Warren Robinett
Virtual Reality Games, Inc.
719 E. Rosemary St.
Chapel Hill NC 27514

Richard Holloway
Department of Computer Science,
CB 3175
University of North Carolina
Chapel Hill, NC 27599-3175

The Visual Display Transformation for Virtual Reality

Abstract

The visual display transformation for virtual reality (VR) systems is typically much more complex than the standard viewing transformation discussed in the literature for conventional computer graphics. The process can be represented as a series of transformations, some of which contain parameters that must match the physical configuration of the system hardware and the user's body. Because of the number and complexity of the transformations, a systematic approach and a thorough understanding of the mathematical models involved are essential. This paper presents a complete model for the visual display transformation for a VR system; that is, the series of transformations used to map points from object coordinates to screen coordinates. Virtual objects are typically defined in an object-centered coordinate system (CS), but must be displayed using the screen-centered CSs of the two screens of a head-mounted display (HMD). This particular algorithm for the VR display computation allows multiple users to independently change position, orientation, and scale within the virtual world, allows users to pick up and move virtual objects, uses the measurements from a head tracker to immerse the user in the virtual world, provides an adjustable eye separation for generating two stereoscopic images, uses the off-center perspective projection required by many HMDs, and compensates for the optical distortion introduced by the lenses in an HMD. The implementation of this framework as the core of the UNC VR software is described, and the values of the UNC display parameters are given. We also introduce the vector-quaternion-scalar (VQS) representation for transformations between 3D coordinate systems, which is specifically tailored to the needs of a VR system. The transformations and CSs presented comprise a complete framework for generating the computer-graphic imagery required in a typical VR system. The model presented here is deliberately abstract in order to be general purpose; thus, issues of system design and visual perception are not addressed. While the mathematical techniques involved are already well known, there are enough parameters and pitfalls that a detailed description of the entire process should be a useful tool for someone interested in implementing a VR system.

1 Introduction

A typical virtual reality (VR) system uses computer-graphic imagery displayed to a user through a *head-mounted display (HMD)* to create a perception in the user of a surrounding three-dimensional virtual world. It does this by tracking the position and orientation of the user's head and rapidly generating stereoscopic images in coordination with the user's voluntary head movements as the user looks around and moves around in the virtual world.

The hardware for a typical VR system consists of an HMD for visual input, a

Presence, Vol. 4, No. 1, Winter 1995, 1–23
© 1995 The Massachusetts Institute of Technology

Meta Platforms, Inc.
Exhibit 1087
Page 001

tracker for determining position and orientation of the user's head and hand, a graphics computer for generating the correct images based on the tracker data, and a hand-held *input device* for initiating actions in the virtual world. The visual environment surrounding the user is called the *virtual world.* The world contains *objects,* which are collections of graphics primitives such as polygons. Each object has its own position and orientation within the world, and may also have other attributes. The human being wearing the HMD is called the *user,* and also has a location and orientation within the virtual world.

A good graphics programmer who is given an HMD, a tracker, an input device, and a computer with a graphics library can usually, after some trial and error, produce code to generate a stereoscopic image of a virtual object that, as the user moves to observe it from different viewpoints, appears to hang stably in space. It often takes several months to get to this point. Quite likely, the display code will contain some "magic numbers" that were tweaked by trial and error until the graphics seen through the display looked approximately right. Further work by the programmer will enable the user to use a tracked manual input device to pick up virtual objects and to fly through the virtual world. It takes more work to write code to let the user scale the virtual world up and down, and have virtual objects that stay fixed in room or head or hand space. Making sure that the constants and algorithms in the display code both match the physical geometry of the HMD *and* produce correctly sized and oriented graphics is very difficult and slow work.

In short, writing the display code for a VR system and managing all of the transformations (or transforms, for short) and coordinate systems can be a daunting task. There are many more coordinate systems and transforms to keep track of than in conventional computer graphics. For this reason, a systematic approach is essential. Our intent here is to explain all of the coordinate systems and transformations necessary for the visual display computation of a typical VR system. We will illustrate the concepts with a complete description of the UNC VR display software, including the values for the various display parameters. In doing so, we will introduce the vector-quaternion-scalar (VQS) representation for 3D transformations and will argue that this data structure is well suited for VR software.

2 Related Work

Sutherland built the first computer-graphics-driven HMD in 1968 (Sutherland, 1968). One version of it was stereoscopic, with both a mechanical and a software adjustment for interpupillary distance. It incorporated a head tracker, and could create the illusion of a surrounding 3D computer graphic environment. The graphics used were very simple monochrome 3D wire-frame images.

The VCASS program at Wright-Patterson Air Force Base built many HMD prototypes as experimental pilot helmets (Buchroeder, Seeley, & Vukobradatovitch, 1981).

The Virtual Environment Workstation project at NASA Ames Research Center put together an HMD system in the mid-1980s (Fisher, McGreevy, Humphries, & Robinett, 1986). Some of the early work on the display transform presented in this paper was done there.

Several see-through HMDs were built at the University of North Carolina, along with supporting graphics hardware, starting in 1986 (Holloway, 1987). The development of the display algorithm reported in this paper was begun at UNC in 1989.

CAE Electronics of Quebec developed a fiber-optic head-mounted display intended for flight simulators (CAE, 1986).

VPL Research of Redwood City, California, began selling a commercial 2-user HMD system, called "Reality Built for 2," in 1989 (Blanchard, Burgess, Harvill, Lanier, Lasko, Oberman, & Teitel, 1990).

A prototype see-through HMD targeted for manufacturing applications was built at Boeing in 1992 (Caudell & Mizell, 1992). Its display algorithm and the measurement of the parameters of this algorithm are discussed in Janin, Mizell, and Caudell (1993).

Many other laboratories have set up HMD systems in the last few years. Nearly all of these systems have a stereoscopic HMD whose position and orientation are

Meta Platforms, Inc.
Exhibit 1087
Page 002

measured by a tracker, with the stereoscopic images generated by a computer of some sort, usually specialized for real-time graphics. Display software was written to make these HMD systems function, but except for the Boeing HMD, we are not aware of any detailed, general description of the display transformation for HMD systems. While geometric transformations have also been treated at length in both the computer graphics and robotics fields (Foley, van Dam, Feiner, & Hughes, 1990; Craig, 1986; Paul, 1981), these treatments are not geared toward the subtleties of stereoscopic viewing in a head-mounted display. Therefore, we hope this paper will be useful for those who want to implement the display code for a VR system.

3 Definitions

We will use the symbol T_{A_B} to denote a transformation from coordinate system B to coordinate system A. This notation is similar to the notation $T_{A \leftarrow B}$ used in Foley et al. (1990). We use the term "A_B transform" interchangeably with the symbol T_{A_B}. Points will be represented as column vectors. Thus,

$$\mathbf{p}_A = T_{A_B} \cdot \mathbf{p}_B \tag{1}$$

denotes the transformation of the point P_B in coordinate system B by T_{A_B} to coordinate system A. The composition of two transforms is given by

$$T_{A_C} = T_{A_B} \cdot T_{B_C} \tag{2}$$

and transforms a point in coordinate system C into coordinate system A. Note that the subscripts cancel, as in Pique (1980), which makes complicated transforms easier to derive. The inverse of a transform is denoted by reversing its subscripts:

$$(T_{A_B})^{-1} = T_{B_A} \tag{3}$$

Figure 1 shows a diagram of a point P and its coordinates in coordinate systems A and B, with some example values given. For simplicity, the transform in Figure 1 is limited to translation in 2D. The transform T_{A_B} gives the position of the origin of coordinate system B with respect to coordinate system A, and this matches up with the vector going from A to B in Figure 1. Note, however, that transform T_{A_B} converts the point P from B coordinates ($\mathbf{p}_B$) to A coordinates ($\mathbf{p}_A$)—not from A to B as you might expect from the subscript order.

In general, the transform T_{A_B} converts points from coordinate system B to A, and measures the position, orientation, and scale of coordinate system B with respect to coordinate system A.

4 The VQS Representation

Although the 4×4 homogeneous matrix is the most common representation for transformations used in computer graphics, there are other ways to implement common transformation operations. We introduce here an alternative representation for transforms between 3D coordinate systems, which was first implemented for and tailored specifically to the needs of virtual-reality systems.

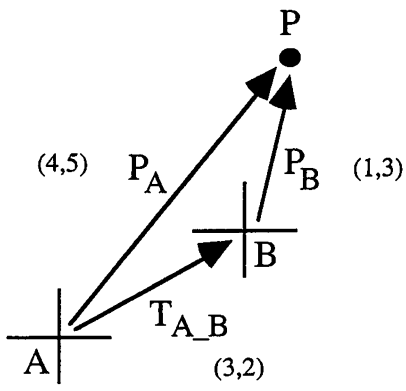
The *VQS* data structure represents the transform between two 3D coordinate systems as a triple $[\mathbf{v}, \mathbf{q}, s]$, consisting of a 3D vector $\mathbf{v}$, a unit quaternion $\mathbf{q}$, and a scalar s. The vector specifies a 3D translation, the quaternion specifies a 3D rotation, and the scalar specifies an amount of uniform scaling (in which each of the three dimensions are scaled by the same factor).

4.1 Advantages of the VQS Representation

The VQS representation handles only rotations, translations, and uniform scaling, which is a subset of the transformations handled by the 4×4 homogeneous matrix. It cannot represent shear, nonuniform scaling, or perspective transformations. This is both a limitation and an advantage.

We have found that for the core work in our VR system, translations, rotations, and uniform scaling are the only transformations we need. Special cases, such as the perspective transformation, can be handled using 4×4 matrices. For operations such as flying, grabbing, scaling, and changing coordinate systems, we have found

Meta Platforms, Inc.
Exhibit 1087
Page 003



$$P_A = T_{A_B} \cdot P_B$$
$$(4,5) = (3,2) + (1,3)$$

T_{A_B} converts points in B to points in A.
T_{A_B} measures the position of B's origin in A.
The vector runs from A to B.

