Using Tiled Head-Mounted Displays with a Single Video Source

Brian Douglas Strege

A paper submitted to the
Computer Science & Electrical Engineering Department
in partial fulfillment of the requirements for the M.S. degree at
University of Maryland Baltimore County

April 2009

Certified by: Dr. Marc Olano

Advisor's Signature:_____ Date_____

ABSTRACT

Title of Paper: Using Tiled Head-Mounted Displays with a Single Video Source

Brian Douglas Strege, Computer Science, 2008

Paper directed by: Dr. Marc Olano, Associate Professor
Department of Computer Science and
Electrical Engineering

Head-mounted displays (HMDs) are used to create a sense of immersion known as “virtual reality” for various applications, such as flight simulation, data visualization, or simply entertainment. Recently, some HMDs are using multiple screens “tiled” around each eye in order to give the user a greater sense of peripheral vision by providing for a wider field of view. Currently, only specialized applications that output individual video streams to each of these “microdisplays” can be used with tiled HMDs. This paper describes a system extending their use to off-the-shelf 2D and 3D applications. The system was developed using Graphics Processing Units (GPUs) as a platform, and creates multiple video streams for each of the microdisplays, all from a single source video stream. This is performed in real-time, with frame rates high enough to run interactive applications on a tiled HMD.

ACKNOWLEDGMENTS

Special thanks to Yuval Boger for the piSight™ panel alignment files, Nathan Villagaray-Carski for the converted Oval Office model, and Peter Sharkey for the original Oval Office model.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

The term “virtual reality” has entered the modern lexicon to describe any system which provides a convincing sense of immersion in some alternate computer-generated world. Virtual Reality (VR) systems find applications in a number of fields, where their uses range from real-world simulations such as flight training, to completely artificial realities such as immersive video games. For real-world simulations in particular, it is easy to see the value in training someone for a potentially dangerous situation inside of a virtual world as opposed to reality which may put the user in harm’s way.

1.1 Overview

For a virtual reality system to successfully immerse its user into a virtual world, a number of the user’s physical senses must be affected. While various hardware systems exist to provide audio and even tactile information, the centerpiece of any virtual reality system is its ability to provide visual information. Many virtual reality systems today use Head-Mounted Displays, or HMDs, to provide this visual information. The earliest computer-based HMD was a very large unwieldy system comprised of two heavy CRTs placed on either side of the user’s head, having the video reflected to their eyes via half-silvered mirrors, and was so heavy it had to be suspended from the ceiling [1]. This device earned itself the nickname “Sword of Damocles” – presumably due to the

intimidating appearance of its machinery suspended from the ceiling – and with it the HMD was born [2].

As optical technologies progressed, HMDs were able to become lighter and less cumbersome, as well as have much better resolution than they did in the days when only CRT displays were available. Another aspect of HMD technology that has improved over time is their visible field of view, allowing the system to provide the user some sense of peripheral vision instead of only very narrow "tunnel-vision," as can be seen in Figure 1.1. However, as their field of view is widened, HMDs tend to have a significant amount of geometric distortion around the periphery of the display [3]. Some modern HMDs are combating this distortion while offering wider fields of view by utilizing multiple small displays, or "microdisplays," for each eye as opposed to the usual single display per eye. Each microdisplay can then be tilted to its proper orientation and "tiled" next to one another so as to create a contiguous visual area that curves around each of the user's eyes, allowing for much greater perceived peripheral vision without the distortion of a single display.

1.2 Objective

Currently, each microdisplay in tiled HMDs must receive their own distinct video signal; every application that wants to be used in tandem with the HMD must be specialized so that it can provide individual video sources for each microdisplay. Therefore, a potential user of a tiled HMD would need to provide a large number of video sources in order to be able to drive the display device. This translates into the user needing much more hardware of their own: additional video cards to provide the extra source outputs, and perhaps even racks of additional computer systems in order to accommodate the additional video cards. All of this extra hardware greatly drives up the overhead cost to be able to use a tiled HMD, and keeps their use out of reach for the average computer user. The ability to use tiled HMDs with applications that provide only a single video

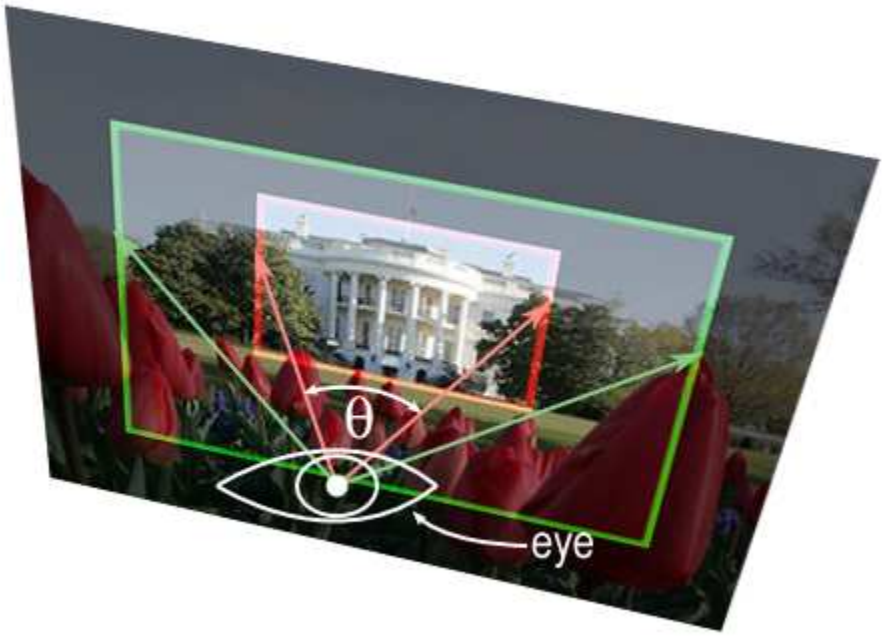

FIG. 1.1. Example illustrating HMD field of view. The angle θ shown is the horizontal field of view corresponding to the smaller viewable image result. As HMD technology has progressed, their field of view has increased to allow the user to see more at the periphery of their vision, resulting in a much more immersive experience.

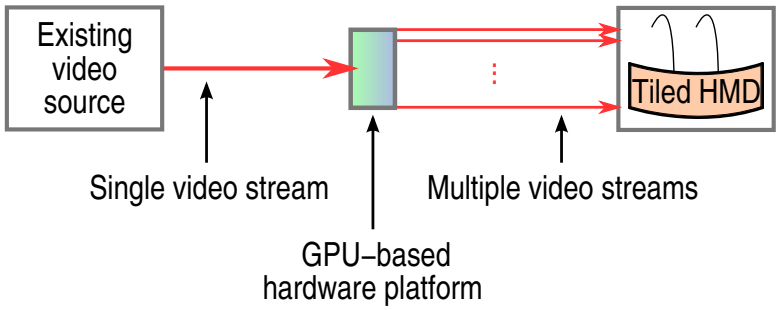

FIG. 1.2. Schematic of tiled HMD system using GPU-based hardware platform. The platform would take a single video stream as an input, and output as many video streams as the tiled HMD has microdisplays.

source (or possibly a stereo source) is clearly desirable, as this would greatly reduce their overhead cost of operation, increasing their potential user base. Allowing tiled HMDs to be driven by a single video source would allow them to be used with any off-the-shelf application; everything from video games to DVDs would then be compatible.

The research described in this paper creates a framework which expands the functionality of tiled HMDs, so that they may be used with a single video source. Algorithms have been developed using a modern Graphics Processing Unit, or GPU, as a platform. The use of a versatile platform such as a GPU allows the finished product to be easily adapted into a useable piece of hardware. This piece of hardware would sit between any existing video source and a tiled HMD, as shown in Figure 1.2, so as to allow functionality with that existing source.

Chapter 2

BACKGROUND AND RELATED WORK

There has been quite a bit of work done which is related to this research, both in terms of HMD research as well as general computer graphics research which will be useful to this system. A description of the relevant previous work and background information is given in this section.

2.1 Virtual Reality

The first HMD was a large unwieldy piece of equipment nicknamed the Sword of Damocles [1, 2]. This first HMD, like many other pieces of display equipment that followed it, was part of a larger VR system. A sense of immersion was created by tracking the user's head movements and causing them to directly and immediately affect what was being displayed to their eyes, as if they were "looking" around this 3D virtual world. An early discovery from this system is that having accurate representation of stereo vision in an HMD greatly increases the realism of the virtual world, as opposed to simply displaying the same image viewing from a single point in that world to both of the user's eyes.

It is worth noting that HMDs are not the only devices used to convey the visual data in VR systems - certain applications are more suited to other types of displays. One such alternate visual system is the relatively simple Fish Tank VR, where stereo images of a 3D scene are displayed on a regular monitor, the user wears special stereo viewing glasses to have the correct halves of the

stereo image shown only to the corresponding eye, and the user's head is tracked to create a sense of immersion by having movements affect what is actually being displayed [4]. Another system design is known as the CAVE, a recursive acronym for CAVE Automatic Virtual Environment [5]. This system is essentially a theater that a user walks into, where the virtual environment is projected onto the surrounding walls. The user must still wear stereo glasses and a tracking device. More recent CAVE-like systems have strived to eliminate the user-worn equipment altogether while still providing stereo imagery and tracking, resulting in a less constrained experience for the user [6].

