It also provides all the extrinsic parameters that map any camera to any other camera on the device, based on the factory calibrations when the device was manufactured.

Additionally, the ZED SDK provided by Stereolabs for the ZED camera provides all the internal parameters of both the left and right camera. Both of the RGB cameras on the ZED are at the exact same orientation and are only separated by a baseline distance b provided by the SDK. Because of this, it is trivial to calculate the extrinsic matrix that maps left to right:

$$E_{l,r} = \begin{bmatrix} 1 & 0 & 0 & -b \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.4}$$

The rotation is the identity matrix as both cameras have the same orientation, and the translation is of b metres along the x-axis. As extrinsic parameters map from camera-space to camera-space, the translation is relative to the right camera, where positive x extends to the right.

Now we have all the intrinsic parameters for all the cameras, and all the extrinsics that map between different views on each camera, all that is left is to be able to map between the different cameras. This is achieved by treating the Realsense F200's RGB camera and the ZEDs left eye as a stereo pair, and finding the extrinsics that map between those using OpenCV's `cvStereoCalibrate`. If we use `cvStereoCalibrate` to calculate a rotation matrix $R_{R,Z} \in \mathbb{SO}_3$ and translation vector $T_{R,Z} \in \mathbb{R}^3$, we can build an extrinsic matrix:

$$E_{R,Z} = \begin{bmatrix} R_{R,Z} & T_{R,Z} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{4.5}$$

Using this extrinsic matrix, we can then map from any camera to any other camera which is part of the prototype by going through $E_{R,Z}$ obtained from the offline calibration step. Say we want to find the extrinsics which maps the Realsense depth sensor to the right ZED sensor. We can do this by using `librealsense` to obtain the depth to RGB extrinsics, and the baseline parameter from the ZED SDK to obtain the ZED left-to-right extrinsics. Afterwards, as the extrinsic parameters are just 4x4 matrices, we can combine them together by matrix multiplication to generate a combined extrinsic mapping:

$$E_{D,r} = E_{l,r}E_{R,Z}E_{D,RGB} \tag{4.6}$$

Here, $E_{D,r}$ (depth to ZED right) is created by first mapping from Realsense depth to RGB, then Realsense RGB to ZED left (from the calibration) and finally ZED left to right. This is illustrated in Figure 4.3.

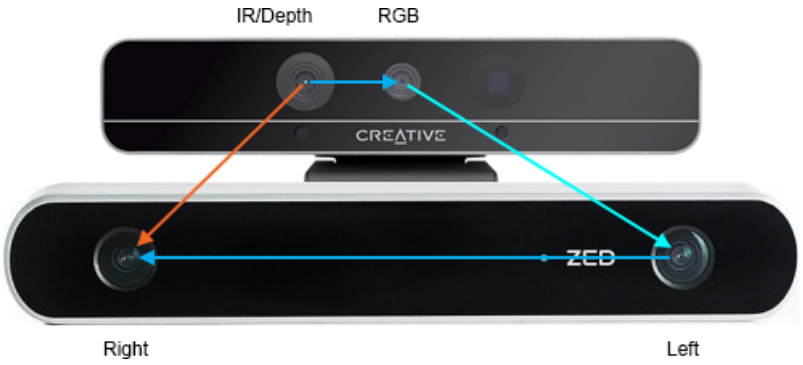

Figure 4.3: An illustration of how different camera extrinsic parameters are combined together. To map from a 3D position relative to the IR/Depth sensor to the right ZED camera (orange), we map from Depth to RGB (top blue), Realsense to ZED (cyan) and left to right (bottom blue). Blue mappings are provided to us by different libraries, and the Cyan mapping is from the stereo pair calibration step.

4.2 Rendering on the Oculus Rift

The Oculus Rift HMD contains a single 1080p panel which is split into two 960-by-1080 views, one for each eye. When using the rift, the left eye sees the left half of the screen, and the right eye sees the right half. The Oculus SDK expects the application to render the scene in such a way that there are two viewpoints which are translated by the configured interpupillary

distance (IPD). As mentioned in Section 3.1.1, the lenses create a pincushion distortion. As a result of this, the SDK applies post-processing to each view with an equivalent but opposite barrel distortion, and also corrects chromatic aberration.

The Oculus SDK documentation provides a comprehensive guide on how to render frames to the headset at [21]. The SDK makes use of a separate *compositor process* which is running on the PC alongside the application, and this process is responsible for taking frames from an application, applying post-processing and presenting it on the Rift. The application computes the FOV and allocates two framebuffer objects, and passes these to the SDK. Once it enters the rendering loop, the application captures the current pose of the headset and performs rendering for each eye. At the end of the render step, the application calls a function named `ovr_SubmitFrame` which passes the framebuffer object to the compositor process which then performs post-processing and ensures that a frame is displayed on the Rift headset ready for the next refresh cycle.

To display frames on the Oculus Rift with the correct size, we need to take the aspect ratio of the ZED camera, and the horizontal field-of-view of the ZED camera and the Oculus Rift into account. Let Z_w and Z_h be the ZED camera frame's width and height respectively, D_w and D_h be the Oculus Rift display's width and height for each eye, and F_z and F_r be the horizontal field-of-view of the ZED and the Oculus Rift. Unfortunately, the ZED SDK does not provide its field-of-view, however we can easily calculate it using some intrinsic parameters by writing $F_z = \tan^{-1} \frac{Z_w}{2f_x}$. Now, we can calculate the width of the frame O_w to be the proportion of the Rift's field-of-view which is covered by the ZED camera, multiplied by the width of the display:

$$O_w = \frac{F_z}{F_r} D_w \tag{4.7}$$

Once we have the width of the frame, we want to preserve the aspect ratio so that images captured by the ZED does not appear stretched. Therefore, multiplying the width by the inverse of the aspect ratio of the ZED gives us the height of the frame O_h:

$$O_h = O_w \frac{Z_h}{Z_w} \tag{4.8}$$

4.3 Overlay Rendering

4.3.1 Warping Depth Information

To allow objects in real life to realistically interact with virtual objects, such as a hand covering up a virtual object in front of the camera, we need to make use of the depth information captured by the Realsense camera. However, the user of the prototype is seeing through the lenses of the ZED camera which obviously has different intrinsics and extrinsics compared to the Realsense, so simply using the Realsense depth data unmodified is going to look incorrect. As we know the intrinsic and extrinsic parameters of all the cameras, we can *warp* the depth from the Realsense depth sensor's perspective to the ZED camera's perspective via re-projection.

Recover the 3D position of an image point

Let d be the depth value recorded by the Realsense depth sensor at the image point (x, y), $\mathbf{P_R} = \begin{bmatrix} X, Y, d \end{bmatrix}^T$ be the 3D position relative to the depth sensor and K_R be the intrinsic matrix of the depth sensor. To reconstruct $\mathbf{P_R}$, we refer to Section 2.1.2 which defines the image point $\mathbf{p_R}$ to be:

$$\mathbf{p_R} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} X/d \\ Y/d \\ 1 \end{bmatrix} = \frac{\mathbf{P_R}}{d} \tag{4.9}$$

By convention, the w coordinate in homogeneous coordinates is usually set to 1, so therefore $f = 1$. Unfortunately, this pinhole camera model assumes that the focal point is 1 and the principal point offset is the null vector. If we take the intrinsic parameters of the depth sensor into account, we get a new definition of $\mathbf{p_R}$:

$$\mathbf{p_R} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x X/d + x_0 \\ f_y Y/d + y_0 \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X/d \\ Y/d \\ 1 \end{bmatrix} = K_R \frac{\mathbf{P_R}}{d} \tag{4.10}$$

The depth sensor in the Realsense F200 camera has some radial lens distortion, which has been calibrated according to the inverse of a modified version of the Brown-Conrady Distortion model. This allows distorted points to be mapped to undistorted points via the

following equation [49]:

$$D_R^{-1}\left(\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}\right) = \begin{bmatrix} x(1+c_0 l + c_1 l^2 + c_4 l^3) + 2c_2 xy + c_3(l+2x^2) \\ y(1+c_0 l + c_1 l^2 + c_4 l^3) + 2c_3 xy + c_2(l+2y^2) \\ 1 \end{bmatrix}, l = x^2 + y^2 \tag{4.11}$$

Here, c_0 to c_4 are the lens distortion parameters. Taking this into account, we reach the true definition of $\mathbf{p_R}$ in 4.12. From this definition, we can recover the 3D position $\mathbf{P_R}$ in 4.13:

$$\mathbf{p_R} = K_R D_R \left(\frac{\mathbf{P_R}}{d}\right) \tag{4.12}$$

$$\mathbf{P_R} = d D_R^{-1}\left(K_R^{-1}\mathbf{p_R}\right) \tag{4.13}$$

Re-projection to the new image point

Once we have recovered the 3D position $\mathbf{P_R}$ relative to the depth sensor, we can make use of the extrinsics from Section 4.1.3 to map to a 3D position $\mathbf{P_Z}$ relative to the left or right sensor of the ZED camera. $\mathbf{P_Z}$ is given by:

$$\mathbf{P_Z} = \begin{cases} E_{R,Z} E_{D,RGB} \mathbf{P_R} & \text{for the left sensor} \\ E_{l,r} E_{R,Z} E_{D,RGB} \mathbf{P_R} & \text{for the right sensor} \end{cases} \tag{4.14}$$

Now we have a 3D position relative to the left or right sensor, we can then use an equation similar to 4.12 to project this point into the ZED sensors image plane. Let d' be the new depth value relative to the ZED camera which is the z component of $\mathbf{P_Z}$ and K_Z be the intrinsic matrix of the left and right ZED sensors. As the ZED camera has no lens distortion, the distortion function is just the identity function. We then end up with the final image point $\mathbf{p_Z}$ in homogeneous coordinates as follows:

$$\mathbf{p_Z} = K_Z \frac{\mathbf{P_Z}}{d'} \tag{4.15}$$

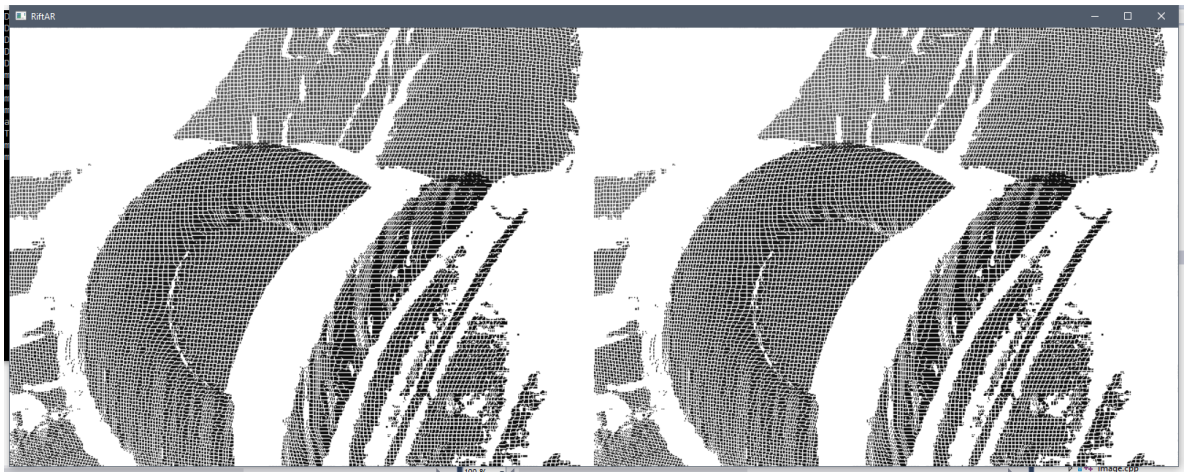

Figure 4.4: Gaps in the warped depth map due to a mismatch in source and destination resolution.

Implementation and Performance

The algorithm devised above mapping $\mathbf{p_R}$ to $\mathbf{p_Z}$ has been implemented in our prototype as a class named `RealsenseDepthAdjuster`. This class contains a method which takes a Realsense depth frame as input, and warps it to a stereo camera pair, which is given as a destination intrinsic matrix and a pair of extrinsic matrices. Due to the source and destination images having different dimensions, simply mapping each point from the source image to the destination image is going to leave many gaps as the destination image is either 720p and higher, and the source image is 640-by-480. This effect is shown in Figure 4.4. To deal with the gaps, we made use of a similar technique deployed in the `librealsense` library, which instead transforms the top-left and bottom-right corners of each image, and fills a rectangle bound by the transformed counterparts of these two points with the depth value d'. In practice, this was very effective in eliminating almost all of the gaps introduced by the warping process.

