CHAPTER 1

INTRODUCTION

1.1 Virtual Reality

Virtual Reality (VR) refers to the use of a computer to create a simulated environment that can be entered and interacted with as if it were real. The virtual environment is presented to the user through computer graphics and/or other devices designed to stimulate non-visual senses. Systems have been created for every one of the body's senses. Sound (without any visuals) has been used in VR to create a virtual environment in "AudioDoom," an educational game designed for blind children [1]. Using smell in VR has been explored by Yanagida et al. [2] and even the sense of taste was used in the work by Hiroo Iwata et al at the University of Tsukuba in Japan with their development of a food simulator [3].

For my work, I will be concentrating on the visual display, because sight is the dominant perceptual sense (for most individuals) and the principal means for acquiring information. Most Virtual Reality systems have also concentrated on the visual display with a variety of devices using head-mounted displays, large projection screens, and/or simple desktop graphics workstations [4].

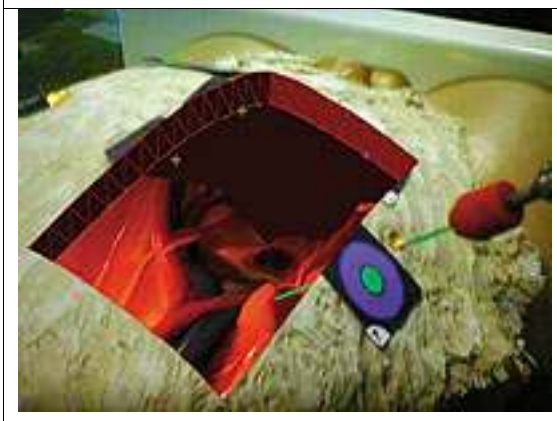
1.2 Applications

Virtual Reality is much more than an interesting academic investigation; it is a field with many practical applications, as well. VR displays (along with haptic devices) are commonly used in the entertainment industry at theme parks. Outside of entertainment, VR is finding application in the medical community. VR is being explored as a training tool that allows medical students to practice surgery using an immersive interface. Traditionally, textbook images or cadavers were used for training purposes. Textbooks, however, limit one's perspective of anatomical structures to the two-dimensional plane. Virtual Reality simulations allow students to view the anatomy from a wide range of angles and to "fly through" organs to examine bodies from the inside [5]. Detailed 3-D anatomy models would also enable the user to perform a procedure countless times, helping to eliminate the need for cadavers, which can (generally) only be used once, are limited in supply, and are more difficult to acquire than a computer system. In essence, the “practice makes perfect” motto can be put into direct action for a medical student using a VR system; instead of acquiring a new cadaver, the computer simulation of a human “avatar” can be restarted.

In addition, VR is used by architectural designers to comply with the Americans with Disabilities Act (ADA). Prairie Virtual Systems produced a software package that has been dubbed "wheelchair VR." The system enables architects, designers, and facility planners to maneuver around in a proposed

space as if they were in wheelchairs. A real wheelchair is hooked up to a computer and the user dons a head mounted display (HMD) and a haptic DataGlove that enables her to see and "feel" her way around the virtual environment [4]. By using the program, architects discovered a flaw in their design of a 140-room hospital; the bathroom countertops were two inches too high for wheelchair-bound patients to use the sinks [6].

Figure 1.2: The user's three-dimensional view (left) of internal organs as seen through a "synthetic pit." The system allows the user to see parts of the real world (right) while superimposing 3-D computer graphics, a technique known as "augmented-reality." *Photos courtesy of University of North Carolina Department of Computer Science* [5].



Most of the applications for virtual reality, such as the wheelchair VR, require a presentation of accurate depth perception. If the depth perception in a VR system is not correct, then the user is trained to systematically underestimate or overestimate distances. This could be a serious problem, especially in

medical training or architectural design, where minute distances make a significant difference in delicate operations.

Unfortunately, for reasons as yet unknown, virtual environments are being perceived as significantly smaller than is intended, a phenomenon known as compression [7]. It is this problem that motivates my research.

1.3 The Problem

Work by University of Utah and other researchers indicate a systematic underestimation of distances in a virtual environment [7-11]. A person walking on a treadmill virtual environment reports that she "should already be" at her destination before "reaching" it [7]. A brief review of the results found in past studies comparing the accuracy of distance perception in the real world versus a virtual environment is shown in Figure 1.3:

Figure 1.3 These studies compared real-world distance judgments ("Real") and judgments made with computer generated images ("VR"). The "Distances" column indicates the range of distances studied. The percentage is the ratio of perceived distance to actual distance.

Study	*Distances*	*Real*	*VR*	*Task*
Witmer & Sadowski (1998)	4.6m – 32m	92%	85%	treadmill walking
Knapp (1999)	5m – 15m	100%	42%	triangulated walking
Durgin, Fox, Lewis & Walley (2002)	2m – 8m		65%	direct walking

Willemsen & Gooch (2002)	2m – 5m	100%	81%	direct walking
Mohler, Thompson, et al (2004)	5m – 15m	95%	44%	triangulated walking

Creating an accurate three-dimensional perception using a two-dimensional computer screen presents a multitude of challenges. There are many depth cues that need to be properly portrayed for an individual to get a sense of depth from a 2D image. Examples of these are the effect of one object in 3-D space blocking another object from view (known as occlusion), edge interpretation, familiar size, shading and shadows, the gradual change in size, change in the texture pattern as objects recede into the distance (known as the texture gradient), and position relative to the horizon. These depth cues are easily replicated in a computer generated image, but even included in experimental systems, the compression effect exists. There is something wrong (or missing) in the graphic display.

1.4 Processing Depth

There are two different types of depth cues: absolute and relative. Absolute depth cues enable the human visual system to determine the actual distance to objects, whereas relative depth cues merely give information as to how far objects are in relation to each other. Stereopsis – the slightly different positions each eye occupies on the head to form slightly different images – is the

only absolute depth cue thus far implemented in virtual reality systems. Most virtual systems only present relative depth cue information (such as the cues listed in the previous paragraph) as opposed to absolute depth cues, which may be a factor of this compression effect.

The importance of absolute depth cues versus relative depth cues for human perception and interaction in an environment has been explored by neuroscience researchers Goodale and Milner. Goodale and Milner have proposed that the brain processes images in two different cortical pathways: the dorsal and ventral stream. The ventral stream is where the conscious mind uses vision and relative depth cues such as size, position, and occlusion to understand the relations of objects in a scene to each other and identify objects. Patients with brain damage in the ventral stream are unable to recognize objects or distinguish shapes yet are still able to accurately grasp and catch objects, and move about the world without assistance. In the case when brain damage affects the dorsal stream of the brain (where absolute depth cues are processed), the person can consciously see and conceptualize the world, and can describe an object as further away or closer to them, but is unable to accurately grasp or manipulate objects [12]. Without absolute depth cues, interaction with objects in a real or virtual environment will be flawed.

1.5 Hypothesis

Seeing a completely in-focus and detailed image on a computer screen (regardless of the realism of the display) sends cues that the virtual environment is flat. In the real world, objects behind and in front of an object of attention are seen as blurred or out-of-focus, with the area in focus known as the depth of field. The blurring occurs because the eye lens accommodates for any given point in focus. Accommodation, which is the tension of ciliary muscles in the eye and the resulting curvature of the eye's lens as it focuses on particular objects, is considered to be an absolute depth cue.

Although accommodation cannot be safely or practically manipulated in the lab, an image created on a person's retina due to correct accommodation can be replicated. This visual image with blur from a finite depth of field has not been established as a depth cue, but there is evidence to suggest it aids in depth perception. I hypothesize that the visual image does play a role and that by adding the correct depth of field blur – producing the image that would have appeared on the retina had accommodation been operating – we will improve the accuracy of depth perception on 3-D computer displays and help eliminate the compression factor currently affecting VR systems. To test this, this project aims to build a real-time graphical display system that will show the correct depth of field according to a user's gaze position and compare that with traditional displays that have an infinite depth of field (a completely in-focus image).

CHAPTER 2

BACKGROUND

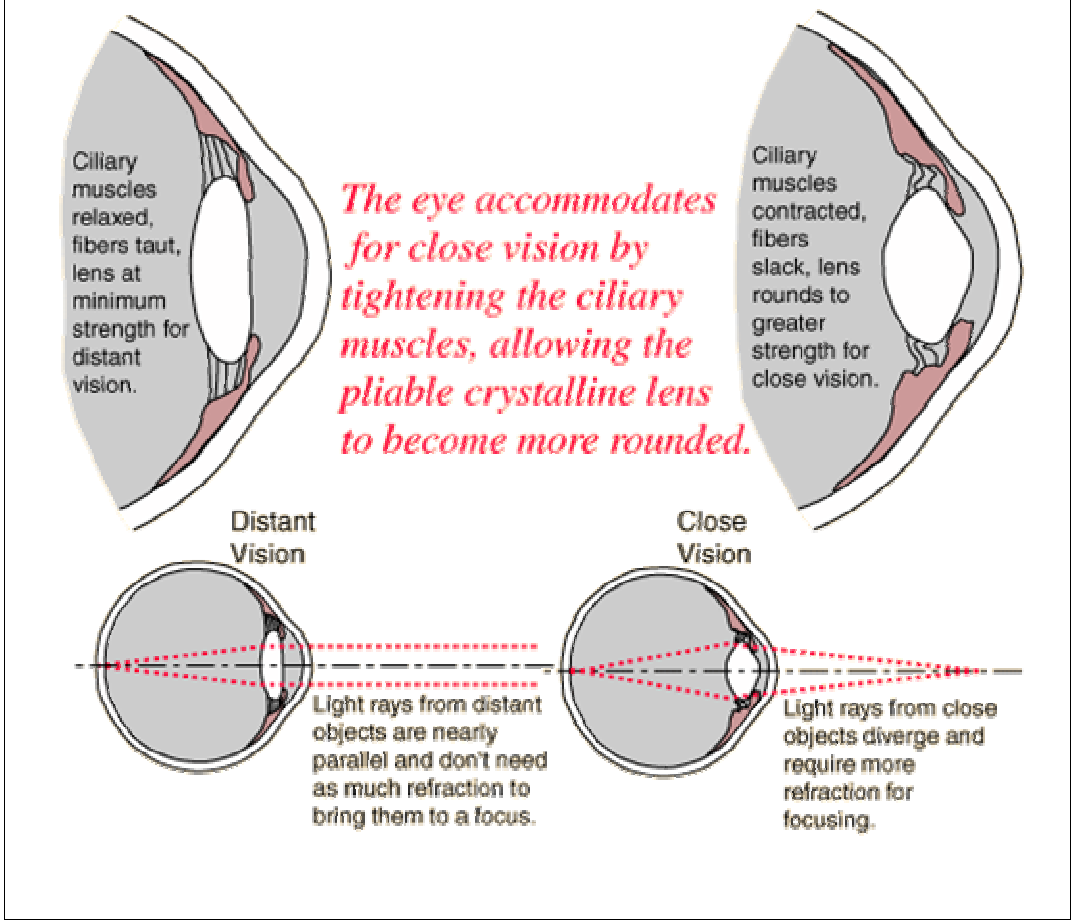
My work concentrates on modifying the visual image in a VR system for the user. Before I can create a non-static image on a computer screen of what humans normally see in the real world, it is important to understand certain aspects of the visual system. This chapter will discuss optics and the properties of light passing through the human eye, explain the properties of the circle of confusion, how the phenomena of depth of field operates in relation to the circle of confusion, summarize evidence that depth of field blur is a depth cue, and review past work in computer graphics to replicate a finite depth of field and show how it influenced my approach.

2.1 Accommodation & The Human Eye

Light rays enter our eye through the cornea (the transparent, circular part of the front of the human eyeball) and then to the lens. The lens of the human eye actively changes its shape to focus light from objects of interest onto the retina (a "screen" that then sends information about the image to the brain via the optic nerve). This adjustment is called accommodation and is achieved by the contraction and relaxation of the ciliary muscle. [13]. It is this process that

allows humans to have a variable focusing capability. When the eye lens thins, light from objects far away are brought into focus on the retina. When the eye lens thickens, objects nearby become in focus.

Figure 2-1: **Human eye lens and ciliary muscles during accommodation**.[14]

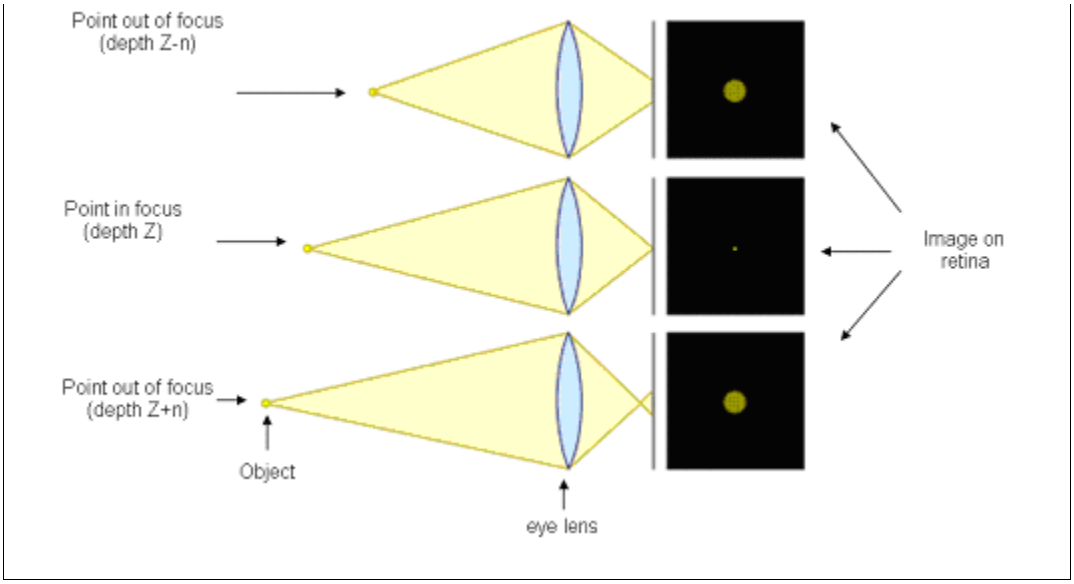


2.2 Circle of Confusion and Depth of Field

The human visual system needs to changes its lens shape and thereby accommodate (and focus) to different depths because we exist in a three-dimensional world where not all objects lie at the exact distance away from our

eye lens (and thereby produce light rays that converge perfectly to a point on the retina). When the eye is accommodated to a particular depth, anything in front or behind that depth will appear blurred to varying degrees. This visual blur is the result of light rays from the non-accommodated objects being spread in a circular distribution on the retina. Each circular distribution of light is known as the circle of confusion. This natural phenomenon is shown in Figure 2-2 below.

Figure 2-2: E**xamples of light rays producing a circle of confusion**. In all cases below the aperture and focal length are accommodated to view a point at depth Z in perfect focus. The middle diagram shows the image of a point at the accommodated distance. Its retinal image is sharp and not blurred. The top diagram shows a point on a plane closer than the one in focus, and the subsequent blur on the retinal image as the light from that point is spread in a circle. The bottom image shows a point too faraway from the accommodated distance, producing a blur on the retinal image as the light from that point is also spread in a circle.



When the circle of confusion for point sources of light becomes large enough to notice, we say the image is blurred. The range in which we are unable to notice the existence of the circle of confusion (the distance in front of and behind the subject or object in visual attention that appears to be in sharp focus) is known as depth of field. This is determined by the focal length (the distance behind the lens through which all light rays pass [13]) and the aperture of a lens. A large aperture or a long focal length produces a shallow depth of field (small range in focus) and a small aperture or a short focal length produces a wide depth of field (more range in focus).

The phenomenon that produces a finite depth of field (large, noticeable circles of confusion) occurs in any lens, whether it is in an eye or a camera. Examples of images with a noticeable, finite depth of field are in Figures 2-3 and 2-4 below.

Figure 2-3: Depth of field from a camera lens. The center text is in focus while text further away and closer to the camera lens than the center text appears blurred. Image source: http://en.wikipedia.org/wiki/Depth_of_field

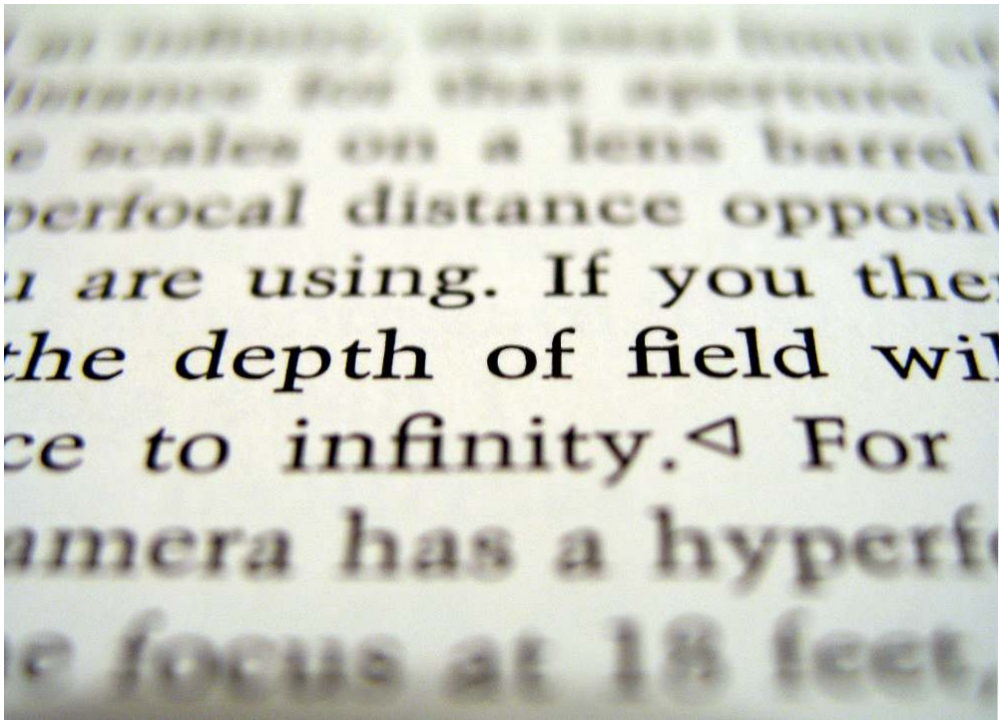
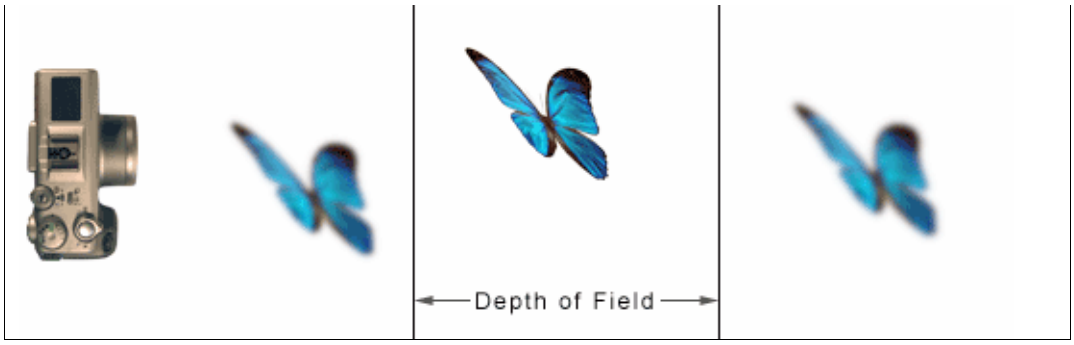


Figure 2-4: Example of depth of field. Image source: http://en.wikipedia.org/wiki/Depth_of_field



2.3 Depth of Field as Depth Cue

The proprioceptive sensors from the tension in the eye lens' muscles provide the human visual system information about the distance to the focused object. This information from accommodation is only practical for judging sizes and distances up to 2.4 meters away [15]. However, this is the range in which most interaction between humans and objects occur and should not be dismissed.

We are currently unable to control the ciliary muscles to produce this proprioceptive depth cue experimentally. However, due to the optics of the eye there is a significant side effect on the retinal image. A small depth of field is produced such that objects in front and behind the point of attention on the retinal image appear notably blurred. Unlike accommodation, this visual effect can be replicated in a virtual reality system. In addition, there is evidence that the visual blur from a finite depth of field, and not just the ciliary muscle tension, provides this depth information.

Bailey et al. [16] tried to quantify the importance of different depth cues in the real world. When studying accommodation, they built a monocular

apparatus to view three-dimensional objects. Subjects were made to look through a small hole in the top of a box where cylindrical objects of different heights were arranged. The lighting was set so that no shadows were cast and the size of the hole through which subjects could view the stimuli was varied. When the size of the hole became smaller than the pupil (reducing the aperture of the visual system), the depth of field became virtually infinite. Their experiments (with a $p<.0002$) showed that "the effect of accommodation disappears when the depth of field is increased to the point of removing the blur." Without a depth of field blur, no depth information was obtained by the viewer! This suggests that the visual image caused by accommodation acts as an important depth cue [16].

Other work has shown that the blur from a finite depth of field can be used to provide absolute distance information. Schneider et al. used the amount of blur to calculate absolute depth in computer vision [17]. By only comparing the blur in a couple of unfocused images of the same scene, taken with different apertures, they could determine the actual distances of the objects in the scene.

2.4 Circle of Confusion & Optics

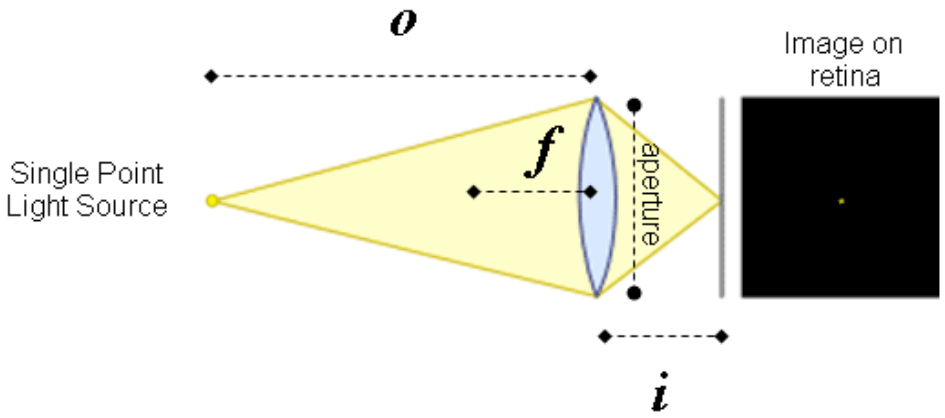
Schneider et al. [17] were able to calculate absolute distances from depth of field blur because of the basic physics of a lens and the subsequent sizes of the circles of confusion it produces. The following known relationship occurs

between a lens' focal length (*f*), the distance from the lens to the depth plane in focus (*o*), and the distance from the lens to the retina, (*i*):

$$\frac{1}{i}+\frac{1}{o}=\frac{1}{f}$$

From this equation the amount of focal blur and the exact distance of an object from the lens can be derived. A visual representation of *i, o,* and *f* is shown in Figure 2-5 below.

Figure 2-5: **Light from a point focused onto the retina.** Whenever a point is seen in perfect focus, light from that point had passed through the lens and was refocused onto the retina.