On a different side of this discussion, it is also worth noting that some HMDs are not intended for VR; they are not attempting to create an immersive world for a user. Rather, these systems are intended to *augment* reality by overlaying artificial computer-generated information on top of the real world. Certain systems augment reality by capturing real-world images and rendering them behind computer-generated information, using image capturing devices attached to a head unit along with the displays for each eye. The captured real-world images are then merged with the computer-generated ones, and displayed to the user. These systems are known as "video see-through HMDs," in contrast to "optical see-through HMDs," which let the user directly see through the screens into the real world. This method of augmented reality relies on optical combiners – essentially half-silvered mirrors – for the displays, in order to allow the user to see through them while remaining partially reflective to display the computer-generated imagery [7]. Systems such as these have found use in creating Heads-Up Displays (HUDs) which relay pertinent information to a user in real-time while not restricting their vision [8].

Also as mentioned earlier, the visible field of view of HMDs intended for use in immersive VR applications has become much better over the years. While the Sword of Damocles had only a 40 degree field of view, certain modern tiled HMDs have a significantly wider 150 degree field of view [9]. A much greater sense of immersion is created by these tiled HMDs due to the increased visual information at the user's periphery. An academic audience of 1,381 people participated in a recent survey about satisfactory HMD specifications, and the results indicated that a field of view

that is at least 120 degrees would be adequate for most academic purposes [10].

2.2 Graphics Processing Units

GPU technology has also advanced greatly over the years. Of particular interest to this research is use of the inherent programmability of modern GPUs, using a process called *shading*. Shading was first introduced a decade ago with the creation of a proprietary system called PixelFlow, which utilized its own specialized shading language and could run procedures written in this language to render high-quality images in real-time [11]. The GPU platform is already being used to help drive new types of VR devices, such as immersive stereoscopic displays, via these programmable shaders [12]. All modern GPUs execute shading procedures, which are now written in more standardized languages and with well-defined memory and instruction requirements to which many different GPUs conform [13].

2.3 Resampling

The algorithms described in this paper map from the existing video source to the multiple video streams needed for the tiled HMD. Whatever these mappings may be, it is immediately obvious that the resolutions of the tiles and the locations they are mapped to may not match. Converting from one resolution to another is known as *resampling*, and the best technique to accomplish this is called *filtering;* making each pixel of the microdisplay the weighted sum of the original pixels, according to some weighting scheme known as the *filter kernel.* When we are discussing filtering, we will specifically be talking about *texture filtering* with our *source image* – the visual data before it is mapped to the microdisplays – as the texture. *Bilinear filtering* is a very common method which samples the four texels nearest to the exact point on the texture to be mapped, and combines them via weighted average based on their distance to that point. Another technique used in texture filtering is *mipmapping*, where a texture is stored along with sequentially smaller copies of itself,

halving its dimensions all the way down to a single pixel. Mipmaps are used to speed up rendering and reduce image aliasing. *Trilinear filtering* is an extension of bilinear filtering which interpolates between the two closest mipmap levels. Trilinear filtering is a very common filtering method, is quite easy to implement, and is readily available in many development platforms.

In physics, if some physical property of an object is directionally dependent – i.e. results in different values when measured along different axes – it is referred to as an *anisotropic* property. There are many physical properties which can exhibit anisotropy, but a simple one to visualize is the change in appearance of certain materials when viewed from different angles. For example, a flat surface of brushed metal will appear darker or lighter based upon the viewing angle, even though the lighting in the room has not changed. In terms of appearance, the brushed metal is an *anisotropic surface*. In computer graphics, when a texture is being mapped at a steep glancing angle to the viewer, visual artifacts can form; angle-dependent effects such as these are referred to as *anisotropic effects* [14]. Trilinear filtering does not take into account any angle information from the mapping – it is not an *anisotropic filter* – so it may not perform well during mappings involving steep glancing angles. This may very well be the case with tiled HMDs, as the microdisplays towards the periphery of the eye may be at quite steep angles so as to stretch around the full range of the user's vision. With that in mind, more robust filtering techniques should be explored for use with tiled HMDs.

One such filtering technique called Elliptical Weighted Average (EWA), is a very accurate resampling technique for arbitrary mappings that may involve steep glancing angles [15]. EWA works by treating each mapped pixel as an ellipse whose major and minor axes are directly computed based upon the angles of the mapping. It uses a Gaussian filter, which is applied over the entirety of this ellipse. However, despite its accuracy, EWA is quite an elaborate and costly algorithm, and would be difficult to incorporate into a real-time system. Also, it would be quite difficult to implement in such a way as to be executed on the GPU, e.g. inside of a *pixel shader*. A system called Feline (Fast Elliptical Lines) was developed to approximate EWA visually, while requiring

much less computation [16]. Feline is built on top of an existing trilinear filtering engine, and is a much simpler algorithm than EWA. Since trilinear filtering is inherently available in modern GPUs, Feline would seem a good fit to be implemented inside of a pixel shader.

EWA and Feline both use ellipses to cover the swath of texture area – in our case, source image area – which is to be sampled. As the glancing angles get steeper, the ellipses will get more eccentric, and this will result in more area being sampled as seen in Figure 2.1. In contrast to EWA, Feline does not apply a filter over the entire area of this ellipse. Instead, Feline samples the area defined by these ellipses by using a certain number of "probes" along the length of the ellipse, as seen in Figure 2.2. These probes in turn use trilinear filtering to determine their own results, which are then combined by weighted average, using Gaussian weights. By using trilinear filtering to sample the ellipse at only a few locations, Feline greatly cuts down on computation as compared to the exhaustive EWA technique.

The location of the probes is determined by first calculating the major and minor radii of the ellipse, and the number of probes used is determined by the ratio of the radii. To determine the radii of each ellipse, as well as the eccentricity, Feline must be told the rates of change in texture space with respect to screen space at each pixel. More specifically, using the convention u, v to represent coordinates in texture space, and x, y to represent coordinates in screen space, the partial derivatives $\partial u/\partial x$, $\partial u/\partial y$, $\partial v/\partial x$ and $\partial v/\partial y$ must be known to Feline in order for it to execute.

The benefits of using Feline over simple trilinear filtering can be seen in Figures 2.3 and 2.4. Using only trilinear, the text applied to the plane at a glancing angle appears quite blurry, while the text filtered using Feline is clear.

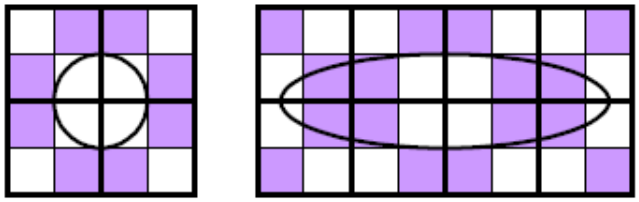

FIG. 2.1. Feline: example showing ellipses with different eccentricities over a texture map. Image taken from “Feline: Fast Elliptical Lines for Anisotropic Texture Mapping,” by McCormack et al.

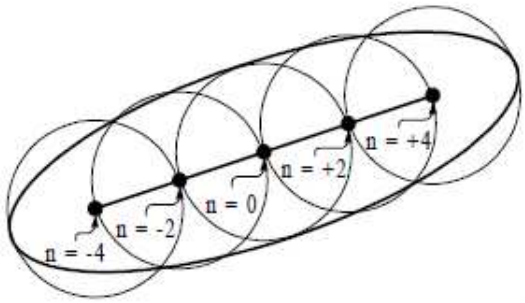

FIG. 2.2. Feline: example showing probes over an ellipse. Note that n is used to step through the probes. Image taken from “Feline: Fast Elliptical Lines for Anisotropic Texture Mapping,” by McCormack et al.

FIG. 2.3. Feline: using trilinear filtering alone produces blurry text. Image taken from “Feline: Fast Elliptical Lines for Anisotropic Texture Mapping,” by McCormack et al.

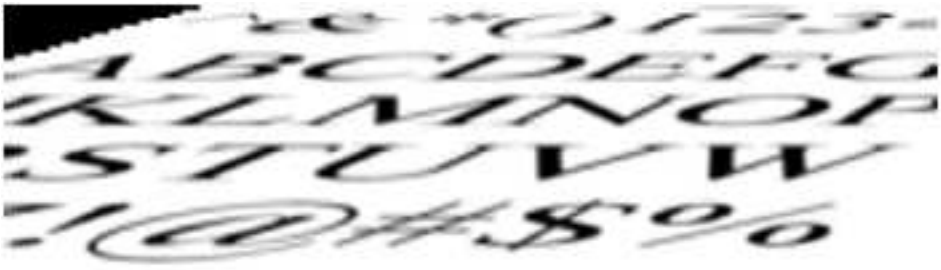

FIG. 2.4. Feline: using Feline produces clear text. Image taken from “Feline: Fast Elliptical Lines for Anisotropic Texture Mapping,” by McCormack et al.

Chapter 3

APPROACH

There are a number of technical challenges in mapping a single video source to the individual microdisplays of a tiled HMD. For the duration of this paper, we will consider these problems with just one eye, assuming in any implementation of the system there would be distinct video sources for each eye – i.e. a stereo source.

3.1 Problem Description

Before we can begin describing the implementation details of our system, we must clearly define the technical problems that we will encounter.

3.1.1 Pixel Map

The image from the original video source must comprise the entire field of view of the tiled HMD, so that there can be image data displayed around the entire periphery of the user's vision. Due to the curvature of the microdisplays around each eye, there is not a one-to-one mapping of pixels from the source image to the microdisplays. As can be seen in Figures 3.1 and 3.2, each pixel in the microdisplays will correspond to a certain region of pixels in the rendered image, depending on the physical position of the pixels in the microdisplays.