Firstly, we attempted to implement the algorithm as fast as possible using the CPU. We optimised it by simplifying the matrix multiplications taking the 0 and 1 factors into account in the intrinsic matrices, and splitting the extrinsic matrix into a rotation and translation. However, fundamentally, the algorithm still required a sequential iteration over each pixel in the source depth image. As this could clearly benefit from parallel processing, we ported the algorithm to CUDA, using a block size of 32-by-32 threads. As a result of this, we reduced the runtime of the algorithm from an average of 34.35 milliseconds on an Intel Core i5 4670K CPU to an average of 3.73 milliseconds when using CUDA on a Geforce GTX 780 Ti graphics card. This is an over 9-times improvement, allowing a potential maximum frame rate of over 250 frames per second.

Due to a combination of distortion and projection being a non-affine transformation, it is possible for two different source points to be mapped to the same destination point, such as parallax caused by a hand covering an object from the ZED but not from the Realsense. This ambiguity can be resolved by only writing the minimum depth to the destination image. However, due to the nature of CUDA being a parallel system, we have a race condition where depth values can be written from thread A after thread B has already performed the check to determine whether to overwrite the existing depth value. Fortunately, we know that the minimum function is associative i.e. $min(min(x, y), z) = min(x, min(y, z))$, therefore it does not matter what order the pixels are written to the destination image, only that every potential depth value is considered. CUDA provides *atomic* operations [50] which are read-modify-write operations on a single 32-bit or 64-bit integer that are guaranteed to be free from interference from other threads. We make use of `atomicMin` in our implementation to guarantee that the minimum pixel is always written to the destination image.

4.3.2 Loading 3D models

We were provided with a number of 3D models which complement the polystyrene head used as part of the prototype, including a Magnetic resonance imaging (MRI) scan of a brain, blood vessels, a tumour, a skull, and a smooth scan of the head of the patient where these models and the polystyrene head came from.

These models are provided in the binary **StereoLithography** (STL) file format, created by 3D Systems Inc. for 3D printing [51]. This format exists in two forms: ASCII and Binary. The ASCII format encodes all the data in string form and occupies a lot of storage space, whilst the binary format eliminates a lot of redundancy by storing the vertex data in a more compact form. A binary STL file has an 80-byte header which can be ignored, followed by a 4-byte unsigned integer which indicates the number of triangles stored in this file. All the data that follows this point are 50-byte blocks which represent each triangle. A triangle block consists of 12 32-bit floating point numbers which describe the normal vector, and the three vertices for the triangle. A block is then terminated with a so-called 2-byte *attribute byte count* which is usually 0.

4.3.3 Bridging Virtual and Reality

Bridging the gap between the virtual world and the reality captured by the different sensors requires having to think about both how to get depth information from the depth sensor into the virtual world, and how to correctly project virtual objects so that they appear correctly

alongside real world objects. Our approach consists of deriving an OpenGL projection matrix from the intrinsic parameters of the ZED camera which is used by virtual objects, and rendering the warped depth stream to a full screen quad which writes depth information to the depth buffer.

Projection Matrix

Kyle Simek from *"Sightations - A Computer Vision Blog"* posted a technique about how to derive an OpenGL projection matrix from an intrinsic matrix [52], which we implemented in our prototype. To understand how this works we need to understand how perspective projection works in OpenGL. A perspective projection can be represented by the following matrix:

$$P = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{4.16}$$

Here, it takes a frustum bounded by 6 parameters (left, top, right, bottom, near, far) as illustrated in Figure 4.5, and transforms points within this frustum to *normalised device coordinates* (NDC). This places points in a box that ranges between $[1, 1, 1]$ and $[-1, -1, -1]$ which can then be easily scaled to the viewport size by the graphics driver. This also preserves depth unlike the intrinsic matrix so that data can be then written to the depth buffer.

This transformation performed by P can be broken down into two different transformations: transform the frustum-shaped space into a cuboid-shaped shape, then transform the cuboid-shaped space into NDC space. This can be written as:

$$P = M_{\text{NDC}} \times M_{\text{perspective}} \tag{4.17}$$

Let K be a 3-by-3 intrinsic matrix from Section 2.1.3. To use our intrinsic matrix in OpenGL, we first need to make some modifications. Firstly, as OpenGL looks down the *negative* z-axis, we need to invert the 3$^{\text{rd}}$ column of the intrinsic matrix so that $K_{3,3}$ is -1. Secondly, we need to insert another row and column to preserve depth information. After both of these changes, we end up with the following perspective matrix:

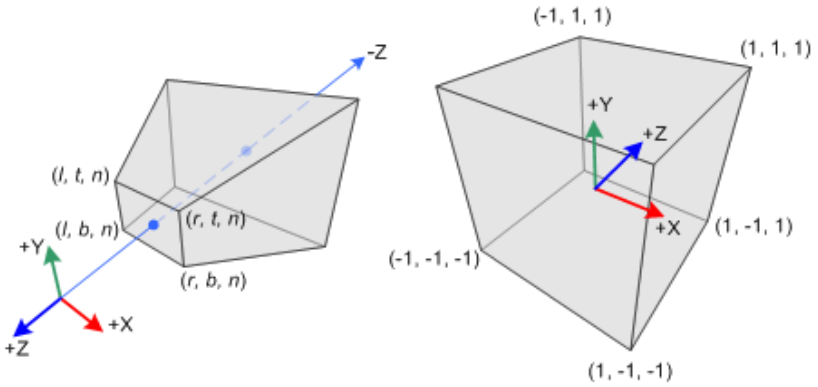

Figure 4.5: An illustration of how the perspective projection maps from a frustum shape to normalised device coordinates. Adapted from "OpenGL Projection Matrix" - URL: `http://www.songho.ca/opengl/gl_projectionmatrix.html`

$$M_{\text{perspective}} = \begin{bmatrix} f_x & s & -x_0 & 0 \\ 0 & f_y & -y_0 & 0 \\ 0 & 0 & n+f & nf \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{4.18}$$

In this matrix, f_x, f_y, x_0, y_0 and s are obtained from the intrinsic matrix, and n and f define the near and far planes respectively. The next step is to generate the NDC matrix, which is surprisingly identical to an *orthographic projection*:

$$M_{\text{NDC}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.19}$$

The choice of the left, top, right and bottom clipping planes depend on the dimensions of the image and the coordinate system used in calibration. In our system, the origin is defined to be the bottom-left of the image, therefore $l = 0$, $r =$ width, $b = 0$ and $t =$ height.

We can show that there is an equivalence between the product of the NDC and perspective matrix and the projection matrix given in 4.16 when f_x and f_y is set to n and s, x_0 and y_0 is set to 0:

$$
\begin{aligned}
P &= M_{\text{NDC}} \times M_{\text{perspective}} \\
&= \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & nf \\ 0 & 0 & -1 & 0 \end{bmatrix} \\
&= \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}
\end{aligned}
\tag{4.20}
$$

To relax the constraints given above, we can scale the left and right clipping planes by n/f_x and the top and bottom clipping planes by n/f_y. Principal point offsets can also be factored in by shifting the horizontal clipping planes by $-x_0$ and the vertical clipping planes by $-y_0$.

Handling Occlusion

To allow objects in the real world to interact in a convincing way with objects in the virtual world, real objects need to both occlude and be occluded by opaque virtual objects. We do this in our prototype by taking the depth values from the warped depth stream and project them so that they lie in normalised device coordinates. If we take some vector where we don't care about the x and y coordinates, and the z coordinate is the depth value d of the warped depth stream, we can derive a projected z coordinate from a general projection matrix as follows:

$$
\begin{bmatrix} \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ \cdots & \cdots & -1 & 0 \end{bmatrix} \begin{bmatrix} \cdots \\ \cdots \\ -d \\ 1 \end{bmatrix} = \begin{bmatrix} \cdots \\ \cdots \\ d\frac{f+n}{f-n} - \frac{2fn}{f-n} \\ d \end{bmatrix}
\tag{4.21}
$$

Note that the camera looks down the negative z axis, hence why the depth value is negative. Given this homogeneous coordinate, we can then perform perspective division to give an equation which maps the depth value in metres to a depth coordinate:

$$z = \frac{f+n}{f-n} - \frac{1}{d}\frac{2fn}{f-n} \tag{4.22}$$

This equation is implemented in the fragment shader for the 2D rectangle which displays the ZED colour stream mentioned in Section 4.2. As the depth calculation is embedded in the fragment shader, depth testing is not done until after the 2D rectangle has been rendered. This ends up having the effect where objects with greater depth values than what is written by fragment shader are "covered up" by the colour stream, and other objects with smaller depth values stay visible.

Dealing with rendering overlays on top of objects is trickier, as we need to consider the fact that some parts of the scene will be occluding the overlay (such as a hand in front of a head with a projection), but some parts of the scene will not be (such as a brain scan appearing "inside" a head). Our method of dealing with this involves using the **Stencil Buffer**. The Stencil Buffer is an additional buffer that stores integer values which is used in conjunction with the colour buffer and depth buffer to allow additional control over which pixels are rendered.

First, we render the camera colour and depth to the rectangle as normal, as well as any virtual objects which are not overlays. If we have a transform $T \in \mathbb{SE}_3$ which represents the location of a 3D model in the real world with vertices $v = \{v_1, v_2, \ldots, v_n\}$ relative to some fixed coordinate system, we then render an "expanded" version of the model with transform T, only writing a 1 to the stencil buffer if this model would be visible, and writing nothing to the colour and depth buffers. We modify the transformation matrix T to "expand" the model by first transforming the mesh by a scale matrix:

$$T_e = T \begin{bmatrix} \lambda & 0 & 0 & 0 \\ 0 & \lambda & 0 & 0 \\ 0 & 0 & \lambda & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.23}$$

Scale factor λ should be decided in such a way that compensates for any degree of error in the transform T. This should inevitably enclose the entire physical head even if T is slightly off from the true transform of the physical head. Once we have the values in the stencil buffer, we can render any overlays by clearing the depth buffer and using a stencil value of 0 as a mask. As this operation is destructive to the depth buffer, this step happens after all virtual objects have been rendered already.

4.4 Localisation and Object Alignment

To implement marker-free augmented reality, we are required to have a precise understanding of the geometry of the scene to accurately interact with it. Our prototype makes use of the **KinectFusion** algorithm discussed in Section 2.3.2 to both understand where the prototype is in the scene, and incrementally build a 3D model of the scene. It makes use of an open source implementation of KinectFusion called **KFusion**, which is implemented almost entirely on the GPU with CUDA. Using the signed distance volume for the scene which is build by KFusion, we created a cost function which converges to 0 when a set of transformed vertices of a virtual object perfectly matches a physical counterpart in the scene. This cost function takes a set of vertices and a transform T_a which is then optimised using the Downhill Simplex method to generate a new transform T_b that most accurately matches the true transform of the physical object with respect to a fixed coordinate system.

4.4.1 Converting between Coordinate Systems

Unfortunately, the KFusion library makes use of a coordinate system which is different from the OpenGL coordinate system that we use. This difference is illustrated in Figure 4.6. Let the OpenGL coordinate system's basis vectors be $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ respectively, and the KFusion coordinate system's basis vectors be $\mathbf{x}'$, $\mathbf{y}'$ and $\mathbf{z}'$ respectively. From observations, we determined that $\mathbf{x}' = \mathbf{x}$, $\mathbf{y}' = -\mathbf{y}$ and $\mathbf{z}' = -\mathbf{z}$, meaning that the coordinates are flipped along the y-axis and the z-axis. Additionally, the origin of the OpenGL coordinate system is at the bottom-left front corner of the signed distance volume, but the origin of the KFusion coordinate system is at the top-left front corner, a translation along OpenGL's y-axis by a factor s, where s is the height of the volume.