Figure 1. *The meaning of transform T_{A_B}.*

the VQS representation to be superior for the following reasons:

- The VQS representation separates the translation, rotation, and scaling components from one another, which makes it both convenient and intuitive to change these components independently. With homogeneous matrices, it is somewhat more complex to extract the scaling and rotation portions since these two components are combined.
- Renormalizing the rotation component of the VQS representation is simpler and faster than for homogeneous matrices, since the rotation and scale components are independent, and because normalization of quaternions is more efficient than normalization of rotation matrices.
- Uniform scaling is useful for supporting the operations of shrinking and expanding virtual objects and the virtual world without changing their shape.
- The VQS representation is tailored specifically for 3D coordinate systems, not for 2D or higher than 3D. This is because the quaternion component of the VQS data structure represents 3D rotations. Again, this aspect of the VQS representation was motivated by the application to VR systems, which deal exclusively with 3D CSs.
- The advantages of the unit quaternion for representing 3D rotation are described in Shoemake (1985), Funda, Taylor, and Paul (1990), and Cooke, Zyda, Pratt, and McGhee (1992). Briefly, quaternions have several advantages over rotation matrices and Euler angles:
  —Quaternions are more compact than 3 × 3 matrices (4 components as opposed to 9) and therefore have fewer redundant parameters.
  —Quaternions are elegant and numerically robust (particularly in contrast to Euler angles, which suffer from singularities).
  —Quaternions represent the angle and axis of rotation explicitly, making them trivial to extract.
  —Quaternions allow simple interpolation to make possible a smooth rotation from one orientation to another; this is complex and problematic with both matrices and Euler angles.
  —Quaternions can be more efficient in computation time depending on the application (the tradeoffs are discussed in the references above), especially when operand fetch time is considered.

Quaternions are an esoteric and obscure bit of mathematics and are generally not familiar to people from their mathematical schooling, but their appropriateness, simplicity, and power for dealing with 3D rotations have won over many sophisticated users, in spite of their lack of familiarity. The ability to use quaternions to interpolate between rotations is sufficient, by itself, to merit adopting them in the VQS representation.

It may be objected that nonuniform scaling and shear are useful modeling operations, and that the perspective transform must also be a part of any VR system. This is

Meta Platforms, Inc.
Exhibit 1087
Page 004

absolutely correct. The UNC VR system uses both representations—4 × 4 homogeneous matrices are used for modeling and for the last few operations in the viewing transformation, and VQS data structures are used everywhere else. While this may seem awkward at first, keep in mind that the viewing transform is hidden from the user code and, in most applications, so are the modeling operations. Thus, the application code often uses only VQS transformations and is generally simpler, more elegant, and more efficient as a result. In addition, there are certain nonlinear modeling operations (for example, twist) and viewing-transform steps (for example, optical distortion correction) that cannot be handled even by 4 × 4 matrices, so a hybrid system is often necessary in any case.

4.2 VQS Definitions

The triple $[\mathbf{v}, \mathbf{q}, s]$, consisting of a 3D vector $\mathbf{v}$, a unit quaternion $\mathbf{q}$, and a scalar s, represents a transform between two 3D coordinate systems. We write the subcomponents of the vector and quaternion as $\mathbf{v} = (v_x, v_y, v_z)$ and $\mathbf{q} = [(q_x, q_y, q_z), q_w]$. The vector specifies a 3D translation, the quaternion specifies a 3D rotation, and the scalar specifies an amount of uniform scaling.

In terms of 4 × 4 homogeneous matrices, the VQS transform is defined by composing a translation matrix, a rotation matrix, and a scaling matrix:

$$[\mathbf{v}, \mathbf{q}, s] = M_{\text{translate}} \cdot M_{\text{rotate}} \cdot M_{\text{scale}} = \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\cdot \begin{bmatrix} 1 - 2q_y^2 - 2q_z^2 & 2q_xq_y - 2q_wq_z & 2q_xq_z + 2q_wq_y & 0 \\ 2q_xq_y + 2q_wq_z & 1 - 2q_x^2 - 2q_z^2 & 2q_yq_z - 2q_wq_x & 0 \\ 2q_xq_z - 2q_wq_y & 2q_yq_z + 2q_wq_x & 1 - 2q_x^2 - 2q_y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\cdot \begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A complete treatment of quaternions for use in computer graphics is given in Shoemake (1985). However, we will briefly describe some aspects of how quaternions can be used to represent 3D rotations.

A unit quaternion $\mathbf{q} = [(q_x, q_y, q_z), q_w]$ specifies a 3D rotation as an axis of rotation and an angle about that axis. The elements q_x, q_y, and q_z specify the axis of rotation. The element q_w indirectly specifies the angle of rotation θ as

$$\theta = 2 \cos^{-1}(q_w)$$

The formulas for quaternion addition, multiplication, multiplication by a scalar, taking the norm, normalization, inverting, and interpolation are given below, in terms of quaternions q and r, and scalar α:

$$q + r = [(q_x, q_y, q_z), q_w] + [(r_x, r_y, r_z), r_w]$$

$$= [(q_x + r_x, q_y + r_y, q_z + r_z), q_w + r_w]$$

$$q * r = [(q_x, q_y, q_z), q_w] * [(r_x, r_y, r_z), r_w]$$

$$= \begin{bmatrix} \begin{pmatrix} q_xr_w + q_yr_z - q_zr_y + q_wr_x, \\ -q_xr_z + q_yr_w + q_zr_x + q_wr_y, \\ q_xr_y - q_yr_x + q_zr_w + q_wr_z \end{pmatrix} \\ -q_xr_x - q_yr_y - q_zr_z + q_wr_w \end{bmatrix}$$

$$\|q\| = \|(q_x, q_y, q_z), q_w\| = \sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2}$$

$$\alpha \cdot q = \alpha \cdot [(q_x, q_y, q_z), q_w]$$

$$= [(\alpha \cdot q_x, \alpha \cdot q_y, \alpha \cdot q_z), \alpha \cdot q_w]$$

$$normalize(q) = \frac{1}{\|q\|} \cdot q$$

$$q^{-1} = [(q_x, q_y, q_z), q_w]^{-1}$$

$$= \frac{1}{\|q\|^2} \cdot [(-q_x, -q_y, -q_z), q_w]$$

$$nterp(\alpha, q, r) = normalize[(1 - \alpha) \cdot q + \alpha \cdot r]$$

Composing two 3D rotations represented by quaternions is done by multiplying the quaternions. The quaternion inverse gives a rotation around the same axis but of opposite angle. Smooth linear interpolation between two quaternions gives a smooth rotation from one orientation to another.

Meta Platforms, Inc.
Exhibit 1087
Page 005

The rotation of a point or vector **p** by a rotation specified by a quaternion **q** is done by

$$\mathbf{q} * \mathbf{p} * \mathbf{q}^{-1}$$

where the vector **p** is treated as a quaternion with a zero scalar component for the multiplication, and the result turns out to have a zero scalar component and so can be treated as a vector. Using this notation, the VQS transform can be defined more concisely as

$$\mathbf{p}' = [\mathbf{v}, \mathbf{q}, s] \cdot \mathbf{p} = s \cdot (\mathbf{q} * \mathbf{p} * \mathbf{q}^{-1}) + \mathbf{v}$$

This is completely equivalent to the earlier definition.

It can be verified that the composition of two VQS transforms can be calculated as

$$\begin{aligned} T_{A_B} \cdot T_{B_C} &= [\mathbf{v}_{A_B}, \mathbf{q}_{A_B}, s_{A_B}] \cdot [\mathbf{v}_{B_C}, \mathbf{q}_{B_C}, s_{B_C}] \\ &= [(s_{A_B} \cdot (\mathbf{q}_{A_B} * \mathbf{v}_{B_C} * \mathbf{q}_{A_B}{}^{-1})) \\ &\quad + \mathbf{v}_{A_B}, \mathbf{q}_{A_B} * \mathbf{q}_{B_C}, s_{A_B} \cdot s_{B_C}] \end{aligned}$$

The inverse of a VQS transform is

$$\begin{aligned} T_{A_B}{}^{-1} &= [\mathbf{v}_{A_B}, \mathbf{q}_{A_B}, s_{A_B}]^{-1} \\ &= [1/s_{A_B} \cdot (\mathbf{q}_{A_B}{}^{-1} * (-\mathbf{v}_{A_B}) * \mathbf{q}_{A_B}), \mathbf{q}_{A_B}{}^{-1}, 1/s_{A_B}] \end{aligned}$$

We will now move on to describing the transformations making up the visual display computation for VR. We believe that the VQS representation of 3D transforms has advantages, but VQS transforms are not required for VR. In the rest of the paper, it should be understood that, wherever VQS data structures are used, 4×4 homogeneous matrices could have been used instead.

5 Coordinate System Graphs

Many coordinate systems coexist within a VR system. All of these CSs exist simultaneously, and although over time they may be moving with respect to one another, at any given moment, a transform exists to describe the relationship between any pair of them. Certain transforms, however, are given a higher status and are designated the *independent* transforms; all other transforms are considered the *dependent* transforms, and may be calculated from the independent ones. The independent transforms are chosen because they *are* independent of one another: they are either measured by the tracker, constant due to the rigid structure of the HMD, or used as independent variables in the software defining the virtual world.

We have found it helpful to use a diagram of the coordinate systems and independent transforms between them. A CS diagram for an early version of the UNC VR software was presented in Brooks (1989) and a later version in Robinett and Holloway (1992). We represent a typical multiple-user VR system in Figure 2. Each node represents a coordinate system, and each edge linking two nodes represents a transform between those two CSs. Each user is modeled by the subgraph linking the user's eyes, head, and hand. A transform between any pair of CSs may be calculated by finding a path between corresponding nodes in the graph and composing all the intervening transforms.