There is more than one way to create this mapping, as the orientation of the microdisplays

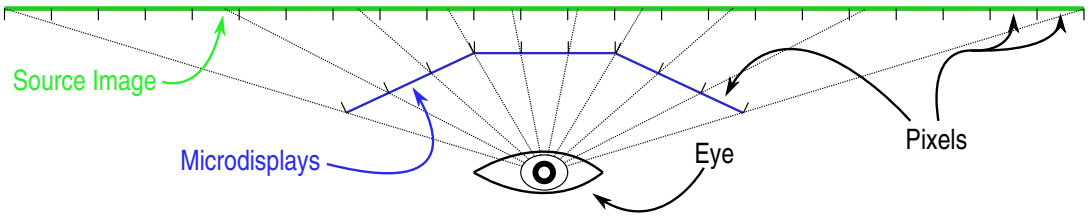

FIG. 3.1. Illustration of the mapping of pixels from the microdisplays to the source image, from the top down. Note that at the periphery of the user's vision a single pixel of the microdisplay corresponds to many more pixels of the source image than that of the pixels near the center.

says nothing of where the source image actually lies. Where this source image is placed will greatly affect the mapping, as it will determine exactly where the mapped regions of each microdisplay will reside. Also, the dimensions of the source image must be sized so as to allow the microdisplay mappings to cover a large portion of this original image. If the microdisplays only covered a small portion of the source image, much of the original data would not be seen by the user, and the immersive effect of the peripheral tiles may be lost. Finally, the mapping should be able to account for potential lens distortion of the microdisplays themselves. Treating each microdisplay as a plane will not necessarily be an accurate representation of the physical hardware being used.

3.1.2 Image Resampling

Another consideration is the fact that the source image can be any arbitrary resolution, as well as aspect ratio. As stated earlier, the pixels of the microdisplays will not directly correspond to the pixels of the source image, meaning that some form of resampling will have to be performed during the execution of the mapping on the GPU. Trilinear filtering was tested, as well as a pixel shader-based implementation of the Feline algorithm.

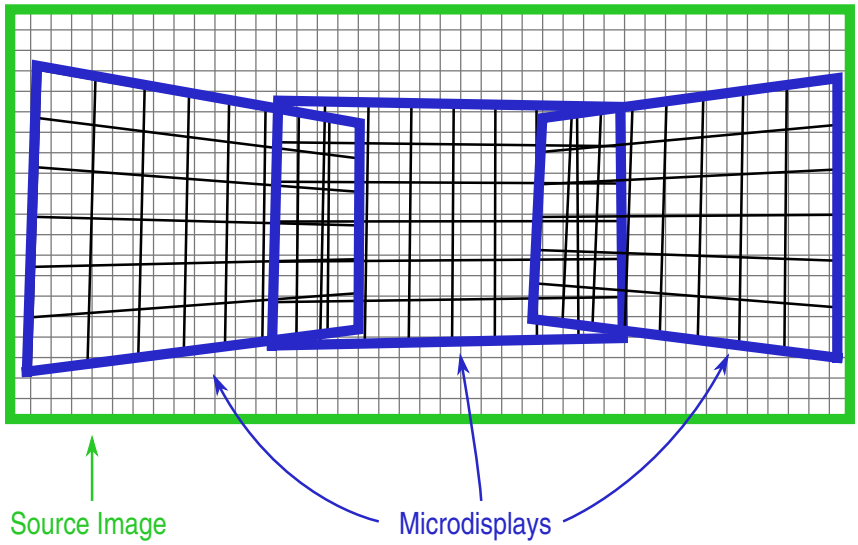

FIG. 3.2. Illustration of the mapping of pixels from the microdisplays to the source image, from the eye's perspective. Note that at the periphery the covered regions "fan out" and become non-rectangular in terms of the original pixels.

3.2 Implementation

In order to develop an implementation of our mapping system, a number of things must first be defined. First and foremost, the size and orientation of each individual microdisplay must be precisely defined, as well as the location of the eye. Next, the size and orientation of the source image must be chosen, preferably in such a way to make the microdisplays all map inside of it – i.e. have nothing map out-of-bounds. Once this is complete, the microdisplays can be projected onto the source image, and the pixel map calculations can be made. Each pixel of each microdisplay will be projected onto the source image, and its mapped location will be stored. Finally, once all of the pixel maps have been created, the microdisplays can receive inputs from a single source image by simply using the mappings. All of these steps, in addition to other elements of our implementation, are described in detail in this section.

3.2.1 Description of Microdisplays

The first step that must be taken in any implementation of pixel maps is to mathematically define exactly where in space a microdisplay lies. For now, we will consider each microdisplay to be a finite flat plane. The easiest method of defining where in the world each microdisplay lies is to simply keep track of its four corner points. In order to do this, we will require the exact width and height of each microdisplay – including any anomalies due to the manufacturing of the hardware. Since a height and width by itself means nothing in a world of infinite space, we will use the convention established by the HMD chosen for use on this project (discussed later in Chapter 4), namely that each microdisplay is at a distance of 1 from the eye. This assumes that all microdisplays have a common focal point – namely, the eye – and for the sake of simplicity, we will use this focal point as the origin of our world. Lastly, we will also require the rotational angles of each microdisplay about each axis (i.e. the pitch, roll, and yaw), again to exact manufacturing specifications. Knowledge of the distance between each eye would be required for stereo vision, but since we are dealing with distinct sources for each eye this will not be needed.

In the implementation, defining a microdisplay first starts by creating and storing the four corner points of a given microdisplay, without rotation. If a given microdisplay, m, has a width of w and a height of h, the corner points of the non-rotated version of m, denoted n, would be defined in space as follows:

$$n_{TL} = \begin{bmatrix} -\frac{w}{2} \\ \frac{h}{2} \\ -1 \end{bmatrix}, n_{TR} = \begin{bmatrix} \frac{w}{2} \\ \frac{h}{2} \\ -1 \end{bmatrix}, n_{BL} = \begin{bmatrix} -\frac{w}{2} \\ -\frac{h}{2} \\ -1 \end{bmatrix}, n_{BR} = \begin{bmatrix} \frac{w}{2} \\ -\frac{h}{2} \\ -1 \end{bmatrix}$$

where a subscript of TL denotes a *top left* corner point, and so on. Note that the z coordinate of all of the corner points is -1 as opposed to 1. This maintains the distance of 1 from the origin, but makes some of the math a bit simpler later on. Now the microdisplay must be rotated to its proper

orientation. In order to do this, we simply must rotate each corner point with the same sequence of rotation matricies. If the angles of rotation for the microdisplay m are θ_r, θ_p and θ_w, for roll, pitch and yaw, respectively, then the rotation matricies are as follows:

$$R_r = \begin{bmatrix} cos(\theta_r) & sin(\theta_r) & 0 \\ -sin(\theta_r) & cos(\theta_r) & 0 \\ 0 & 0 & 1 \end{bmatrix}, R_p = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta_p) & sin(\theta_p) \\ 0 & -sin(\theta_p) & cos(\theta_p) \end{bmatrix},$$

$$R_w = \begin{bmatrix} cos(\theta_w) & 0 & -sin(\theta_w) \\ 0 & 1 & 0 \\ sin(\theta_w) & 0 & cos(\theta_w) \end{bmatrix}$$

To rotate the corner points, the rotation matricies are simply multiplied with the non-rotated points. Using the top left corner of m as an example:

$$m_{TL} = R_r R_p R_w n_{TL}$$

These calculations are performed on each microdisplay to store all of the proper corner point locations. A visual representation of the data computed up to this point on a three-microdisplay example can be seen in Figure 3.3, where each microdisplay is shown as a plane with a different color, and the world's axes are shown with the origin defined as the location of the user's eye. This particular example set of microdisplays will be used frequently for the duration of this paper.

3.2.2 Location of Source Image

The next step in the process is to determine where the source image lies in world space. Like the microdisplays, the source image can be thought of as a plane. However, the only information available about the source image is its resolution, and since that is not related to the resolutions of

the microdisplays, the only information that can be gathered from this is the source image's aspect ratio. The location of the source image will ultimately decide where the pixels get mapped, so the method used to determine this will have a profound effect on the resulting images.

We align the source image with the axes, so that the normal coming from the center of the source image goes directly through the eye. This will keep the image directly in front of the eye – much as the microdisplays were before they were rotated – in a logical relationship with the parameters of the world surrounding it. However, unlike the microdisplays, the source image's distance from the eye will not be 1. Instead, it will be chosen to be the closest it can be to the eye without cutting through any of the microdisplay planes. As can be seen in Figure 3.4, the microdisplays at the sides "stick through" the microdisplay in the center, so the source image in this case would come as close to the eye as possible without intersecting a microdisplay plane. Therefore, the z-coordinate of the source image plane, s_z will be the minimum tranformed z-coordinate (since we are dealing with negative values) in all of the microdisplay planes. To determine this, we can simply check all of the corner points of the microdisplays, and set s_z to be the minimum z-coordinate found.

Now that s_z has been determined, the height and width of the source image in world space must be calculated. Let s_x and s_y denote *half* of the height and width of the source image, respectively – half so that the corner points of the source image can be defined as follows:

$$s_{TL} = \begin{bmatrix} -s_x \\ s_y \\ s_z \end{bmatrix}, s_{TR} = \begin{bmatrix} s_x \\ s_y \\ s_z \end{bmatrix}, s_{BL} = \begin{bmatrix} -s_x \\ -s_y \\ s_z \end{bmatrix}, s_{BR} = \begin{bmatrix} s_x \\ -s_y \\ s_z \end{bmatrix}$$

The method used to calculate s_x and s_y will be to make the source image as small as possible, while having no mapped pixels from the microdisplays fall out-of-bounds of the source image, and also while maintaining its aspect ratio. This will be accomplished by determining the polygons that

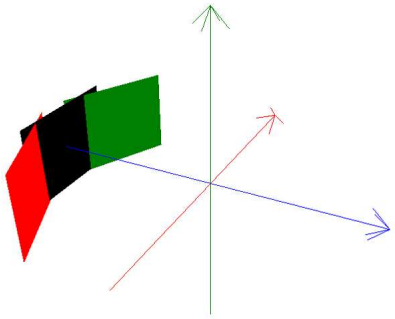

FIG. 3.3. View of the three microdisplay planes in world space. The microdisplays are shown as different colors to distinguish them from one another. The x axis is red, y green and z blue. The eye is positioned at the origin, looking straight down the z axis in the negative direction; towards the microdisplays.

