VIRTUAL REALITY ENGINE DEVELOPMENT

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Varun Varahamurthy

June, 2014

COMMITTEE MEMBERSHIP

TITLE:	Virtual Reality Engine Development
AUTHOR:	Varun Varahamurthy
DATE SUBMITTED:	June 2014
COMMITTEE CHAIR:	Jane Zhang, PhD Associate Professor of Electrical Engineering
COMMITTEE MEMBER:	Wayne Pilkington, PhD Associate Professor of Electrical Engineering
COMMITTEE MEMBER:	Fred W. DePiero, PhD Associate Dean for Undergraduate Programs & Curriculum Innovation Professor of Electrical Engineering

ABSTRACT

Virtual Reality Engine Development

Varun Varahamurthy

With the advent of modern graphics and computing hardware and cheaper sensor and display technologies, virtual reality is becoming increasingly popular in the fields of gaming, therapy, training and visualization. Earlier attempts at popularizing VR technology were plagued by issues of cost, portability and marketability to the general public. Modern screen technologies make it possible to produce cheap, light head-mounted displays (HMDs) like the Oculus Rift, and modern GPUs make it possible to create and deliver a seamless real-time 3D experience to the user. 3D sensing has found an application in virtual and augmented reality as well, allowing for a higher level of interaction between the real and the simulated. There are still issues that persist, however. Many modern graphics/game engines still do not provide developers with an intuitive or adaptable interface to incorporate these new technologies. Those that do, tend to think of VR as a novelty afterthought, and even then only provide tailor-made extensions for specific hardware. The goal of this paper is to design and implement a functional, general-purpose VR engine using abstract interfaces for much of the hardware components involved to allow for easy extensibility for the developer.

TABLE OF CONTENTS

LIST OF FIGURES

1. Introduction

Virtual Reality (VR) is a broad term typically used to refer to and describe a variety of technologies associated with immersion into a simulated 3D environment. As a field of study, it can be thought of primarily as the point where Human Computer Interaction (HCI), Computer Graphics (CG), Computer Vision/Digital Image Processing (CV/DIP) and 3D Sensing meet. This chapter will outline the basics of VR along with the objectives of this project.

1.1. Definition

The term “Virtual Reality” can be traced back to times predating the first computer. In his book *The Theater and Its Double (1938)*, French playwright Antonin Artaud used the words “la réalité virtuelle” to describe theatre itself, referring to its ability as an art form to immerse audiences within the story being told. In modern times, the term has evolved to allude specifically to the use of computers to create simulated virtual experiences for users. Other terms have been coined over the years to describe related concepts such as “Augmented Reality”, “Immersive Multimedia” and “Artificial Reality” which arguably fall under the umbrella of Virtual Reality. In this paper, the term Virtual Reality will be used in a more general context. It will refer to the use of computers to provide an abstraction so that users can interact with virtual resources in a more natural, intuitive and realistic way.

1.2. Motivation

Traditional user interfaces that we have grown accustomed to are due for an upgrade. Traditional interface devices like keyboards, mice and computer monitors are clearly unintuitive and represent relics from a much older time. In fact, the modern keyboard layout comes from that of typewriters…a layout designed to impede typing speed to prevent the jamming of the mechanism. It may not seem that way to most people who have been using devices like these for much of their adult lives, but many people who remember first learning how to use a keyboard or a mouse might tell you that it was a frustrating experience. Consider the fact that the mouse limits user input to a two dimensional space. That may seem like an odd issue to take with an interface device, but think about how frustrating life would be if you were only allowed two degrees of freedom to interact with objects in the real world. In light of these issues, it would be fair to say that the goal of VR is to provide a means by which to facilitate natural interaction between humans and machines.

The future of VR depends heavily on the demonstrable benefits of current technology. VR has existed for a long time but was considered simply a novelty for the gaming community. Much of

the early innovation in VR came from the military for use in training and simulations. These systems were large, complex and very expensive. For this reason, making VR systems available to the general public was generally considered a pipe dream. Today, recent innovations in head mounted displays (HMD) technology, 3D depth cameras, graphics hardware and motion controllers have become widely available and relatively cheap. Yet, even with these massive leaps in technology, modern systems involving VR are still only developed specifically for a given purpose with the VR components added in later as an afterthought. To illustrate this point further, modern game systems are built specifically for traditional PC or Console gaming, forcing the user to interact with the application using a mouse, keyboard or gamepad; an interfacing paradigm that has remained largely unchallenged for years. Part of the reason for this is that developers are largely unaccustomed to the new emerging paradigm of "natural" user interfaces like head-mounted displays and 3D skeleton tracking sensors. Some game engines developers such as UDK and Unity have lead the charge in facing this issue by providing extensions for independent developers to allow for the use of these new technologies in their own creations. This is a step in the right direction, but a small step nonetheless. The game industry is being discussed in detail here because it has been and will most likely continue to be the greatest proponent to VR technology, yet even here there seems to be a stagnant response.

Another issue is that there is no compatibility standard between specialized VR devices. This hinders the motivation of developers who see time invested in development for a specific device as a waste if it only benefits the small number of consumers who own that device. Specialized VR hardware must be held to a basic compatibility standard much like most modern input/output devices, thereby incentivizing the development of software applications that can take advantage of said new hardware with a relatively small effort. This is not unlike the argument made for the development of the USB (Universal Serial Bus) device standards in the mid-1990s.

For non-gamers who still use computers for other purposes, this may seem like a waste of time, but the fact remains that there are several other industries that would be greatly benefitted by the introduction of VR functionality. VR has applications that extend far beyond video gaming; into robotics, biomedical imaging, therapy, entertainment and education just to name a few.

1.3. Objectives

Put simply, the objective of this paper is to explore, develop and implement a working VR engine primarily for use in an educational setting. A series of abstractions will be developed and demonstrated that will aid developers and students in creating robust and intuitive VR applications.

The majority of the practical work done on this project will be in the form of C++ code. The rest will be the compilation and further development of a theoretical foundation for users and students looking to utilize this work in their own projects. The engine should at the very least provide a simplified abstract framework for basic graphics programming, stereo rendering, HMD feedback, gesture recognition and 3D sensing. The framework should abstract the student or user away from the nitty-gritty details of the hardware integration and low level APIs so that they may deal specifically with the creation of VR applications. The software interface must remain at least a layer above the hardware so as to accommodate other similar hardware systems. In other words, integration of new hardware should only require a specific extension of an abstract interface in software to maintain compatibility with the rest of the system. This goes for the graphics rendering system as well.

The rendering interface will be composed of a series of abstract objects, with a specific implementation for OpenGL 3.x. Support for other low level graphics libraries will be easy enough to implement through the abstractions provided. The code for the engine will be written in standard C++. All external libraries will be cross-platform, with the exception of the Kinect SDK which is currently only officially supported on Windows.

The Oculus Rift will be the stereo HMD system that will be used, but the framework will be written so as to be agnostic to the specific HMD in use. The stereo HMD interface will include basic head tracking functionality and a raw sensor feedback pipeline. Any HMD capable of providing this information should be compatible with the framework.

A basic gesture recognition framework will be implemented as well. This will be done in the context of the 3D sensing system, which is in this case the Microsoft Kinect. The gesture recognition framework will be used to implement a simple grab detection algorithm as a proof of concept. Because the results of this project are meant for human consumption and developer friendliness, the success of the project as a whole can only truly be judged in terms of end user ease of use and design pattern.

2. Related Works & History

To understand where VR is headed, it is important to understand where it came from. As stated earlier, the origins of VR are deeply intertwined with those of Computer Graphics and Human-Computer Interaction. The purpose of this chapter is to provide a brief timeline for the major developments in either of these areas and provide insight into their effects on the growth of VR as we know it today.

2.1. The Sensorama

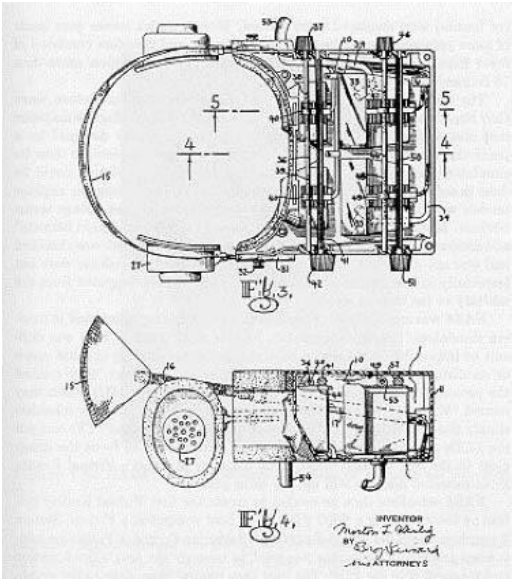
Many consider the birth of VR to be the development of a single user console known as the Sensorama. Invented in the mid-1950s by Morton Eilig, a cinematographer, the Sensorama included a stereoscopic display, stereo speakers and even a tilting chair. It was designed to provide a multi-sensory, immersive experience for users by engaging as many senses as possible, even including a mechanism to trigger aromas during a film. Heilig's described his ideas for the Sensorama as early as 1955 before building the actual prototype in 1962, but the project was halted due to lack of funding. To this day, the Sensorama is considered to be the first VR system, despite it being an entirely mechanical machine with no digital computing capabilities.

Figure 2-1: The Sensorama

2.2. The HMD

Heilig's Sensorama was not his only VR contribution. In 1962, he filed a patent for what is widely considered the first Head-Mounted Display (HMD) ever designed. The system called for the use of optics to simulate a wide angle view of 3D photographic slides, along with a stereo sound system and an "odor generator". It was essentially meant to be a portable version of the Sensorama.

Figure 2-2: Heilig's HMD



Heilig's HMD was never built, but his ideas had sparked the interests of others who desired to make the HMD a reality. In 1961, two employees of the Philco Corporation managed to construct the first HMD, called the Headsight, out of a single CRT screen strapped to a helmet and a magnetic tracking system to provide orientation feedback. The system was designed to be used in conjunction with a remote CCTV system to provide viewing into relatively dangerous environments. Around the same time, computer scientists and engineers like Sutherland, Roberts and Warnock were working on algorithms and systems that would allow computers to display graphical data on CRT displays. Naturally, the two areas of research collided resulting in the first computer interfaced HMD called the Ultimate Display, invented in 1965 by Ivan Sutherland [1]. The Ultimate Display had a mechanical tracking system rigged to a skyline system on the ceiling. The technology quickly found funding and applications in military helicopters to be worn by the pilots.

Figure 2-3: The Ultimate Display

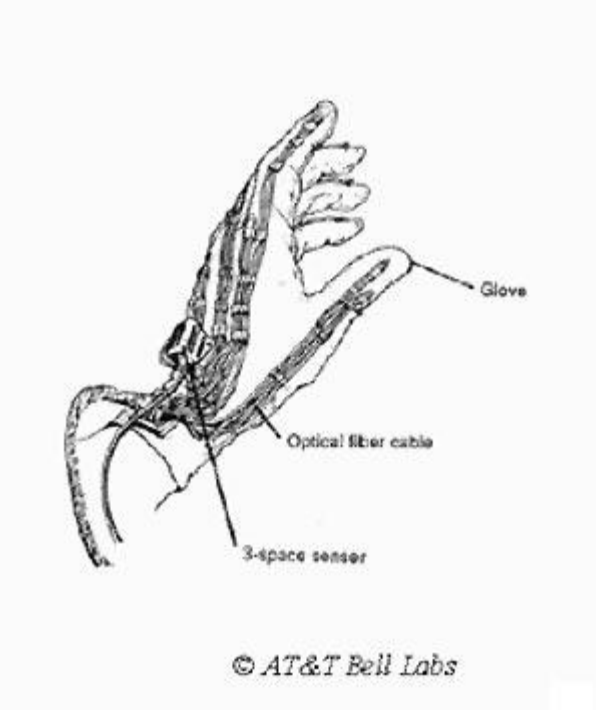
Over the next 20 years or so, many other HMD designs would emerge, each generation [2]improving upon the last.

2.3. Natural Interfacing

Another front of the HCI movement was starting to investigate new ways to allow users to be more hands-on with their data. The Sayre Glove, described by Tom Defanti and Daniel Sandin in 1977, was one of the first instrumented gloves to be. The glove used light sensors and photocells on opposite ends of optical fibers on a glove to determine how bent each finger was.

In 1983, Dr. Gary Grimes at Bell Labs developed the Grimes' Digital Data Entry Glove which had tactile sensors at the fingertips, flex sensors along each finger and position/orientation sensors at the wrist. It was designed to allow for "fingerspelling" or entering alphanumeric characters via hand position.

Figure 2-4: The Sayre Glove



Many other variants of these controllers would emerge over the course of the decade, with the most recent relying on computer vision techniques. Pnevmatikakis and Anagnostopoulos described a fast process by which to track the center of the palm by finding the point with the largest distance to its nearest contour point [2]. Amayeh, Bebis, Erol and Nicolescu proposed a way to segment the hand based on fitting the largest circle to the palm and inferring the position of the fingers from the non-palm regions [3]. With faster processing hardware, vision based hand tracking is gaining popularity not just because of its speed but also because it requires no wearable hardware.

2.4. Current State of VR

The issues facing earlier designs in VR hardware such as cost, portability and power have been greatly mitigated, allowing for the emergence of a new generation of VR technology. New HMD systems such as the Oculus Rift are made possible by smaller and higher resolution displays. Research in IR imaging and computer vision has made real-time 3D depth sensing a reality, which is in turn used in RGB-D cameras like the Microsoft Kinect, making it possible to track a human

body in 3D without the use of wearable devices. Audio systems have also become more commonplace and have since been standardized; so much so that modern audio hardware provides hardware acceleration for 3D audio for use in a wide variety of multimedia applications. Dedicated graphics hardware designed with a high degree of parallelization in mind is capable of processing and rendering millions of vertices and polygons at high resolutions in real-time. These innovations and advances in hardware must be matched in software to make VR a more viable medium, as it should be already.

3. System Overview

It is important to get a top-down perspective of the engine as a whole which is what this chapter is meant to provide. The architecture of the engine and how it relates to the hardware will be explained in detail here along with some of the basic motivation and methodology behind the abstractions used throughout the engine. No code will be presented here, but standard object oriented programming terminology will be used as well as some C++ specific terminology.

3.1. Abstractions

The first thing that must be understood before going any further is the concept of abstraction and how it applies to this project. Webster's dictionary defines abstraction as follows: "a general idea or quality rather than an actual person, object, or event: an abstract idea or quality." In general, we can think of abstraction as the process by which we form symbolic representations of objects and concepts. The simplest example of this is found in language. We use words, which are basic human constructs, to represent an immense variety of things. The word "automobile" for example represents any powered, four-wheeled vehicle that can carry people. There are many different variants of automobile, yet most of the time, it does us no good to be any more specific in conversation. This brings us to the other useful quality of abstraction, which is that it provides us a means by which to simplify a set of seemingly varied objects to single abstract representative object.

3.2. Object Oriented Programming

This is the fundamental idea behind object oriented programming, where concepts, ideas and structures can be represented as **objects** defined through code structures called **classes**. For example, one could define a class called "automobile" which has certain properties, also called **fields**, such as horsepower, mpg rating, and capacity. The class could also define behaviors that the object would be capable of performing through constructs called **functions**. In the case of the automobile, these functions might include "drive", "stop", "speed-up/down", etc. Every object created through that class would be considered an **instance** of that class and may have properties that vary. Automobile instances might differ in terms of their properties but they would all be able to perform the functions defined. The fact that such a class is so general allows us to describe a multitude of automobiles but gives us no way to distinctly describe cars, busses or motorcycles. Under the current structure of our class, there is no difference between these things. They are all

simply automobiles. To do that, we would have to define new classes, with each one **extending** the automobile class.

Classes that extend another class are called **subclasses** or **derived classes**. The class they extended is referred to as the **superclass** or **base class**. A derived class **inherits** all of its functionality and properties from the base class but is free to define fields and functions unique to itself. For example, the bus class inherits from the automobile class but it is the only type of automobile that can stop at a bus stop.

Now consider the situation in which two automobiles perform the same function, but differ in the way that they perform it. For example, all automobiles must be able to brake but they don't all have to do it in the same way. A large bus might use a pneumatic system to release a failsafe brake pad while a car may accomplish the same task using a hydraulic system. In a case such as this where the specific implementation of a function varies between subclasses, a concept known as an **overloading** is used. The base class might define a default **overloadable** function that the derived classes are free to implement differently in their own definitions. In C++, this type of function is called a **virtual function.** If there is no context to define the function in the base class and it is consequently left undefined, the function it is known as a **pure virtual function**, and the base class is known as an **abstract** or **virtual** base class and no instances of the base class may exist. In the case of the automobile, if the "brake" function is pure virtual within the automobile class, the bus and car derived classes must each define a specific implementation of this function if they are to be instanced within the application. A base class that is composed purely of virtual functions is called an **interface.**

3.3. Interfaces

The idea behind interfaces in a modular design is essentially to insulate components from others and to aide interoperability between them. From a systems engineering standpoint, interfaces are essential for maintaining compatibility during upgrades and setting up a standard for extensibility

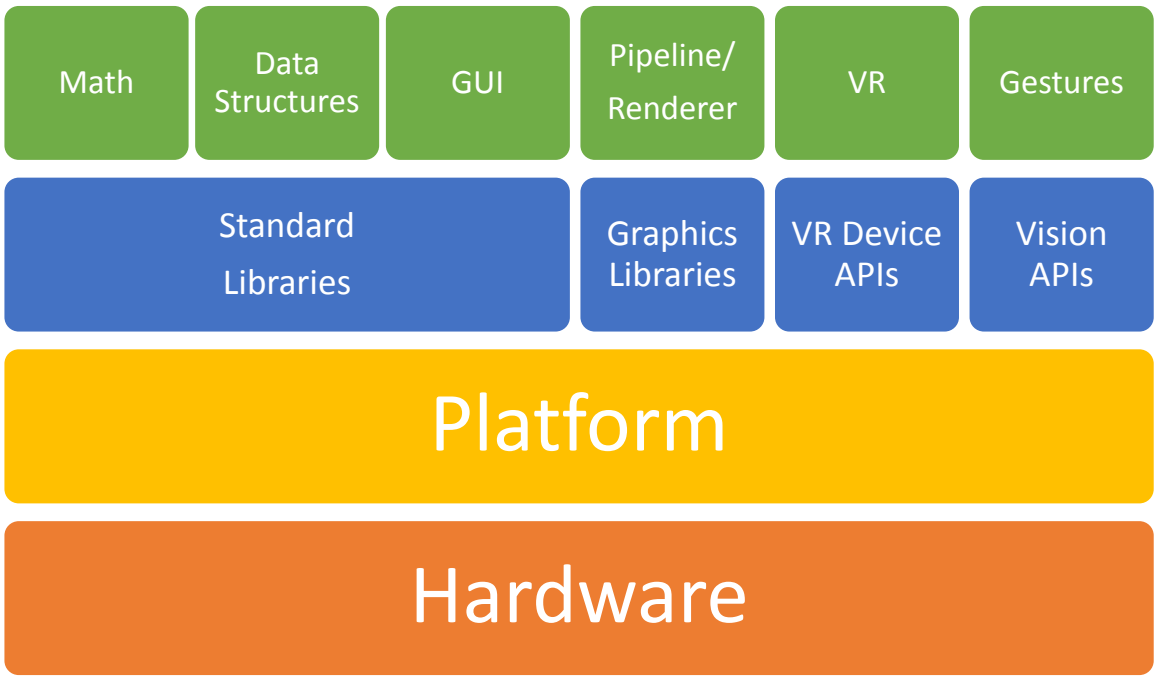
which is ultimately what this project aims to do. The main targets for abstract interfacing within this project are the graphics, HMD and 3D Sensor layers. This is important because it allows the developer to write applications without having to worry about hardware specific issues. This will be explained in further detail in later sections. For now, it is useful to further extend the automobile analogy. For the sake of explanation, we can limit this discussion to cars. All cars have a set of user controls that allow a driver to operate them. There is a steering wheel, an accelerator pedal, a brake pedal, a gearstick and signaling controls. Some cars may have more, but all cars have at least these controls which we can call the basic user interface for a car. This is what allows a single experienced driver to successfully operate a large variety of cars with relative ease without having to learn or master any new controls. The manufacturers of these cars comply with this interface with the understanding that if any of these controls deviated from the standard, drivers may find it difficult and awkward which would ultimately affect sales. Beneath this standard interface, however, the manufacturers are free to do whatever they like. The steering wheel could connect to a simple transmission or an electronic column and the honk control might actuate to an air horn or a speaker but this would make no difference to the driver.