To convert between coordinate system A and coordinate system B (where A and B can be either OpenGL or KFusion, as our method is bi-directional), we need to deal with the rotation of the object and the translation of the object separately. Let T_A be the transformation matrix of an object in coordinate system A, as $T_A \in \mathbb{SE}_3$, we can separate it into a rotation followed by a translation:

$$T_A = \begin{bmatrix} R_A & \mathbf{t}_A \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} I & \mathbf{t}_A \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} R_A & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{4.24}$$

Converting the translation from A to B is trivial, we simply inverse the direction on the y-axis and z-axis, and add the height of the signed distance volume to the y-axis:

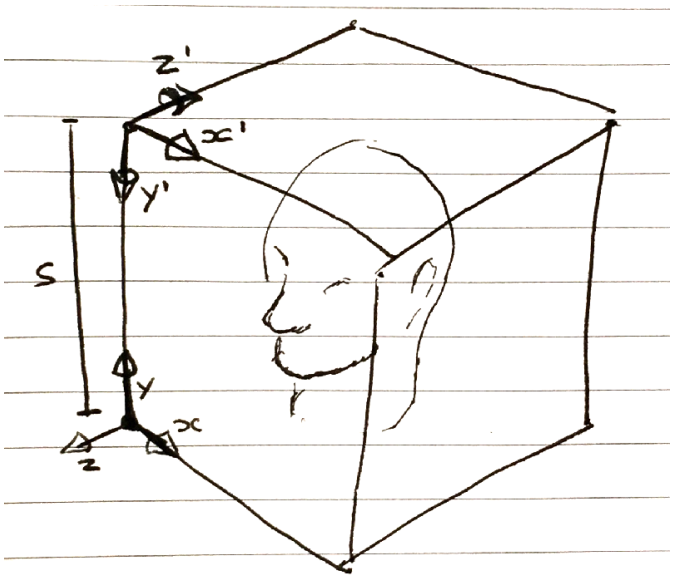

Figure 4.6: An illustration of the OpenGL (bottom) and KFusion (top) coordinate systems.

$$\mathbf{t}_B = \begin{bmatrix} \mathbf{t}_{Ax} \\ s - \mathbf{t}_{Ay} \\ -\mathbf{t}_{Az} \end{bmatrix} \tag{4.25}$$

Mirroring a Rotation

Mirroring a rotation along an axis is not as simple as multiplying the 3-by-3 rotation matrix by a scale matrix that inverts the y-axis and z-axis. Instead, we need to think of the rotation in *angle-axis* form and reflect it along the y-axis and z-axis. Euler's rotation theorem states that any rotation about a fixed point is equivalent to a single rotation of angle θ about an axis that runs through the fixed point.

Quaternions form a number system which extend the complex numbers that provide a convenient mathematical notation for representing a rotation in 3-dimensions in angle-axis form [53]. Mathematically, a Quaternion is a number in the form $\mathbf{q} = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$, with a single real component and three imaginary components. If we let θ be a rotation about an axis defined by the unit vector $\mathbf{u} = [x, y, z]^T = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$, then using an extension of Euler's formula, we can represent this rotation as a quaternion:

$$\mathbf{q} = e^{\frac{\theta}{2}(x\mathbf{i}+y\mathbf{j}+z\mathbf{k})} = \cos\frac{\theta}{2} + (x\mathbf{i} + y\mathbf{j} + z\mathbf{k})\sin\frac{\theta}{2} \tag{4.26}$$

When a rotation is in Quaternion form, it is then possible to mirror this rotation about one or more axis' [54]. If we define a plane through the origin using a normal vector $\mathbf{n} = [n_x, n_y, n_z]^T$, a reflection of the vector $\mathbf{v} = [v_x, v_y, v_z]^T$ is defined to be:

$$\mathbf{v}_{\text{refl}} = \mathbf{v} - 2\mathbf{n}\frac{\mathbf{v} \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{n}} \tag{4.27}$$

In our case, we need to reflect the vector in the XY plane, then in the XZ plane. As a result, we have two planes with normal vectors $\mathbf{n}_{xy} = [0, 0, 1]^T$ and $\mathbf{n}_{xz} = [0, 1, 0]^T$, and we reflect $\mathbf{v}_A$ in the XY plane to get an intermediate vector $\mathbf{v}_{\text{int}}$, then reflect $\mathbf{v}_{\text{int}}$ in the XZ plane to get the final vector $\mathbf{v}_B$:

$$\begin{aligned} \mathbf{v}_{\text{int}} &= \mathbf{v}_A - 2(\mathbf{v} \cdot \mathbf{n}_{xy})\mathbf{n}_{xy} \\ &= \begin{bmatrix} v_{Ax} \\ v_{Ay} \\ v_{Az} \end{bmatrix} - 2v_{Az} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} v_{Ax} \\ v_{Ay} \\ -v_{Az} \end{bmatrix} \end{aligned} \tag{4.28}$$

$$\begin{aligned} \mathbf{v}_B &= \mathbf{v}_{\text{int}} - 2(\mathbf{v}_{\text{int}} \cdot \mathbf{n}_{xz})\mathbf{n}_{xz} \\ &= \begin{bmatrix} v_{Ax} \\ v_{Ay} \\ -v_{Az} \end{bmatrix} - 2v_{Ay} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} v_{Ax} \\ -v_{Ay} \\ -v_{Az} \end{bmatrix} \end{aligned} \tag{4.29}$$

Going back to our quaternion $\mathbf{q}$, if we reflect the imaginary vector in the XY plane, followed by the XZ plane, we end up with a new quaternion $\mathbf{q}'$ which represents the mirrored rotation that we're looking for:

$$\mathbf{q}' = \cos\frac{\theta}{2} + (x\mathbf{i} - y\mathbf{j} - z\mathbf{k})\sin\frac{\theta}{2} \tag{4.30}$$

Unfortunately, our rotations are stored as 3-by-3 matrices in $\mathbb{SO}_3$ and converting it into a quaternion and back is quite involved and inefficient. Fortunately, it is possible to operate on the matrix directly. If we generate two rotation matrices R_A and R_B from the quaternions $\mathbf{q}$ and $\mathbf{q}'$, we can find a simple mapping from R_a to R_b:

$$\begin{aligned}
R_A &= \begin{bmatrix} a^2+b^2-c^2-d^2 & 2bc-2ad & 2bd+2ac \\ 2bc+2ad & a^2-b^2+c^2-d^2 & 2cd-2ab \\ 2bd-2ac & 2cd+2ab & a^2-b^2-c^2+d^2 \end{bmatrix} \\
R_B &= \begin{bmatrix} a^2+b^2-c^2-d^2 & -2bc+2ad & -2bd-2ac \\ -2bc-2ad & a^2-b^2+c^2-d^2 & 2cd-2ab \\ -2bd+2ac & 2cd+2ab & a^2-b^2-c^2+d^2 \end{bmatrix} \\
&= \begin{bmatrix} a^2+b^2-c^2-d^2 & -(2bc-2ad) & -(2bd+2ac) \\ -(2bc+2ad) & a^2-b^2+c^2-d^2 & 2cd-2ab \\ -(2bd-2ac) & 2cd+2ab & a^2-b^2-c^2+d^2 \end{bmatrix} \\
&= \begin{bmatrix} R_{A1,1} & -R_{A1,2} & -R_{A1,3} \\ -R_{A2,1} & R_{A2,2} & R_{A2,3} \\ -R_{A3,1} & R_{A3,2} & R_{A3,3} \end{bmatrix}
\end{aligned} \tag{4.31}$$

To simplify the matrices above, we have used the following substitutions: $a = \cos\frac{\theta}{2}$, $b = x\sin\frac{\theta}{2}$, $c = y\sin\frac{\theta}{2}$, $d = z\sin\frac{\theta}{2}$. Much like how $\mathbf{q}$ can be reflected to $\mathbf{q}'$ by reversing the $\mathbf{j}$ and $\mathbf{k}$ components of the quaternion, we can generate a reflected rotation matrix R_B by taking R_A and inverting elements $R_{B1,2}$, $R_{B1,3}$, $R_{B2,1}$ and $R_{B3,1}$.

Bringing it all together

To construct an equivalent transformation matrix T_B in coordinate system B, we reverse the decomposition of T_B using the reflected translation vector $\mathbf{t}_B$ and reflected rotation matrix R_B:

$$T_B = \begin{bmatrix} I & \mathbf{t}_B \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} R_B & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} R_B & \mathbf{t}_B \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{4.32}$$

From this, we can write the following matrix function that maps from coordinate system A to coordinate system B:

$$\text{convAB}\left(\begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & t_x \\ r_{2,1} & r_{2,2} & r_{2,3} & t_y \\ r_{3,1} & r_{3,2} & r_{3,3} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}\right) = \begin{bmatrix} r_{1,1} & -r_{1,2} & -r_{1,3} & t_x \\ -r_{2,1} & r_{2,2} & r_{2,3} & s - t_y \\ -r_{3,1} & r_{3,2} & r_{3,3} & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.33}$$

4.4.2 Determining the transform of a model

Now that we have a method to convert points from our coordinate system to KFusion, we can determine the position of a 3D model counterpart as seen by the depth camera and constructed by the KinectFusion algorithm. If we have a model consisting of a set of n vertices $v = \{v_1, v_2, \ldots, v_n\}$. KinectFusion provides us with a 3D representation of the scene in the form of a truncated signed distance function (TSDF) [12], which we will denote as $V : \mathbb{R}^3 \mapsto \mathbb{R}$. If we transform each vertex in the model v by some transformation T, we can write a cost function as follows which has a global minimum when the transform T corresponds exactly to a transform T_m, which is the transform of the models counterpart in a perfect scene reconstruction:

$$\text{cost}(T) = \frac{1}{n}\sum_{i=1}^{n} \min(|V(Tv_i)|, \lambda) \tag{4.34}$$

The cost function can be seen calculating as the average distance that each point is from a surface, i.e. zero-crossings in the TSDF volume. Unfortunately, as the reconstruction from KinectFusion tends to be somewhat inaccurate with missing information such as the area behind the physical counterpart of the model, we need to introduce a value λ to truncate the cost function to correctly deal with outliers.

Our prototype employs an assisted manual method in an attempt to find T_m. Firstly, the user activates "search mode" which places a model (currently, a head model) in front of the camera with transform T_c. Let $\mathbf{s} = [s_x, s_y, s_z]^T$ be the size of the model, and T_p be the transformation matrix for the current pose of the camera, obtained from KFusion. If we place the model in front of the camera at a distance which is twice the depth of the object, we can calculate T_c as follows:

$$T_c = T_p \begin{bmatrix} 1 & 0 & 0 & -\frac{s_x}{2} \\ 0 & 1 & 0 & -\frac{s_y}{2} \\ 0 & 0 & 1 & -\frac{5s_z}{2} \end{bmatrix} \tag{4.35}$$

The prototype then continuously evaluates the cost function against T_c until it reaches some threshold μ which is set to 200mm in the prototype. We need to be careful with coordinate systems, as the cost function expects the transform provided to be in the KFusion coordinate system, whilst our camera pose T_p is in the OpenGL coordinate system. Once we reach the threshold μ, finding the optimal translation T_m becomes an optimisation problem:

$$T_m = \arg\min_{T}(\text{cost}(\text{convAB}(T))) \tag{4.36}$$

There are a number of different methods we can employ to find T_m, however the best for our purposes is the Nelder-Mead method, also known as Downhill Simplex [55]. This method uses the concept of a *simplex*, which is a generalisation of a triangle or tetrahedron to an arbitrary dimension n, which contains $n+1$ vertices, and takes a series of steps to find the minimum of an objective function with n parameters. This method is sufficient for our purposes compared to other methods such as Newtons, which rely on the derivative of the objective function being known.

A transformation matrix $T \in \mathbb{SE}_3$ has 6 degrees of freedom, so it can be parametrised as 3 values for translation, and 3 values for rotation in the form of *Euler Angles*. We make use of the XYZ order of Tait-Bryan angles, however this is chosen rather arbitrarily as eleven other conventions of Euler angles exist. T_c is converted into a vector of 6 parameters which is used to initialise already existing in-house implementation of Downhill Simplex employed in our prototype. Afterwards, the Downhill Simplex optimiser repeatedly calls an objective function which simply unpacks the vector of parameters into a transformation matrix which is then passed into our cost function.