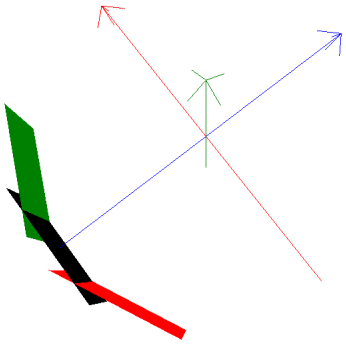

FIG. 3.4. View of the three microdisplay planes from above. Note that the microdisplay planes cut into each other; this is an overlap which will result in a seamless image when viewed with the actual hardware. The source image will be placed as close to the eye as possible without intersecting any of the microdisplay planes.

each microdisplay will map to when projected onto the source image, and then "fitting" the source image to these projections. Since the z value of the source image is already known, determining the projections of the microdisplay planes is rather simple. Again, this can be done on the basis of performing calculations only on the corner points of the microdisplays. Each corner point will have a ray cast through it, originating at the eye (the origin), as seen in Figure 3.5. The resulting projected point will be the point on that ray which has a z value equal to s_z. The basic parametric equation for a line between two points a and b in 3-d space is as follows:

$$l(t) = ta + (1 - t)b$$

However, since we are always casting our rays from the origin, let b be zero. Therefore, a ray cast through any corner point of any microdisplay, m, will be defined by the following extremely simple formula:

$$r(t) = tm$$

This can be used to determine the x and y values of the projected point, denoted p, when the z value is s_z:

$$\begin{aligned} & r(t) = tm = p \\ & \Rightarrow tm_z = p_z = s_z \\ & \Rightarrow t = \tfrac{s_z}{m_z} \\ & \Rightarrow p_x = \tfrac{s_z}{m_z} m_x, p_y = \tfrac{s_z}{m_z} m_y \end{aligned}$$

Once this is done for every corner point, we will have determined the corner points of the projections of all of the microdisplays onto the source image. As seen in Figure 3.6, this has projected the microdisplays onto the source image plane – since we have not yet determined the

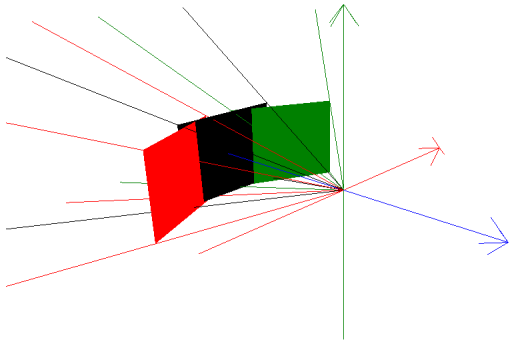

FIG. 3.5. View of raycasting through the corners of the three microdisplay planes. Since the HMD has a large field of view, these rays "fan out" quite a bit. Therefore, the farther away the source image is placed, the larger it must be to prevent any microdisplay pixels from being mapped out-of-bounds.

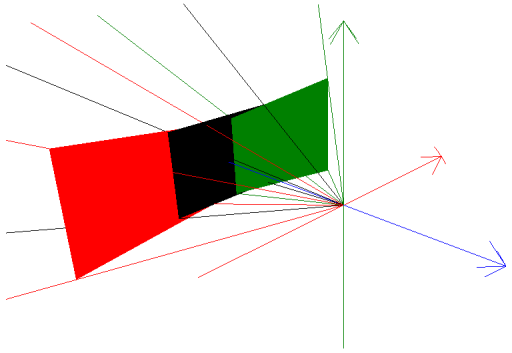

FIG. 3.6. View of the projections of the three microdisplay planes. The microdisplay image areas have been essentially "flattened" onto a plane that is some distance s_z from the eye.

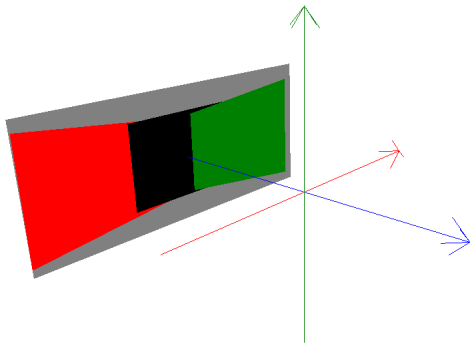

FIG. 3.7. View of the source image shown with the projections of the three microdisplay planes. The source image, shown in grey, has been made as small as possible while still fully containing all of the microdisplay projections, as well as maintaining its original aspect ratio. This image most clearly shows the mapping results as a whole, since the regions of the source image that the microdisplays map to can be easily seen along with the source image itself.

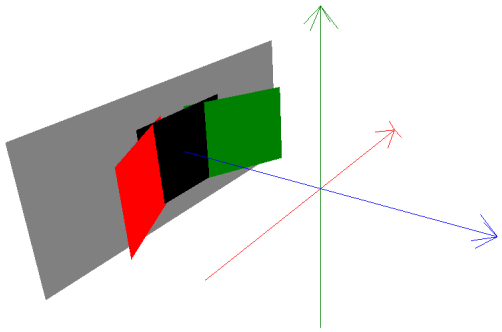

FIG. 3.8. View of the source image shown with the three microdisplay planes. This illustrates how large the source image must be in comparison to the original microdisplay planes, so as to prevent any out-of-bounds mapping.

boundaries of the source image, it is not shown. Now that the corner points have been found, s_x is chosen to be the largest x absolute value of these projected points, and likewise with s_y. The source image has now been fitted to the projections, and the final step is to increase either s_x or s_y, depending on the specific configuration, so as to maintain the aspect ratio of the source image. The source image concept can be seen in Figure 3.7, which shows the source image plane fitted to the microdisplay projections. Notice that the source image has been made as small as possible in the horizontal direction, while still containing all of the microdisplay projections (the vertical direction is not tightly fit, in order to maintain the original aspect ratio). Figure 3.8 shows the source image plane along with the original microdisplay planes, which illustrates how large the source image must be to prevent any out-of-bounds pixel mapping from the microdisplays.

3.2.3 Pixel Map Generation

Now that the coordinates of the microdisplays and source image have been determined, work can begin on computing the pixel map. What is required is an exaustive computation of the exact sub-pixel location in the source image corresponding to every pixel of each microdisplay. In other words, rays will be cast from the eye through the center of each microdisplay pixel, and the resulting pixel in the source image that it hit will be recorded.

In the previous section, it was seen that given a point – microdisplay corner points, at the time – it is very easy to calculate the location on the source image which a ray cast through that point will hit. Therefore, the last part that remains is finding the exact center point of each microdisplay pixel in world space, since they are the points that the rays will be cast through. Figure 3.9 shows a microdisplay, m, in world space, as well as denoting some relevant vectors to the pixel map. The corner points are initially known, all other vectors and points are unknown.

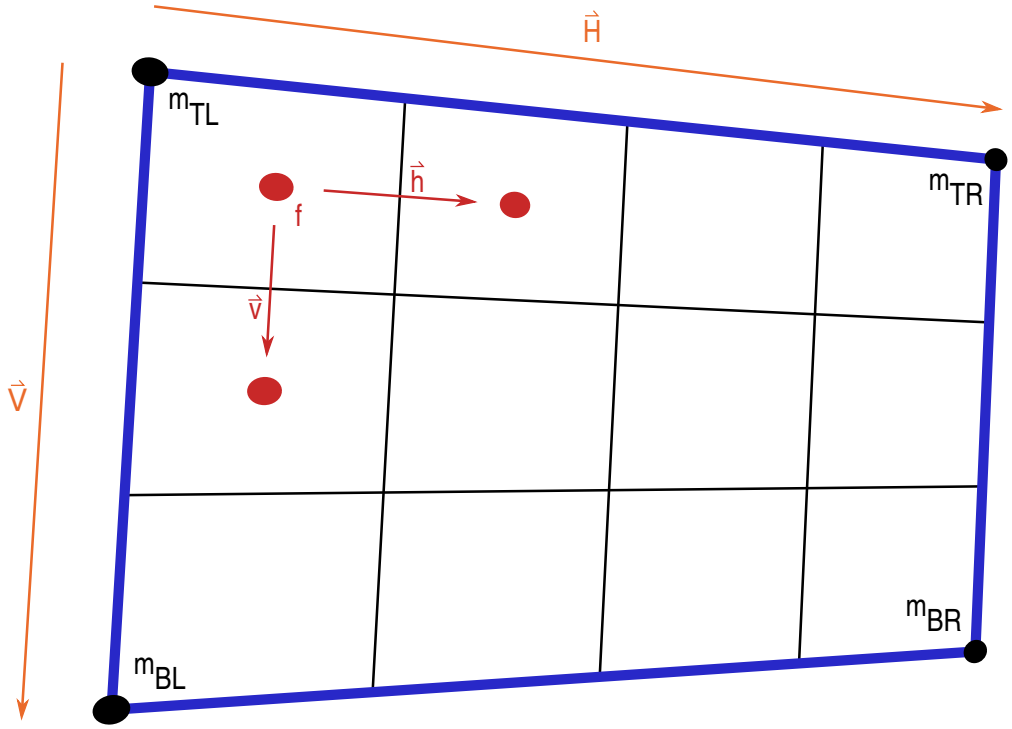

FIG. 3.9. A microdisplay plane in world space. $\vec{H}$ is the vector from m_{TL} to m_{TR}, and $\vec{V}$ is the vector from m_{TL} to m_{BL}. The point f is the exact center point of the first pixel in the microdisplay, $\vec{h}$ is an increment vector that will move one pixel over horizontally, and $\vec{v}$ is an increment vector that will move one pixel down vertically.