In the same way, when developing a software system for a project such as this, it is important to define a solid, robust and intuitive interface for hardware components or even low level software libraries. The VR engine is like the car and the VR developer is like the driver in the analogy. If an interface proves to be popular amongst developers, there is an incentive for the hardware manufacturers to provide specific implementations of their systems compatible with this interface which in turn allows for developers to produce applications without having to worry about compatibility issues with specific hardware. This process of interface development and manufacturer compliance leads in many cases to standardization, which further pushes the development of VR technology.

3.4. System Architecture

As with any large project, it is useful to describe the system as a whole before delving further into the details. The basic strategy for development will involve breaking the system up into layers starting with the hardware and associated low-level software libraries at the bottom. Every layer above that will abstract away from these low level components to create a clean, developer friendly API at the top. The figure below illustrates a basic representation of this.

Figure 3-1: Engine Hierarchy



The layer that must be implemented in the engine is on the top (green). Each green module must remain relatively independent of the others to maximize developer discretion.

4. Mathematics

The heart of VR, like many other fields of science and engineering, lies in mathematics. It is highly recommended that the reader already be somewhat familiar and comfortable with 3D geometry and linear algebra. In no way is this chapter meant to provide an in-depth understanding of the mathematical structures, operations and formalisms used throughout this thesis. Instead, it serves as a brief overview of the concepts and notation at the core of the engine in addition to some details relating to how these structures will be represented computationally. Concepts specific to this project will be developed further in later chapters.

4.1. Matrices

A matrix, put simply, is a 2-D rectangular array of elements. They are usually denoted by capital letters in many mathematical texts, which is a convention we will be sticking to here as well. Below is a typical representation of a matrix.

$$A = \begin{bmatrix} A_{1,1} & \cdots & A_{1,N} \\ \vdots & \ddots & \vdots \\ A_{M,1} & \cdots & A_{M,N} \end{bmatrix} \qquad 4.1\text{-}1$$

The matrix shown is a $M \times N$ matrix. This means that it has M rows and N columns. The notation $A_{i,j}$ refers to the element of A in the i-th row and j-th column.

There are a few options relating to how a matrix can be stored in memory. The first, and probably most intuitive, is to store the elements of the matrix in a simple 2D array. This is an abstraction provided by many modern programming languages. This is an abstraction because in reality, the elements are stored contiguously just as a regular 1D array would be stored but accessing each element requires two indices. For example, consider the following 2D array of data.

Figure 4-1: 2D Array:

A	B	C
D	E	F
G	H	I

To the programmer, each element can only be uniquely indexed by its specific row and column. In memory, however, the data is arrange more like this.

Figure 4-2: 1D Row Major

A	B	C	D	E	F	G	H	I

This is a perfectly valid way to store a 3x3 matrix, or by extension a $M \times N$ matrix. In fact, when talking about matrices in particular, this arrangement is called the row-major representation of a matrix. The term row-major refers to the fact that the elements of the matrix in the same row are next to each other in memory. In row-major representation, the array in Figure 4-2 represents the following matrix.

$$\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \qquad 4.1\text{-}2$$

The alternate representation is referred to as the column-major representation. In column-major representation, the array in Figure 4-2 represents the following matrix.

$$\begin{bmatrix} A & D & G \\ B & E & H \\ C & F & I \end{bmatrix} \qquad 4.1\text{-}3$$

The engine will be utilizing the column major representation for reasons of compatibility with the OpenGL graphics library.

With a solid grasp on computational representation, we can now define a few useful operations on matrices. First we have addition, which is defined as follows.

$$(A + B)_{i,j} = A_{i,j} + B_{i,j} \qquad 4.1\text{-}4$$

Put simply, the sum of two matrices is a matrix containing the sum of corresponding components. Similarly, subtraction is defined as follows.

$$(A - B)_{i,j} = A_{i,j} - B_{i,j} \qquad 4.1\text{-}5$$

In each case, the code for each operator simply involves iterating through each array and performing the operation element-wise.

You will notice that for these two definitions to make sense, A and B must be the same dimension. In other words, the number of rows and columns should be the same for both.

The next logical step is to define multiplication procedures. There are two kinds of matrix multiplication. The first is called scalar multiplication, and it is pretty straightforward.

$$(cA)_{i,j} = cA_{i,j} \qquad 4.1\text{-}6$$

Here c is a scalar and A is a matrix of arbitrary size. The resulting matrix is formed by multiplying every element of A by c, a scalar. Again, the code involves iterating through the array and performing the multiplication element-wise.

The second kind of multiplication is defined between two matrices of compatible sizes. It is defined like so.

$$(AB)_{ij} = \sum_{k=1}^{R} A_{i,k}B_{k,j} \qquad 4.1\text{-}7$$

Compatible sizes refers to the requirement that the first matrix must have the same number of columns as the second matrix has rows. The resulting matrix will have the same number of rows as the first matrix and the same number of columns as the second.

The Transpose of a matrix results simply in the rows and columns switching places. The transpose will be denoted by a superscript T, and is formally defined as follows.

$$A^T_{i,j} = A_{j,i} \qquad 4.1\text{-}8$$

In terms of size, a matrix with dimensions $M \times N$ will transpose to a $N \times M$ matrix.

4.2. Vectors

In the practical sense, a vector is a quantity that can represent a magnitude and a direction. At this point, we simply define it as a list of elements. Vectors will be represented as column matrices, like so.

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = [v_1\ v_2 \dots v_n]^T \qquad 4.2\text{-}1$$

Unlike with matrices, vectors only require one index when referring to a specific element. The transposed expression shown above is a more convenient representation of a column vector, to save space within this document. The vector shown above may also be referred to as an "n-vector" where "n" is the number of elements, or dimension of the vector.

Just as with any matrix, we can add and subtract vectors of the same size, which simply results in a new vector whose elements are the sums and differences respectively of the corresponding elements of the input vectors.

4.3. Square Matrices

When a matrix has the same number of rows as columns, it is referred to as a square matrix. Most of the matrices that will be used within this thesis are square matrices. Everything defined up to this point in terms of operations and functions on matrices apply to square matrices as well.

The first thing to note is that the result of multiplying two square matrices results in a third square matrix of the same size. Similarly, the transpose of a square matrix is also a square matrix with the same.

These properties make it possible to define a more extensive set of operations, starting with the determinant. It is defined as follows for a $N \times N$ matrix.

$$|A| = \det A = \sum_{i=1}^{N} (-1)^{i+j} A_{i,j} M_{i,j} = \sum_{j=1}^{N} (-1)^{i+j} A_{i,j} M_{i,j} \qquad 4.3\text{-}1$$

Here, $M_{i,j}$ refers to the i,j-th minor of the matrix A. The i,j-th minor of A is the determinant of the $(N-1) \times (N-1)$ formed by eliminating the i-th row and j-th column of A.

The determinant is useful in defining another square matrix operation called inversion. Inversion plays a large part later in this thesis and in the engine as a whole.

$$A^{-1} = \text{inv}\, A = \frac{1}{\det A} adj\ A \qquad 4.3\text{-}2$$

Here, $adj\ A$ represents the adjoint of the matrix A defined in terms of the cofactor matrix C, as follows

$$(adj\ A)_{i,j} = C_{i,j}^{T} = (-1)^{i+j} A_{i,j} \qquad 4.3\text{-}3$$

Also notice that the inverse is undefined if the determinant is zero. In this case, the matrix is said to be non-invertible or singular.

Before the significance of the inverse is discussed, one more definition is required. The identity matrix extends the idea of "one" to matrix algebra. It has the following defining property.

$$IA = AI = A \qquad 4.3\text{-}4$$

As stated, any matrix multiplied by a properly sized identity matrix results in the same matrix. The form of an identity matrix is as follows.

$$I_N = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \vdots \\ \vdots & 0 & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{bmatrix}; I_{i,j} = \delta_{i,j} \qquad 4.3\text{-}5$$

The identity matrix is simply a diagonal matrix with ones along the diagonal. The subscript N represents the size of the identity matrix. Its significance to the inverse operation is shown below.

$$I = AA^{-1} = A^{-1}A \qquad 4.3\text{-}6$$

These concepts can be used to solve basic linear systems of equations of the form, where x and b are column vectors.

$$Ax = b \qquad 4.3\text{-}7$$

The solution for x in this case could be found like so, assuming A is a non-singular, invertible matrix.

$$x = Ix = A^{-1}Ax = A^{-1}b \quad \textit{4.3-8}$$

Similarly, if the equation was as follows, where x and b are row vectors:

$$xA = b \quad \textit{4.3-9}$$

The solution would be

$$x = xI = xAA^{-1} = bA^{-1} \quad \textit{4.3-10}$$

Unlike in regular arithmetic, because matrix multiplication is non-commutative, the multiplication of the inverse matrix must be done in the right order on the correct side. The same concept can be applied to pure matrix equations like

$$AX = B \text{ or } XA = B \quad \textit{4.3-11}$$

In which case, the solutions would take the form

$$X = A^{-1}B \text{ or } X = BA^{-1} \quad \textit{4.3-12}$$

These concepts are central to the development of a 3D spatial representation system, which will be covered in the following sections.

4.4. Points and Spaces

In general "points" and "spaces" can be extremely abstract notions. The term "space" is used very broadly in mathematics and is defined as a set with some added structure. A point on a space is a single, unique element of that set. The dimensionality of a "space" is defined as the smallest number of elements required to uniquely identify a "point" on that space.

For our purposes, "space" simply refers to 3D Euclidian/Geometric space. Under this definition, a point on this space is a unique location which can be represented most simply by a 3-element vector.

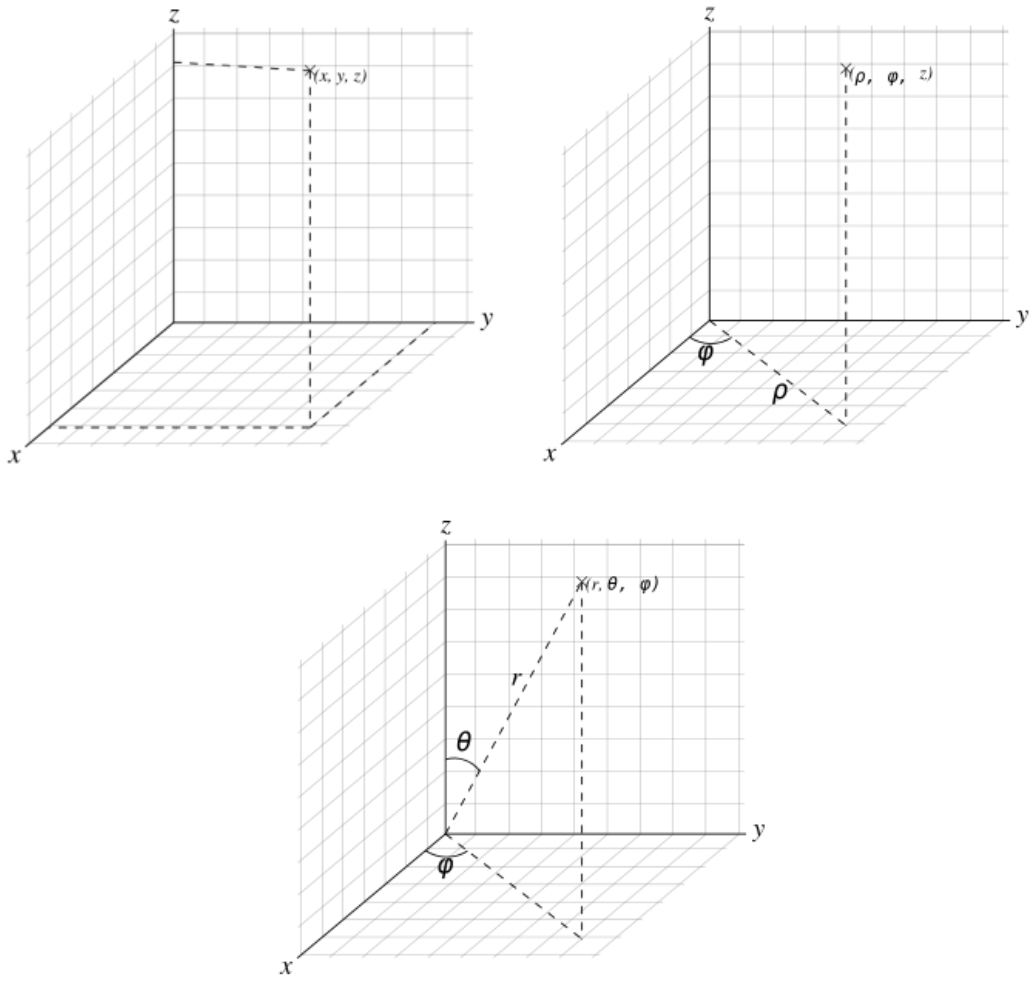
4.5. Coordinate Systems

A coordinate system's main purpose is to give context to a chosen representation of a point in a space. A point in space can only be represented meaningfully if there is an implied and agreed upon system through which information can be extracted from an otherwise meaningless list of numbers.

There are several coordinate systems, the most familiar of which is called the Cartesian or Rectangular coordinate system. In 3D, the Cartesian coordinate system uses three numbers to represent an x, y and z coordinate in space, which in turn represent a fixed distance in each of the three dimensions from a special point known as the origin.

Other common 3D coordinate systems include the Cylindrical and Spherical Systems. The cylindrical coordinate system represents a point in 3D space with a vector containing ρ, ϕ, z components which represent the polar distance from the origin, azimuthal angle and elevation respectively. Similarly, the spherical coordinate system represents a point in 3D space with a vector containing r, ϕ, θ components which represent the distance from the origin, azimuthal angle, and inclination, respectively. The three are shown below. In all three cases, the origin is represented by the zero vector.

Figure 4-3: Coordinate Systems



For the most part, our VR engine will use the right-handed Cartesian coordinate system as the native system, as it is better suited in general for most applications of interest to us. That is not to say however that other systems cannot be used, but it would take some extra work on the part of the programmer or engineer to convert coordinates if there is a need to do so.

4.6. Homogeneous Coordinates

With a firm grasp of traditional coordinate systems, the next logical topic of discussion should be homogeneous coordinates for spatial representation. It may seem like an over complication at first but the benefits will become apparent soon. The concepts in this section are further discussed in detail in [4].

Like before, we will start by defining a point in homogeneous Cartesian coordinates. We will still use a vector, but this time it will have one more element, shown below.

$$v = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \qquad 4.6\text{-}1$$

The first three elements, x, y, z relate to the spatial components of the quantity, just like the non-homogeneous case, but the last element denoted by w above is new. This is the element that allows us to "project" points in Euclidian space to an associated homogeneous space, called projective space. It works like so.

The vector in 4.6-1 maps to 3D space in the following way.

$$v = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \rightarrow v' = \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix} \qquad 4.6\text{-}2$$

Here, v' is a 3-element vector representing a point in 3D space. If $w = 1$, we're right back where we started, with $v' = [x, y, z]^T$ being an ordinary point. Now, let's consider the case where $w \rightarrow 0$. The "point" v' extends out to infinity which for our purposes is somewhat meaningless. Notice however, that even as the components of v' tend towards ∞, the ratios of the components to each other remain the same. No matter how small w becomes, the ratio $x{:}\,y{:}\,z$ does not change. This allows us to interpret the resulting quantity not as a point, but a direction in 3D space.

The notation is pretty straightforward, but it is somewhat limiting due to the fact that as of now, we can only express these coordinates in terms of one frame of reference. A vector representing a point or direction will have different component values depending on what the frame of reference is. In other words, the vector representing a point or direction is entirely based on our selection of the origin and axes. Being able to transform coordinates between frames of reference is an extremely useful and powerful tool.

