Calibrating Head-Coupled Virtual Reality Systems

by

Alexander Stevenson

B.Sc., The University of British Columbia, 2000

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

April 2002

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

April 23, 2002
Date

Department of Computer Science
The University of British Columbia
201-2366 Main Mall
Vancouver, BC
Canada V6T 1Z4
http://www.cs.ubc.ca

Abstract

Head-tracking virtual environments are difficult to implement because of the need to calibrate such systems accurately, as well as the difficulty in computing the correct off-axis image for a given eye location. The situation is further complicated by the use of multiple screens, the need to change the calibration for different users, and the desire to write portable software which can be reused on different hardware with varying screen configurations.

This thesis presents a solution to these problems, allowing greatly simplified development of head-tracking software. By making use of the head-tracking sensors built into the environment, we can quickly and accurately calibrate not only user-specific measurements, such as eye-positions, but also system measurements, such as the size and locations of display screens. A method of doing this calibration is developed, as well as a software library which will read a system configuration and integrate with OpenGL to compute correct off-axis projections for a user's viewing position. The calibration makes use of a novel "sighting" technique which has the great advantage of accurately finding the true rotational centre of a user's eyes. To complement this, the software library includes functions which predict the optical centre of a user's eye based on a given gaze point.

As a demonstration of both the calibration method and the utility library, a hardware rendering application is discussed. This application performs the real-time rendering of view-dependent LaFortune reflectance functions in graphics hardware. As with all view-dependent lighting methods, both the viewing angle and position of the light are taken into account while rendering. Head-coupling allows the system to use the user's true viewing direction in the lighting computation, and the position of the virtual light is controlled by a 3D sensor in the user's hand. The method in which the view-dependent lighting model is implemented in hardware is explained, as well as possible improvements.

Throughout, the Polhemus FASTRAK is used as the tracking system, though all the results are easily applicable to any six degree-of-freedom tracking system.

Contents

List of Tables

List of Figures

Acknowledgements

I wish to thank the many people who have helped to make this thesis possible. I am deeply indebted to my supervisor Kellogg Booth who first helped convince me to enter the graduate program and has supported and encouraged me throughout. I am embarrassed to think of the patience he has shown with my specious thesis schedule, and I owe him a great debt of gratitude for his understanding and flexibility.

Wolfgang Heidrich shared the ideas which form the real-time rendering aspects of this thesis, and introduced me to the fun of trying to find new uses for graphics hardware. Barry Po helped me understand off-axis projections, and I hope he finds this thesis useful. Sid Fels provided many helpful suggestions which improved this thesis greatly. Brian Fisher was the source of many enjoyable discussions, and I particularly appreciated his pragmatic philosophies about academia.

I would like to thank all the members of the Imager Graphics Lab, and especially Arthur Louie, Brook Bakay, Adrian Secord, Matthew Brown, and Hamish Carr, for their support and friendship. I could not ask for better company during those late night coding sessions that are *de rigueur* among us computer scientists. Many thanks also to my lifelong friend Robert Bruce who listened to me ramble about my thesis and then pointed out my mistakes.

I would like to thank my family for the countless ways they supported and encouraged me throughout this thesis and before, and my father for helping to proofread. Finally, my deepest thanks go to Jacqueline Carey who has put up with me more than any person should have to and shows every sign of continuing to do so for the foreseeable future, even though she should know better. Thank you all!

ALEXANDER STEVENSON

The University of British Columbia
April 2002

Chapter 1

Introduction

It would be safe to say every computer graphics laboratory has a number of experiments and visualisations that would benefit from display in a head-tracking environment. Not only does head-tracking add to a user's ability to perceive depth in a field where "graphics" is all but synonymous with 3D, but research has shown the benefit head-coupling has on task performance in virtual environments [arth93]. Despite its benefits, however, head-coupling is rarely used because of the inconvenience. The added complexity of dealing with a tracking system, off-axis projections, calibrating user eye positions, calibrating screen positions, registering the virtual world with the physical one, and maintaining and debugging the associated code can easily double the size of a modest project. Instead, most computer graphics projects fall under the category of simple visualisations and the more weighty term of *virtual reality* is usually applied to those for which the time and effort has been expended to allow greater user interaction.

In our lab, the Imager Graphics Lab at the University of British Columbia, we found ourselves very much in the situation described above. Despite having several Polhemus FASTRAK tracking systems, which had made their way back into regular lab use after I developed a cross-platform driver for them as part of my undergraduate thesis [stev00], head-tracking was either done for its own sake or not

at all. That is to say the few projects that used head-tracking existed to provide a *demonstration of head-tracking*, and did not use it as a mere visualisation tool. It is my hope that this thesis, and it's companion software libraries and programs, will help to rectify this situation, allowing anyone who decides his application would benefit from head-coupling to add it with a minimum of effort and added complexity.

1.1 Background

The term "virtual reality" was introduced by Ivan Sutherland in the 1960s. In his vision of virtual reality [suth65, suth68], a user wears a device on his head which places a small screen directly in front of each eye. These screens are updated based on the direction the user is facing to show an appropriate view of a virtual world. Today, the term "virtual reality" is broadly used to describe many types of three dimensional displays, and Sutherland's VR is usually referred to as "head-mounted" and "immersive", because of a user's inability to see anything but the screens directly in front of his eyes.

We will use the term "head-coupled virtual reality" to refer to a system that tracks the position of a user's head, but is not mounted on it.[1] Instead of small screens mounted in front of a user's eyes, regular computer displays are used to display virtual objects that are displayed correctly based on a user's eye positions. These images correspond exactly to what a user would see if the virtual world were real, and located in a volume extending behind the plane of the screen. Because of this notion of a volume in which the scene is viewed, this type of VR is sometimes referred to as "fish tank VR", though is more often termed "head-coupled VR". An additional technique that is frequently used with head-coupled VR to increase the perception of depth is the stereoscopic display. In these displays, a user wears shut-

[1] Of course, head-mounted displays are also "coupled" to the position of a user's head. Throughout this thesis, however, when we use the term "head-coupled VR" we will exclude head-mounted displays and refer instead to displays where the user's perspective of the screens changes as their head moves.

tered LCD glasses, which prevent both eyes from seeing the screen simultaneously. The glasses are synchronised with the refresh rate of the screens, allowing different images to be viewed by each eye. This allows the system to show the correct perspective image for each eye position, which increases the accuracy of the display as well as allowing the user to perceive depth.

Both head-coupling and stereoscopy are used in "augmented reality" systems: VR systems that are used to overlay displays of virtual objects over a user's view of real physical objects. These systems are like their all-virtual counterparts except that some of the objects seen are in fact real objects, while others are computer generated displays. Calibration for these systems is especially critical because of the fact that the virtual world must be registered precisely with the real objects.

Many parameters must be considered in order to generate the correct image for each eye. Naturally, the positions of the screens must be known, as well as the location of the user's head, but the situation is even more complicated that it would appear at first glance. Deering [deer92] discusses several issues which should be taken into account when designing a head-coupled VR system. While some issues are no longer as important as they once were, many are still valid, and tricky to contend with. For instance, the effect of CRT screen curvature is less of a problem with modern CRT technology than it was in 1992, and can be avoided entirely by the use of projection systems or LCD panels. Deering's observation that the optical centre of the eye moves depending on gaze direction, however, is still a problem.

The fact that head-coupled VR requires both the position of the screen, or screens, as well as the position of a user's eyes means it is heavily reliant on calibration. With head-mounted VR, it is fairly easy to assume that the user's eyes lie directly in front of the small screens installed in the helmet, and the location of those screens is known from the construction of the helmet itself. On the other hand, head-coupled VR frequently makes use of existing screens, like those on a user's desk, which can be arbitrarily located. Further, the sensor used to locate the

user's head can be in various positions with respect to the eyes. Calculating not only where the eyes are in relation to that sensor, but also re-calibrating for different users, whose inter-ocular distances vary, is important to the display of accurate VR images.

The calibration process is critical to the success of head-coupled VR. Not only does it determine the quality of the rendering, but how it is designed greatly affects the overall usefulness of head-coupled VR. If the calibration procedure requires a large amount of time and effort, users and experimenters alike are deterred from using head-coupled VR in their applications.

1.2 Contributions of the thesis

This thesis describes the development of a method for performing fast, simple calibration for head-coupled VR displays. A novel technique for determining the location of points in 3D, as well as the rotational centre of the eye, is introduced. This technique uses the notion of "sighting" a point from different positions to pinpoint its location. This new method for finding eye position is, in most cases, faster to perform and more accurate than traditional methods, and should facilitate future research into the effects of calibration errors on a user's performance in a VR environment.

The text of this thesis provides a description of the motivation and development of the calibration method, but much of the value in this work lies in the accompanying software. The calibration method itself is fully implemented in software in such a way as to allow it to be used by others needing similar calibration. The software guides a user through the calibration process, as well as automatically performing necessary measurements and calculations using a Polhemus FASTRAK tracking system. The results of this calibration are written to text files, which can be easily edited manually if necessary. Utility libraries are also included that allow a developer to read in the calibration data from the text files and use them to generate

correct off-axis perspectives in OpenGL, without having to know the complexities of the underlying math.

As an example of how this software can facilitate the addition of head-coupling to existing software, head-tracking was added to a non-trivial real-time rendering application: an object viewer capable of rendering the LaFortune reflectance model in real-time. This application was further expanded to allow a user-controlled, virtual local light. This thesis will describe both the advantages of using head-tracking in this application, as well as the hardware rendering method used to calculate the Lafortune reflectance model in real-time.

Finally, the thesis contains detailed instructions on how to run the calibration software and how to integrate the calibration method into other applications.

1.3 Overview of the thesis

This thesis is intended not only to show the development of the calibration method, but also to be used as a guide for those wishing to use the calibration system in their own work. With that in mind, descriptions of how to use the calibration software, and how to integrate it into other applications, have been placed in appendices. Those who only need to know how to run the calibration should read Appendix A, and might find the description of file formats in Appendix C useful. Those wishing to develop new applications should look at Appendix B, as well as the appropriate header files in the source code. Otherwise, the remainder of the thesis can be referenced as curiosity dictates.

Chapter 2 describes the elements behind the design of the calibration system. First it examines the requirements our system needs to fulfil, and then describes some design goals for the system. The method we will use for finding points in 3D is described, and analysed in terms of the advantages it provides, specifically with regard to locating eye positions.

Chapter 3 describes the complete calibration procedure, as well as the depen-

dencies between individual calibration steps. The reason for each calibration step is explained, and we see how the design goals described in Chapter 2 are applied in the final system. For an idea of how accurate the calibration system is in practice, sample calibration data has been collected and included in Appendix E.

Chapter 4 explains how, given a set of calibration data, an application makes use of that data to produce a correct final image. How the calibration data is used to compute an appropriate off-axis projection is explained in some detail. For reference, the source code implementation of this step has also been included, in Appendix F.

With the calibration system fully described by the preceding chapters, Chapter 5 looks at a practical application which makes use of the calibration software: a view-dependent lighting model, which can be explored in real-time using head-tracking and a user-controlled light position. This example is interesting both because of the calibration and head-tracking, explained in earlier chapters, as well as the use of advanced graphics hardware features to implement the lighting model itself. A brief look at the new features of graphics accelerators is followed by a description of how this flexibility is used to implement the Lafortune reflectance model used in the application. Much of this explanation was originally written for a course project, but has been expanded and revised to reflect the new version of the software which incorporates the calibration system and head tracking. A method to speed up this rendering by avoiding frame buffer read-backs has also been included, and is in Appendix D.

Finally, in Chapter 6, possible extensions and improvements to the calibration process are discussed, as well as other applications and areas of future work.

Chapter 2

Designing a Calibration System

2.1 Calibration requirements

The design goals for developing the calibration system described by this thesis are broader than those of most similar systems. The reason for this is that the intention was to design a calibration method which worked under a variety of configurations. This is necessary for the system to be useful in the Imager Graphics Lab, where we develop software on many different platforms. In the lab we have several single-monitor Linux machines, stereo-capable IRIX machines, and an immersive projection display consisting of three 8-foot high screens powered by an SGI Onyx. In addition to this in-house panoply of hardware, we have off-site access to a four-screen stereo-capable CAVE [cruz92, cruz93]. Typically, dealing with these different set-ups requires extensive modifications to the VR software. Because there has been no uniform way of doing this, most software is changed as needed, and measurements for screen parameters are hard-coded into the software itself. The result is that things are rarely calibrated correctly for the different environments, and that calibration intensive techniques, such as head-coupling, are generally avoided.

Our calibration system, then, must not only provide a method to calibrate with sufficient accuracy to allow head-coupling and realistic rendering, but also be simple and convenient enough to promote its use. There are a number of design

factors that help achieve this goal.

Minimising interdependence between calibration steps makes the calibration more convenient. A calibration system that must be performed in its entirety for every new user is inefficient and frustrating. Changes to one aspect of the environment should require only a re-calibration of that aspect. Good modular design will help ensure that this goal is met.

The calibration process should strive to be transparent. There should be no mystery as to what the calibration system is doing and the values it returns should make sense to the experimenter. This is especially important to allow the experimenter to judge whether the calibration is reasonable or not and avoid obvious errors. For that reason, all the data in our calibration system should be stored in commented ASCII text files. These plainly show the values computed by the calibration, and can be easily edited manually if the experimenter knows the dimensions of certain aspects of the environment, or simply wishes to observe the effects of perturbing the calibration. Further, the calibration software should return check values and helpful indicators of accuracy wherever possible.

The final design goal is to make per-user calibrations fast and simple. It is always good system design to streamline tasks that are performed frequently, and in virtual reality nothing changes as often as the viewer, or, more specifically, the position of the sensor on a viewer's head. Because the user's eye positions must be located in relation to this sensor each time the system is used, this calibration should be as fast and convenient as possible. The calibration of other attributes that change less often, such as monitor positions, can be longer and more rigorous.

2.2 Locating points in 3D

Achieving our goal of quick and easy calibration depends greatly on the method we choose to locate points in 3D. The act of locating points in 3D is at the heart of any calibration system, and the method we choose must be both accurate enough to

provide useful data, as well as convenient enough to be practical when calibrating many points. Finding such a balance is difficult, because greater accuracy usually results in less convenience.

Measuring the location of points can be done directly, by touching a sensor to the location we wish to measure, or indirectly, by using other measurements and known correspondences to calculate the position of an unknown point. The tracking system we use most often in the lab is the Polhemus FASTRAK, which is affected by the magnetic fields near the surface of CRT monitors and even LCD screens. Though magnetic interference only applies to magnetic tracking systems like the FASTRAK, CRT monitors are still problematic because the thick glass in front of the lit phosphor makes it impossible to place a tracker directly on the point we see lit. Regardless of the display technology we use, however, we still have the problem of locating the optical centre of a user's eyes. Because we cannot place a sensor inside the eye, accurately measuring the position of the eyes must be done indirectly, though it is not uncommon to measure a point just in front of the eye, and then treat this point as the location of the eye itself.

Of course, because our system endeavours to allow calibration using any of the forementioned display technologies, and because we are indeed using a magnetic tracking system, our method of locating points must be an indirect one. Before we look at the method used to locate points in 3D, it helps to understand a little more about the tracking system we are using.

2.2.1 The Polhemus FASTRAK

The Polhemus FASTRAK [polh00] is a tracking system which returns six degree-of-freedom data for up to four attached sensors. The system consists of a transmitter, which emits a magnetic field, and sensors that detect the strength and orientation of the field to determine their position and orientation. Both transmitter and sensors are pictured in Figure 2.1. Each second, 120 position and orientation records are

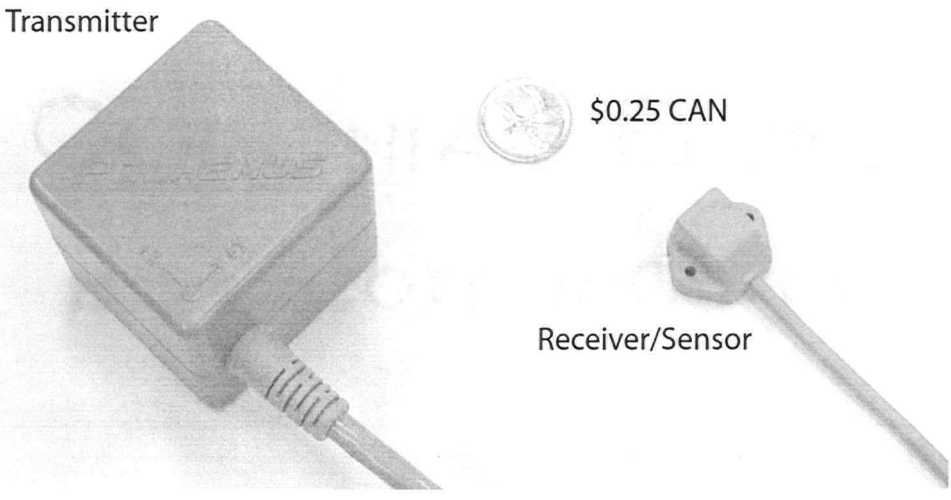

Figure 2.1: Polhemus FASTRAK transmitter and sensor.

sent to the computer, so the latency of the system is dependent on the total number of sensors attached to the system.