This, in a nutshell, is how the display computation for VR works: For each virtual object, a path must be found from the object to each of the screens, and then the points defining the object must be pumped through the series of transforms corresponding to that path. This produces the object's defining points in screen coordinates. This must be done for each screen in the VR system. An object is seen stereoscopically (on two screens) by each user, and in a multiuser system an object may be seen simultaneously from different points of view by different users.

As an example, it may be seen from the diagram that the path to Left Screen 1 from Object 3 is

Left Screen 1, Left Eye 1, Head 1, Room 1,

World, Object 3

and thus the corresponding transforms for displaying Object 3 on Left Screen 1 are

$$T_{LS1_O3} = T_{LS1_LE1} \cdot T_{LE1_H1} \cdot T_{H1_R1} \cdot T_{R1_W} \cdot T_{W_O3}$$

As another example, finding a path from Head 1 to Right Screen 2 allows User #2 to see User #1's head.

Note that the CS graph is *connected* and *acyclic*. Dis-

Meta Platforms, Inc.
Exhibit 1087
Page 006

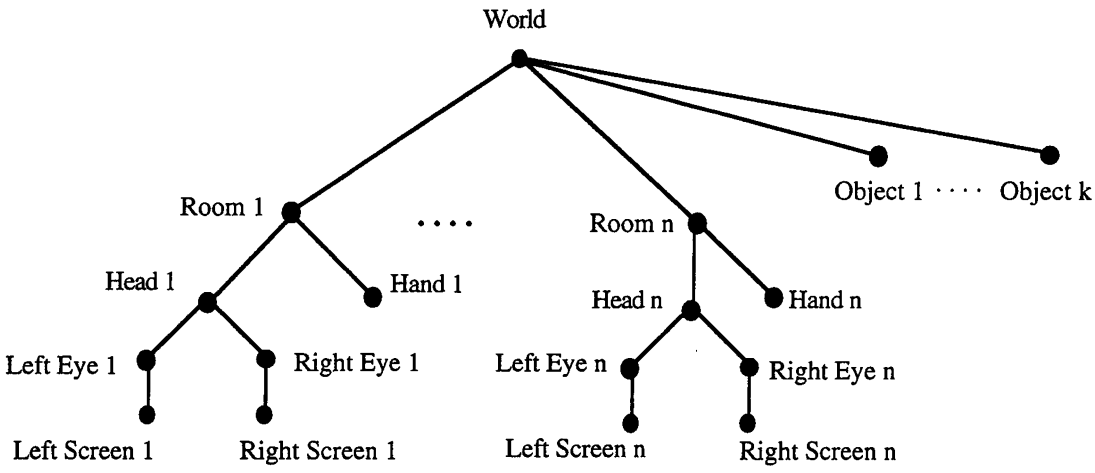


Figure 2. *Coordinate systems for a multi-user virtual world.*

connected subgraphs are undesirable because we want to express all CSs in screen space eventually; a disconnected subgraph would therefore not be viewable. Cycles in the graph are undesirable because they would allow two ways to get between two nodes, which might be inconsistent.

It is primarily the topology of the CS graph that is significant—it shows which CSs are connected by independent transforms. However, the World CS is drawn at the top of the diagram to suggest that all the virtual objects and users are contained in the virtual world. Likewise, the diagram is drawn to suggest that Head and Hand are contained in Room, that Left Eye and Right Eye are contained in Head, and that each Screen is subordinate to the corresponding Eye.

The independence of each transform in the CS graph can be justified. To have independently movable objects, each virtual object must have its own transform (World_Object) defined in the VR software. Likewise, each user must have a dedicated and modifiable transform (Room_World) to be able to change position, orientation, and scale within the virtual world. (This is subjectively perceived by the user as flying through the world, tilting the world, and scaling the world.) The tracker measures the position and orientation of each user's head and hand (Head_Room, Hand_Room) within the physical room where the tracker is mounted. The user's eyes must have distinct positions in the virtual world to see stereoscopically (Left Eye_Head, Right Eye_Head). The optics and geometry of the HMD define the final transform (Screen_Eye).

We can further characterize transforms as *dynamic* (updated each frame, like the tracker's measurements) or *static* (typically characteristic of some physical, fixed relationship, like the positions of the screens in the HMD relative to the eyes).

There can be many users within the same virtual world, and many virtual objects. The users can see one another if their heads, hands, or other body parts have been assigned graphic representations, and if they are positioned in the same part of the virtual world so as to face one another with no objects intervening. We have represented virtual objects as single nodes in the CS graph for simplicity, but objects with moving subparts are possible, and such objects would have more complex subgraphs.

There are other ways this CS diagram could have been drawn. The essential transforms have been included in the diagram, but it is useful to further subdivide some of the transforms, as we will see in later sections of this paper.

Meta Platforms, Inc.
Exhibit 1087
Page 007

6 The Visual Display Computation

The problem we are solving is that of writing the visual display code for a virtual reality system, with provision that

- multiple users inhabit the same virtual world simultaneously;
- each user has a stereoscopic display;
- the user's viewpoint is measured by a head tracker;
- the display code matches the geometry of the HMD, tracker, and optics;
- various HMDs and trackers can be supported by changing parameters of the display code;
- the user can fly through the world, tilt the world, and scale the world; and
- the user can grab and move virtual objects.

In this paper, we present a software architecture for the VR display computation that provides these capabilities. This architecture was implemented as the display software for the VR system in the Computer Science Department at the University of North Carolina at Chapel Hill. The UNC VR system is a research system designed to accommodate a variety of models of HMD, tracker, graphics computer, and manual input device. Dealing with this variety of hardware components forced us to create a flexible software system that could handle the idiosyncrasies of many different VR peripherals. We believe, therefore, that the display software that has evolved at UNC is a good model for VR display software in general, and has the flexibility to handle most current VR peripherals.

We present a set of algorithms and data structures for the visual display computation of VR. We note that there are many choices that face the designer of VR display software, and therefore the display code differs substantially among current VR systems designed by different teams. Some of these differences arise from hardware differences between systems, such as the physical geometry of different HMDs, different optics, different size or position of display devices, different geometries for mounting trackers, and different graphics hardware.

However, there are further differences that are due to design choices made by the architects of each system's software. The software designer must decide what data structure to use in representing the transforms between coordinate systems, define the origin and orientation for the coordinate systems used, and define the sequence of transforms that comprise the overall Screen_Object transform, and decide what parameters to incorporate into the display transform.

The VR display algorithm presented in this paper is a general algorithm that can be tailored to most current VR systems by supplying appropriate values for the parameters of the algorithm. For concreteness, we discuss the implementation of this display algorithm on the UNC VR system. The UNC VR software is based on a software library called *Vlib*. Vlib was designed by both authors and implemented by Holloway in early 1991. A brief overview is given in Holloway, Fuchs, and Robinett (1991).

Vlib was originally written for use with PPHIGS, the graphics library for Pixel-Planes 5 (Fuchs, Poulton, Eyles, Greer, Goldfeather, Ellsworth, Molnar, Turk, Tebbs, & Israel, 1989), the graphics computer in the UNC VR system. However, it was subsequently ported to run on a Silicon Graphics VGX using calls to the GL graphics library. Since Silicon Graphics machines are widely used for VR software, we will describe the GL-based version of Vlib.

6.1 Components of the Visual Display Transform

The Vlib display software maps a point $\mathbf{p}_o$ defined in Object coordinates into a point in Screen coordinates $\mathbf{p}_s$ using this transform:

$$\mathbf{p}_s = T_{S_E} \cdot T_{E_H} \cdot T_{H_R} \cdot T_{R_W} \cdot T_{W_O} \cdot \mathbf{p}_O \qquad (4)$$

This is consistent with the CS diagram of Figure 2. However, there are some complications that make it useful to further decompose two of the transforms above: the Head_Room transform T_{H_R} and the Screen_Eye transform T_{S_E}.

The primary function of the Head_Room transform is to contain the measurement made by the tracker of head

Meta Platforms, Inc.
Exhibit 1087
Page 008

position and orientation, which is being updated each display frame as the user's head moves around. The tracker hardware measures the position and orientation of a small movable sensor with respect to a fixed frame of reference located somewhere in the room. Often, as with the Polhemus magnetic trackers, the fixed frame of reference is a transmitter and the sensor is a receiver.

The two components of tracker hardware, the tracker's base and the tracker's sensor, have native coordinate systems associated with them by the tracker's hardware and software. If the tracker base is bolted onto the ceiling of the room where the VR system is used, this defines a coordinate system for the room with the origin up on the ceiling and with the X, Y, and Z axes pointing whichever way it was mechanically convenient to mount the tracker base onto the ceiling. Likewise, the sensor is mounted somewhere on the rigid structure of the head-mounted display, and the HMD inherits the native coordinate system of the sensor.

In Vlib, we decided to introduce two new coordinate systems and two new static transforms, rather than use the native CSs of the tracker base and sensor as the Room and Head CSs. This allowed us to choose a sensible and natural origin and orientation for Room and Head space. We chose to put the Room origin on the floor of the physical room and orient the Room CS with X as East, Y as North, and Z as up. We chose to define Head coordinates with the origin midway between the eyes, oriented to match the usual screen coordinates with X to the right, Y up, and Z toward the rear of the head.

Thus, the Head_Room transform is decomposed into

$$T_{H_R} = T_{H_HS} \cdot T_{HS_TB} \cdot T_{TB_R} \tag{5}$$

where the tracker directly measures the Head-Sensor_Tracker-Base transform T_{HS_TB}. The mounted position of the tracker base in the room is stored in the Tracker-Base_Room transform T_{TB_R}, and the mounted position of the tracker sensor on the HMD is stored in the Head_Head-Sensor transform T_{H_HS}.

When using more than one type of HMD, it is much more convenient to have Head and Room coordinates be independent of where the sensor and tracker base are mounted. The T_{H_HS} and T_{TB_R} transforms, which are static, can be stored in calibration files and loaded at run-time to match the HMD being used, allowing the same display code to be used with all HMDs. If a sensor or tracker is remounted in a different position, it is easy to change the calibration file. To install a new tracker, a new entry is created in the tracker calibration file. Without this sort of calibration to account for the tracker mounting geometry, the default orientation of the virtual world will change when switching between HMDs with different trackers.