Fortunately, the ability to describe points and directions so concisely with vectors gives us the ability to describe a frame of reference with respect to another without having to store location and orientation information in separate structures. A Cartesian frame can be thought of as three axes and an origin. These quantities can be represented by a row of three unit direction vectors plus one position vector to form a 4x4 matrix like the one shown below.

$$F = {}^{W}X_{F} = \begin{bmatrix} \hat{u}_x & \hat{v}_x & \hat{n}_x & p_x \\ \hat{u}_y & \hat{v}_y & \hat{n}_y & p_y \\ \hat{u}_z & \hat{v}_z & \hat{n}_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad 4.6\text{-}3$$

In this paper, the axis vectors will be denoted by $\hat{u}, \hat{v}, \hat{n}$ and the origin, or location of the frame, will be denoted by p. The hats indicate that vectors are unit vectors. The $\hat{u}, \hat{v}, \hat{n}$ axes correspond to, and can be thought of respectively as the $\hat{x}, \hat{y}, \hat{z}$ axes of the frame. Common sense dictates that a free frame in 3D space has six degrees of freedom…three for position and three for orientation. The matrix above has 12 elements, which is inconsistent with common sense. We must introduce a few constraints.

$$|\hat{u}| = |\hat{v}| = |\hat{n}| = 1 \qquad 4.6\text{-}4$$

$$\hat{u} \cdot \hat{v} = \hat{v} \cdot \hat{n} = \hat{n} \cdot \hat{u} = 0 \qquad 4.6\text{-}5$$

The constraints in 4.6-5 can be written more compactly as

$$\hat{u} \times \hat{v} = \hat{n} \qquad 4.6\text{-}6$$

In reality, all this means is that the vectors that represent the axes of the frame must be unit vectors and they must be orthogonal in a right-handed sense. Notice that there are no constraints placed on p indicating that a frame can be placed anywhere in 3D space.

The superscript in the 4.6-3 indicates that the values contained within the matrix are in reference to a W frame. Unless explicitly specified, the components of a frame are always expressed in the coordinates of the "world" or "universe" frame, denoted by W. This notation can be applied to points and vectors as well.

$$p = {}^{W}p = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}, v = {}^{W}v = \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix} \qquad 4.6\text{-}7$$

4.6-7 is an example of quantities expressed in world coordinates with both implicit and explicit notation.

The next step is to explore a case where the explicit frame of reference is not the world frame. That is where transformations come in.

4.7. Transformations

In the mathematical sense, a transformation is a function that maps points from once space to another. In the geometric sense, we can think of a transformation as a way to move a point or set of points through space. There are many types of transformations out there, but only a few of them will be useful to us in this context. At the moment, there are only two types of transformations we care about: translations, and rotations.

A translation can be thought of as a simple move. Every point is moved through space by the same amount in the same direction. In a non-homogeneous Cartesian system, this type of transformation is easily realized with vector addition.

$$p' = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} p_x + \Delta x \\ p_y + \Delta y \\ p_z + \Delta z \end{bmatrix} \quad \textit{4.7-1}$$

Here, p is an arbitrary point being translated by an arbitrary vector. The result, p' is simply the sum of the two vectors. In homogeneous coordinates however, we can represent a translation as a 4x4 matrix, shown below.

$$T(\Delta x, \Delta y, \Delta z) = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \textit{4.7-2}$$

Before we decide if this is a good idea, we must verify that it works.

$$p' = T(\Delta x, \Delta y, \Delta z)p$$

$$p' = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x + \Delta x \\ p_y + \Delta y \\ p_z + \Delta z \\ 1 \end{bmatrix} \quad \textit{4.7-3}$$

It works! Multiplying a point by the translation matrix resulted in a new point whose coordinates are shifted by the specified amounts. Now let's see what happens when the transformation is applied to a direction.

$$v' = T(\Delta x, \Delta y, \Delta z)v$$

$$v' = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix} \quad \textit{4.7-4}$$

As is expected (and required), a translated direction vector is still a direction vector pointing the same direction. This homogeneous representation of translation not only produces the correct results, but also seems to take into account the type of vector it is being applied to!

The inverse of a translation matrix is another translation matrix with the parameters negated.

$$T^{-1}(\Delta x, \Delta y, \Delta z) = T(-\Delta x, -\Delta y, -\Delta z) = \begin{bmatrix} 1 & 0 & 0 & -\Delta x \\ 0 & 1 & 0 & -\Delta y \\ 0 & 0 & 1 & -\Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad 4.7\text{-}5$$

This is necessary because a translation followed by the inverse of the same translation should result in no translation at all. The obvious consequence of this is the following

$$I = T(\Delta x, \Delta y, \Delta z)T^{-1}(\Delta x, \Delta y, \Delta z) = T(0,0,0) \quad 4.7\text{-}6$$

Next, we must define a rotation. One way to characterize a rotation is with an axis known as the axis of rotation and an angle known as the angle of rotation. None of the points lying directly on the axis of rotation must move while every other point revolves around the axis of rotation by the specified angle. In 3D space, a rotation can be represented by a 3x3 matrix. Given an arbitrary axis, $\hat{v}$ and an angle θ, a rotation matrix is defined like so.

$$R(\hat{v}, \theta) = R_{\hat{v}}(\theta)$$

$$= \begin{bmatrix} \mathrm{C}\,\theta + \hat{v}_x^2(1 - \mathrm{C}\,\theta) & \hat{v}_x\hat{v}_y(1 - \mathrm{C}\,\theta) - \hat{v}_z\,\mathrm{S}\,\theta & \hat{v}_x\hat{v}_z(1 - \mathrm{C}\,\theta) + \hat{v}_y\,\mathrm{S}\,\theta & 0 \\ \hat{v}_y\hat{v}_x(1 - \mathrm{C}\,\theta) + \hat{v}_z\,\mathrm{S}\,\theta & \mathrm{C}\,\theta + \hat{v}_y^2(1 - \mathrm{C}\,\theta) & \hat{v}_y\hat{v}_z(1 - \mathrm{C}\,\theta) - \hat{v}_x\,\mathrm{S}\,\theta & 0 \\ \hat{v}_z\hat{v}_x(1 - \mathrm{C}\,\theta) - \hat{v}_y\,\mathrm{S}\,\theta & \hat{v}_z\hat{v}_y(1 - \mathrm{C}\,\theta) + \hat{v}_x\,\mathrm{S}\,\theta & \mathrm{C}\,\theta + \hat{v}_z^2(1 - \mathrm{C}\,\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad 4.7\text{-}7$$

$$C = \cos\theta, S = \sin\theta$$

Again, it is important to stress that the axis must be in the form of a unit vector for this to work. An extra row and column is added to make the transformation homogeneous. Notice that this time, the column vectors of the matrix represent directions. The last column is zero because there is no uniform translation occurring.

If we know our axis of rotation in advance, it is wise to define specific rotation matrices to avoid the obvious cost of computing the matrix above. We can define rotation matrices that represent rotations explicitly about our Cartesian axes.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad 4.7\text{-}8$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad 4.7\text{-}9$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad 4.7\text{-}10$$

Rotation matrices in particular have an interesting property. All rotation matrices are unitary which means that they can be inverted by simply transposing them, which is equivalent to negating either the axis of rotation or the angle (but not both).

$$R^{-1}(\hat{v}, \theta) = R(-\hat{v}, \theta) = R(\hat{v}, -\theta) = R^T(\hat{v}, \theta) \quad 4.7\text{-}11$$

Also, as expected, a rotation of zero is equivalent to the identity matrix.

$$I = R^{-1}(\hat{v}, \theta)R(\hat{v}, \theta) = R(\hat{v}, 0) \quad 4.7\text{-}12$$

Another important thing to note is that because matrix multiplication is non-commutative, changing the order of multiplication can dramatically change the nature of the transformation.

The final transform needed is known as a scaling transform. Its only purpose is to stretch or squeeze coordinates along the specified axes. It will be denoted here as S, and is defined like so.

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad 4.7\text{-}13$$

The values s_x, s_y, s_z are simple scaling factors which result in a squeeze if their absolute value is less than 1, or a stretch if greater. If the factors are negative, the axis inverts and the frame essentially flips in that dimension.

4.8. Transforming Between Frames

It is interesting to note that all of the transformation matrices so far except for the scaling transform would fit our definition of a frame. The first three vectors in the translation and rotation matrices follow the rules outlined in 4.6-4 and 4.6-5. In fact, the product matrix of a translation and a rotation looks a lot like the frame in 4.6-3. This is not surprising considering the fact that any frame can transformed to another with a single translation to line up the origins and a rotation to line up the axes. This is an important fact; one that requires an update in notation, shown below.

$$F = {}^{W}F = {}^{W}X_F = \begin{bmatrix} \hat{u}_x & \hat{v}_x & \hat{n}_x & p_x \\ \hat{u}_y & \hat{v}_y & \hat{n}_y & p_y \\ \hat{u}_z & \hat{v}_z & \hat{n}_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = T(p_x, p_y, p_z)R \qquad 4.8\text{-}1$$

The ${}^{W}X_F$ in the equation above represents the transformation matrix that transforms between F and W, where W represents a predefined "world" frame. The X simply indicates that the quantity is a transformation matrix. In this case, because W is the world frame, the matrix representation of F with respect to the world is the same as the matrix representation of the corresponding transform. The transform is simply composed of a rotation followed by a translation along the axes of the reference frame, which in this case is the world frame.

This new notation gives us a way to intuitively change basis between multiple frames of reference. For example, let's assume that we have two frames: M and C. M is the local frame of reference for an arbitrary object and C is the local frame of reference of an observer. If M is located at a point $[1, 2, 3]^T$ with respect to the world and then rotated $90°$ about its own n axis, its transformation would look like this.

$$M = {}^{W}X_{M} = T(1,2,3)R_{z}(90°) = \begin{bmatrix} 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad 4.8\text{-}2$$

Note the order of transformations here. The translation is performed first and is in reference to the world frame. The second transform is a rotation with respect to the n axis of the translated frame, which is why it is multiplied on the right. If the rotation was in reference to the z axis of the world frame, it would have been multiplied on the left. Additionally, let's assume that there is a point p at the origin of the M frame. We can express this statement as follows.

$$^{M}p = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad 4.8\text{-}3$$

Finally, the frame C is located 3 units above the world frame with no rotations.

$$C = {}^{W}X_{C} = T(0,0,3) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad 4.8\text{-}4$$

If we wanted to find what p looks like from the W frame, all we would have to do is align the transformations to get an expression for $p = {}^{W}p$.

$$p = {}^{W}p = {}^{W}X_{M}{}^{M}p = \begin{bmatrix} 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \end{bmatrix} \quad 4.8\text{-}5$$

We can cross cancel the subscript and superscript M leaving us with an expression for p in the world frame. The answer makes sense because p is located at the origin of M which in turn is located at $[1,2,3]^{T}$. Now let's consider what an observed in frame C would see in frame M. This time, we want an expression for ${}^{C}M = {}^{C}X_{M}$.

$${}^{C}M = {}^{C}X_{M} = {}^{C}X_{W}{}^{W}X_{M} \quad 4.8\text{-}6$$

At first glance, there seems to be an error. There doesn't seem to be a matrix for ${}^{C}X_{W}$. All we have is ${}^{W}X_{C}$. Well, that is where the inverse operation comes in again.

You might recall that the inversion process for a generic 4x4 matrix is a pretty complex affair. The good news is that homogeneous 4x4 matrices that represent frames are easier to invert and require a smaller number of computations to achieve.

$$F = \begin{bmatrix} \hat{u}_x & \hat{v}_x & \hat{n}_x & p_x \\ \hat{u}_y & \hat{v}_y & \hat{n}_y & p_y \\ \hat{u}_z & \hat{v}_z & \hat{n}_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow F^{-1} = \begin{bmatrix} \hat{u}_x & \hat{u}_y & \hat{u}_z & -\hat{u} \cdot p \\ \hat{v}_x & \hat{v}_y & \hat{v}_z & -\hat{v} \cdot p \\ \hat{n}_x & \hat{n}_y & \hat{n}_z & -\hat{n} \cdot p \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad 4.8\text{-}7$$

Inverting a transform will flip the superscript and subscript.

$$\begin{aligned} {}^{C}M \rightarrow {}^{W}X_{C}^{-1}{}^{W}X_{M} &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \qquad 4.8\text{-}8$$

Again, no surprises here. From C, M looks just like it did before, but since C is at the same height as M, M appears not to have a relative z offset.

These types of manipulations form the basis of the 3D rendering system.

5. Graphics

Modern VR systems should be capable of realistic 3D rendering to keep the user immersed in the experience. The goal of a VR engines, or any CG engine for that matter, is to convince a user that what they are seeing on the screen is not merely a 2D array of pixels, but a window into a 3D world. The purpose of this chapter is to expose some of the tricks used by graphics programmers to fool the brain into thinking exactly that. This chapter will also discuss the state of modern graphics hardware, and how it is utilized in the engine. The following sections require a strong grasp of the mathematical concepts covered in the previous chapters. Much of the information for this part of the project came from [5].

5.1. The Graphics Processing Unit (GPU)

Modern high performance is impossible without the dedicated GPU. While the calculations that make 3D graphics possible run fine on a traditional CPU, the sheer volume of geometrical data that must be processed in a modern graphics application makes real time rendering impossible. The dedicated graphics chip approaches the task from a different point of view. Instead of piping all of the data through the few cores of a powerful modern CPU for processing, the GPU makes it possible for large scale multiprocessing, which means that a large number of relatively simple transformation calculations can be done in parallel resulting in faster rendering.

5.2. Low-Level Graphics Libraries

The modern graphics card is for all intents and purposes a computer within a computer. It has its own memory resources and processing unit. Special programs called **shaders** can be written, compiled, linked and executed directly on the card. These programs are written in a simple, C-like language specially outfitted to perform transformation and interpolation tasks. These programs take large lists of vertices that must be stored on the card to minimize latency between the data and the GPU. For this purpose, high speed memory and caches are built in to these cards. Low level graphics libraries allow applications running on the CPU to interact with these resources. The programs are written with shaders, and memory is accessed and addressed with structures called **buffers** and **textures**. These bindings allow for huge customizability in graphics applications but because these libraries provide such low level access, it can often be tedious to perform even the simplest of tasks. It is for this reason that the graphics module is designed in essence as a two tier system. The first tier simply abstracts the resources as graphics independent classes while the second tier aims to build useful interfaces and classes with them.

5.3. OpenGL vs Direct3D

One cannot discuss Low-Level Graphics Libraries without discussing the two major libraries currently in use. The first is called OpenGL and it was originally started by Silicon Graphics in the

early 1990s. The intent was to create an open standard for all forms of cross platform graphics applications. Today, it is used in CAD, games, media players, visualization and much more. Soon after, Microsoft decided to release their own library. They called it DirectX and it was meant primarily for gaming. The 3D Graphics module was called Direct3D. DirectX is a proprietary API that only runs on Windows PC and Xbox systems. Today, Direct3D is used mostly in gaming.

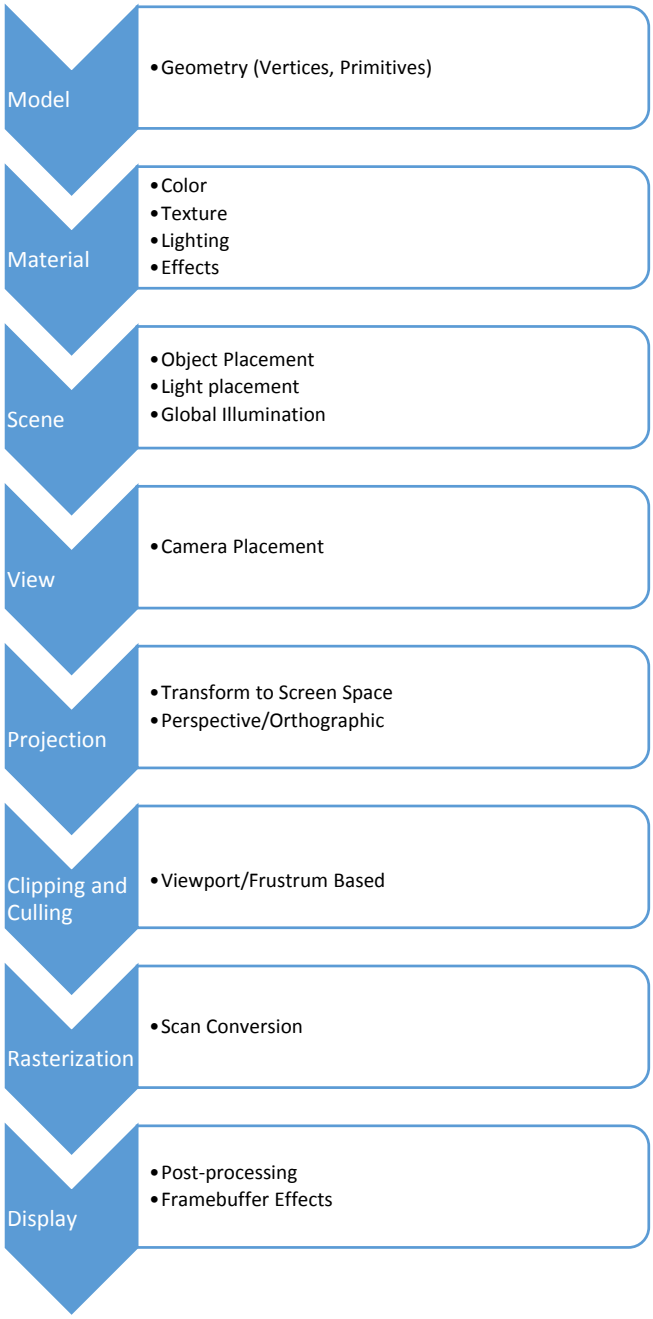
As graphics hardware improves, the libraries must be updated to make newer features accessible to the developer.

5.4. The Rendering Pipeline

The Rendering Pipeline is a term that refers to the sequence of steps employed to produce a rasterized 2D image representation of a 3D scene. The traditional pipeline can be presented in the form of the following flowchart on the right. Each stage in the process hands off its results to the next stage until the scene is presented on screen.

In the younger days of computer graphics, the pipeline referred simply to the process of taking 3D polygon data out of memory, calculating the corresponding point in screen space directly on the CPU, and setting the value of the corresponding pixel on the monitor. Today, with the advent of newer technologies and dedicated video cards (aka GPUs), a rendering pipeline has evolved into a substantially more complicated process. The following sections will explain each stage.