The design of the sensors holds particular interest for our application, as will soon become apparent. Each sensor has two small screw holes, with a diameter of 0.292 cm and a length of 0.635 cm. These holes are precisely located on each sensor at an exact distance from the sensor's magnetic centre. Because the holes are very precisely located in relation to the reported position and orientation of the sensor, they can be used to calibrate other points. The next section will describe a way of using these screw holes to locate other points in 3D. Though the method does not rely on the particular type of tracking system, or the construction of the sensors, the fact that these sensors have small holes in them will prove convenient.

2.2.2 "Sighting" points

Our method for locating points in 3D is based on the fact that the intersection of any two non-parallel, intersecting lines is a point. The idea is to generate two lines in 3D which intersect at the target point we wish to calibrate, thus allowing us to determine the position of the target point without the need to place a sensor directly

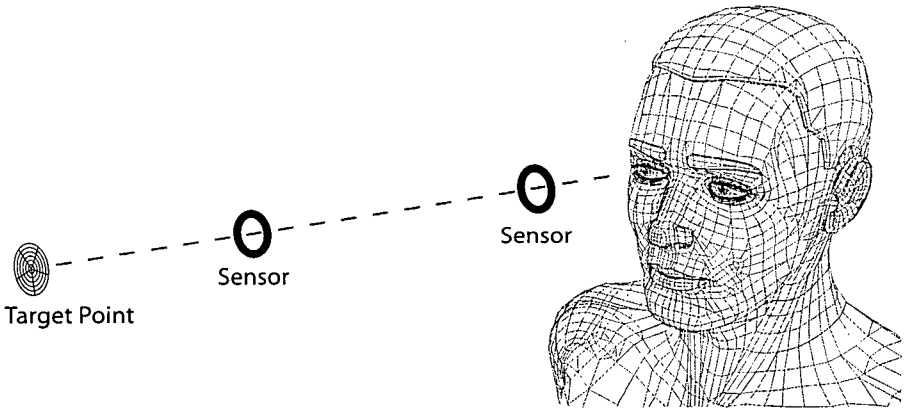

Figure 2.2: Sighting a target.

on it.

In order to generate these lines, we will use a technique we will refer to as "sighting". A user looks through two small holes, or sights, which are rigidly attached to sensors. By lining up the target point in these sights, we can use the sensor data to measure the exact location of the holes, and therefore the line that passes through the target point, as shown in Figure 2.2. By doing this twice, from different positions, we obtain two different lines in 3D that should intersect at the target point. In practice, however, the lines will not intersect due to the fact that there are small errors in aiming and measuring their positions. Instead, we must compute the point at which they come closest to intersecting.

To do this, we first calculate the shortest line segment connecting the two lines, and then treat its midpoint as the near-intersection of the lines. To find the shortest line segment, we use the method described by Bourke [bour98]. In essence, by making use of the fact that the shortest line segment connecting the two lines will be perpendicular to both lines, we know the dot product of a vector pointing along the line segment and a vector pointing along a line will be zero. By using this equation for each line and then expanding the expressions in terms of their x, y and z components, we arrive at equations that can be solved for the endpoints of

the line segment. Once we have found the endpoints of the line segment, averaging gives us its centre, which we use as the effective intersection of the two lines in 3D.

To improve our estimate, we can also take into account more than two line samples. To do this, we compute the intersection of every pair of lines and then average our intersection points to produce a final approximation of the target point.

This method of sighting points allows us to keep the sensors away from the magnetic interference of the monitor, while still obtaining accurate measurements of points on the screen. Also, by restricting the distance we need to move the sensors from the magnetic field generator used by the sensing system, we improve the accuracy we get from the tracker. The biggest advantage of the sighting method, however, is the ease with which we can use it to find the centre of a user's eye.

2.2.3 Sensitivity to measurement errors

One aspect of the "sighting" method which is difficult to contend with is that its sensitivity to measurement errors varies depending on the samples collected. To illustrate this point, we will look at the simple case of two line samples, one of which is measured flawlessly and passes through the target point, the other of which has some displacement error ε. If the line samples were collected so that the angle between them (θ) is close to 90 degrees, then the resulting error (δ) in locating the target point is also very near ε. However, if the angle between the line samples is much less than 90 degrees, small errors in one line cause the intersection of the lines to vary a great deal. The error in the target point can be expressed by:

$$\delta = \frac{\varepsilon}{\sin\theta}$$

This means, given that users make small errors in the measurement of the line samples, that the results of our system will be much better if a user collects line samples that are at right angles to one another, and avoids collecting samples that are nearly parallel to one another.

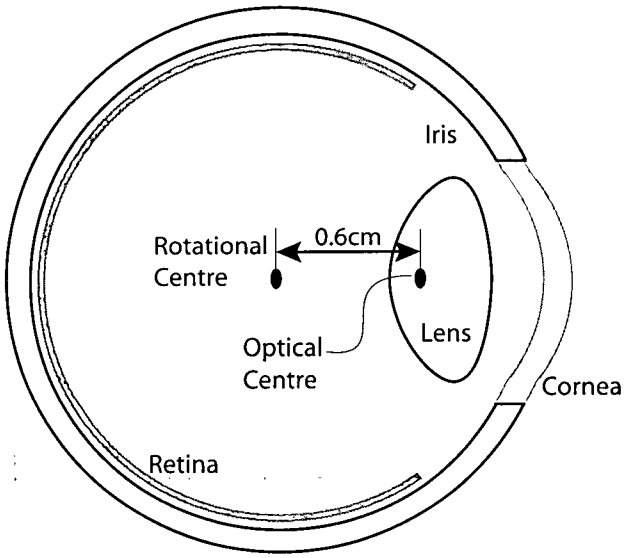

Figure 2.3: Rotational and optical centres of the eye.

2.3 Locating a user's eyes

To understand the great advantage our method provides in precisely locating a user's eyes, we must first understand the problems involved in trying to make this measurement. Deering [deer92] points out that the point we use when we render an image in computer graphics is equivalent to the first nodal point of the eye's lens system, also known as the "optical centre" of the eye. This point, however, is not the same as the rotational centre of the eye, and on average lies approximately 0.6 cm in front of it. Because there is a distance between the rotational centre and the optical centre, the optical centre moves as the eye changes gaze direction. Though it may move, its position is predictable: it always lies in the direction of the gaze from the rotational centre. What this means is that the optical centre, which is required for accurate rendering, can be found if we know the position of the rotational centre and the gaze direction.

Fortunately our sighting technique is well suited to the task of accurately determining the rotational centre of the eye. When we look through the two sights

while performing the sighting technique, our optical centre and rotational centre are by necessity lined up with those sights. It doesn't matter what we see through the sights; as long as we can see through the centre of both, the line connecting the sights will trace directly through the rotational centre of our eye. By placing a sensor rigidly on one's head, to keep track of changes in our head position, and then taking two line samples using different gaze directions, we will have two lines that intersect at the rotational centre of the eye. We can now find their intersection as described before, in Section 2.2.2, and thereby accurately measure the true rotational centre of the eye.

This represents a huge advantage over traditional calibration methods, which simply try and approximate this by measuring inter-pupillary distance. Traditional methods have no way of identifying the rotational centre of the eye, and cannot take measurements inside the eye in any event.

Using sighting is also much more convenient than traditional methods. Because the sights can be rigidly affixed on a tripod next to the viewing screen, and do not need to be moved (as the movement of the user's head is all that's required to generate the different viewing directions), the calibration can be done very quickly and without the need of a second person to assist. If the sights used are already rigidly affixed in front of the user, the four samples necessary to find the rotational centre of both eyes can easily be taken in less than ten seconds.

Even if the sensors are not mounted on a tripod, calibrating eye position using this method is not difficult. After a short demonstration of how to perform the calibration, new users of the system did not take more than a minute to correctly calibrate their eye positions, despite having to hold the sensors themselves. As one would expect, this time improves as the user gains experience performing the calibration.

Of course, because we are using the screw holes of the Polhemus FASTRAK sensors as our sights, we know their position in relation to the reported sensor

positions without having to calibrate them separately. It is also possible to use only one sensor to perform the calibration by rigidly attaching it to a sighting device, such as a scope or gun-sight, which is then used to generate the line samples to locate the target point. This has the advantage of lowering the latency of reading from the FASTRAK, because fewer sensors need to be polled for data, as well as providing a more intuitive aiming device than the screw holes of the FASTRAK sensors. For this method to be useful, however, the aiming device must be carefully measured and the position and orientation of the FASTRAK sensor mounted on it must be known exactly. Otherwise, errors present in the calculation of the position of the device will adversely affect the accuracy of the system.

Once the rotational centre of the eye is located, we can approximate the position of the optical centre. To do so, we need to assume a gaze direction. Usually we assume the user is looking in the direction of the centre of the screen, which in many cases is a good approximation, though in some systems, with a user controlled pointer, assuming the user is looking at the pointer will give better results [deer92]. Once we have an assumed gaze direction, it is a simple matter to move 0.6 cm from the rotational centre in that direction.

Further details on calibrating a user's eye positions will be provided in Section 3.5; first we look at how our sighting method and ideas about calibration design combine to produce a complete working calibration method.

This page intentionally left blank.

Chapter 3

Calibration Method

Keeping in mind the requirements and design goals we outlined for our system in Section 2.1, we will now look at how the final calibration system fulfils these requirements and allows the convenient, flexible calibration of a variety of head-coupled VR systems.

3.1 Overview of the calibration procedure

In essence, a calibration system is simply a way of determining a set of coordinate transformations. Each transformation tells us how to move from one coordinate system, or "space", to another. In head-coupled VR we are concerned with knowing how to move between six different coordinate systems:

Frame buffer coordinates This space is a 2D grid of computer memory that is used to hold the picture data seen on all attached screens. In multi-screen systems, rectangular sections of the frame buffer are displayed on different screens.

Screen space This is a 2D space that exists for each display screen in the environment, and represents the visible surface of the screen. It is 2D because each screen is considered to be planar, but it is understood that these 2D

spaces have associations with both the 3D world (where the screens are in fact positioned and oriented) as well as with frame buffer coordinates.

Tracker/World space This space is the native 3D coordinate system of whatever tracking system we are using. The raw numbers that the tracking system returns as the positions and orientations of its sensors are said to be in tracker space. Because it exists in the physical world, we will use it as our real-world physical space, though in systems with multiple tracking systems, often there is a separate standardised world space to which all the tracking systems are calibrated.

Head space This is the 3D coordinate system which originates at the tracking sensor placed on the user's head. The user's eyes do not move in head space, and so once located are considered fixed points. Note that head space does not need to be calibrated with respect to tracker space because it is defined in terms of the position and orientation of a tracking sensor, which is already in tracker space.

Eye space For the sake of completeness we can treat each eye as being its own coordinate system. In practice, it is usually more convenient to treat eyes as points in head space, though generalising eye-position as a separate space can be useful for discussing calibration in terms of coordinate transforms. When treated as a separate coordinate system, the orientation of this space is unimportant. This is because the optical centre of the eye has no associated orientation, and only a position.

Virtual space Virtual space is the 3D coordinate system of the virtual world. For head-coupled VR, after calibration it should line up with real-world space (in our case, tracker space) to provide the illusion that virtual objects exist and can be viewed as real objects.

It can be said that we have a complete calibration when we can move from any one coordinate system to any other, and thereby perform every coordinate transformation required to render the images on the screens in the correct locations.

Though looking at coordinate systems is a good way to visualise the calibration process, the steps required to calibrate the system are actually quite intuitive. Two obvious calibration steps are locating the eyes of the user with respect to a head-mounted sensor and locating the screens with respect to the tracking system. Knowing these two things lets us determine the direction from which a user is viewing the screen, so that we can compensate for his off-axis perspective. This is not enough, however, to provide a useful display of a virtual scene. Two more critical pieces of information are missing.

The first missing piece of information is the mapping from the computer's frame buffer to the screen space of different screens. In most multi-screen set-ups, each screen is mapped to a rectangular section of frame buffer and, in order to display the correct image on a screen, we must know which section of frame buffer is shown on each screen.

The second missing piece of information is the location of our virtual world in relation to the physical world. In other words, we need to know the relationship of virtual space to world space. This information is required in order to give each screen and the user a sensible initial position in the virtual world. Without this extra transformation to register tracker space with virtual space, changing the locations of the screens would always change the view of the virtual world shown on them. This is inappropriate for most head-coupled VR applications, where frequently users prefer that a screen in front of them show them a "front" view, aligned with the virtual horizon. For this to happen, there must be a way to specify the default position and orientation.

This gives us four pieces of calibration information we need to know in order to allow head-coupled VR: the location of screens in the frame buffer, the location

of screens in 3D tracker space, the default position and orientation of the virtual world, and the position of the user's eyes. These pieces of information correspond to the coordinate transformations from frame buffer coordinates to screen space, tracker space to screen space, tracker space to virtual space, and eye space to head space, respectively. Because the transformation from head space to tracker space is given by the position and orientation of the sensor on the user's head, the four pieces of information listed above are all that's required to provide us with coordinate transformations between all six spaces. If our system can collect all of this information, it will have the complete set of coordinate transformations needed to provide a head-coupled VR display.

Each piece of information will be collected in a separate step. This provides the advantage of allowing parts of the calibration to be run independently of other parts. Also, errors made while running a calibration can be resolved by running the most recent step again, as opposed to having to throw out an entire calibration and begin from scratch. To understand to what extent we can re-run separate parts of the calibration, we will first look at what is involved in each step, and then examine the dependencies between the steps themselves.

3.2 Step 1: Mapping the frame buffer to physical screens

The first step of our calibration is to determine the number of screens, and then find out what section of frame buffer maps to each of those screens.

To start the calibration, the experimenter runs the 2D screen calibration program from the command line, specifying how many screens are attached to the system. The program opens a window to the full size of the available frame buffer, and proceeds to lead the experimenter through the process of finding the boundaries of each screen. This is done by providing a horizontal and vertical guide line, which can be positioned by clicking with the mouse, or with the cursor keys of the keyboard. These guide lines make it easy to see when the user has found the edge of a screen.

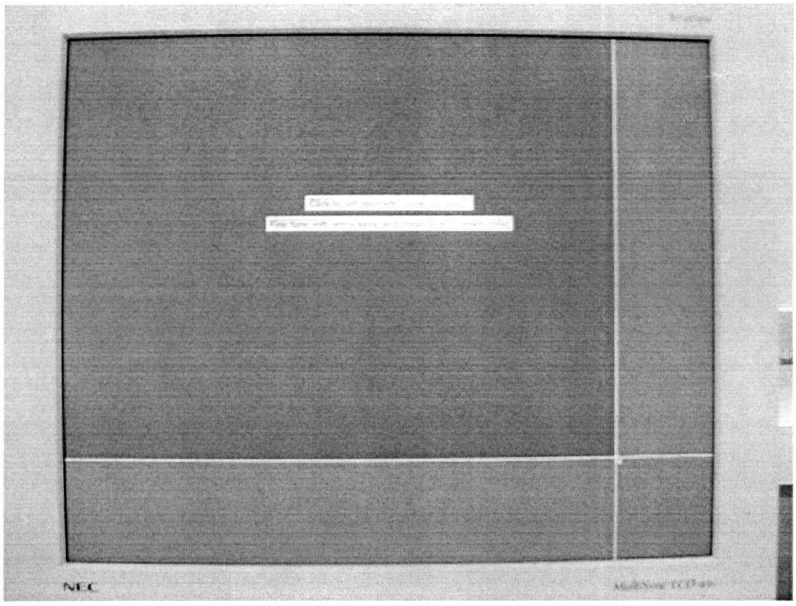

Figure 3.1: Part way through the 2D screen calibration.

After the last screen has been located, the areas of frame buffer used for each screen are recorded in an output file and the program exits. Figure 3.1 shows the screen after the first corner of a screen has been selected. This file now contains the information required to display an image on any screen that takes its image from the frame buffer.

Though in theory this mapping could be determined by the software itself, without user intervention, this is not practical for a number of reasons. The way to query the operating system for this information is not portable across different platforms, and so the software would have to be rewritten for each new environment on which it was run. Also, by forcing the user to specify what area of the frame buffer is visible on each screen we guarantee that the system is correctly calibrated even if parts of the frame buffer are not displayed, as is the case around the edges of some CRT monitors or on projection displays that project images larger than their screens.