Key:
f = focal length
o = distance from lens to the depth plane in focus
i = distance from the lens to the retina

The size of the circle of confusion can be calculated due to the relationship between *f*, *o*, and *i* and the relationship between similar triangles. Figure 2-6 gives a visual example of the optics involved when a point is in focus and when a point is out of focus. Figure 2-7 explains the mathematical relationship between the circle of confusion's radius, *C*, and the *o, f,* and *i* and aperture.

Figure 2-6: A circle of confusion formed from a point too close to the lens' accommodated focus. *O** = distance from the lens to the accommodated object; *i** = distance from the lens to the retina; *O* = distance from the lens point away from the accommodated distance; *i* = distance from lens to the image of object at *O*. Light from a point at distance *O** passes through the lens and converges to a sharp point on the retina, at distance *i** from the lens. Light from a point closer to the lens than distance *O* would ideally converge to a sharp point at distance *i* from the lens. Since it does not converge to a point on the retina, the light that does hit the retina forms a larger, circular image of light known as the Circle of Confusion (shown in red).

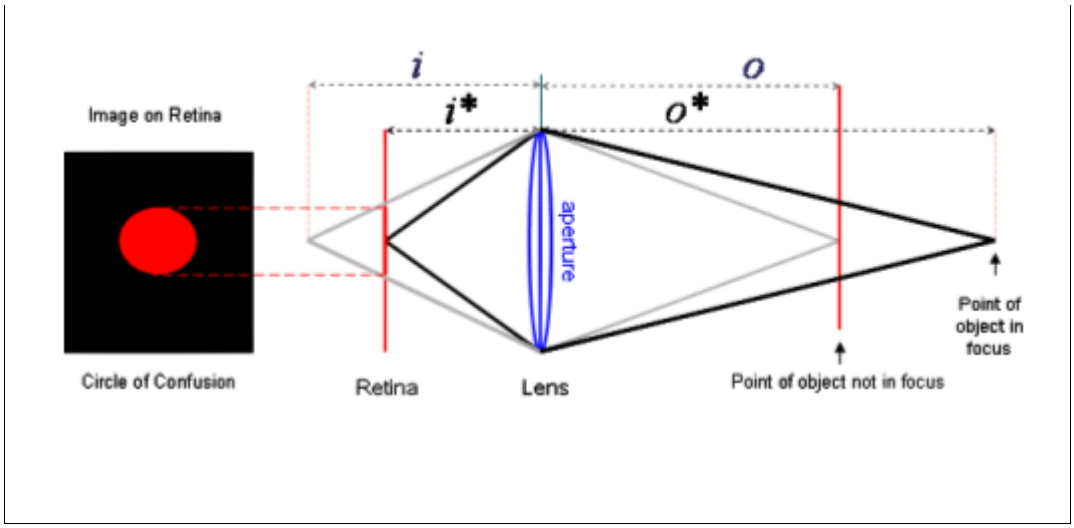
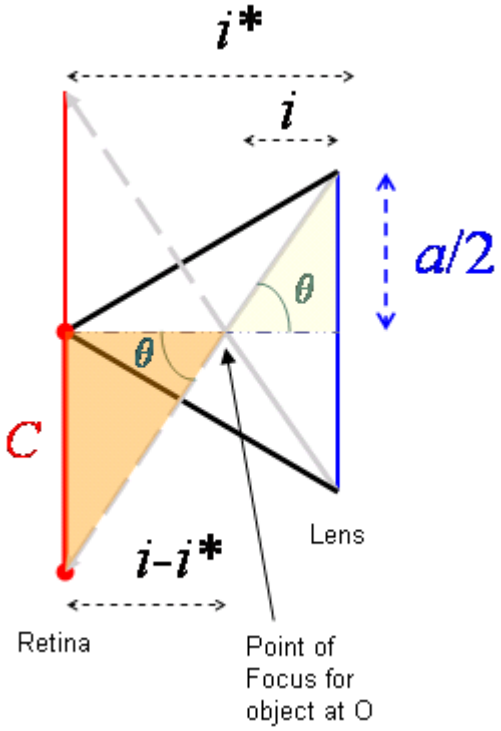


Figure 2-7: M**athematical relationship between circle of confusion, *a, i*,* and *i***. Key: *a* = lens aperture; *C* = circle of confusion radius; *i** = distance from the lens to the retina; i = distance from lens to the ideal image plane; θ = common angle between the similar triangles. Further description below figure.

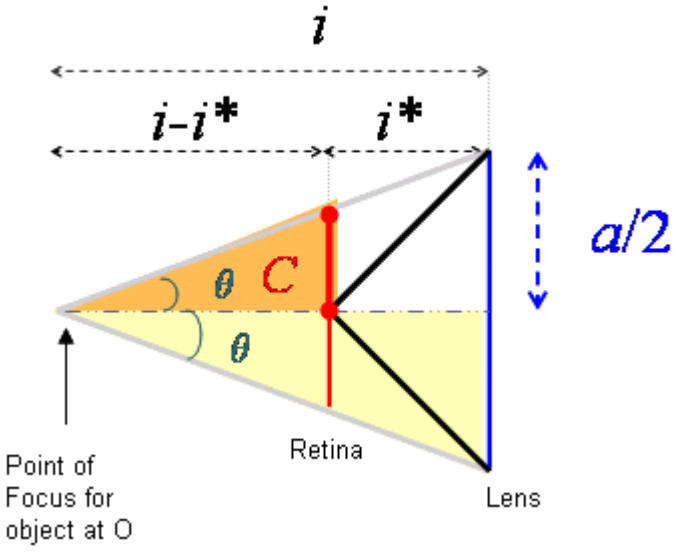


If $i - i^* < 0$: (left image)

$$\tan\theta = \frac{a/2}{i} = \frac{C}{i^* - i}$$

$$C = \frac{a/2}{i} \cdot (i^* - i)$$

$$C = \frac{a}{2} \cdot \left(\frac{i^* - i}{i}\right) = \frac{a}{2} \cdot \left(\frac{i^*}{i} - 1\right)$$



If $i - i^* > 0, (\text{above})$:

$$\tan\theta = \frac{a/2}{i} = \frac{C}{i - i^*}$$

$$C = \frac{a/2}{i} \cdot (i - i^*)$$

$$C = \frac{a}{2} \cdot \left(\frac{i - i^*}{i}\right) = \frac{a}{2} \cdot \left(1 - \frac{i^*}{i}\right)$$

In both images above, the black lines represent light rays from a point at the accommodated distance that converge perfectly on the retina (represented with a red line), the radius of the circle of confusion (*C*) is drawn with a solid red line between the two red dots, and half of the lens' aperture (*a*/2) is labeled with the dotted blue line with the double arrows.

In the topmost image, light rays from a point farther from the lens than the accommodated distance converge before they can reach the retina. They are represented by the gray line. In the lower image, light rays from a point closer to the lens than the accommodated distance would ideally converge after reaching the retina. They are also represented by a gray line.

In both images, the yellow triangle (with length i and height a/2) is similar to the darker orange triangle (with length i-i* and height C). The common angle shared by these triangles is θ. Since the yellow and orange triangles are geometrically similar, the following relationship can be mathematically derived.

2.5 Calculating the Circle of Confusion

Using the relationship derived in Figure 2-7, we can find the circle of confusion's size on the retina generated by any point away from the accommodated focus depth by only knowing the aperture of the lens (a), the distance between the retina and the lens (i*), and distance from lens to the ideal image plane (*i*).

For example purposes, let us choose a focal point that the eye has accommodated to (*o**) at 90.0 cm (about 3 ft) from the eye lens. Given that the distance from the retina to the eyes' lens (i*) is 2.4 cm and that the average human eye lens aperture (a) is .40 cm [18] our next step is to find the current focal length of the human eye lens:

Since $\frac{1}{f} = \frac{1}{i} + \frac{1}{o} = \frac{1}{i^*} + \frac{1}{o^*}$, then $f = \frac{i^* o^*}{i^* + o^*} = \frac{io}{i + o}$

Yielding:

$$f = \frac{i^* o^*}{i^* + o^*} = \frac{(2.4\text{cm})(90.0\text{cm})}{2.4\text{cm} + 90.0\text{cm}} = 2.34\text{cm}$$

So the focal length of a human eye lens when accommodated to a distance 90.0 cm away is 2.34cm.

With this information and the equation generated in Figure 2-7, the radius of the circle of confusion for a point farther than the accommodated distance is:

$$C = \frac{a}{2} \cdot \left(\frac{i^*}{i} - 1 \right) = \frac{.40}{2} \cdot \left(\frac{2.4}{i} - 1 \right)$$

$$\text{since } \frac{1}{i} + \frac{1}{o} = \frac{1}{f}, \quad i = \frac{f \cdot o}{o - f}$$

$$C = \frac{.40}{2} \cdot \left(\frac{2.4}{\frac{f \cdot o}{o - f}} - 1 \right) = \frac{.40}{2} \cdot \left(\frac{2.4}{\frac{2.34 \cdot o}{o - 2.34}} - 1 \right)$$

And the radius of the circle of confusion for a point closer than the accommodated distance is:

$$C = \frac{a}{2} \cdot \left(1 - \frac{i^*}{i} \right) = \frac{.40}{2} \cdot \left(1 - \frac{2.4}{i} \right)$$

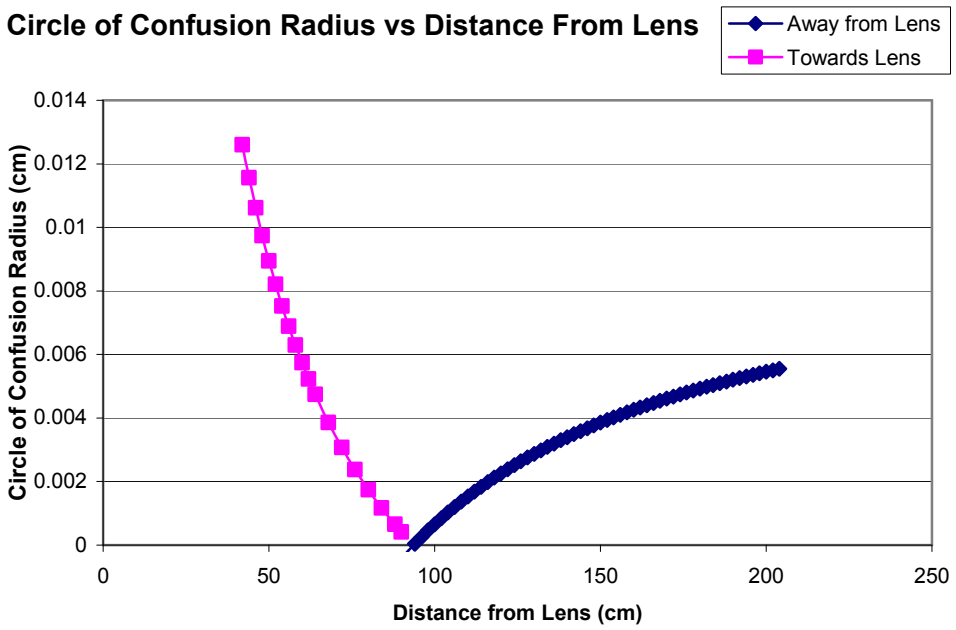
$$\text{since } \frac{1}{i} + \frac{1}{o} = \frac{1}{f}, \quad i = \frac{f \cdot o}{o - f}$$

$$C = \frac{.40}{2} \cdot \left(1 - \frac{2.4}{\frac{f \cdot o}{o - f}} \right) = \frac{.40}{2} \cdot \left(1 - \frac{2.4}{\frac{2.34 \cdot o}{o - 2.34}} \right)$$

Now, we can determine the size of the circle of confusion for any point away from the accommodated distance of 90.0cm. The relationship between circle of confusion size and *o* (distance from the lens) is graphed below in Figure 2-8.

Figure 2-8: The size of the circle of confusion in relation to the distance from the eye lens. The position of the lens is at 0 cm on the x-axis in which the lens

is accommodated at 90.0cm and the focal length is 2.34cm.



Although the sizes of the circle of confusion graphed above seem too small to produce a visual difference, when it is compared to the size of the retina (32 mm from ora to ora along the horizontal meridian), and when considering the number of light points that would be spread in total, it is significant [19].

This mathematical relationship was also briefly experimentally confirmed with use of a point source light, a lens, and a white sheet of paper to act as a viewing screen. See Figure 2-9. Light hit the lens and was redirected onto the screen. The location of the light source that resulted in a perfectly in focus dot on the screen was found and recorded, and then the light source was moved several times (away and later towards the lens). A circle of confusion (a blurred, enlarged circle of light) was formed on the screen. The different sizes

of the circle of confusion were measured and recorded along with the distance of the light source from the lens. The same relationship between circle of confusion size and the distance the light source (or object) is from a lens was found. The results are graphed in Figure 2-10.

Figure 2-9: Experimental Test Apparatus. Light hit the lens and was redirected onto the screen. If the light source was moved farther away or closer to the lens than at the accommodated (in-focus) distance, then the light was spread in a circle (the circle of confusion). The diagram is not to scale.

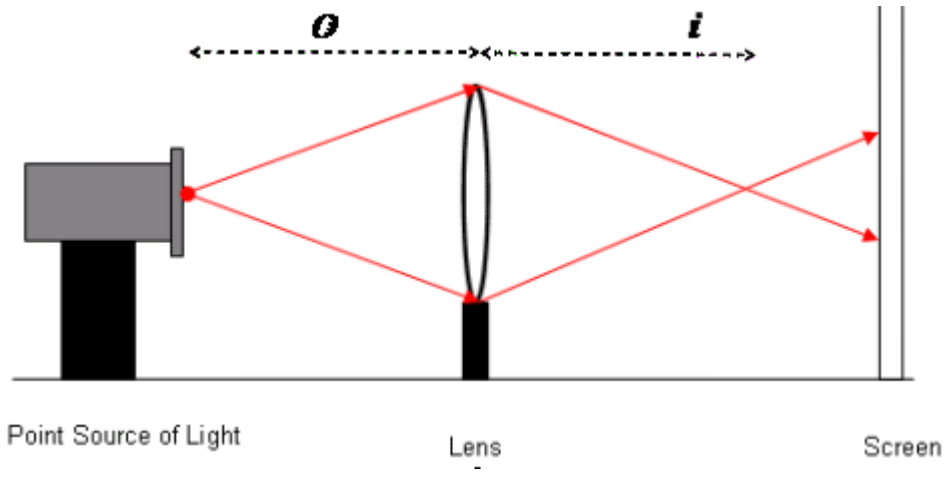
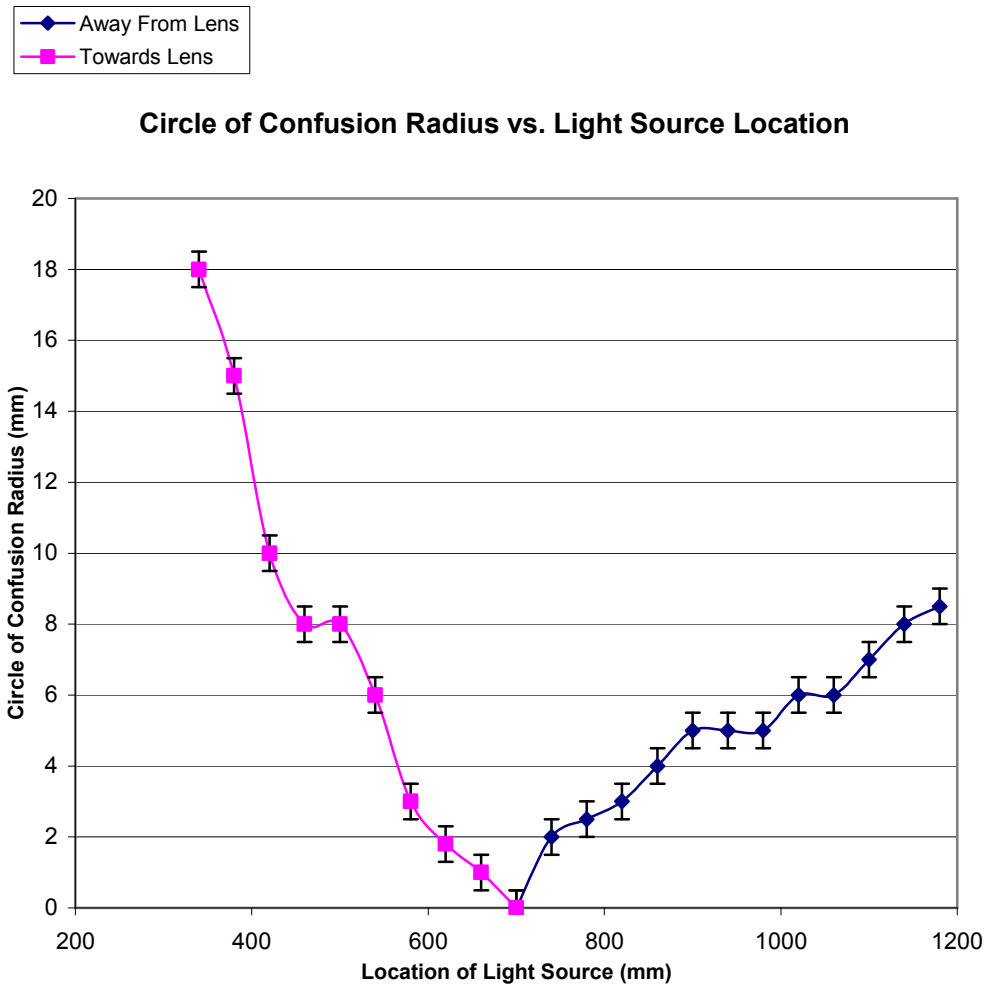


Figure 2-10: The size of the circle of confusion in relation to the distance the light source was from the lens. The position of the lens is at 0 mm on the x-axis in which the lens is in focus for light sources (and objects) at 700 mm. The

slight fluctuation from the mathematical ideal is largely due to human error in accurately measuring the size of the circle of confusion on the screen, which only differed by a few millimeters. However, the same trends from Figure 2-8 can clearly be seen.



An interesting trend to note is that the amount of focal blur (circle of confusion size) increases rapidly as an object approaches the eye lens but increases slowly as the non-accommodated object (or light source) moves away

from the lens. This phenomenon shows the importance of calculating the exact size of the circle of confusion in order to reproduce an accurate amount of blur. This fact plays an important role in deciding which algorithm and method to use in creating the computer graphic display with a finite depth of field.

2.6 Rendering & Computer Graphics

One of the goals of this project was to use a rendering method that could produce an accurate depth blur in real-time, so that the appropriate image that would have appeared on the human retina would be displayed on the computer screen according to the user's gaze. Before we can discuss the appropriate method to generate an accurate, real-time depth of field blur in a computer generated image, it will be helpful to briefly explain computer graphics.

Computer graphics utilize computers to generate visual images, often integrating or altering visual and spatial information sampled from the real world. Generating a computer graphic can be compared to taking a photograph. Before taking a picture, there must be appropriate lighting (else nothing would be visible) and of course, actual objects to take a photograph of in the real world. With computer graphics, a virtual world is defined where objects are mathematically described and modeled. Most computer graphic applications define their virtual objects in terms of polygons, as any object (fictional or otherwise) can be roughly defined by breaking it up into a series of geometric shapes. For example: to create the image of a cube, the dimensions of the

cube's sides and the placement of its vertices in 3D space would have to be known, thereby defining the 6 polygons (sides) that make up the cube. After creating the mathematical model, the placement of light sources must be set. Additional steps can also take place to describe the color and texture (often referred to as the "materials") of the defined polygons.

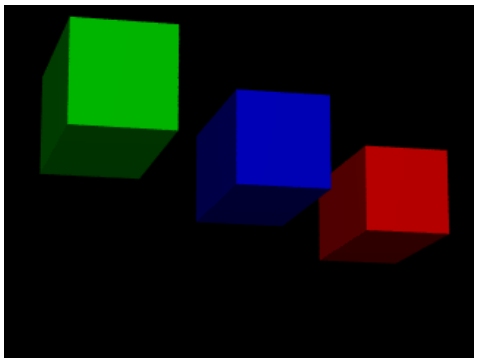
Going back to our photographic analogy, an actual camera would be positioned to capture the desired view of the real world scene. In computer graphics, the viewpoint into the virtual world is also referred to as the "camera" position. Then the process of generating the final image, known as rendering, can take place.

A rendered image consists of pixels (the smallest visible element the display hardware can display) on the computer monitor. It is during the rendering stage that the machine computes, pixel by pixel, a 2D bitmap image from 3D scene model data computed from the viewpoint of the simulated camera. There are many rendering techniques such as ray-tracing, point-based surface splatting, and scan-line. Achieving a more realistic image requires more time, memory and processing power. An extremely complicated scene can take up to several days to render. Less complicated models can be rendered in a few seconds.

Ray-tracing is one of the most realistic methods for rendering images. As its name implies, ray-tracing works by tracing the path taken by a ray of light from the virtual scene, and calculating the reflection, refraction, or absorption of

the ray whenever it intersects an object in the virtual world. Ray-tracing also takes into account the location, strength, and quality of all light sources within the virtual environment; shiny objects reflect other objects and realistic shadows are cast. Not surprisingly, ray-tracing is one of the most time consuming rendering techniques. See Figure 2-11 for an example of a ray-traced computer image.

Figure 2-11: Ray-Traced Computer Generated Images. Both scenes (originally 320 x 240 pixels) were rendered using the open source Persistence of Vision Raytracer (POV-Ray). On a machine with 2.00 GB of RAM and a 2.80GHz, 2.79GHz dual processor it took 4 minutes 47 seconds to render the more complicated "Balcony" scene on the left and only .28 seconds to render the simple, cubed image on the right. The "Balcony" image was programmed by Christoph Hormann, 2001 [20].



Scan-line rendering is faster than ray-tracing, but does not produce as realistic results. The image is rendered one scan line (a single row of pixels) at a time rather than object-by-object as in ray-tracing. This method is normally used in OpenGL, an application programmer's interface (API) that allows programmers to access graphics hardware. OpenGL images can be rendered as wireframe (see Figure 2-12), flat-shaded (see Figure 2-13), or with Gouraud shading -- a method for linearly interpolating a color or shade across a polygon (see Figure 2-14). Gouraud shading is used to achieve smooth lighting on low-polygon surfaces without the heavy computational requirements of calculating lighting for each pixel. By default, OpenGL does not render shadows, also reducing the rendering time significantly [21].

Figure 2-12: Wireframe Image in OpenGL. The Utah Teapot rendered in wireframe mode. The polygons that make up the model are easily visible.

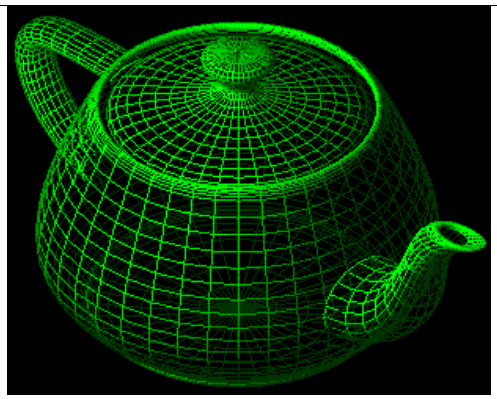


Figure 2-13: Flat Shading in OpenGL. The Utah Teapot rendered with flat shading. The image appears two-dimensional.