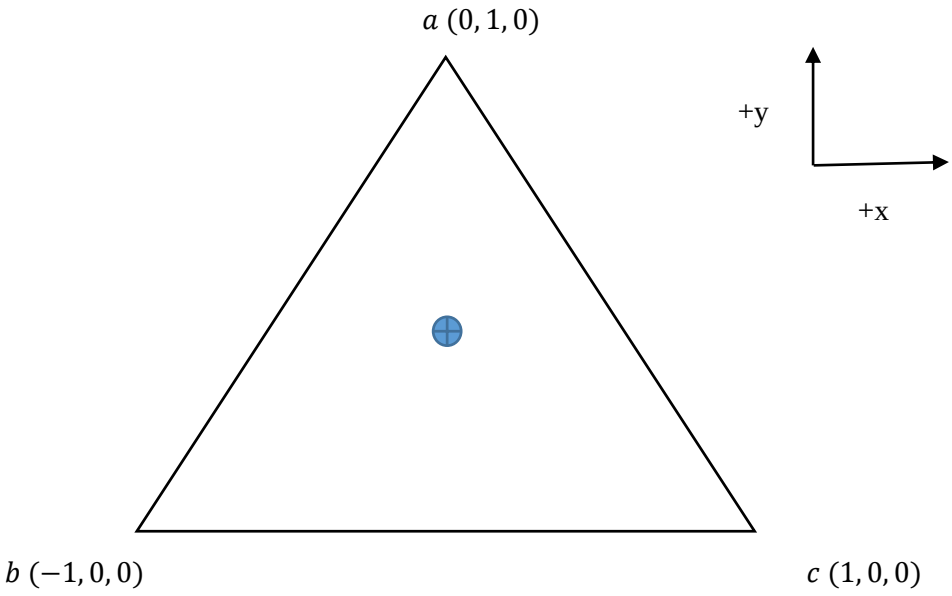
Figure 5-1: Simplified Rendering Pipeline



5.5. Model

The first step in the pipeline is the Model Representation. A model can be thought of most simply as a collection of points in space coupled with some information/context about how those vertices relate to each other. Typically, the points are referred to as vertices and the context is conveyed in the form of vertex order and primitive data. Let's consider the example of a triangle. Shown below is a simple isosceles triangle with its vertices and corresponding coordinates labeled. The drawing is not to scale.

Figure 5-2: A Simple Triangle



The model is characterized by a list of vertices. Each vertex is defined in what is known as model space. In the case of the triangle above, each vertex is defined in terms of a local origin, which is designated at the center of the triangle. Using the notation developed earlier, we can express this as follows.

$$^{M}a = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}, ^{M}b = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, ^{M}c = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad 5.5\text{-}1$$

Each vertex is defined in terms of the homogeneous vector that expresses its position within the frame of the model. The frame M represents the frame of the model in world space. The matrix that represents this transformation is known as the model matrix.

$$M = {}^{W}X_{M} \qquad 5.5\text{-}2$$

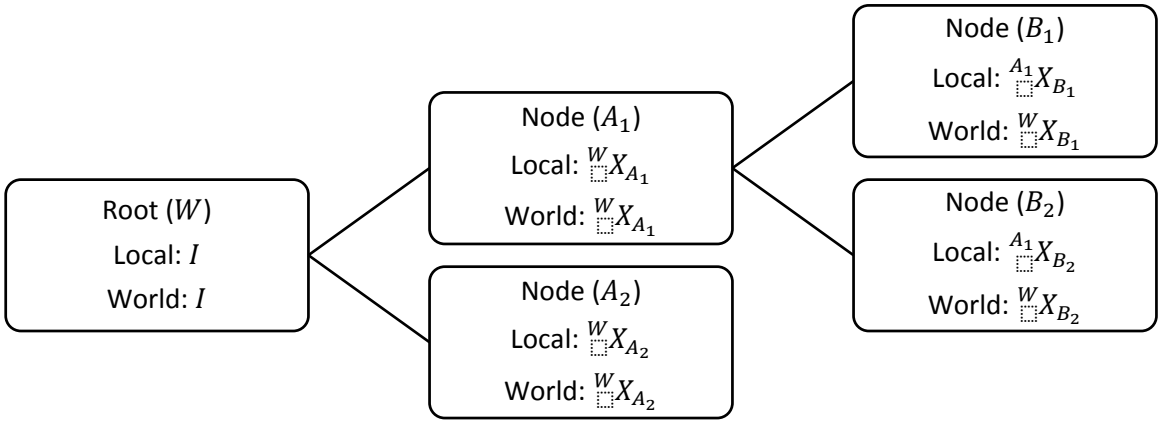
5.6. Material

Depending on where you look, this word may mean multiple things. As far as this paper is concerned, the word material refers to the presentation characteristics of the models. The reason this step is separate to the modeling step is because one model can be used to represent multiple entities within the engine by simply using a different material. For example, a cube model can represent a crate or an ice cube and all it would take is a different texture or effect. This approach saves memory since the geometric data isn't being stored redundantly.

5.7. Scene

Once we have a set of models and materials, we have all we need to specify entities within our virtual world. The only thing left to do is place them in whatever arrangement we see fit. This arrangement is called the scene. Within the engine, the scene is represented as a tree-like structure. Each node of the tree contains the homogeneous 4x4 matrix that transforms from the space of the parent to the space of the node. In other words, the nodes basically represent homogeneous frames. The root node represents the world frame, with an Identity transform matrix.

Figure 5-3: Scene Tree



The Figure above shows the basic structure of a scene tree. Each node contains a local transform matrix and a world transform matrix. The local matrix is the transform that relates a node to its parent, and the world matrix stores the absolute transform.

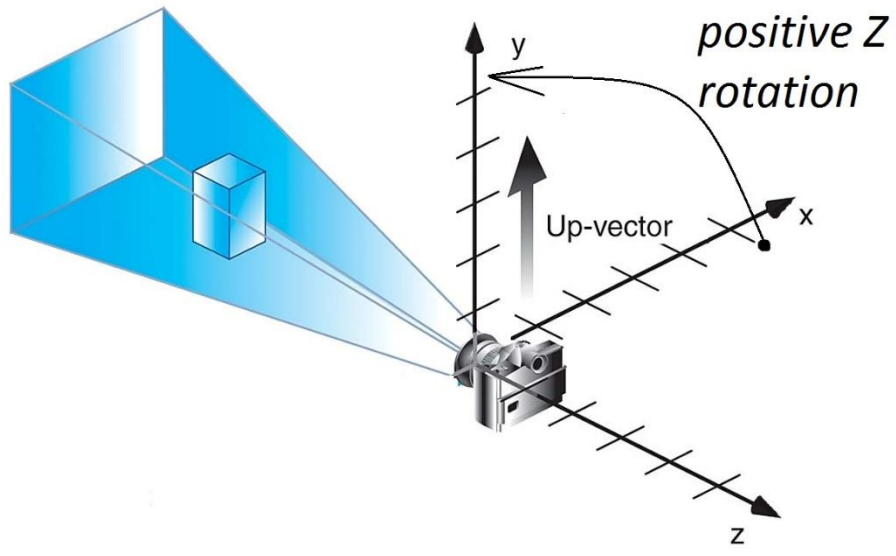
It is useful at this point to introduce a simple analogy. Suppose that a photographer wants to set up a scene and take a picture. He has a room containing a table and a few random objects to arrange on the table to set up his scene. First he puts the table in place and arranges the objects randomly on top of the table. He checks his shot and finds that some of the objects are blocked by others, so he goes in and modifies the arrangement to remedy the occlusion. He checks the shot again and finds that the table is not centered in the view, so once again he goes over and moves the table without moving any of the objects on top of it. In terms of the tree structure just introduced, the table is the only child node of the scene root, which in this case is the room. The other objects on the table are children of the table node. The first adjustment he makes simply results in the modification of the local transforms of the objects on the table with respect to the table. The second adjustment modifies the local transform of the table with respect to the room, which despite affecting the world transforms of the objects, does not affect their local transforms with respect to the table.

The convenience of this structure becomes even more apparent when compound objects are defined. For example, a car might be made up of multiple different parts, each one represented by a separate model in the engine. If each part was placed independently, moving the car would require moving each part independently by the same amount. It would be a pain to manually iterate through every part. With the tree structure on the other hand, moving the car would simply mean changing the local transform of the top-most car node. Since the transforms of all child nodes are in relation to the top node, they would be automatically moved the same amount.

5.8. View

At this point, we have a scene that is ready to be "viewed". The whole point of this section is to figure out what the scene looks like from the point of view of an observer. If we go back to the analogy from the previous section, the view is what the camera sees. In order to determine mathematically what the camera is seeing, we need to know where it is positioned and how it is oriented in the world as well. The coordinate system for the camera must be well defined before we can specify its orientation. In this paper, the $\hat{x}$ axis points to the right of the camera, $\hat{y}$ points to the top of the camera and $\hat{z}$ points behind the camera.

Figure 5-4: Conventional Camera Coordinates



The definition of the scene tree in the previous section gives us a convenient way to represent the frame of a camera along with the rest of the objects in the scene. Refer back to the example in Section 4.8. For each object in the scene whose absolute world transform is given by the expression ${}^{W}X_{M}$ where M is the frame of the object, the coordinates in terms of camera space are given by ${}^{C}X_{M} = {}^{C}M = {}^{W}X_{C}^{-1}{}^{W}X_{M}$. The matrix that transforms an arbitrary point from model space to camera space is ${}^{W}X_{C}^{-1}$. This specific matrix will be called the view matrix, denoted by V.

$$V = {}^{C}X_{W} = {}^{W}X_{C}^{-1} \qquad (5.8\text{-}1)$$

Note that by adding the camera to the scene as a node, all that needs to be done to extract the view matrix is to take the inverse of the absolute world transform of the camera itself.

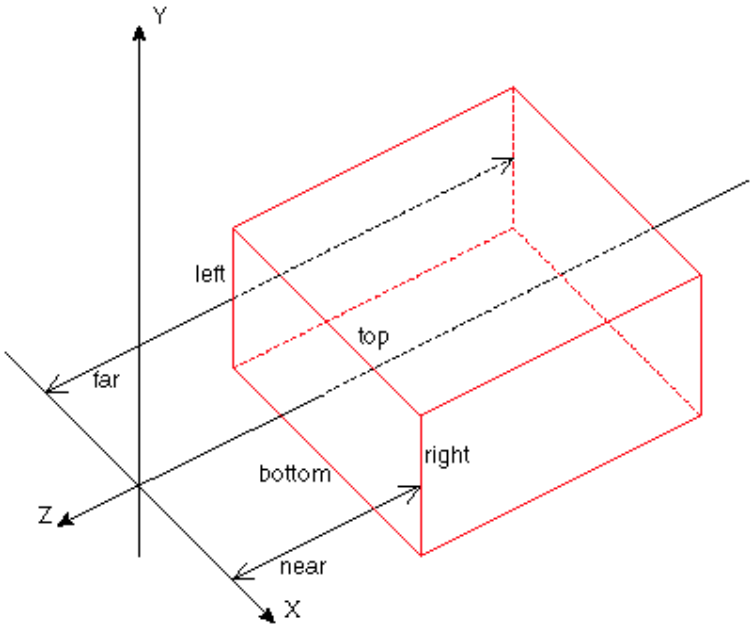
5.9. Projection

Once the view matrix of the camera is determined, the next step is to transform the vertices from view space to screen space. This projection stage is not as straightforward as the others because it involves transforming from a 3D frustum onto a 2D plane. Once again, referring to the photography analogy, just as the placement of the camera determines the view matrix, the lens system of the camera determines the projection.

The two types of projections most commonly dealt with are the orthographic projection and the perspective projection. The orthographic projection is used in applications in which judging alignment is more important than providing a sense of realism. 3D CAD Software, for example, renders models and working parts in the orthographic (aka parallel) projection. The engine does not make use of the orthographic projection for its 3D applications, but it is still useful for putting things flush against the user's screen such as HUD indicators or text.

It is important to keep in mind that the vertex coordinates coming into this stage are with respect to the frame of the camera. The purpose of the orthographic projection is essentially to collapse the incoming vertex data directly into a 2D plane.

Figure 5-5: The Orthographic Frustum



The function that will calculate the orthographic projection in the engine will take six arguments that correspond to the six faces of a prism within which the transformed scene will be centered and stretched. That is, given the positions of a left, right, top, bottom, near and far clipping planes, the function will calculate a matrix that will perform this projection. Anything outside of this volume will not be rendered to the screen. To mathematically derive the orthographic projection transform, it is best to break it down into two simpler ones: a translation followed by a scaling transform.

The translation is required to bring the view into the center of the clipping prism, and the scaling transform will then stretch or compress it to "fill" the cube.

$$T = \begin{bmatrix} 1 & 0 & 0 & -\frac{x_{left} + x_{right}}{2} \\ 0 & 1 & 0 & -\frac{y_{top} + y_{bottom}}{2} \\ 0 & 0 & -1 & \frac{z_{far} + z_{near}}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \textit{5.9-1}$$

$$S = \begin{bmatrix} \frac{2}{x_{right} - x_{left}} & 0 & 0 & 0 \\ 0 & \frac{2}{y_{top} - y_{bottom}} & 0 & 0 \\ 0 & 0 & \frac{2}{z_{far} - z_{near}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \textit{5.9-2}$$

The product of these two matrices results in what is known as the orthographic projection transform.

$$P_{Ortho} = \begin{bmatrix} \frac{2}{x_{right} - x_{left}} & 0 & 0 & -\frac{x_{right} + x_{left}}{x_{right} - x_{left}} \\ 0 & \frac{2}{y_{top} - y_{bottom}} & 0 & -\frac{y_{top} + y_{bottom}}{y_{top} - y_{bottom}} \\ 0 & 0 & \frac{-2}{z_{far} - z_{near}} & -\frac{z_{far} + z_{near}}{z_{far} - z_{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \textit{5.9-3}$$

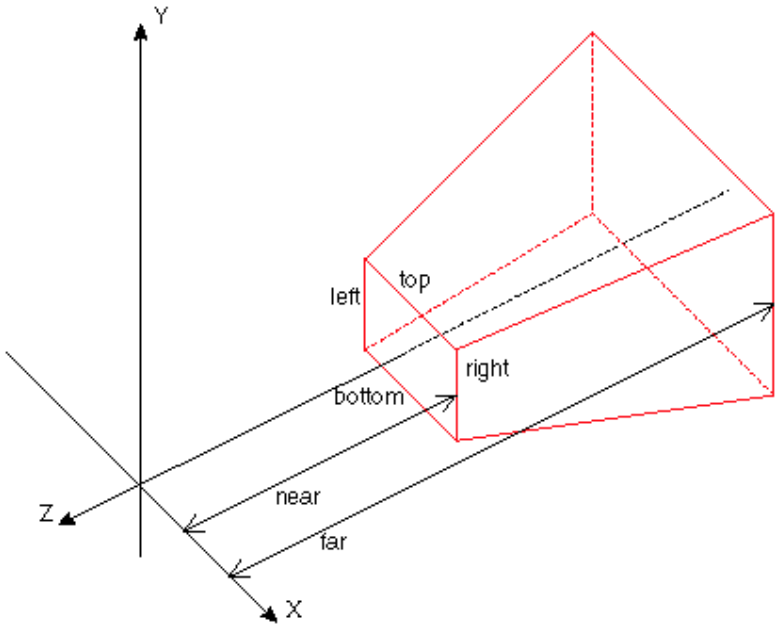
While the orthographic projection is a good choice for putting text on a flat screen or judging alignment in a scene, it is not the best option when modelling a real camera. For that, the perspective projection is required.

The perspective projection is something we are all used to. It is the projection that our eyes see more or less. The effects of "perspective" are well known to artists and photographers, because it is a necessary component of visual realism. The first major effect of the perspective projection is

what is commonly referred to as a vanishing point. The vanishing point is the point in a view where parallel lines in the scene seem to converge to. This is also related to the phenomenon that makes distant objects appear smaller than those that are closer. The second effect of the perspective projection is the apparent difference in the "speed" of objects moving through the field of view based on their respective distances from the observer. More specifically, motion in objects close to the observer is far more noticeable that motion of objects that are far away. These effects, and others are described further in [6].

It is difficult to explain the concepts behind the perspective projection without first explaining the concept of a view frustum. In geometry, a frustum is defined simply as the solid formed by clipping another solid between two parallel planes. Although it seems arbitrary, the frustum is quite useful in describing the field of view of the camera. Figure 5-6 shows a helpful visualization of the view frustum of a typical camera, which looks like a pyramid with the top cut off. The red volume contains everything that is within the field of view of the camera, which is located at the origin where the pointy part of the pyramid would have been, had it not been clipped.

Figure 5-6: Perspective Frustum



Mathematically, the perspective projection can be expressed as follows.

$$P_{Pers} = \begin{bmatrix} \frac{1}{a \tan \frac{\phi_{fov}}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan \frac{\phi_{fov}}{2}} & 0 & 0 \\ 0 & 0 & \frac{z_{near} + z_{far}}{z_{near} - z_{far}} & -1 \\ 0 & 0 & \frac{2 z_{near} z_{far}}{z_{near} - z_{far}} & 0 \end{bmatrix} \quad 5.9\text{-}4$$

The aspect ratio a and field of view (in radians) ϕ_{fov} must be known ahead of time in order to compute this matrix. The aspect ratio is calculated by dividing the width of a viewport by its height. A more complete derivation of this matrix can be found in Chapter 5.

Typical values for the field of view in a standard, non-stereoscopic application typically range between 45° and 90°. A smaller field of view results in a more "zoomed in" view whereas a large field of view results in a more "fisheye" view.

5.10. Clipping & Culling

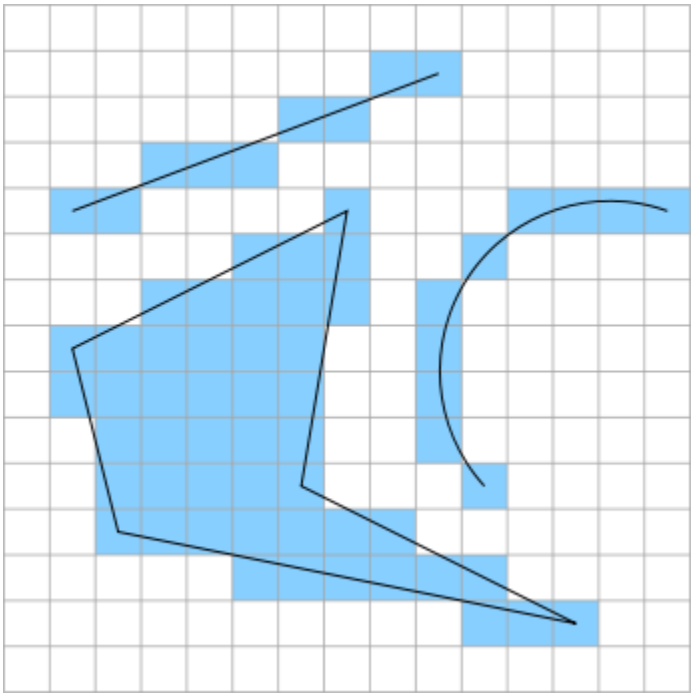
Once the view and projection has been determined, clipping and culling procedures are used to reduce the number of required rendering computations and draw operations performed by the GPU. This is done by first testing to see if vertices will end up on screen after the prescribed transformations. Only the vertices that the user will end up looking at will need to be rendered. The term culling is in actuality a wider term referring to a number of different tests for vertex visibility. Frustum culling, for example refers to the test of whether or not a vertex exists within the frustum of the camera. Clipping, on the other hand, refers to the process of handling polygons that may not be completely visible on screen. For example, a triangle rendered at the edge of the screen with its top vertex occluded should look more like a trapezoid, which has four vertices instead of the three defined in the triangle model. Clipping procedures are used to add these vertices in. Many of these procedures do not have to be defined explicitly within the engine, because the lower graphics layers

typically handle most of it automatically. Additional functionality on this front can be defined by the developer, however.