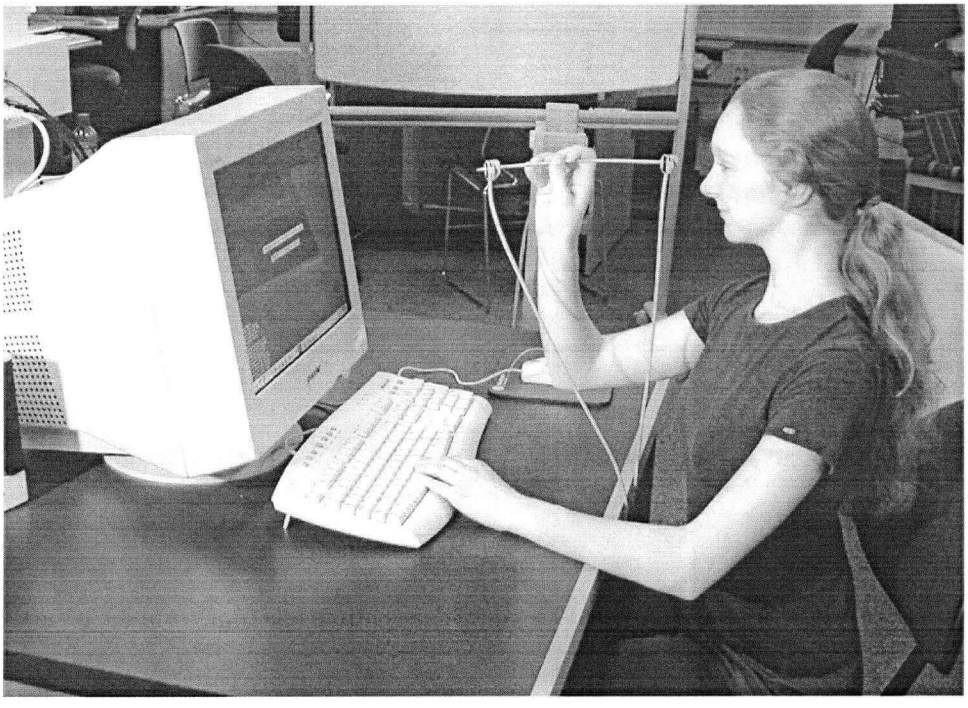

Figure 3.2: A user performs the 3D screen calibration.

3.3 Step 2: Finding 3D screen positions

Now that we have identified which parts of the frame buffer map onto visible screens, we can proceed to locate these screens in tracker space. This means finding where, physically, the monitors are positioned relative to our tracking system. The method used to do this is to display points on a screen and then have a user determine their location by sighting them, as described in Section 2.2.2. Figure 3.2 shows a user performing this step.

First, the frame buffer data from step one is read to determine the position of the screens to calibrate. For each of the screen areas in the frame buffer, a point is displayed near each corner of the screen. The reason for not choosing the corners themselves is that many display devices have non-linear distortions very near their corners. These distortions would risk skewing our calibration, and are avoided by choosing points slightly towards the centre of the display. The sighting method is

used to obtain 3D line samples that nearly intersect each of the on-screen points. By analysing these line samples, we find the best approximation of the 3D location of the point on the screen. After all the points are collected, they are then scaled outwards from their centre (their average) so that their position now corresponds with the true corners of the screen. We now have good approximate 3D positions for each of the four corners of a screen.

In order for the positions we collect to be valid screen dimensions, they must be both planar and lie in a rectangle. However, because of the nature of measuring points, there will be small errors which mean that this is not exactly the case. For that reason, each set of four points undergoes a procedure to perturb the points slightly so that they do indeed form a planar rectangle.

Let's suppose we have four measured corner points, numbered clockwise from top left, called P_0, P_1, P_2, and P_3. We first calculate the horizontal (X_0, X_1) and vertical (Y_0, Y_1) axes of the screen, using $X_0 = \frac{P_0+P_3}{2}$, $X_1 = \frac{P_1+P_2}{2}$, $Y_0 = \frac{P_2+P_3}{2}$, and $Y_1 = \frac{P_0+P_1}{2}$. These two axes are guaranteed to intersect at the screen centre, $C = \frac{P_0+P_1+P_2+P_3}{4}$. Now, the angle θ between the screen axes can be found by $\theta = \arccos\left(\frac{(X_1-C)\cdot(Y_1-C)}{\|X_1-C\|\,\|Y_1-C\|}\right)$, and the screen normal is found by $\mathbf{n} = \frac{(X_1-C)\times(Y_1-C)}{\|X_1-C\|\,\|Y_1-C\|}$, where $\mathbf{n}$ is the normal vector and "$\times$" is the vector cross-product operator. Next, two correction angles are computed. The first, $\varepsilon_1 = \frac{\theta-90}{2}$ is the correction angle for the x-axis and $\varepsilon_2 = -\varepsilon_1 = \frac{90-\theta}{2}$ is the correction angle for y-axis. The points X_0 and X_1 are rotated around the normal by ε_1 and the points Y_0 and Y_1 are rotated around the normal by ε_2. The axes are now guaranteed to be orthogonal, and a new set of corner estimates, Q_0, Q_1, Q_2, Q_3, can be found by $d_x = X_1 - C$, $d_y = Y_1 - C$, $Q_0 = C - d_x + d_y$, $Q_1 = C + d_x + d_y$, $Q_2 = C - d_x - d_y$, and $Q_3 = C + d_x - d_y$. The points Q_0, Q_1, Q_2 and Q_3 are now guaranteed to be planar and rectangular.

To help the experimenter determine whether the values obtained for the screen positions are very far from correct, the values calculated during this procedure for the angles between the screen axes are displayed, as well as the total distance

the points are moved from their initial positions. This can give an experimenter an idea of whether a mistake was made while taking the original line samples, and can allow him to determine whether the calibration step should be repeated.

The result is that we obtain a mapping, for each screen, from 2D frame buffer coordinates to 3D tracking system coordinates, which is saved to a data file. For information about the format of the data files used by the system, see Appendix C.

3.4 Step 3: Setting the origin of the virtual world

Step three allows an origin position and orientation to be specified by the experimenter to indicate where the origin of the virtual coordinate system should lie in relation to the tracker coordinate system. This can be set in different ways, depending on the needs of the experimenter.

The first, and most useful, way to specify the origin is in relation to a screen. The experimenter can specify a screen that should be considered the "front" screen. The origin calibration program will then compute the centre of that screen, in tracker space, and set this to be the origin of the virtual coordinate system. Also, the orientation of the virtual world will be set to correspond with the x-axis and y-axis of the screen.

The second method supported by the system for setting the default position and orientation is to make the virtual origin correspond with the position of a given sensor. The sensor's position and orientation are read, and the virtual coordinate system is made to originate from that spot.

The final method supported by the system is to allow the user to specify the values for position and orientation manually. The position is specified in centimetres from the origin of the tracking system, and the orientation is specified in Euler angles.

If this step is omitted, then the system defaults to having both virtual space and tracker space share the same origin and orientation.

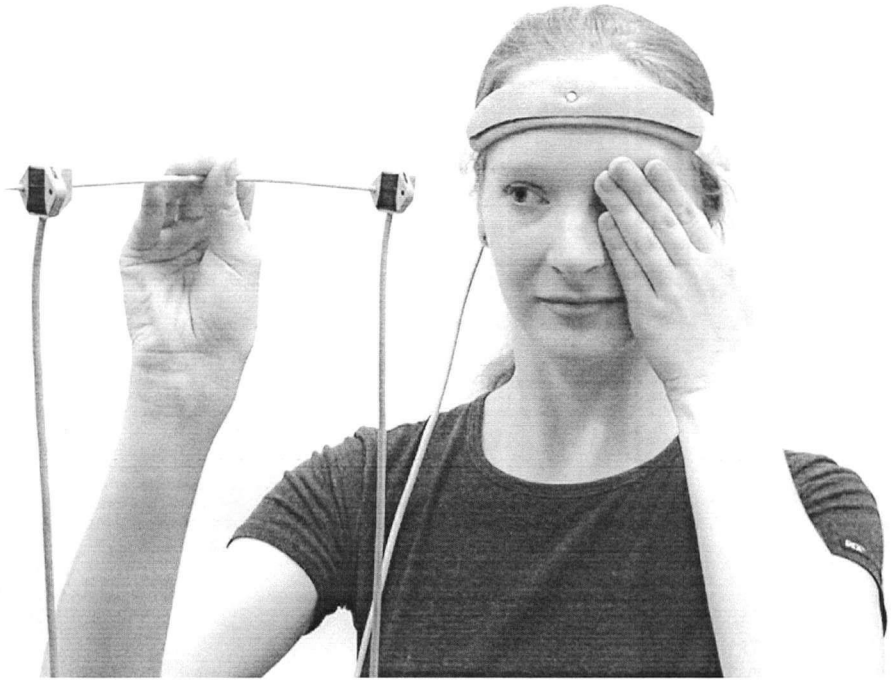

Figure 3.3: A user calibrates her eye positions.

3.5 Step 4: Accurately locating a user's eyes

The final step in our calibration is to locate a user's eyes with respect to a given sensor, which is assumed to be attached to a user's head. This is done, as described in Section 2.3, by having a user perform the sighting method without targeting any specific point. Instead, it is only important that his eye be looking in different directions (with respect to his head position) for each line sample. This allows us to compute the rotational centre of his eye. Figure 3.3 shows a user performing this step, and we can clearly see the sensor used to track their head position.

Because this is the last step in the calibration, and must be performed every time either a new user wishes to use the system or an existing user changes the position of the sensor on his head, this step is performed very often. To simplify performing this step, it is useful to mount the sensors on a stand so that the user does not need to hold them every time the calibration needs to be performed and

instead can simply put his eye up to one sensor's screw hole to look through it to other sensor's screw hole. Notice that because the sensors report their own positions and orientations, it doesn't matter if the exact shape or size of the stand they are mounted on is not known. As long as the user can see through both screw holes, the calibration will be correct, and does not depend on the construction of the stand.

In order to help the experimenter determine if the calibration step was accurately performed, the inter-ocular distance of the user is computed and displayed. If this value is around 6 cm, then it is likely that the calibration was successful, and we have successfully computed the location of the rotational centres of the user's eyes.

3.6 Increasing accuracy

The first step of the calibration, where the experimenter locates each screen in the frame buffer, should always be accurate because the interface provided by the calibration program makes it is easy to see, to the pixel, how much of the frame buffer is contained on a given screen. Also, in step three, where we specify the origin of the virtual coordinate system, the origin is either chosen by the experimenter or calculated exactly by the software based on screen positions. Here again, accuracy is not an issue.

It is in the steps involving the measurement of 3D positions, using the sighting method, that we must consider the accuracy of the system. In these cases, accuracy is influenced by two things: the accuracy of the tracking system, and the care with which the line sample measurements are performed.

The accuracy of the Polhemus FASTRAK is 0.08 cm RMS for the x, y, or z sensor position as long as it is used within 76 cm of the magnetic field transmitter. Because the FASTRAK is a magnetic tracker, this accuracy diminishes the farther from the transmitter the sensors are positioned, or if the sensors are located near metal or magnetic objects.

In order to help reduce the random errors present in the measurements of sensor position, the calibration software uses a low-pass filter to remove some of the high frequency noise that is present in the positional information. This software solution is used instead of the hardware supported filtering that is built into the FASTRAK itself. The filter mechanism is implemented as an abstract class, and can easily be replaced by other filtering schemes in the future, if the need should arise. One possible improvement would be to use the more robust Kalman filter [kalm61, brow83, lian91]. For now, we average the most recent three positional values of the centre of a sighting hole to determine the value to record as the actual position. Because we average only three samples, we obtain a latency of 0.05 seconds while calibrating the screen positions, and a latency of 0.075 seconds while calibrating the eye positions (the extra sensor required to keep track of the head lowers the rate at which we can collect samples from the FASTRAK). These latencies should not adversely affect the calibration, especially when we consider the way in which a calibration is performed: first the sensors are lined up, and then a sample is taken. The interval between when an experimenter decides that the sensors are correctly aligned and when he records a line sample should always exceed 0.075 seconds, so by the time he presses the key to record the results of the measurement, the averaging of three data samples will have already taken place.

The second factor which affects accuracy is how much care is taken while gathering the line samples used in the sighting method. It is much easier to ensure that the sensors are correctly lined up with their targets if they are rigidly attached to a stand, rather than hand held, as they can be placed and then checked to make sure they line up precisely with the target point. Also, the individual performing the calibration should try to ensure that he is looking through the centre of each screw hole, and that the target point falls in the centre of both screw holes. For a better idea of how the system performs, refer to the tables in Appendix E which compare values recorded by the system for the same set of points calibrated in

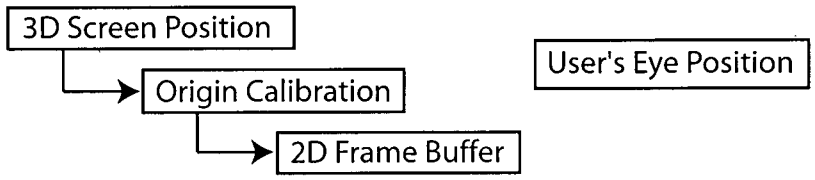

Figure 3.4: Calibration step interdependencies.

several consecutive runs.

In general, locating the position of the screens is more difficult, and therefore more prone to error, than locating the user's eyes. If working with a system where the display screens are fixed and built to a known specification, the experimenter may wish to consider manually entering the values for the known positions of the screens into the calibration file. The rest of the calibration can still be run unchanged, but will benefit from the precision of the known screen positions.

3.7 Dependence between calibration steps

A great advantage of a modular calibration design is the ability to run parts of the calibration separately from others, so that changes in one part of the system need not require re-calibration of the entire system. To know what changes we can make without affecting other parts of the system, however, we have to understand the dependencies between different parts of the calibration system, shown in Figure 3.4.

The first step of the calibration, the mapping of the frame buffer to the appropriate screens, will be correct as long as we do not change the fundamental system configuration, either by changing the number of screens present or by changing any of the screen resolutions.

The second step, locating the screens in tracker space, depends upon the first step in order to determine both the number of screens as well as how to display points on them to allow the experimenter to calibrate them. For this reason, step two will always have to be performed again if step one has been redone.

The third step, that of setting the origin of the virtual coordinate system, is dependent on the second step. This is true not only because of the fact that the origin is frequently chosen based on the position of a screen from step two, but also because of an implementation decision to use the same configuration file to hold the origin information as well as the 3D screen positions. Though this goes against our general principle of minimising dependencies, the fact that this step takes no time to perform, and is computed immediately by running the `calibrateOrigin` program, means that in practice having this step depend on prior steps ensures that old origin-positions are not reused accidentally.

The fourth step, that of finding the location of the user's eyes with respect to the head-mounted sensor, is not dependent on any other step of the calibration, and no other step in the calibration is dependent on it. This means it can be re-run at any time, without affecting any other step of the calibration.

Another way to perform the calibration of the screens in step two would be to first locate the user's eyes, and then sight points on the screen by looking through only one sensor, using the position of the eye as the other point from which to determine our line sample. That is, we would be drawing lines from the centre of the user's eye through the screw hole of a sensor in order to locate the points on the screen. While this method would make it easier for the experimenter to line up the sensor with the target location on the screen, it would propagate any errors from the location of the eye through to the calibration of the screens. Also, it would mean that step two would be dependent on having a valid calibration of a user's eyes, which introduces needless complexity to the system. For that reason, the method that uses two sensors to perform the sighting is preferred.

This page intentionally left blank.

Chapter 4

Producing the Final Image

A calibration system, however accurate and convenient, is of no use unless it can be easily integrated into software applications. To make this integration easy, libraries were developed to read in the calibration data and to use it to render the correct off-axis perspective image to the correct screen. Though it is assumed that the application will handle reading from the tracking system, the libraries do provide functions to help compute the correct optical centre of a user's eyes, given their rotational centres and a gaze point.

While detailed information on developing software with these libraries is provided in Appendix B, this section will describe how some of the functions necessary for the production of images in head-coupled VR are implemented.

4.1 The role of the application

Once all the calibration data have been saved to files by the calibration system, it is up to the application to make use of these data to implement a head-coupled VR system. To do this, the application must perform two essential tasks: reading from the tracking system and rendering the virtual world to the screens. A brief description of how this is done will give an idea of how the work is divided between the application and the provided utility libraries.

For the Polhemus FASTRAK, reading from the tracker can be performed with the aid of the driver developed as part of my undergraduate thesis [stev00]. This driver can be polled each display cycle to obtain the latest position and orientation of the head-mounted sensor. Once this information is obtained, it is sent to the calibration utility library and the eyes' rotational centres are automatically calculated. The application can then supply a 3D point to use as a guess for where the user is looking, so that the optical centres of the eyes can be calculated. All the calculations required to approximate the true eye-point are performed by functions in the utility library. The application need only provide the latest sensor position and gaze point.

Once the eyes are accurately located, the application specifies the screen number to which it wishes to render. The utility library then automatically sets the viewport to render into the portion of the frame buffer associated with that screen. The library also computes the required off-axis projection to compensate for the user's viewing position in relation to the specified screen, and configures OpenGL to use this matrix when rendering. These calls replace the usual OpenGL calls to `gluPerspective` and `glViewport`. The application can then proceed to issue commands to render as normal, and everything will be drawn correctly to the selected screen. This step is repeated for each screen attached to the system.

Other than these two steps, locating the eyes and choosing which screen should be rendered, the application performs exactly as its non-head-coupled counterpart. By automating and abstracting the math required to calculate the off-axis projections, all of the extra difficulty is removed from developing applications that use a head-coupled VR display.

4.2 Other applications

Though the calibration system utility libraries were designed with head-coupled VR in mind, there are some applications that prefer other types of projection. These

applications are also possible using the calibration system, though it is then up to the application to generate the desired projection.