The other transform which it is convenient to further decompose is the Screen_Eye transform T_{S_E}, which can be broken down into

$$T_{S_E} = T_{S_US} \cdot T_{US_N} \cdot T_{N_E} \tag{6}$$

T_{S_US} is the *optical distortion correction transformation,* T_{US_N} is the *3D viewport transformation* described in Foley et al. (1990), and T_{N_E} is the *normalizing perspective transformation.* The 3D viewport transformation is the standard one normally used in computer graphics. The perspective transform is slightly unusual in that it must, in general, use an off-center perspective projection to match the geometry of the HMD being used. The details of this are discussed in a later section. A transformation to model the optics of the HMD is something not normally encountered in standard computer graphics, and it causes some problems which are discussed in more detail later.

Plugging in the decomposed transforms of Eqs. (5) and (6) into the overall display transform of (4) gives

$$T_{S_O} = T_{S_US} \cdot T_{US_N} \cdot T_{N_E} \cdot T_{E_H} \cdot T_{H_HS} \cdot T_{HS_TB} \cdot T_{TB_R} \cdot T_{R_W} \cdot T_{W_O} \tag{7}$$

This is the visual display transform used by the UNC VR software. Note that the leftmost five transforms are static and can therefore be precomputed once to yield T_{S_HS}.

Table 1 lists each of the transformations involved in the overall display transform. Note that several instances of each transform in the table must be maintained to allow for multiple objects, multiple users, and the two

Meta Platforms, Inc.
Exhibit 1087
Page 009

Table 1. *Component Transforms of the Visual Display Transform in VR*

Symbol	Coordinate systems	Instances	Function	Static/dynamic
T_{W_O}	Object to World	1 per object	Position of object in world (changes when object moves)	Dynamic
T_{R_W}	World to Room	1 per user	Position of room in world (changes when flying, etc.)	Dynamic
T_{TB_R}	Room to Tracker Base	1 per user	Position of tracker in room	Static
T_{HS_TB}	Tracker Base to Head Sensor	1 per user	Measurement of head position and orientation by tracker	Dynamic
T_{H_HS}	Head Sensor to Head	1 per user	Position of sensor on HMD	Static
T_{E_H}	Head to Eye	2 per user	Positions of left and right eyes	Static
T_{N_E}	Eye to Normalized	2 per user	Off-center perspective projection	Static
T_{US_N}	Normalized to Undistorted Screen	2 per user	Convert to device coordinates	Static
T_{S_US}	Undistorted Screen to Screen	2 per user	Optical distortion correction	Static

eyes of each user. Example values for these transforms are given in Table 4.

6.2 Coordinate System Definitions

Using transforms between coordinate systems in a VR system requires that the various CSs be precisely defined. A standard orthogonal coordinate system is completely specified by giving its origin, its orientation, and its units. Table 2 gives the CSs defined by Vlib. Vlib is not a graphics library and therefore defines only the coordinate systems going from Object space to Eye space. The remaining low-level coordinate systems are defined in the graphics library being used. The transforms between the CSs listed in Table 2 are all represented in Vlib by the VQS representation, since translation, rotation, and uniform scaling are all that are needed.

The transforms going from Eye space to Screen space are handled by the graphics library supporting the graphics hardware of the VR system. These transforms include off-center perspective projection, optical distortion correction, and mapping to screen coordinates. The coordinate systems for SGI's GL are listed in Table 3.

The US (Undistorted Screen) coordinate system above requires explanation. It is not officially supported by GL. It is included in Table 3 to indicate the point at which nonlinear image predistortion to compensate for optical distortion could be done: namely, in the step from US to S coordinates.

7 Discussion of Component Transforms

We now discuss each of the component transforms of the full visual display transform for VR, as implemented in UNC's Vlib software, running on top of GL on a Silicon Graphics VGX. The complete display transform, again, is

$$T_{S_O} = T_{S_US} \cdot T_{US_N} \cdot T_{N_E} \cdot T_{E_H} \cdot T_{H_HS} \cdot T_{HS_TB} \cdot T_{TB_R} \cdot T_{R_W} \cdot T_{W_O} \tag{8}$$

In discussing a transformation between two coordinate systems A and B, it is easy to get confused as to whether the transform should be measured from A to B, or from B to A. In the following sections, keep in mind that the transform T_{A_B} contains the position, orientation, and scale of coordinate system B as measured from coordinate system A, as was discussed earlier in Section 3.

Meta Platforms, Inc.
Exhibit 1087
Page 010

Table 2. *Vlib Coordinate Systems*

Symbol	Name	Origin	Orientation	Units
O	Object	Center of object	X = right Y = front Z = top	Same as World, unless object was scaled
W	World	Initially on floor directly underneath tracker base	Same as Room, until world is tilted	Same as Room, until world is scaled
R	Room	Lower southwest corner of room	X = east Y = north Z = up	Meters
TB	Tracker Base	Center of base	Depends on mounting location	Meters
HS	Head Sensor	Center of sensor	Depends on mounting location	Meters
H	Head	Mid-point between user's eyes	X = right Y = up Z = backward	Meters
E	Eye	Center of pupil of eye	X = right Y = up Z = backward	Meters

7.1 World_Object Transform (Position of Object in Virtual World)

The World_Object transform $T_{W_O} = [\mathbf{v}_{W_O}, \mathbf{q}_{W_O}, s_{W_O}]$ determines the position, orientation, and size of each object in the virtual world. Each object has its own instance of this transform, which permits each object to be independently moved, rotated, or scaled. The vector $\mathbf{v}_{W_O}$ defines the object's position, the quaternion $\mathbf{q}_{W_O}$ defines its orientation, and the scalar s_{W_O} defines its size.

7.2 Room_World Transform (Position of User in Virtual World)

The Room_World transform $T_{R_W} = [\mathbf{v}_{R_W}, \mathbf{q}_{R_W}, s_{R_W}]$ determines the position, orientation, and size of each user in the virtual world. Each user has a dedicated instance of this transform, and this permits users to have independent locations, orientations, and scales within the virtual world. In a multiuser virtual world, this means that different users can fly through the world independently of one another. Similarly, different users can also see the virtual world from different orientations or different scale factors.

The vector $\mathbf{v}_{R_W}$ defines the user's position in the virtual world, and it may be incrementally modified to cause flying through the world to occur.

The quaternion $\mathbf{q}_{R_W}$ defines the user's orientation, and can be modified to tilt the entire virtual world to a new orientation. Because the force of gravity constantly reminds the user of the direction of real-world down, the user perceives the virtual world to be turning, rather than that his or her own body is turning within the virtual world.

The scalar s_{R_W} defines the user's size within the virtual world. This value can be multiplied each frame by a constant slightly greater or less than 1 to cause the virtual world to shrink or expand.

The precise quantities represented by $\mathbf{v}_{R_W}$, $\mathbf{q}_{R_W}$, and s_{R_W} are the position, orientation, and scale of World coordinates with respect to Room coordinates. Exactly

Meta Platforms, Inc.
Exhibit 1087
Page 011

Table 3. *GL Coordinate Systems (Silicon Graphics 91)*

Symbol	Name	Origin	Orientation	Units
E	Eye	Center of pupil of eye	X = right Y = up Z = backward	Meters
C	Clip	Same as above	X = right Y = up Z = forward	Meters
N	Normalized	Same as above	X = right Y = up	Normalized
W	Window	Same as above	X = right Y = up	Pixels
US	Undistorted Screen	Lower left corner of viewport	X = right Y = up	Pixels
S	Screen	Lower left corner of viewport	X = right Y = up	Pixels

how these variables must be modified to implement the operations of flying through a virtual world, tilting the world, scaling the world, and grabbing virtual objects is discussed in detail in Robinett and Holloway (1992).

7.3 Tracker-Base_Room Transform (Mounting Position of Tracker Base)

The Tracker-Base_Room transform $T_{TB_R} = [\mathbf{v}_{TB_R}, \mathbf{q}_{TB_R}, 1]$ describes the position and orientation of the tracker base (often a transmitter) within the physical room where the VR system is set up. This value is stored in a calibration file for the tracker currently being used. Note that the scale factor is required to be 1 for this transform.

To be precise, the Tracker-Base_Room transform T_{TB_R} contains the position and orientation of Room coordinates as measured from Tracker-Base coordinates. When relocating the tracker, it is usually most convenient to measure the position and orientation of the tracker base with respect to the fixed coordinate system of the room, and then calculate the inverse transform as the value for T_{TB_R}.

7.4 Head-Sensor_Tracker-Base Transform (Measurement by Tracker)

The Head-Sensor_Tracker-Base transform $T_{HS_TB} = [\mathbf{v}_{HS_TB}, \mathbf{q}_{HS_TB}, 1]$ holds the inverse of the measurement of head position and orientation most recently read from the tracker. Most trackers provide the position and orientation of a sensor with respect to the tracker base (T_{TB_HS}).

Not all trackers deliver orientation as a quaternion, so the tracker driver software may have to do a conversion from a 3×3 matrix or from Euler angles. The driver may also have to perform a scaling to convert the position measurement to the required units of meters.

Since the tracker only measures position and orientation, the scale factor for this transform is required to be 1.

The tracker driver outputs a vector and a quaternion. The vector defines the position of the sensor with respect to the tracker's base and the quaternion defines the sensor's orientation with respect to the tracker's base. This vector and quaternion read from the tracker comprise a transform that must be inverted to get the Head-

Meta Platforms, Inc.
Exhibit 1087
Page 012

Sensor_Tracker-Base transform T_{HS_TB}. Note that, as discussed in Section 4, inverting a VQS transform is not equivalent to simply inverting its vector, quaternion, and scalar components.