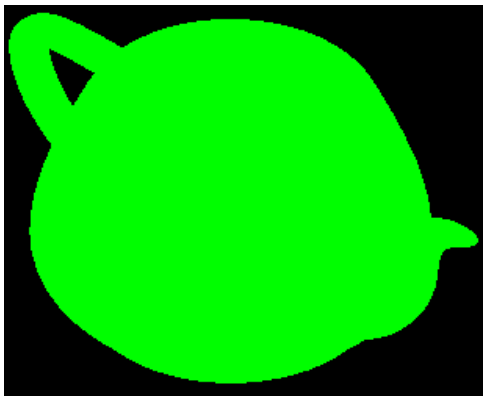
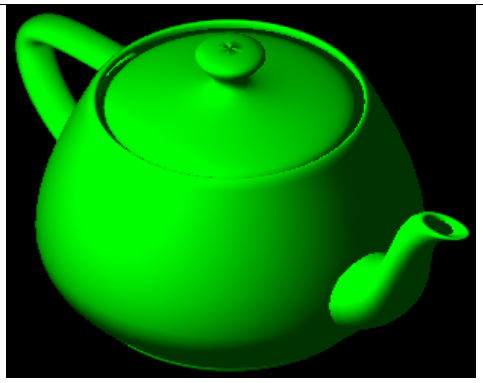


Figure 2-14: Gouraud Shading in OpenGL. The Utah Teapot rendered with Gourand shading. The method was developed by Gouraud in 1971 and it simulates the differing effects of light and color across the surface of an object.

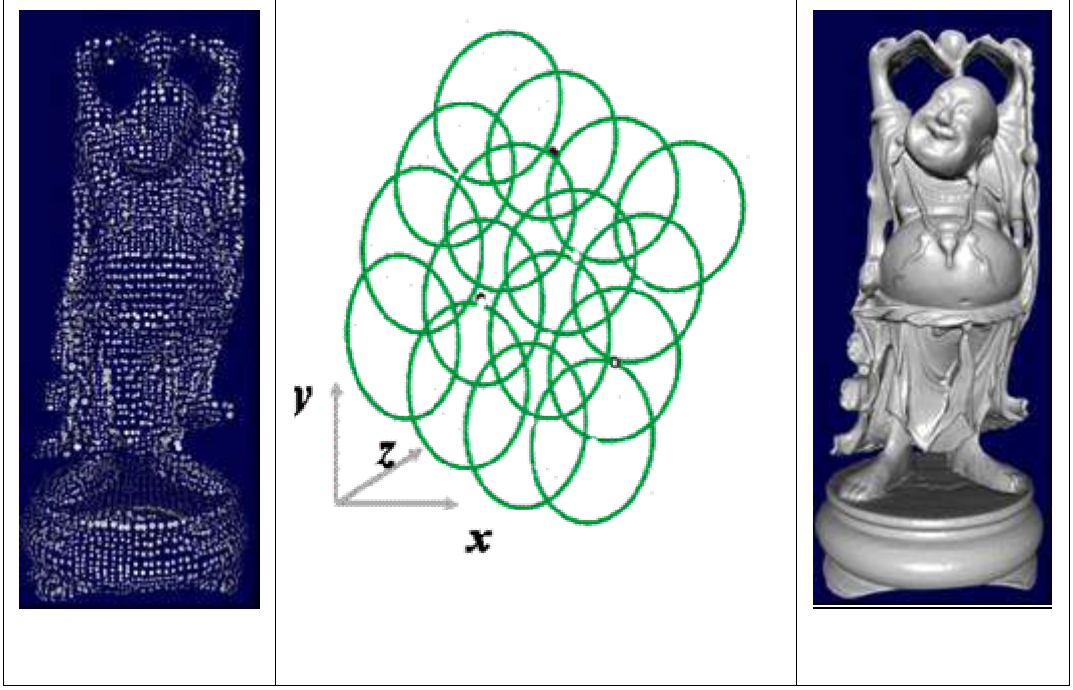


In OpenGL, information about the pixels (for instance, its color or virtual location in 3D space) is organized in memory into bitplanes. A bitplane is an area of memory that holds one bit of information for every pixel on the screen; the bit might indicate how red a particular pixel is supposed to be, for example. The bitplanes are themselves organized into a framebuffer, which holds all the

information that the graphics display needs to control the color and intensity of all the pixels on the screen [21].

Point-based surface splatting rendering, unlike ray-tracing or scan-line rendering, does not use polygonal models in its virtual environment. Instead, data from modern scanning devices are assembled as unconnected point-clouds. Each point, often referred to as a "surfel" contains geometric attributes such as position and normal (depth), and color information [22]. The surfels are enlarged and made elliptical, overlapping and blending with each other to form a crisp image [23, 24]. See Figure 2-15. Point-based rendering is fairly efficient for complex models, but is highly inefficient for simple geometric shapes.

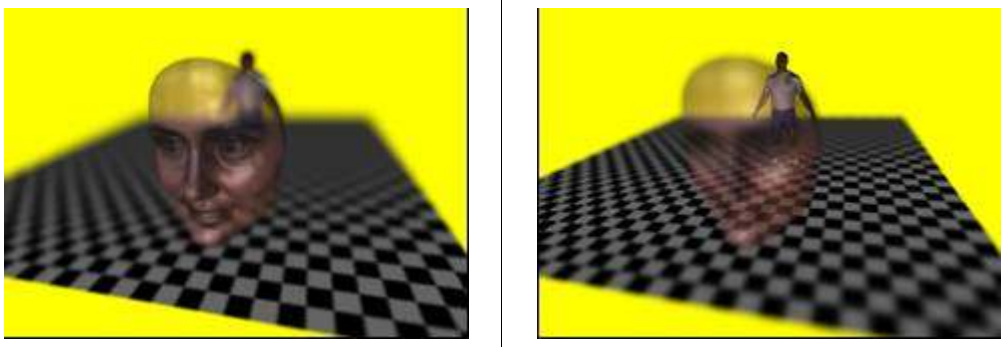
Figure 2-15: Points and Surface Splatting [22]**.** The leftmost image shows the surfels of the model before they have been fully "splatted" to form the finished image on the far right. The middle image shows how the surfels are enlarged and overlap each other to form the image on the far right.



2.7 Past Implementations of Blur & Depth of Field

Multiple techniques to produce blur in relation to a depth of field have been explored. Krivanek and Jiri [18]used point-based surface splatting to generate an accurate depth of field blur. Their algorithm called for calculating the size of the circle of confusion for each point in a model, then increasing the size of the surfel's splat accordingly. Although this method produces an accurate depth blur, it was not chosen for this project because the rendering time was too long for a real-time application (depending on the gaze point and amount of depth blur, the same model would take between .75 sec to 20.2 sec to render on a 1.4GHz Pentium 4 machine with 512MB RAM) [18]and was only ideal for highly complex models whose points were generated from a scanning device. See Figure 2-16.

Figure 2-16: Depth of Field Effect with Surface Splatting. The image to the left has the point of attention (focus) on the mask. The image to the right has the point of attention on the man [18].



Normally, OpenGL behaves similarly to a pin-hole camera, where its virtual camera lens' aperture is infinitely small and the depth of field is too large to produce any depth blur. However, two different implementations have been used to create a depth of field effect in OpenGL. Haeberli and Akeley [25]implemented a hardware-accelerated finite depth of field using an accumulation buffer, a technique often referred to as "jittering." A scene is rendered multiple times with slightly different point of views ("jittering" the camera eye). See Figure 2-17. Each view has an identical image plane that is viewed in exactly the same position. The different scenes are then averaged together using the accumulation buffer. The final result is a scene where objects are more blurred as their distance from the plane of perfect focus, (the plane that always remained in the same position), increases. However, it is only a rough

estimation of the depth of field effect and does not blur the other depth planes relative to the size of the circle of confusion. See Figure 2-18.

However, jittering only approximates a depth of field blur, and the characteristics of blur from a finite depth of field (as graphed in Figure 2-8) are not reproduced; objects are not more blurred in front of the point of focus than those that are behind. In addition, jittering is not a very fast or efficient rendering method because it has to render multiple scenes and eventually combine them. On a machine with 2.00 GB of RAM and a 2.80GHz, 2.79GHz dual processor, the bitmap on the screen will visibly flicker while it is accumulated into the final image. See Figure 2-19.

Figure 2-17: Jittering the Camera in OpenGL. The figure below shows an image of three cubes from three different camera views. "A" and "B" are jittered from the normal view on top. As a result, the viewing volumes are different for each image, where the middle cube lies on the image plane that always has the same viewing volumes and will appear in focus [21].

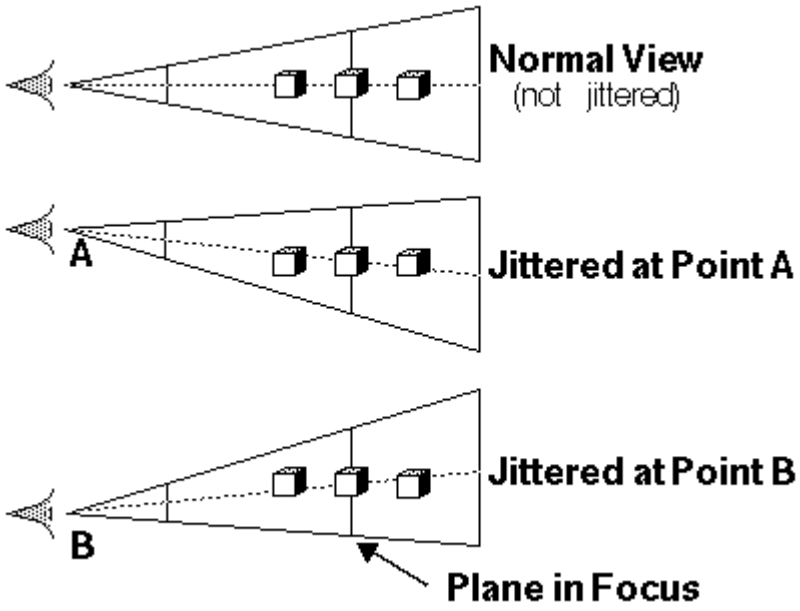


Figure 2-18: Accumulating the Jittered Viewing Volumes. When the different jittered viewing volumes (A and B) are combined, objects outside of the in-focus depth plane will appear blurred, simulating a depth-of-field effect. In this case, the middle cube appears in perfect focus, while the objects behind and in front are blurred.

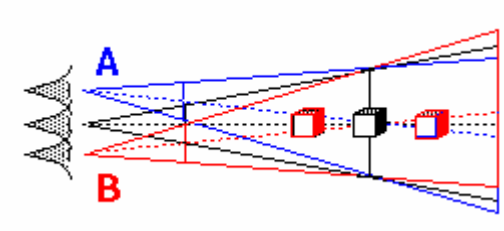


Figure 2-19: Teapots Jittered in OpenGL. In the row of teapots, the gold teapot is in focus while the teapots in front and behind are blurred. However, the front most teapot (the red one) should be blurred more than the ones behind the gold teapot.

Tin-Tin Yu [26] proposed a method to create an accurate depth of field in OpenGL. First, the original CGI is rendered (in perfect focus). Then the red, green, blue (RGB) color data for each pixel of the image is stored into an array and remains unaltered to be a reference of the original image. A secondary color data array holds the information for the image with depth of field blur. A separate array stores the depth values (distance from the virtual camera) of each pixel. Once a gaze position is known, the depth value considered in-focus (accommodated) can be determined by accessing the depth array.

Given the average aperture of the human eye lens, and the depth value of any given pixel in comparison to the depth value in-focus, the size of the circle of confusion generated by each pixel can be calculated. Each pixel that generates a circle of confusion (which is farther from or closer to the lens than

the depth plane in focus) spreads its color data in an equal, circular distribution to all other pixels within the radius of its circle of confusion. See Figure 2-20. This information is stored in the secondary color data array. Then the depth of field image that has been accumulated in the secondary array is drawn. See Figure 2-21 and Figure 2-22.

Figure 2-20: Spreading the Color of a Pixel in OpenGL. The color of point at (x,y) is distributed to all points within its circle of confusion, centered at (x,y). The radius of the circle of confusion is r.

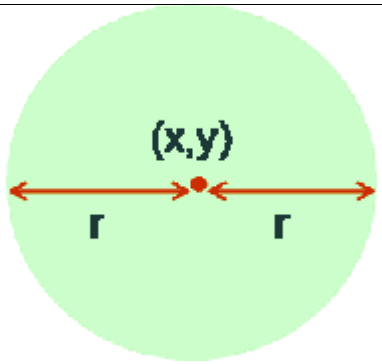
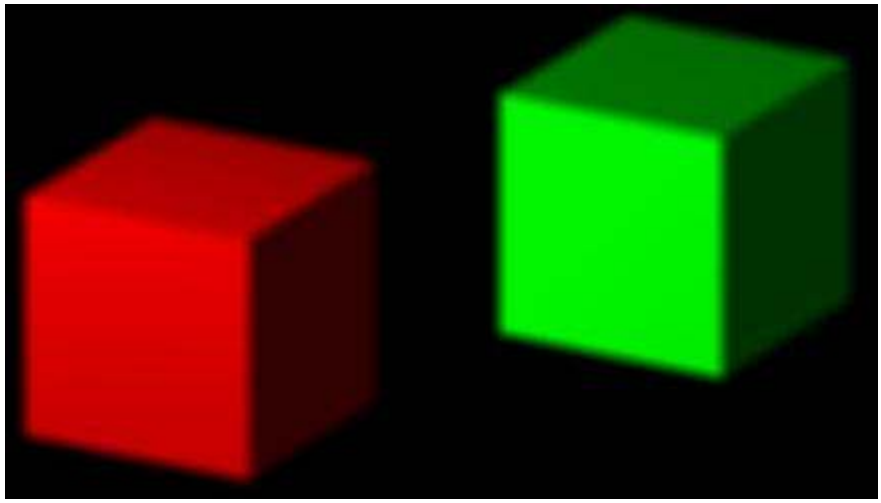


Figure 2-21: OpenGL Depth of Field Algorithm

1. Read in color & depth value of each pixel
2. Calculate the Circle of Confusion diameter for each pixel
3. Color of each pixel distributed to neighboring pixels within its CoC diameter
4. Color data accumulated into a new framebuffer
5. Swap original image buffer with new buffer

Figure 2-22: Blur in OpenGL. Both cubes are blurred uniformly when the circle of confusion size is set as a constant for all pixels.



After exploring a number of algorithms, I decided that rendering images quickly enough for a real-time depth of field display would take us beyond the time available for this project. With rendering time no longer a priority, my exploration focused on generating an accurate finite depth of field.

Ray-tracing can produce a very accurate depth of field blur by calculating the way light from the virtual scene would interact thorough a lens. Maya, a 3D modeling application that can use ray-tracing, was chosen for this project because its user interface makes it fairly easy to control and manipulate the virtual environment and define the point of accommodated depth. Further information on how Maya was used for this project is described in Chapter 3.

2.8 Eye Trackers & Perceptually Adaptive Graphics

Computer graphics are finding unique applications by incorporating an eye tracker, a device that is able to analyze the corneal reflections and the position of the pupil and thereby determine the location of a user's gaze on a computer screen. By knowing the user's point of regard (the position of the eye in space, the direction of gaze, and where/what a person is looking at) an interactive system can "dynamically alter the rendered image in real-time to match the perceptual limits of the Human Visual System"[27].

Interactive systems with eye trackers are generally classified as selective or gaze-contingent. In selective applications, the user's gaze acts as the input, replacing the use of a mouse, keyboard, or touch-screen. These systems have been produced to aid individuals with motor impairments, with applications ranging from games to word processing tools.

Figure 2-23: Eye Tracking System Allows User to Type With His Eyes [28]

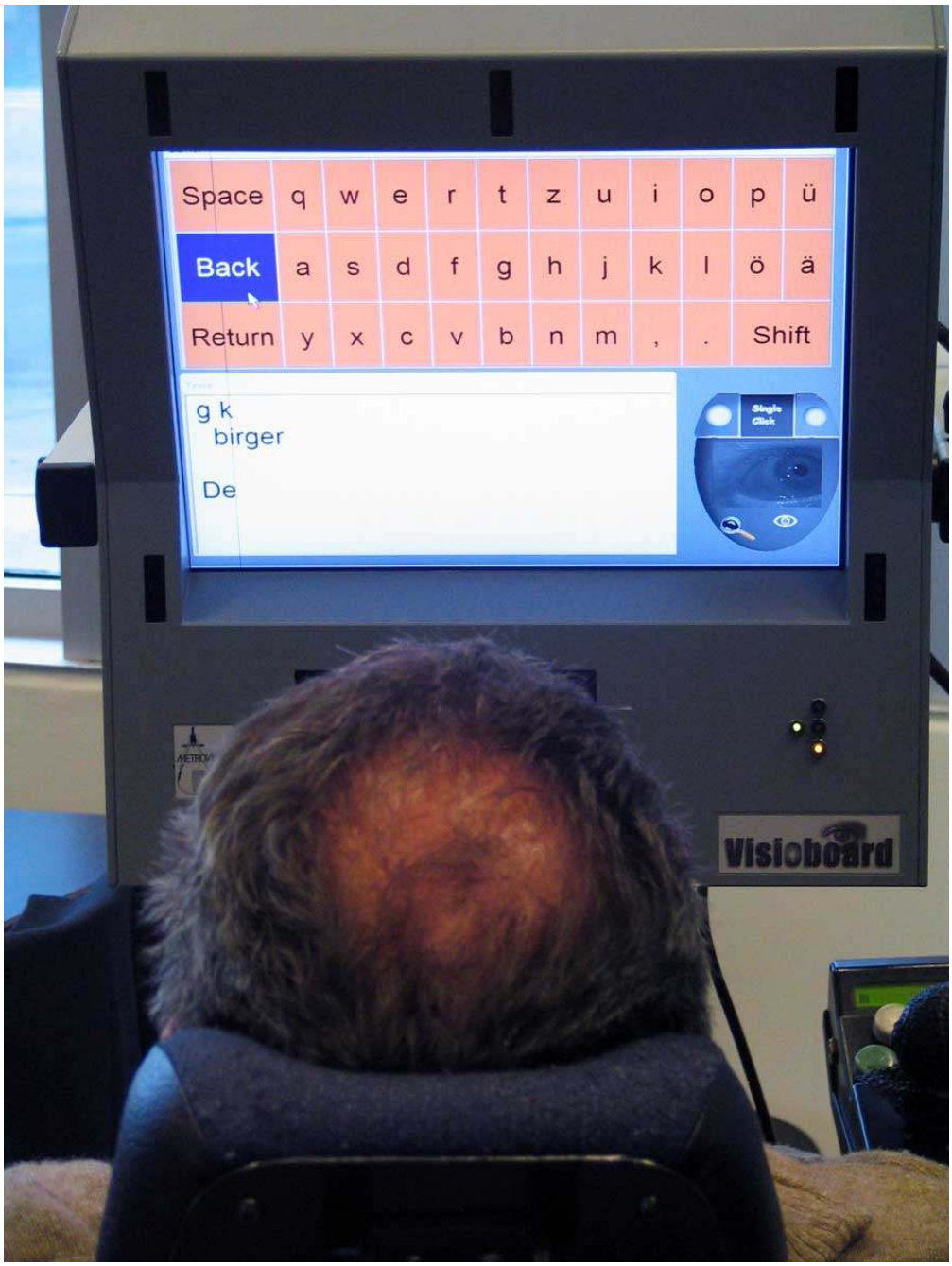

Space q w e r t z u i o p ü
Back a s d f g h j k l ö ä
Return y x c v b n m , . Shift
g k
birger
De
Visioboard

Figure 2-24: Eye Tracking System Allows User to Play Chess With His Eyes [28]

Gaze-contingent displays show a different image at the point of attention than outside the point of attention. The motivation behind most gaze-contingent systems is to use two images, with one at much lower resolution, to minimize the display bandwidth [27]. By reducing the quality of the image only in the subject's peripheral, no vital visual information is lost. A user is often unaware of the decreased resolution because visual acuity drops rapidly outside of the fovea (the small region at the center of the retina with the most densely packed cone photoreceptors). Just 6° away from the line of sight, acuity is reduced by

75% [13, 29]. This phenomenon can be demonstrated when trying to read the words on any line of this page beyond the word being fixated on; it is extremely difficult if not impossible. (The main reason humans spend so much time moving their eyes around is to direct the foveas of the eyes to objects of interest, since the highest visual acuity is so restricted). Making use of this, most gaze-contingent displays use a high-resolution foveal point of attention surrounded by a low-resolution peripheral region.

In implementing gaze-contingent displays there are two main approaches: screen-based and model-based. The model-based approach aims at reducing resolution by directly manipulating the geometric models prior to rendering. An eye tracker is required to present high resolution portions of the scene or object only at the point of highest visual acuity (the foveal point of attention). This method was adapted by Mount Holyoke undergraduate Sanaa Tabani in her senior thesis on Attention Driven Rendering. Tabani optimized a ray tracing algorithm to reduce calculations on the geometric models outside of the user's point of attention [30].

The screen-based approach manipulates the framebuffer contents (the part of memory in the video card destined to contain the image completed. Its size determines the maximum resolution and color depth of the image) just prior to display. The periphery is often masked or smoothed in some way, reducing the bandwidth by compressing the information (in bits-per-pixel)

required to display or transmit the final image[27]. This will be the approach I will take, explained further in Chapter 3.

2.9 Previous Work with Gaze-Contingent Depth of Field

No previous work was found on using a gaze-contingent display in order to produce an accurate depth of field. We speculate that this is because real-time ray-tracing systems are (currently) not possible.

CHAPTER 3

METHODOLOGY

The examination of the impact depth of field may have on human depth perception in a virtual environment involves many challenges: a method to accurately render the computer graphics must be chosen, and the exact gaze position of the user must be known, the correct corresponding visual image must be displayed, all in real-time. In addition, a method for accurately measuring the subject's depth perception must be designed and validated. This section will explain the methodology I used to solve these issues and assemble my system.

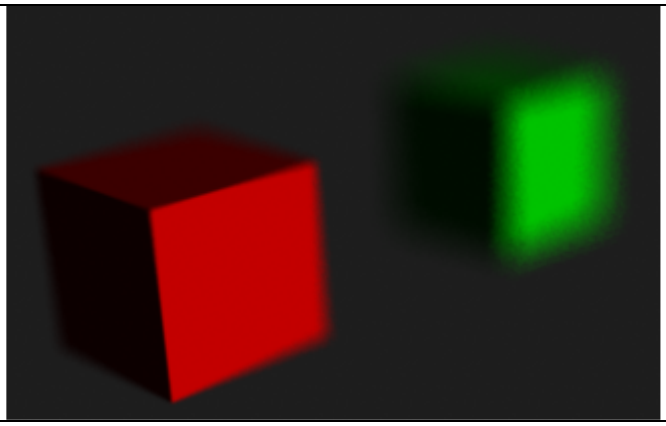
3.1 Rendering Method

The correct amount of blur at a user's gaze position must be displayed within the time constraint of saccades (rapid eye movements) which average only a few milliseconds. Ray-tracing can generate an accurate finite depth of, but even a simple model consisting solely of cubes is computationally expensive and time consuming. As a result, a new strategy was developed; instead of generating the appropriate image on demand in real-time according to the position of a user's gaze, all possible depth-blurred images would be rendered in advance. Then all the images would be loaded into memory such that only the appropriate image would be visible to the user, determined by her gaze position.