For example, some systems use the "Delft Virtual Window System" [smet87, gave95] method of displaying the virtual world. In these systems, instead of computing the correct off-axis projection, a normal on-axis viewing perspective is computed taking into account the current viewer position and assuming that the viewing direction is towards the centre of the screen. Though this type of projection makes it impossible to do correct stereoscopic display, and prevents the virtual world from being registered correctly with the real world, the images produced by this type of display do not appear distorted when viewed from perspectives other than the correct head-coupled perspective. This means that multiple users can share this type of display more easily than with true head-coupled VR. In essence, the movement of the viewer's head simply pans around the virtual scene, which is shown on screen as in normal 3D computer graphics.

This type of display can be easily implemented with the help of the calibration system. The screen position and eye positions would still be obtained from the calibration system, but the projection can now be performed by the regular OpenGL `gluPerspective` call.

There are many other possible non-traditional uses for the calibration system as well. Multi-screen displays can use the system to determine the relative positions of different screens, to allow a user to drag a window intuitively from one screen to another, for example. Other systems might provide interaction based on the position of a stylus or 3D sensor in front of the screen.

Though other uses of the system are certainly valid and useful, the vast majority of applications will make use of the calibration system to provide a head-coupled VR perspective of a virtual world. For this reason, the rest of this chapter focuses on how the calibration system works with OpenGL to produce correct off-axis projections for a given viewer position and a set of calibrated screen locations.

4.3 Configuring OpenGL

The utility libraries provided with the calibration system replace the traditional OpenGL functions used to set-up projection matrices and viewports for rendering. Instead of having to use these functions directly, the utility library automates the process of figuring out the correct projection matrix for a given screen, sets the viewport so that the correct portion of the frame buffer is used, and automatically translates and rotates the virtual world to correspond with the calibrated origin.

In a typical 3D OpenGL application, two functions are used to control how OpenGL displays the virtual world on the screen: `glViewport` and `gluPerspective`. `glViewport` specifies an area of the frame buffer to use for rendering. This function controls where on-screen the image will be displayed. `gluPerspective` computes an axis-aligned perspective projection matrix, given an eye position, up-vector and look-at direction. This function controls what 3D geometry is rendered on screen, and how it looks. When we move to a head-coupled environment, we must now contend with the added complexity of a changing viewpoint, multiple screens, and a different virtual origin. The `Projector` class of our utility libraries provides this added functionality in two simple commands: `glSetViewport` and `glSetOffAxisView`.

The `glSetViewport` function performs the same task as the `glViewport` function in OpenGL, but instead takes only one parameter: the screen number to which we wish to render. The `glSetViewport` function automatically sets the viewport so that the image is drawn to the specified screen.

The `glSetOffAxisView` function also takes the current screen number as a parameter and uses the user's current eye position as well as the calibrated screen position to compute the correct off-axis projection matrix required to display the correct image as seen from the user's viewpoint.

Thus, instead of the typical command sequence:

```
glViewport(x, y, width, height);
```

```
gluPerspective(fovy, aspect, near, far);
drawObjects();
```

We instead have the commands:

```
for (i=0; i < NUM_SCREENS; i++) {
  projector->glSetViewport(i);
  projector->glSetOffAxisView(eyePos, i, near, far);
  drawObjects();
}
```

The loop is used to ensure that we draw on each screen in an environment. Otherwise, the second code fragment looks very similar to the first, and will therefore be intuitive to developers used to using OpenGL.

To appreciate how much effort the developer has been spared, and to better understand what is involved in computing an image in head-coupled VR, we will now examine the process of calculating off-axis projection matrices. For further details on developing software using the utility libraries, see Appendix B.

4.4 Generating correct off-axis projections

Computing the off-axis projections used to show the viewer a realistic image from his true eye position is a critical part of head-coupled VR. Unfortunately, the complexity of this step discourages many people who would otherwise want to use head-coupling. Despite the fact that it has been already implemented in the utility library, some will still wish to know how this step is accomplished. This section will explain how the arbitrarily oriented off-axis projections needed in head-coupled VR are calculated using the help of some standard OpenGL functions.

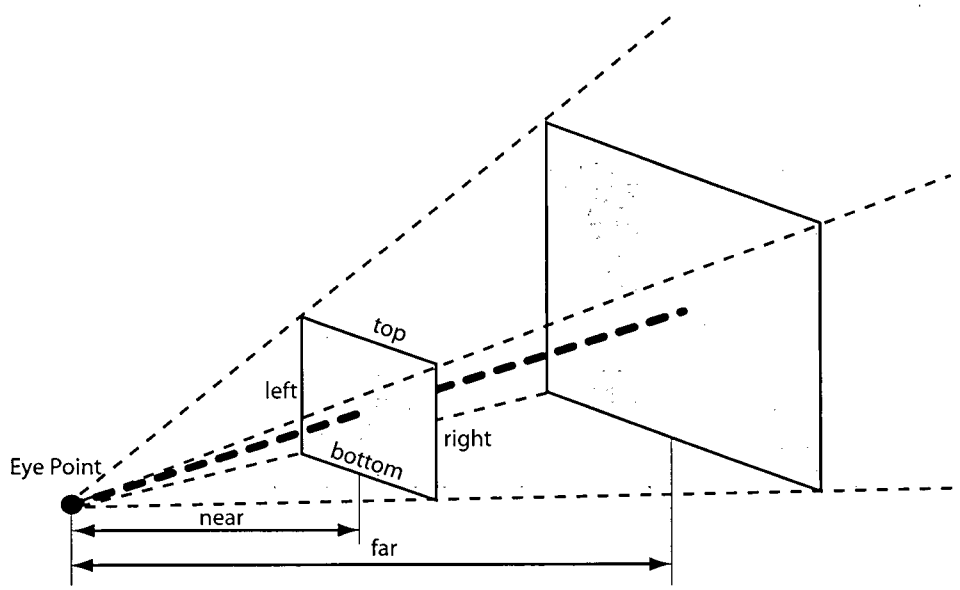

Figure 4.1: The viewing frustum.

4.4.1 Introduction to projections

The goal of the projection stage is to take a volume of 3D space and flatten the objects within it onto a plane, which corresponds to the screen. This volume is determined by the position of several "clipping planes" that discard, or "clip", any objects to one side of them. All the objects that lie within the volume laid out by the clipping planes are then flattened onto the 2D plane (screen). For perspective projections, six clipping planes are used, as shown in Figure 4.1. Four of these form the top, bottom, left and right sides of a pyramidal volume. Two more, called the near and far clipping planes, restrict the depth of the volume. This enclosed volume of space determines what is seen on screen and is called the "viewing frustum".

In normal 3D computer graphics two assumptions are made that simplify specifying the viewing frustum: the eye-point is always assumed to lie on a line perpendicular to the centre of the frustum, and the screen's orientation is aligned with the viewer's orientation. Figure 4.2 shows this simple case.

In head-coupled VR, however, we must contend with arbitrarily oriented

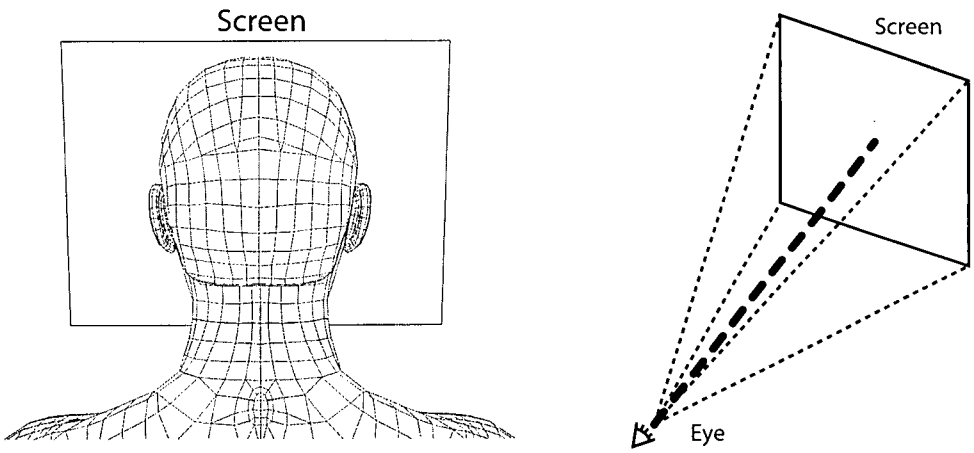

Figure 4.2: On-axis viewing of a normally oriented screen.

off-axis projections. In these projections, unlike the normal 3D graphics case, the eye-point can be anywhere and the frustum rotated at arbitrary angles, as shown in Figure 4.3. The reason head-coupled VR requires this type of projection is intuitive: the screens may be at arbitrary orientations, and the user's eye may move about. Fortunately computing these frusta is not as difficult as one might think, thanks to some functions provided by the OpenGL graphics library.

4.4.2 Specifying frusta in OpenGL

Frusta can be defined two ways using OpenGL. The common way is to use the `gluPerspective` function. This uses an eye-point, an up-vector, a look-at-point, and a field-of-view parameter to calculate the shape and orientation of the viewing frustum, as well as near and far clipping plane values to compute its depth. `gluPerspective` computes the appropriate transformation matrix and places it on the stack so that subsequent drawing commands are projected and transformed according to the specified parameters. Unfortunately, `gluPerspective` does not provide a way to specify an off-axis eye-point, so we will have to use the second OpenGL function for specifying viewing frusta, the appropriately named `glFrustum`.

The `glFrustum` function allows the specification of a viewing frustum in eye

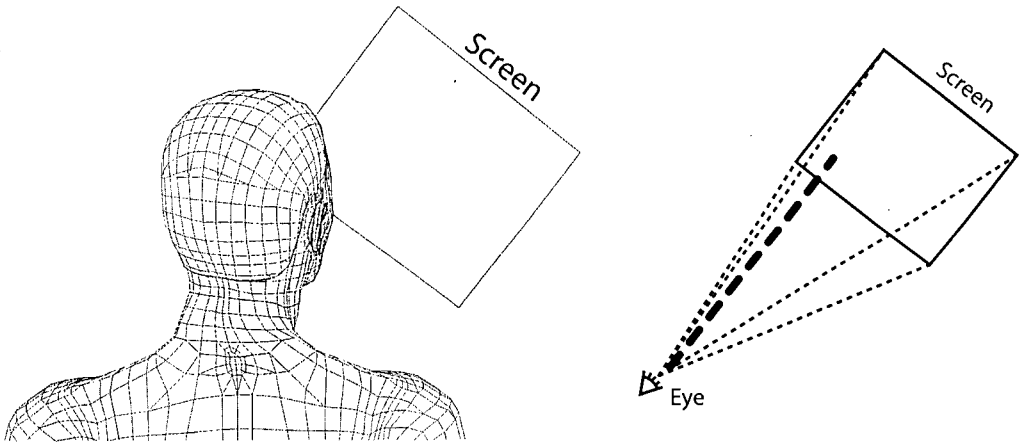

Figure 4.3: Off-axis viewing of an arbitrarily oriented screen.

space. The orientation assumed is with the x-axis to the right, the y-axis up, and a viewing direction along the negative z-axis. Notice that this implies a right handed coordinate system, and that because we are in eye space, the eye-point is the origin. Also, `glFrustum` assumes that the near and far clipping planes are parallel to the xy-plane, and assumes that the screen is located at the near clipping plane.

The `glFrustum` function accepts values for the $left$, $right$, top, $bottom$, $near$ and far clipping planes that make up the frustum. All of the values correspond to appropriate x, y and z coordinates with respect to the eye. These values also indirectly specify the corners of the 2D screen area onto which the frustum will be projected, as follows: the point $(x, y, z) = (left, top, -near)$ is the top left corner of the screen area and the point $(x, y, z) = (right, bottom, -near)$ is the bottom right corner. The reason we must negate the near clipping plane value is that by convention the near and far clipping planes are specified as their distance in front of the eye, and our viewing direction is along the negative z-axis of a right handed coordinate system. Because `glFrustum` allows the frustum to be specified anywhere with respect to the eye, it can be used to calculate the off-axis projections we need, even though the default orientation that it provides may not apply to a given screen. With a little effort, we can still generate the appropriate frustum for any eye-position

and screen orientation.

4.4.3 Computing the frustum

If the screen happens to be oriented in the default orientation that `glFrustum` assumes, specifying the viewing frustum is trivial. We subtract the current eye position, which reflects the assumed optical centre of the eye as discussed in Sections 2.3 and 3.5, from the position of the screen corners to find their position with respect to the eye. We can then use the position of these corners to enter values for the left, right, top and bottom clipping planes. The near and far clipping planes can be chosen based on what objects we are interested in rendering, and we end up with a correct off-axis projection that will show the user an appropriate image given his current eye position.

Unfortunately, the screen is generally not in the default orientation that `glFrustum` assumes. To cope with this, we first must first generate an appropriate viewing frustum, and then rotate it to correspond with the true screen orientation. Equivalently, some people prefer to think of this as rotating the screen to correspond with the default viewing frustum. Regardless, the important thing is to find an appropriate rotation matrix, and use this to obtain the correct viewing transform for a user's viewing position.

To obtain the correct values for the screen as if it were correctly oriented, we need to calculate the unit x vector ($\mathbf{u_x}$), unit y vector ($\mathbf{u_y}$) and unit z vector ($\mathbf{u_z}$) for the screen coordinate system, in the default orientation. This means, if we assume the origin of the screen is the lower left corner, that the x-axis should point to the lower right corner and the y-axis should point to the upper left corner, as shown in Figure 4.4. If we assume corners numbered clockwise from the lower left P_0, P_1, P_2 and P_3, this gives $\mathbf{u_x} = \frac{P_3 - P_0}{\|P_3 - P_0\|}$, $\mathbf{u_y} = \frac{P_1 - P_0}{\|P_1 - P_0\|}$ and $\mathbf{u_z} = \mathbf{u_x} \times \mathbf{u_y}$.

Linear algebra [leon90] tells us that the matrix $[\,\mathbf{u_x}\ \mathbf{u_y}\ \mathbf{u_z}\,]$ forms a basis for the screen orientation. By specifying the position of the screen corners to `glFrustum`

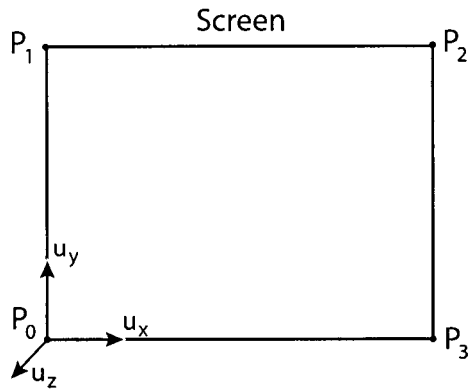

Figure 4.4: Calculating the correct viewing frustum orientation.

with respect to this new basis, we can produce a frustum that is of the correct shape and size, but oriented in the default orientation. We can then rotate the frustum to correspond with our true screen orientation and thereby produce the correct viewing frustum.

The first step is to compute the positions of the lower left and upper right corners that `glFrustum` requires to compute the frustum. Recalling that `glFrustum` uses the eye point as the origin, the new lower left corner P_0' can be found in two steps: translating the corner so that the eye point is the origin, and representing the corner with respect to the new basis. If the eye point to be used as the origin is E, then translating the lower left corner P_0 by $-E$ results in the vector $\mathbf{t} = P_0 - E$. We can find the position of the lower left corner with respect to our new basis by $P_0' = (\mathbf{t} \cdot \mathbf{u_x}, \mathbf{t} \cdot \mathbf{u_y}, \mathbf{t} \cdot \mathbf{u_z})$. The position of the upper right corner P_2' with respect to the new basis can be calculated in the same way. The *left*, *right*, *top*, and *bottom* values required by `glFrustum` can now be calculated from these two corner points as described in Section 4.4.2. We can now use `glFrustum` to produce an appropriate off-axis projection matrix, which corresponds to the position of one's eye in relation to the screen. The final step is to rotate the frustum to correspond with our true screen orientation. This is accomplished by multiplying the frustum by the transpose of our basis, or $[\,\mathbf{u_x}\ \mathbf{u_y}\ \mathbf{u_z}\,]^\top$.

The result of these operations is an off-axis projection that corresponds to the position and orientation of the screen as seen by the eye-point, though we are restricted in our choice for the near clipping plane. This is because `glFrustum` assumes that the value specified for the near clipping plane is the distance, along the z-axis, from the eye to the screen. Frequently we wish to have the near clipping plane set independently of the position of screen, and so we must work around this assumption.

To do this, we scale the size of the screen based on its distance from the viewer and its distance from the near clipping plane. We can do this in such a way that the frustum still represents the correct viewing perspective for the user, yet allows us to position the near clipping plane at an arbitrary location. If d is the distance along the z-axis from the viewer to the screen, and *near* is the distance along the z-axis from the viewer to the near clipping plane, then this scale factor is $\frac{near}{d}$. That is, if before calling `glFrustum` we multiply the *left*, *right*, *top*, and *bottom* values by this scaling factor, the *near* clipping plane value can be set to any value we choose. The resulting frustum will still be correct for the user's eye position and the given screen position, but the near clipping plane can now be repositioned independently of the screen.

We have seen all the steps required to compute an off-axis, arbitrarily oriented viewing frustum, with arbitrarily placed near and far clipping planes. If any further details on the implementation of this step are required, they can be readily found by examining the source code implementation of the `glSetOffAxisView` function, included in Appendix F. More details on using `glFrustum` can be found in the OpenGL documentation [kemp97].