To help with conversion to and from the vector of 6 parameters, we will decompose our transformation matrix into a rotation and translation:

$$T = \begin{bmatrix} R & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{4.37}$$

Converting from a 6D vector to a Matrix

Let $\mathbf{P} = [x, y, z, \psi, \theta, \phi]$ be the parameters obtained from the 6D parameter vector. From this, we can write the translation vector as $\mathbf{t} = [x, y, z]$. Given a set of three rotations about each principal axis, where ψ is a rotation about the x-axis, θ is a rotation about the y-axis and ϕ is a rotation about the z-axis, we can then build a rotation matrix from this sequence of

rotations:

$$
\begin{aligned}
R &= R_z(\phi)R_y(\theta)R_x(\psi) \\
&= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi \\ 0 & \sin\psi & \cos\psi \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} \cos\phi & -\sin\phi & \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \cos\theta\cos\phi & \sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi & \cos\psi\sin\theta\cos\phi + \sin\psi\sin\phi \\ \cos\theta\sin\phi & \sin\psi\sin\theta\sin\phi + \cos\psi\cos\phi & \cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi \\ -\sin\theta & \sin\psi\cos\theta & \cos\psi\cos\theta \end{bmatrix}
\end{aligned} \tag{4.38}
$$

Converting from a Matrix to a 6D vector

Let T be a transformation matrix with a rotation matrix R in the upper-left corner and a translation vector $\mathbf{t}$ in the upper-right corner, as in 4.37. We can then write the parameter vector $\mathbf{P} = [t_x, t_y, t_z, \psi, \theta, \phi]$. To extract the three Tait-Bryan angles ψ, θ and ϕ from the rotation matrix R, we need to examine the 9 equations from 4.38 using a method described in [56]. Firstly, we can extract θ from $R_{3,1}$ as follows:

$$
\begin{aligned}
\theta_1 &= -\sin^{-1}(R_{3,1}) \\
\theta_2 &= \pi - \theta_1 = \pi + \sin^{-1}(R_{3,1})
\end{aligned}
$$

We need to be careful here, because as $\sin\theta = \sin(\pi - \theta)$, there are in fact *two* values of θ which satisfy this equation. Next, we observe that $R_{3,2}/R_{3,3} = \tan\psi$. To calculate the arc tangent of $R_{3,2}/R_{3,3}$, we make use of a function named $\text{atan2}(y, x)$ which takes the signs of its parameters into account to determine the correct quadrant of the result. We need to be careful, because if $\cos\theta > 0$, then $\psi = \text{atan2}(R_{3,2}, R_{3,3})$, but if $\cos\theta < 0$, then $\psi = \text{atan2}(-R_{3,2}, -R_{3,3})$. Taking this into account, we end up with the following values for ψ:

$$
\begin{aligned}
\psi_1 &= \text{atan2}\left(\frac{R_{3,2}}{\cos\theta_1}, \frac{R_{3,3}}{\cos\theta_1}\right) \\
\psi_2 &= \text{atan2}\left(\frac{R_{3,2}}{\cos\theta_2}, \frac{R_{3,3}}{\cos\theta_2}\right)
\end{aligned}
$$

In a similar process to finding the value of ψ, we observe that $R_{2,1}/R_{1,1} = \tan\phi$, and from this we can obtain the following values for ϕ:

$$\phi_1 = \text{atan2}\left(\frac{R_{2,1}}{\cos\theta_1}, \frac{R_{1,1}}{\cos\theta_1}\right)$$
$$\phi_2 = \text{atan2}\left(\frac{R_{2,1}}{\cos\theta_2}, \frac{R_{1,1}}{\cos\theta_2}\right)$$

Now, in the case of $\cos\theta \neq 0$, we have two solutions which produce the same rotation matrix: $r_1 = (\psi_1, \theta_1, \phi_1)$ and $r_2 = (\psi_2, \theta_2, \phi_2)$. Ultimately, it does not matter which solution is chosen, so to ensure we have sensible angles we choose the rotation with the shortest manhattan distance.

Unfortunately, the technique above does not work in the degenerate case where $\cos\theta = 0$, which corresponds to $\theta = \pi/2$ or $\theta = -\pi/2$. Examining $R_{1,2}$ and $R_{1,3}$ in both cases gives us an infinite amount of solutions in the form $R_{1,2}/R_{1,3} = \tan(\psi - \phi)$. If we set $\phi = 0$, then we end up with the following cases:

if $\theta = \pi/2$	$(0, \pi/2, \text{atan2}\,(R_{1,2}, R_{1,3}))$
if $\theta = -\pi/2$	$(0, -\pi/2, \text{atan2}\,(-R_{1,2}, -R_{1,3}))$

4.5 Image Stabilisation

When integrating a live video stream into the Oculus Rift, especially from a head mounted camera, we need to be aware of a number of caveats. Head mounted displays are usually very sensitive to latency, and if the application cannot render at a frequency which matches the refresh rate of the display, then the user is going to perceive unnatural lag and juddering in their vision. To avoid any unpleasant side effects, we need to take some extra steps to keep the refresh-rate high and stabilise the image.

4.5.1 Threaded Frame Capture

Our first problem is the frame rate of the camera. The Realsense F200 camera operates at 60 frames-per-second, and ZED camera operates at either 60, 30 or 15 frames-per-second

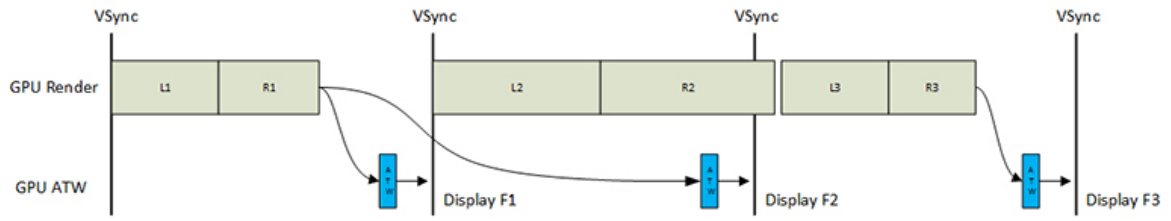

Figure 4.7: Asynchronous Timewarp. Adapted from "Asynchronous Timewarp on Oculus Rift" [58]

depending on what resolution we select. If we decide to capture frames from both cameras in the same thread as where we render the frames to the Oculus Rift, it is going to be limited by the minimum frame rate of the two cameras, which will certainly be less than the 75Hz target for the DK2. Therefore, capturing frames from both cameras needs to be asynchronous with the rendering.

As suggested in "Oculus Rift In Action" [57], our prototype achieves this by initialising both cameras and subsequently starts a new so-called *capture thread*. This contains an infinite loop which captures a frame from both cameras, and copies the frame data from both cameras into some intermediate storage. The main thread next acts as a *consumer* by attempting to grab a frame from the intermediate storage, then performs further processing before being displayed on the Rift.

4.5.2 Asynchronous Timewarp

Developers of VR applications need to take the amount of time spent rendering into consideration, as a compelling experience relies on the display having a very precise frame rate. In order to produce a perceptually correct representation of the virtual world, the displays must be updated at each refresh cycle. Unfortunately, complex real-world graphics applications usually have a varying workload frame-to-frame, such as when a player enters a zone with a high polygon count, or there are lots of special effects being rendered. Additionally, the operating system will frequently schedule other processes on the system which means that it is sometimes difficult to prevent any kind of juddering.

Timewarp is a technique that warps the rendered image before it is sent to the display, in order to correct for any subsequent head motion after the frame was rendered in an attempt to reduce the perceived latency. The idea behind Timewarp is to capture the pose of the headset *before* the frame is rendered, and again before it is displayed. Once we're ready to submit the frame, we can then shift the frame so that objects are positioned on the display where the user expects them to be, accounting for any movement since they were rendered.

This is also useful to handle cases where the frame is not going to be rendered in time for the display refresh cycle, so instead the previously completed frame can be taken and timewarped, as illustrated in Figure 4.7.

Fortunately, the Oculus SDK provides an implementation of this technique which we have made use of in our prototype, called **Asynchronous Timewarp (ATW)** [58]. Synchronous timewarp has existed in the Oculus SDK since the second Developer Kit (DK2) was released, and this was effective at reducing the perceived latency. However, this technique was still sensitive to whether the application could submit a frame at a reliable frame rate that matched the refresh rate of the display. To deal with a varying workload and consistently warp the most recently submitted frame before a display refresh, the rendering loop and the timewarp must be decoupled from each other. Asynchronous Timewarp works by employing CPU and GPU preemption to interrupt the rendering of a frame and allow the Oculus SDK to warp the previously submitted frame to the new orientation of the headset and display it.

As the video streams from the cameras attached to our prototype operate at a frame rate which is lower than the refresh rate of the headset's display, we capture the pose of the headset at the point of obtaining a new frame, and relying on the SDK's ATW feature to warp this image until a new frame is received.

4.6 Stereo Vision

Stereo Vision introduces new challenges to ensuring a comfortable experience for the wearer. We make use of the Horizontal Image Translation technique from Section 2.4.1 to reduce the disparity in the left and right eye of some object of interest visible to the wearer. This is done to prevent their eyes from converging or diverging when an object is held in front of the ZED stereo camera in an attempt to work around the camera pair's fixed convergence angle.

4.6.1 Deciding the point of interest

Deciding a meaningful focal point is a tricky problem. One method is to simply use the middlemost depth value as the focal point f. However, this is very unstable, especially on object bounderies. For example, if the wearer places their hand in front of the sensor, the space between the fingers can cause the focal point to jump between the hand and the wall behind the hand. As a result, a more sophisticated approach is required.

If we take a small 32-by-32 region in the centre of the depth image, we could compute

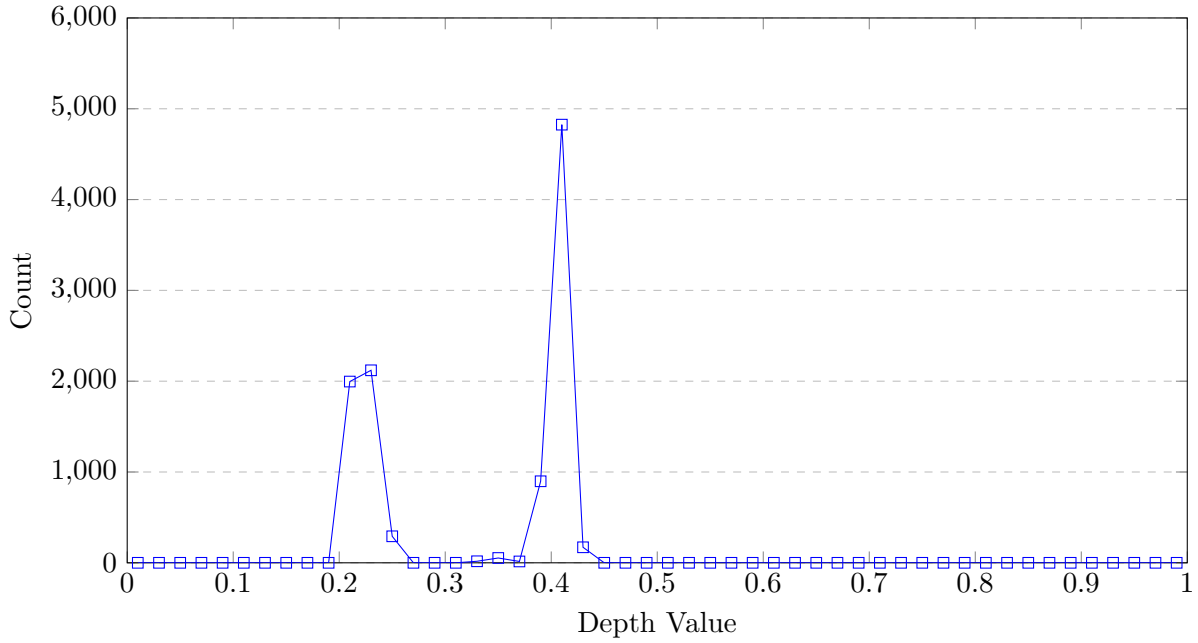

Figure 4.8: A histogram of depth values where the polystyrene head (0.2-0.3) is placed in front of a wall (0.38-0.45). Depth values are scaled to be in the range 0-1 by dividing the depth in metres by the range of the depth sensor.

the average depth in an effort to smooth the effect. Nonetheless, there is a case where this produces an incorrect result. If we have two objects which take up exactly the same amount of area in the region but are placed far apart, such as a hand and the wall behind it, then the average depth value will be placed incorrectly between both objects.

From the given region, let's assume we have a histogram of all the depth values, arranged into reasonably small class widths. We can then use **histogram based segmentation** in an effort to identify "points of interest" in this region. For example, if we have a polystyrene head in front of a wall, this can be represented as two curves in the histogram as shown in Figure 4.8. We can interpret these curves as being distinct objects visible in the 32-by-32 region. We can say that the area underneath each curve defines how many depth pixels that the corresponding object consists of.