As the gaze position changed, the image would be swapped out and replaced with the corresponding depth-blurred image on the monitor.

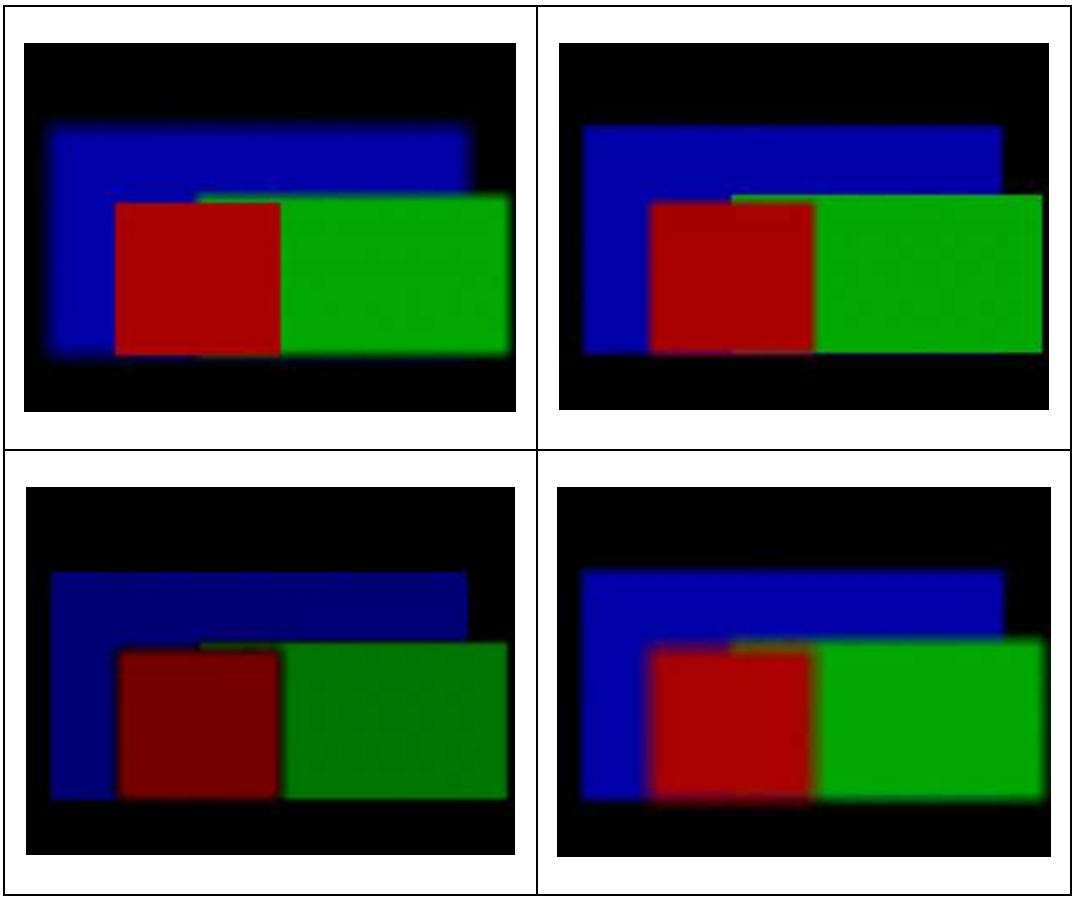
After exploring several different approaches to generate an accurate depth of field blur in a VR system (see Chapter 2), Maya (a 3D modeling application) was chosen because of its ray-tracing algorithm "Mental Ray" which renders an accurate depth of field. (Maya 6.5 was used to generate the necessary images, using Mental Ray 2.1). Mental Ray simulates the physics of a light passing through a lens once the user inputs the focal length, camera lens aperture, and the location of the depth plane in focus. Without using Mental Ray and the appropriate inputs, the image would be rendered as if seen through a pin-hole camera (creating an infinite depth of field and a sharp image). An example of a simple scene rendered in Maya 6.5 with Mental Ray is in Figure 3-0.

Figure 3-0: Depth of Field in Maya. The point of gaze is on the red square and all areas away from this depth are blurred accordingly. The image (before being cropped for this document) was 800 x 600 pixels.



The fewer pre-rendered images used, the smaller the strain on the computer's RAM. To then limit the total number of possible depth-blurred images to a manageable size, the virtual environment needed to be arranged so that there were a small variety of images. The virtual scene configuration chosen has three cubes slightly occluding each other, positioned towards the camera so that only the front face was visible. This is illustrated in Figure 3-1 and 3-2 below. From this set up, there are only four possible depth-blur images: the front (red) cube in focus and all others blurred, the middle (green) cube in focus and all others blurred, the back (blue) cube in focus and all others blurred, eye gaze on the black space with all cubes blurred.

Figure 3-1: Four Possible Depth-Blur Images. The upper left image has the front cube in focus. The upper right image has the middle cube in focus. The lower left image has the back cube in focus. The lower right image has all cubes out of focus. Only the front face of each cube is visible to the camera's point of view to simply the variety of depth-blur possible.

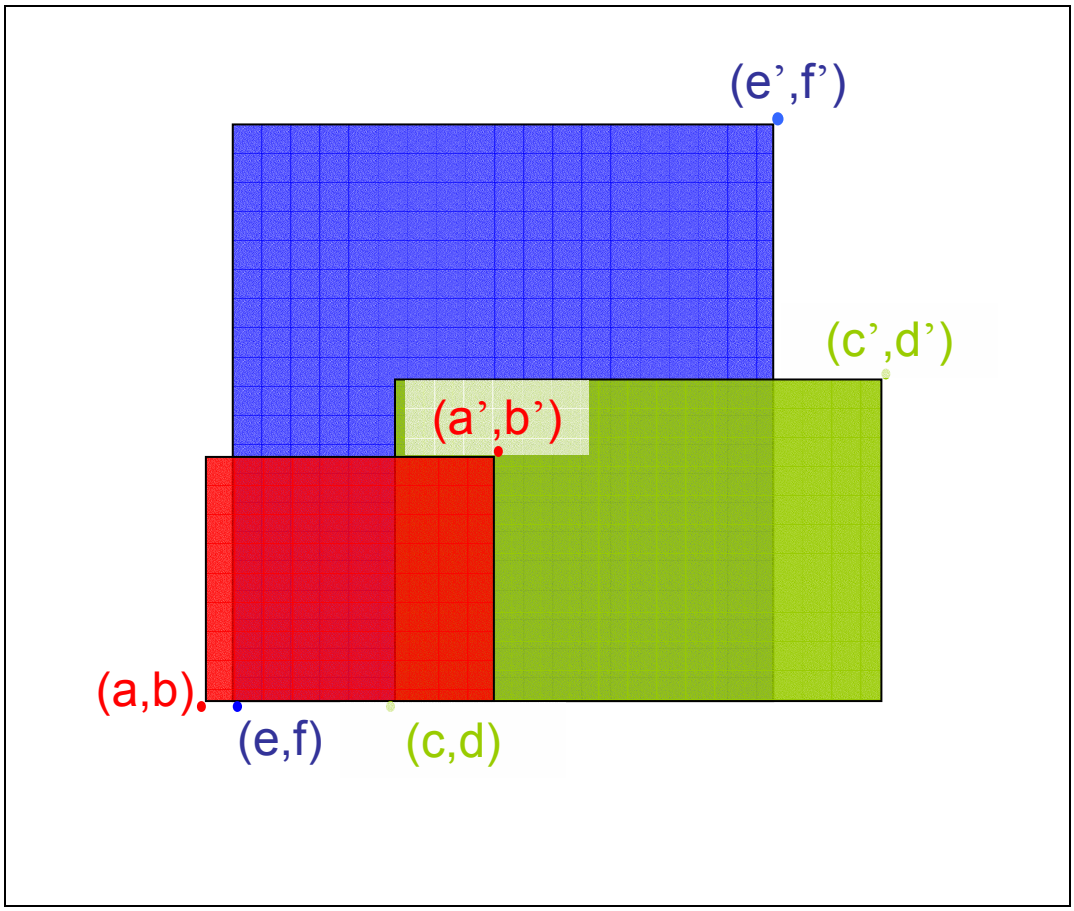


Since the cubes are occluding each other, the gaze-position obtained via an eye-tracker must be analyzed to see which cube the user is actually looking at. The lower left corner coordinates and upper right corner coordinates of the objects are determined beforehand and stored in the program. The lower left corner coordinate of the red cube (always the object in front) is considered to be at position (a, b) and its upper corner coordinates at (a', b'). The middle cube (always in green) and the back cube (always in blue) are labeled similarly, with the corner coordinates of the green object at (c, d) and (c', d'), and the blue object at (e, f) and (e', f'). An illustration of the occluded objects and how their variables are assigned is shown in Figure 3-2. With a screen resolution of ***x***

width and ***y*** height, the eye position data gathered from the eye tracker assumes the upper left corner is at (0,0), the lower left corner at (0,y), the upper right corner at (x,0) and the lower right corner at (x,y). Once the gaze position of the user is known, its x and y coordinates are tested in a series of if-statements:

1. if x > a AND x < a' AND y < b AND y> b', then the user is looking at the front, red object
2. else if x > c AND x < c' AND y < d AND y> d', then the user is looking at the middle, green object
3. else if x > e AND x < e' AND y < f AND y> f', then the user is looking at the back, blue object
4. else, the user is looking at none of the objects

Figure 3-2: Determining which Cube is in Focus. In each image the viewer will be presented, there are three rectangular objects (red, green, and blue) that slightly occlude each other. The coordinates of the lower left corners of the objects are labeled with lowercase letters, and the upper right corners are labeled with prime lowercase letters.



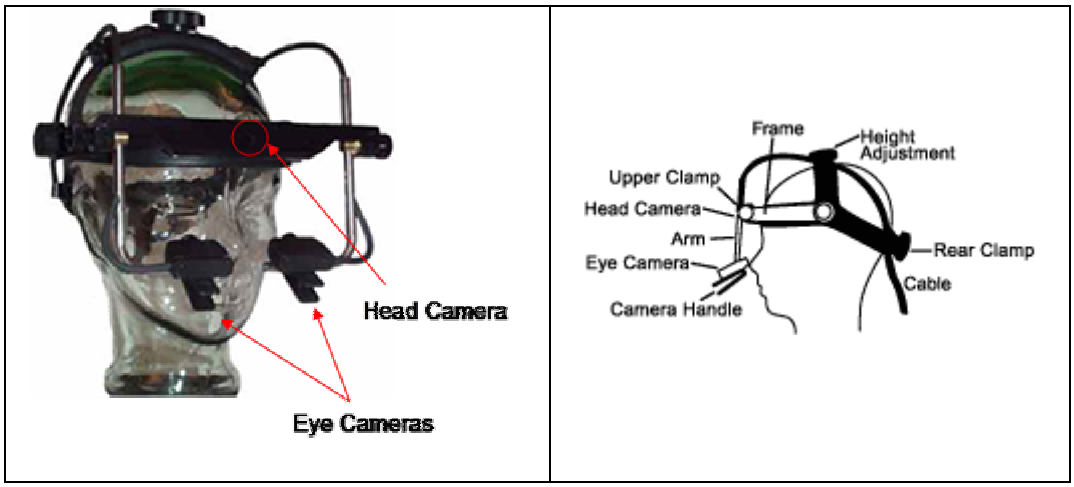
3.2 Eye Tracker: EyeLink II

We purchased the EyeLink II eye tracker (developed by SR Research) to determine the user's gaze position on the screen. A head mounted, video-based eye tracker, the EyeLink II system consists of three miniature cameras mounted on a padded headband. Two eye cameras allow for either binocular or monocular eye tracking of the subject's eye. Each camera has built-in illuminators, which are digitally corrected for even lighting of the entire field of view. Eye camera sensitivity is high enough that most subjects with contacts or

eyeglasses can be successfully tracked, allowing a large pool of possible subjects.

The third camera is for tracking the movements of the subject's head and it is integrated into the headband, above the bridge of the subject's nose. (See Figure 3-3.) The use of a third head tracking camera eliminates the need for a bite bar – normally used to keep the subject still enough for accurate tracking of the subject's point of gaze. The head camera uses four infrared markers – attached to the corners of the monitor display with Velcro – to calculate necessary compensation for minor head movements during eye recording.

Figure 3-3: EyeLinkII Headband & Cameras. The left image is a photograph of the eye tracker on a glass head. The right image shows a peripheral diagram detailing the different parts of the eye tracker. The system is lightweight (the headset is about 420 grams) with a low center of mass for stability and minimal rotational inertia so as not to be uncomfortable to the subject.



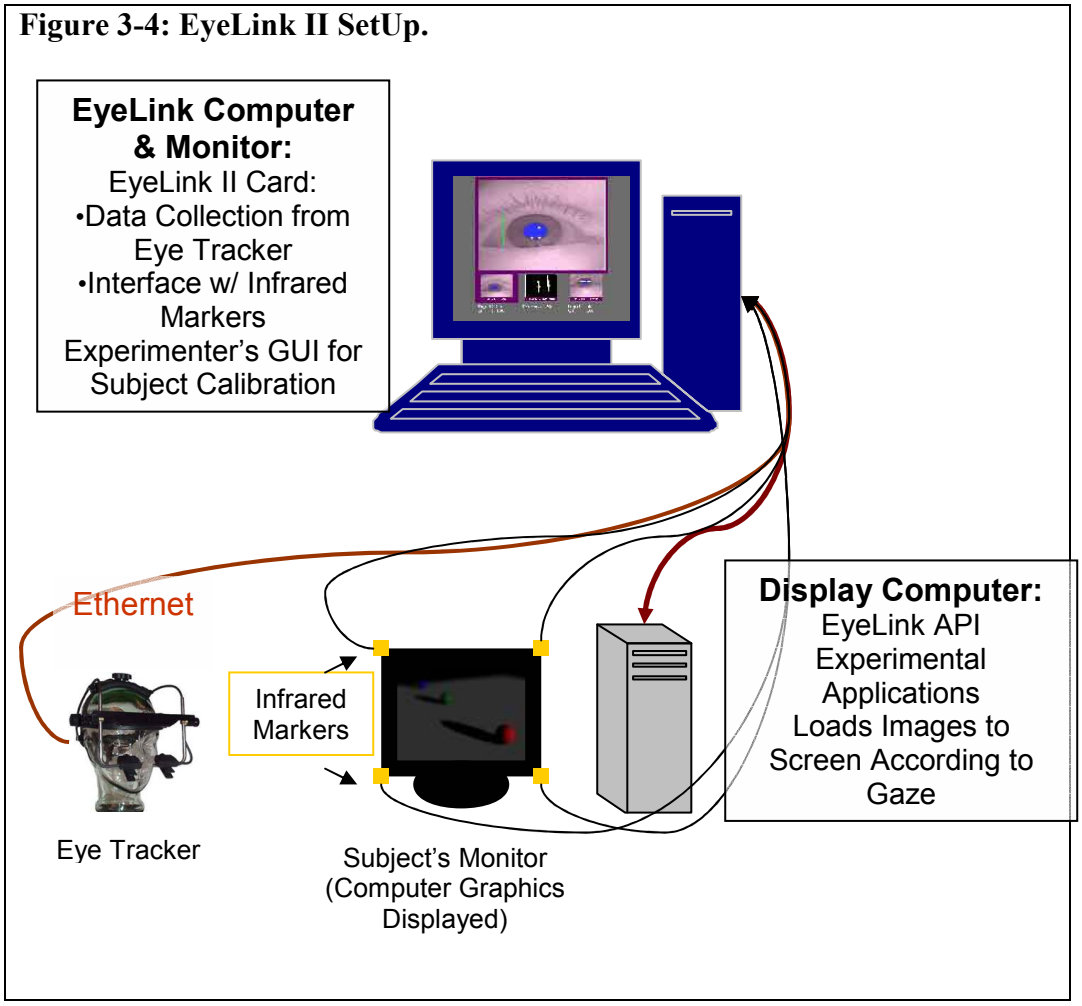
After a subject is calibrated, the point of gaze information gathered is very accurate. The EyeLinkII provides a 500Hz sampling rate, an average gaze position error <0.5°, and real-time access to the eye position data (only 3 msec delay). The gaze tracking range is ±20° in the horizontal and ±18° in the vertical. The head tracking range is between 40-140 cm and the head rotation compensation range is ±15° [31, 32]. In addition, a chin rest will be used for both the comfort of the subject (reducing neck muscle tremor and permitting long periods of use without fatigue) and as a means to reduce head movements compensation.

Collecting data from the eye tracker uses up most of the CPU cycles, therefore the most efficient way to use the EyeLink II system is to have the eye tracker interface with a second computer, the one displaying the graphics, via a high-speed Ethernet link. See Figure 3-4.

Experimental applications, written using the EyeLink API library software on the display computer, configure and control the EyeLink II, which

acts as an intelligent peripheral device. The EyeLink II Tracker Application runs on the tracker's computer in a ROMDOS 7.1 operating environment. This application allows the experimenter to calibrate the subject with the eye tracker, notifies the experimenter if the subject's gaze position is ever lost, and allows termination of any trial or experiment in progress. The application also processes the eye camera and head camera data, and sends any requested data to the display computer in real-time.

Figure 3-4: EyeLink II SetUp.



The display-computer has the EyeLink API (programmed in C) which provides access to the eye sample data in the form of C structures. Samples are available as quickly as 3ms from when the camera image was taken. As data is sent from the tracker's computer to the display's computer via a dedicated Ethernet connection, it is placed in a data queue on the display's computer. This way, no data is lost. Sample data includes:

Field	Contents
flags:	Bits indicating what types of data are present, and for which eye(s)
px, py:	Camera X, Y of pupil center
hx, hy:	HEADREF angular gaze coordinates
pa:	Pupil size (arbitrary units, area or diameter as selected)
gx, gy:	Display gaze position, in pixel coordinates set by the screen_pixel_coords command
rx, ry:	Angular resolution at current gaze position, in screen pixels per visual degree
status:	Error and status flags (reports corneal reflection status and tracking error)

Modifications made to the EyeLink's programming for this project are documented in the Appendix.

3.3 Windows XP & Real-Time Graphics

There were a significant number of obstacles to be overcome in order to have the Windows XP operating system present gaze-contingent graphics in real-time. One such issue was communication between the Windows XP display computer and the eye tracker's computer. Windows NT and its successors (Windows 2000 and XP) do not allow programs to access I/O ports

directly. This problem was solved by use of a hardware I/O port driver that came with the EyeLink II system that allows direct access to I/O ports.

Other problems stemmed from the fact that Windows is a multitasking operating system, which means that other programs can steal time from my experiment's graphic display program at any moment. Even with no other programs running, the Windows kernel will try to steal some time about once a second for maintenance tasks. This can be a serious problem if the operating system seizes control at the instant I wish to update the display; the window movement of a gaze-contingent window display (which I use) would be delayed. Left alone, the execution time of my graphic functions would appear slow and unpredictable.

However, Windows 2000 and XP allow experimental applications to force real-time priority, so graphics and the network may continue to work while almost all other Windows tasks are disabled, including background disk access. The limitation to real-time mode is that certain system functions ceased to work: the keyboard and sound were disabled [32, 33]. As a result, the keyboard of the display computer was not used for subject control or subject responses.

Once in real-time mode, the drawing speed depends on the selected resolution and colors of the graphics. In general, more colors, higher resolution and faster refresh rates all result in slower graphics drawing. Furthermore, in Windows XP graphics are not drawn immediately. Instead, they are "batched" and drawn up to 16 milliseconds later[32]. Fortunately, batching can be turned

off by calling the GdiSetBatchLimit(1) from the EyeLink II API. Batching can prove useful, however, if a moving object needs to be erased and then redrawn at a different position as quickly as possible (which is exactly what a gaze contingent display demands). I enabled batching before drawing and disabled it afterwards.

For further optimization, many display drawing operations under Windows (especially copying bitmaps to the display, as I did) are done in hardware by the display card. The image on the phosphor of the display monitor is redrawn from the video memory, proceeding from top to bottom in about 800/refresh rate milliseconds. This causes changes in graphics to appear only at fixed intervals, when the display refresh process transfers that part of the screen to the monitor. Graphics at the top of the screen will appear about 0.5-2 millisecond after refresh begins, and those at the bottom will appear later [32].

Unfortunately, drawing graphics usually require more than one display refresh period. As a result, the progressive drawing of the visual stimuli to the display will be visible to the subject. To eliminate this hiccup and make the display appear more rapidly, the graphics are drawn to a memory buffer then copied to the display. In this way, the entire display can be updated in one refresh period. This is carried out by the image_file_bitmap() function as part of the EyeLink II API, which uses the freeware FreeImage library to load an image file from disk and create a bitmap from it. Many picture formats work, including BMP, PCX and JPG, but not GIF. Although, JPG files are smallest,

they load more slowly than some other file formats. As a result, I used BMP images in this experiment. These images are also the same resolution as the display mode for the experiment (800 x 600 pixels).

When copying a bitmap to the display, the entire image can be displayed or only a rectangular section of the bitmap can be copied. Obviously, it is faster to create smaller bitmaps and copy these to the center of the display rather than the full sized image. For example, the margins of a page of text usually are 2° from the edges of the display. Copying only the area of the bitmaps within the margins will reduce the copying time by 40%[32]. As a result, my experiment only displays a rectangular section of the correct depth-blur image centered on the point of gaze, while areas outside the rectangle show a similar background image. The size of the rectangle is slightly larger than the foveal range of the subject (about 25% of the display)[29, 34]. In this way, the image can be refreshed in real-time, and the subject will not notice the limited amount of the image displayed to them.

These manipulations require high sampling rates and low delay. Ideally, the experiment should run at a faster display rate (at least 120 Hz), because higher refresh rates mean lower delays between eye movements and the motion of the window. However, the highest refresh rate possible from the equipment available was 85 Hz. The bitmap within the rectangular area in the foveal range is created by copying areas from a "foreground" bitmap to draw within the window, and from a "background" bitmap to draw outside the window. Once the

window and background are initially drawn, the code only needs to redraw those parts of the window or background that changed due to window motion, or if the subject's gaze position resulted in a different depth-blurred image. Even during saccades, only a small fraction of the window area will need to be updated because the window is updated every 2 to 4 milliseconds [32].

At high display resolutions (where pixels are as small as 0.02°), eye tracker noise and microsaccades – a natural phenomena of small, rapid eye movements that always occur, even when our full attention is on a small area [29]– can cause a "quivering" of the window, which makes the edges of the window more visible to the subject. This constant drawing also makes Windows XP less responsive. The solution for this is implementing a "deadband" filter to remove the quivering while not adding any effective delay. This filter can be visualized as a ring on a piece of paper being moved by a pencil through its hole – the ring only moves when the pencil reaches the sides of the hole, and small motions within the hole are ignored. The deadband filter is set to 0.1°, which results in a negligible error in window position.

