Chapter 1

Introduction

It was Euclid and Leonardo who were the first to understand that the human vision system is able to view different perspectives from each eye. In 1838, Charles Wheatstone, through his discoveries, formally announced that the human visual system is capable of producing a unique sense of "depth" called *stereopsis* or *solid-seeing* [SH97]. Stereopsis is a phenomenon perceived by the human mind. The mind is able to fuse two retinal images into one and therefore perceives the image pair as one, not two.

Fundamentally, a stereoscopic image pair presents to the left and right eyes of the viewer with viewpoints of marginally differing perspectives. From these dissimilar viewpoints, the viewer's visual system synthesizes an image with stereoscopic depth. In other words, the viewer will perceive the image pair to be "popping out of" or "receding into" the plane of camera projection.

1.1 Stereoscopic display systems

As stated in [SH97], “A stereoscopic display system is an optical system whose final component is the human mind”. Such a system presents to each eye similar images with different perspectives.

A simple and commonly available example of a stereoscopic display system is the View-Master, as shown in Figure 1-1. The View-Master is an example of a *plano-stereoscopic* [SH97] display system that uses two planar images as its stereoscopic image pair.

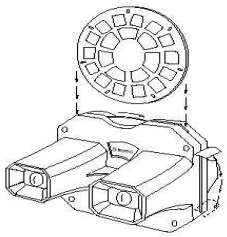


Figure 1-1. View-Master

It functions by presenting a stereoscopic image pair to the viewer. When this stereoscopic device is viewed with only one eye, the image will look flat and planar. However, when viewed with both eyes, the image will appear to have stereoscopic depth.

A stereoscopic display system like the View-Master is considered a mechanical stereoscopic system. An *electro-stereoscopic* or *field-sequential* display system, on the other hand, is an example of an electronic plano-stereosopic system that uses the CRT (Cathode Ray Tube) for rendering and display.

A field-sequential stereoscopic display configuration uses a *time-multiplexing* [B97] encoding scheme to display alternating left and right images. When viewed directly such display systems produce images that are look overlaid and super-imposed, an effect similar to pictures that have been double-exposed. In order to correctly view such systems, a *de-multiplexing* device such as a shutter-type eyewear is required to occlude the unwanted and transmit the wanted images in a synchronized manner. The eyewear shutters in synchronization with the CRT display refresh to let the left eye receive the left image and the right receive the right image.

1.2 StereoGraphics StereoEyes

Figure 1-2. StereoGraphics StereoEyes

StereoGraphics StereoEyes, as shown in Figure 1-2, is an electro-stereoscopic visualization system that allows the user to view stereoscopic images on a personal computer. The StereoEyes system comprises of the following devices: StereoEyes shutter eyewear, infrared emitter, and StereoEnabler.

The StereoEyes eyewear functions by shuttering or occluding the left and right LCD lenses at a rate of about 60-72 Hz [S03] per eye, fast enough that there is no perceptible flicker. The shutter occludes each eye in an alternating fashion such that at any time only one eye is able to see the transmitted images.

The StereoEnabler unit works by intercepting the video sync signal and relaying an electronic signal to the infrared emitter to broadcast an infrared beam upon receiving the prescribed blue line signal from the frame buffer. The StereoEyes eyewear receives this infrared signal and synchronizes its shutters to the video sync rate.

The exact setup and interaction between the different components of StereoEyes is shown below in Figure 1-3.

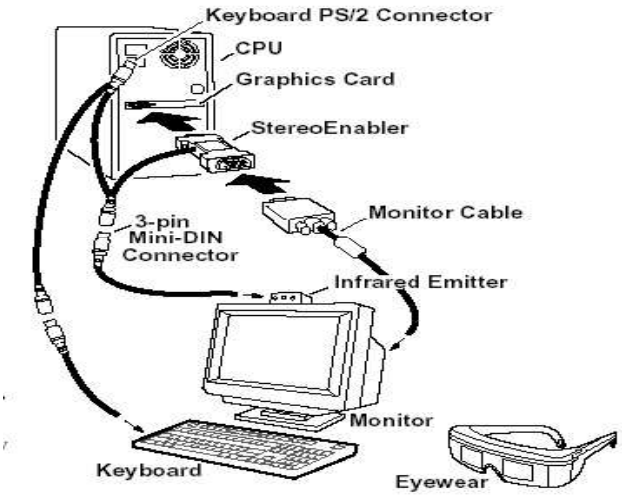


Figure 1-3. StereoEyes setup

1.3 Depth Cues

- Monocular Cue

  A monocular depth cue is also known as an *extra-stereoscopic* [SH97] depth cue. Monocular depth cues form the basis for visual sensory perception in normal non-stereoscopic viewing. Such depth cues include: *interposition*, *relative sizes*, *lighting and shading*, and *perspective*.