To calculate the unknown variables, we will start with the vectors leading along the edges of the microdisplay, from corner point to corner point:

$$\vec{H} = m_{TR} - m_{TL}$$
$$\vec{V} = m_{BL} - m_{TL}$$

Now the increment vectors $\vec{h}$ and $\vec{v}$ must be computed. Let m_h and m_v denote the horizontal and vertical resolutions of the microdisplay m. The increment vectors are as follows:

$$\vec{h} = \frac{\vec{H}}{m_h}$$
$$\vec{v} = \frac{\vec{V}}{m_v}$$

Finally, the center of the first pixel, f, must be computed. This point will serve as the starting position for the iteration through all of the other pixels:

$$f = m_{TL} + \frac{\vec{h}}{2} + \frac{\vec{v}}{2}$$

Now that all of the unknown variables have been computed, calculating the entire pixel map is simply an exercise in looping through the center point of each pixel by adding the increment vectors to f, and ray casting through those points onto the source image. Once the point on the source image is determined, it must be converted out of world coordinates to the actual pixel value based upon the source image's resolution.

3.2.4 Height Map

All of the computations that have been performed so far to create the pixel maps are implemented as preprocessing on the CPU; this map only needs to be created once, and must be completed before operations can be carried out on the GPU. As was seen in Figure 3.9, these maps are created by starting with the corner pixel and simply incrementing along through each pixel in order to create a full mapping. These mappings could have more easily been computed with a matrix multiplication-based planar transformation on the CPU, or their creation could have been avoided altogether by doing planar transformations on-the-fly on the GPU - in fact, matrix multiplications are faster than texture lookups. However, that solution would be much less versatile for the finished product because it does not allow for treating each microdisplay as anything other than

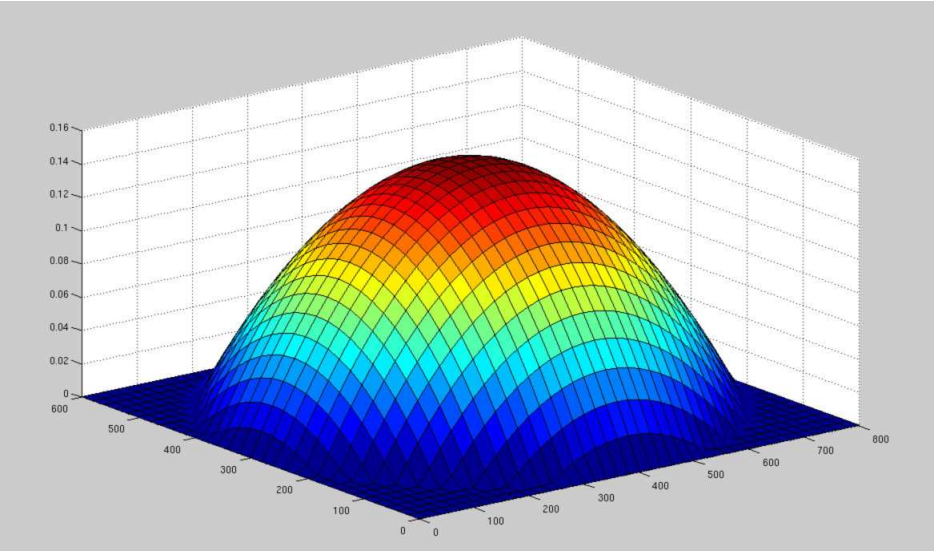

FIG. 3.10. The “bulge” height map function.

a plane. With the incrementing method, a height map can be easily applied to each pixel, based on some function of its position on the microdisplay. An example height map function called “bulge” is shown in Figure 3.10, whose equation $h(x, y)$ is as follows:

$$h(x,y) = b \cdot \left(\frac{-x^2 + xm_h}{\frac{1}{4}{m_h}^2} + \frac{-y^2 + ym_v}{\frac{1}{4}{m_v}^2} - 1 \right)$$

where (x, y) is the current pixel position, and b is the maximum height of the bulge. Note that all negative values are clamped to zero.

An example of the effect of using this height map can be seen in Figures 3.11, 3.12 and 3.13. First, we have a colored grid used as a sample source image, which will be frequently used for the duration of this paper as a source image example. Next, we have the grid source image mapped to the microdisplays, using the three-microdisplay example. Note that discontinuities of the grid lines between the three resulting images are caused by the overlap of the projections, as well as the

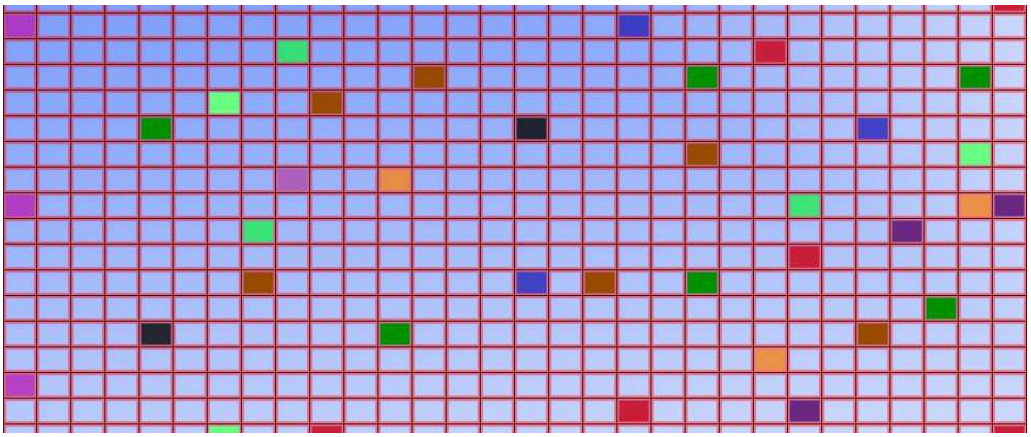

FIG. 3.11. A sample grid source image.

manufacturing anomalies of the microdisplays themselves. The projections overlap a bit because of the optics used to tile the microdisplays around each eye; when the HMD is actually used, the resulting image appears seamless. Finally, the last image is a mapping to the microdisplays using the "bulge" height map function. Notice the distortion produced by the application of the height map; more sophisticated height map functions could be applied to compensate for any potential distortion caused by the optics of the microdisplays themselves. Issues such as barrel distortion may be remedied by including a height map to the microdisplay information. Also, real-world visual issues such as an astigmatism could potentially be simulated by including the proper distortions.

3.2.5 GPU Computations

Now that the computation of the pixel maps has been completed, we can actually perform the mappings on the GPU to generate the multiple video streams that we want from our system. The generation of the pixel maps is implemented as preprocessing on the CPU; the map itself is passed to the GPU by being packed into two 32-bit RGBA textures: one for the integer values of

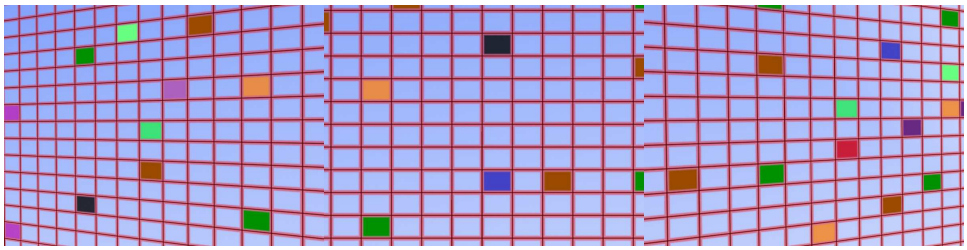

FIG. 3.12. The grid image, mapped to the microdisplays. The three resulting images from each microdisplay are shown side-by-side.

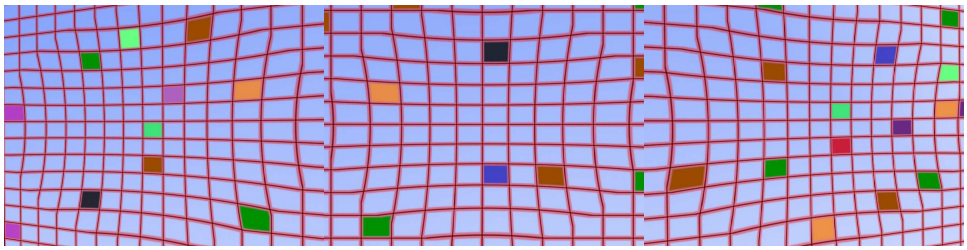

FIG. 3.13. The grid image, mapped to the microdisplays using the “bulge” height map function with a b of 0.15.

the mapping, the other for the sub-pixel floating point values. Both textures use 16 bits for their x coordinates and 16 bits for their y coordinates. Using 16 bits each was chosen both for floating point accuracy, as well as for allowing high resolution source images to be indexed properly. Each microdisplay has its own individual pair of these two textures, each with dimensions equal to the resolution of the microdisplay. Also being passed to the GPU is the existing or "original" video source, as shown back in Figure 1.2.

For each frame, as dictated by the frame rate of the original source, the GPU will receive the original frame as a texture. Then, for each microdisplay, the GPU will receive the textures containing their pixel maps. One microdisplay at a time, the GPU performs the lookups into the original source frame texture based on the locations indicated in the pixel map textures, and renders the output to a texture. These lookups have been implemented in a pixel shader, exploiting the inherent parallelism of the GPU to keep up with the frame rate of the original source, thereby acheiving real-time results. All of the output textures from the GPU can then be sent to the HMD in whatever configuration is necessary to properly match them to their corresponding microdisplays.