5.11. Rasterization

The last step in the pipeline process is known as rasterization, which refers to the process of converting images in a vector graphics format and converting it into pixels so that the image may be stored, displayed or printed. The term raster graphics is sometimes used interchangeably with the term bitmaps, as they are essentially the same thing. The figure below is a good visualization of what this process entails.

Figure 5-7: Rasterization



The black outlined figures represent vector graphics elements, which are defined mathematically. The grid shown represents the pixels on the screen. The decision on whether to turn a given pixel on or off and at what intensity given the underlying structure is made by the graphics card for every frame rendered. This bitmap data is then stored in GPU buffer memory.

5.12. Display

The final stage of the rendering pipeline simply involves taking the data from the rasterization stage and posting it to the screen. In modern graphics systems, however, we have the ability to render the rasterization data to other render targets. A render target in most cases is just another patch of memory in the GPU which can be thrown back into the pipeline for further processing. This is more commonly referred to as post processing, and involves performing digital image processing operations on the incoming bitmap images.

6. Stereoscopic Vision

The most common association with VR is the use of stereoscopic head mounted displays (HMDs), which allow for distinct images to be displayed to each eye. In the past, HMDs were large and typically mounted to a much larger chassis preventing users the range of motion that is enjoyed by users of modern variants. This chapter will outline the development of the stereoscopic vision system for the engine and explain some of the key concepts that make it possible.

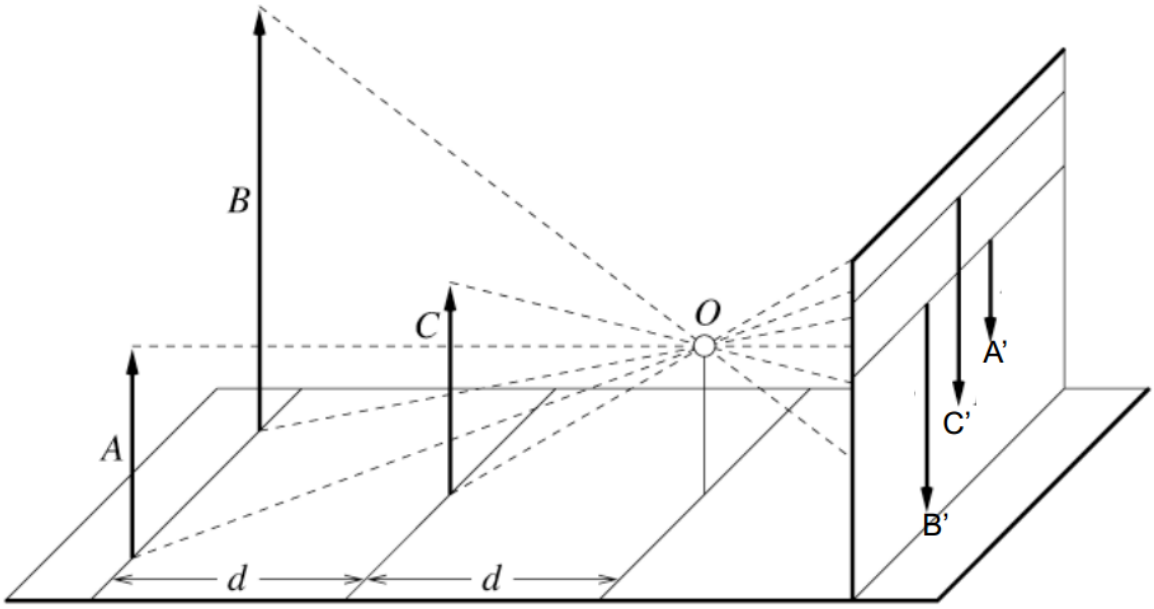
6.1. Pinhole Model

The pinhole model represents the simplest imaging system possible. For most graphics applications, the eye or camera is modeled based on this model. In fact, the perspective projection described in Chapter 4 is derived assuming an idealized pinhole camera. The most important part of a pinhole camera is, not surprisingly, the pinhole.

The Figure below is a basic representation of how pinhole imaging works. Light rays coming off of an object can only enter through the hole. In the limit where the hole is infinitesimally small, each point on the object is represented by the one ray that could make it through the hole, which results in a perfectly focused, inverted image on the back plane of the box.

This model also clarifies why objects that are further away tend to look smaller. Figure 6-1 shows three objects being projected onto a back screen through the pinhole at O. The objects do not remain proportionally sized with respect to each other.

Figure 6-1: Pinhole Projection [7]



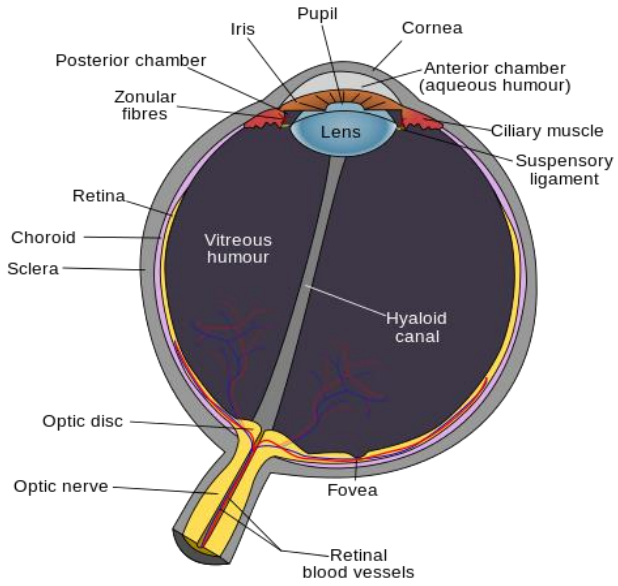
The relationships between the sizes and positions of the objects in the projected image in terms of where the objects are placed in relation to the pinhole can be expressed geometrically. See Section 5.9 for a more in-depth look at the math behind projection.

6.2. The Eye

An accurate model of the eye is crucial to understanding the relatively complicated process of 3D vision in humans. This section will briefly explain how the eye works and define a few terms that will be used throughout the chapter.

Eyes are one of the most intricate and complex organs that human beings possess. For most of us, vision is the predominant sense making the eye the most valuable of the sense organs. Figure 6-2 shows a basic diagram of the human eye.

Figure 6-2: Anatomy of the Eye [6]



If we are only concerned about the parts of the eye which deal directly with the formation of images, we should first look at the pupil. The pupil is the name given to the opening in the iris. The iris is just a ring of muscle that controls the size of this opening, to regulate the amount of light entering

the eye. Just behind the iris is the lens, which serves the same purpose as most other lenses which is to focus the incoming light onto the retina.

Unlike the pinhole system from the previous section, the human eye relies on a flexible lensing system to focus images. The lens is surrounded by fibrous muscles known as ciliary muscles that contract and expand to change the effective focal length of the lens. This gives us the ability to adjust our focus based on the distance to an object. This also gives rise to the "depth-of-focus" effect which results in a sharp focus on objects within a given depth range and dull focus for anything outside that range.

6.3. Depth Perception

The ability to perceive the world in three dimensions is an amazing thing by all accounts. The systems in our body that make it possible for us to extract that type of information from our surroundings is an astonishing thing. If the goal in VR is to simulate an environment and communicate that information to a user in 3D, a strong understanding of depth perception and how it works in humans is an extremely useful tool for developing VR systems.

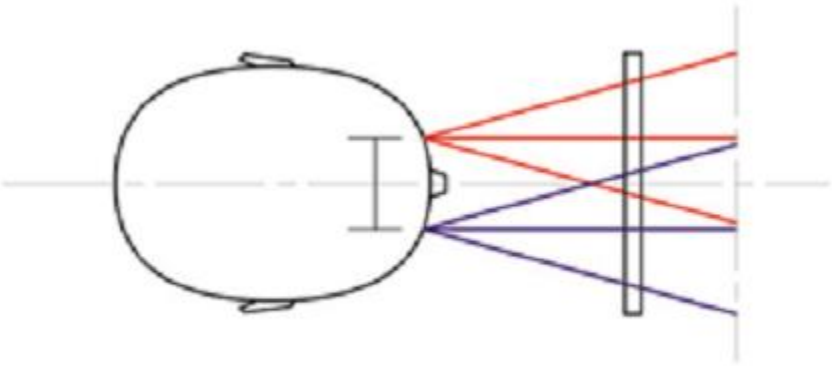
It is important to understand that even though eyes are amazing structures, they are still only sensors. There is no processing going on in the eye itself. The eye is simply an imaging system that provides raw data to the brain for processing. Depth perception along with any other kind of perception occurs in the brain. Contrary to popular belief, depth perception does not require two eyes. Depth is much harder to judge with only one eye, but it is still very possible. Humans judge depth with the help of what are known as "cues". Cues can be thought of most simply as low level hints that when considered together help construct a 3D picture in our minds. There are several monocular cues that take care of a large part of the depth perception task with only one eye. The most important (and familiar) monocular cues are listed below

- **Perspective** is a cue that has already been described in this paper. In this point, it refers strictly to the property of converging parallel lines at a distance. By making the assumption that lines are parallel in 3D space, the brain can judge how far a point on that line is by measuring how close those lines appear.
- **Relative Size** is a cue that helps the brain judge depth between two objects. If the size of the two objects is known to be the same beforehand, the object that appears smaller is considered further away than the other.
- **Motion Parallax** refers to the fact that objects moving across a field of view appear to be moving faster at closer distances and appear stationary when they are sufficiently far away.
- **Depth from Motion** is a cue that helps judge whether or not an object is moving towards or away from an observer. An object moving towards the observer will expand within the field of view while objects moving away will contract.
- **Occlusion** refers to the property that opaque objects will "occlude" or block the view of an observer to an object behind it.

There are many other cues, but these are the most common and the most apparent. A successful graphics engine should be able to simulate these effects successfully, whether stereoscopic vision is being employed or not.

The only binocular cue required to simulate a convincing 3D VR experience is the so called **Binocular Parallax**. For that, the placement of the eyes comes in to play. Most humans have two eyes positioned at the front of the head.

Figure 6-3: Eye Position Assumption [8]



They are close enough to each other so that the assumption can be made that both projection axes are perpendicular to each other. This is an important assumption because it allows for huge simplifications during the stereo rendering process. 3D display systems which are not head mounted have to assume asymmetric or non-parallel projection axes because the display is probably going to far enough away from the eyes. The luxury of a head mounted system is that the screen proximity allows for this approximation.

Even with our eyes separated by such a relatively small amount, there is enough of a difference between the two images for the brain to triangulate depth. Try this. Hold a finger up in front of your face so that both eyes can see it. Focus on a point on the background behind the finger and close one eye. Note where the finger is in the other eye's field of view. Now switch eyes. If done fast enough, the finger seems to jump around. This is basically due to the motion parallax cue mentioned before, except that instead of moving the object, effectively the eye is moving. With some experimenting, it will become apparent that the distance that the finger appears to jump depends on how far it is from the eyes. With each eye seeing a slightly different image, the brain is able to piece together the depths of objects in the near field with a high degree of accuracy. Objects far away tend not to jump around as much, so perceiving an absolute depth for them is more difficult.

6.4. A Brief History of HMDs

Head Mounted Displays (HMDs) have been around for a long time. The first computer tethered HMD was invented in 1966 at MIT, and it worked by reflecting the screens of two bulky CRT monitors into the eyes of the user. This early model was capable of tracking and responding to head movements. The technology was not portable by any means but it set a precedent for a new paradigm in HCI.

The technology seemed to stagnate for a while. It was difficult to make the early systems light enough or portable enough to be practical. Applications were found in the military and helmet mounted displays became common place in aircraft to convey HUD information to pilots.

It wasn't until the early 90s before HMDs found somewhat of a footing in the gaming world. Early HMDs were interesting but they failed to provide a consistently immersive experiences due to limitations in screen technology, field of view and graphics technology. What's worse is that the early models were extremely expensive. The future of HMDs and consequently VR seemed bleak.

Then, in 2012, a prototype for a new HMD was exhibited at the Electronic Entertainment Expo. It was called the Oculus Rift, and it had an interesting approach to what was now a pretty old problem. The shortcomings of previous HMDs was not shared by the Rift, which claimed a massive 110° field of view, which was unheard of up until this point. Following an extremely successful KickStarter Campaign and the support of major game developers, Oculus Rift broke through as the first viable option for an HMD in decades.

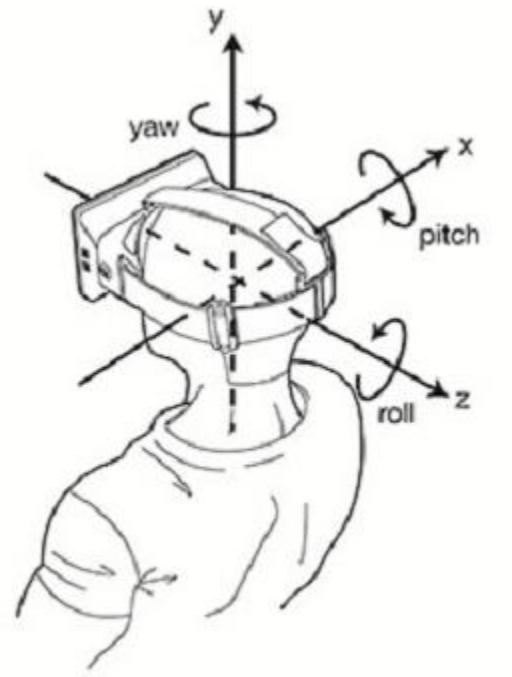
This paper will discuss the Rift in further detail in a later section, but the rest of this chapter will be dedicated to the fundamental concepts of HMD operation, and how the engine incorporates this functionality.

6.5. HMD Integration

The first thing to realize about the HMD in general is that practically speaking, it is nothing more than a monitor screen. Modern HMDs have sensors built in along with hi-fi stereo audio systems, but the true meaning of HMD is in the name…Head Mounted Display. A monitor strapped to a bicycle helmet would technically be considered and HMD. In this paper, an HMD will refer to any portable display system that can be worn on the head capable of providing real time sensor feedback relating to the orientation and/or position of the head. The engine was built and tested for the Oculus Rift, but it was designed to integrate any similar system with minimal additional code. The rest of this section will describe the process of integrating an HMD with the engine. The process outlined here is specific to the Oculus Rift and described thoroughly in [8].

The first step is to establish an appropriate coordinate model for the HMD. There is no reason not to use the same system that was employed to describe the free camera. It is also worthy to note the Rift uses this coordinate system as well so this is an easy enough call to make. Figure 6-4 provides a good pictorial representation of this frame.

Figure 6-4: Rift Coordinate System [8]



A good starting point for modeling this arrangement in virtual space is to just model the entire head. That is, the eyes will be represented by distinct Camera objects within the engine which are attached to a "head" node whose frame will be represented by what is shown in the model above. Mathematically, it can be written out as follows.

$$^{W}H = {}^{W}X_{H} \qquad (6.5\text{-}1)$$

All this does is define the frame of the head which will be further defined shortly.

$$^{H}C_{left} = {}^{H}X_{C-} = T(-\frac{d_{ip}}{2}, 0, -d_z)$$

$$^{H}C_{right} = {}^{H}X_{C+} = T(+\frac{d_{ip}}{2}, 0, -d_z) \qquad 6.5\text{-}2$$

These two equations establish the positions of the left and right eyes with respect to the head frame. Here, d_{ip} refers to the inter-pupillary distance which is known to be around 64mm on average.

Each eye is placed at half this distance on the x axis on opposite ends. The quantity, d_z represents the position of the eyes in z with respect to the center of the head.

The last step is to define H in terms of the yaw, pitch and roll angles, with the assumption that all of the rotations are centered at the origin of the frame. The only thing to watch out for is that the rotations are done in the right order to prevent gimbal lock. The transform is given by

$$^{W}H = {}^{W}X_{H} = T(x, y, z)R_z(\theta_{roll})R_y(\theta_{yaw})R_x(\theta_{pitch}) \qquad 6.5\text{-}3$$

The translation represents and arbitrary positioning of the head in space.

Since the eyes are expressed in relation to the head, nothing more needs to be done to the eye frames before calculating the corresponding view matrices, given by:

$$V_{left} = {}^{W}X_{C-}^{-1}$$
$$V_{right} = {}^{W}X_{C+}^{-1} \qquad 6.5\text{-}4$$

It would be nice it that was all there was to do. Unfortunately, setting the proper view matrices is only half the battle.

For HMDs which house a single display monitor, stereoscopic rendering is achieved by splitting the screen into two halves; one for each eye. There are then effectively two viewports to which the engine has to render a scene to, each with a different perspective projection. For that, a few things need to be computed beforehand.

Assuming that the whole screen is characterized by a resolution of wxh pixels, the aspect ratio for a stereoscopic arrangement is given by

$$a = \frac{w}{2h} \qquad 6.5\text{-}5$$

The extra factor of 2 in the denominator comes from the fact that each viewport only uses half of the horizontal resolution or $\frac{w}{2}$.