An important issue for gaze-contingent displays is the system delay from an eye movement to a display change. There are three elements to this delay: the eye tracker delay, the drawing delay, and the display delay. The eye tracker delay is, on average, no more than 3 msec. Some time will always be lost to draw the new gaze-contingent window and for the changed image data to appear on the monitor. The drawing delay is the time from new data being read until

drawing is completed. This depends on the CPU speed, the VGA card, and the display resolution. This is usually less than 1 millisecond for typical saccades using a 10° gaze-contingent window[32] (which is what I will be using), as only small sections of the window are erased and redrawn. The final delay is caused by the time it takes to read out the image from the VGA card's memory to the monitor. This takes at most one refresh period, ranging from 17 milliseconds (for a 60 Hz display) to 6 milliseconds (for a 160 Hz display)[32].

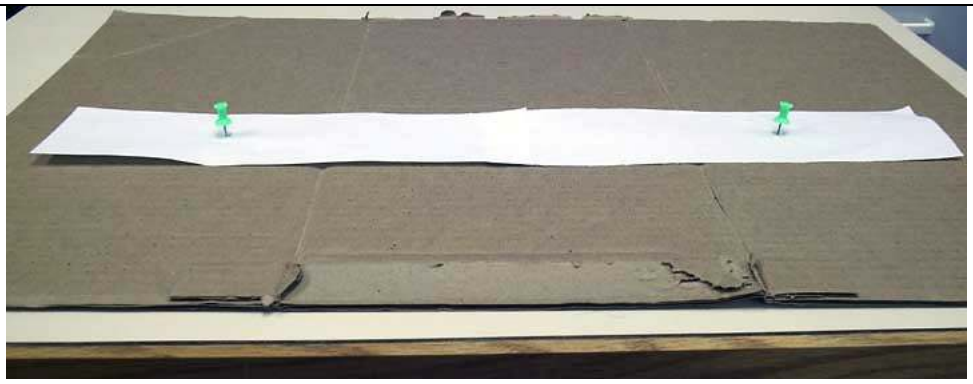
However, the experiment redraws the rectangular gaze-contingent window whenever new eye position data is available and does not wait for a display refresh, which reduces the average delay to ½ of a refresh period. This unsynchronized drawing does not cause any issues because only sections of the window that have changed are redrawn— during most saccades, only a few pixels are changed. The average delay for a gaze-contingent display with this scheme can be as low as 7 milliseconds (for 500 Hz sample rate, no filter, and a 160 Hz refresh rate). Worst-case delays will be higher by ½ refresh period and 1 sample, or 15 milliseconds. This delay has been confirmed using an electronic artificial pupil and a light sensor attached to a monitor [32].

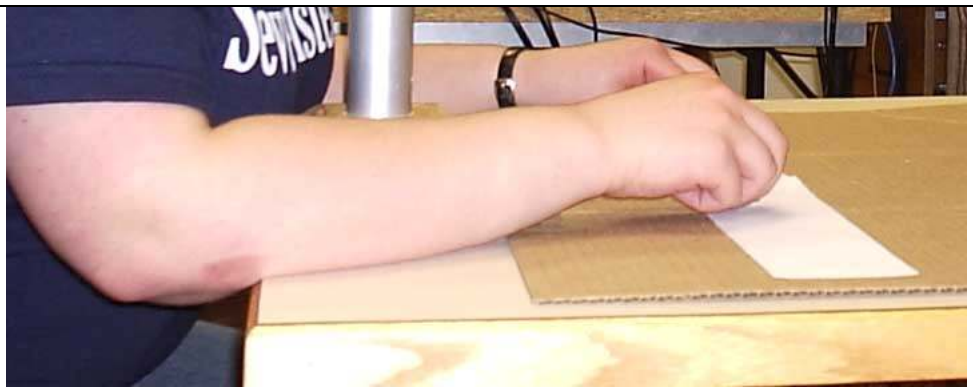
3. 4 Gauging Human Depth Perception

When obtaining a person's depth perception, verbal reports are inaccurate and unreliable [11, 35]. Consequently, past studies of depth and distance perception in virtual environments have relied on walking tasks as a haptic response. Witmer [8] used treadmill walking, Knapp [9] and Mohler [7]

used triangulated walking, Durgin et al. [10] and Willemsen [7] used direct walking. However, the eye tracker requires minimal head and body movement, eliminating the use of a walking task for this study. As a solution, I designed a simple push-pin haptic response to be used for the hands. See Figure 3-5.

Figure 3-5: Push-Pin Haptic Response. Subjects use two push pins and, without looking down, secure them into paper-covered cardboard. The paper is then retrieved, labeled, and the distance between the puncture marks measured and recorded.





CHAPTER 4

EXPERIMENTS & RESULTS

A three-part experiment was implemented to 1) validate the push-pin haptic as a depth perception measurement, and 2) compare the depth perception in real environments and virtual environments with and without a finite depth of field. The subjects' main task was to use the push-pin haptic to indicate the perceived depth between different objects in real and virtual environments.

33 female subjects between the ages of 18 and 25 from Mount Holyoke College participated in this experiment. Data for 5 subjects were not used because it was not possible to calibrate the eye tracker with them. All subjects were naïve to the purposes of this experiment and had normal or corrected-to-normal vision. Subjects enrolled in PSYCH 100 or PSYCH 110 received one research credit for their participation. Other subjects received a movie ticket.

Section 4.1 describes training the subjects to use the push-pin haptic and make depth judgments; section 4.2 describes the experimental set-up for gauging participants' depth perception in a real world environment; section 4.3 discusses the results from section 4.2; section 4.4 describes the experimental set-up for gauging participants' depth perception in virtual environments with the

eye tracker and discusses its results; section 4.5 discusses unexpected results from the experiments.

4.1 Part 1: Training

Subjects were instructed on the definition of depth and the type of judgments required of them. Subjects practiced making depth judgments between two 13.3 cm x 6.0 cm colored blocks of wood (one red and one blue) and practiced using the push-pin haptic to indicate the depth between the two until they felt comfortable. To isolate any possible effect from the order of subjects making judgments in real and virtual environments, half of the subjects proceeded to Part 2 and then Part 3 of the experiment, while half proceeded to Part 3 and then Part 2.

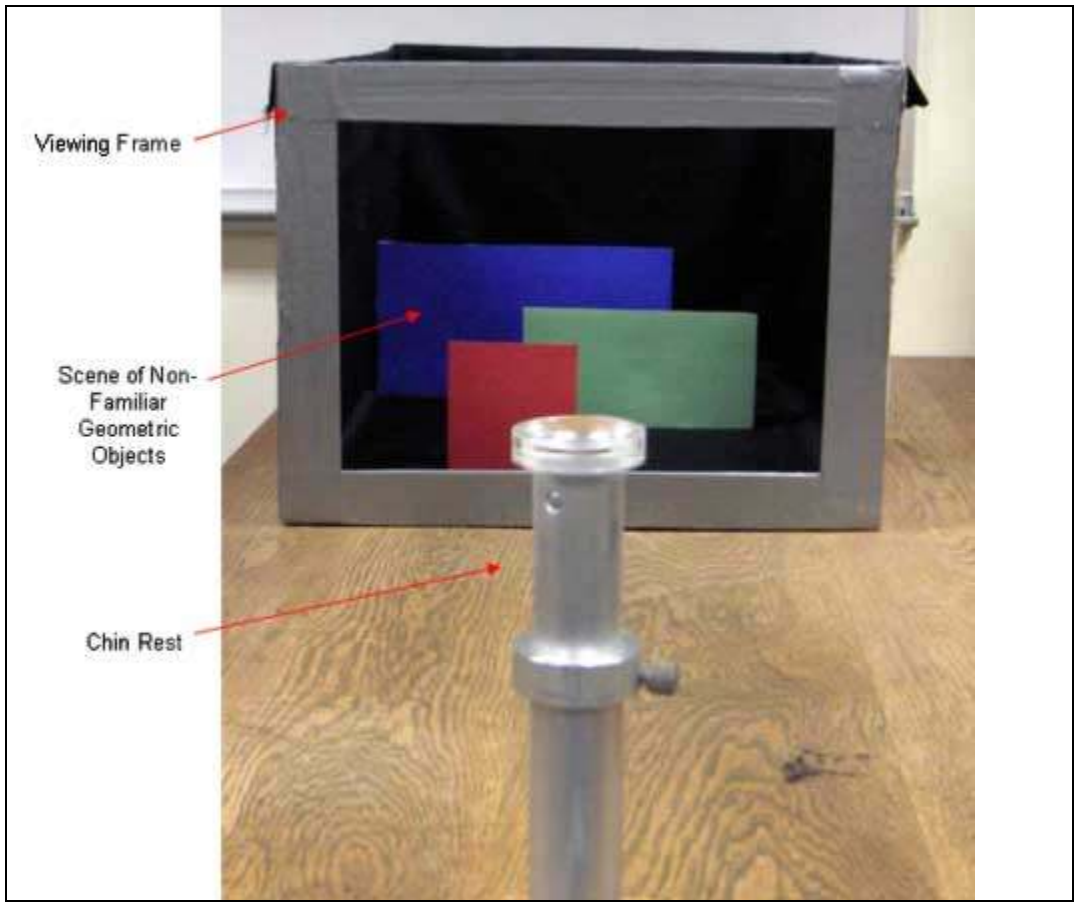
4.2 Part 2: Real World Scenario

Subjects viewed the insides of four different real-world boxes (RWB), labeled “A”, “B”, “C”, and “D”. The viewing order was randomized for each participant. Each RWB was made of cardboard 31 inches deep, 14 inches high and 20 inches wide. The field of view was determined by a 16 x 12 inch rectangular opening, the same dimensions as the computer monitor. Black matte cloth lined the insides of the RWB to help eliminate shadows, to remove any texture gradient as a depth cue, and to limit any depth information from a horizon. A chin rest placed 32 inches in front of the RWB prevented the

participant from moving her head and viewing the scene inside the box from different angles. The view was blocked from the participant until she was situated on the chin rest.

In each box were three unfamiliar objects: cardboard rectangles of different dimensions and colors, in varying positions, perpendicular to the bottom of the box. Each RWB had one red, blue, and green rectangle. See Figure 4-1. For each RWB, subjects used the push-pin haptic to indicate the perceived depth distance between the red and green rectangles (R&G), the green and blue rectangles (G&B), and the red and blue rectangles (R&B). Subjects were always asked these perception questions in the same order: R&G, G&B, R&B.

Figure 4-1: Real World Box "B" and Chin Rest

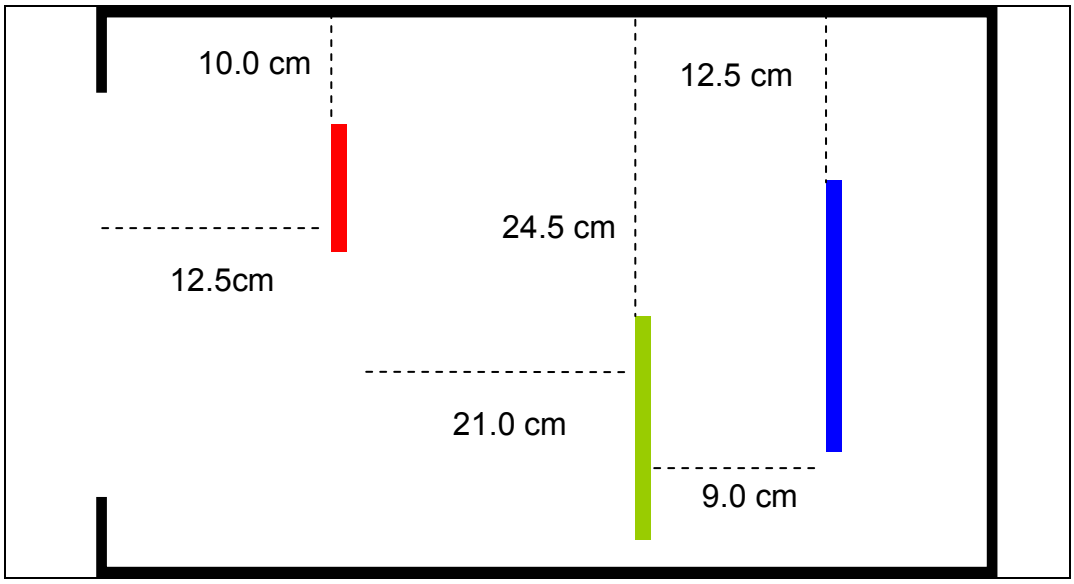


In box "A" the arrangement was:

- 11.0 cm x 9.0 cm red rectangle, 12.5 cm inside the RWB
- 22.0 cm x 13.0 cm green rectangle, 21.0 cm behind the red rectangle
- 22.0 cm x 20.5 cm blue rectangle, 9.0 cm away from the green rectangle.

Further description of the objects' orientation is in Figure 4-1.

Figure 4-2: Top view of RWB "A". The dotted lines represent the distance the object was placed from the side panel of the box. In this configuration, the green object and red object occlude the blue object, but the red object does not occlude the green. Not to scale.

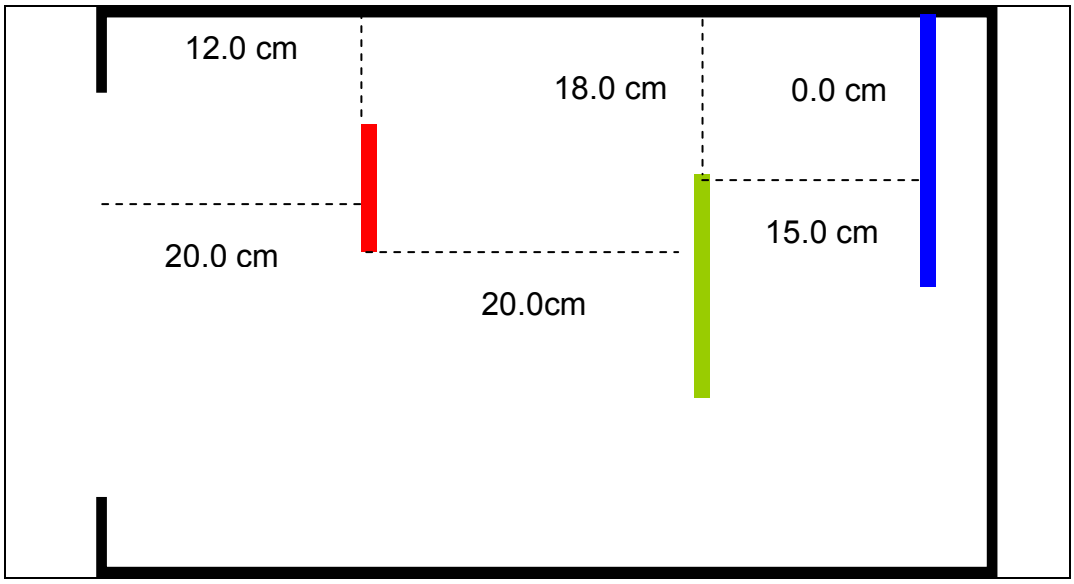


In box "B" the arrangement was:

- 13.0 cm x 12.0 cm red rectangle, 20.0 cm inside the RWB
- 24.5 cm x 12.5 cm green rectangle, 20.0 cm behind the red rectangle
- 33.0 cm x 18.0 cm blue rectangle, 15.0 cm away from the green rectangle.

Further description of the objects' orientation is in Figure 4-3.

Figure 4-3: Top view of RWB "B". The dotted lines represent the distance the object was placed from the side panel of the box. In this configuration, the green object and red object occlude the blue object, and the red object also occludes the green. Not to scale.

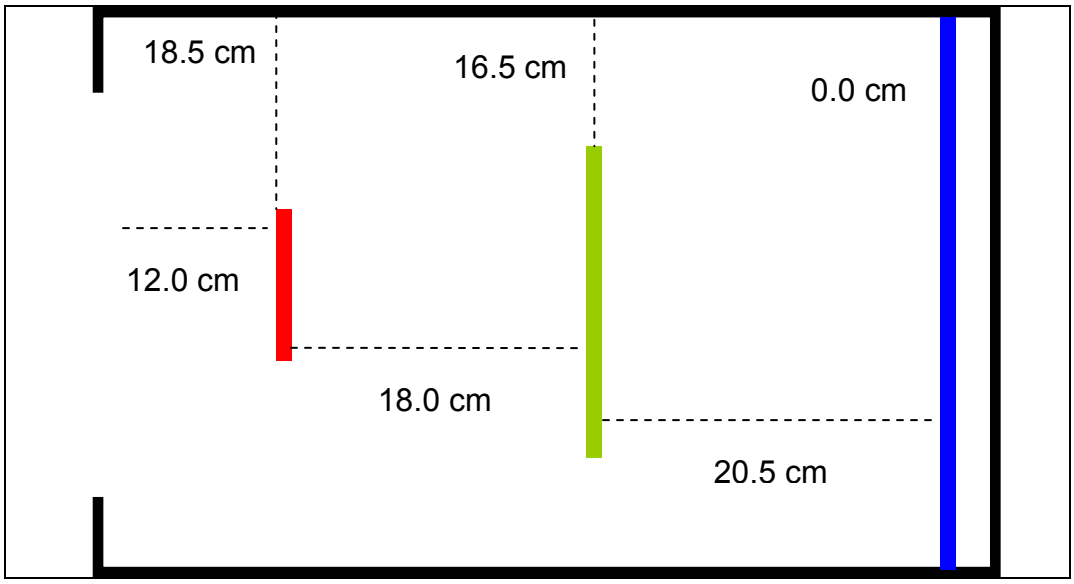


In box "C" the arrangement was:

- 5.0 cm x 5.0 cm red rectangle, 12.0 cm inside the RWB
- 9.0 cm x 17.5 cm green rectangle, 18.0 cm behind the red rectangle
- 18.5 cm x 18.0 cm blue rectangle, 20.5 cm away from the green rectangle.

Further description of the objects' orientation is in Figure 4-4.

Figure 4-4: Top view of RWB "C". The dotted lines represent the distance the object was placed from the side panel of the box. In this configuration, the green object and red object partially occlude the blue object, and the red object also partially occludes the green. Unlike the other configurations, the objects are all centered in the box. Not to scale.

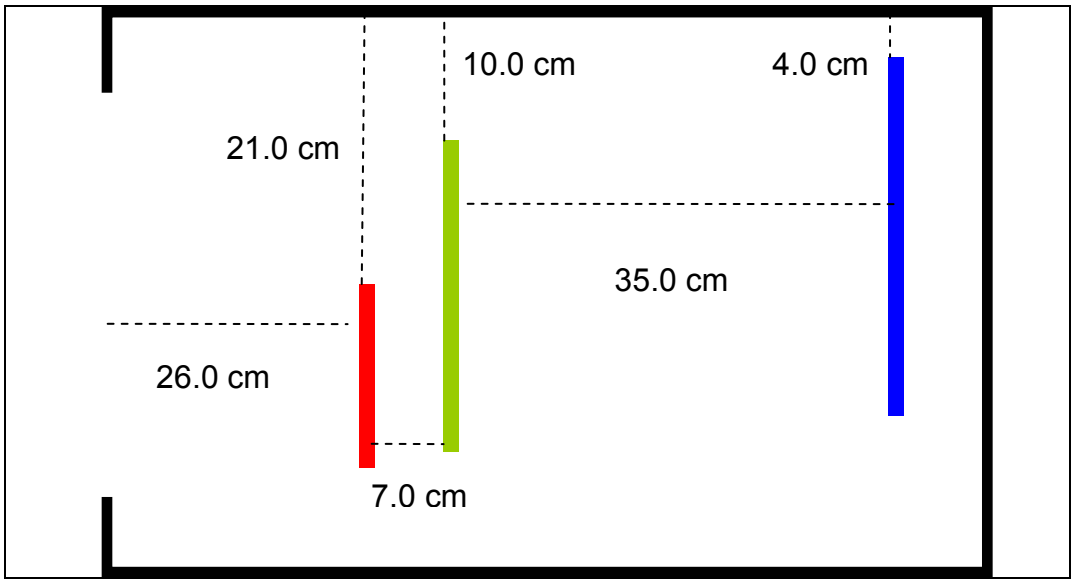


In box "D" the arrangement was:

- 21.0 cm x 10.5 cm red rectangle, 26.0 cm inside the RWB
- 24.0 cm x 7.5 cm green rectangle, 7.0 cm behind the red rectangle
- 24.0 cm x 13.5 cm blue rectangle, 35.0 cm away from the green rectangle.

Further description of the objects' orientation is in Figure 4-5 below.

Figure 4-5: Top view of RWB "D". The dotted lines represent the distance the object was placed from the side panel of the box. In this configuration, the green object and red object partially occlude the blue object, and the red object also partially occludes the green. Not to scale.



4.3 Part 2: Results

The results for Part 2 are charted below in Figure 4-6. The average ratio of depth perceived to the actual depth is 89%. This value is only 3% lower than Witmer & Sadowski's [8]study using direct walking and indicates that the push pin haptic works adequately enough to measure subjects' perceived distance and depth. The results also provide us with a comparison metric for the distances and depths perceived in the virtual environments in Part 3.

Figure 4-6: Real World Judgments. The ratio of the average distances perceived to the actual distances for each RWB. R & G is the distance between the red and green objects, G & B is the distance between the green and blue objects, and R & B is the distance between the red and blue objects.

	Box A	**Box B**	**Box C**	**Box D**
R&G	83%	83%	80%	99%
G&B	100%	95%	87%	87%

R&B	89%	88%	87%	96%
AVERAGE	**90%**			

4.4 Part 3: Virtual Environments w/ Eye Tracker

Subjects wore the head mounted eye tracker (while on a chin rest) and were calibrated. The chin rest was 32 inches from the monitor (twice the display width) for optimal tracking. Subjects viewed 8 randomized images rendered from models of the scenes inside the four RWB. A chin rest prevented the participant from moving her head, viewing the monitor from different angles and from losing calibration with the eye tracker. Four of the virtual scenes were static and had an infinite depth of field, and four of the virtual scenes were gaze-contingent and had a finite depth of field. If the user gazed at the red object, the green and blue objects would be blurred accordingly, etc. For each scene, subjects used the push-pin haptic to indicate the perceived depth between the red and green rectangles, the green and blue rectangles, and the red and blue rectangles.

Figure 4-7: Subject Wearing Eye Tracker and Viewing Virtual Environment.

The results for the virtual environment are charted below in Figures 4-8 and 4-9. For static images without any depth of field blur, the average ratio of distances perceived to the distances modeled in the virtual environment was 15%. For the dynamic, gaze-contingent images with depth of field blur the average ratio of distances perceived to the distances modeled in the virtual environment was 22%.

Figure 4-8: Judgments Without Depth of Field Blur. The ratio of the average distances perceived to the actual distances modeled for the virtual image. R & G is the distance between the red and green objects, G & B is the distance between the green and blue objects, and R & B is the distance between the red and blue objects.