3.2.6 Resampling

The final piece of the implementation is the resampling being performed on the GPU. Simply looking up a pixel color based solely on the pixel maps would make the finished image look quite rough, since the resolutions of the microdisplays and the mapped area will not match up. A result resembling a poorly scaled image wouldn't help the realism factor for any VR application. Trilinear filtering is a very common filtering method, and is easy to implement – in fact, it is intrinsic to HLSL, the shading language used by XNA. However, trilinear filtering does not perform particularly well during texture mapping where the surface is at a steep angle away from the user. Since this may be somewhat akin to what will happen to the mapped pixels toward the periphery of the user's vision, a more robust filtering technique should be examined for this research. An existing system called Feline appears to be a suitable algorithm to both provide high-quality results

at steep angles as well as be simple enough to implement within a pixel shader. Note that the algorithm of Feline requires some branching, which in turn makes any attempt to implement it inside of a pixel shader require Shader Model 3.0 or higher, as previous versions do not support dynamic flow control [17].

To implement Feline, the partial derivatives indicating rates of change between texture space (original image pixels) and screen space (microdisplay pixels) must be calculated for each microdisplay pixel. This information must be passed to the GPU in order for Feline to be able to run properly in a pixel shader. As with the pixel maps, these partial derivatives can be calculated during the preprocessing phase of execution, packed into a texture, and passed to the GPU. They use the convention u, v to represent coordinates in texture space, and x, y to represent coordinates in screen space, so we will use that convention in regards to Feline as well. During the same preprocessing loop as determines the pixel map, rays will now be cast through the *corners* of the microdisplay pixels as well, not just the centers. From this, we can determine their locations on the source image, and in turn the partial derivatives $\partial u/\partial x$, $\partial u/\partial y$, $\partial v/\partial x$ and $\partial v/\partial y$.

Figure 3.14 shows a single pixel mapped from a microdisplay to the source image. The corner points are known due to ray casting during preprocessing. Also, ∂x and ∂y are both equal to 1, since the change in pixels in screen space in both directions is a single pixel. Computing the partial derivatives is now just a matter of averaging:

$$\frac{\partial u}{\partial x} = \frac{(TR_x - TL_x) + (BR_x - BL_x)}{2}$$
$$\frac{\partial u}{\partial y} = \frac{(BL_x - TL_x) + (BR_x - TR_x)}{2}$$
$$\frac{\partial v}{\partial x} = \frac{(TR_y - TL_y) + (BR_y - BL_y)}{2}$$
$$\frac{\partial v}{\partial y} = \frac{(BL_y - TL_y) + (BR_y - TR_y)}{2}$$

Once these partial derivatives have been computed during preprocessing for every pixel of every microdisplay, they are packed into a single 32-bit RGBA texture. Much like the pixel map,

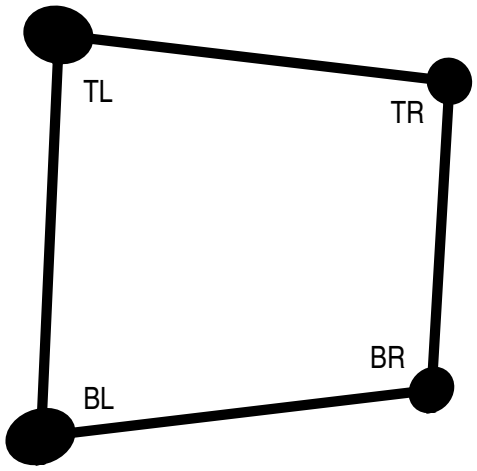

FIG. 3.14. A microdisplay pixel mapped to the source image. TL, TR, BL and BR are all two-element column vectors containing their source image coordinates for the top left, top right, bottom left and bottom right points of the mapped pixel, respectively.

there is one of these textures for each microdisplay – except that each derivative uses only eight bits. This texture is passed to the GPU along with the pixel map textures, and is then used for resampling in the pixel shader implementation of Feline.

Chapter 4

RESULTS AND ANALYSIS

A working implementation of our system was created based upon the algorithms defined in the previous chapter. We have performed a series of tests using this implementation, and we present the results as well as a complete analysis in this chapter.

4.1 Test Platform

A single platform was used throughout the development, implementation and testing of our system, and it is described below. While a specific type of tiled HMD was used throughout the work on this paper, this research is general enough to apply to any tiled HMD.

4.1.1 HMD

The tiled HMD used for the development of this system was the piSight^{TM} from the Baltimore-based company Sensics, Inc. They offer a range of models with various fields of view going up to 180 degrees, as well as having models with many different numbers of microdisplays per eye, with each microdisplay having a resolution of 800×600. All of our microdisplay description requirements mentioned in Chapter 3, such as the exact sizes of each tile, are provided in a configuration file unique to each individual HMD. Actual testing was done with a model using six microdisplays per eye. A picture of the piSight^{TM} can be seen in Figure 4.1. The array of tiles for each eye are

FIG. 4.1. Image of the piSight™ tiled HMD. Image used with permission from Sensics, Inc.

both kept behind a larger U-shaped black cover, to minimize ambient light seen by the user. The remaining pieces of the HMD are for structural support, to keep the entire unit firmly on the user's head while also providing a platform to mount tracking devices and cable guides.

4.1.2 GPU

Development and testing took place on a desktop with an ATI Radeon HD 3870 X2. While this video card supports the very latest DirectX 10.1 and Shader Model 4.0 features, they are not actually needed; a GPU that meets only the specifications for DirectX 9 and Shader Model 3.0 would have been perfectly adequate for the implementation of the algorithms in this paper. Regardless, the improved performance of the latest graphics hardware does allow the test bed to drive more tiles per HMD.

4.1.3 Software Environment

All implementation was done in Microsoft's XNA Game Studio 2.0, which is a set of tools geared towards game developers that provides easy interfacing with human input devices and

graphics hardware [18]. This platform was chosen to simplify the development of the system, as it has many built-in functions useful for the manipulation of graphics primitives. XNA makes it easy to send data to and from the GPU, as well as to create and use custom shaders. The majority of the code for this research was written in the C# XNA coding environment, with the exception of the shader code which was written in HLSL.

4.2 Test Application

In addition to implementing all of the algorithms described in the previous chapter, a small test application was also developed in order to see the mapped images while interacting with a scene in real-time. Because this framework was developed to be run on a single system, all of the source images will be generated on the same GPU as all of the mappings will be performed. This can take processing power away from the GPU, so any rendered scenes are generally kept relatively simple. This test platform can perform the mappings on its own rendered scene, a still image, or video clips. Figure 4.2 shows a simple example office scene in the test application. The user is free to fly around the environment via mouse and keyboard input. The scene mapped to three microdisplays is shown in Figure 4.3, where the original office image has the microdisplay mappings applied, and the three resulting images are displayed side-by-side. Note that this mapping occurs in the test application in real-time; the user is still free to fly around the environment.

Another feature of the application, available only when rendering scenes (i.e. not displaying still images or video clips), is to actually render the scene as many times as there are microdisplays, with each camera pointed at the proper angle for each microdisplay so as to “look through” it to see what it would be displaying. This mode can be used in comparison with the mapping technique, to show what the mapped scene “should” look like. This will be seen in the discussion of image quality in Section 4.6.

FIG. 4.2. Office scene rendered in test application.

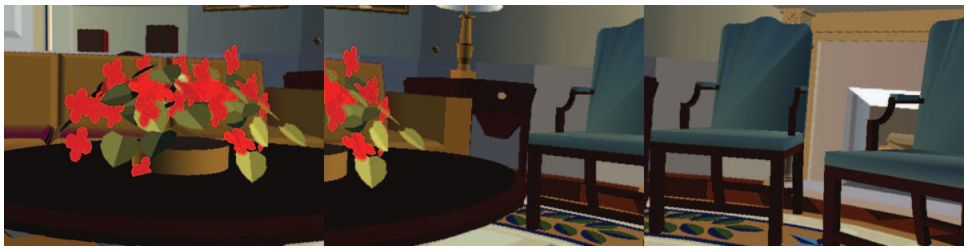

FIG. 4.3. Office scene rendered in test application, with microdisplay mapping applied.

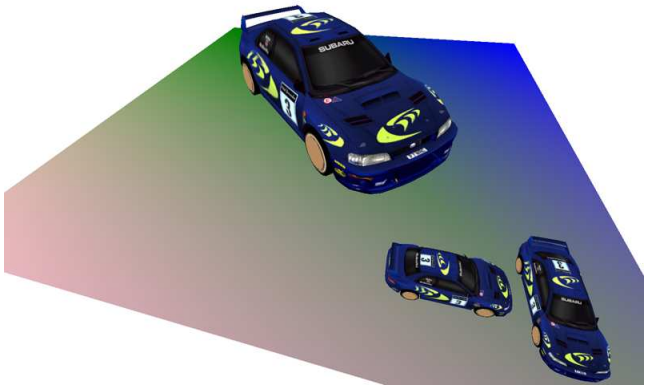

FIG. 4.4. Car scene rendered in test application.