If we define the object we want to focus on as the one that takes up the most space in the 32-by-32 region, we can choose the curve with the largest area and take the average of all the depth values in this curve. Using this heuristic as the focal point for the auto-focus algorithm gave convincing and relatively stable results.

4.6.2 Horizontal Image Translation

In Section 2.4.1, we introduced the **Horizontal Image Translation** technique in an attempt to reduce the disparity of a target object. Our situation is slightly different as instead of having a screen surface, we have a fixed convergence angle. If we let θ be the convergence angle and e be the inter-ocular distance, we can work out the distance to the convergence point s as:

$$s = \frac{e}{2\tan\frac{\theta}{2}}$$

We know from Section 2.4.1 that objects closer than s have a negative disparity, object at s have zero disparity, and objects beyond s have a positive disparity. Using equation 2.12, we can calculate the disparity d based on the distance z of an object:

$$d = e\left(1 - \frac{s}{z}\right)$$

In this form, the disparity calculated from equation 2.12 is the distance between the image points on the screen surface in metres. To be able to accurately translate the left and right images inside the head-mounted display correctly, we need to calculate the disparity of the projected points in pixels. Let K be the calibration matrix of the camera visible to the wearer, and $\mathbf{P}_L = [x_L, y, s]$ and $\mathbf{P}_R = [x_R, y, s]$ be the position of the image points in Cartesian coordinates. We can first project each point into homogeneous coordinates $\mathbf{h}_L$ and $\mathbf{h}_R$ respectively as follows:

$$\begin{aligned}\mathbf{h}_L &= K\mathbf{P}_L \\ &= \begin{bmatrix} f_x & 0 & x_0 \\ 0 & f_y & x_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_L \\ y \\ s \end{bmatrix} \\ &= \begin{bmatrix} f_x x_L + x_0 \\ f_y y + x_1 \\ s \end{bmatrix}\end{aligned} \qquad \begin{aligned}\mathbf{h}_R &= K\mathbf{P}_R \\ &= \begin{bmatrix} f_x & 0 & x_0 \\ 0 & f_y & x_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_R \\ y \\ s \end{bmatrix} \\ &= \begin{bmatrix} f_x x_R + x_0 \\ f_y y + x_1 \\ s \end{bmatrix}\end{aligned}$$

Once this is done, we can perform perspective division to obtain the on-screen cartesian coordinates $\mathbf{p}_L$ and $\mathbf{p}_R$:

$$\mathbf{p}_L = \begin{bmatrix} (f_x x_L + x_0)/s \\ (f_y y + x_1)/s \end{bmatrix} \qquad \mathbf{p}_R = \begin{bmatrix} (f_x x_R + x_0)/s \\ (f_y y + x_1)/s \end{bmatrix}$$

Subtracting the x coordinate of $\mathbf{p}_L$ from $\mathbf{p}_R$ obtains the following equation to calculate the disparity in pixels:

$$d' = \frac{f_x}{s}(x_R - x_L) = \frac{f_x}{s}d \tag{4.39}$$

5 Evaluation

5.1 Performance

5.1.1 Frame Breakdown

Here, we analyse the performance of the prototype by breaking down a single frame into its constituent stages. Currently, the frame rate varies between 40Hz and 60Hz which is mostly affected by the time KFusion takes to perform tracking and integration.

Stage	**Time**
Grab Frame	2.0021ms
KFusion	7.005ms
Search for Model	1.001ms
Warp Depth	4.0025ms
Calculate Focal Point	0ms
Render to Rift	5.9853ms
Total	19.9959ms

Table 5.1: A breakdown of a single frame when the most amount of work is being done: when searching for the location of the polystyrene head.

Grabbing a frame

Capturing all the frame data from both cameras is managed in a separate thread, where the rate of capture is limited by the camera with the lowest frame rate. However, in the main thread, we attempt to copy over the most recently captured frame if we haven't done so already. This averages out to about **2ms** per frame due to having to copy $1280 \times 720 \times 4 + 640 \times 480 \times 2 = 4.3$MB of data, and potentially wait on a mutex lock if the other thread is busy capturing new frame information.

KFusion Tracking and Integration

The KFusion Tracking and Integration stage performs all of the work for the KinectFusion algorithm. It creates a copy of the depth image into device-mapped memory, which is memory managed by CUDA where memory write operations on the host get mirrored on the device. The depth image additionally gets converted into mm by sequentially multiplying every value in the image by a scale factor. This process take a small but constant amount of time.

The remaining processing is the work undertaken by the KFusion library. First of all, the algorithm attempts to perform tracking by building a voxel cloud from the depth image, then using Iterative Closest Points from Section 2.3.3 to fit onto the previous voxel cloud. If this process is successful, it performs an integration and raycast step. This stage tends to vary depending on whether tracking was successful or not, ranging from **4ms** to **7ms**.

Searching for the model

When the user engages 'search mode', the prototype executes the cost function detailed in Section 4.4.2. The cost function makes use of the signed distance volume stored on the GPU from KFusion, by calculating the truncated signed distance value for 10% of the vertices in parallel using CUDA, then returning the sum of these values to find the cost. For a model with 50,000 vertices, this process takes approximately **1ms**.

Warping the Depth Stream

Warping the depth is a relatively expensive operation, as each depth pixel from the Realsense needs to be transformed by a 4-by-4 matrix four times in total (twice for each eye), then written to the destination depth image which matches the size of the ZED camera image. Our first implementation was done completely on the CPU, and took an astonishing **35ms** on average. We then reimplemented this technique using CUDA to process each pixel from the source image in parallel, and this reduced the runtime to approximately **4ms**.

Calculating the Focal Point for HIT

Calculating the focal point from the depth stream is a very inexpensive operation compared to the other stages. 1024 pixels from the depth image is organised into a histogram which then has a single processing pass to determine the depth range of an object of interest. Afterwards,

the average depth from the subset of the 1024 pixels becomes the focal point. This takes a negligible amount of time, as the highest precision clock available in Windows reports that this stage takes **0ms**.

Rendering to the Rift

Rendering the overlays and displaying them on the Rift varies by the largest margin, ranging from taking only **0.5ms** up to **10.5ms**. Upon further analysis, we determined that rendering overlays takes a negligible amount of time which cannot be measured. This is likely because the GPU is rendering a maximum of approximately 40,000 triangles, a far cry from the maximum rendering rate of approximately 5 billion triangles per second using a NVidia GeForce GTX 780 Ti [59].

However, submitting a frame to the Oculus Rift headset proved to be the cause of the extra time taken by this step. The Oculus SDK documentation states this effect: "`ovr_SubmitFrame` triggers distortion and processing which might happen asynchronously. The function will return when there is room in the submission queue and surfaces are available. Distortion might or might not have completed." [60].

5.1.2 Latency

It is essential that the perceived latency by the wearer is minimised as much as possible, to prevent any unpleasant side effects. Unfortunately, without decreasing the transmission time of a frame and any processing required to almost zero, there will always be some degree of lag between what is perceived by the wearer and reality.

If we are aware of precisely the amount of latency between capture and display, we can use prediction to estimate the position and rotation of the users head when the frame is displayed to them. But as the camera is fixed to the head-mounted display, we deal with this problem by shifting the camera image in the opposite direction to the rotation, which places objects in the frame where the user would expect them. This is implemented by leveraging the Asynchronous Timewarp functionality implemented by the Oculus SDK, which does this image shifting for us. Despite this, this doesn't correct any perceived lag caused by other objects moving relative to the background, or if an object is fixed in front of the camera. We discuss a potential solution to this shortcoming in Section 6.2.2.

In addition to this, the system of many different cameras are not synchronised. Our prototype attempts to deal with this by capturing from both cameras together rather than

capturing from both in two different threads. However, from observation, the ZED camera seems to appear to lag a number of frames behind the Realsense. This caused an effect where the occlusion based on the depth frame was *ahead* of the colour frame by tens of milliseconds, breaking the illusion of coherence between the virtual world and reality.

5.1.3 Occlusion

Implementing accurate occlusion is crucial to convince the user that rendered objects truly coexist with objects in the real world, rather than just being displayed on top. The occlusion handling implemented in our prototype is effective at accurately occluding virtual objects as shown in Figure 5.1. However, it is directly tied with the quality of the depth stream.

The Realsense F200 calculates depth by using an infrared projection of a complex pattern, and uses an infrared sensor to determine the surface shape and depth of every pixel based on this projected pattern. This has the effect where the resulting depth map is inaccurate on the edges of objects, and suffers from "shadow" artifacts where objects in front appear to cast a shadow of indeterminate depth values on objects behind. Additionally, the edges of objects that appear in colour are smooth, whilst the edges of objects in the depth map are jagged and appear aliased. This causes real objects such as the hand in Figure 5.1 to appear almost like a cardboard cut-out.

Figure 5.1: Occlusion in our prototype. The aliased effect is clearly shown around the edge of the hand compared to the colour stream.

5.1.4 Auto-focus

We implemented the auto-focus feature using the Horizontal Image Translation technique described in Section 2.4.1 and 4.6.2. This technique was effective at preventing the eyes of the user from converging or diverging by bringing the object in the centre of the screen into focus. Looking at Figure 5.2, we can see that the frames displayed on the Oculus Rift headset move apart from each other, but the polystyrene head is kept at roughly the same position in each eye. When displayed on the headset, the user is able to comfortable gaze at the head.

One limitation of this approach is how it deals with the background. As the disparity d of the object in the centre of the field of vision is reduced to 0, the disparity of objects in the background get shifted in the opposite direction. This causes the same uncomfortable feeling as before if the users gaze is in any other direction except straight ahead. This problem could be addressed in the future by building a disparity map from every pixel in the depth map, and warp the texture coordinates of the overlay instead of translating the entire image, causing the background to stay in place but the head to move by itself.

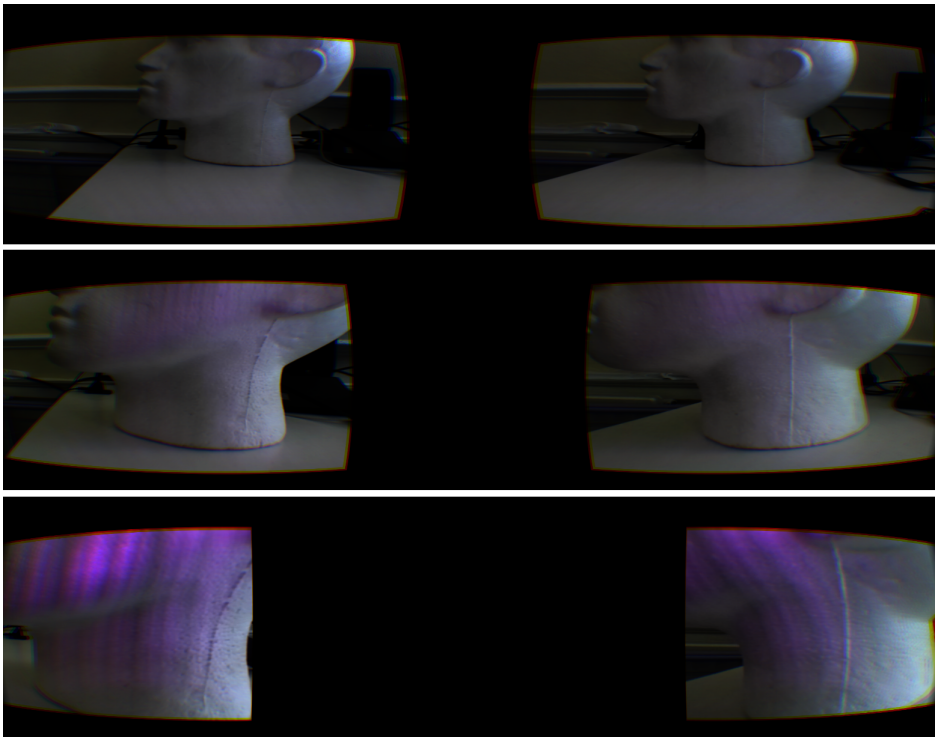

Figure 5.2: Horizontal Image Translation in action, implementing auto-focus. The position of the polystyrene head is kept at roughly the same position inside the users vision, which prevents their eyes from converging or diverging too much. The infrared projection from the Realsense can also be seen as the head gets closer.