	Scene A	**Scene B**	**Scene C**	**Scene D**
R&G	7.0%	9.0%	6.0%	54%
G&B	21%	19%	7.0%	7.0%
R&B	13%	14%	8.0%	17%

AVERAGE	**15%**

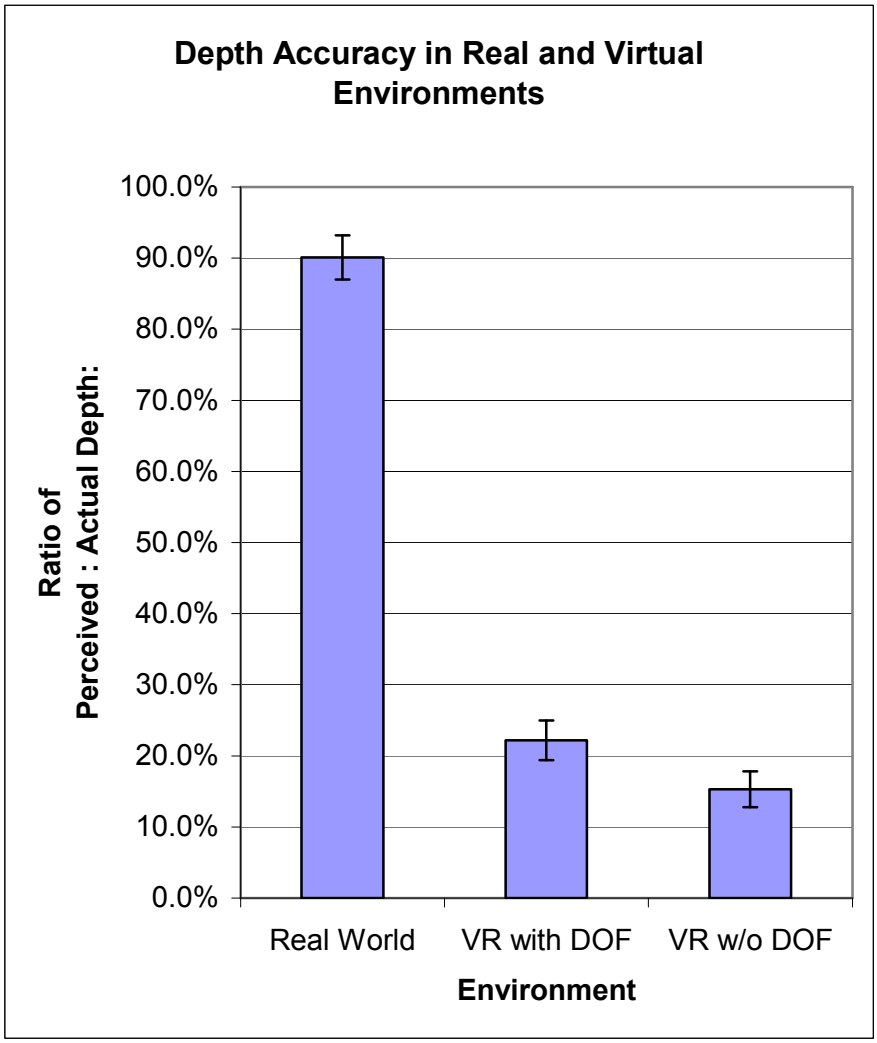
Figure 4-9: Judgments With Gaze-Contingent Depth of Field Blur. The ratio of the average distances perceived to the actual distances modeled for the virtual image. R & G is the distance between the red and green objects, G & B is the distance between the green and blue objects, and R & B is the distance between the red and blue objects.

	Scene A	**Scene B**	**Scene C**	**Scene D**
R&G	10%	17%	17%	76%
G&B	29%	25%	15%	10%
R&B	13%	20%	16%	21%
AVERAGE	**22%**			

A repeated measures analysis of variance (ANOVA) showed significant differences between the three different environments. On average, depth judgments in the real environment vs. the virtual environment without depth of field blur were 75% more accurate ($p<.001$) , but judgments in the real environment vs. the virtual environment with gaze-contingent depth of field blur were only 68% more accurate ($p<.001$). In addition, depth judgments in the virtual environment with gaze-contingent depth of field blur were 7% greater than judgments in the virtual environment without depth of field blur ($p<.005$). See Figure 4-10.

Figure 4-10: Average Accuracy of Depth Perception in VR and Real World. There is a significant difference ($p<.001$) between the Real World judgments and VR with DOF (virtual environment with finite depth of field), between the

Real World judgments and VR w/o DOF (virtual environment without blur and an infinite depth of field) and between VR with DOF and VR w/o DOF ($p<.005$).



4.5 Surprising Results

Scene configuration affected depth perception by a significant factor ($.001<p<.05$). Scenes A and B showed no significant differences, but Scene C was significantly less accurate (distances were greatly compressed) and Scene D

was significantly more accurate (distances were less compressed) than the other scene configurations.

Figure 4-11: Scene Configuration Affects Depth Perception. The average accuracy between different scene configurations among the real and virtual environments is graphed below. Scene C was significantly less accurate while Scene D was significantly more accurate than the other scene configurations.

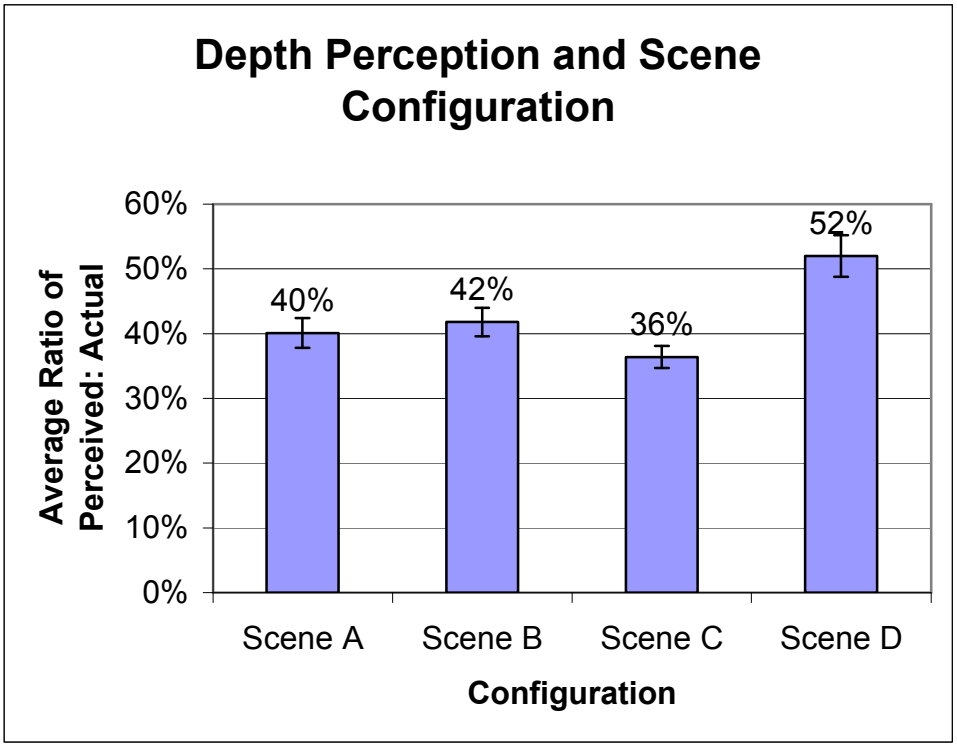


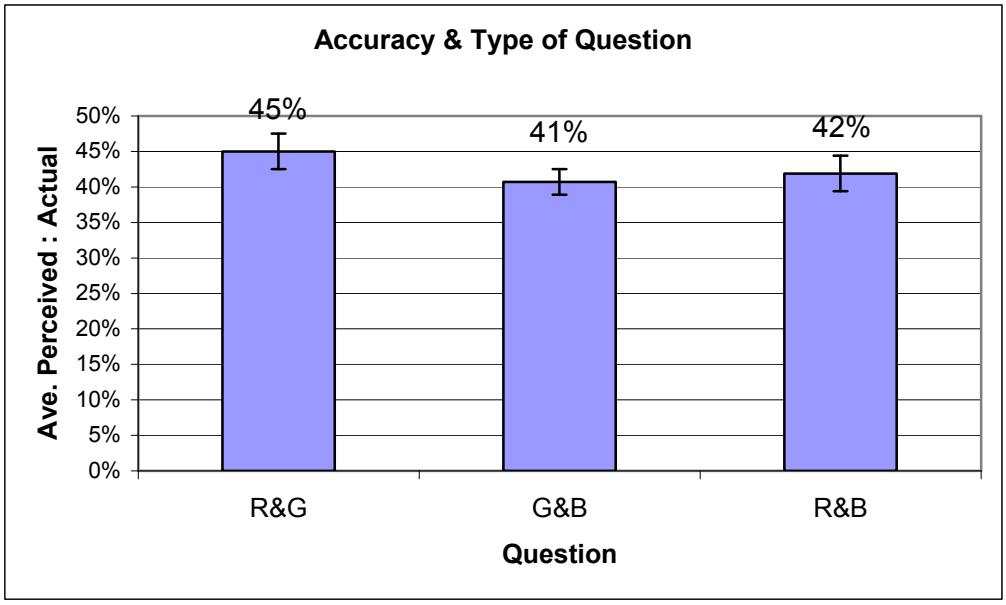
Figure 4-12: Statistically Significant Mean Differences Between Scene Configurations. An asterisk (*) indicates significant difference. Scenes C and D always yielded a significant difference.

Scene (I)	Scene (II)	Mean Difference (I - II)	Std. Error	Significance
A	B	-.017	.017	.337

	C	.037(*)	.018	.048
	D	-.119(*)	.028	.000
B	C	.054(*)	.015	.002
	D	-.102(*)	.026	.001
C	D	-.156(*)	.026	.000

Another unexpected result was an effect from the type of question: depth between the R&G, G&B, or R&B. When averaged across the real and virtual environments, a significant difference between the R&G and G&B judgments was found ($p<.02$), with R&G judgments being 4% more accurate than G&B. However, there was no significant difference between R&B judgments with any other question-type. See Figure 4-13.

Figure 4-13: Question Effect Averaged Across All Environments. R&G yielded 4 % higher accuracy than G&B ($p<.02$).



CHAPTER 5

CONCLUSIONS & FUTURE WORK

This research project aimed to explore the depth compression in virtual environments and attempted to improve those depth perceptions. The general hypothesis was that the compression effect in virtual environments was due to missing and conflicting depth cues in the visual scene. From this general hypothesis came several sub-hypothesis: 1) accommodation is a significant conflicting depth cue in present virtual reality systems, 2) the visual effect accommodation causes, depth of field blur, also acts as a depth cue, and 3) adding a correct depth of field blur will improve depth perception in a virtual environment. Experiments were conducted to directly test the third sub-hypothesis, and the results supported it.

The following section discusses the results and possible explanations for those results.

5.1 Discussion of Hypothesis and Results

When we added a gaze-contingent depth of field blur to a set of computer generated images, there was a 7% increase in the depth perceived when compared with the same images without a depth of field blur. Although this supports our hypothesis, there was a larger compression factor in both sets of images than recorded in pervious work. See Figure 5-1.

Figure 5-1: Compression in Virtual Environments. These studies compared real-world distance judgments ("Real") and judgments made with computer generated images ("VR"). "Compression" is the difference between the Real and VR judgments. The percentage is the ratio of perceived distance to actual distance. The results from this study are in the last two rows.

Study	***Real***	***VR***	***Compression (Real – VR)***
Witmer & Sadowski (1998)	92%	85%	15%
Knapp (1999)	100%	42%	58%
Willemsen & Gooch (2002)	100%	81%	19%
Mohler, Thompson, et al (2004)	95%	44%	51%
No Depth of Field Blur	**90%**	**15%**	**75%**
With Depth of Field Blur	**90%**	**22%**	**68%**

This larger underestimation of distances may be due to two conflicting depth cues: accommodation and stereopsis. Our stereo vision and proprioceptive sense in our ocular muscles tell us that the virtual environment viewed on the computer monitor is flat. However, this alone does not explain why the compression factor was greater in this study than in the others shown above. It can be speculated that an additional cognitive factor, the belief that the virtual environment is only a computer generated image on a monitor, may also have contributed to this compression.

From the results, a finite depth of field sends cues that the distances between objects are farther than if there was no blur and an infinite depth of field. However, Figures 5-2 and 5-3 make it apparent that this increase in perceived distances is not relative. For example, subjects will gauge a distance

they previously believed (in the real world) to be 17.3 cm and then perceive it to be 2.2 cm in the virtual environment. Then, for a distance they previously believed (in the real world) to be 16.5 cm they then perceived it to be 3.4 cm in the virtual environment. Therefore, a finite depth of field and blur does not provide absolute depth information nor provide relative depth information. It simply adds to the amount of distance perceived between objects.

Figure 5-2: Average Distances Perceived Without Blur and in the Real World. NoD is the perceived distance in the virtual environment with an infinite depth of field and no blur, and R is the real world perceived distance. R & G is the distance between the red and green objects, G & B is the distance between the green and blue objects, and R & B is the distance between the red and blue objects. Measurements are in centimeters.

	Scene A		**Scene B**		**Scene C**		**Scene D**	
	NoD	*R*	*NoD*	*R*	*NoD*	*R*	*NoD*	*R*
R&G	1.4	17.3	1.8	16.5	1.0	14.4	3.8	6.9
G&B	1.9	9.0	2.8	14.2	1.5	17.8	2.5	30.5
R&B	3.9	26.8	4.8	30.6	3.2	33.4	7.3	40.3

Figure 5-3: Average Distances Perceived With Gaze-Contingent Depth of Field Blur and in the Real World. DOF is the perceived distance in the virtual environment with blurring and a finite depth of field, and R is the real world

perceived distance. R & G is the distance between the red and green objects, G & B is the distance between the green and blue objects, and R & B is the distance between the red and blue objects. Measurements are in centimeters.

	Scene A		**Scene B**		**Scene C**		**Scene D**	
	DOF	*R*	*DOF*	*R*	*DOF*	*R*	*DOF*	*R*
R&G	2.2	17.3	3.4	16.5	3.1	14.4	5.3	6.9
G&B	2.6	9.0	3.7	14.2	3.0	17.8	3.6	30.5
R&B	3.9	26.8	6.9	30.6	6.2	33.4	8.8	40.3

5.2 Future Work from Hypothesis

Our work eliminated most other depth cues from the real and virtual environments. It is possible there is an interaction between depth of field blur and other depth cues -- shadows, textures, and a visible horizon – that would yield a more accurate perception of depth than with the sum of those depth cues alone.

In the Real World Boxes, the effect a horizon would play on depth perception was limited by the black interior, but in the future this could be completely eliminated by suspending the objects with thin wire or fishing line. This would better emulate the virtual environment.

When participants were viewing objects in the virtual environments with the eye tracker, it was observed that they lingered on the edges between objects when making depth judgments. If a subject became marginally un-calibrated

during the experiment, then it is possible they the appropriate image depth-blurred image did not appear at their gaze position. Having a more complex scene with textures and more complex objects may draw attention away from the edges and help to eliminate this potential source for error.

It is probable that the computer graphic images with depth of field blur did not have the appropriate amount of blur. An exact relationship between amount of depth blur and the depth perceived in computer generated images has yet to be established. A larger amount of blur may improve depth perception further, or at a certain point stop or even detract from improving depth perception.

Another potential source for error was in the push pin haptic. Subjects were not always successful in aligning the push pins perfectly parallel. An adjustment to the push pin haptic could be made such that it mimicked a slide rule, providing a guide to ensure the punctures were aligned.

It is possible that the lack of realism in the virtual environments played an important factor in underestimating depth. If the user feels overly conscious that they are viewing objects on a flat monitor, they may be unconsciously reducing their depth estimates. A photorealistic texture on the same rectangular objects with the same configuration, or a more complex and realistic scene may yield different results.

To eliminate the depth information provided by stereopsis, future work could require participants to wear an eye patch. Accommodation is considered

to be a monocular depth cue, so it is reasonable to believe depth of field may be as well.

5.3 Unexpected Results Explained

There were two statistically significant factors found in the experiments that were unexpected: scene configuration and type of question. Scene configuration may significantly help or hinder depth perception (Chapter 4, Figures 4-11 & 4-12) and when making depth judgments subjects were significantly more accurate judging between the red and green objects than between the green and blue.

However, this effect may have nothing to directly do with the scene configuration. When comparing the ratio of perceived distance to modeled distance, subjects were the least accurate in judging Scene C and G&B (which on average had the largest distances between objects) than in Scene D and R&G (which on average had the closest distances between objects). However, this metric was skewed to favor Scene D and G&B judgments because small increases in the perceived distance would yield a larger ratio than the same amount of increase in Scene C or in G&B. See Figures 5-2 and 5-3. When performing a comparison of the distances perceived in the virtual environment with an infinite depth of field vs. the perceived distance in the virtual environment with blurring and a finite depth of field, Scene C yields a larger increase in perceived distance than Scene D. See Figure 5-4.

Figure 5-4: Average Distances Perceived With Gaze-Contingent Depth of Field Blur and in Real World. NoD is the perceived distance in the virtual environment with an infinite depth of field and no blur, DOF is the perceived distance in the virtual environment with blurring and a finite depth of field, and % is the ratio between DOF and NoD (DOF/NoD). R & G is the distance between the red and green objects, G & B is the distance between the green and blue objects, and R & B is the distance between the red and blue objects. The average increase in perceived distance from the DOF is shown below.

	Scene A			**Scene B**			**Scene C**			**Scene D**		
	NoD	*DOF*	*%*	*NoD*	*DOF*	*%*	*NoD*	*DOF*	*%*	*NoD*	*DOF*	*%*
R&G	1.4	2.2	152%	1.8	3.4	191%	1.0	3.1	314%	3.8	5.3	140%
G&B	1.9	2.6	138%	2.8	3.7	134%	1.5	3.0	202%	2.5	3.6	142%
R&B	3.9	3.9	100%	4.8	6.9	143%	3.2	6.2	193%	7.3	8.8	120%
Average Increase in Perceived Distance				**164%**								

5.4 Conclusion

In conclusion, the results show (with statistical significance) that depth perception in virtual environments can be positively affected by incorporating blur and a finite depth of field. Virtual Reality systems that are currently suffering from distance compression may benefit from including this feature and attention driven rendering techniques could be applied to generate this cue. However, this gain is expected to be small, as blur and depth of field do not provide absolute or relative depth information.

References

[1] M. Lumbreras and J. Sánchez, "3D aural interactive hyperstories for blind children," in 2nd European Conference on Disability, Virtual Reality and Associated Technologies, 1998, pp. 119-128.

[2] Y. Yanagida, S. Kawato, H. Noma, A. Tomono and N. Tesutani, "Projection based olfactory display with nose tracking," *Virtual Reality, 2004.Proceedings.IEEE,* pp. 43-50, 2004.

[3] A. Ananthaswamy. (2003, 31 July 2003). Virtual reality conquers sense of taste. *New Scientist* pp. 15 Feb. 2006. Available: http://www.newscientist.com/article.ns?id=dn4006&print=true

[4] G. Burdea and P. Coiffet, *Virtual Reality Technology.* New York: Wiley, 1994, pp. xvi, 400 , [8].

[5] K. Kania. (2000, May 2000). Virtual reality moves into the medical mainstream. *MDDI* Available: http://www.devicelink.com/mddi/archive/00/05/004.html

[6] A. Bianchi. (1994, July 1994). Virtual reality gets real. *Inc. Magazine* pp. December 1, 2005. Available: http://www.inc.com/magazine/19940701/3000.html

[7] B. J. Mohler, W. B. Thompson, S. Creem-Regehr, Pick, Jr. Herbert L., J. Warren William, J. J. Rieser and P. Willemsen, "Visual motion influences locomotion in a treadmill virtual environment," in *First SIGGRAPH Symposium on Applied Perception in Graphics and Visualization,* August 2004,

[8] B. G. Witmer and W. J. Sadowski, "Nonvisually guided locomotion to a previously viewed target in real and virtual environments," *Hum. Factors,* vol. 40, pp. 478-488, SEP. 1998.

[9] J. M. Knapp, "The Visual Perception of Egocentric Distance in Virtual Environments," 1999.

[10] F. H. Durgin, A. Pelah, L. F. Fox, J. Lewis, R. Kane and K. A. Walley, "Self-motion perception during locomotor recalibration: More than meets the eye," *Journal of Experimental Psychology: Human Perception and Performance,*

[11] W. B. Thompson, P. Willemsen, A. A. Gooch, S. H. Creem-Regehr, J. M. Loomis and A. C. Beall, "Does the Quality of the Computer Graphics Matter when Judging Distances in Visually Immersive Environments?" *Presence: Teleoperators & Virtual Environments,* vol. 13, pp. 560-571, 10//. 2004.

[12] M. A. Goodale and A. D. Milner, *Sight Unseen : An Exploration of Conscious and Unconscious Vision.* Oxford ; New York: Oxford University Press, 2004, pp. ix, 135.

[13] S. E. Palmer, *Vision Science : Photons to Phenomenology.* Cambridge, Ma: MIT Press, 1999, pp. xxii, 810.

[14] C. R. Nave, "HyperPhysics: Accommodation," vol. 2006, 2005.

[15] H. Wallach and L. Floor, "Use of Size Matching to Demonstrate Effectiveness of Accommodation and Convergence as Cues for Distance," *Percept. Psychophys.,* vol. 10, pp. 423-&, 1971.

[16] M. Bailey, T. Rebotier and D. Kirsh, "Quantifying the relative roles of shadows, stereopsis, and focal accomodation in 3D visualization," in *3rd IASTED International Conference Visualization, Imaging, and Image Processing,* 2003, pp. 992-997.

[17] G. Schneider, B. Heit, J. Honag and J. Bre'mont, "Monocular depth perception by evaluation of the blur in defocused images," in *International Conference on Image Processing,* 1994, pp. 116-119.

[18] J. Krivanek and J. Zara, "Rendering depth-of-field with surface splatting," Czech Technical University, Prague, Czech Republic, Tech. Rep. DC-2003-02, 2003.

[19] Buren, J. M. Van, *The Retinal Ganglion Cell Layer. A Physiological-Anatomical Correlation in Man and Primates of the Normal Topograhical Anatomy of the Retinal Ganglion Cell Layer and its Alterations with Lesions of the Visual Pathways.* Springfield, Illinois: Charles C. Thomas, 1963,

[20] C. Hormann, "balcony.pov," 2001.

[21] M. Woo, J. Neider, T. Davis and D. Shreiner, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2.* ,3rd ed.Reading, Massachusetts: Silicon Graphics, 1999,

[22] H. Pfister, M. Zwicker, J. van Baar and M. Gross, "Surfels: Surface elements as rendering primitives," in *SIGGRAPH 2000,* 2000,

[23] M. Zwicker, H. Pfister, J. van Baar and M. Gross, "Surface splatting," in *SIGGRAPH 2001,* 2001,

[24] J. Rasanen, "Surface splatting: Theory, extensions and implementation." May 2002.

[25] P. Haeberli and K. Akeley, "The accumulation buffer: Hardware support for high- quality rendering," in *SIGGRAPH,* 1990, pp. 309-318.

[26] T. Yu, "Depth of Field Implementation with OpenGL," *Journal of Computing Sciences in Colleges,* vol. 20, pp. 136-146, October 2004. 2004.

[27] A. T. Duchowski, "Eye tracking techniques for perceptually adaptive graphics," in *SIGGRAPH 2001 Campfire on Perceptually Adaptive Graphics,* 2001,

[28] Information Society Technologies: Communication by Gaze Interaction. (2006, January 16, 2006). COGAIN: Communication by gaze interaction. *2006(02/13),*

[29] L. C. Loschky and G. W. McConkie, "Investigating Spatial Vision and Dynamic Attentional Selection Using a Gaze-Contingent Multiresolutional Display," *Journal of Experimental Psychology: Applied,* vol. 8, pp. 99-117, June 2002. 2002.