7.5 Head_Head-Sensor Transform (Mounting Position of Sensor on HMD)

The Head_Head-Sensor transform $T_{H_HS} = [\mathbf{v}_{H_HS}, \mathbf{q}_{H_HS}, 1]$ describes the position and orientation of where the head sensor is mounted on the HMD. These measurements are with respect to the Head coordinate system, centered at the midpoint between the eyes. The vector $\mathbf{v}_{H_HS}$ defines the position of the sensor with respect to the Head CS and the quaternion $\mathbf{q}_{H_HS}$ defines the sensor's orientation in the Head CS. The scale factor is required to be 1 for this transform also.

This value is stored in a calibration file for the HMD currently being used.

7.6 Eye_Head Transform (Separation of the Eyes)

For each user, the Eye_Head transform has two instances, one for each eye. These two transforms position the viewpoints of the user's eyes at slightly separated points within the virtual world, thus allowing stereoscopic vision through the HMD. Figure 3 shows a diagram of this.

The Left-Eye_Head transform $T_{LE_H} = [\mathbf{v}_{LE_H}, \mathbf{q}_{LE_H}, 1]$ and Right-Eye_Head transform $T_{RE_H} = [\mathbf{v}_{RE_H}, \mathbf{q}_{RE_H}, 1]$ describe the positions of the eyes with respect to the Head coordinate system centered at the midpoint between the user's eyes. (Because we are transforming from Head space to Eye space, T_{E_H} actually describes the position of the Head CS with respect to the Eye CS, not vice versa.) Thus, for a given interpupillary distance (IPD), using the Head coordinate system described in Table 2 we have

$$\mathbf{v}_{LE_H} = (+\mathrm{IPD}/2, 0, 0)$$

$$\mathbf{v}_{RE_H} = (-\mathrm{IPD}/2, 0, 0)$$

There are considerable individual differences among the IPDs of adults, with 95% falling in the range from 49 to 75 mm (Woodson, 1981). Wide-eyed and narrow-eyed people will perceive the same scene in an HMD to have different absolute sizes and distances (Rolland, Ariely, & Gibson, 1993). To avoid this problem, the IPD for each user needs to be measured (with a device such as an optician uses) and the user's IPD needs to be entered into a calibration file specific to that user. Since most people wear eyeglasses or contact lenses customized to their vision and facial geometry, it should not be surprising that HMDs and their display software need to be customized for each user.

However, it is often not practical or not worth the trouble to change the IPD setting each time a new user dons the HMD. In practice, the IPD of the UNC HMDs often remains set at an average value of 62 mm.

It is important to emphasize that the geometric model used in the graphics software must recognize that the eye focuses on the *virtual image* of the screen, not the screen itself. The graphics software for some HMDs erroneously ignores the optics, and models the eyes as focusing directly on screens a few centimeters away (VPL, 1989). If this were accurate, small displacements of the pupil from the assumed center of projection would cause large distortions in the perceived image. Rotation of the eye to gaze at different points in the image and the variation in IPD from one user to another both cause the pupil to be displaced from its assumed location. But since the eyes in fact focus on distant virtual images of the screens, these small displacements of the pupil have a relatively small effect on the perceived image.

The orientation of eye space should match that of the optical system in the HMD. If the HMD being used has parallel optical axes for the two eyes, then the orientation of Left-Eye and Right-Eye space match that of Head space. In this case, the two quaternions will be the identity quaternion $\mathbf{q}_{ident} = [(0,0,0),1]$, corresponding to zero rotation.

However, some HMDs use diverged optical axes to obtain a wider field of view. The parameter $\varnothing$ describes the angle between the optical axes, giving divergence angles of $-\varnothing/2$ and $+\varnothing/2$ around the Υ-axis for the two eyes. This translates into quaternions as

Meta Platforms, Inc.
Exhibit 1087
Page 013

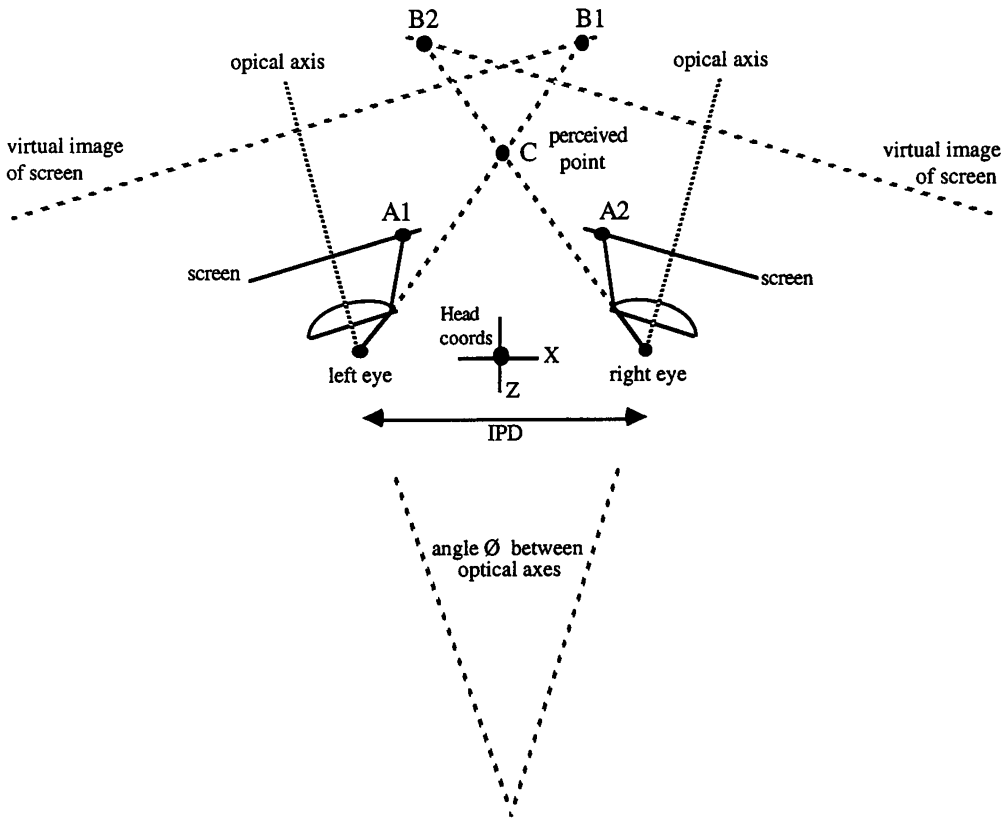


Figure 3. *Stereoscopic optics model for an HMD.*

$$q_{LE_H} = [(0, \sin(+\varnothing/4), 0), \cos(+\varnothing/4)]$$

$$q_{RE_H} = [(0, \sin(-\varnothing/4), 0), \cos(-\varnothing/4)]$$

As another example of how the Eye_Head transform can be used, an HMD was built at UNC in which one display device (an LCD display for the left eye) was upside-down due to mounting constraints. For correct operation, the left image had to be displayed upside down. This was accomplished by composing a 180° rotation around the Z axis with the q_{LE_H} quaternion.

This concludes the list of Vlib transforms, all of which use the VQS representation. To get from Eye space to Screen space, several different representations of the transforms are necessary. The following transforms are described as they are implemented for the GL-based version of Vlib.

7.7 Normalized_Eye Transform (Perspective Transform)

The perspective projection used in an HMD must match the field of view of the HMD, and must position the eyes correctly relative to the screens' virtual images. This usually requires an off-center perspective projection, since the user's eye is generally not lined up with the center of the screen in an HMD.

Figure 4 shows some of the important parameters in eye space pertaining to the perspective projection and is similar to that used in Foley et al. (1990). In this diagram, the screen image defines the plane of projection and the eye is at the origin looking along the $-Z$ axis. We have chosen the eye coordinate system so that the $-Z$ axis is parallel to the optical axis, and is perpendicular to the screen image and intersects it at the view refer-

Meta Platforms, Inc.
Exhibit 1087
Page 014

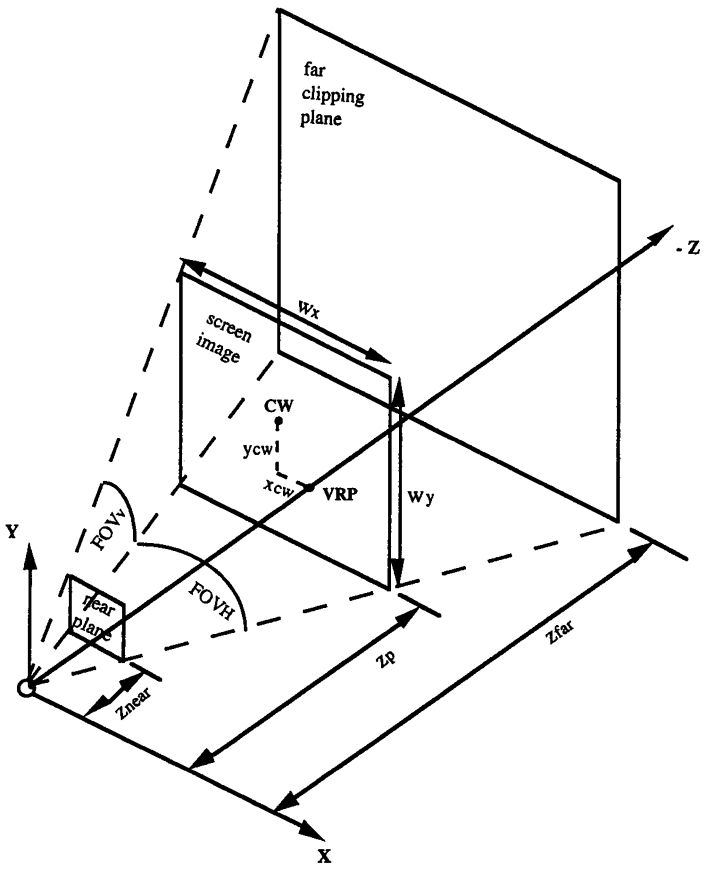


Figure 4. *Viewing parameters for one eye using an off-center perspective projection.*

ence point (VRP) at $z = z_p$. The center of the screen image or *viewing window* is denoted by CW and is not generally at the VRP. The distances x_{cw} and y_{cw} give the offsets to the screen-image center relative to the VRP. The viewing window size is just the size of the screen image and is denoted by w_x and w_y. The near and far clipping plane distances z_{near} and z_{far} determine when objects are too close or too distant for display.