This page intentionally left blank.

Chapter 5

Rendering of View-Dependent Lighting

To test the success of the calibration system design, the system was integrated with an existing rendering application. This application is especially interesting in a head-tracked environment because the Lafortune lighting model it uses to render objects is view-dependent.

The fact that the lighting of objects is view-dependent means that as the user moves his head to view the scene from different angles, he sees not only the changing perspective of the object on screen, but also a change in the lighting. This allows effects such as retro-reflection and off-specular reflection. In addition, the user is given control over the light position by making it correspond to the position of a sensor held in the user's hand. This way, the user is free to experiment in an intuitive way with the effects that different viewing angles and light positions have on rendered objects.

The workings of the calibration system have already been discussed in detail, so this chapter will focus on describing the implementation of the view-dependent lighting model using advanced graphics hardware features. This method was developed by Wolfgang Heidrich and myself as part of a graduate course in computer

graphics. Much of the following description is taken from a paper written for that course. That description has been updated to reflect the latest version of the software, which has been modified to consider the user's true eye position and to read the light position from a hand-held sensor.

The way the implementation uses the graphics hardware is non-trivial, and without clever use of this hardware it would not be possible to perform the rendering in real-time as is required for head-coupled VR. Because of this, no discussion of the the method's use in a head-coupled VR environment would be complete without an explanation of how the rendering is accomplished.

5.1 Graphics hardware features

The past few years have seen rapid development in the graphics hardware industry, specifically in the area of 3D graphics cards for the PC. As 3D graphics hardware continues to advance we find that there are increasingly more techniques that, while previously too slow to be practical, can now be performed in real-time using hardware acceleration.

Recent generations of 3D accelerated hardware, such as NVIDIA's GeForce3, provide not only faster speeds but also new features that greatly increase the flexibility of such hardware. The most exciting of these new features is the addition of programmable transform and lighting on a per-vertex basis, known as "vertex programs". This allows a developer to have full control over how the hardware performs the transformation and lighting of each vertex, as well as how texture coordinates and colours are assigned. This new programmability, as well as established features such as multi-texture, cube-mapping, and register combiners, provide great flexibility in the rendering pipeline. It is worth noting that these new features and extensions are explained in much greater detail elsewhere [nvid01], and will be merely summarised here.

5.1.1 Vertex programs

The latest advance in consumer graphics accelerators is the addition of a fully programmable hardware transform and lighting stage. Termed "vertex programs", this OpenGL extension allows a developer to write the hardware transform stage himself, using assembly language.

The assembly language consists of a repertoire of 17 SIMD instructions, each of which executes in a single clock cycle. Because of this, program execution time is directly proportional to program length. While the programs can only operate on a single vertex at a time, and have no support for branching or looping, they can take a number of constant parameters as input, as well as many per-vertex attributes, such as texture coordinates, colour, and position.

The output of these programs is at the very least a clip-space transformed vertex, but can also include colour, texture, and fog information.

Because of the flexibility of this assembly language, as well as the large number of inputs and outputs, we can use vertex programs to implement a number of non-traditional algorithms directly in hardware.

5.1.2 Texture mapping advances: Multi-texture and cube mapping

Texture mapping has always been a useful tool for improving the realism of polygonal models. Lately, however, graphics cards are supporting two new extensions to the well known texture mapping function.

Multi-texture allows a graphics card to apply multiple textures to a fragment in a single pass. This can be done in hardware when the graphics card contains several independent texture units. Each unit is configured to render a different texture, and then the texture colour values for each pixel are either blended together or used as the input to register combiners, described later, for more complicated operations.

Cube-mapping is a texturing method in which the texture is parameterised using three coordinates instead of two. This triple is interpreted as an un-normalised direction vector, and the texture itself can be thought of as lining the inside of a cube. The returned colour value corresponds to what would be seen by a viewer in the centre of the cube, looking in the specified direction at the texture lining the inside of the cube. In practice, separate textures must be specified for each of the six faces, and then the hardware performs the calculation to figure out which part of which texture a given vector specifies.

Normally cube-mapping is used to simulate reflections and refractions, but this technique can also be used to normalise vectors. First, the vector is encoded as texture coordinates. Next, textures are generated for each of the cube faces such that for a given direction $\mathbf{p}$, the point on the cube specified by $\mathbf{p}$ has a colour value that corresponds to the colour-coded vector $\frac{\mathbf{p}}{\|\mathbf{p}\|}$. We must "colour-code" this vector, as a texture's colour values may only range from 0 to 1, while components of a normalised vector can range from -1 to 1. To perform this conversion on a unit vector $\mathbf{v}$, we calculated the colour-coded vector $\mathbf{c}$ as follows:

$$\mathbf{c} = 0.5 + 0.5\mathbf{v}$$

In this way, the result of the cube-mapping texturing process for any given vector direction is simply the same vector, colour-coded and normalised. Not only does this method provide a cheap way of doing the normalisation, but it performs this normalisation per-pixel, which prevents interpolation artifacts, caused by linearly interpolated normals.

5.1.3 Register combiners

With the advent of new texture methods, and specifically multi-texture, the older OpenGL mechanism for combining pixel primary colour with several texture colours was lacking. For this reason, NVIDIA came up with a flexible way of combining several parameters to arrive at a final pixel colour.

The "register combiner" extension consists of two or more general combiners, followed by a "final combiner". While the "final combiner" is only meant to perform a colour sum and fog computation, the general combiners are much more versatile. They allow a developer to specify a number of inputs, perform addition, multiplication or dot products, and produce an output, which may in turn be used as the input to a subsequent combiner. Inputs can be any of the texture colours, vertex colours, or even constants. Because the developer has control over the inputs and operations performed on them, register combiners can be used to help implement some functions in hardware. Lafortune reflectance functions fall into this category.

5.2 Lafortune reflectance functions

Lafortune reflectance functions [lafo97] are a generalisation of the classic cosine lobe model. The most convenient form of the Lafortune reflectance function gives the amount of reflected light, f, for a single lobe, based on a unit vector $\mathbf{u}$ indicating the direction of incoming light and a unit vector $\mathbf{v}$ indicating the direction of the viewer. This form of the Lafortune reflectance function is given as:

$$f(\mathbf{u}, \mathbf{v}) = \rho \left(C_x u_x v_x + C_y u_y v_y + C_z u_z v_z\right)^n$$

where subscript notation has been used to refer to the x, y and z components of the vectors $\mathbf{C}$, $\mathbf{u}$, and $\mathbf{v}$. In this equation, the scalar n is the specular exponent, similar to the specular exponent in the Phong shading model, the scalar ρ is the albedo, which controls the overall amount of light reflected by the surface, and $\mathbf{C}$ is a normalisation factor that controls the shape of the lobe. Both $\mathbf{u}$ and $\mathbf{v}$ are defined in a local coordinate system at the point on the reflective surface being rendered.

This form of the reflectance function consists of a symmetric component-wise multiplication between $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{C}$, a three-way sum, an exponentiation, and finally a multiplication by the albedo. The useful thing about this form is that it can be simplified to merely four operations: a component-wise multiplication between $\mathbf{u}$

and $\mathbf{v}$, a dot product with $\mathbf{C}$, an exponentiation, and a multiplication. Both the component-wise multiplication and the dot product can be performed by register combiners, meaning that a large part of this function can be implemented directly in hardware.

5.3 The existing software

Register combiners were available in graphics cards well before vertex programs, and originally Wolfgang Heidrich had developed a way to use register combiners alone to implement the Lafortune reflectance model, with some limitations. We'll first examine how the old software worked, and then discuss how to use recent graphics hardware features to implement a more general solution, suitable for the head-tracking environment where we intend to use it.

The original program had a multi-stage rendering method that evaluated the function $f(\mathbf{u}, \mathbf{v})$ in parts. This method made use of the flexible way the register combiners can combine the output from texture units.

To start, two texture units are initialised. The first texture unit contains the colour-coded $\mathbf{C}$ vector for the Lafortune function that depends on the surface reflectance. The second texture unit contains a standard cube-map used for normalisation, as described in Section 5.1.2. By setting the texture coordinates correctly, the second texture unit will generate normalised, interpolated vectors indicating viewing position.

These two texture units now produce values that can be used by the register combiners. The first register combiner is configured to perform a component-wise multiplication between the first texture unit's output ($\mathbf{C}$) and a constant light direction ($\mathbf{u}$). The second register combiner performs a dot product between the output of the first register combiner and the output of the second texture unit ($\mathbf{v}$). The final combiner stage simply takes this resulting dot product and passes it through as an intensity to the frame buffer.

At this point, each pixel in the frame buffer contains an expression equivalent to $(C_x u_x v_x + C_y u_y v_y + C_z u_z v_z)$. This result is then read back into system memory from the frame buffer and the exponentiation performed using a look-up table [sloa79, bass81]. Finally, an albedo texture is rendered and multiplied with the exponentiated result to produce the final output for the function.

This process only considers a single lobe, of course, and so for models with multiple lobes this whole process is repeated, storing the intermediary results, and then computing the final result by multiplying all the lobe contributions together at the end.

The problem with this implementation is that it does not use a surface-local coordinate system for $\mathbf{u}$ and $\mathbf{v}$. Because of this, it only works for planar geometry.

Further, because the older hardware for which the software was written only had two texture units, the program could not use the cube-map method to provide a normalised light direction. Because of this, it is limited to rendering objects lit with an infinite light. In order to render with a local light, like the one controlled by the sensor in the user's hand, the vector for lighting, $\mathbf{u}$, would also have to be normalised and interpolated as is done with the viewing vector, $\mathbf{u}$.

Fortunately, by using features found in the newest graphics cards, specifically NVIDIA's GeForce3's vertex programs, we are able to overcome these shortcomings.

5.4 Implementing surface-local coordinate frames

The previous implementation had two major shortcomings: it used a constant light direction for every vertex, and it assumed all the vertices lay in the $z = 0$ plane. We will now look at how to solve these problems using vertex programs.

Vertex programs completely replace the standard OpenGL transform stage, and so we must implement not only the features that help us calculate the Lafortune equation, but also any operations that are normally provided by default OpenGL. In our case, this includes the transformation of the vertex from model space to clip

space, as well as passing the texture coordinates for the first texture unit[1] through the vertex program.

Passing the texture coordinates through our program is trivial, and can be done with the single vertex program assembly language [nvid01] command:

```
MOV o[TEX0],v[TEX0];
```

The transformation from model space to clip space is slightly more complicated. It is performed by multiplying the vertex position by the combined Modelview and Projection matrices. This can be done using the following four dot product instructions:

```
DP4 o[HPOS].x,v[OPOS],c[4];
DP4 o[HPOS].y,v[OPOS],c[5];
DP4 o[HPOS].z,v[OPOS],c[6];
DP4 o[HPOS].w,v[OPOS],c[7];
```

where `v[OPOS]` is the position of the vertex we wish to transform, and `c[4]`-`c[7]` are the rows of the combined Modelview-Projection Matrix.

The vertex program is now capable of correctly transforming a vertex, and must now perform the calculations required to light it correctly. We will assume that the texture units and register combiners are set up as described in Section 5.3, except that the constant valued input to the first general register combiner (used for the light direction) is replaced with an input read from the primary colour. The reason for this is that we are no longer assuming a constant light direction for each vertex and need to provide a way for our vertex program to pass the correct light direction to our register combiners. Primary colour is a per-vertex attribute, which is interpolated for pixels between vertices. This means it will work well for passing

[1]According to OpenGL conventions, the texture units are numbered starting at 0. The assembly language examples, therefore, refer to the first texture unit as `TEX0`, the second texture unit as `TEX1`, etc.

the light direction, as long as our models are well tessellated. If our model is not well tessellated we will run into artifacts caused by the fact that the linear interpolation of our normals is not length preserving.

We can now formulate a definition of what our vertex program needs to output, and consequently what it needs to receive as input. The output of our program will be a vector pointing to the viewer, and a vector pointing to the light. The vector pointing to the viewer will be output from our program coded as the texture coordinates for texture unit two which, as we recall, is a cube-map set up for normalisation. The vector pointing to the light will be colour-coded as the vertex primary colour. This way each of these vectors can then be input to the register combiners for further computation.

In order to calculate the vector pointing to the light and to the viewer, we need to consider the local coordinate frame of each vertex. This requires a normal, binormal and tangent for each vertex. While OpenGL provides a mechanism for assigning a normal to each vertex, and the binormal can be calculated by taking the cross-product of the normal and tangent, we still need a way to pass the tangent to our vertex program. We will do this by encoding the tangent as the texture coordinates for the third texture unit. Even though this texture unit is not otherwise used, it's coordinates can be passed to the vertex program to provide a tangent direction.

We will also need three vertex program constants to complete our calculations: the position of the light, in model space; the position of the viewer, in model space; and the vector $[0.5\ \ 0.5\ \ 0.5\ \ 0]^{\mathsf{T}}$, used in the colour-coding calculation.

With our inputs and outputs defined, we are now ready to begin calculations. First, we have to calculate the local binormal, by computing the cross product of the normal and tangent. We first read the tangent into register `R1`, and then calculate the cross product in register `R2`:

```
MOV R1,v[TEX2];
```

```
MUL R2,v[NRML].zxyw,R1.yzxw;
MAD R2,v[NRML].yzxw,R1.zxyw,-R2;
```

We can now calculate the model space viewing direction as $ViewerDirection = VertexPosition - ViewerPosition$, or, in vertex program assembly where R3 is the viewer direction and c[1] is the viewer position:

```
ADD R3,v[OPOS],-c[1];
```

Our vertex program can now formulate a change-of-basis matrix to transform this model space vector to surface-local coordinate space by using the tangent, binormal and normal as the columns of a matrix [leon90], as in Section 4.4.3 when the viewing frustum was calculated. We then multiply our vector by this change-of-basis matrix to rotate the viewing direction into surface-local coordinates using three dot products:

```
DP3 o[TEX1].x,R3,R1;
DP3 o[TEX1].y,R3,R2;
DP3 o[TEX1].z,R3,v[NRML];
```

This surface-local viewer direction is now the output to the second texture unit.

The model space vector for lighting direction is calculated and stored in register R4 by:

```
ADD R4,v[OPOS],-c[2];
```

It must be explicitly normalised, however, as it is not passed to a cube-map for normalisation. The normalised vector is stored in R3:

```
DP3 R0.w,R4,R4;
RSQ R0.w,R0.w;
MUL R3.xyz,R4,R0.w;
```

It is then transformed to surface-local space in the same way as with the viewing vector:

```
DP3 R6.x,R3,R10;
DP3 R6.y,R3,R2;
DP3 R6.z,R3,v[NRML];
```

Finally, this result is colour-coded, according to the formula given in section 5.1.2. We assume `c[3]` has been set as the constant vector $(0.5, 0.5, 0.5, 0)$. The result is then returned as the primary vertex colour:

```
MUL R4.xyz,R6,c[3];
ADD o[COL0].xyz,c[3],R4
```

The complete vertex program consists of only 21 commands. This is remarkably efficient considering all that the program does, and because each command runs in a single clock cycle, it means that on a 300 MHz processor the program would be capable of processing in excess of 14.28 million vertices per second! This efficiency allows us to use highly tessellated models to reduce any error resulting from interpolations between vertices.

5.5 Results

This method is capable of rendering arbitrary geometry, lit using Lafortune reflectance functions, in real-time. Further, because the parameters to the Lafortune functions are specified in textures, they can change per-pixel. This means we are able to have specular lighting on some parts of geometry and retro-reflective lighting on others, for example.

There is one problem with doing the calculations in the way described, however: we are limited by a low-precision frame buffer. Because we perform our exponentiation with eight bits of precision, we run into quantisation problems, which

can be seen in the sample image, Figure 5.1. In this image, the yellow part of the texture is retro-reflective, while the black kangaroo shows a specular reflection. The quantisation caused by the lack of precision is easily seen in the specular highlight in the centre of the image, and is worsened by the low complexity of the geometry. This problem is much less apparent on curved geometry, with more vertices, as shown in Figure 5.2.

Even though our method runs in real-time we waste a lot of time by reading back pixels from the frame buffer. At very high resolutions, this will slow down the rendering substantially, and also limits the number of lobes we can render in a timely fashion. Though this does not prevent the method's use in our head-coupled environment, looking at a way to eliminate frame buffer read-backs would improve overall performance. One way this could be done is covered in Appendix D.

Integrating the calibration software with the application was not difficult, despite the application's extensive use of OpenGL extensions and its vertex program implementation of the transform and lighting stage. No special provisions were required to adopt the utility library functions to the application, and most of the modifications were to allow the application to read the light and eye position from the tracking system. The ease in which the calibration system was integrated into this advanced rendering application shows the soundness of its design, and its feasibility as a useful tool for would-be VR practitioners.