5.1.5 KFusion

KinectFusion as an algorithm allowed us to achieve our objective of locating an object and projecting information on to it. However, the accuracy of the algorithm was not perfect and suffered from a number of issues. Due to the quality of the depth stream from the Realsense camera and the placement of the camera at runtime, the scene reconstruction created by KFusion would suffer with artifacts and holes in the mesh as seen in Figure 5.3.

When the Nelder-Mead optimisation is applied in Section 4.4.2, the cost function could never reach the theoretical minimum of 0, but instead would only approach this and reach some minimum value above 0, caused by the λ truncation designed to prevent outliers caused by holes in the mesh. As a result, we noticed that the transform generated by the Nelder-Mead optimisation could only converge close to the true transform as a result of the shortcoming of the cost function.

We deal with this effect in our prototype by allowing some leeway in the masking model for rendering overlays as described in *Handling Occlusion* in Section 4.3.3. Despite this, any overlay rendered onto the head such as a skull, or brain scan, still suffers from a very slight adjustment from where the user would expect the overlays to reside.

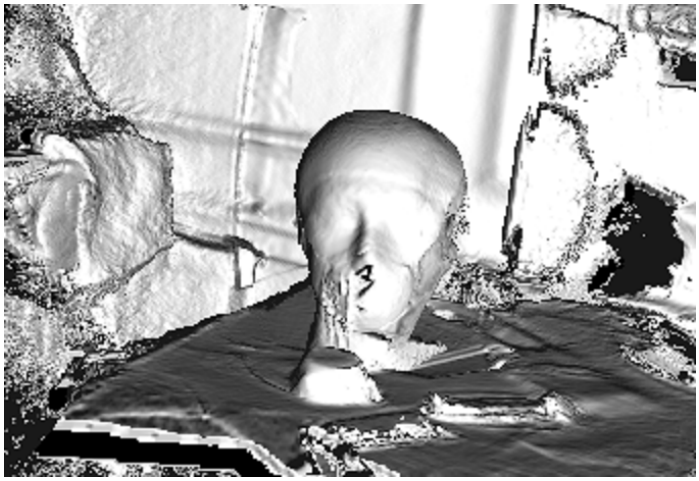

Figure 5.3: An example of a KinectFusion reconstruction of the polystyrene head model and the surrounding environment. We can notice that the jawline of the head is reconstructed very poorly compared to the forehead and scalp.

5.2 Hardware Issues

5.2.1 ZED Camera

Both the ZED Camera and Realsense F200 are high bandwidth devices, which both make use of the USB 3.0 standard, which has a maximum bandwidth of 640MB per second [61]. The ZED camera alone needs to send 2 x 1920-by-1080 images at 32 bits per pixel, which at 30 frames per second is approximately 498MB of data per second. This is approximately 78% of the maximum theoretical throughput of the USB 3.0 standard. As a result, when the ZED camera was connected to the same USB root hub as other devices such as the Realsense F200, we noticed some artifacts in the stream such as dropped frames or flickering. The Realsense F200 sends a 640-by-480 4-byte colour stream and 640-by-480 2-byte depth stream at 60 frames per second, which requires an additional bandwidth of approximately 111MB per second.

We determined that when both devices were operating, the USB controller in the motherboard seemed to struggle to deal with the bandwidth of the two cameras, alongside all the other peripherals connected such as the Oculus Rift. Our solution was to make use of an additional PCI-Express USB 3.0 extension card, which allowed us to add a 2nd USB controller to the system that was entirely dedicated to the ZED camera.

Furthermore, the ZED camera we had in possession had an issue where sometimes the maximum frame rate was severely capped to 2 frames per second at 1920-by-1080, and would simply crash at a 2K resolution. After some period of testing, we determined that the device was only operating at USB 2.0's *High-speed* (a maximum of 60MB/s) rather than *Super-speed* (a maximum of 640MB/s). This was later discovered to be caused by a faulty ZED camera as it was consistently reporting that it only supports High-speed to the operating system, despite being connected to a blue Super-speed USB port.

The Department of Computing Society was able to provide a replacement camera to allow us to implement the prototype before Stereolabs was able to send a replacement through the return merchandise authorization (RMA) process. Unfortunately, the replacement camera had a different issue where it would no longer respond after being used in any application. The only way to work around this was to reconnect the camera after every run. Additionally, the replacement camera had a different sensitivity to light wavelengths compared to the original camera, and was able to see the infrared projection produced by the Realsense at runtime.

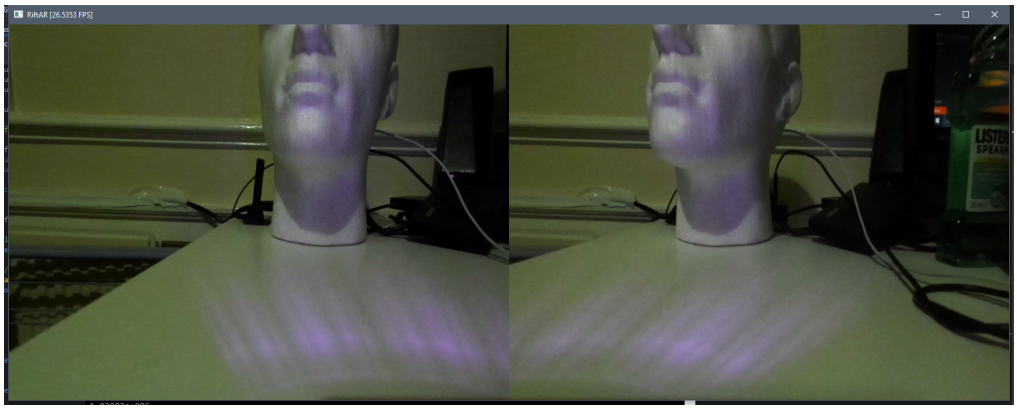

Figure 5.4: Infrared projection from the Realsense visible to the ZED.

5.2.2 Realsense F200

The Realsense F200 camera is built in a frame with a lot of cooling. For the purposes of our prototype, we decided to strip the Realsense camera's body and cooling to the point where it is just a sensor and an exposed PCB wrapped in a thin layer of plastic. A photograph of this setup is shown in Figure 5.5. This was done to significantly reduce the weight of the camera and allow the wearer of the prototype to have a more comfortable experience.

As expected, there were a number of drawbacks. We noticed that occasionally, the device would report a "Temperature Control Loop" warning which indicated that the device was likely reaching a temperature that was hotter than expected. Additionally, the camera would suffer from seemingly random reconnections, which happened more frequently when the cable was tugged. Our code deals with this by simply attempting to re-acquire the camera in the capture thread every time the grab function returns an error.

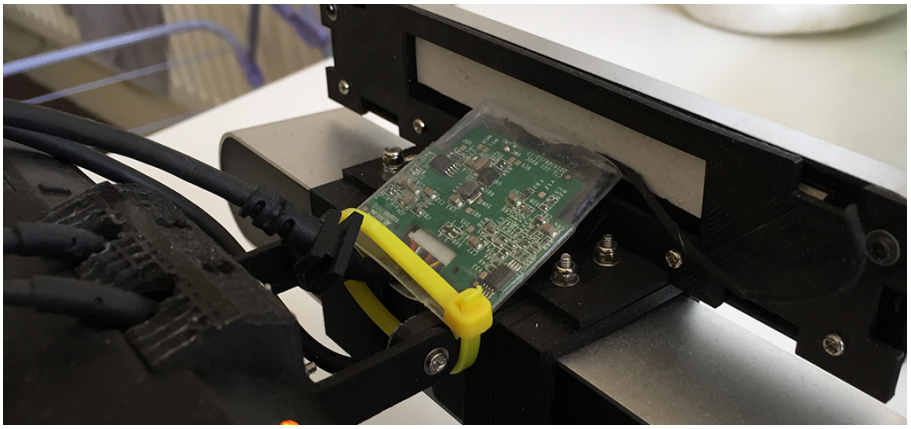

Figure 5.5: The Realsense camera with cooling stripped away.

6 Conclusion

6.1 Limitations

The first immediate restriction of the prototype is the resolution of the cameras. In the ideal scenario, the field of view of the ZED camera would precisely match the field of view of the Oculus Rift both horizontally and vertically, and have a one-to-one pixel correspondence when distortion is taken into account. The ZED camera provides a resolution of 1280-by-720 at 60 frames per second, which horizontally provides almost enough pixels, but the vertical field of view is much lower than the Rifts, which causes black bars to appear above and below the vision field.

Another limitation is the accuracy of the reconstruction of the scene from the KinectFusion algorithm. Given the low quality of the depth image, KinectFusion builds a relatively good reconstruction. However, as discussed in Section 5.1.5, some degenerate regions form depending on how the camera is moved. This subsequently affects both the tracking capabilities and the accuracy of the calculated location of the head model, due to an inaccurate signed distance volume used by the Nelder-Mead optimisation.

The Oculus SDK limited us when integrating the timewarp technique to implement Image Stabilisation described in Section 4.5. Originally, we were going to use our own method to calculate the corners of the image given a rotation offset from when the frame was captured to when the frame was displayed in the headset. However, the Oculus SDK implements Asynchronous Timewarp in a separate 'compositor' process, which could not be disabled. As a result, we were forced to make use of ATW to implement image stabilisation. This worked well, however as the timewarp was no longer under our control, it made other more advanced image stabilisation techniques such as Zoom much harder to implement.

6.2 Future Work

6.2.1 New Hardware Releases

Since the beginning of the project in November 2015, there has been a number of new hardware releases which might have influenced our hardware decisions when building the prototype. These days, modern hardware develops at a very rapid pace, and there are new devices with more powerful processors or greater resolutions being released year after year. Here we consider the latest releases of head-mounted displays and potential camera attachments that we could use in a future prototype.

Oculus CV1

In March 2016, pre-orders for the first Consumer Version (CV1) of the Oculus Rift opened on the Oculus website, with shipments beginning in late-April. It improves upon the DK2 in a number of ways, with a higher resolution, refresh rate and a much lighter and more ergonomic design. The box also comes with a number of other peripherals, such as a touch controller called the 'Oculus Remote', an enhanced infrared sensor and an Xbox One controller.

The CV1 headset increases the display resolution to 2160-by-1200 (1080-by-1200 in each eye) and increases the refresh rate to 90Hz [62], compared to the DK2 having a resolution of 1920-by-1080 and a refresh rate of 75Hz. The DK2's tracking capabilities was limited by the fact it only has an array of infrared LEDs on the front of the device. The CV1 now has infrared LEDs surrounding the entire body which allows complete 360-degree tracking. Additionally, the infrared sensor is now mounted on a stand and placed on the side of the desk, increasing the effective tracking area to 5-by-11 feet.

HTC Vive

The HTC Vive is another head-mounted display developed by both HTC and Valve Corporation, and released in April 2016 [63]. It has a display resolution of 2160-by-1200 and a refresh rate of 90Hz, with a 110-degree field of view. It is the most expensive of all the consumer head-mounted displays released in the last 6 months, with a RRP of £689. Despite this, it has excellent tracking capabilities. Similar to the Oculus Rift, the Vive comes with an array of infrared LEDs scattered around the headset. The box comes with two wireless infrared "Lighthouse" cameras, which when placed in the corners of a room allow the wearer to move freely within a large area, illustrated in Figure 6.1. Additionally, the infrared cameras are ca-

pable of dealing with multiple headsets in the same area. Alongside the headset and cameras, the Vive kit comes with a pair of wireless controllers, which contain infrared LEDs used in conjunction with the Vive headset and are tracked using the infrared cameras.

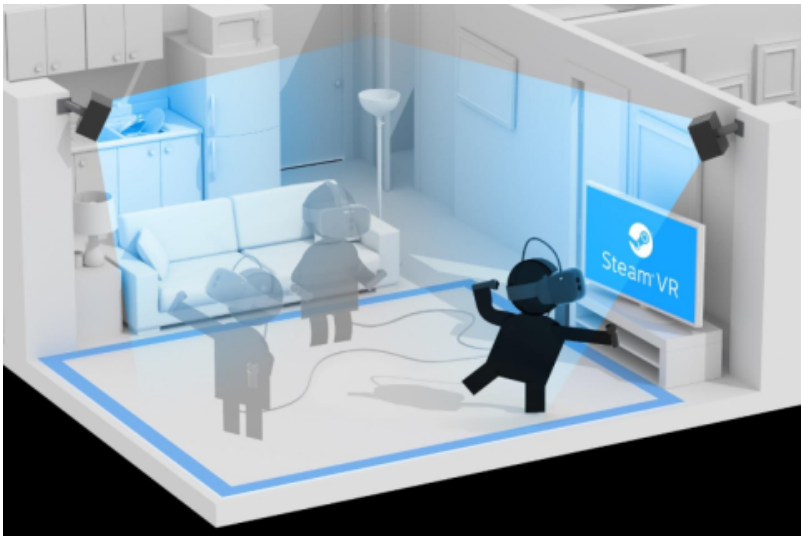

Figure 6.1: An illustration of the Lighthouse camera system used with the HTC Vive. Adapted from "Advanced VR Rendering" at GDC 2015 [64].