To calculate the field of view for both eyes, the following formula is employed.

$$\phi_{fov} = 2\,\mathrm{atan}\frac{S_V}{2d_{E2S}} \tag{6.5-6}$$

Where S_V denotes the physical vertical screen size of the display and d_{E2S} denotes the distance between the eyes and screen. The perspective transform can be calculated with respect to the center of the screen like so.

$$P = \begin{bmatrix} \frac{1}{a\tan\frac{\phi_{fov}}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\frac{\phi_{fov}}{2}} & 0 & 0 \\ 0 & 0 & \frac{z_{near} + z_{far}}{z_{near} - z_{far}} & -1 \\ 0 & 0 & \frac{2z_{near}z_{far}}{z_{near} - z_{far}} & 0 \end{bmatrix} \tag{6.5-7}$$

The next step is to shift the perspective transform left and right based on a new parameter, δx given by

$$\delta x = 1 - \frac{d_{lens}}{2S_H} \tag{6.5-8}$$

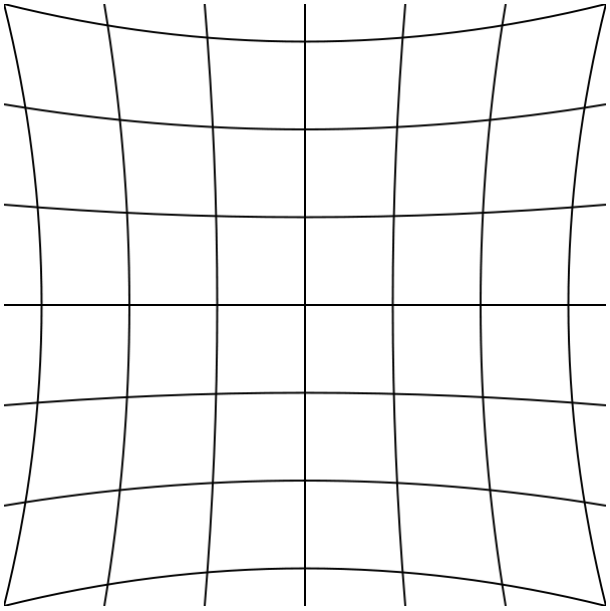
Here, d_{lens} is the separation between the lenses, S_H is the physical horizontal screen size and δx is a horizontal perspective shift factor that will be used to modify the final perspective projections for each eye like so.

$$P_{left} = T(+\delta x, 0,0)P$$
$$P_{right} = T(-\delta x, 0,0)P \tag{6.5-9}$$

6.6. Rift Integration and Lens Correction

The Oculus Rift is innovative for its use of convex lenses to increase the perceived field of view for the user. The problem with using lenses is that they introduce a distortion between the screen and user. The specific type of distortion is called a pincushion distortion, shown below.

Figure 6-5: Pincushion Distortion [8]

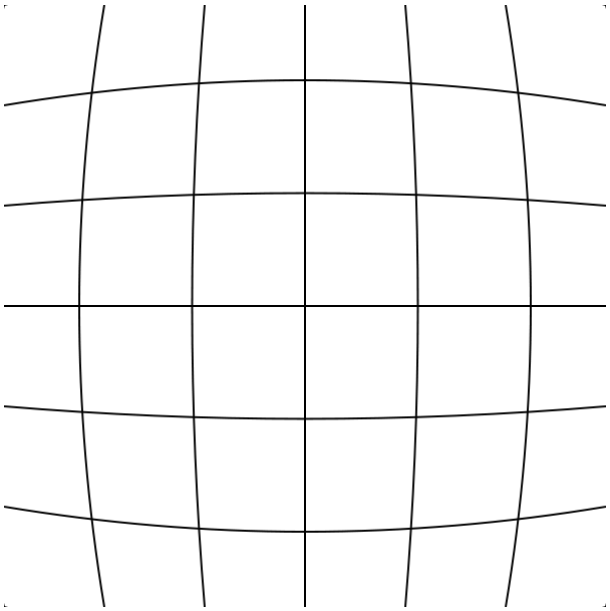


This can be approximated mathematically by a simple transformation in polar coordinates.

$$(r, \phi) \rightarrow (r\, f(r), \phi)$$
$$f(r) = k_0 + k_1 r^2 + k_2 r^4 + k_3 r^6 \qquad 6.6\text{-}1$$

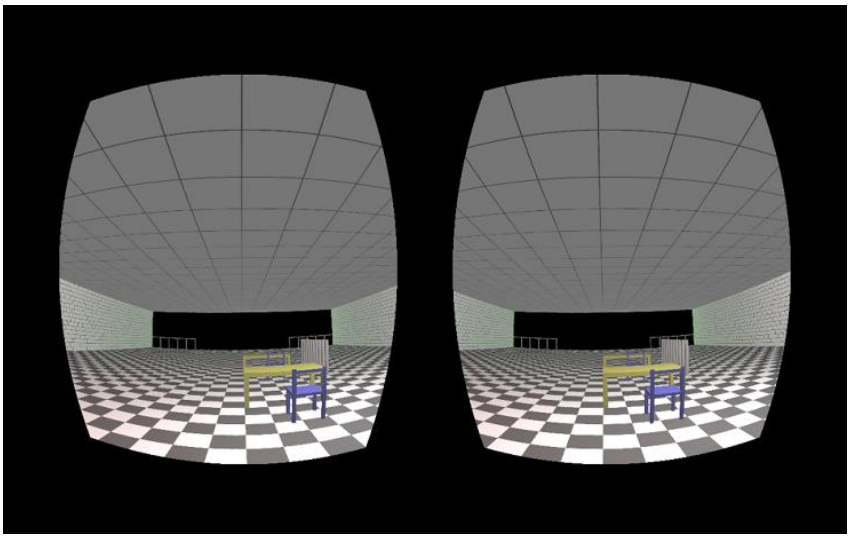
Notice that the distortion only affects the radial component of each point while the polar angle remains unaffected. In order to cancel out the pincushion distortion effect, the original image must be rendered to the screen through a barrel distortion, shown below.

Figure 6-6: Barrel Distortion [8]



This is accomplished by applying the reciprocal of the polar transform shown in 6.6-1 to the image as a post processing affect. The typical result of such an operation is shown below.

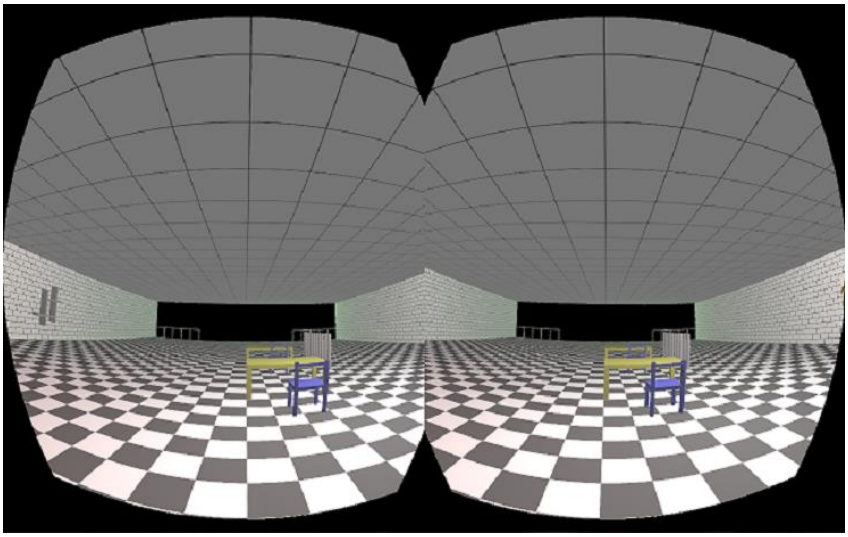
Figure 6-7: Shrinking Effect [8]



This, however, introduces a new inconvenience. The barrel distortion essentially "pulls in" the outer pixels leaving a large amount of unused viewport, thus limiting the true field of view

experienced by the user. To avoid this, the viewport being rendered to must be enlarged so that the distorted images are stretched to the outermost bounds of the screen.

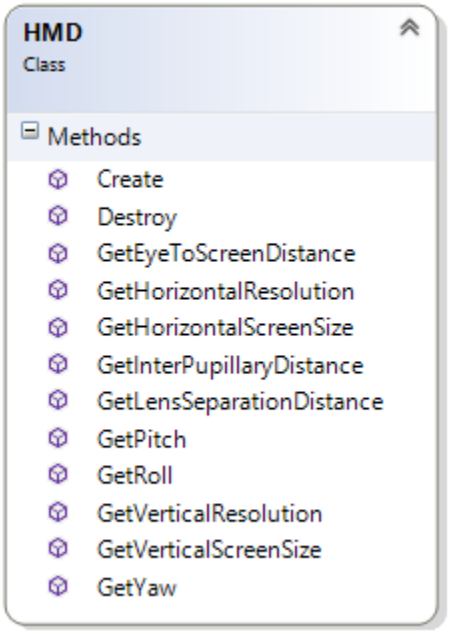
Figure 6-8: Better Scaling [8]



6.7. HMD Abstraction

With the necessary prerequisite knowledge of how HMDs work, the logical next step is to define the abstract interface. This is the interface that the developer will interact with. The idea is to create a list of functions that any HMD must be capable of performing and any manufacturer interested in making their HMD compatible with this interface must write a specific implementation of a derived class. From the information in this chapter, we can say that a compatible HMD must be able to report its physical dimensions, roll-pitch-yaw information and lens positions. **Error! Reference source not found.** is a representation of this class structure that a compliant HMD must derive from.

Figure 6-9: Generic HMD Abstract Base



7. 3D Sensing

The term 3D Sensor refers to a class of technologies that allow for accurate spatial measurement of objects. Interest in 3D sensing for VR applications gained traction with the release of technologies like the Microsoft Kinect and PrimeSense Carmine, which made it possible to interpret real time depth information. While 3D Sensing refers to a much wider range of technologies, the purpose of this chapter is to discuss specifically the use of depth sensor data within the VR engine. The 3D sensor being used for this project is the Microsoft Kinect.

7.1. Microsoft Kinect

The Microsoft Kinect is a line of 3D motion sensing devices. The Kinect was first announced in 2009 at E3 under the codename "Project Natal", in reference to Microsoft's view that the technology represented the birth of a new paradigm in user interfacing. This paradigm has been dubbed the "Natural User Interface" (NUI), referring to that fact that "natural" interfacing without the use of other hardware peripherals is possible through the new technology.

The hardware is based on range camera technology designed by an Israeli company called PrimeSense, and the software was developed internally.

Figure 7-1: Microsoft Kinect [9]

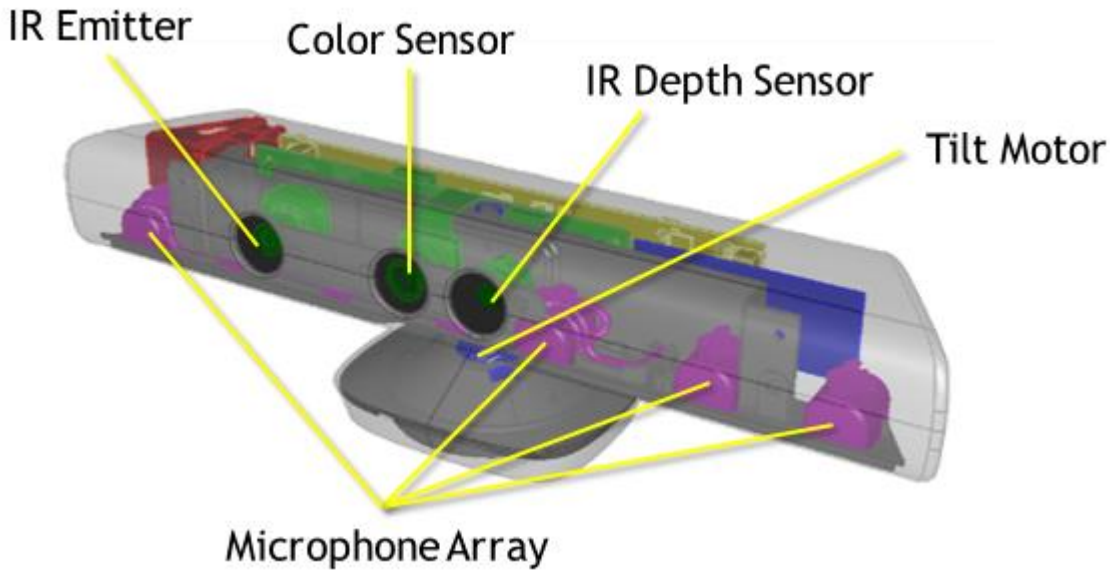
Figure 7-1 is a picture of the original Kinect for Xbox 360. Other variants of the sensor have since been developed for use with the newer Xbox One and Windows PCs. This project makes use of the Kinect for Windows sensor.

7.2. RGB-D

At the heart of the Kinect, and other similar technologies, lies an RGB-D sensor. The RGB refers to the standard Red, Blue and Green color components and the D refers to depth. Most RGB-D sensors actually consist of two cameras. The first is a standard, run-of-the-mill optical camera whereas the second is a slightly more exotic Infrared (IR) camera. Figure 7-2 shows where each component is located on the physical device.

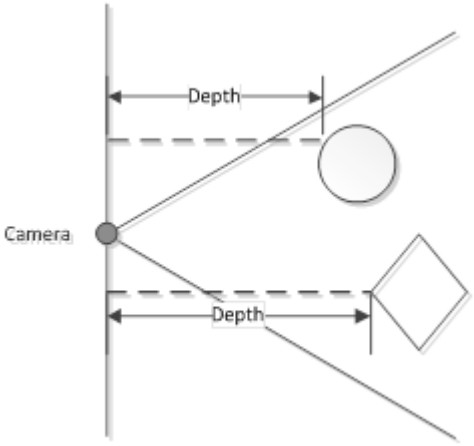
The depth information is extracted from the IR image through a process that PrimeSense refers to as “Light Coding”. It works by analyzing distortions in reflected IR light, also produced by the sensor.

Figure 7-2: Kinect Components [9]



The Kinect is capable of streaming image data in a variety of resolutions. According to the specifications of the Kinect for Windows device used in this project, the color camera is capable of streaming a 32-bit RGB image at resolutions between 640x480 pixels @30fps and 1280x960 pixels @10fps. The depth camera is capable of resolutions between 80x60 and 640x480 @30fps.

Figure 7-3: Depth Representation [9]

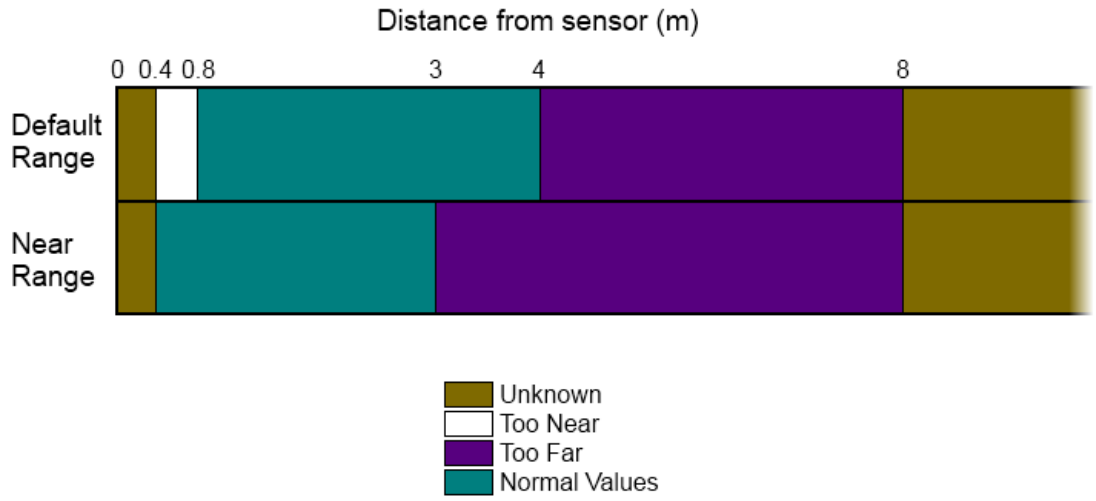


7.3. Depth Stream

Depth data is returned in the form of a 2D image. Each pixel in this image represents a 16-bit gray value. The higher 13 bits of this value represents the depth of that pixel with respect to the frame of the camera in millimeters. The lower 3 bits contain a skeleton ID which will be discussed further in a later section.

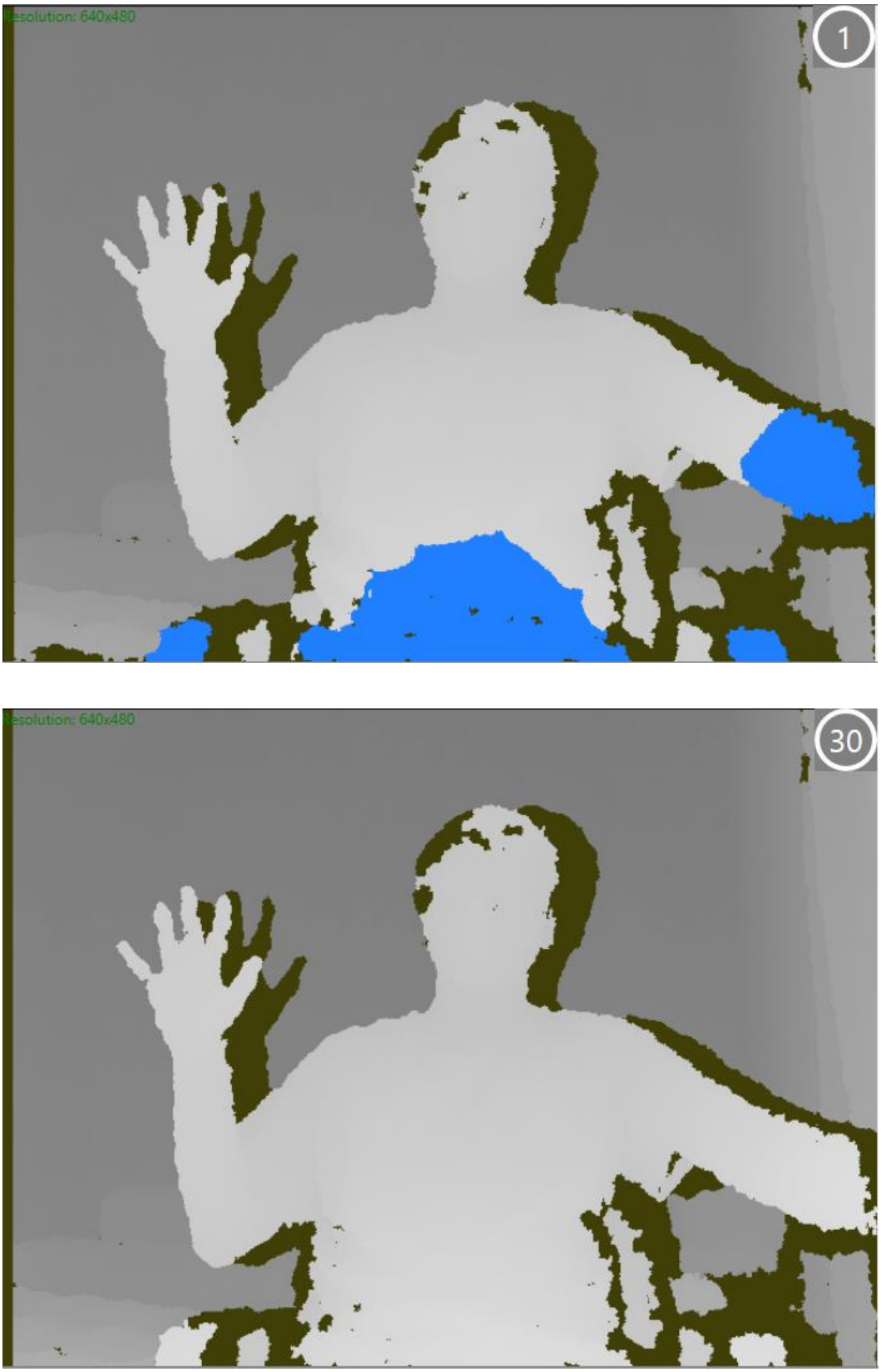
There are two ranges for depth sensing available with the Kinect for PC, each suited for a different application. The codes for each of the extreme cases is shown in **Error! Reference source not found.**.