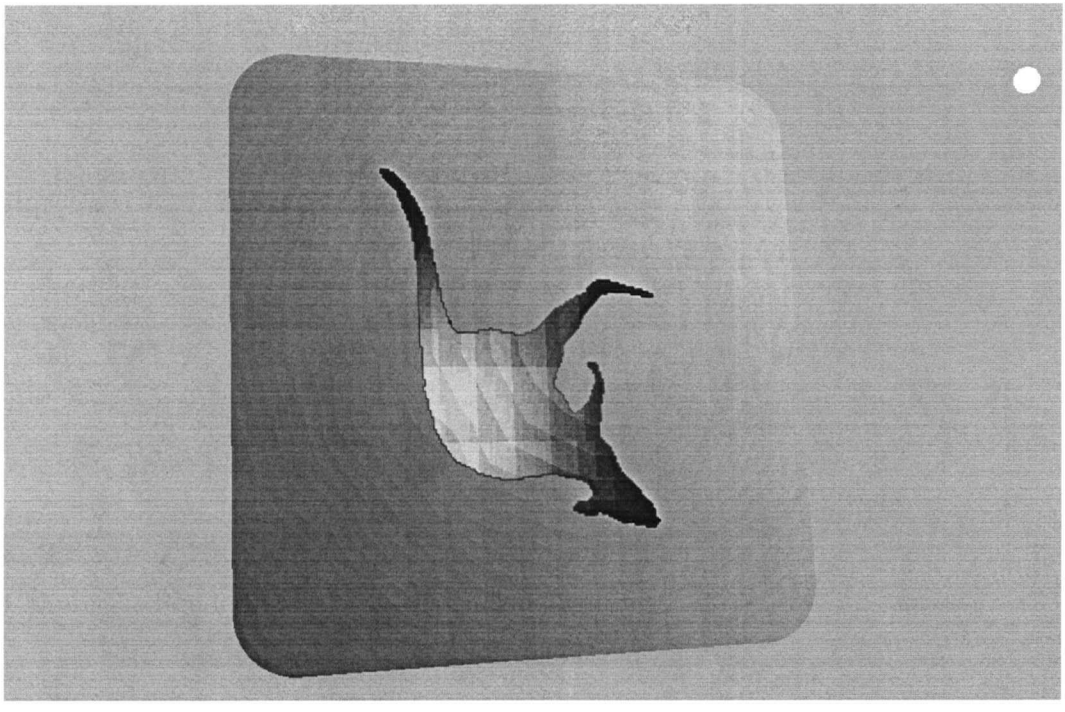

Figure 5.1: Quantisation artifacts due to low frame buffer precision.

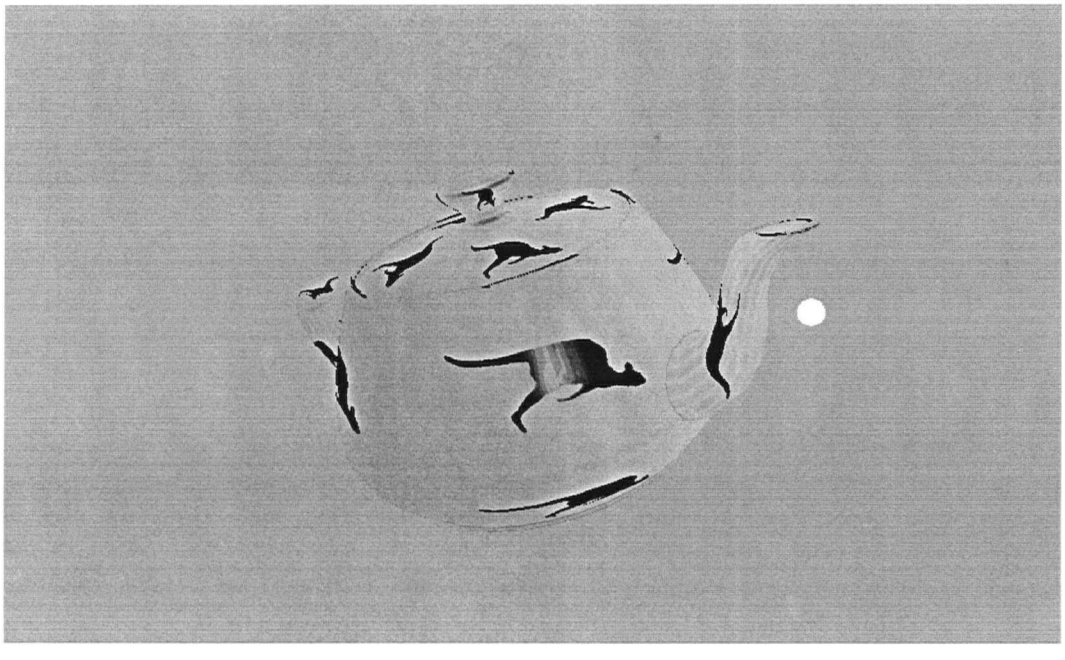

Figure 5.2: Curved, dense geometry masks precision problems.

This page intentionally left blank.

Chapter 6

Conclusions

This thesis has presented a system for calibrating head-coupled VR environments. By examining the requirements for such a system, as well as relevant design principles, we have succeeded in producing a convenient, flexible solution to the problem of calibrating both desktop and large-scale systems for use in a head-coupled virtual reality environment.

The "sighting" technique has been shown to be an extremely flexible tool for locating points in 3D. By taking line samples that intersect target points, and analysing these samples to compute the location of the targets, we can accurately measure the location of points without placing the sensors near the actual target. Though this is useful in avoiding the interference that display screens have on magnetic sensing equipment, it is a major advance in finding the position of the rotational centre of the eye. By providing a method to precisely measure the true rotational centre of the eye, and doing this in a fast, convenient way, the system provides distinct advantages over traditional methods of calibration. Further, because it is easy to modify the measurements stored in the calibration files produced by our system, other calibration methods can be used if they are more appropriate for a given environment and the values they produce inserted into the appropriate calibration files. These values can then be used by the utility libraries provided to

easily and automatically provide appropriate off-axis projections based on the user's eye positions.

Beyond the process of mere calibration, this thesis has also described a new way to render view-dependent Lafortune reflectance functions in real-time. The method correctly considers the lighting model in a coordinate system local to the surface of the object being rendered, and correctly calculates lighting and viewing vectors for each vertex of the object based on the positions of a light and viewer. Further, the fact that the parameters for the Lafortune model are stored in texture coordinates means they can be varied per-pixel, in a single pass. This allows different parts of an object to exhibit different lighting characteristics, with no penalty in rendering time. Though the technique suffers in quality from the poor precision of the frame buffer, which is used to store an intermediate value in the lighting calculation, it still holds promise as a real-time rendering technique. Moreover, it provides an engaging demonstration of how both the lighting as well as the view of the rendered objects can be linked to a head-coupled perspective.

6.1 Future work

Though the system is currently complete in that it offers all the features necessary for calibrating most head-coupled VR environments, there are a number of ways it could be improved. These range from new features to more robust methods of analysing the data it collects. In general, the system favours speed over accuracy, and convenience is chosen over rigour. This section will look at some ways work could be done to improve the accuracy and robustness of the system. We will also look at possible improvements to the Lafortune rendering method we have described, and describe new studies that can be done to qualify the need for accurate calibration in VR environments.

6.1.1 Tracker input

Though the system currently uses a low-pass filter to help reduce random noise while taking calibration measurements, there exist better forms of filtering. Kalman filtering [kalm61, brow83, lian91] would be especially appropriate as it is able to match the type of movement common in tracking systems better than a regular averaging filter. Also, because Kalman filtering can be used to either predict data, at the cost of accuracy, or to smooth it, increasing latency, it would be suitable for both the calibration measurements as well as use in applications. Measurements taken during the calibration process could be smoothed to reduce errors, while applications could choose predictive filtering to minimise latency while tracking head-position.

Currently, filtering is only used in the calibration programs themselves, and no general interface is provided for reading from the tracking system. Work could be done to create a general "input driver" that not only supports filtering of different types, but also abstracts the type of tracking system used from the rest of the system. This abstraction would be very useful in extending the flexibility of the calibration system as a whole, as the calibration system currently assumes the use of a Polhemus FASTRAK. Not only could different input devices be used seamlessly with the calibration system, but applications could use this input driver as well. This would avoid having each application re-implement code to read from the tracking system, as is currently required.

6.1.2 Intersecting line samples

Work could be done to bring statistical rigour to the method of finding the best intersection of a number of lines in 3D, taking into account the way they were sampled. The fact that errors are likely to be greater when the angles between the lines are very small could be considered, and methods could be used to try to estimate the error associated with different measurements. One such method would be to weight line sample pairs according to the distance of the shortest line segment

between them (the shorter the line segment, the closer they are to intersecting), as well as according to the cosine of the angle between them (lines that are near parallel are more sensitive to measurement errors).

6.1.3 More flexible transformations

Currently, the transformations between different spaces have certain assumptions made about them. The screens are assumed to be perfectly flat and rectangular, and only four points are taken to approximate their position in 3D. By sampling the positions of a grid of points on the screen, much more information could be obtained about the mapping from frame buffer to 3D position. This would allow not only better approximations of the true screen position, but also automatic corrections for keystone effects and other uniform geometric distortions. Though such a system would be much more difficult to incorporate into regular OpenGL, it would add further flexibility to the system.

6.1.4 Lafortune rendering improvements

The method described for rendering Lafortune reflections is far from perfect. Certainly the first step would be to implement the method described in section D.2, once hardware is available that supports texture shaders on five texture units. As hardware progresses further, other features could be added that would improve both the quality and speed of the rendering.

One promising feature being introduced in the latest graphics hardware is the addition of special modes for high-precision frame buffer values [nvid01]. Different pixel storage formats can be specified that have different amounts of precision, and there is a chance that a way could be found to avoid the precision errors present in the current method.

In the future, finding a way to implement self-shadowing of objects and distance attenuation for lights would also add greatly to the realism of the rendering.

Ultimately work could be done to develop the rendering method into a rendering engine capable of displaying an entire virtual world in real-time.

6.1.5 More hardware support

High-end SGI systems sometimes have multiple rendering pipelines, each with its own frame buffer. On these systems, each rendering pipeline must be calibrated separately, and the drawing to each pipeline then controlled by the application. The calibration system will currently work on these systems, but treats each pipeline separately. In the future, support could be added to allow the calibration system to automatically support multiple rendering pipelines. This would require changes to both the utility libraries, as well as the existing file formats, and would probably result in unnecessary complications for users who only need to use the system on single-pipeline systems. It would, however, allow software to run unmodified on these multi-pipeline systems and take advantage of both the additional displays and speed provided by the extra rendering pipelines.

As new display technologies are developed, and as new tracking systems are developed, the calibration method could be adjusted to match. The method of finding eye position, for example, is fundamentally sound, and could be used with gaze tracking cameras and an auto-stereoscopic display to allow accurate head-coupled stereo display without the need for glasses. Future work can centre around ways to use the ideas behind the calibration method to make the most out of emerging tracking and display technologies.

6.1.6 Studying the effects of error

Not only does this thesis facilitate the development of future head-coupled VR software, but it also allows new lines of research into the effects of calibration errors on task performance in a VR environment. Using our accurate, simple method for finding eye position, future studies can now examine how different types of induced

error in the position of a user's eyes affect the perception of the VR environment. By varying different aspects of a user's eyes' calibration, such as the interocular distance, or the predicted location of the optical centres of his eyes, we can determine which measurements are most important in providing a realistic VR display. This will provide new information about the human visual system, and allow future calibration systems to take advantage of this to provide increasingly compelling virtual environments.

Bibliography

[arth93] Kevin W. Arthur, Kellogg S. Booth, and Colin Ware. "Evaluating 3D task performance for fish tank virtual worlds". *ACM Transactions on Information Systems*, 11 (3), pp. 239-265, July 1993.

[bass81] Daniel H. Bass. "Using the video lookup table for reflectivity calculations: Specific techniques and graphics results". *Computer Graphics and Image Processing*, 17 (3), pp. 249-261, November 1981.

[bour98] Paul Bourke. "The shortest line between two lines in 3D". April 1998. *http://astronomy.swin.edu.au/~pbourke/geometry/lineline3d/*

[brow83] R.G. Brown. *Introduction to Random Signal Analysis and Kalman Filtering*, John Wiley & Sons, Inc., 1983.

[cruz92] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. DeFanti, Robert V. Kenyon, and John C. Hart. "The CAVE Audio Visual Experience Automatic Virtual Environment". *Communications of the ACM*, pp. 64-72, 1993.

[cruz93] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. "Surround-screen Projection-based Virtual Reality: The Design and Implementation of the CAVE". *Proceedings of SIGGRAPH '93*, pp. 135-142, 1993.

[deer92] Michael F. Deering. "High resolution virtual reality". *Proceedings of SIGGRAPH '92*, 26 (2), pp. 195-202, 1992.

[fole82] James D. Foley, Andries Van Dam, Steven Feiner, and John Hughes. *Computer Graphics: Principles and Practice*, 2nd edition, Addison-Wesley Publishing Company, 1990.

[gave95] William W. Gaver, Gerda Smets, and Kees Overbeeke. "A Virtual Window on Media Space". *Proceedings of the Conference on Human Factors in Computing Systems (CHI'95)*, pp. 257-264, 1995.

[kalm61] R.E. Kalman and R.S. Bucy. "New results in linear filtering and prediction theory". *Transactions of ASME (Journal of basic engineering)*, 83d, pp. 95-108, 1961.

[kemp97] Renate Kempf, Chris Frazier and the OpenGL Architecture Review Board. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.1*, Addison-Wesley Developers Press, 1997.

[lafo97] Eric P.F. Lafortune, Sing-Choong Foo, Kenneth E. Torrance, and Donald P. Greenberg. "Non-Linear Approximation of Reflectance Functions". *Proceedings of SIGGRAPH '97*, pp. 117-126, 1997.

[leon90] Steven J. Leon. *Linear Algebra With Applications*, fourth edition, Macmillan Publishing Company, 1990.

[lian91] Jiandong Liang, Chris Shaw, and Mark Green. "On temporal-spatial realism in the virtual reality environment". *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 19-25, 1991.

[lind01] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. "A User-Programmable Vertex Engine".. *Proceedings of SIGGRAPH 2001*, pp. 149-158, 2001.

[nvid01] NVIDIA OpenGL Extension Specifications. May 2001. *http://developer.nvidia.com*

[polh00] *3SPACE FASTRAK User's Manual*, 2000 edition, revision A. Polhemus Incorporated, June 2000.

[sloa79] K.R. Sloan, Jr., and C. M. Brown. "Color map techniques". *Computer Graphics and Image Processing*, 10 (4), pp. 297-317, August 1979.

[smet87] G.J.F. Smets, C.J. Overbeeke, and M.H. Stratmann. "Depth on a flat screen". *Perceptual and Motor Skills*, 64, pp. 1023-1034, 1987.

[stev00] Alexander Stevenson. "A Driver for the Polhemus FASTRAK and Its Integration Into the OpenGL Graphics Interface". April 2000. *http://www.cs.ubc.ca/~alex/*

[summ99] Valerie A. Summers, Kellogg S. Booth, Tom Calvert, Evan Graham, and Christine L. Mackenzie. "Calibration For Augmented Reality Experimental Testbeds". *Proceedings of 1999 ACM Symposium on Interactive 3D Graphics*, pp. 155-162, 1999.

[suth65] Ivan Sutherland. "The ultimate display". *Proceedings of IFIP Congress*, pp. 506-508, 1965.

[suth68] Ivan Sutherland. "A head-mounted three dimensional display". *Fall Joint Computer Conference, AFIPS Conference Proceedings*, 33, pp. 757-764, 1968.

[swin00] Colin Swindells, John C. Dill, and Kellogg S. Booth. "System lag tests for augmented and virtual environments". *Proceedings of the 13th Annual Symposium on User Interface Software and Technology (UIST-00)*, pp. 161-170, November 2000.

This page intentionally left blank.

Appendix A

Running the Calibration Software

This appendix describes how to run the calibration system and will be of most interest to those who plan on actually using the system. It will also be of interest for those who wish to see how the steps explained in Chapter 3 were implemented in the final calibration system.

A.1 Program conventions

Each step of the calibration is performed by a separate program. Though they all perform different tasks, they share many similarities, which was done to make the calibration as simple as possible. Each program will be described in detail later. First we look at some important assumptions that are made and common aspects of all the programs.

A.1.1 Tips on "sighting" accurately

"Sighting" is the way the system locates points in 3D. For the calibration to be reliable, it is very important that the sighting be done correctly and carefully, so that accurate measurements are obtained.

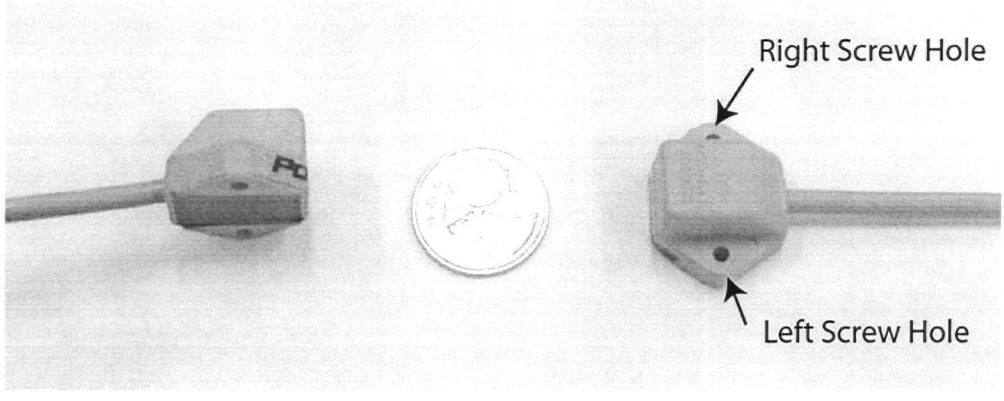

Figure A.1: FASTRAK sensors and screw holes.