[30] S. F. Tabani. (2003, Attention driven rendering. [MH]. Available: http://www.mtholyoke.edu/acad/compsc/Honors/Sanaa-Tabani/index.htm

[31] SR Research. (2006, March 10, 2006). SR research EyeLink: EyeLink II - head mounted system overview. *2006*Available: http://www.eyelinkinfo.com/mount_main.php

[32] SR Research Ltd., "Programming EyeLink® Experiments in Windows: Version 2.1 (GDI)," 3/14/2004. 2004.

[33] D. L. Farquhar and I. NetLibrary, *Optimizing Windows for Games, Graphics and Multimedia [Electronic Resource]*. ,1st ed.Beijing: O'Reilly, 2000, pp. 278.

[34] W. S. Geisler and J. S. Perry, "Variable-resolution displays for visual communication and simulation," *The Society for Information Display,* vol. 30, pp. 420-423, 1999.

[35] A. U. Mutsuddi and Mount Holyoke College. Computer Science Dept. (2004, The effects of texture mapping techniques on slant perception of planar surfaces textured with flat textures. [MH]. pp. 124 leaves.

APPENDIX: CODE FOR GAZE-CONTINGENT EXPERIMENTS

Alterations from the EyeLink II code by SR Research indicated with comments such as “Chi modified: [date]:”

A. w32_demo_main.c

```
/*************************************************
******/

/* CONTENTS:                                                */

/* - Stub for Windows (WinMain(), etc)                      */

/* - app_main() for experiment program:                     */

/* - connect to tracker                                     */

/* - check display mode, create full-screen window          */

/* - set up calibration colors, targets                     */

/* - open EDF file on tracker                               */

/* - configure tracker                                      */

/* - call run_trials() to do experiment                     */

/* - close EDF file, transfer over link                     */

/* - clean up, exit                                         */

/*************************************************/


#include <windows.h>

#include <windowsx.h>


#include "gdi_expt.h"

#include "w32_demo.h"
```

```
      int is_eyelink2;   // set if we are connected to EyeLink
II tracker


                  // the application instance: required to create
windows and get resources

      HANDLE application_instance = NULL;


                  // display information: size, colors, refresh
rate

      DISPLAYINFO dispinfo;


                  // The colors of the target and background for
calibration and drift correction

      COLORREF target_foreground_color = RGB(0,0,0);

      COLORREF target_background_color = RGB(192,192,192);


                  // Name for experiment: goes in task bar, and
in EDF file

      char program_name[100] = "Windows Sample Experiment 2.0";


      app_main()

      {

        int i, j;

        char our_file_name[260] = "TEST";


        if(open_eyelink_connection(0)) return -1;    // abort if
we can't open link

        set_offline_mode();
```

```
        flush_getkey_queue();                          //
initialize getkey() system

        is_eyelink2 = (2 == eyelink_get_tracker_version(NULL) );


        get_display_information(&dispinfo);          // get
window size, characteristics


           // NOTE: Camera display does not support 16-color
modes

           // NOTE: Picture display examples don't work well
with 256-color modes

           //       However, all other sample programs should
work well.

        if(dispinfo.palsize==16)      // 16-color modes not
functional

          {

            alert_printf("This program cannot use 16-color
displays");

            goto shutdown;

          }

        if(dispinfo.refresh < 40)  // wait_for_refresh doesn't
work!

          {

            alert_printf("No refresh synchroniztion
available!");

          }

        if(dispinfo.palsize)     // 256-color modes: palettes
not supported by this example

          {

            alert_printf("This program is not optimized for 256-
color displays");

          }
```

```
        if(make_full_screen_window(application_instance)) goto
shutdown;  // create the window

        if(init_expt_graphics(full_screen_window, NULL)) goto
shutdown;   // register window with EXPTSPPT



        i = SCRWIDTH/60;        // select best size for
calibration target

        j = SCRWIDTH/300;       // and focal spot in target

        if(j < 2) j = 2;

        set_target_size(i, j);  // tell DLL the size of target
features



        target_foreground_color = RGB(0,0,0);    // color of
calibration target

        target_background_color = RGB(128,128,128);       //
background for calibration and drift correction

        set_calibration_colors(target_foreground_color,
target_background_color); // tell EXPTSPPT the colors



        set_cal_sounds("", "", "");

        set_dcorr_sounds("", "off", "off");

                                          // draw a title
screen

        clear_full_screen_window(target_background_color);    //
clear screen

        get_new_font("Times Roman", SCRHEIGHT/32, 1);         //
select a font

                                                              //
Draw text

        graphic_printf(target_foreground_color,
target_background_color, 1, SCRWIDTH/2, 1*SCRHEIGHT/30,
```

```
                        "EyeLink Demonstration Experiment: Sample
Code");

        graphic_printf(target_foreground_color,
target_background_color, 1, SCRWIDTH/2, 2*SCRHEIGHT/30,

                        "Included with the Experiment Programming
Kit for Windows");

        graphic_printf(target_foreground_color,
target_background_color, 1, SCRWIDTH/2, 3*SCRHEIGHT/30,

                        "All code is Copyright (c) 1997-2002 SR
Research Ltd.");

        graphic_printf(target_foreground_color,
target_background_color, 0, SCRWIDTH/5, 4*SCRHEIGHT/30,

                        "Source code may be used as template for
your experiments.");



        i = edit_dialog(full_screen_window, "Create EDF File",
// Get the EDF file name

                        "Enter Tracker EDF file name:",
our_file_name, 8);

        if(i==-1) goto shutdown;          // ALT-F4: terminate

        if(i==1)  our_file_name[0] = 0;   // Cancelled: No file
name



        if(our_file_name[0])    // If file name set, open it

          {

            if(!strstr(our_file_name, "."))
strcat(our_file_name, ".EDF");  // add extension

            i = open_data_file(our_file_name);
// open file

            if(i!=0)
// check for error

              {

                alert_printf("Cannot create EDF file '%s'",
our_file_name);
```

```
                goto shutdown;

              }
// add title to preamble

            eyecmd_printf("add_file_preamble_text 'RECORDED BY
%s' ", program_name);

          }

                                                  // Now configure
tracker for display resolution

        eyecmd_printf("screen_pixel_coords = %ld %ld %ld %ld",
// Set display resolution

                        dispinfo.left, dispinfo.top,
dispinfo.right, dispinfo.bottom);

        eyecmd_printf("calibration_type = HV9");
// Setup calibration type

        eyemsg_printf("DISPLAY_COORDS %ld %ld %ld %ld",
// Add resolution to EDF file

                        dispinfo.left, dispinfo.top,
dispinfo.right, dispinfo.bottom);

        if(dispinfo.refresh>40)

          eyemsg_printf("FRAMERATE %1.2f Hz.",
dispinfo.refresh);



                  // SET UP TRACKER CONFIGURATION

                       // set parser saccade thresholds
(conservative settings)

        if(is_eyelink2)

          {

            eyecmd_printf("select_parser_configuration 0");  //
0 = standard sensitivity

          }

        else

          {
```

```
            eyecmd_printf("saccade_velocity_threshold = 35");

            eyecmd_printf("saccade_acceleration_threshold =
9500");

          }

                        // set EDF file contents

        eyecmd_printf("file_event_filter =
LEFT,RIGHT,FIXATION,SACCADE,BLINK,MESSAGE,BUTTON");

        eyecmd_printf("file_sample_data  =
LEFT,RIGHT,GAZE,AREA,GAZERES,STATUS");

                        // set link data (used for gaze cursor)

        eyecmd_printf("link_event_filter =
LEFT,RIGHT,FIXATION,SACCADE,BLINK,BUTTON");

        eyecmd_printf("link_sample_data  =
LEFT,RIGHT,GAZE,GAZERES,AREA,STATUS");

                            // Program button #5 for use in
drift correction

        eyecmd_printf("button_function 5
'accept_target_fixation'");



        if(!eyelink_is_connected() || break_pressed()) goto
end_expt;  // make sure we're still alive



                  // RUN THE EXPERIMENTAL TRIALS (code depends
on type of experiment)

                      // Calling run_trials() performs a
calibration followed by trials

                      // This is equivalent to one block of an
experiment

                      // It will return ABORT_EXPT if the
program should exit

        i = run_trials();



      end_expt:               // END: close, transfer EDF file
```

```
        set_offline_mode();               // set offline mode so
we can transfer file

        pump_delay(500);                  // delay so tracker is
ready

        eyecmd_printf("close_data_file"); // close data file



        if(break_pressed()) goto shutdown;          // don't get
file if we aborted experiment

        if(our_file_name[0])                        // make sure
we created a file

          receive_data_file(our_file_name, "", 0);  // transfer
the file, ask for a local name



      shutdown:                // CLEANUP

        close_expt_graphics();           // tell EXPTSPPT to
release window

        close_eyelink_connection();      // disconnect from
tracker

        close_full_screen_window();

        return 0;

      }




              // WinMain - Windows calls this to execute
application

      int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE
hPrevInstance, LPSTR lpCmdLine, int nCmdShow)

      {

        application_instance = hInstance;  // record the
application instance for accessing resources

        full_screen_window = NULL;
```

```
        app_main();                         // call our real
program

        close_eyelink_connection();         // make sure EYELINK
DLL is released

        if(full_screen_window)

          close_full_screen_window();

        return 0;

      }
```

B. w32_gcwindow.c

```
      /*********************************************************

      /* CONTENTS:
/* - draw and move rectangular gaze-contingent window     */

/* - use GdiSetBatchLimit() to allow erasing and drawing  */

      /*   of differences with minimum possible delay
*/

      /*********************************************************
*/

      #include <windows.h>

      #include <windowsx.h>

      #include "gdi_expt.h"


      /************* DISPLAY PART OF BITMAP **********/


              // Copies a rectangular area of a bitmap to the
display

              // Copies rectangle <xs1, ys1, xs2, ys2> of DDB
<hbm>

              // Places top-left corner at <xd, yd>

              // <hbm> is source bitmap, <hwnd> is the window to
draw to

      static void display_rect_bitmap(HWND hwnd, HBITMAP hbm,
int xd, int yd,

                                      int xs1, int ys1, int xs2,
int ys2)

      {

        HDC hdc;

        HDC mdc;

        HBITMAP obm;
```

```
        int h, w;


        if(!hwnd || !hbm) return;

        w = xs2 - xs1 + 1;

        h = ys2 - ys1 + 1;


        hdc = GetDC(hwnd);                // display drawing
context

        mdc = CreateCompatibleDC(hdc);    // memory context

        obm = SelectObject(mdc, hbm);     // select bitmap


        BitBlt(hdc, xd, yd, w, h, mdc, xs1, ys1, SRCCOPY);   //
copy relevant part to display


        SelectBitmap(mdc, obm);           // release GDI
resources

        DeleteDC(mdc);

        ReleaseDC(hwnd, hdc);

      }




      /********** FAST GAZE-CONTINGENT WINDOW ************/


              // This gaze-contingent window uses differential
updates for speed.

              // If the window moves a small amount, then only a
small area

              // of the display will actually need to be
redrawn.
```

```
              // These functions identify and update the regions

              // that need to be redrawn using the foreground
and background bitmaps.



              // This variable determines if the foreground or
background area is redrawn first.

              // If <erase_first> is zero, then the background
area is updated first.

              // This will help to hide visual information in
the window.

              // Otherwise, the foreground area is updated
first, to hide information outside.

              // So it should be set only if the window is
masking foveal information.

      static int erase_first = 1;



      /*

      Chi 2-27-06: originally window_w = 90, window_h = 75

      */



      static int window_w = 250;     // the width of the
gcwindow in pixels

      static int window_h = 250;     // The height of the
gcwindow in pixels



      static int curr_wl, curr_wr;  // The sides of the window
as currently drawn

      static int curr_wt, curr_wb;



      #define GCWINDOQW_HIDDEN   0     // window not visible

      #define GCWINDOQW_VISIBLE  1     // window visible
```

```
      #define GCWINDOQW_INIT    -1     // window not yet drawn


      static int window_drawn;      // window state (on of the
GCWINDOW_xxx values).


      static int last_x, last_y;    // last drawn position, for
motion detection

      static int wdeadband;


      static HWND hwnd;      // the window we will draw in

      static RECT dr;        // the display rectangle


      // Chi modified: extra variables

      static HBITMAP fgbm1;   // front focus

      static HBITMAP fgbm2;   // middle focus

      static HBITMAP fgbm3;   // back focus

      static HBITMAP fgbm;   // foreground bitmap

      static HBITMAP bgbm;   // background bitmap

      static HBITMAP last_fg = NULL; // last foreground image
used in gcwindow


              // Initial setup of gaze-contingent window before
drawing it.

              // Sets size of window, and whether it is a foveal
mask.

              // If height or width is -1, the window will be a
bar covering the display

              // <deadband> sets number of pixels of anti-jitter
applied
```

```
      void CALLTYPE initialize_gc_window(int wwidth, int
wheight,

                                    HBITMAP window_bitmap, HBITMAP
background_bitmap,

                                                HBITMAP
window_bitmap1, HBITMAP window_bitmap2, //Chi modified 3-08-06
more params

                                                HBITMAP
window_bitmap3,

                                    HWND window, RECT display_rect,
int is_mask, int deadband)

      {

        window_w = wwidth;        // set window parameters

        window_h = wheight;

        erase_first = !is_mask;



        window_drawn = GCWINDOQW_INIT;       // "false" window
covers full screen

        dr = display_rect;

        hwnd = window;

        fgbm = window_bitmap;

        bgbm = background_bitmap;

        //Chi modified 3-08-06

            fgbm1 = window_bitmap1;

            fgbm2 = window_bitmap2;

            fgbm3 = window_bitmap3;



        curr_wt = dr.top;             // This will properly draw
window and background

        curr_wl = dr.left;

        curr_wr = dr.right;
```

```
        curr_wb = dr.bottom;



        wdeadband = deadband;

        last_x = last_y = MISSING;

      }


      #define MIN(a,b) ((a<b)?(a):(b))

      #define MAX(a,b) ((a>b)?(a):(b))


              // Fill in required areas with background bitmap

      static void erase_gc_window(HBITMAP fg, HBITMAP bg, int
wl, int wt, int wr, int wb)

      {

        if(wt>curr_wb || wb<curr_wt || wl>curr_wr || wr<curr_wl)
// if no intersection,

          {
// just erase current window

            display_rect_bitmap(hwnd, bg, curr_wl, curr_wt,

                                                          curr_wl,
curr_wt, curr_wr, curr_wb);

          }

        else                  // otherwise, update non-
intersecting regions

          {

            if(curr_wt<wt)    // erase any needed top

            display_rect_bitmap(hwnd, bg, curr_wl, curr_wt,


curr_wl, curr_wt, curr_wr, wt-1);
```

```
            if(curr_wb>wb)    // erase any needed bottom
            display_rect_bitmap(hwnd, bg, curr_wl, wb+1,
curr_wl, wb+1, curr_wr, curr_wb);

            if(curr_wl<wl)    // erase any needed left
            display_rect_bitmap(hwnd, bg, curr_wl, MAX(wt,
curr_wt),
curr_wl, MAX(wt, curr_wt),
                                                    wl-1,
MIN(wb, curr_wb));
            if(curr_wr>wr)    // erase any needed right
            display_rect_bitmap(hwnd, bg, wr+1, MAX(wt,
curr_wt),
                                                    wr+1,
MAX(wt, curr_wt),
                                              curr_wr, MIN(wb,
curr_wb));
          }
      }

              // Fill in required areas with foreground bitmap
      static void draw_gc_window(HBITMAP fg, HBITMAP bg, int wl,
int wt, int wr, int wb)
      {
        // Chi modified 3-13-06: will slow it down, but
elliminates image discrepency overlap
            if (fg!= last_fg) // different bitmap in gcwindow
                  display_rect_bitmap(hwnd, fg, wl, wt, wl, wt,
wr, wb); // just fill in all of window
```

```
      /*

        if( wt>curr_wb || wb<curr_wt ||        // if no
intersection or window not drawn:

            wl>curr_wr || wr<curr_wl ||

            window_drawn==GCWINDOQW_INIT )

          {                                    // just fill in
all of window

            display_rect_bitmap(hwnd, fg, wl, wt, wl, wt, wr,
wb);

          }

      */




        else                  // Otherwise, fill in new regions

          {

            if(wt<curr_wt)    // draw any needed top

            display_rect_bitmap(hwnd, fg, wl, wt, wl, wt, wr,
curr_wt-1);



            if(wb>curr_wb)    // draw any needed bottom

            display_rect_bitmap(hwnd, fg, wl, curr_wb+1, wl,
curr_wb+1, wr, wb);



            if(wl<curr_wl)    // draw any needed left

            display_rect_bitmap(hwnd, fg, wl, MAX(wt, curr_wt),

                                                          wl,
MAX(wt, curr_wt),

                                                   curr_wl-1,
MIN(wb, curr_wb));

            if(wr>curr_wr)    // draw any needed right
```

```
            display_rect_bitmap(hwnd, fg, curr_wr+1, MAX(wt,
curr_wt),


curr_wr+1, MAX(wt, curr_wt),

                                                    wr, MIN(wb,
curr_wb));

          }



        last_fg = fg;

      }



             // Draw GC window at a new location

             // The first time window is drawn,

             // the background outside the window will be filled
in too.

             // If X or Y is MISSING_DATA (defined in
eyelink.h), window is hidden.

                  // determine which fgbm to use and call
draw_gc_window with

      /*void CALLTYPE redraw_gc_window(int x, int y)*/

      //Chi modified function: takes in param "type" for switch
statement

      void CALLTYPE redraw_gc_window(int x, int y, int type)

      {

            int wt, wl, wb, wr;

            //type = 1; //TEMP

            if(last_x==MISSING_DATA || x==MISSING_DATA) last_x =
x;    // record old position if valid

            else if(x < last_x-wdeadband) x = last_x =
x+wdeadband;    // deadband filter

            else if(x > last_x+wdeadband) x = last_x = x-
wdeadband;
```

```
            else x = last_x;


            if(last_y==MISSING_DATA  || y==MISSING_DATA) last_y
= y;   // same for Y coordinate

            else if(y < last_y-wdeadband) y = last_y =
y+wdeadband;

            else if(y > last_y+wdeadband) y = last_y = y-
wdeadband;

            else y = last_y;


            wt = (window_h<0) ? dr.left   : y - (window_h>>1);
// compute new window edges

            wb = (window_h<0) ? dr.bottom : wt + window_h - 1;

            wl = (window_w<0) ? dr.left   : x - (window_w>>1);

            wr = (window_w<0) ? dr.right  : wl + window_w - 1;


            GdiSetBatchLimit(50);  // allow ops to accumulate


            if(erase_first)    // draw background first

          {

                  if(window_drawn != GCWINDOQW_HIDDEN)

                        erase_gc_window(fgbm, bgbm, wl, wt, wr,
wb);


                  if(x==MISSING_DATA || y==MISSING_DATA)  //
window is not visible

              {

                        window_drawn = GCWINDOQW_HIDDEN;

                        return;

              }
```

```
                    //Chi modified: added switch statment, instead
of just draw_gc_window(fgbm...)

                switch(type)

                {

                    case 0:     // front object focus

                          draw_gc_window(fgbm1, fgbm, wl, wt, wr,
wb);  //switched bgbm w/ fgbm

                    break;

                    case 1:     // middle object focus

                          draw_gc_window(fgbm2, fgbm, wl, wt, wr,
wb);

                    break;

                    case 2:     // back object focus

                          draw_gc_window(fgbm3, fgbm, wl, wt, wr,
wb);

                    break;

                    case 3:     // all blur

                          draw_gc_window(fgbm, fgbm, wl, wt, wr,
wb);

                    break;

                    case 4:     // no blur

                          draw_gc_window(bgbm, bgbm, wl, wt, wr,
wb);

                    break;

                }

            }

              else              // draw foreground first

            {

                    if(x==MISSING_DATA || y==MISSING_DATA)   //
window is not visible
```

```
            {
                        if(window_drawn != GCWINDOQW_HIDDEN)
                              erase_gc_window(fgbm, bgbm, wl,
wt, wr, wb);
                        window_drawn = GCWINDOQW_HIDDEN;
                        return;
            }
          //Chi modified: added switch statment, instead of
just draw_gc_window(fgbm...)
            switch(type)
            {
                case 0:     // front object focus
                      draw_gc_window(fgbm1, fgbm, wl, wt, wr,
wb);
                break;
                case 1:     // middle object focus
                      draw_gc_window(fgbm2, fgbm, wl, wt, wr,
wb);
                break;
                case 2:     // back object focus
                      draw_gc_window(fgbm3, fgbm, wl, wt, wr,
wb);
                break;
                case 3:     // no focus
                      draw_gc_window(fgbm, fgbm, wl, wt, wr,
wb);
                break;
                case 4:     // no blur
                      draw_gc_window(bgbm, bgbm, wl, wt, wr,
wb);
```

```
                    break;
                }
                    if(window_drawn != GCWINDOQW_HIDDEN)
                          erase_gc_window(fgbm, bgbm, wl, wt, wr,
wb);
          }


             wait_for_drawing(NULL);   // make sure we are not
delayed
             GdiSetBatchLimit(1);      // restore immediate
drawing


             curr_wl = wl;   // record new window edges.
             curr_wr = wr;
             curr_wt = wt;
             curr_wb = wb;
             window_drawn = GCWINDOQW_VISIBLE;
       }
```