Although this is a complete model for a single eye, there is a complication introduced by some stereoscopic displays. Because the VRP is defined relative to the eye, as the IPD changes for different users, the VRP's horizontal placement changes as well. If the display does not have a physical IPD adjustment, then the VRP moves laterally with respect to CW. In this case x_{cw} changes with the IPD.

Figure 5 shows the situation for the left eye. Here, x_{H_CW} is the X coordinate of the left screen's CW relative to Head space (the value for this must be derived from the specifications for the HMD) and x_{H_LE} is the X coordinate of the left eye relative to Head space and for the left eye, which is just

$$x_{H_LE} = \frac{-\mathrm{IPD}}{2}$$

Thus,

$$x_{CW} = x_{H_CW} - x_{H_LE} = x_{H_CW} + \frac{\mathrm{IPD}}{2} \qquad \text{(left eye)}$$

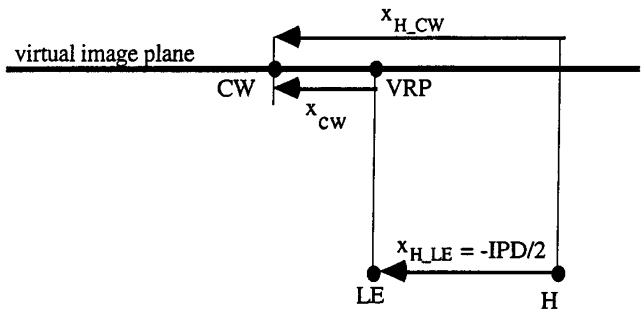


Figure 5. *Dependence of x_{cw} on IPD (left eye).*

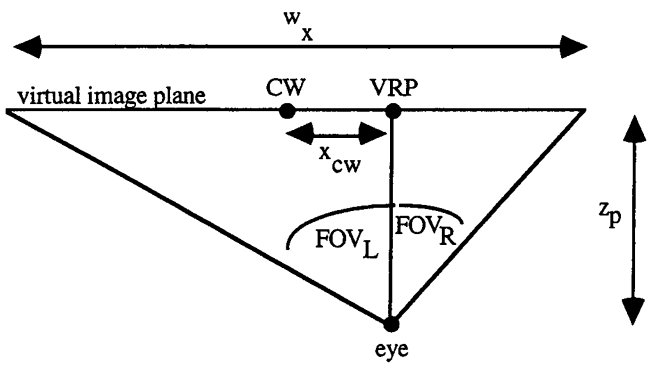


Figure 6. *Relation of horizontal FOV to off-center projection parameters.*

The situation for the right eye is similar:

$$x_{CW} = x_{H_CW} - x_{H_RE} = x_{H_CW} - \frac{IPD}{2} \qquad \text{(right eye)}$$

Note that x_{H_CW} is negative for the left eye and positive for the right.

x_{cw} and y_{cw} can also be used to correct for misalignments of the HMD hardware. For example, in the VPL EyePhone, an accidental displacement of the display device in the object plane by 1 mm results in an angular error of roughly 1.5°. Errors in vertical placement on the image plane produce vertical angular offsets, and can result in corresponding pixels in the left and right displays being vertically misaligned. This is called *dipvergence.* Dipvergence can be corrected by adjusting y_{cw} to reflect this offset in the computer graphics model. Similarly, horizontal placement error can be fixed by adjusting x_{cw}.

The calculation of the field of view is also more complicated for off-center projections. Figure 6 shows the relationship between the horizontal field-of-view angles and the off-center projection parameters. Because the eye is off-center with respect to the screen, the left and right components, FOV_L and FOV_R, of the horizontal field of view FOV_H are not equal and must be calculated separately.

$$FOV_L = \tan^{-1}\left[\frac{(w_x/2) - x_{cw}}{|z_p|}\right]$$

$$FOV_R = \tan^{-1}\left[\frac{(w_x/2) + x_{cw}}{|z_p|}\right]$$

$$FOV_H = FOV_L + FOV_R$$

The calculation is similar for the vertical field of view FOV_V, which is divided into top and bottom angles.

$$FOV_T = \tan^{-1}\left[\frac{(w_y/2) + y_{cw}}{|z_p|}\right]$$

$$FOV_B = \tan^{-1}\left[\frac{(w_y/2) - y_{cw}}{|z_p|}\right]$$

$$FOV_V = FOV_T + FOV_B$$

It is important to note that although the field-of-view angle is an important parameter for characterizing a head-mounted display, it is not the best choice of parameter for specifying off-center projections (for reasons which are beyond the scope of this paper). For example, the GL library has two different calls for setting up the perspective transformation: the *perspective* call is used for on-center projections and takes the field of view as a parameter, whereas the *window* call (discussed below) is intended for off-center projections and does not use the field of view as a parameter.

Now that we have discussed the meanings and proper values for the parameters in specifying the perspective transform, we can move on to a discussion of how these values are used.

The GL coordinate systems listed in Table 3 are not usually accessed directly. Normal applications do not

Meta Platforms, Inc.
Exhibit 1087
Page 016

need to compose a transformation between each pair of CSs in the table. Rather, a few GL calls are used to set up the transforms from Eye space to Screen space. The *window* call sets up the normalizing perspective transformation. The syntax is

window (*left*, *right*, *bottom*, *top*, *near*, *far*)

where *near* and *far* are the positive distances from the eyepoint to the near and far clipping planes. Since our Eye CS is right-handed with the $-Z$ axis away, we will need to negate *near* and *far* before using them in equations for which sign is important.

Note that there is no specification of the projection-plane distance z_p in the *window* call. This is because GL assumes that the window is inscribed in the near clipping plane. Therefore, the window specification describing the virtual image of the screen must be projected onto the near clipping plane (which usually must be closer to the eye than the virtual screen-image distance). Thus we have

$$left = \frac{-\text{near} \cdot (x_{cw} - w_x/2)}{z_p}$$

$$right = \frac{-\text{near} \cdot (x_{cw} + w_x/2)}{z_p}$$

$$bottom = \frac{-\text{near} \cdot (y_{cw} - w_y/2)}{z_p}$$

$$top = \frac{-\text{near} \cdot (y_{cw} + w_y/2)}{z_p}$$

(*near* has been negated to match the sign convention for z_p).

The matrix generated is (Silicon Graphics, 1991)

$$T_{N_E} = \begin{bmatrix} \frac{2 \cdot \text{near}}{\text{right} - \text{left}} & 0 & \frac{\text{right} + \text{left}}{\text{right} - \text{left}} & 0 \\ 0 & \frac{2 \cdot \text{near}}{\text{top} - \text{bottom}} & \frac{\text{top} + \text{bottom}}{\text{top} - \text{bottom}} & 0 \\ 0 & 0 & -\frac{\text{far} + \text{near}}{\text{far} - \text{near}} & -\frac{2 \cdot \text{far} \cdot \text{near}}{\text{far} - \text{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

This matrix performs the shear required for off-center projections, as well as scaling and translating the viewing frustum into the unit cube such that $-1 \leq x,y,z \leq 1$ after the division by w.

7.8 Undistorted-Screen_Normalized Transform (Convert to Pixel Coordinates)

The T_{US_N} transformation converts from the normalized viewing volume just described to device coordinates. The *viewport* call of GL implements T_{US_N} and is straightforward:

viewport (x_{min}, x_{max}, y_{min}, y_{max})

where all parameters are pixel coordinates.

The scaling/translation matrix for X and Y that accomplishes this is

$$T_{US_N} = \begin{bmatrix} \frac{x_{max} - x_{min}}{2} & 0 & x_{min} + \frac{x_{max} - x_{min}}{2} \\ 0 & \frac{y_{max} - y_{min}}{2} & y_{min} + \frac{y_{max} - y_{min}}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

This matrix translates the normalized window (after projection) so that the lower left corner is at the origin, then scales it to fit into the given viewport, and finally translates the scaled window so that its lower left corner is at x_{min}, y_{min}. The left- and right-eye views are typically mapped to different viewports for single-frame-buffer systems and then scanned out as separate video signals for display. For systems without distortion correction, this is the final transformation.

7.9 Screen_Undistorted-Screen Transform (Optical Distortion Correction)

Most HMDs (particularly those with wide fields of view) have some degree of optical distortion, which is a nonlinear warping of the virtual screen image. This optical aberration can be minimized through careful optical design, electronic prewarping of the image in the display circuitry, or by correcting it in the rendering process.

Meta Platforms, Inc.
Exhibit 1087
Page 017

Optical correction has the advantage of not reducing the frame rate, but is not always feasible due to other constraints, such as cost, weight, and minimization of other optical aberrations.[1] Electronic correction does not reduce the frame rate either, but is not available or feasible for many systems. Thus, although we would prefer to correct the distortion either optically or electronically, these are not always options, and we are forced to either live with it or correct it in the rendering process.

The Virtual Research *Flight Helmet* (which uses the LEEP optics) is a typical case in point. It has significant optical distortion and we know of no system for correcting it electronically. Without correction, lines that fall near the edge of the field of view are noticeably curved. Correcting the distortion in the rendering process requires predistorting the image by a function that is the inverse of the optical distortion function. If this is done correctly, the final image will appear undistorted. The problem with this approach, of course, is that it is computationally expensive, and tends to reduce the frame rate significantly. For this reason, most current systems that have significant distortion (including most of the systems in daily use at UNC) simply ignore the problem. We believe, however, that accurate rendering of scenes in VR will become more important in the future as precision tasks are undertaken with HMDs. With see-through HMDs in particular, the need to accurately register virtual objects with the real world will demand accurate rendering (Janin et al., 1993). What follows is a brief description of the distortion problem and one model for correcting it in software, with pointers to other papers on the subject.