Sighting is performed by lining up the screw holes of two FASTRAK sensors and looking through them. Because each FASTRAK sensor has two screw holes, it is critical that the correct hole be used. Holding the sensor, with the cable trailing down and the flat part facing away from you, you must look through the *left* screw hole, as shown in Figure A.1. It is important that this be understood by the experimenter as well any user of the system. One way to ensure this is to block the other hole completely so that it cannot be used to look through. This can be easily done with one of the plastic screws that ship with these sensors when they are purchased from Polhemus.

Sighting finds the location of a point by intersecting two or more lines, which are aimed at that point. To ensure that errors are minimised, these lines should be at large angles to one another, as close to 90° as possible. This way, small errors in aiming will not cause large errors in determining the intersection of the lines. You will find that mounting the sensors on a bracket, so they both can be held with one hand, will make aiming much easier. Even better, mounting the sensors on a tripod makes it easy to ensure that the sensors are precisely positioned at the time a line sample is taken.

Because the FASTRAK is a magnetic tracking system, it is adversely affected by aspects of the environment that emit or affect magnetic fields. If the FASTRAK

sensors or transmitter are located very near large metal objects, CRT monitors, or LCD panels, the tracker's accuracy will be greatly diminished. For this reason, the sensors should be kept at least 10 cm away from the surface of either CRT monitors or LCD panels while measurements are being taken, and metal objects in the environment should be avoided.

Finally, the accuracy of the FASTRAK system depends on the distance between the sensors and the transmitter. In order to obtain the best results, one should try to keep the sensors within approximately one metre of the transmitter.

A.1.2 Use of the calibration files

All the calibration programs write their results to one of three calibration files: a 2D screen calibration file, a 3D screen calibration file, or a user calibration file. These files are plain ASCII text files and can be opened up for editing in any text editor. Examining these files during the calibration will help you understand the type of information that has been collected by a given program. See Appendix C for sample calibration files, as well as an explanation of their formatting.

A.1.3 General notes on running the software

Running any of the programs with no command line arguments will display a help screen showing the correct syntax for running the command. Because each program takes a number of parameters, these help screens are invaluable reminders of how to run the calibration system. Any parameters listed on the help screen in square brackets are optional.

None of the programs writes to an output file until it finishes the calibration step it is performing, so if you decide that you do not want the program to write to a file you have specified, close the program before it finishes the calibration.

A.2 `calibrate2D`: Assigning the frame buffer to screens

The first step in the calibration is finding which parts of the frame buffer are shown on each display screen. This step is performed by the `calibrate2D` program, which takes two parameters: the number of screens, and a filename to use as the 2D screen calibration file. This file will be created if it does not exist, and overwritten if it does.

The program will prompt the user to specify two opposite corners of each screen, and use the position of these corners to determine the extent of each screen in the frame buffer. This is done as follows: the user is prompted to select the first corner. The user clicks with the mouse in the corner of one of the visible screens, and then uses the cursor keys to make sure that the horizontal and vertical guide-lines are as close to the edge of the screen as possible, while still being visible. The user presses the space-bar to record the position of that corner. The user then clicks the opposite corner of the screen and ensures that the guide-lines are positioned correctly with the cursor keys, as before. Once correctly positioned, the user presses the space-bar again to record the corner position. This process is repeated for each screen attached to the system.

This step is very easy to perform, as long as one remembers that the cursor keys are the best way for finely adjusting the on-screen guidelines. This step will need to be redone if the number of display screens attached to a system changes, or if the resolution of these screens change. Fortunately, these types of changes happen very infrequently. Also, notice that the FASTRAK is not required for this step of the calibration.

A.3 `calibrate3D`: Finding screens in the real world

Once the system knows where to place images so that they are seen on each screen, these screens must be located in 3D space. This is done with the `calibrate3D`

program, which takes a number of command line parameters: the name of the 2D configuration file generated by `calibrate2D`, the file-descriptor that refers to the FASTRAK sensing system, the baud rate at which the FASTRAK is operating, the number of the first sensor used for sighting, the number of the second sensor used for sighting, and optionally the name of the file to use for the 3D screen calibration file. The file-descriptor for the FASTRAK is that of the serial port to which the FASTRAK is connected, usually `/dev/ttyS0` or `/dev/ttyS1`. The "number of the sensor" is the number next to the input port that the sensor is plugged into on the body of the FASTRAK system, from one to four. If no filename is specified for the 3D screen calibration file, the name "`3D.cfg`" will be used by default, and that file will be created or overwritten as necessary.

The number of screens attached to the system is determined by reading the 2D calibration file, so it is important that the 2D calibration file be correct for the current system configuration. For each screen, the program will prompt the user to aim at a target near a corner of the screen and press the space-bar to take a sample. To aim at the target, look through the left screw hole (left is defined as explained in Section A.1.1) of each sensor you specified as a sighting sensor, and line up the target at the centre of both holes. If the target is obscured by text on the screen, pressing "T" will toggle the display of on-screen messages and allow you to see the target unobstructed. Once the target is lined up through the holes, press the space-bar to record a line sample. Next, aim at the target point again from a different location and take another line sample. As many line samples as desired can be taken this way, pressing the space-bar to record each. The total number of line samples collected is indicated by the on-screen display. If you make a mistake, pressing "Backspace" will delete the most recent sample from memory and allow you to retake it. Of course, to erase several samples you can press "Backspace" several times. As long as you have two or more line samples recorded, you can proceed to the next target point by pressing "Enter".

Once four points on each screen are found they will be used to estimate the position of the screen corners. These corners, in turn, will be run though a procedure to make each screen rectangular and planar, if it is not already. The results of these adjustments will be displayed after the file is written, and you should inspect these values to make sure that everything seems reasonable.

Because this step of the calibration locates the screens in 3D with respect to the tracking system, it will need to be repeated if either the screens change position or the FASTRAK transmitter is moved.

In some cases, where the geometry of the screens has been measured already or is known by other means, you may wish to generate a 3D calibration file by hand. If you generate a 3D calibration file by hand and wish to run only the check procedure, without the calibration, this can be done by running `calibrate3D` with the "`-fix`" parameter. In this mode, you need only specify the 2D calibration file, an existing 3D calibration file, and optionally a filename to output the new 3D calibration data (otherwise the default "`3D.cfg`" is used, as before). The existing 3D calibration file will be checked, adjusted to ensure the screens are planar and rectangular, and a new calibration file will be written with the new data.

A.4 `calibrateOrigin`: Specifying which way is up

Now that we know the position of the screens in 3D, we need to update the 3D calibration file with the origin of the virtual world. That is, we have to indicate from where in real space the virtual world should originate. This is easily done with the `calibrateOrigin` program. This program can be used to set the origin and default orientation of the virtual coordinate system in three different ways, according to either the position of a screen, a sensor position, or manually specified values.

The first form of the command is chosen with the `-screen` switch. This is by far the easiest and most useful of the three modes. In this mode, a screen number,

a 2D calibration file and a 3D calibration file are specified. The calibration files are read, and the 3D calibration file is updated to reflect a virtual coordinate system originating in the centre of the specified screen, and having a default orientation such that the x and y axes of the screen correspond to the x and y axes of the virtual world. This command makes it very easy to pick a centre screen, and automatically have the virtual world appear in front of this screen, in the correct orientation.

The second form of the command is selected with the `-sensor` switch. For this command, the FASTRAK file descriptor, baud rate, sensor number, 2D calibration file and 3D calibration file must be specified. The origin and orientation are then read from the FASTRAK sensor.

The final form of the command is used to manually specify values for the origin and orientation. This mode is selected with the `-manual` switch, and expects the x, y, and z position and h, p, and r Euler angles of the origin as well as the names of the 2D and 3D calibration files.The origin information is stored in the 3D calibration file, so this calibration step should be repeated anytime the 3D calibration file is generated.

A.5 `calibrateUser`: Locating a user's eyes

The final step in calibrating the VR system is locating the user's eyes with respect to a head-mounted sensor. This step is accomplished by the `calibrateUser` program. This program takes the FASTRAK file descriptor, baud rate, first "sighting sensor" number, second "sighting sensor" number, head-mounted sensor number, and optionally the user calibration file filename from the command line. If no filename is specified for the user calibration file "user.cfg" is used. Notice that this program does not read in either a 3D calibration file or a 2D calibration file, and is completely independent of the other calibration steps. This step must be redone, however, each time the sensor is moved on the user's head. This can happen if the sensor is bumped or pulled, or if it is removed and replaced. In either of these cases,

the calibration should be re-run to ensure accurate results.

This program functions very much like the `calibrate3D` program. For each eye, the user is asked to look through the left screw holes (left is defined as explained in Section A.1.1) of two sensors. No particular target needs to be lined up through the holes. After the first line sample is taken, by pressing the space-bar, the user should turn his eyes in another direction and look through the sighting sensors again. This can be repeated for as many samples as desired, and as with the `calibrate3D` program the "Backspace" key can be used to erase the most recent sample. Anytime after two or more samples have been taken for an eye, the "Enter" key can be pressed to proceed.

As long as the user can see through the centre of both holes the calibration will work, but for best accuracy the user should be aware of the direction he is looking with respect to his head. Samples should be taken with the user looking in different directions. For example, if one sample is taken with the user's eyes turned left, the next should be taken with the user's eyes turned right. It doesn't matter which way the user's eyes are turned, as long as each line sample is taken with the user looking in a different direction with respect to his head.

After both eyes are calibrated, the interocular distance is computed and displayed. This value should be near 6 cm for most people, and if it is far from that value something has gone wrong and the calibration should be repeated. Once this calibration is performed the user should not remove or reposition the head-mounted sensor until he is finished using the head-coupled environment, or this calibration should be redone.

This calibration can be made quicker and easier by rigidly mounting the sighting sensors on a bracket next to the viewing screens. This way, the user does not have to touch the sensors in order to look through them, and can progress though the calibration very quickly.

Appendix B

Developing Software Using `Projector` and `VRConfig`

This appendix explains what is necessary to allow a developer to easily add head-tracking and multiple monitor support to his own virtual reality programs, using the `Projector` and `VRConfig` utility libraries. In addition to this appendix, the header files for these libraries as well as the source code for the Lafortune rendering application are extremely helpful when developing applications that make use of the calibration system. For more information regarding the free, cross-platform `sg` vector library that is required by `Projector` and `VRConfig`, please refer on-line to `http://plib.sourceforge.net`.

B.1 Using the `VRConfig` class

The `VRConfig` class encapsulates all the calibration information for a given VR setup, as well as the methods to read and write appropriate configuration files. The functions in this class deal with the tasks of reading and writing calibration files. Each function takes a filename as a parameter and then reads or writes the appropriate calibration information to that file. The definitions of these functions are as follows:

```
int load2DScreenPos(const char * filename);
int load3DScreenPos(const char * filename);
int loadUserData(const char * filename);
void save2DScreenPos(const char * filename);
void save3DScreenPos(const char * filename);
void saveUserData(const char * filename);
```

The functions to load data return an integer, which is 1 if the file was loaded and parsed successfully and 0 otherwise. Note that the 2D calibration data file (read by load2DScreenPos) should always be loaded first when loading data because this file is used to determine the number of screens in the system. When saving data, the specified filename will be created if it doesn't exist, and overwritten if it does.

Once the calibration data is loaded it is stored in a number of public class variables. These variables consist of an integer that holds the sensor number attached to the user's head, from one to four, vectors that hold the position of the left eye, right eye, virtual origin and Euler angles for the orientation of the virtual coordinate system. For the most part these are handled by functions in the Projector library, and typical applications do not need to worry about them. Also in the VRConfig library are arrays of vectors that hold the position of the screen corners in both 3D tracker space and 2D frame buffer space. These are accessed by screen number and corner number, where the corners are numbered in the order of top left, top right, bottom left, and bottom right. Most of the time these values are only used by functions in the Projector library.

The number of screens attached to the system is stored in the variable numScreens. This number should be used to determine how many times an application needs to draw the virtual world to ensure it is displayed on each screen attached to a system. More details on how this is done will be provided later, in Section B.3.

B.2 Using the Projector class

A Projector object is assigned a VRConfig object, which it uses to determine correct off-axis projections and eye-positions for a given environment. This allows an application to simply call functions in the Projector class, instead of having to deal with the VRConfig data itself. Though the Projector class only has five functions, these provide everything one needs to produce a head-coupled off-axis perspective with a minimum of effort.

`setConfig(VRConfig * newCfg)`

This function must be called before a Projector object can be used. It gives the Projector object a pointer to a valid VRConfig object, which it should use to read in all the required calibration information.

`findEyeCentre(sgVec3 eyePos, sgVec3 headPos,`
`sgVec3 headEuler, bool rightEye)`

This function locates the rotational centre of an eye, based on a given head position (headPos) and orientation (headEuler), by looking up the calibrated position in the VRConfig object. This eye position is then returned as the value for eyePos. If rightEye is true then the position of the right eye is returned, otherwise the position of the left eye is returned.

`moveEye(sgVec3 eyePos, sgVec3 gazePoint, int distance=0.6)`

Applications that wish to predict the exact location of the optical centre of a user's eye can use the moveEye function to perturb the position of the rotational centre appropriately. If eyePos is the current rotational centre, and gazePoint is the point that the user is looking at, then this function will move eyePos by a distance (default is 0.6 cm) in the direction of gazePoint. The gazePoint must be approximated depending on the application. Often either the centre of a screen that has the user's focus, or the location of a 3D pointer is used.

`glSetViewport(int ScreenNum)`

This function gives an appropriate `glViewport` command to OpenGL to restrict drawing in the frame buffer to the area specified for screen number `ScreenNum`.

`glSetOffAxisView(sgVec3 eyePos, int ScreenNum, float near, float far)`

This function computes the correct off-axis viewing frustum for screen number `ScreenNum`, given the optical centre of an eye `eyePos`. The `near` and `far` clipping planes are specified as the distance from the viewer to the planes. Also, the origin of the virtual coordinate system will be read from the `VRConfig` object and the correct transformation computed. When finished, this function will have set the OpenGL Projection Matrix to correctly perform the off-axis projection and origin transformation, and the OpenGL Modelview Matrix will be set to the identity. In short, this function performs all the steps required to ensure that the virtual world is correctly projected onto the screen `ScreenNum`.

B.3 Designing your VR application using GLUT

The libraries for interfacing with the calibration system work very easily in conjunction with the GLUT application framework. This section will outline some suggestions on how to best use the libraries with the GLUT call-back system. More detailed information about using GLUT in applications is readily available online.[1]

An application that uses the calibration system should provide some way to specify which calibration files should be used, and then load these files into a `VRConfig` object. The 2D calibration file should be loaded first, followed by the 3D calibration file and then the user calibration file. A pointer to this `VRConfig` object can then be given to a `Projector` object using the `Projector.setConfig` function.

An application that uses head-coupling must perform at least two tasks:

[1] `http://www.opengl.org/developers/documentation/glut.html`

reading the current position of the sensors in the tracking system, and displaying the virtual world to the screen.

Reading from the Polhemus FASTRAK is described in detail by my undergraduate thesis,[2] and the driver which accompanies that thesis makes this task very simple. It is most easily accomplished in the context of a GLUT application by having the GLUT "idle" call-back process new data from the tracking system and save the results. This provides a position for the head-mounted sensor.

The position of the head-mounted sensor can then be fed to the `Projector` object, so that an eye position can be determined. This is done by calling the `Projector.findEyeCentre` function to find the rotational centre, followed by the `Projector.moveEye` function to locate the approximate optical centre. Stereo displays are easy to implement, as one need only change the `rightEye` flag to generate positions for one eye or the other.

The final required component of a head-coupled application consists of drawing the virtual world to the screen. Initiated by a call to the display call-back, a typical 3D application would perform this task by setting the OpenGL viewport to determine where the image will be drawn, setting the OpenGL Projection Matrix to a perspective transformation, and finally using the OpenGL Modelview Matrix to adjust the position of geometry as it is drawn to the screen. The `Projector` class makes the head-coupled case just as easy.

The display calls are placed in a loop, which repeats `VRConfig.numScreens` times, once for each attached screen. A call to `Projector.glSetViewport` automatically sets the viewport correctly for the current screen, and similarly a call to `Projector.glSetOffAxisView` sets the correct OpenGL Projection Matrix for the viewer's off-axis perspective. The application can then proceed to draw the virtual scene as for a regular OpenGL application. The net result will be that every attached screen will automatically show a correctly calibrated, head-coupled perspective view

[2]The undergraduate thesis is available online at `http://www.cs.ubc.ca/~alex/`.

of the virtual world.

Appendix C

File Formats For Calibration Data

This appendix describes the file formats used for storing the calibration data. These files can be easily modified with a regular text editor, and by default they are generated with comments explaining the values on each line.