  Perspective is the single most important characteristic among the monocular depth cues since it incorporates *perspective foreshortening*. Perspective foreshortening models how real viewing occurs in the eyes of the viewer. The relative size of the perspective projection of an object varies inversely with the distance from the viewer.

  Also, images that are rich in monocular depth cues can be easier to view when stereoscopic cues are added.

- Binocular Cue

  A binocular depth cue such as stereopsis is a unique visual sensory perception brought on by having a *congruent* stereoscopic image pair called *fields*. To be congruent, the left and right image fields must be identical in every aspect except for the horizontal offset displacement.

An interesting side effect of stereopsis is that when a three-dimensional image with monocular perspective is viewed with only one eye, it will still look three-dimensional. However, when a stereoscopic image pair with perspective is viewed with only one eye instead of two, it would lose its "essence" of depth.

1.4 Visual sensory perception

- *Interaxial Separation*

  Interaxial separation is also termed *interocular separation*. This separation is the horizontal distance between the viewer's eyes that forms the centers of projection. In general, the stereoscopic strength of a projection is directly proportional to the interaxial separation. Thus an increase in the interaxial separation will cause a corresponding increase in the stereoscopic strength of an image.

- *Retinal Disparity*

  Retinal disparity refers to the horizontal distance or displacement between corresponding left and right image points that form the projections unto the human retina. The corresponding projected image points are called *homologous* or *conjugate* image points. Retinal disparity is induced by the fact that the eyes of an average adult are approximately two and a half inches apart [SH97, A99].

- *Accommodation*

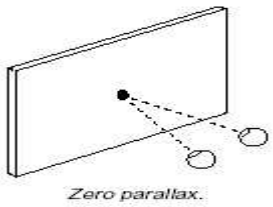
Accommodation refers to the natural physiological response of a viewer's vision system to cause the lens in the eyes to change its concavity thus allowing focus. Accommodation implies the focusing of the lens.

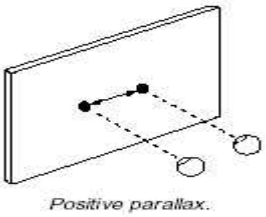
- *Convergence*

  Convergence is also a natural physiological response. It causes the viewer's eye to rotate the axis of the lens away or toward each other such that the center of attraction can be changed. This center of attraction point can be altered such that it can be brought closer towards or farther away from the viewer.

- *Parallax*

  Parallax can get classified into the following three main categories: *zero*, *positive*, and *negative* as shown in Figure 1-4.





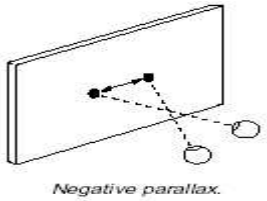


Figure 1-4. Parallax

For zero parallax, homologous or conjugate image points of the two projected points correspond or coincide exactly with each other. In this scenario, the viewer will perceive the projected point as lying flat on the plane of projection.

For *convergent* positive parallax, the plane of convergence or the center of attraction would appear to be behind the plane of projection. In other words, the viewer's eyes would converge at a point behind the projection plane. Positive parallax forms an optical illusion that causes the projected image to manifest behind or recede into the plane of projection. In an electro-stereoscopic display system, positive parallax forms projections that exist within the *CRT space*. It is conceivable to have *divergent* positive parallax such that the axes of the eyes diverge, but such effects are not natural and may not be physiologically possible.

In negative parallax, the plane of convergence would appear to be in front of the plane of projection. In other words, the viewer's eyes would converge at a point in front of the projection plane. In this case, negative parallax forms an optical illusion that causes the projected image to manifest in front of or pop out of the

plane of projection. In an electro-stereoscopic display system, negative parallax forms projections that exist within the *viewer space*.

1.5 Possible issues

- Parallax

  In terms of viewing comfort, positive parallax is gentler to view compared to negative. However, in terms of stereopsis, the effects of negative parallax are more spectacular compared to positive parallax.

  Parallax and retinal disparity are fundamentally similar entities. Parallax is usually measured at the display screen while retinal disparity is measured at the retina. When viewing electro-stereoscopic display systems using shutter glasses, parallax coincides with retinal disparity. In other words, shutter glasses cause parallax to become retinal disparity, and retinal disparity in turn induces stereopsis.