As detailed in Robinett and Rolland (1992), in systems with optical distortion, the magnification of a point in the image is a function of its distance from the optical axis. If we neglect higher-order terms, this aberration can be modeled to third order as

$$r_v = mr_s + k(mr_s)^3 \qquad (9)$$

where r_v is the radial distance to the displayed point in image space, m is the *paraxial*[2] magnification of the optical system, r_s is the radial distance to the point in screen space, and k is the third-order coefficient of optical distortion. This assumes that the optical system is radially symmetric around an optical axis, which is true for the LEEP optics used in the Flight Helmet, but is not true of all optical systems. Figure 7 shows a simple model of the optics for a single eye in an HMD.

If k is positive, the magnification increases for off-axis points, and the aberration is called *pincushion distortion* (Figure 8); if k is negative, the magnification decreases, and it is called *barrel distortion.* Pincushion distortion is more common in HMD systems and will be assumed in what follows.

Note that if $k = 0$, there is no distortion and the virtual image of the screen seen by the user is just a linear magnification of the screen itself. This is what the graphics model typically assumes. The problem with distortion is not only that the image is warped, but also that the scale (i.e., magnification) is different as we move out from the center of the image. Thus, if we set our window parameters w_x and w_y to match the corners of the distorted image, the scale will be right at the corners and wrong in the center; i.e., objects in the center will be scaled down relative to their real sizes, since we used the inflated scale from the image corner [this can also introduce error into the projection window offsets (x_{cw} and y_{cw}) for systems with screens not centered on the optical axes]. On the other hand, if we use the paraxial magnification to determine the window parameters, object sizes will be correct in the center but will be stretched out in the periphery. On the whole, using paraxial or near-paraxial values seems to introduce less error than using the distorted values. The bottom line, though, is that there is no good way to approximate a cubic function (the distortion) with a linear function (linear scaling), and if the distortion is not corrected, objects will appear, grow, and shrink depending on their location in screen space.

Computing the inverse of the distortion function can be done a number of ways. A very accurate but time-

1. It can be done, however. For example, the sales literature for the CAE FOHMD quotes its distortion as less than 1.5% (CAE, 1986).

2. This term refers to the first-order model for optics which assumes that rays strike the lens at small angles. This approximation breaks down in real systems with finite apertures, resulting in *optical aberrations,* one of which is distortion.

Meta Platforms, Inc.
Exhibit 1087
Page 018

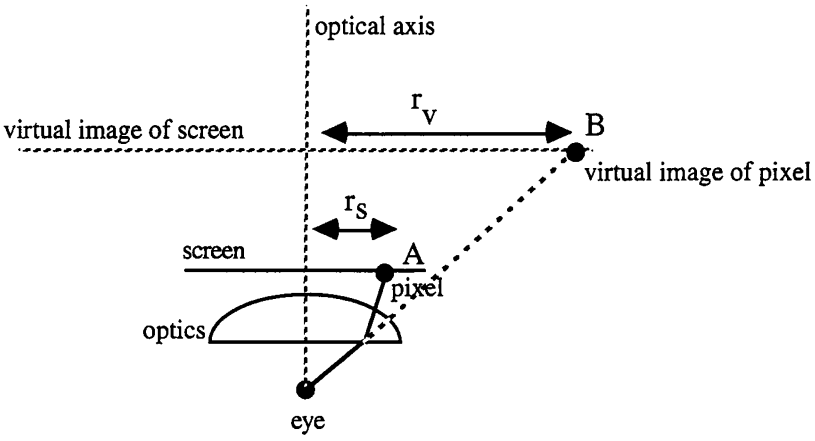


Figure 7. *Single-eye optics model.*

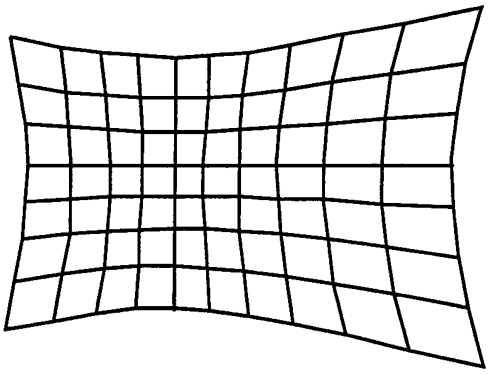


Figure 8. *Pincushion distortion.*

consuming method would be to use a ray-tracing program (such as Code V) to compute the distortion and its inverse for a set of r_s values, which could then be interpolated to find the required predistortion for any value of r_s. Another approach is to use the exact closed-form solution to the third-order equation, as is done in Rolland and Hopkins (1993). Finally, a third-order approximation to the inverse gives a reasonable fit for many systems, and is described in Robinett and Rolland (1992). The second method has been implemented at UNC for Pixel-Planes 5 by Anselmo Lastra, Jannick Rolland, and Terry Hopkins. We present the third method because of its algorithmic simplicity.

Predistortion is a 2D image warp that moves a point on the screen (x_s, y_s) to a new position (x_d, y_d). This warping can either be applied to polygon vertices or to each pixel; the efficiencies and complexities of each approach are beyond the scope of this paper.

To simplify the calculations, we can re-express Eq. (9) in the following way:

$$r_{vn} = r_{sn} + k_n r_{sn}^3 \tag{10}$$

Here, r_{vn} is the normalized radius in image space, r_{sn} is the normalized radius in screen space, and k_n is the normalized coefficient of optical distortion. The normalized coefficient of distortion gives the percentage distortion at some image radius r_{norm}, which is usually chosen to be the distance from the optical axis to one of the edges of the image. Because k_n is defined in terms of r_{norm} and because the choice of r_{norm} is somewhat arbitrary, one can have different values of k_n that describe the same system. The parameter k [from Eq. (9)] does not have this dependency, but is not as intuitive for describing distortion.

The inverse of Eq. (10) can be approximated with a third-order polynomial as shown in the following algorithm:

$(x_s, y_s) = (x - x_{axis}, y - y_{axis})$ express the point (x, y) relative to optical axis

$r_s = \sqrt{x_s^2 + y_s^2}$ calculate radius

$r_{sn} = \dfrac{r_s}{r_{norm}}$ normalize

$$r_{pdn} = r_{sn} + k_{pd} \cdot r_{sn}^3$$ apply inverse distortion (note: $k_{pd} < 0$)

$$r_{pd} = r_{norm} \cdot r_{pdn}$$ map back to pixel units

$$d = \frac{r_{pd}}{r_s}$$ calculate radial scaling factor for this point

$$(x_{sv}, y_{sv} = (d \cdot x_s, d \cdot y_s)$$ scale vector centered at optical axis

$$(x_d, y_d) = (x_{sv} + x_{axis}, y_{sv} + y_{axis})$$ translate back to screen coordinates

Thus, the parameters of the image warp algorithm are

- the position of the optical axis in screen coordinates: x_{axis}, y_{axis}
- the normalizing radius in pixels: r_{norm}
- the normalized distortion coefficient and the normalized predistortion coefficient: k_n and k_{pd}

These parameters depend on the specifications of the optics and the positioning of the display device relative to the optics.

This algorithm shrinks the image in a nonlinear fashion so that when it is distorted by the optics, it will match the paraxial model. In this case, w_x, w_y and x_{cw}, y_{cw} should be set to their paraxial values since predistortion will cancel out any change in these values induced by the distortion (both values for these parameters are given in Table 4). Since this scaling of the image leaves some of the frame buffer unused, an alternative method is to scale the predistorted image so that it fills the frame buffer as much as possible and to use the distorted window extents and offsets.

8 Parameters of the Display Transformation

The numerical values of the parameters of the display code tailor the code to a particular VR system. We give the display parameters of the UNC VR system as an example.

In particular, Table 4 gives the complete specification of all of the transformation parameters for the Vlib GL version for use with the Virtual Research *Flight Helmet.* Note that Table 4 includes only the static parameters that are known at startup time (a complete listing was given in Table 1). Also, some of the transforms listed have been inverted to their more intuitive form for the sake of clarity.

Many of the optical parameters listed were measured by Jannick Rolland of UNC, and some are derived in Rolland and Hopkins (1993). The value for k_{pd} was derived numerically. The paraxial and distorted figures are given for the window parameters since there is no single correct value for systems without distortion correction. Also, small variations in the manufacturing process of HMDs can change the optical parameters substantially, so the numbers given above should be taken as typical values rather than absolutes. Finally, in our non-see-through systems, we have found that a wide range of values for the window parameters will yield an acceptable 3D percept, which suggests that individual, perception-based calibration will still be necessary for certain applications.

It should be clear at this point that making a usable system involves many parameters with complex interactions. Table 4 is an attempt to list the parameters of most interest to a system designer without getting bogged down in some of the subtler details. More in-depth discussions of optical issues can be found in the papers already cited in this section.

The parameters given in Table 4, together with the series of transforms defined in this paper, define the visual display transform for the UNC VR software (Vlib) running on a Silicon Graphics VGX computer. This display transform takes points defined in Object coordinates and transforms them to Screen coordinates. This display algorithm should work for many current VR hardware configurations, provided appropriate values are supplied for the display parameters.