4.3 Performance

Performance testing was done while rendering an even simpler scene, so as to have as minimal a performance hit on the GPU as possible; this scene is shown in Figure 4.4. XNA clamps performances to no higher than 60 Frames Per Second (FPS), and it is generally desirable to achieve 60 FPS for real-time applications. The more 800×600 microdisplays that the application has to drive, the slower the performance will be due to the increased number of texture lookups. The test bed, an ATI Radeon HD 3870 X2, has a Texture Fill Rate of 26,400 $\frac{MTexels}{sec}$ [19]. The theoretical maximum number of microdisplays, n, to run at 60 FPS can be found as follows:

$$60FPS = \frac{t}{m_h \times m_v \times 8a \times n}$$

where t is the texture fill rate, $m_h \times m_v$ is the total pixels per microdisplay, and a is the number of texture accesses needed by the mapping. Note that a is always multiplied by 8 since XNA inherently uses trilinear filtering. When only trilinear filtering is used, $a = 3$, since there are

texture accesses to the source image, the integer mapping texture, and the floating point mapping texture. When Feline is used, $a \geq 4$, since there is an additional required texture access to the derivatives texture, as well as how ever many additional probe accesses are needed by the Feline algorithm. That puts the theoretical maximum panels driven at 60 FPS as follows:

$$n_t \approx 38.19$$

$$n_f \leq 28.65$$

where n_t is for trilinear only, and n_f is for Feline. Actual performance tests are listed in the following table:

Number of Microdisplays	FPS w/ Trilinear	FPS w/ Feline
3	60	60
6	60	60
12	60	60
18	60	60
24	60	30
30	60	30
36	12	10
48	7	5

These results are not far off from the expected results, with the trilinear implementation not suffering a performance hit until 36 microdisplays (expected about 38.19), and Feline not suffering a performance hit until 24 microdisplays (expected somewhat less than 28.65). It does appear that the performance of trilinear "drops off" faster than the performance of Feline; trilinear goes straight from 60 to 12 FPS when the number of microdisplays is increased by only 20%, while Feline doesn't achieve similarly bad performance until the number of microdisplays is doubled. This discrepancy is difficult to explain, but could be due to a number of reasons. Trilinear filtering

is built into XNA as well as the GPU, so there may be a large amount of unseen optimization happening with trilinear that is not present with Feline. This could cause Feline to show performance decay more uniformly, and at a slower pace relative to the increase in microdisplays. Another possible cause is the way in which texture cache affects performance; since Feline requires more texture accesses, there is more opportunity for cache utilization.

4.4 Memory Usage

Another aspect of the GPU utilization is the question of how much memory the different filtering methods are consuming. Each mapping texture has a number of pixels equal to $n \times m_h \times m_v$ – essentially the panel resolutions times the amount of panels – and the individual maps are packed into these textures so as to leave them as contiguous rectangles. For example, the mapping textures for a set of three microdisplays at 800×600 each could have a resolution of 800×1800. When the number of microdisplays gets large, so too do these textures! DirectX 10 hardware allows for texture sizes up to a maximum of 8192×8192, therefore setting a very high limit of 130 microdisplays at 800×600 each – packed 10×13 into an 8000×7800 texture.

The basic trilinear filtering method described earlier requires two of these mapping textures to be loaded onto the GPU. The Feline implementation requires one extra: the derivatives texture. The memory usage for various numbers of 800×600 microdisplays utilizing either trilinear or Feline is as follows:

Number of Microdisplays	Mapping Texture Size	Trilinear	Feline
3	800×1800	10.99 MB	16.48 MB
6	1600×1800	21.97 MB	32.96 MB
12	2400×2400	43.95 MB	65.92 MB
24	2400×4800	87.89 MB	131.84 MB
48	4800×4800	175.78 MB	263.67 MB
130	8000×7800	476.07 MB	714.11 MB

While most modern GPUs have the ability to store all but the largest of these memory requirements, consideration must be taken that the effective memory of the GPU will be reduced by storing these large maps in texture memory; it will limit the memory available to the GPU to render other things, such as the source image itself if everything is being run on a single platform.

4.5 Feline Utilization

The Feline algorithm was implemented in a pixel shader to potentially perform the resampling for the microdisplays. If a particular point on a microdisplay is at a steep enough angle in reference to the viewer's eye, Feline will be used for the filtering – otherwise it reverts to a basic trilinear filtering scheme.

Figure 4.5 shows an image that was fed directly into the test application as the source image, using the three-microdisplay example used throughout Chapter 3 with a 4:3 grid source image aspect ratio. The application was modified to draw red pixels everywhere the Feline algorithm does not revert to trilinear filtering, which can be seen in Figure 4.6. This figure shows that the

vast majority of the image uses simple trilinear filtering, whereas it is only using the more complex portions of the algorithm at its very left edge. Feline's usage only towards the edges was expected, as the edges are angled away from the user more so than the center of the picture.

The three-microdisplay configuration used is a fairly conservative example, however. Some other configuration files have microdisplays very close to the edges of the user's periphery, resulting in much more stretched panel projections. An example of this can be seen in Figure 4.7, which shows a 12-microdisplay configuration where the panels towards the periphery stretch out a great deal in terms of their corresponding source image area. Figure 4.8 indicates where the Feline algorithm does not revert to trilinear on the 12-microdisplay configuration, and yet just as with the three-microdisplay configuration, Feline is used only at the extreme left edge. A close-up comparison of the bottom left corner where Feline is used is shown in Figure 4.9.

As can be seen in Figure 4.9, Feline does make the resulting image a bit clearer when it actually comes into usage. However, this improvement is only noticable when the source image has very clearly defined edges, such as the shown grid image or text. In other scenes, or especially video clips, this improvement is barely noticable; and again, it is only used in small portions at the extreme periphery of the mappings. Given the performance hit taken by using Feline, in addition to its very small usage area for even tiled HMDs with very wide fields of view, it appears that Feline is quite unnecessary for the purposes of this system. Simple trilinear filtering runs much faster, saves memory, and the microdisplay panels are never at such a steep angle so as to noticeably effect the results.

4.6 Image Quality

The test application allows for the rendered scene to either determine the microdisplay images via the mapping method described in this paper, or by simply directly rendering them. This is useful for image quality comparisons, as it allows us to see what the mapping result "should"

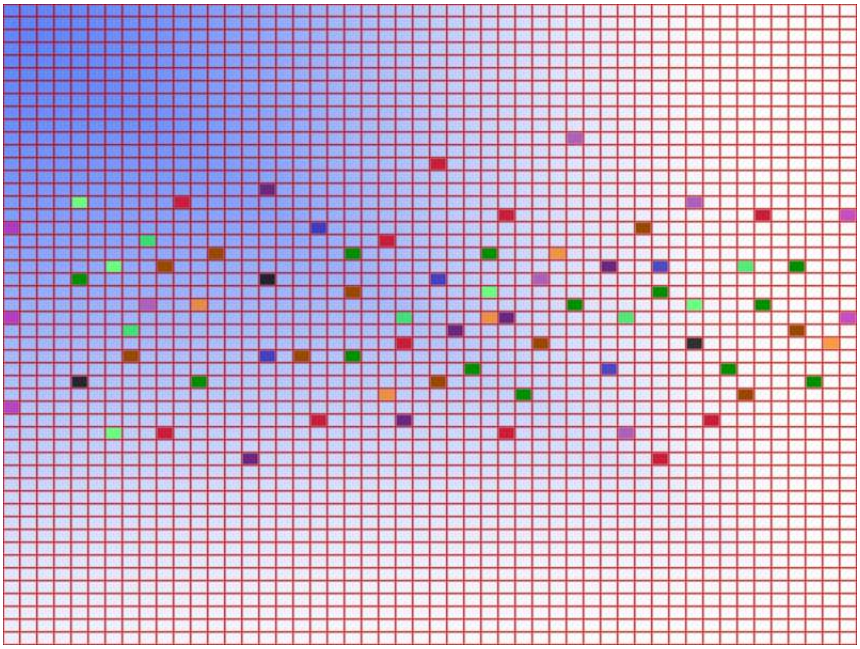

FIG. 4.5. Grid picture used as source image.

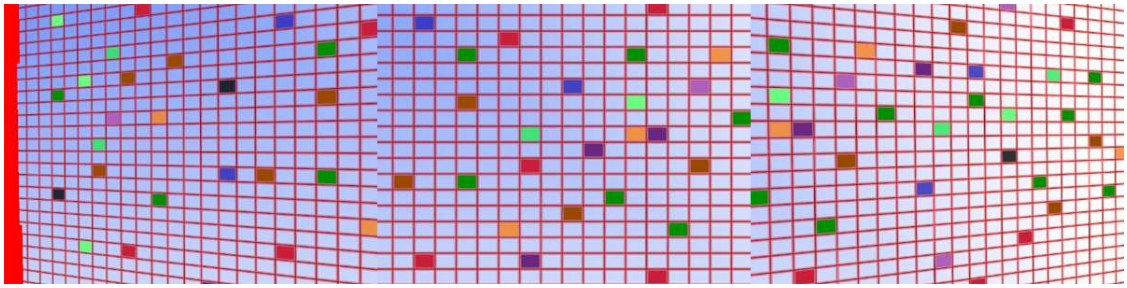

FIG. 4.6. Microdisplay results using grid picture as source image. The red region on the left is where trilinear filtering would not be used.

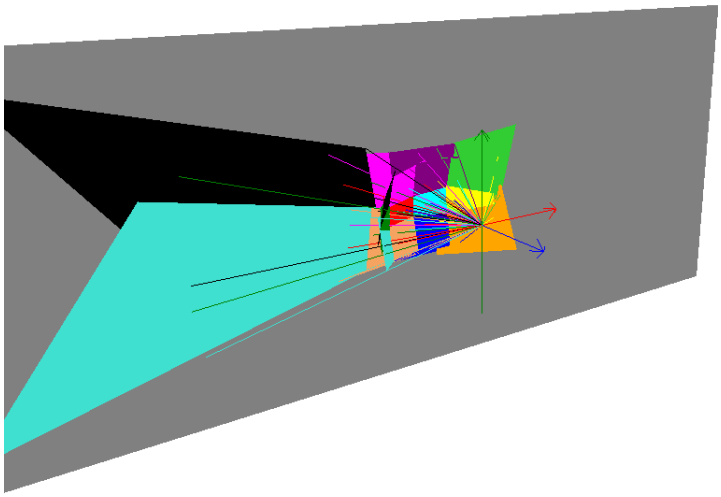

FIG. 4.7. Example 12-microdisplay configuration, with panels towards the left edge very stretched out over the user's periphery.