Figure 7-4: Depth Sensing Modes [9]



A typical depth image is shown in Figure 7-5. Notice that there appears to be a shadow behind the subject. This is due to the fact that the IR light is blocked and the shadow is represented by "unknown" pixels. The depth image on top is in default mode whereas the image on the bottom is in near mode. The blue in the top image represents pixels that are too close. In near mode, these regions are perfectly readable by the sensor.

Figure 7-5: Default (Top), Near (Bottom)



7.4. Color Stream

The Kinect is capable of streaming color information in one of three formats. The first is an RGB format. Each image is represented by a 32-bit linear X8R8G8B8 bitmap in the RGB color space. The second option is YUV, in which each pixel is represented by a 16-bit gamma-corrected linear UYVY-formatted bitmap. The last option is the Bayer format which is designed to complement the physiology of the human eye by favoring green pixels over red and blue. For this project, the default RGB format will be used.

The color image corresponding to the depth image above is shown in Figure 7-6.

Figure 7-6: RGB Color Stream

7.5. IR Stream

The Kinect is also capable of streaming the raw IR data. The intensity is stored with a 10-bit resolution with each pixel represented by a 16-bit value with the 6 least significant bits always set to zero.

Figure 7-7: IR Stream

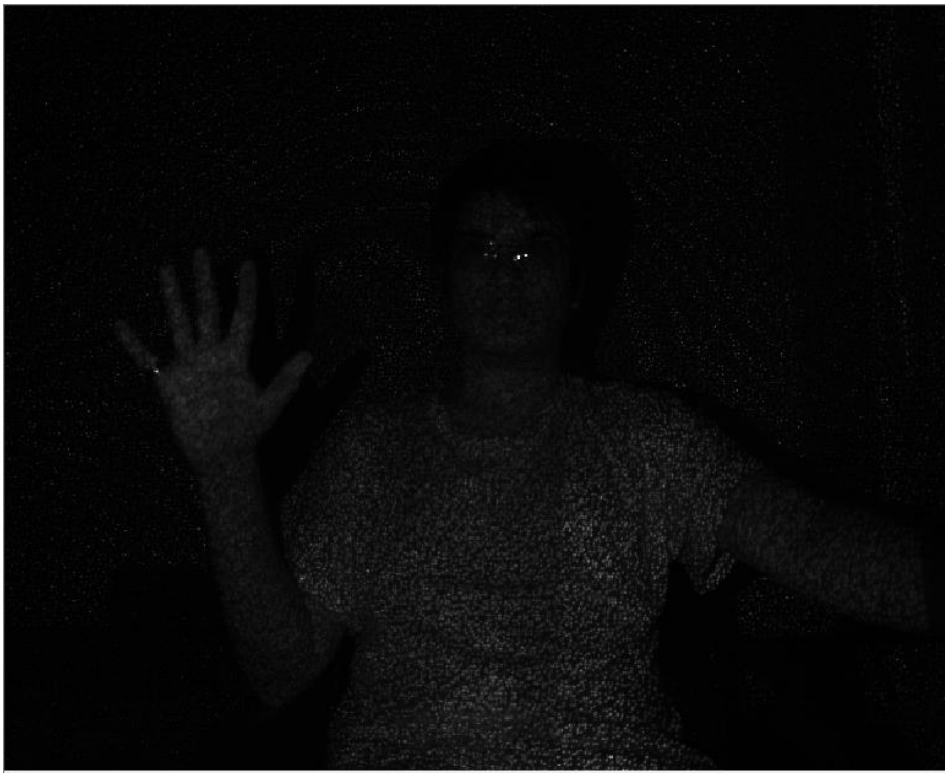
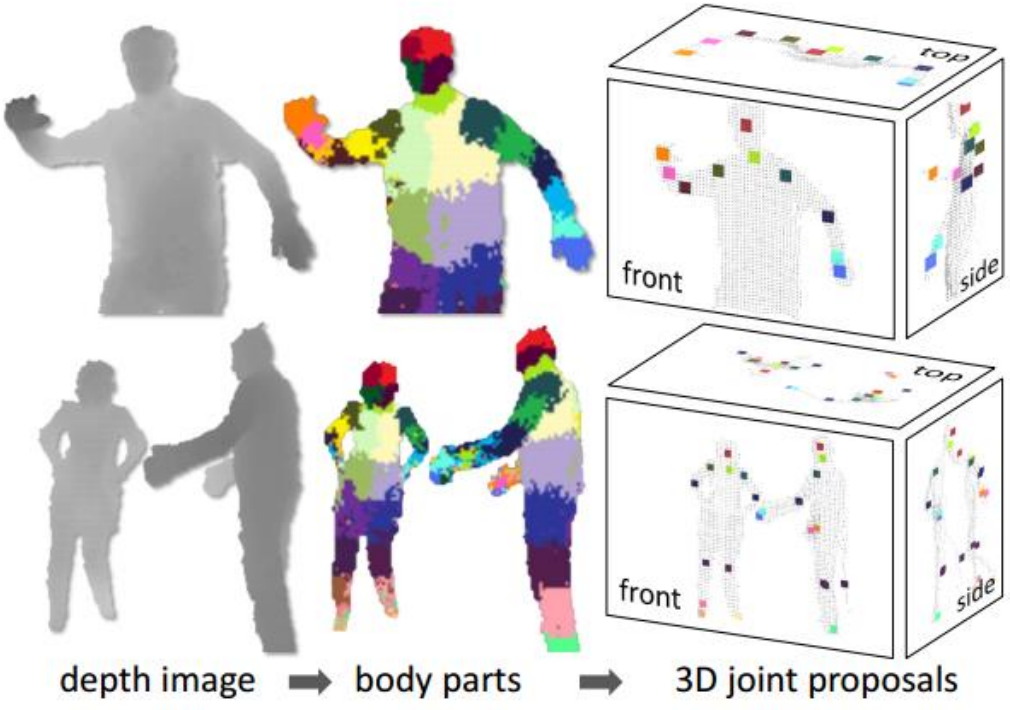


Figure 7-7 shows a typical IR image obtained from the Kinect. Notice that the shadow in the background corresponds to the unknown pixels in the depth images.

7.6. Skeletal Tracking

The Kinect's skeletal tracking system is by far one of its most useful features. According to the research paper put out by the Microsoft Research team responsible for this feature, "A single input depth image is segmented into a dense probabilistic body part labeling, with the parts defined to be spatially localized near skeletal joints of interest". The process is illustrated in Figure 7-8, obtained from the original paper [10].

Figure 7-8: Body Part Recognition [10]



The process uses a depth image as an input in order to avoid the massive variability of skin color, lighting, clothing, hair and other superficial properties associated with a standard color image. Another advantage of using a depth image over color images is the ability to easily subtracting out the background of a scene with a simple depth thresholding operation. The researchers also stated that the ability to easily generate synthetic depth silhouettes of human poses made the training and testing phase far easier.

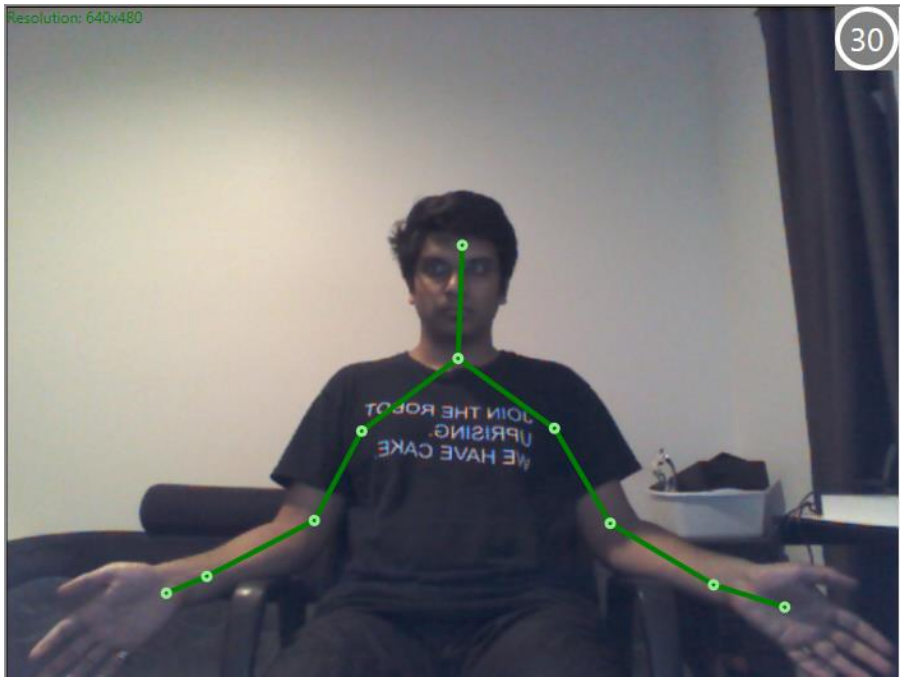
Figure 7-9: Training Data [10]

Figure 7-9 shows a small sample of the numerous real and synthetic poses from the training database used in the paper. Note the wide variety of body types and clothing in the sample data.

The colored regions in the above images are labels for different body regions. These regions in turn are either used to localize specific joints or to contextualize others points of interest to improve joint localization. There are in all 31 distinct body regions segmented by the algorithm. Joint recognition is a per pixel process, that uses the segmentation information to generate a reliable set of possible joint positions in 3D.

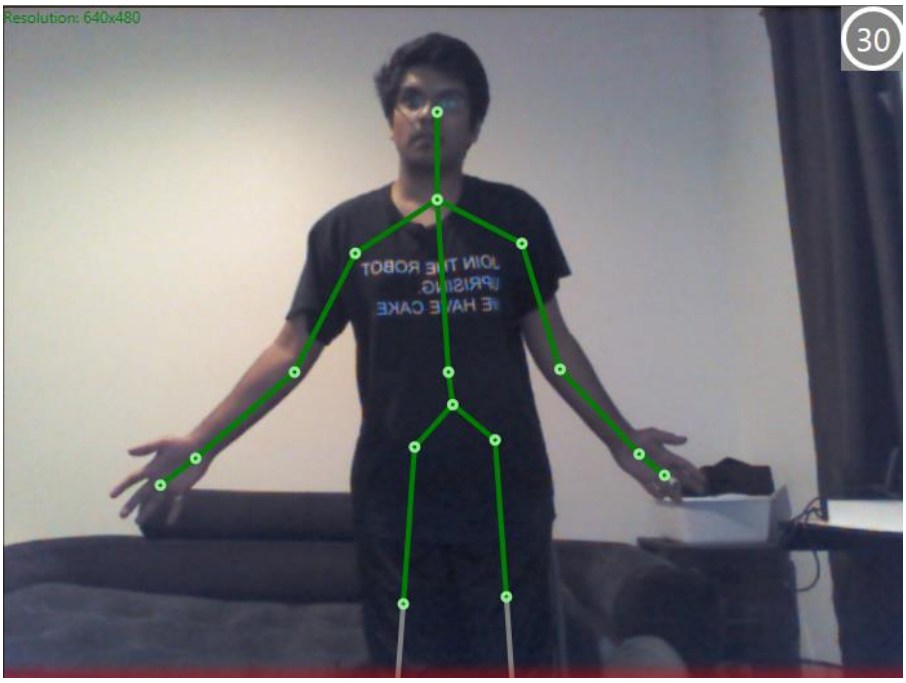
The Kinect provides two tracking modes: Seated and Standing. When seated, the tracker only keeps track of the upper body joints and ignores the rest (Figure 7-10).

Figure 7-10: Skeletal Tracking (Seated)



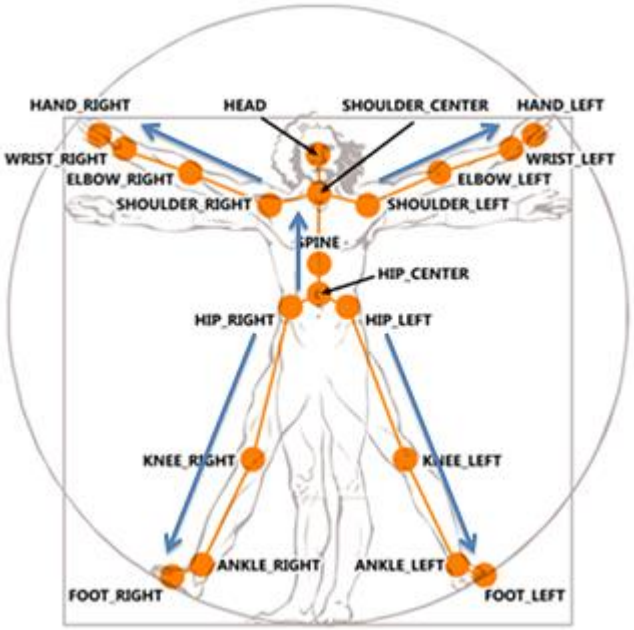
When in standing mode, the entire skeleton is tracked. Joints which are not on frame are inferred and their positions are guessed (Figure 7-11).

Figure 7-11: Skeletal Tracking (Full)



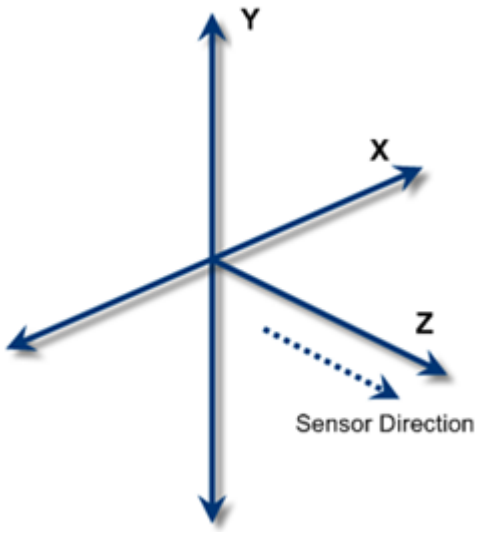
The beads indicate the positions of the 20 joints in 3D space. The labeled joints are shown below.

Figure 7-12: Joint Names [9]



The coordinates of the joints are expressed in "skeleton" space. The depth of the point with respect to the Kinect is the z coordinate. The x axis increases to the left and the y axis increases upwards.

Figure 7-13: Sensor Coordinates [9]



7.7. Transforming Coordinates

The convenience of the homogeneous coordinate system developed earlier in the paper makes representing the joints within the 3D graphical environment of the engine an almost trivial matter. All that needs to be done is to define the matrices that relate the joints to the sensor and the sensor to the world. Each joint can be represented as a homogeneous point in the frame of the Kinect sensor.

$$^{K}p_j = \begin{bmatrix} x_j \\ y_j \\ z_j \\ 1 \end{bmatrix} \qquad 7.7\text{-}1$$

Here, K is the frame of the Kinect sensor. The actual matrix defining this frame is up to the programmer. The only thing to keep in mind is that in world coordinates, the z axis points up whereas in skeleton space, the z axis points towards the skeleton.

There is also occasionally a need to transform between depth, color and skeleton space. The transformation between depth and skeleton space is straightforward as it follows directly from the

discussion of the perspective transform in previous sections. The transformation that maps the horizontal and vertical coordinates of the skeleton to the image is essentially a perspective transformation. The FOV and aspect ratio are simply characteristics of the IR camera. The depth information is encoded in the actual pixel values. A more difficult task is to transform pixels between the color image and depth image. This requires knowledge of the relative positions of the two camera systems with respect to each other. A color pixel must first be unprotected using the inverse perspective transform of the color camera to get a rough estimate of its "position" in 3D space, and then re-projected into depth coordinates using the perspective transform of the depth camera. The last step is to then match the estimated 2D positions to the associated depth pixels.

These processes are mathematically straightforward but finding the actual values to build the transformation matrices can be tricky and dependent on the hardware device itself, which is why these procedures should be provided by the manufacturers. Thankfully, Microsoft provides functions within the API to perform these transformations.

7.8. Multiple Sensors

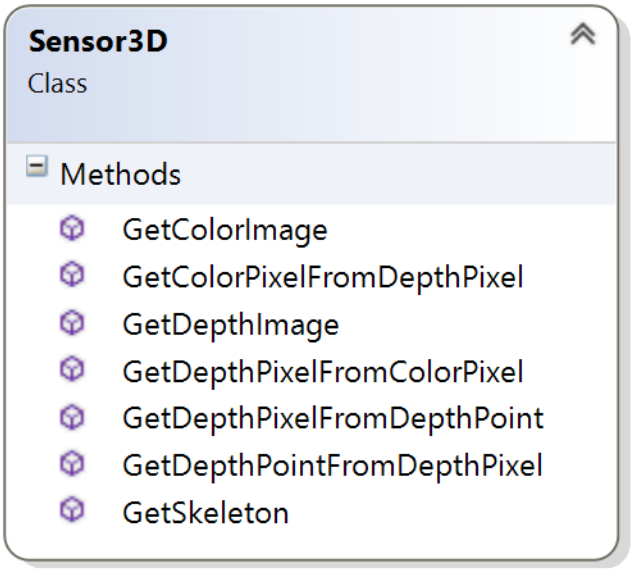
It is worth taking the time to explore the idea of using multiple 3D sensors in one application. One of the main limitations of a fixed sensor is that it can only access information within its own field of view. It is possible to have more than one sensor providing information about the world from several points of view, in which case it would be useful to know where each sensor is with respect to the scene. The spatial representation system developed in this project makes it easy to express relative positions, which in turn largely simplifies the process of integrating date from multiple 3D sensors into a single 3D picture. This is a topic worth exploring further but it is out of the scope of this paper.