9 Conclusion

We have presented the complete visual display transform for virtual reality, as implemented on the UNC VR system. The considerable number of trans-

Meta Platforms, Inc.
Exhibit 1087
Page 020

Table 4. *Parameters of Vlib Display Transformation*

Transform	Parameters	Value in UNC VR system	Description of parameters
T_{W_O}	$\mathbf{v}_{W_O}$	(Object poses stored in model data)	Object's position in world
	$\mathbf{q}_{W_O}$		Object's orientation in world
	s_{W_O}		Object's scale in world
T_{W_R}	$\mathbf{v}_{W_R}$	Initially (0, 0, 0)	Room's position in world
	$\mathbf{q}_{W_R}$	Initially [(0, 0, 0), 1]	Room's orientation in world
	s_{W_R}	Initially 1	Room's scale in world
T_{R_TB}	$\mathbf{v}_{R_TB}$	(2.5, 2.0, 2.0) (m)	Tracker base's position within room
	$\mathbf{q}_{R_TB}$	[(0.5, 0.5, −0.5), −0.5]	Tracker base's orientation within room
T_{H_HS}	$\mathbf{v}_{H_HS}$	(0.0, 0.19, 0.03) (m)	Head sensor's position on the HMD
	$\mathbf{q}_{H_HS}$	[(0.5, 0.5, −0.5), 0.5]	Head sensor's orientation on the HMD
T_{H_E}	$\mathbf{v}_{H_E}$	Left: (−IPD/2, 0, 0) Right: (IPD/2, 0, 0) with IPD = 0.062 m	Position of eye CS relative to head CS (in meters)
	$\mathbf{q}_{H_E}$	Left: [(0, 0, 0), 1] Right: [(0, 0, 0), 1] since $\varnothing = 0$	Rotation of eye CS relative to head CS
T_{N_E}	FOV_h	77° w/distortion (66.2° paraxial)	Horizontal monocular field of view
	FOV_v	60.8° w/distortion (53.5° paraxial)	Vertical monocular field of view (assuming an eye relief of 25 mm)
	z_p	−1.18 m	Distance from eye to virtual image of screen (assuming an eye relief of 25 mm)
	x_{cw}	Left: −0.187 m (paraxial) −0.190 m (w/distortion) Right: 0.187 m (paraxial) 0.190 m (w/distortion)	Projection window center offsets (assuming IPD = 0.062 m)
	y_{cw}	Left: 0 m Right: 0 m	
	w_x	1.57 m paraxial, 2.09 m w/distortion	Projection window extents
	w_y	1.19 m paraxial, 1.38 m w/distortion	
	z_{near}, z_{far}	0.05 m, 1000 m	Near and far clipping planes
T_{US_N}	x_{min}, x_{max}	Left: 0, 640 Right: 640, 1280	Screen viewport x and y bounds in pixels[a]
	y_{min}, y_{max}	Left: 512, 1024 Right: 512, 1024	
T_{S_US}	k_n	0.1933	Normalized coefficient of optical distortion
	k_{pd}	−0.1	Normalized coefficient for predistortion
	r_{norm}	256 (pixels)	Normalizing radius in pixels[b]
	x_{axis}, y_{axis}	Left: (395.9, 256) (pixels) Right: (244.1, 256) (pixels)	Optical axis in screen coordinates (assuming 640 × 512 frame buffer viewport resolution)[a]

[a]For simplicity, these figures neglect the pixel cropping problem detailed in Rolland and Hopkins (1993).

[b]The normalizing radius used here is the distance in pixels from the optical axis to the top or bottom of the screen; the choice was arbitrary.

Meta Platforms, Inc.
Exhibit 1087
Page 021

forms and coordinate systems in VR requires a systematic method for dealing with them all, and we have presented our method.

The coordinate system graphs were used because they give an intuitive feel for the relationships between coordinate systems. The T_{A_B} notation for transforms was used because it is concise and because it provides a check on correctness by requiring subscripts of adjacent terms to match. The VQS representation was presented as a concise and useful alternative to 4×4 homogeneous matrices for many VR operations. Finally, we have supplied a complete specification of the display transform in the UNC VR system and have also given the numerical parameter values required by the display transform.

We believe that this software architecture for the visual display computation for virtual reality is sufficiently flexible that with different values for the display parameters, it can handle many different HMDs, trackers, and other hardware devices, and that it can be used in many different applications of virtual reality. This display algorithm is not tied to any particular display hardware, and can be implemented on any computer used to generate graphics imagery for virtual reality.

Acknowledgments

We thank the HMD and Pixel-Planes teams at UNC for creating the various components, hardware and software, of our VR system. We thank Fred Brooks for clarifying discussions on nomenclature. An early version of some parts of this display transform was done at NASA Ames Research Center, and we thank Scott Fisher, Jim Humphries, Doug Kerr, and Mike McGreevy for their contributions. We thank Ken Shoemake for help with quaternions. We thank Jannick Rolland for educating us about optics, for help defining the formulas describing optical distortion, and for optical measurements. We thank Anselmo Lastra and Terry Hopkins for discussions about implementation of optical distortion. We thank the Zentrum für Kunst und Medienteknologie in Karlsruhe, Germany, for supporting the port of Vlib to the Silicon Graphics VGX. We thank the Banff Centre for the Arts in Canada, where a draft of this paper was written in a pleasant studio in the woods. Finally, we thank the reviewers for their helpful comments on improving this paper. This research was supported by the following grants: ARPA DABT 63-93-C-0048, NSF Cooperative Agreement ASC-8920219, and ARPA: "Science and Technology Center for Computer Graphics and Scientific Visualization," ONR N00014-86-K-0680, and NIH 5-R24-RR-02170.

References

Blanchard, C., Burgess, S., Harvill, Y., Lanier, J., Lasko, A., Oberman, M., & Teitel, M. (1990). Reality built for two: A virtual reality tool. *Proceedings of the 1990 Workshop on Interactive 3D Graphics,* 35–36.

Brooks, F. P., Jr. (1989). Course #29: Implementing and interacting with real-time virtual worlds. *Course Notes: SIGGRAPH '89.*

Buchroeder, R. A., Seeley, G. W., & Vukobradatovich, D. (1981). Design of a catadioptric VCASS helmet-mounted display. Optical Sciences Center, University of Arizona, under contract to the U.S. Air Force Armstrong Aerospace Medical Research Laboratory, Wright-Patterson Air Force Base, Dayton, Ohio, AFAMRL-TR-81-133.

CAE. (1986). Introducing the visual display system that you wear. CAE Electronics, Ltd., C.P. 1800 Saint-Laurent, Quebec, Canada H4L 4X4.

Caudell, T. P., and Mizell, D. W. (1992). Augmented reality: An application of heads-up display technology to manual manufacturing processes. *Proceedings HICCS 1992,* 659–669.

Cooke, J. M., Zyda, M. J., Pratt, D. R., and McGhee, R. B. (1992). NPSNET: Flight simulation dynamic modeling using quaternions. *Presence, 1*(4), 404–420.

Craig, J. (1986). *Introduction to robotics.* Reading, MA: Addison-Wesley.

Fisher, S. S., McGreevy, M., Humphries, J., & Robinett, W. (1986). Virtual environment display system. *Proceedings of the 1986 Workshop on Interactive 3D Graphics,* 77–87.

Foley, J., van Dam, A., Feiner, S., & Hughes, J. (1990). *Computer graphics: Principles and practice* (2nd ed.). (pp. 222–226). Reading, MA: Addison-Wesley.

Fuchs, H., Poulton, J., Eyles, J., Greer, T., Goldfeather, J., Ellsworth, D., Molnar, S., Turk, G., Tebbs, B., & Israel, L. (1989). A heterogeneous multiprocessor graphics system using processor-enhanced memories. *Computer Graphics: Proceedings of SIGGRAPH '89, 23*(4), 79–88.

Funda, J., Taylor, R. H., Paul, R. P. (1990). On homoge-

Meta Platforms, Inc.
Exhibit 1087
Page 022

neous transforms, quaternions, and computational efficiency. IEEE Trans. on robotics and automation. V6n3, June.

Janin, A. L., Mizell, D. W., & Caudell, T. P. (1993). Calibration of head-mounted displays for augmented reality applications. *IEEE Virtual Reality Annual International Symposium,* Seattle WA, 246–255.

Holloway, R. L. (1987). *Head-mounted display Technical Report.* Technical Report TR87-015, Dept. of Computer Science, University of North Carolina at Chapel Hill.

Holloway, R., Fuchs, H., & Robinett, W. (1991). Virtual-worlds research at the University of North Carolina at Chapel Hill. *Proceedings of Computer Graphics '91,* London, England, 181–196.

Paul, R. (1981). *Robot manipulators: Mathematics, programming, and control.* Cambridge, MA: MIT Press.

Pique, M. (1980). *Nested dynamic rotations for computer graphics.* M.S. Thesis, University of North Carolina, Chapel Hill, NC.

Robinett, W., & Holloway, R. (1992). Implementation of flying, scaling, and grabbing in virtual worlds. *ACM Symposium on Interactive 3D Graphics,* Cambridge, MA, March.

Robinett, W., & Rolland, J. P. (1992). A computational model for the stereoscopic optics of a head-mounted display. *Presence, 1*(1), 45–62. Also UNC Technical Report TR91-009.

Rolland, J. P., Ariely, D., & Gibson, W. (1993). Towards quantifying depth and size perception in 3D virtual environments. *Presence,* in press. Also Technical Report TR93-044, Dept. of Computer Science, University of North Carolina at Chapel Hill.

Rolland, J. P., & Hopkins, T. (1993). *A method of computational correction for optical distortion in head-mounted displays.* Technical Report TR93-045, Computer Science Department, University of North Carolina at Chapel Hill. (Available via anonymous ftp from ftp.cs.unc.edu.)

Shoemake, K. (1985). Animating rotations using quaternion curves. *Computer Graphics: Proceedings of SIGGRAPH '85,* 245–254.

Silicon Graphics. (1991). *GL reference manual.* Silicon Graphics, Inc., Mountain View, CA.

Sutherland, I. E. (1968). A head-mounted three-dimensional display. *1968 Fall Joint Computer Conference, AFIPS Conference Proceedings, 33,* 757–764.

VPL. (1989). *VPL EyePhone Operations Manual.* VPL Research, 656 Bair Island Rd., Suite 304, Redwood City, California 94063, p. B-4.

Woodson, W. E. (1981). *Human factors design handbook.* New York: McGraw-Hill.

Meta Platforms, Inc.
Exhibit 1087
Page 023