- Screen Surround

  In an electro-stereoscopic display system, the region outside of the viewing area of the CRT is called the *surround*. The surround consists of four edges that form the clipping region within the display view port. When objects are projected such that they overlap or clip the surround, conflicting visual cues are raised that

directly contradict the parallax effects. Hence, it is generally a good idea to render images in a consistent manner such that the final image composition and the overall stereoscopic impression are not unduly affected by the screen-surround.

- Accommodation / Convergence

  Under normal viewing, accommodation and convergence are closely related. In fact, the accommodation and convergence relationship is a habitual response in the vision system of a normal human. When viewing an object, the eyes accommodate to the object and converge to it such that the object can be seen as one.

  When electro-stereoscopic display systems are used, this accommodation and convergence correlation is broken. The eyes accommodate on the plane of the projection but converge based on parallax. As detailed in [SH97], such breakdown of a habitual response may cause discomfort, eyestrain and headache for some viewers.

  Also, as mentioned in [SH97], most theme parks with stereoscopic motion pictures use exaggerated parallax effects with large displays viewed over great distances. Such an arrangement reduces the breakdown in accommodation and convergence and thus stereoscopic viewing can be done much more comfortably.

Finally, the viewing of stereoscopic images is described as a learned or conditioned response [SH97] that requires training and gradual exposure until the viewer is comfortable with the effects caused by the breakdown of accommodation and convergence.

1.6 Stereo display formats

As detailed in [L97], an electro-stereoscopic format is formally defined as: "a method used for assigning pixels (or aggregates of pixels – lines or fields) to respective left and right images, thus making them available at the display screen, to the eyes of the observer, as an image with binocular stereopsis."

For displaying output in an electro-stereoscopic display system, one of the four types of field sequential stereoscopic display formats [L97] must be used. They are: *Interlace*, *Above-and-Below*, *Side-by-Side*, and *White-Line-Code*. The White-Line-Code format is commonly referred to as *Blue-Line-Code* in the technical literature from StereoGraphics.

There is also a non-field sequential stereoscopic display format called *anaglyph* or *red-blue*. Images displayed in anaglyph mode have a characteristic appearance of red and blue composite blending. Anaglyph images are usually rendered in such a manner that one image is polarized with one color while the other image is polarized with another. Thus to view such images, a de-polarizing filter is required. The red-blue display format also has the disadvantage of being band-limited to particular color ranges outside of the

polarizing filters. Also, the anaglyph format can be described as a “poor-man’s” stereoscopic imaging system since the only hardware that is required is a pair of inexpensive red-blue polarizing glasses.

This project is based on the White-Line-Code or Blue-Line-Code stereoscopic encoding format. The BLC format is commonly used on multimedia personal computers and workstations. It offers a high-quality but low cost stereoscopic solution that is within easy access for software professionals, scientists, and hobbyists alike.

The BLC format specification, as shown in Figure 1-5, is mandated as follows: On the bottom of every field, a thin blue rectangle (minimum three pixels tall) is tagged onto the respective fields in frame buffer memory to denote the left or right views. In order to properly encode the respective fields, this convention must be closely followed.

- For the left eye view – The bottom row pixels of the frame buffer must be blue for 75% of the width of view port and black for the remaining 25%
- For the right eye view – The bottom row pixels of the frame buffer must be blue for 25% of the width the view port and black for the remaining 75%

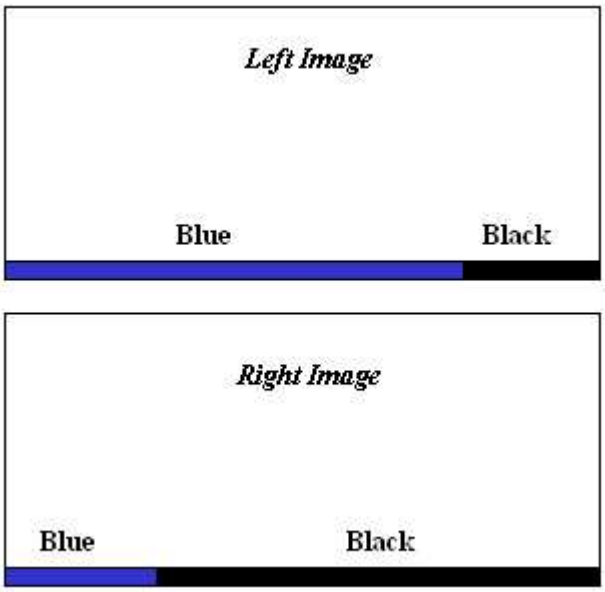


Figure 1-5. Blue Line Code

1.7 Stereo display problems

- *Crosstalk*

  Crosstalk refers to a common class of problems in an electro-stereoscopic display system where one eye receives a perspective view that it is not supposed to be seeing. An ideal shutter at its optimal operating frequency would occlude and transmit only the designated set of images. However, due to hardware imperfections [L87], such a strict isolation of channels may sometimes not be possible. The result of which is a visual perception called *ghosting*. Ghosting causes the viewer to perceive double or superimposed images.