Intel Realsense SR300 Camera

The Realsense SR300 is second generation front-facing camera developed by Intel Labs and designed as an evolution of the Realsense F200 camera which we used in our prototype [65]. Similarly to the F200, the SR300 implements an infrared laser projector system, which is read by an infrared camera and transformed into a depth map. However, the SR300 builds upon this by using a so called "Fast VGA depth mode" which reduces the exposure time and allows the camera to detect a motion of up to 2 metres per second. It also provides additional improvements such as improved colour quality under low light, decreased power consumption, and better synchronisation between colour and depth images.

As seen in Figure 6.2, the quality of the depth stream is significantly improved. The SR300 is optimised for a range between 0.2 metres and 1.2 metres much like the F200, but at larger distances the SR300 suffers from much less noise with a smaller number of indeterminate points. As a result, a vast number of higher level features such as Hand Tracking and Face Tracking have an expanded working range. The depth range of the SR300 is improved by

50% - 60%, and is still able to detect a hand at 120cm whilst the F200 is unable to detect anything.

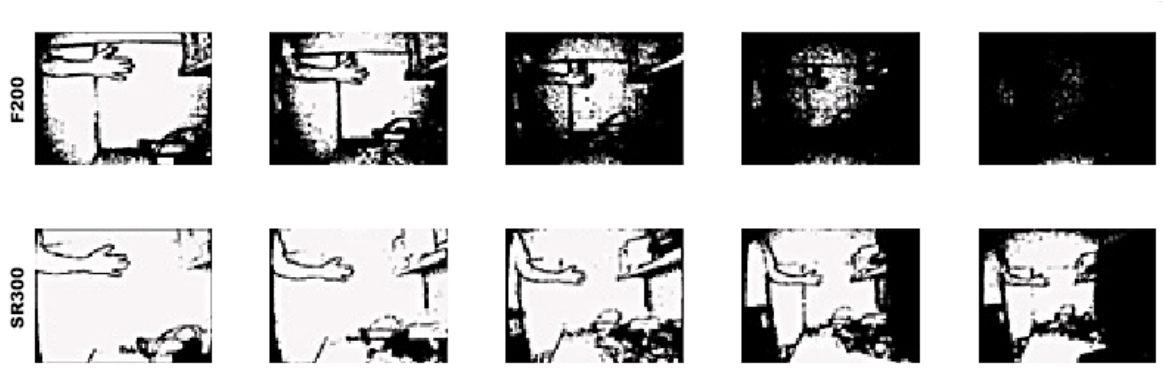

Figure 6.2: A comparison of the F200 depth stream (top) and the SR300 depth stream (bottom) ranging from 80cm to 160cm. Adapted from "A Comparison of Intel RealSense Front-Facing Camera SR300 and F200" [65].

6.2.2 Enhancements

Looking at the final prototype, there are a range of algorithms which we applied in an effort to enhance the experience. We employed methods to reduce latency caused by subsequent head movement after a frame is captured, and Horizontal Image Translation to help prevent an uncomfortable experience for the wearer. However, there are a number of other methods we can make use of to help improve the experience further.

Tracking Recovery

As mentioned in Section 6.1, the KinectFusion algorithm is susceptible to tracking failure when the camera is moved fast between frames. Glocker et. al. in [66] proposed a relocalisation method based on frame encoding which stores successful tracking states and then uses the most similar keyframes to reinitialise the tracking system and restore tracking capability. Integrating a similar system would allow our prototype to be much more robust against tracking failure, usually caused by the wearer moving too fast, or obstructing most of the scene.

Optical Flow Based Warping

In our prototype, we deal with the latency issue caused by movement between frame capture and frame display by shifting the image to place image features in the position that the wearer would expect them. However, this doesn't solve the issue where objects moving in the scene are still tens of milliseconds behind where they should be. For example, if the wearer holds their hand in front of the camera and then rotates whilst holding their hand in the same position, they see an effect where their hand will appear to lag behind when rotating, due to the difference in time between frame capture and frame display.

Taking our solution to the next level, we can determine the **Optical Flow** between two consecutive frames using the Lucas-Kanade technique [67] or the Farnebäck method [68]. Let $\mathbf{p}_A$ be the location of a point in an image, and $\mathbf{p}_B$ be the corresponding point in the next frame. We can define a *displacement map* $d : \mathbb{R}^2 \mapsto \mathbb{R}^2$ as follows:

$$d(\mathbf{p}_A) = \mathbf{p}_B - \mathbf{p}_A \tag{6.1}$$

If we have an optical flow between frame n as image A and frame $n-1$ as image B such as in Figure 6.3, represented as a displacement map, we can use it to estimate the position of of each point in frame $n+1$:

$$\mathbf{p}_{n+1} = \mathbf{p}_n - d(\mathbf{p}_n) \tag{6.2}$$

This could be implemented by generating an OpenGL texture which contains the displacement to the previous frame's corresponding points as an `GL_RG16F` texture, and use equation 6.2 to transform the UV coordinate when rendering the frame to the headset.

Depth Map Filtering

The depth stream produced by the Realsense F200 camera suffers from a number of artifacts due to the use of infrared light to project a pattern, which creates occasional holes, uneven edges and borders around objects such as a hand. Performing reprojection of the depth values to warp the depth information to the intrinsics and extrinsics of the ZED camera only exacerbates the problem, as seen in Figure 6.4.

There are a number of methods we could use in future work to improve the situation.

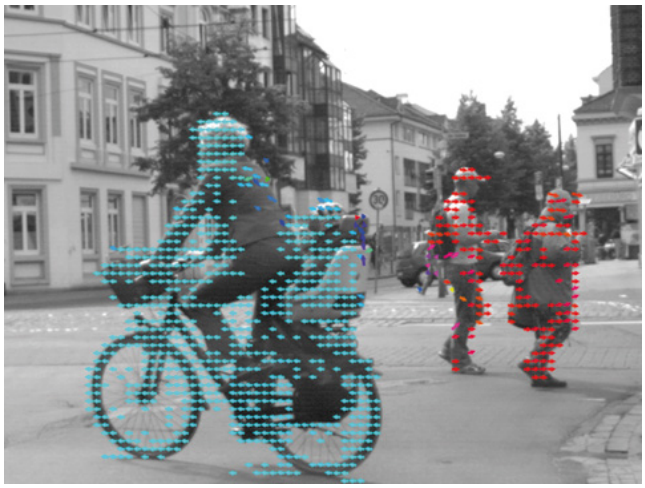

Figure 6.3: An example of optical flow detecting motion between two images. The bike (blue) is moving to the left, and the couple (red) is moving to the right. Adapted from "CVB Optical Flow" - URL: `http://www.imaco.pl/pl/produkty/rodzina/cvb.optical-flow`.

Firstly, the depth map could be cleaned up using a bilateral filter [69], which is an edge-preserving and noise-reducing smoothing filter for images. This would help improve the quality of the depth map when used for occlusion handling, however there would still be structural problems with the resulting depth map such as indeterminate borders around objects due to shadowing.

Yang et. al. propose a method in [70] which performs high quality adaptive colour-guided depth recovery. Using the ZED camera as a colour guide, this algorithm uses an auto-regressive model to recover depth information. It has the capability of filling in holes in the depth stream to produce a much smoother looking depth map, which results in a better looking integration between the virtual world and reality.

Facial Recognition for HIT

Currently, our prototype uses a rectangular region in the centre of the depth stream to determine the depth of the object immediately in front of the sensor, and uses this as a focal point for Horizontal Image Translation. This is a reasonable estimation of the point of interest, however its main drawback is having to keep the object in the centre of the vision at all times. For example, if a surgeon is interested in keeping focus on a patient at all times, the patient will need to be in the centre of the surgeons vision. If the surgeon looks away slightly, then the patient will become out of focus as the system will decide that a wall behind the patient should become the new focal point.

Figure 6.4: Holes and borders in the depth map produced by the Realsense F200. White pixels are indeterminate.

We could deal with this by first introducing hysteresis by requiring a certain amount of time before the focal point adjusts, or interpolating the focal point at a slow rate. Additionally, we can employ the `dlib` machine learning toolkit [71] and use its facial recognition features to attempt to find a rectangular sub-image which contains the face of the patient. We could then find the modal depth d_m from a histogram of all the depth values in this sub-image, and focus on d_m where the rectangular region in the centre of the image intersects with the rectangular sub-image containing the face. This can more intelligently direct the focus to the subject of interest rather than focusing on a wall directly behind the patient when the gaze of the surgeon is slightly off.

6.3 Further Applications

Our prototype is just one of many possible applications of marker-free augmented reality using a head-mounted display and an array of one or more cameras. In this section, we briefly explore a number of other applications which are only possible with a set-up like this, compared to other pass-through augmented reality platforms such as the Microsoft HoloLens.

6.3.1 Zoom

One immediate benefit of capturing frames from a camera and presenting them on a head-mounted display, is the ability to *fully* control the image that the user is seeing at any given moment. Using a camera with a sufficiently high resolution, we could implement a prototype which performs digital zooming and allows the wearer to gain telescopic vision. An immediate

problem that we face with this application is motion sickness. If we simply expand the image displayed on head-mounted display, it will amplify the effects of any rotational movement of the head, and unless the wearer has a exceptional stability, there will be a large degree of shaking.

One solution to this problem that can be employed is image stabilisation similar to Asynchronous Timewarp (ATW), which is addressed in Section 4.5.2. If we have a rotation matrix defined as the change in orientation of the camera between the first frame and the current frame, we can *dampen* this rotation by converting it to angle-axis form, and dividing it by the telescopic zoom factor. We can then perform a similar transformation to ATW in the opposite direction of the rotation to sample a different part of the image, and make the image appear to move as much as it would if there was no zoom applied.

6.3.2 Infrared Vision

Another interesting application is overlaying sensor data from a completely different range of the Electromagnetic spectrum. Typical Night Vision Devices (NVD) work by passing near-infrared photons through an **Image intensifier tube**, which increases the energy of the photons and moves them to the visible light spectrum for viewing. We could potentially build a system which uses a camera such as the Intel Realsense F200 that has both a colour and infrared stream warped to match the colour intrinsics and extrinsics, then allow the user to toggle between the two. More interestingly, we could use some equipment sensitive to high frequency wavelengths such as ultraviolet or x-ray radiation, and allow the user to experience some vivid sights.

6.3.3 Therapeutical Applications

There is a huge potential for a number of different applications in a therapeutical setting. **Cognitive Behavioural Therapy (CBT)** is a type of psychotherapy that help people manage their problems by changing the way they think and behave. CBT is used to treat a wide variety of problems such as anxiety, depression, low self-esteem and other problems. However, it has use in a number of more specialised treatments, such as treating phobias [72].

Treating phobias is usually done by repeated exposures, starting with low stress situations in an effort to train the brain to avoid the *flight-fight* response. A system can be developed using a combination of virtual reality and augmented reality technologies to simulate these situations. For example, a depth camera can be used in conjunction with a colour camera

to render an enclosed space to help with claustrophobia, or render a tarantula crawling on a patients hand to help with arachnophobia.

Phantom limb pain (PLP) sensations are described as ongoing painful sensations that appear to come from a limb which is no longer there. It is thought to be not caused by trauma, but from mixed signals from your brain or spinal cord. One method used to deal PLP is known as **Mirror box therapy**, which uses a mirror to convince the brain that a missing limb is still attached, as shown in Figure 6.5. We can employ augmented reality as a more sophisticated form of mirror therapy [73]. By rendering a 3D representation of a missing limb in the right location, the brain can be fooled into thinking that that the limb still exists.