Samples of these files will be reproduced here for easy reference. In any file, lines beginning with the pound sign (#) are treated as comments and are ignored by the parser. Indications in comments as to which points are which are only for the benefit of human readers; the system uses the order that the points are found in the file to determine which are which. Finally, individual components of a point should be separated by a comma.

C.1 2D Screen Data calibration file

The 2D Screen Data calibration file contains the locations of screen corners in frame buffer coordinates. No explicit indication of the number of screens present is read by the parser. Instead, the system will read in groups of four points at a time, and increment the number of screens accordingly.

```
# 2D screen positions in frame buffer coords (1 screen)
# Top Left Corner, Screen 1
0.00, 818.00
# Top Right Corner, Screen 1
1151.00, 818.00
# Bottom Left Corner, Screen 1
0.00, 1.00
# Bottom Right Corner, Screen 1
1151.00, 1.00
```

C.2 3D Screen Data calibration file

The 3D Screen Data calibration file contains the locations of screen corners in tracker space, as well as the origin and orientation of virtual space with respect to tracker space. The first non-comment line of the file is a six-tuple of values: the first three represent the x, y, and z position of the origin position, and the next three are the Euler angles for the orientation.

Successive lines in the file correspond exactly to the lines of the 2D Screen Data file, and the parser will expect the same number of screens defined in the 3D data file as were defined in the 2D file. For this reason, the 2D file is always read in first to determine the number of screens, and this number is assumed for the rest of the system.

```
# Origin XYZ Position and Euler Angles
36.49, -44.65, -25.56, 3.92, -97.48, 4.53
# 3D screen positions in cm from origin (1 screen)
# Top Left Corner, Screen 1
17.69, -47.55, -36.35
# Top Right Corner, Screen 1
53.59, -45.09, -39.19
# Bottom Left Corner, Screen 1
19.40, -44.21, -11.93
# Bottom Right Corner, Screen 1
55.29, -41.75, -14.78
```

C.3 User Data calibration file

The User Data calibration file contains the location of the user's eyes from the head-mounted sensor, as well as the sensor number to read for the head position (this applies to the Polhemus FASTRAK, which supports up to four sensors). The first non-comment line of the file is the sensor number, from one to four, and the following two lines are 3D points that represent the rotational centre of the right and left eye, respectively.

```
# Sensor attached to head (1-4)
4
# Position of right eye from head sensor, in cm
5.35, -1.13, 16.60
# Position of left eye from head sensor, in cm
3.11, -7.58, 17.41
```

This page intentionally left blank.

Appendix D

Further Lafortune Rendering Optimisations

Though fast enough to allow head-tracking in real-time, the method for rendering Lafortune reflectance functions described in Chapter 5 is limited to using two lobes for the lighting model because of performance considerations. Currently, the slowest parts of this method for rendering Lafortune reflectance functions are the calls to read back pixels from the frame buffer. This is a notoriously slow operation, and the problem is compounded by having to read the frame buffer once for the first lobe, to perform the exponentiation, and twice for each additional lobe, because we need to store intermediary results.

It is possible to get around this, however, by offloading more of the computation to our vertex program, and then making use of the new texture shader extension, which provides new modes for indexing into and combining textures. By computing the component-wise multiplication of the light and viewer direction in the vertex program, and then using texture shaders to compute the dot product, we can use a dot product texture lookup to calculate the exponentiation, allowing us to perform the entire lighting computation without reading back the frame buffer. It is worth noting, however, that these optimisations have not yet been implemented,

as they require five texture units to completely eliminate frame buffer reads and the GeForce3 we used to run our software has only four texture units. For the sake of completeness, we will now examine how we could implement this method, given a card with enough texture units. First, however, we need to look at another OpenGL extension: the texture shader.

D.1 Texture shaders

The texture shader extension not only offers a number of new texture modes, allowing the calculation of dot products and dependent texture lookups, but also introduces new pixel formats, including a signed RGBA8 representation. Of course, the regular GL_TEXTURE_2D mode is still supported, as is the conventional unsigned RGBA8 pixel format, but by using the new modes, we can coax the hardware into doing more of our Lafortune calculation.

Each texture shader stage takes an input from a previous shader stage, and provides both a shader stage output as well as a colour output. The shader stage output can be used as the input to subsequent texture shaders, while the colour output behaves as a regular texture output, and can be used as input to the register combiners.

The two shader modes useful to us are the ones used to compute simple dot product 2D texture lookups.

The GL_DOT_PRODUCT_NV mode computes the dot product of the texture shader's coordinates with the shader input. It then passes this result as the first component of the shader output.

The GL_DOT_PRODUCT_TEXTURE_2D_NV shader mode must be preceded by a GL_DOT_PRODUCT_NV. It also computes a dot product between an input and its texture coordinates, but then uses the result from the first dot product as well as the result from the dot product it just computed to form a 2D lookup into a texture, which has been bound to the unit. The colour result returned is that of

the filtered 2D target texel.

D.2 Avoiding frame buffer read-backs

We now have all the tools we need to perform the exponentiation and blending of the albedo in hardware.

In order to do this, five texture units, numbered zero through four, are initialised as follows:

Texture unit zero contains a texture that has in its red channel the specular exponent n to be used in the Lafortune function. This texture unit is initialised as a GL_TEXTURE_2D shader, and the pixel format for the texture should be the standard unsigned RGBA.

Texture unit one is configured as a GL_TEXTURE_2D shader with a signed RGBA8 texture that contains the Lafortune parameters C_x, C_y and C_z in the RGB components.

Texture unit two is a GL_DOT_PRODUCT_NV shader that takes its input from texture unit one. The texture coordinates for this unit are provided by the vertex program, and are equivalent to $[\, u_x v_x \;\; u_y v_y \;\; u_z v_z \,]^\top$. This way, the output of this texture unit is the dot product

$$q = \begin{bmatrix} C_x \\ C_y \\ C_z \end{bmatrix} \cdot \begin{bmatrix} u_x v_x \\ u_y v_y \\ u_z v_z \end{bmatrix} = (C_x u_x v_x + C_y u_y v_y + C_z u_z v_z)$$

which is the un-exponentiated part of the Lafortune equation.

Texture unit three has an unsigned RGBA texture bound to a shader of type GL_DOT_PRODUCT_TEXTURE_2D_NV, and takes its input from texture unit zero. The texture coordinates provided by the vertex program must always be $(1, 0, 0)$ so that the dot product calculated is equal to the specular exponent (contained in the red channel) specified by the input texture.

This shader then uses the result q from texture shader two as the first component for a texture lookup, and the specular exponent n as the second component. Texture unit three will now return a texel from its currently bound texture, at location (q, n). To allow this process to calculate our exponentiation, we must generate a 2D exponent lookup table and bind it as an unsigned RGBA texture to unit three. This way, the result of texture unit three will be the exponentiated portion of the Lafortune model: $(C_x u_x v_x + C_y u_y v_y + C_z u_z v_z)^n$.

Texture unit four is a GL_TEXTURE_2D shader that contains the unsigned RGBA albedo texture. The result of this texture is equivalent to the albedo term in the Lafortune equation, ρ.

Now that our textures are set up, and generating both the exponentiated term and the albedo term, all that remains is for us to multiply them together. A single register combiner can be used to multiply the output of texture three with the output of texture four, and this final result can now be written to the frame buffer.

We have thus managed to compute an entire lobe in a single pass, and can therefore blend subsequent lobes without reading back from the frame buffer. This represents a significant improvement in speed, without sacrificing our ability to support local lighting and a local viewer.

Not yet described is the vertex program which generates the correct sets of texture coordinates to allow the texture shaders to perform the desired dot products correctly. This vertex program is fairly straightforward, and is largely the same as the vertex program used in the current implementation. The only differences are: the eye vector and view vector are normalised and then multiplied inside of the program, and returned as a texture coordinate; a constant texture coordinate of $(1, 0, 0)$ is returned for texture unit three; and finally the albedo texture unit coordinates must be passed through the program unchanged.

The only downside to this method is that parts of the equation are calculated

per-vertex, and other parts are calculated per-pixel. This could lead to interpolation problems for the parts that are calculated per-vertex. For well tessellated models, however, this should not prove too serious a concern.

This page intentionally left blank.

Appendix E

Calibration Results

This appendix provides tables of sample data collected by the calibration system. Because the accuracy of the system depends on the care with which the calibration is performed, as well as on the distance between the FASTRAK sensors and the transmitter when a measurement is taken, it is not practical to do a formal error analysis of the system. Instead, by running the calibration numerous times on the same environment, and comparing the results obtained each time, we can get some idea of how well the system performs. Though this type of test will not reveal sources of systematic error in our measurements, it will give us an idea of the accuracy of the system and the effect of random errors.

The only two steps of the calibration that are dependent on tracking system measurements are the calibration of 3D screen position and the calibration of the user's eyes. Each of these steps has been repeated ten times and the results of these runs have been summarised.

E.1 Calibrating Eye Position

A sensor was rigidly affixed to the plastic liner of a construction helmet. This liner was removed from the helmet, and adjusted to fit securely on a user's head. Two more sensors were attached to each other so that they were approximately 15 cm

Table E.1: Measured eye positions and distance from the means.

Left Eye	Dist. from Mean	Right Eye	Dist. from Mean
(7.89, 5.04, 12.19)	0.19 cm	(6.92, 0.31, 15.56)	0.22 cm
(8.11, 5.16, 12.20)	0.14 cm	(6.99, 0.53, 15.44)	0.08 cm
(7.95, 5.14, 12.16)	0.07 cm	(7.10, 0.35, 15.51)	0.16 cm
(8.00, 5.23, 12.07)	0.07 cm	(6.99, 0.36, 15.42)	0.10 cm
(8.05, 5.23, 12.19)	0.10 cm	(6.99, 0.51, 15.53)	0.12 cm
(7.99, 5.23, 12.11)	0.05 cm	(7.16, 0.56, 15.39)	0.18 cm
(7.91, 5.22, 12.04)	0.12 cm	(6.99, 0.56, 15.31)	0.16 cm
(7.94, 5.13, 12.05)	0.10 cm	(6.92, 0.45, 15.38)	0.11 cm
(8.06, 5.22, 12.10)	0.08 cm	(7.08, 0.49, 15.29)	0.15 cm
(8.04, 5.21, 12.12)	0.05 cm	(7.08, 0.45, 15.42)	0.06 cm

apart, and so that a user could see through both left screw holes of the sensors. This apparatus was held by the user in one hand to perform the calibration and the other hand was used to press the keys on the keyboard. The entire calibration was performed by a single person, unaided, and the sensors were not mounted on a tripod or other rigid mounting.

Table E.1 shows the measured positions for the rotational centre of the each eye. The average position of each eye was calculated, and the table also shows the distance of each measurement from the average position for that eye. We see that the our values lie in a very small range (around 2 mm) and this suggests that our calibration method is quite accurate. If the sensors had been rigidly mounted on a tripod of some sort, the values may have been even better. Also, it is possible that some of the variance in the positions is due to movement of the sensor on the user's head. Movement of both the user's hair and skin as well as the flexible plastic helmet liner could account for the few millimetres of change we see in our data.

E.2 Calibrating 3D Screen Position

The 3D screen calibration step was performed with the same apparatus as the calibration of the eyes: the user performed the calibration unaided, and though the

Table E.2: Measured screen corner positions, and distance d from the means.

Top Left Corner	d	Top Right Corner	d
(29.38, -63.68, -6.13)	0.46 cm	(63.62, -63.17, -6.78)	1.29 cm
(29.45, -63.24, -6.91)	0.62 cm	(63.90, -61.71, -7.69)	0.54 cm
(29.62, -63.32, -6.80)	0.49 cm	(64.08, -61.85, -7.17)	0.45 cm
(29.58, -63.62, -6.43)	0.18 cm	(63.74, -62.19, -7.17)	0.25 cm
(29.61, -64.35, -6.42)	0.63 cm	(63.69, -62.44, -7.23)	0.46 cm
(29.57, -63.73, -6.79)	0.23 cm	(63.55, -61.28, -7.77)	0.86 cm
(29.61, -64.08, -6.54)	0.34 cm	(63.60, -62.17, -7.34)	0.20 cm
(29.50, -63.92, -6.35)	0.28 cm	(63.36, -61.00, -7.26)	1.04 cm
(29.85, -63.71, -6.58)	0.31 cm	(63.71, -61.85, -7.00)	0.33 cm
(29.30, -63.77, -6.64)	0.26 cm	(63.42, -62.24, -7.55)	0.43 cm
Bottom Left Corner	d	Bottom Right Corner	d
(29.80, -61.45, 17.90)	0.66 cm	(64.04, -60.95, 17.26)	1.90 cm
(29.90, -60.84, 17.82)	0.07 cm	(64.35, -59.31, 17.04)	0.40 cm
(29.78, -60.83, 17.75)	0.14 cm	(64.24, -59.35, 17.37)	0.49 cm
(29.98, -60.67, 17.85)	0.18 cm	(64.14, -59.24, 17.12)	0.22 cm
(29.98, -60.66, 17.76)	0.17 cm	(64.07, -58.75, 16.95)	0.32 cm
(30.08, -60.95, 17.68)	0.22 cm	(64.06, -58.51, 16.69)	0.65 cm
(30.00, -60.72, 17.75)	0.12 cm	(63.98, -58.81, 16.94)	0.27 cm
(29.90, -61.06, 17.69)	0.25 cm	(63.76, -58.15, 16.78)	0.99 cm
(29.97, -60.48, 17.39)	0.50 cm	(63.83, -58.62, 16.97)	0.50 cm
(29.81, -60.49, 17.97)	0.40 cm	(63.93, -58.97, 17.07)	0.15 cm

sensors were attached so that they could be held with one hand, they were not mounted on a rigid tripod of any sort. An ordinary desktop CRT monitor was used for the calibration. Table E.2 shows the measured positions of the four screen corners, as they are reported by the calibration software after they have undergone the adjustment procedure described in Section 3.3. Also shown is the distance from the mean for each corner measurement.

The values collected for this calibration step vary farther from the mean than the values for eye position, suggesting they are less accurate. This is understandable given that the task of lining up a target point in the sights is more difficult than the act of merely looking through the sights. Also, it is difficult to hold the sensors, aim them, and press the key to collect the sample simultaneously. It is likely that

having the sensors rigidly mounted on a tripod would make aiming much easier, and therefore improve the accuracy of the measurements. Even so, no value varies more than 2 cm from the mean, and the majority of the values lie within 5 mm.

Appendix F

Source Code of glSetOffAxisView

This appendix provides a listing of the source code implementation of the utility library's `glSetOffAxisView` function, which computes a user's viewing frustum as described in Section 4.4.3. It makes extensive use of the free sg vector library, which can be obtained from `http://plib.sourceforge.net`.

```
void Projector::glSetOffAxisView(const sgVec3 eyePos, int ScreenNum,
                                 float near_clip, float far_clip) {

    sgVec3 eyeToScreen, xProj, yProj, zProj;
    float width, height, distance, left, right, bottom, top;

    // Calculate vector from eye to screen origin (lower left corner)

    sgSubVec3(eyeToScreen, eyePos, cfg->screens[ScreenNum][2]);

    // Calculate vectors from bottom left corner of screen that form
    // the screen coordinate frame

    sgSubVec3(xProj, cfg->screens[ScreenNum][3], cfg->screens[ScreenNum][2]);
    width = sgLengthVec3(xProj);
    sgScaleVec3(xProj, SG_ONE / width);
    sgSubVec3(yProj, cfg->screens[ScreenNum][0], cfg->screens[ScreenNum][2]);
    height = sgLengthVec3(yProj);
    sgScaleVec3(yProj, SG_ONE / height);
    sgVectorProductVec3(zProj, xProj, yProj);
```

```
    // Now, specify an off-axis viewing frustum in order to define the
    // viewing volume with respect to the user's eye position.

    left = sgScalarProductVec3(eyeToScreen, xProj);
    right = width - left;
    bottom = sgScalarProductVec3(eyeToScreen, yProj);
    top = height - bottom;
    distance = sgScalarProductVec3(eyeToScreen, zProj);
    left = -left * (near_clip / distance);
    right = right * (near_clip / distance);
    bottom = -bottom * (near_clip / distance);
    top = top * (near_clip / distance);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    glFrustum(left, right, bottom, top, near_clip, far_clip);

    // Rotate the viewing frustum to correspond with the actual
    // screen orientation.

    GLfloat worldToScreenM[16] = {xProj[SG_X], yProj[SG_X], zProj[SG_X], 0.0,
                                  xProj[SG_Y], yProj[SG_Y], zProj[SG_Y], 0.0,
                                  xProj[SG_Z], yProj[SG_Z], zProj[SG_Z], 0.0,
                                          0.0,         0.0,         0.0, 1.0};
    glMultMatrixf(worldToScreenM);

    // Initialize some variables

    sgVec3 zeroVec;
    sgZeroVec3(zeroVec);
    sgMat4 originTransform;

    // Render as seen from user's eye position

    glTranslatef(-eyePos[SG_X], -eyePos[SG_Y], -eyePos[SG_Z]);

    // Do correction for offset origin of Universe

    sgMakeCoordMat4(originTransform, cfg->originPos, cfg->originEuler);
    glMultMatrixf((GLfloat *)originTransform);

    // Make Modelview matrix the identity.

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

}
```