C. w32_gcwindow_trial.c

```
/*************************************************************
/* CONTENTS:
*/
/* - end_trial(): blank display, stop recording       */
/*  cleans up, gives Windows time          */
/* - run_trial(): performs actual trial sequence:       */
/*     - drift correction (possible recalibration)      */
/*     - start recording                                */
/*     - display stimulus to subject                    */
/*     - wait for data, select which eye's data to use  */
/*     - loop while drawing window from samples         */
/*     - wait for timeout, button press, or abort       */
/*     - send result message to EDF file                */
/*     - blank display, stop recording                  */
/*     - report any errors or reqest for repeat         */
 /*************************************************************/

#include <windows.h>
#include <windowsx.h>

#include "gdi_expt.h"
#include "w32_demo.h"  /* header file for this experiment
*/



/********* PERFORM AN EXPERIMENTAL TRIAL  *******/
```

```
      // End recording: adds 100 msec of data to catch final
events

      static void end_trial(void)

      {

            clear_full_screen_window(target_background_color);
/* hide display */

            end_realtime_mode();   // NEW: ensure we release
realtime lock

            pump_delay(100);       // CHANGED: allow Windows to
clean up

            // while we record additional 100 msec of data

            stop_recording();

            while(getkey()) {};

      }



      /* Run a single trial, recording to EDF file and sending
data through link */

      /* This example draws to a bitmap, then copies it to
display for fast stimulus onset */



      // The order of operations is:

      // - Set trial title, ID for analysis

      // - Draw foreground, background bitmaps and create
EyeLink display graphics

      // - Drift correction

      // - start recording

      // - <DON'T copy bitmap to display: draw window when first
sample arrives>

      // - loop till button press, timeout, or abort, drawing
gaze contingent window
```

```
      // - stop recording, dispose of bitmaps, handle abort and
exit


      // NEW CODE FOR GAZE CONTINGENT DISPLAY:

      // - create both foreground and background bitmaps

      // - uses the eyelink_newest_float_sample() to get latest
update

      // - initialize window with initialize_gc_window()

      // - moves window with redraw_gc_window()


      /* Chi added code (2-27-06)


        - Determine what virtual object is being attended upon
on the image

        from predefined coordinates

        - Change the fgbm (foreground image) accordingly via
switch conditional

        - THEN call redraw_gc_window

      */


      // Run gaze-contingent window trial

      // <fgbm> is bitmap to display within window

      // <bgbm> is bitmap to display outside window

      // <wwidth, wheight> is size of window in pixels

      // <mask> flags whether to treat window as a mask

      // <time_limit> is the maximum time the stimuli are
displayed
```

```
      //int gc_window_trial(HBITMAP fgbm, HBITMAP bgbm,

      //                    int wwidth, int wheight, int mask,
UINT32 time_limit)

      //Chi: modified w/ more params 3-08-06 & (4-12-06 add int
trialnum)

      int gc_window_trial(HBITMAP fgbm, HBITMAP bgbm,

                                    HBITMAP fgbm1, HBITMAP
fgbm2,

                                    HBITMAP fgbm3, int trialnum,
int wwidth,

                                    int wheight, int mask,
UINT32 time_limit)

      {

            UINT32 trial_start=0;   // trial start time (for
timeout)

            //  UINT32 drawing_time;  // retrace-to-draw delay
// Chi: commented out 2-27-06

            int button;       // the button pressed (0 if
timeout)

            int error;            // trial result code

            int type; //switch parameter for redraw (Chi add 3-
08-06)

            ALLF_DATA evt;         // buffer to hold sample and
event data

            int first_display = 1; // used to determine first
drawing of display

            int eye_used = 0;      // indicates which eye's data
to display

            float x, y;        // gaze position


      // Chi 4-18-06

            /*
```

```
                Over compensate which object attended to by a
factor of
                "adjust" pixels based on edge detection
            */
            const int adjust = 20;



            /*Chi 2-27-06 */
            //coordinate limits for viewing objects
            // see diagram "variable_notes" for details
            // these values are Dependent on the Image used


       int a,b, a_prime, b_prime, c, d, c_prime, d_prime, e, f,
e_prime, f_prime;


            // NOTE: TRIALID AND TITLE MUST HAVE BEEN SET BEFORE
DRIFT CORRECTION!
          // FAILURE TO INCLUDE THESE MAY CAUSE
INCOMPATIBILITIES WITH ANALYSIS SOFTWARE!



            // Set size and type of gaze-contingent window
            RECT display_rect;


            display_rect.top    = dispinfo.top;
            display_rect.bottom = dispinfo.bottom;
            display_rect.left   = dispinfo.left;
            display_rect.right  = dispinfo.right;
```

```
//Chi 2-14-06
//coordinate limits for viewing objects
// see diagram "variable_notes" for details
// these values are Dependent on the Image used
switch(trialnum)
{
      case 0: // Trial A
            a=23; b=512;
            a_prime = 250; b_prime = 315;
            c=308; d=509;
            c_prime = 761; d_prime = 240;
            e=154; f=511;
            e_prime = 609; f_prime = 87;
            break;

      case 1:     //Trial B
            a=148; b=507;
            a_prime = 416; b_prime = 258;
            c=281; d=506;
            c_prime = 788; d_prime = 248;
            e=40; f=507;
            e_prime = 721; f_prime = 135;
      break;
      case 2: //Trial C
            a=332; b=504;
            a_prime = 435; b_prime = 400;
            c=201; d=504;
```

```
                c_prime = 561; d_prime = 318;
                e=191; f=506;
                e_prime = 573; f_prime = 136;
            break;
            case 3: //Trial D
                a=136; b=530;
                a_prime = 633; b_prime = 376;
                c=164; d=377;
                c_prime = 600; d_prime = 303;
                e=133; f=455;
                e_prime = 628; f_prime = 186;
            break;
            case 4: //Trial A no DOF
            break;
            case 5: //Trial B no DOF
            break;
            case 6: //Trial C no DOF
            break;
            case 7: //Trial D no DOF
            break;

        };


        //Chi modified more params 3-08-06
    initialize_gc_window(300, 300, fgbm, bgbm,
                                    fgbm1, fgbm2, fgbm3,
```

```
                full_screen_window, display_rect,
                mask, SCRWIDTH/300);  // sets 0.1 degree
deadband



      /*

            initialize_gc_window(wwidth, wheight, fgbm, bgbm,
                  full_screen_window, display_rect,
                  mask, SCRWIDTH/300);  // sets 0.1 degree
deadband
      */
               // DO PRE-TRIAL DRIFT CORRECTION
               // We repeat if ESC key pressed to do setup.
            while(1)
          {              // Check link often so we can exit if
tracker stopped
                if(!eyelink_is_connected()) return ABORT_EXPT;
                // We let do_drift_correct() draw target in
this example
                // 3rd argument would be 0 if we already drew
the display
                error = do_drift_correct(SCRWIDTH/2,
SCRHEIGHT/2, 1, 1);
                // repeat if ESC was pressed to access Setup
menu
                if(error!=27) break;
          }


            clear_full_screen_window(target_background_color);
// make sure display is blank
```

```
            // Start data recording to EDF file, BEFORE
DISPLAYING STIMULUS

            // You should always start recording 50-100 msec
before required

            // otherwise you may lose a few msec of data



            //  tell start_recording() to send link data

            error = start_recording(1,1,1,1);   // record with
link data enabled

            if(error != 0) return error;          // ERROR:
couldn't start recording

            // record for 100 msec before displaying stimulus

            begin_realtime_mode(100);   // Windows 2000/XP: no
interruptions from now on



          // DONT DISPLAY OUR IMAGES TO SUBJECT until we have
first gaze postion!



            if(!eyelink_wait_for_block_start(100, 1, 0))  //
wait for link sample data

          {

                  end_trial();

                  alert_printf("ERROR: No link samples
received!");

                  return TRIAL_ERROR;

          }

            eye_used = eyelink_eye_available(); // determine
which eye(s) are available

            switch(eye_used)              // select eye, add
annotation to EDF file

          {
```

```
            case RIGHT_EYE:
              eyemsg_printf("EYE_USED 1 RIGHT");
              break;
            case BINOCULAR:           // both eye's data
present: use left eye only
              eye_used = LEFT_EYE;
            case LEFT_EYE:
              eyemsg_printf("EYE_USED 0 LEFT");
              break;
          }
            // Now get ready for trial loop
            eyelink_flush_keybuttons(0);   // reset keys and
buttons from tracker
            // we don't use getkey() especially in a time-
critical trial
            // as Windows may interrupt us and cause an
unpredicatable delay
            // so we would use buttons or tracker keys only


            // Trial loop: till timeout or response -- added
code for reading samples and moving cursor
            while(1)
          {                           // First, check if
recording aborted
                if((error=check_recording())!=0) return error;
                // Check if trial time limit expired


                /* Chi 2-27-06: Removed/commented-out time-
limit condition*/
```

```
                /*
                if(current_time() > trial_start+time_limit &&
trial_start!=0)
             {
                eyemsg_printf("TIMEOUT");    // message to log
the timeout
                end_trial();                 // local function
to stop recording
                button = 0;                  // trial result
message is 0 if timeout
                break;                       // exit trial
loop
                }
                */



                if(break_pressed())     // check for program
termination or ALT-F4 or CTRL-C keys
                {
                      end_trial();         // local function
to stop recording
                      return ABORT_EXPT;   // return this code
to terminate experiment
                }


                if(escape_pressed())    // check for local ESC
key to abort trial (useful in debugging)
                {
                      end_trial();         // local function
to stop recording
                      return SKIP_TRIAL;   // return this code
if trial terminated
```

```
                }


                /* BUTTON RESPONSE TEST */

                // Check for eye-tracker buttons pressed

                // This is the preferred way to get response
data or end trials

                button = eyelink_last_button_press(NULL);

                if(button!=0)       // button number, or 0 if
none pressed

                {

                      eyemsg_printf("ENDBUTTON %d", button);
// message to log the button press

                      end_trial();
// local function to stop recording

                      break;
// exit trial loop

                }


                // NEW CODE FOR GAZE CONTINGENT WINDOW


                if(eyelink_newest_float_sample(NULL)>0)  //
check for new sample update

                {

                      eyelink_newest_float_sample(&evt);   //
get the sample

                      x = evt.fs.gx[eye_used];    // yes: get
gaze position from sample

                      y = evt.fs.gy[eye_used];

                      if(x!=MISSING_DATA &&

                            y!=MISSING_DATA &&
```

```
                              evt.fs.pa[eye_used]>0)     // make
sure pupil is present

                        {


                              /* Chi 2-27-06: deleted marking of
diplay start because not needed*/

                              // Chi 4-18-06: over compensate by
adjust pixels for larger edge boundaries

                              //Chi - Determine what virtual
object is being attended upon on the image

                              if (trialnum >3)

                                    type =4; // no DOF effect

                              else {

                                    if ((x >= a-adjust && x<=
a_prime+adjust) && (y<= b+adjust && y>= b_prime-adjust))


                                    {

                                          // 1st (most frontal)
object (RED)

                                          type =0;



                                    }

                                    else if ((x >= c-adjust &&
x<= c_prime+adjust) && (y<= d+adjust && y>= d_prime-adjust))

                                    {

                                          //middle object
(GREEN)

                                          type=1;

                                    }

                                    else if ((x >= e-adjust &&
x<= e_prime+adjust) && (y<= f+adjust && y>= f_prime-adjust))

                                    {
```

```
                                        //last object (BLUE)
                                        type=2;
                                    }
                                    else
                                    {
                                        // looking at infinite
space (BLACK)
                                        type=3;
                                    }
                                }


                                target_foreground_color =
RGB(0,0,0);         // color of calibration target
                                target_background_color =
RGB(128,128,128);   // background for drift correction


      set_calibration_colors(target_foreground_color,
target_background_color); // tell EXPTSPPT the colors



                                //
clear_full_screen_window(target_background_color);



                                redraw_gc_window(x, y, type);  //
Chi modified w/ extra param 3-08-06
                                //redraw_gc_window(x, y);  // move
window if visible



                                /* Chi 2-27-06: deleted marking of
diplay start because not needed*/


                            }
```

```
                        else
                        {
                              // Don't move window during blink
                              // To hide window, use:
redraw_gc_window(MISSING, MISSING);
                        }
                  }
            }                       // END OF RECORDING LOOP


            end_realtime_mode();      // safety cleanup code
            while(getkey());          // dump any accumulated
key presses


            // report response result: 0=timeout, else button
number
            eyemsg_printf("TRIAL_RESULT %d", button);
            // Call this at the end of the trial, to handle
special conditions
            return check_record_exit();
      }
```

D. w32_gcwindox_trials.c

```
/*****************************************************/
/* CONTENTS:                                         */
/* - run_trials() loops through trials, and handles  */
/*   aborted/repeated trial requests                 */
/* - do_text_trial() interperts trial number         */
/*   and supplies appropriate TRIALID label and      */
/*   creates a stimulus bitmap for each trial        */
/* - add end_realtime_mode() to clean up trials      */
/* - use image_file_bitmap() to load images          */
/*****************************************************/

#include <windows.h>
#include <windowsx.h>

#include <stdlib.h> // Chi add 4-12-06 for randomizer
#include <time.h>   // Chi add 4-12-06 for randomizer

#include "gdi_expt.h"
#include "w32_demo.h"  /* header file for this experiment
*/

/*Chi 2-27-06: removed text-bitmap options*/

/********* PREPARE BITMAPS FOR TRIALS  **********/

HBITMAP fgbm=NULL, bgbm=NULL;
```

```
      /*Chi code: add extra HBITMAP handlers,

      to load in additional images on prog start*/


      HBITMAP fgbm1=NULL, fgbm2=NULL, fgbm3=NULL;

      // fgbm1 : front object in focus,

      // fgbm2 : middle object in focus,

      // fgbm3 : last object in focus


               // Create foreground and background bitmaps of
picture




      static int create_image_bitmaps(int type)

      {

        target_foreground_color = RGB(0,0,0);         // color
of calibration target

        target_background_color = RGB(128,128,128);   //
background for drift correction

        set_calibration_colors(target_foreground_color,
target_background_color); // tell EXPTSPPT the colors



        clear_full_screen_window(target_background_color);



        get_new_font("Arial", 24, 1);

        graphic_printf(target_foreground_color, -1, 1,
SCRWIDTH/2, SCRHEIGHT/2, "Loading image...");

      /* Chi 2-27-06 (modified switch cases -- foreground image
changes file used, no blank fovea used*/

        switch(type)
```

```
          {

            case 0:   // trial A

                  fgbm = image_file_bitmap("images/SetUpA-
infi.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                  fgbm1 = image_file_bitmap("images/SetUpA-
red.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                  fgbm2 = image_file_bitmap("images/SetUpA-
green.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

               fgbm3 = image_file_bitmap("images/SetUpA-
blue.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

               bgbm =  image_file_bitmap("images/SetUpA-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                  eyecmd_printf("record_status_message 'TRIAL A
- Case 0' ");

               break;

             case 1:     // trial B

                  fgbm = image_file_bitmap("images/SetUpB-
infi.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                  fgbm1 = image_file_bitmap("images/SetUpB-
red.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                  fgbm2 = image_file_bitmap("images/SetUpB-
green.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

               fgbm3 = image_file_bitmap("images/SetUpB-
blue.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

               bgbm =  image_file_bitmap("images/SetUpB-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                  eyecmd_printf("record_status_message 'TRIAL B
- Case 1' ");

               break;

             case 2:     // trial C

               fgbm = image_file_bitmap("images/SetUpC-infi.bmp",
0, SCRWIDTH,SCRHEIGHT,target_background_color);

                  fgbm1 = image_file_bitmap("images/SetUpC-
red.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);
```

```
                fgbm2 = image_file_bitmap("images/SetUpC-
green.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

             fgbm3 = image_file_bitmap("images/SetUpC-
blue.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

             bgbm =  image_file_bitmap("images/SetUpC-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                eyecmd_printf("record_status_message 'TRIAL C
- Case 2' ");

             break;

             case 3:     // trial D

             fgbm = image_file_bitmap("images/SetUpD-infi.bmp",
0, SCRWIDTH,SCRHEIGHT,target_background_color);

                fgbm1 = image_file_bitmap("images/SetUpD-
red.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                fgbm2 = image_file_bitmap("images/SetUpD-
green.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

             fgbm3 = image_file_bitmap("images/SetUpD-
blue.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

             bgbm =  image_file_bitmap("images/SetUpD-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                eyecmd_printf("record_status_message 'TRIAL D
- Case 3' ");

             break;



             case 4:   // trial A  No DOF

                fgbm = image_file_bitmap("images/SetUpA-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                fgbm1 = image_file_bitmap("images/SetUpA-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                fgbm2 = image_file_bitmap("images/SetUpA-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

             fgbm3 = image_file_bitmap("images/SetUpA-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);
```

```
              bgbm =  image_file_bitmap("images/SetUpA-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                   eyecmd_printf("record_status_message 'TRIAL
A no DOF - Case 4' ");

                  break;



              case 5:   // trial B  No DOF

                   fgbm = image_file_bitmap("images/SetUpB-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                  fgbm1 = image_file_bitmap("images/SetUpB-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                  fgbm2 = image_file_bitmap("images/SetUpB-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

               fgbm3 = image_file_bitmap("images/SetUpB-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

              bgbm =  image_file_bitmap("images/SetUpB-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                   eyecmd_printf("record_status_message 'TRIAL
B no DOF - Case 5' ");

                  break;



              case 6:   // trial C  No DOF

                    fgbm = image_file_bitmap("images/SetUpC-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                  fgbm1 = image_file_bitmap("images/SetUpC-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                  fgbm2 = image_file_bitmap("images/SetUpC-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

               fgbm3 = image_file_bitmap("images/SetUpC-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

              bgbm =  image_file_bitmap("images/SetUpC-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                   eyecmd_printf("record_status_message 'TRIAL
C no DOF - Case 6' ");
```

```
                    break;


              case 7:   // trial D    No DOF

                    fgbm = image_file_bitmap("images/SetUpD-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                    fgbm1 = image_file_bitmap("images/SetUpD-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                    fgbm2 = image_file_bitmap("images/SetUpD-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                 fgbm3 = image_file_bitmap("images/SetUpD-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                bgbm =  image_file_bitmap("images/SetUpD-no-
dof.bmp", 0, SCRWIDTH,SCRHEIGHT,target_background_color);

                     eyecmd_printf("record_status_message 'TRIAL
D no DOF - Case 7' ");

                    break;



           }



        eyecmd_printf("clear_screen 0");                    //
clear EyeLink display

        eyecmd_printf("draw_box %d %d %d %d 15",          // Box
around fixation point

                   SCRWIDTH/2-16, SCRHEIGHT/2-16, SCRWIDTH/2+16,
SCRHEIGHT/2+16);



        if(!fgbm || !bgbm || !fgbm1 || !fgbm2 || !fgbm3)  //
Check that both bitmaps exist. Chi Modified 3-08-06

          {

            eyemsg_printf("ERROR: could not load image");

            alert_printf("ERROR: could not load an image file");

            if(fgbm) DeleteObject(fgbm);
```

```
            if(bgbm) DeleteObject(bgbm);
              if(fgbm1) DeleteObject(fgbm1); //Chi Modified 3-
08-06
              if(fgbm2) DeleteObject(fgbm2); //Chi Modified 3-
08-06
              if(fgbm3) DeleteObject(fgbm3); //Chi Modified 3-
08-06
            return SKIP_TRIAL;
          }
        return 0;
      }



      /*********** TRIAL SELECTOR **********/


      #define NTRIALS 8  // 8 trials
      int TrialsToRun[NTRIALS]; // keep track of which trial
finished (Chi add 4-12-06):


               // FOR EACH TRIAL:
               // - set title, TRIALID
               // - Create bitmaps and EyeLink display graphics
               // - Check for errors in creating bitmaps
               // - Run the trial recording loop
               // - Delete bitmaps
               // - Return any error code


               // Given trial number, execute trials
               // Returns trial result code
```

```
int do_gcwindow_trial(int num)

{

      int i;



      if (num>=0  && num<NTRIALS)

      {

            set_offline_mode();                // Must be
offline to draw to EyeLink screen



        eyecmd_printf("record_status_message 'GC IMAGE
MAYA DOF SWAP' "); //Chi's new message

        eyemsg_printf("TRIALID GCTXTB");



            // TRIAL_VAR_DATA message is recorded for
EyeLink Data Viewer analysis

            // It specifies the list of trial variables
value for the trial

            // This must be specified within the scope of
an individual trial (i.e., after

            // "TRIALID" and before "TRIAL_RESULT")


            eyemsg_printf("!V TRIAL_VAR_DATA IMAGE IMAGE
BLURRED");



        if(create_image_bitmaps(num))

            {

            eyemsg_printf("ERROR: could not create
bitmap");

            return SKIP_TRIAL;

            }
```

```
                // IMGLOAD command is recorded for EyeLink
Data Viewer analysis

                // It displays a default image on the overlay
mode of the trial viewer screen.

                // Writes the image filename + path info

                eyemsg_printf("!V IMGLOAD FILL
images/depth_of_field1.bmp");



                // Transfer the bitmap to tracker PC as
backdrop for gaze cursors

                bitmap_to_backdrop(fgbm, 0, 0, 0, 0,

                      0, 0,
BX_MAXCONTRAST|(is_eyelink2?0:BX_GRAYSCALE));



             //i = gc_window_trial(fgbm, bgbm, SCRWIDTH/4,
SCRHEIGHT/3, 0, 60000L);  // Gaze-contingent window, masked
image



                i = gc_window_trial(fgbm, bgbm, fgbm1, fgbm2,
fgbm3, num, SCRWIDTH/4, SCRHEIGHT/3, 0, 60000L);  //Chi modified
3-08-06



                DeleteObject(fgbm);

            DeleteObject(bgbm);

                DeleteObject(fgbm1); //Chi added 3-08-06

                DeleteObject(fgbm2); //Chi added 3-08-06

                DeleteObject(fgbm3); //Chi added 3-08-06

            return i;

          }

            else

                return ABORT_EXPT;  // illegal trial number
```

```
}


/*********** TRIAL LOOP **************/


      /* This code sequences trials within a block */
      /* It calls run_trial() to execute a trial, */
      /* then interperts result code. */
      /* It places a result message in the EDF file */
      /* This example allows trials to be repeated */
      /* from the tracker ABORT menu. */
int run_trials(void)
{
  int i;
  int ntrial, trial;
  srand( (unsigned) time(NULL) ); // seed randomizer
// initialize values for TrialsToRun (Chi 4-12-06)
for (i=0; i< NTRIALS;i++)
{
      TrialsToRun[i]= 0; // have not run trial yet
}


                    // INITIAL CALIBRATION: matches following
trials
    target_foreground_color = RGB(0,0,0);         // color
of calibration target
```

```
        target_background_color = RGB(200,200,200);   //
background for drift correction

        set_calibration_colors(target_foreground_color,
target_background_color); // tell EXPTSPPT the colors



        // TRIAL_VAR_LABELS message is recorded for EyeLink Data
Viewer analysis

        // It specifies the list of trial variables for the
trial

        // This should be written once only and put before the
recording of individual trials

        eyemsg_printf("TRIAL_VAR_LABELS TYPE CENTRAL
PERIPHERAL");



        if(SCRWIDTH!=800 || SCRHEIGHT!=600)

          alert_printf("Display mode is not 800x600, resizing
will slow loading.");



                  /* PERFORM CAMERA SETUP, CALIBRATION */

      //  do_tracker_setup();

                              /* loop through trials */

        for(ntrial=1;ntrial<=NTRIALS;ntrial++)

          {

            if(eyelink_is_connected()==0 || break_pressed())
/* drop out if link closed */

            {

              return ABORT_EXPT;

            }

                              /* RUN THE TRIAL */



              trial = rand() % NTRIALS;
```

```
              while (TrialsToRun[trial]) // choose a trial that
has not been run yet

              {

                  if (trial==NTRIALS-1)

                        trial=0;

                  else

                        trial++;

              }

              TrialsToRun[trial] = 1;


            i = do_gcwindow_trial(trial);

            end_realtime_mode();


            switch(i)               /* REPORT ANY ERRORS */

            {

              case ABORT_EXPT:        /* handle experiment abort
or disconnect */

                eyemsg_printf("EXPERIMENT ABORTED");

                return ABORT_EXPT;

              case REPEAT_TRIAL:      /* trial restart requested
*/

                eyemsg_printf("TRIAL REPEATED");

                trial--;

                break;

              case SKIP_TRIAL:        /* skip trial */

                eyemsg_printf("TRIAL ABORTED");

                break;

              case TRIAL_OK:          // successful trial
```

```
          eyemsg_printf("TRIAL OK");
          break;
        default:                // other error code
          eyemsg_printf("TRIAL ERROR");
          break;
      }
    }  // END OF TRIAL LOOP
  return 0;
}
```