Figure 6.5: A patient using mirror therapy to deal with phantom leg pain. Adapted from `https://www.thinglink.com/scene/712474088106360834`

Bibliography

[1] Rachael Revesz. Google cardboard saves a babys life by helping doctors map out heart surgery. http://www.independent.co.uk/news/world/americas/google-cardboard-saves-a-baby-s-life-by-helping-doctors-map-out-heart-surgery-a6802671html. [accessed 2016-06-02].

[2] Winged1der. A Photo of a NASA X38 display. https://en.wikipedia.org/wiki/Augmented_reality#/media/File:X38_landing_display_from_LandForm_Hybrid_Synthetic_Vision_system.jpg. [accessed 2016-06-02].

[3] George C. Stockman Linda G. Shapiro. *Computer Vision*. Prentice Hall, 2001.

[4] Wikipedia. Sobel Filter. https://en.wikipedia.org/wiki/Sobel_operator. [accessed 2016-04-14].

[5] Ching-Kuang Shene. Homogeneous Coordinates. http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/geometry/homo-coor.html. [accessed 2016-05-02].

[6] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[7] Wikipedia. Camera Matrix. https://en.wikipedia.org/wiki/Camera_Matrix. [accessed 2016-04-16].

[8] Kyle Simek. Dissecting the Camera Matrix, Part 1: Extrinsic/Intrinsic Decomposition. https://ksimek.github.io/2012/08/14/decompose/. [accessed 2016-04-17].

[9] A. J. Davison. Real-time simultaneous localisation and mapping with a single camera. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 1403–1410 vol.2, Oct 2003.

[10] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse. MonoSLAM: Real-Time Single Camera SLAM. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(6):1052–1067, June 2007.

[11] G. Klein and D. Murray. Parallel tracking and mapping for small ar workspaces. In *Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on*, pages 225–234, Nov 2007.

[12] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *Mixed and Augmented Reality (ISMAR), 2011 10th IEEE International Symposium on*, pages 127–136, Oct 2011.

[13] P. J. Besl and H. D. McKay. A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, Feb 1992.

[14] Simon Reeve and Jason Flock. Basic Principles of Stereoscopic 3D. `http://www.sky.com/shop/__PDF/3D/Basic_Principles_of_Stereoscopic_3D_v1.pdf`. [accessed 2016-05-23].

[15] J. S. McVeigh, Mel Siegel, and Angel Jordan. Algorithm for automated eye strain reduction in real stereoscopic images and sequences. In *Human Vision and Electronic Imaging*, volume 2657, pages 307 – 316, February 1996.

[16] Oculus. Oculus Rift: Step Into The Game. `https://www.kickstarter.com/projects/1523379957/oculus-rift-step-into-the-game`. [accessed 2016-04-08].

[17] Ben Kuchera. VR on the cheap: a review of the Vuzix iWear VR920 video eyewear. `http://arstechnica.com/gadgets/2007/11/virtual-reality-headset-review/`. [accessed 2016-05-05].

[18] Wikipedia. Oculus Rift. `https://en.wikipedia.org/wiki/Oculus_Rift#Development_Kit_1`. [accessed 2016-04-07].

[19] Ben Lang. GDC 2014: Oculus Rift Developer Kit 2 (DK2) Pre-orders Start Today for $350, Ships in July. `http://www.roadtovr.com/oculus-rift-developer-kit-2-dk2-pre-order-release-date-specs-gdc-2014/`. [accessed 2016-04-08].

[20] Anton Belev. First Impressions from the New Oculus Rift DK2 3D VR HMD. `http://3dvision-blog.com/9326-first-impressions-from-the-new-oculus-rift-dk2-3d-vr-hmd/`. [accessed 2016-05-03].

[21] Oculus. Rendering to the Oculus Rift. `https://developer.oculus.com/`

documentation/pcsdk/latest/concepts/dg-render/. [accessed 2016-05-24].

[22] Wikipedia. Google Cardboard. https://en.wikipedia.org/wiki/Google_Cardboard. [accessed 2016-04-12].

[23] Samsung. Samsung Gear VR - The Official Samsung Galaxy Site. http://www.samsung.com/global/galaxy/wearables/gear-vr/. [accessed 2016-04-12].

[24] Stereolabs. ZED: 3D Camera for AR/VR and Autonomous Navigation. https://www.stereolabs.com/zed/specs/. [accessed 2016-04-24].

[25] Stereolabs. ZED SDK. https://www.stereolabs.com/developers/documentation/API/. [accessed 2016-04-24].

[26] Code Laboratories Inc. DUO MLX - USB Stereo Camera. https://duo3d.com/product/duo-minilx-lv1. [accessed 2016-04-25].

[27] Code Laboratories Inc. Stack Overflow: DUO 3D Mini Sensor by Code Laboratories. http://stackoverflow.com/a/28555991. [accessed 2016-04-25].

[28] Creative Labs. Intel RealSense 3D Camera: Technical Specifications. http://support.creative.com/kb/ShowArticle.aspx?sid=124661. [accessed 2016-04-28].

[29] GitHub. IntelRealSense/librealsense. https://github.com/IntelRealSense/librealsense. [accessed 2016-04-28].

[30] Jason Tanz. Kinect Hackers Are Changing the Future of Robotics. http://www.wired.com/2011/06/mf_kinect/all/1. [accessed 2016-04-28].

[31] Microsoft Robotics. Kinect Sensor. https://msdn.microsoft.com/en-gb/library/hh438998.aspx. [accessed 2016-04-28].

[32] Wikipedia. Direct3D. https://en.wikipedia.org/wiki/Direct3D. [accessed 2016-05-04].

[33] Aras Pranckeviius. Metal, a new graphics API for iOS 8. http://blogs.unity3d.com/2014/07/03/metal-a-new-graphics-api-for-ios-8/. [accessed 2016-05-04].

[34] GitHub. glfw/glfw. https://github.com/glfw/glfw. [accessed 2016-05-05].

[35] Khronos Group. Rendering Pipeline Overview. https://www.opengl.org/wiki/Rendering_Pipeline_Overview. [accessed 2016-05-10].

[36] Nicolas P. Rougier. Modern OpenGL. https://glumpy.github.io/modern-gl.html. [accessed 2016-05-05].

[37] Khronos Group. Buffer Object. https://www.opengl.org/wiki/Buffer_Object. [accessed 2016-05-10].

[38] Khronos Group. Framebuffer Object. https://www.opengl.org/wiki/Framebuffer_Object. [accessed 2016-05-10].

[39] Khronos Group. Compute Shader. https://www.opengl.org/wiki/Compute_Shader. [accessed 2016-05-11].

[40] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/. [accessed 2016-05-11].

[41] Itseez. OpenCV. http://itseez.com/opencv/. [accessed 2016-04-18].

[42] OpenCV Dev Team. Miscellaneous Image Transformations. http://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html#adaptivethreshold. [accessed 2016-05-15].

[43] MathWorks. Morphological Dilation and Erosion. https://uk.mathworks.com/help/images/morphological-dilation-and-erosion.html. [accessed 2016-05-15].

[44] GitHub. opencv/calibinit.cpp at master Itseez/opencv. https://github.com/Itseez/opencv/blob/master/modules/calib3d/src/calibinit.cpp#L228. [accessed 2016-05-15].

[45] Zhengyou Zhang. A Flexible New Technique for Camera Calibration. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(11):1330–1334, November 2000.

[46] V. Lepetit, F. Moreno-Noguer, and P. Fua. EPnP: An Accurate O(n) Solution to the PnP Problem. *International Journal Computer Vision*, 81(2), 2009.

[47] Xiao-Shan Gao, Xiao-Rong Hou, Jianliang Tang, and Hang-Fei Cheng. Complete Solution Classification for the Perspective-Three-Point Problem. *IEEE Trans. Pattern Anal. Mach. Intell.*, 25(8):930–943, August 2003.

[48] GitHub. opencv/calibration.cpp at master Itseez/opencv. https://github.com/Itseez/opencv/blob/master/modules/calib3d/src/calibration.cpp#L1695. [accessed 2016-05-20].

[49] IntelRealsense. Projection and Deprojection in librealsense. https://github.com/IntelRealSense/librealsense/blob/master/doc/projection.md#distortion-models. [accessed 2016-05-22].

[50] Nvidia. CUDA Toolkit Documentation - Atomic Functions. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions. [accessed 2016-05-23].

[51] Shijun Ji. Research on Compression Methods for Binary STL File. *Journal of Computational Information Systems*, 6(8):2587–2593, 2010.

[52] Kyle Simek. Calibrated Cameras in OpenGL without glFrustum. https://ksimek.github.io/2013/06/03/calibrated_cameras_in_opengl/. [accessed 2016-05-23].

[53] Leandra Vicci. Quaternions and rotations in 3-space: The algebra and its geometric interpretation. Technical report, Chapel Hill, NC, USA, 2001.

[54] Salix alba. Efficient way to apply mirror effect on quaternion rotation? http://stackoverflow.com/a/32441900. [accessed 2016-06-02].

[55] J A Nelder and R Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308–313, 1965.

[56] Gregory G Slabaugh. Computing Euler angles from a rotation matrix. *Retrieved on August*, 6(2000):39–63, 1999.

[57] B.A. Davis, K. Bryla, and P.A. Benton. *Oculus Rift in Action*. Manning Publications Company, 2015.

[58] Dean Beeler and Anuj Gosalia. Asynchronous Timewarp on Oculus Rift. https://developer.oculus.com/blog/asynchronous-timewarp-on-oculus-rift/. [accessed 2016-06-04].

[59] videocardz.com. NVIDIA GeForce GTX 780 Ti official specifications. http://videocardz.com/47576/nvidia-geforce-gtx-780-ti-official-specifications. [accessed 2016-06-08].

[60] Oculus. OVR_CAPI.h File Reference. https://developer.oculus.com/doc/1.4.0-libovr/_o_v_r___c_a_p_i_8h.html#aa84f958b8d78f2594adf6fb0ad372558. [accessed 2016-06-08].

[61] Inc. USB Implementers Forum. USB 3.1 Specification. http://www.usb.org/

developers/docs/. [accessed 2016-06-06].

[62] Digital Trends. Spec comparison: The rift is less expensive than the vive, but is it a better value? http://www.digitaltrends.com/virtual-reality/oculus-rift-vs-htc-vive/. [accessed 2016-06-06].

[63] HTC. Vive — Product Hardware. http://www.htcvive.com/uk/product/. [accessed 2016-06-06].

[64] Alex Vlachos. Advanced VR Rendering. http://media.steampowered.com/apps/valve/2015/Alex_Vlachos_Advanced_VR_Rendering_GDC2015.pdf. [accessed 2016-06-08].

[65] Nhuy L. A Comparison of Intel RealSense Front-Facing Camera SR300 and F200. https://software.intel.com/en-us/articles/a-comparison-of-intel-realsensetm-front-facing-camera-sr300-and-f200. [accessed 2016-06-06].

[66] Ben Glocker, Jamie Shotton, Antonio Criminisi, and Shahram Izadi. Real-time rgb-d camera relocalization via randomized ferns for keyframe encoding. *TVCG*, September 2014.

[67] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81, pages 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[68] Gunnar Farnebäck. Two-frame motion estimation based on polynomial expansion. In *Proceedings of the 13th Scandinavian Conference on Image Analysis*, SCIA'03, pages 363–370, Berlin, Heidelberg, 2003. Springer-Verlag.

[69] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Computer Vision, 1998. Sixth International Conference on*, pages 839–846, Jan 1998.

[70] J. Yang, X. Ye, K. Li, C. Hou, and Y. Wang. Color-guided depth recovery from rgb-d data using an adaptive autoregressive model. *IEEE Transactions on Image Processing*, 23(8):3443–3458, Aug 2014.

[71] Davis E. King. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research*, 10:1755–1758, 2009.

[72] Thomas Straube, Madlen Glauer, Stefan Dilger, Hans-Joachim Mentzel, and Wolf-

gang H.R. Miltner. Effects of cognitive-behavioral therapy on brain activation in specific phobia. *NeuroImage*, 29(1):125 – 135, 2006.

[73] Max Ortiz-Catalan, Nichlas Sander, Morten B. Kristoffersen, Bo Hkansson, and Rickard Brnemark. Treatment of phantom limb pain (plp) based on augmented reality and gaming controlled by myoelectric pattern recognition: a case study of a chronic plp patient. *Frontiers in Neuroscience*, 8(24), 2014.