7.9. 3D Sensor Abstraction

Just like we developed for the HMD in the previous chapter, we can do the same thing for 3D sensors. The assumption here is that all 3D sensors use a depth camera and are capable of providing

skeletal information. Mapping functions are also required as they are essential for extending beyond this functionality. The figure below is a UML representation of the abstract base class.

Figure 7-14: Sensor3D Abstract Base Class



Each of these functions must be overloaded for specific hardware to be compliant with software written with the engine. The Kinect Framework was used in the design of this interface so all that needs to be done for a specific implementation is to wrap the provided SDK functions with the interface functions.

8. Hand Tracking

Part of implementing a truly natural user interface, is building in functionality to allow users to interact with virtual objects as they would with real world objects. That means paying special attention to the hands. The purpose of this chapter is to outline the development of a simple but robust hand tracking model so that a foundation for an extensible gesture recognition framework may be implemented. A basic grab detector will also be implemented as a proof of concept.

8.1. The Human Hand

It would be hard to imagine life without our hands. For most of us, the hands are our primary means of interacting with the world around us. From a purely physiological point of view, the hands are extremely complex biomechanical structures. From a mechanical perspective, the human hand including the wrist has 26 degrees of freedom: three for hand/wrist position, three more for wrist articulation, and four articulated joints per finger. These joints are by no means unconstrained, but they still allow for a huge number of poses, both static and motional.

8.2. Sensors

There are sensors currently available on the market that can track hand and finger positions to a very high degree of accuracy. The Leap Motion Controller is one such device. It uses an IR emitter/receiver to map out a point cloud of a user's hands.

High resolution sensors like these make the problem of hand tracking simpler, but they have their limitations when it comes to applications where users need the freedom to move around a bit more. The rest of this chapter will outline a procedure to make hand tracking at a distance through a traditional RGB-D sensor a more viable option for VR applications. Understandably, the results might not be as accurate as a specialized sensor, but even the ability to reliably detect a few distinct poses allows for a huge increase in interfacing possibilities.

8.3. Localizing the Hand

Logically, the first step of any hand tracking process should be to localize the hands within the image. The ability to track individual joints from the skeleton makes this an almost trivial task. However, it is important to remember that there are two image streams available for this process: depth and color. There are upsides and downsides to using each one but before those issues can be discussed, there is a more immediate problem. The joint coordinates are provided in 3D skeleton space. Before anything else can be done, those points must be projected back onto the 2D images so that they may be dealt with in pixel space. Conveniently, this is not a new concept. Thinking

back to the pipeline, all it takes is a projective transform to figure out where the joints are in pixel space. Luckily, the Kinect SDK provides utility functions that can map coordinates between all three spaces.

Figure 8-1: Hand in Depth Space

Figure 8-2: Hand in RGB Space

Figure 8-1 shows the result of successfully projecting the joint coordinate of the right hand back into the pixel space of the depth image. This is accomplished by applying a perspective projection matrix to the joint coordinate to find the pixel that represents it. The white rectangle representing the Region of Interest (ROI) is centered on the pixel representing the position of the hand. In order to get the ROI in the proper position in the corresponding RGB image, the pixel at the center needs to be transformed into color space. Once again, the built in utility functions take care of this and the resulting ROI in color space is obtained as shown in Figure 8-2.

All that needs to be done now is to crop out the parts of the image outside of the ROI before any other processing is done.

8.4. Binarization

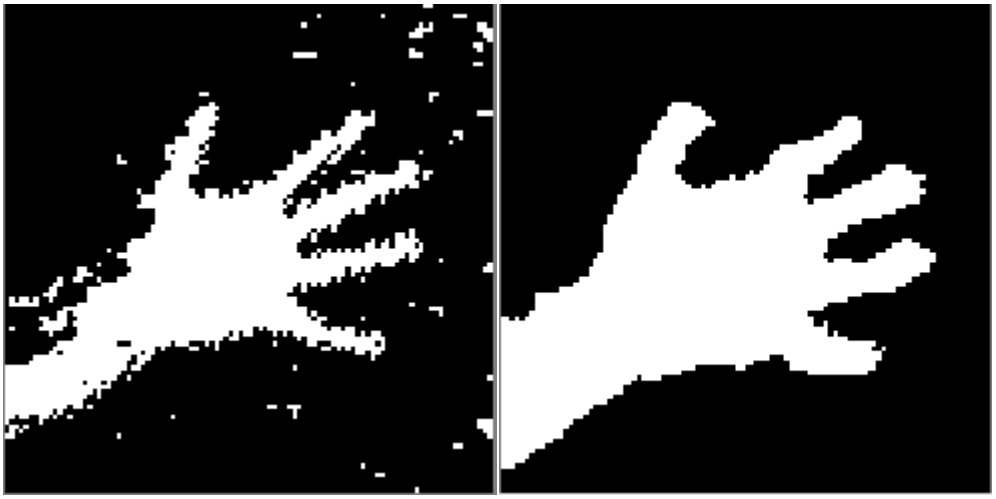
The images coming in from the previous stage might look like what is shown Figure 8-3. The goal of this stage is to convert these images to clean binary images.

Figure 8-3: Localized Images

There is a choice to be made here regarding where the binary image will come from. A simple depth thresholding operation would be good enough to binarize the depth image, but it is already clear in the figure above that the depth image is a fair bit grainier. The other option is to binarize the color image, which is clearly sharper, using a skin classification technique of some sort but good skin classifiers tend to be computationally expensive, and if the application is to run at least 30fps, this could be a problem. There is also the issue of the background rejection, which is more difficult in color images than depth images.

Figure 8-4 shows the results of these two operations. The image on the left is the result of a rough skin classification based solely on hue thresholding of the color image. The image on the right is the result of a direct depth thresholding of the depth image. In this case all depth points in the image greater than the depth of the shoulder center joint were rejected.

Figure 8-4: Binarized Images

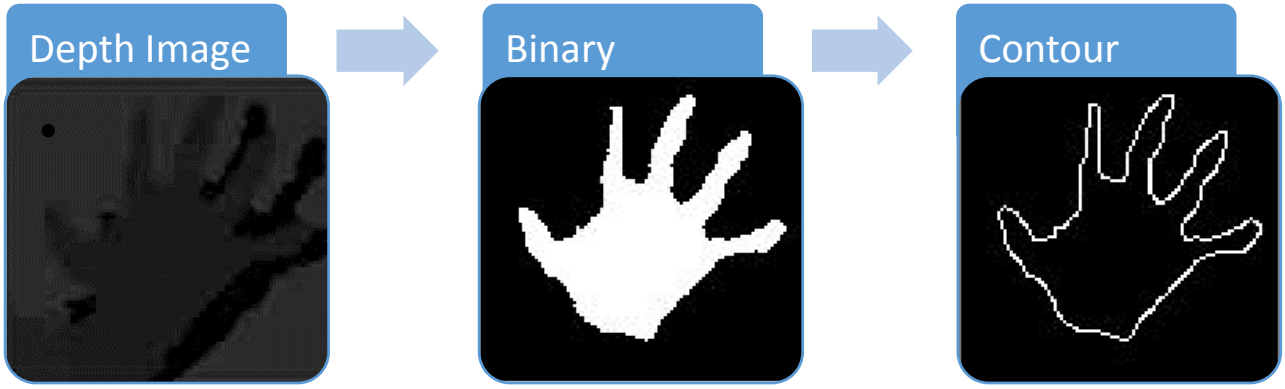


In the interest of frame rate and reliability, the (right) depth image is definitely the better option for hand tracking procedures. It provides an easier way to reject background pixels and is completely independent of lighting conditions.

8.5. Contour Extraction

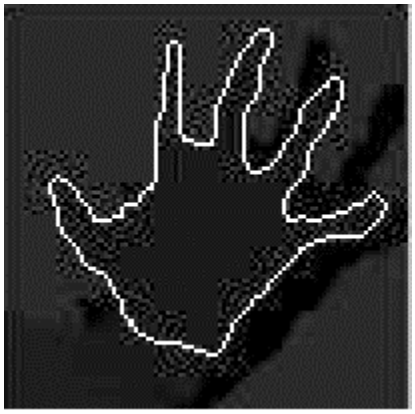
With the hand localized and binarized, we are now free to begin extracting other information. Contour extraction refers to the identification and storage of the edge pixels of the hand. Given that the input is a clean binary image, a simple edge detection routine is all that is needed to produce a crisp edge map of the hand. The images below outline the process.

Figure 8-5: Process



The contour is extracted by creating a list of all connected, non-zero pixels. The longest list is determined to be the contour of the hand. The figure below shows the contour overlaid on the original depth image.

Figure 8-6: Contour on Original

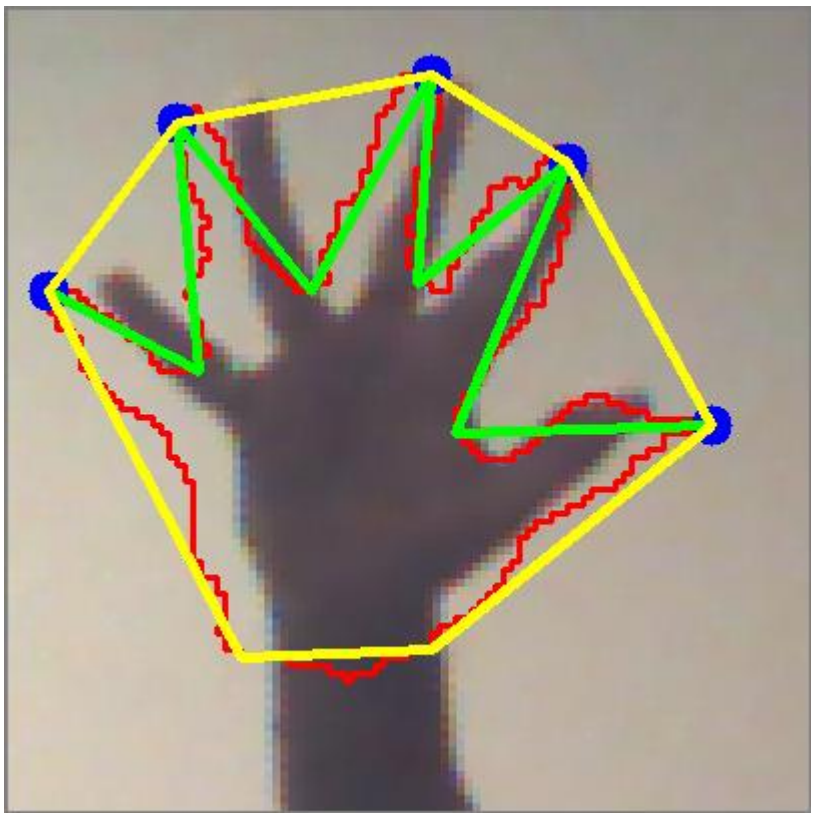


8.6. Finger Counting and Grab Detection

Using this basic information, it is possible to extract some higher level features from the image. For example, it would be useful to know how many fingers are being held up. This may seem like a simple task, but in order for it to be useful, it needs to be fast enough to keep up with the rest of the engine. For this reason, the first step will be to approximate the polygon for the given contour. This is important as it keeps the number of contour points to be further processed low while at the same time producing a clean geometric representation of the user's hand without the contour noise.

Once the contour has been approximated, a convex hull operation is performed on the approximation. The convex hull of a polygon can be thought of as the shape a rubber band would form if wrapped around that polygon. In the case of the hand, the convex hull forms another polygon with vertices at the fingertips and the base of the contour. The figure below illustrates these processes.

Figure 8-7: Finger Tracking



The red line is the contour of the depth image of the hand and the green lines represent the approximation of this contour. The yellow polygon is the convex hull of the approximate polygon. In the figure above, the convex hull has 7 vertices, which are the candidate fingertips. You might notice that the fingertips coincide with smaller angles in the approximate polygon than the angles formed at the base of the hand. This is how fingertips, marked with blue circles, are distinguished. From here, grab detection is accomplished simply by checking that no fingers were found.

9. Conclusions

This project is by no means complete. The nature of the task is such that there is always room for improvement and extension. The purpose of this chapter is to discuss these possibilities as future work, validate results, describe the contributions to the field made through this paper, and provide final thoughts for anyone who would like to use the work in the future.

9.1. Testing and Results

The framework was used to create a series of demos which were tested on a PC with the following specs.

- Intel Core i7 3770K @ 3.5 GHz
- EVGA NVidia GeForce 670 FTW (OpenGL 4.4 Capable)
- 32 GB DDR3 RAM (4 GB used for applications)
- Windows 8.1 Professional x64

As with most graphics based applications, frame rate is an important measure of performance. In all demos, the frame rate was measured using an in game timer system at each frame, by performing the following calculation.

$$F.P.S = \frac{1}{t_{curr} - t_{last}} \qquad 9.1\text{-}1$$

The equation essentially measures the frequency of the main rendering loop. As stated earlier, a sustained frame rate between 25 and 30 frames per second (fps) is our threshold of success. Below this threshold, the lag between frames becomes noticeable. VSYNC was enabled in all cases to prevent screen tearing.

All demos ran at 30 fps with no problems. Demos involving the Oculus Rift also ran comfortably at 30 fps even with a relatively large number of items rendering to the scene despite a two pass rendering pipeline (one for each eye). Demos involving the use of the Kinect and Rift in conjunction suffered a small loss in frame rate due to the simultaneous use of both image streams and skeleton tracking, but still remained above 25 fps. Even with the introduction of the grab detection code operating for both hands, the frame rate remained about the same mostly due to the fact that the vision code was implemented with OpenCV which makes use of multiple threads.

From a developer's point of view, the creation of the demos was greatly eased by the abstracted OOP design of the engine. There was absolutely no need to directly access any of the low level graphics calls from the main application. All OpenGL calls were made through the specific implementation of the OpenGL renderer. The same can be said about interfacing with the Oculus Rift and Microsoft Kinect. The demo application code contained no references to the APIs provided by Oculus or Microsoft. All access to the hardware was successfully abstracted through the custom interfaces designed over the course of this project.

9.2. Future Work

VR is a massive field. It has been in development for over five decades and the possibilities for extension are rich. Here we will discuss other possible modules to extend the functionality of the engine, perhaps even for future thesis projects.

First off, Audio. 3D/Stereo Audio is now commonplace in many multimedia applications. The implementation of a 3D audio engine is a huge undertaking in itself as it involves not only the sourcing and decoding of audio data, but requires real time delay and propagation computations for use in virtual reality environments. Many modern audio libraries such as OpenAL and FMOD allow for abstractions to perform these calculations on dedicated hardware or software by default.

Another important aspect of VR is physics. If the user is to be truly immersed in a simulated reality, objects must behave as if they are under the influence of real physics. Physics simulation in gaming applications has been commonplace for quite a while, and a fast, accurate physics engine would provide a hugely noticeable increase in the qualitative "reality" of VR.

A more functional hand tracking algorithm could provide a massive boost in interface capabilities. As stated earlier, the hand has over 20 degrees of freedom and the rudimentary grab detector outlined in this paper only takes advantage of 3 per hand. A hand tracking system that could reliably

provide information about orientation and finger position would provide a huge improvement in gestural control for the user.

Lastly, a speech recognition module would be an interesting extension that would allow for the user to communicate directly and naturally with other agents within the environment. Along with gesture recognition, it forms the basis of modern natural interfacing and would prove to be a valuable tool within the engine.

9.3. Contribution

The contribution of this work to the field as a whole ultimately comes down to simplification. Creating a series of interfaces that abstract the software away from the hardware and bringing multiple VR technologies under the same engine and API makes it possible for developers to begin experimenting right away. Having to fiddle with low level graphics details or debug a specific hardware component is frustrating and discourages innovation.

In this paper, a common abstract interface for HMDs, 3D Sensors and Graphics libraries has been proposed, implemented and demonstrated to work. Incentivizing a common interface for both developers and manufacturers leads to standardization, implying profits for all involved. This is crucial for the growth and adoption of VR.

REFERENCES

[1] I. E. Sutherland, "The Ultimate Display," *Multimedia: From Wagner to virtual reality*, 1965.

[2] A. Anagnostopoulos and E. Pnevmatikakis, "A real-time mixed reality system for a seamless interaction between real and virtual objects," in *Proceedings of the 3rd international conference on Digital Interactive Media in Entertainment and Arts*, Athens, Greece, 2008.

[3] G. Amayeh, M. Nicolescu and G. Bebis, "Hand-Based Verification and Identification Using Palm-Finger Segmentation and Fusion," *Computer Vision and Image Understanding*, vol. 113, pp. 477-501, 2009.

[4] R. P. Paul, Robot Manipulators, Mathematics, Programming and Control, MIT Press, 1981.

[5] R. S. W. N. H. Graham Sellers, OpenGL SuperBible: Comprehensive Tutorial and Reference, Sams, 2013.

[6] I. P. Howard, Perceiving in Depth, Volume 2: Stereoscopic Vision, Oxford University Press, 2012.

[7] R. E. W. Rafael C. Gonzalez, Digital Image Processing, 3rd Edition, Prentice Hall, 2008.

[8] M. Antonov, N. Mitchell, A. Reisse, L. Cooper, S. La Valle and M. Katsev, "Oculus VR SDK Overview," 21 May 2014. [Online]. [Accessed 27 May 2014].

[9] "Kinect for Windows SDK," Microsoft, 2014. [Online]. Available: http://msdn.microsoft.com/en-us/library/hh855347.aspx. [Accessed 2 May 2014].

[10] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman and A. Blake, "Real-Time Human Pose Recognition in Parts from Single Depth Images," *Communications of the ACM,* pp. 116-124, 2013.

[11] J.-M. T. E. M. Raoul Tubiana, Examination of the Hand and Wrist, CRC Press, 1998.