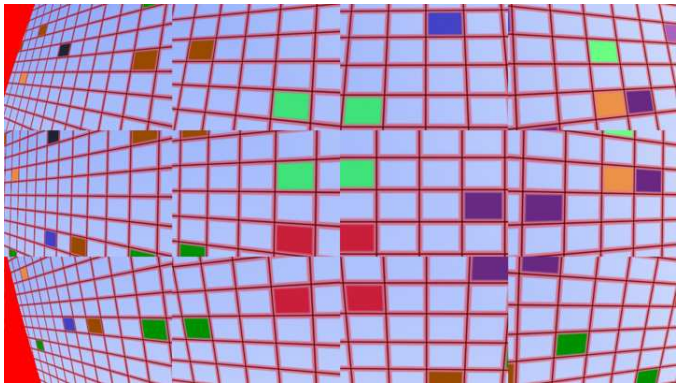

FIG. 4.8. 12-microdisplay results using grid picture as source image. The red region on the left is where trilinear filtering would not be used.

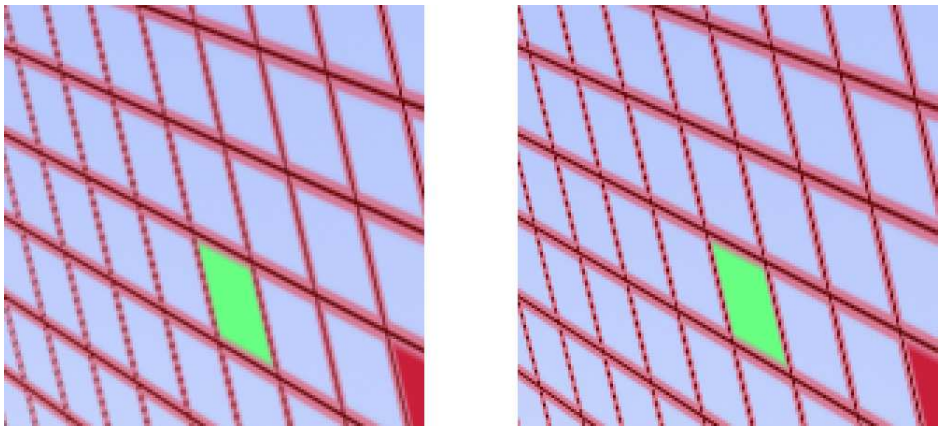

FIG. 4.9. Comparison of basic trilinear filtering to usage of the Feline algorithm. The image on the left uses Feline, whereas the image on the right uses basic trilinear filtering.

be producing. Figure 4.10 shows two side-by-side images, the image on the left being a directly rendered version of the corresponding mapped image on the right. This uses the three-microdisplay example from earlier, and the results are quite similar to the rendered version. Since the directly rendered image and the mapped image look so much alike, the mapping algorithm is working very well; it is producing results based upon a source image that look as if they have been rendered directly from scene information. This success in mapping a rendered scene shows that the system would also work well in mapping any other video source, since no scene information was used to produce the results – only the source image was used.

Figure 4.11 also shows two side-by-side images, except the mapped version here looks significantly pixelated and less visually pleasing. This uses the 12-microdisplay example from earlier, and the disparity in quality as compared to the three-microdisplay example stems from the fact that certain microdisplays only have a small amount of data available for their mapping. As can be seen in Figure 4.7, the microdisplays near the center only map to a very small area within the source image. Contrast this with the three-microdisplay example seen in Figure 3.7, where all of the microdisplays map to a relatively large section of the source image. When a microdisplay maps

to a smaller area, there are fewer source image pixels in that space; therefore the microdisplay is getting filled with less data. Since the resolution of each microdisplay panel does not change, this smaller amount of data must be scaled up a great deal, producing pixelated results. This problem could be mitigated by simply increasing the resolution of the source image, but doing so would slow down the overall performance of the system.

FIG. 4.10. Image quality comparison of the three-microdisplay mapping example. The image on the left is directly rendered, whereas the image on the right has been created via mapping an original source image.

FIG. 4.11. Image quality comparison of the 12-microdisplay mapping example. The image on the left is directly rendered, whereas the image on the right has been created via mapping an original source image.

Chapter 5

CONCLUSION

We have presented a system enabling tiled HMDs for use with existing video sources, which allows them to be used with off-the-shelf applications as opposed to requiring separate video sources for each tile. The GPU was used as a platform to handle the required processing to convert a single video source into many to drive a tiled HMD. Algorithms for execution on the GPU as well as preprocessing on the CPU were described and implemented, and their performance was analyzed. The resampling technique Feline was implemented, and shown not to be advantageous to the system described in this paper. The implementation of this system performed at frame rates high enough for interactive applications.

5.1 Contributions

The development of this system has shown that driving a tiled HMD without having individual video signals for each tile is possible, and can be done using current GPU technology at interactive frame rates. This research has identified the inputs required to describe the physical properties of a tiled HMD – such as each tile's exact height and width – allowing a system to map a single video signal to its tiles. The choice of the source image's location in world space has been recognized as a large variable in any mapping implementation, and a basic method to place the source image has been described. The research described in this paper is a foundation for the usage of tiled HMDs

with a single video source, and could be adapted into a standalone hardware platform allowing a consumer to very easily use a tiled HMD with any off-the-shelf application.

5.2 Future Work

This work has many things that could be done to expand its functionality in the future. The first – and most obvious – way that this research could be extended in the future would be to incorporate stereo vision into the system. This would greatly expand the overall field of view of the system, and could potentially make much greater use of the Feline algorithm, implemented in this research. Also, since this research is geared towards VR devices, it would be useful to perform a user study with various real-world usage situations to determine what "looks good" to an average user. For example, since the source image can be of any resolution, it would be interesting to see what the minimum tolerable source resolution could be.

Another important aspect of this work that could be altered is the way in which the orientation of the source image is decided. Instead of simply snapping around the corners of the microdisplay projections and preserving aspect ratios, perhaps the source image could be rotated and aligned so as to create a tighter fit with the microdisplays. This would avoid issues such as the low usage of the source image, seen in Figure 4.7, and in doing so would help to avoid pixelated resulting images, seen in Figure 4.11.

REFERENCES

[1] I. E. Sutherland, "A head-mounted three dimensional display," in *Proc. Fall Joint Computer Conference*, (Washington, DC, USA), pp. 757–764, Thompson Books, 1968.

[2] H. Rheingold, *Virtual Reality*, ch. 5, pp. 104–128. Simon & Schuster, 1992.

[3] R. Messing and F. H. Durgin, "Distance perception and the visual horizon in head-mounted displays," *ACM Trans. Appl. Percept.*, vol. 2, pp. 234–250, July 2005.

[4] C. Ware, K. Arthur, and K. S. Booth, "Fish tank virtual reality," in *CHI '93: Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, (New York, NY, USA), pp. 37–42, ACM, 1993.

[5] C. Cruz-Neira, D. J. Sandin, and T. A. DeFanti, "Surround-screen projection-based virtual reality: the design and implementation of the CAVE," in *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 135–142, ACM, 1993.

[6] D. J. Sandin, T. Margolis, J. Ge, J. Girado, T. Peterka, and T. A. DeFanti, "The varrier autostereoscopic virtual reality display," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 894–903, 2005.

[7] R. T. Azuma, "A survey of augmented reality," *Presence: Teleoperators and Virtual Environments*, vol. 6, pp. 355–385, 1997.

[8] J. I. Thompson, "A three dimensional helmet mounted primary flight reference for paratroopers," Master's thesis, Air Force Institute of Technology, 2005.

[9] M. Shapiro, "Comparing user experience in a panoramic HMD vs. projection wall virtual reality system," tech. rep., Sensics, Inc., 2006.

[10] Y. Boger, “The 2008 HMD survey: Are we there yet?,” tech. rep., Sensics, Inc., 2008.

[11] M. Olano and A. Lastra, “A shading language on graphics hardware: the PixelFlow shading system,” in *SIGGRAPH ’98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 159–168, ACM, 1998.

[12] K. Jo, K. Minamizawa, H. Nii, N. Kawakami, and S. Tachi, “A GPU-based real-time rendering method for immersive stereoscopic displays,” in *SIGGRAPH ’08: ACM SIGGRAPH 2008 posters*, (New York, NY, USA), pp. 1–1, ACM, 2008.

[13] D. Blythe, “The Direct3D 10 system,” *ACM Trans. Graph.*, vol. 25, no. 3, pp. 724–734, 2006.

[14] D. P. Mitchell and A. N. Netravali, “Reconstruction filters in computer-graphics,” in *SIGGRAPH ’88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 221–228, ACM, 1988.

[15] P. S. Heckbert, “Fundamentals of texture mapping and image warping,” Master’s thesis, University of California, Berkeley, 1989.

[16] J. McCormack, R. Perry, K. I. Farkas, and N. P. Jouppi, “Feline: fast elliptical lines for anisotropic texture mapping,” in *SIGGRAPH ’99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 243–250, ACM Press/Addison-Wesley Publishing Co., 1999.

[17] A. Rege, “Shader model 3.0,” tech. rep., NVIDIA Developer Technology Group, 2004.

[18] R. Miles, “Microsoft XNA game studio 2.0: Learn programming now!,” 2008.

[19] GPUReview.com, “ATi Radeon HD 3870 X2 video card – reviews, specifications, and pictures.”