As detailed in [L87, WT02], crosstalk can be attributed to the following two causes: *CRT phosphor decay* and *shutter leakage*. Phosphor decay or phosphor afterglow causes an image to persist such that a faint residual of the original image is left even when a subsequent image is displayed on the CRT. Shutter leakage, on the other hand, is due to the physical limitations of liquid crystal technology where the shutter does not become 100% opaque. So if the variation in contrast, color or brightness of the display image is large, then the occluded eye may receive part of the image that is not intended for it.

- Flicker

  Flicker occurs whenever the refresh rate of the shutter glasses falls below a threshold frequency of approximately 60 Hz. The CRT refresh rate is closely correlated with the shutter refresh rate. Therefore, any expensive graphical operations that will stall the hardware graphics pipeline will invariably lead to a reduction in the frame refresh rate. The resulting reduction in the CRT refresh rate will lead to lower shuttering frequency. As a result, when the shuttering frequency drops below the shutter's optimal operating frequency, the consequent visual artifact would be disorienting and uncomfortable flicker. Thus, any reduction in the shuttering frequency will eventually lead to flicker.

Chapter 2

Stereoscopic Camera Model

In this chapter, we are going to examine the stereoscopic camera model and the specifics of its use in creating stereoscopic software.

2.1 Perspective projection

As detailed in [A99], for a right-handed coordinate system such as Figure 2-1a, the perspective projection can be mathematically described by the following equation:

$$\left(\frac{xd}{d-z} , \frac{yd}{d-z} \right)$$

(x, y, z) refers to a coordinate on the image to be projected onto the *xy*-plane of projection. The camera or center of projection is positioned on the positive z-axis at (0, 0, *d*) as shown in Figure 2-1a, where *d* refers to the distance from the camera to the *xy*-plane.

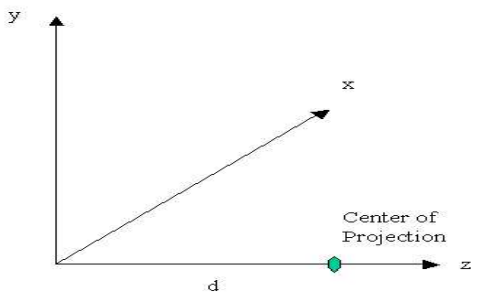


Figure 2-1a. Right-handed Coordinate System

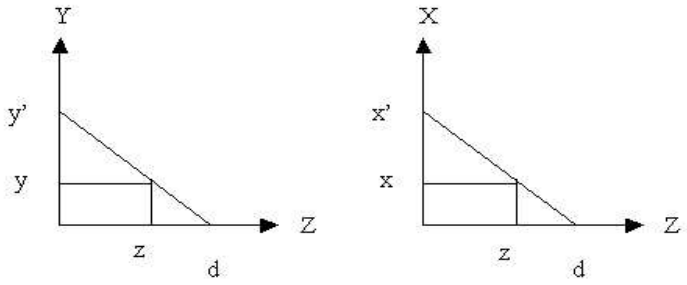


Figure 2-1b. YZ and XZ planes

As shown in Figure 2-1b, the projected point *y'* can be derived as follows:

$$\frac{y'}{d} = \frac{y}{d-z} \Rightarrow y' = \frac{yd}{d-z}$$

Likewise, *x'* follows as:

$$\frac{x'}{d} = \frac{x}{d-z} \Rightarrow x' = \frac{xd}{d-z}$$

2.2 Stereoscopic perspective projection

In order to accurately and correctly model stereoscopic vision, the above equation has to be modified. In this case, we denote the interocular or interaxial separation as *c*. As derived in [A99], to model a left camera perspective, the camera is offset to the left by

half of the interocular distance. Using the principle of model-view duality, the equation can be simplified by shifting the entire scene to the right instead of moving the camera to the left. So the perspective projection for the left eye view can be described by the following equation:

$$\left(\frac{\left(x + \frac{c}{2} \right) d}{d - z} , \frac{yd}{d - z} \right)$$

Likewise, to model a right camera perspective, the camera is offset to the right by half of the interocular distance. As a simplification, the entire scene is similarly shifted to the left instead of moving the camera to the right. So the perspective projection for the right eye view can be described by the following equation:

$$\left(\frac{\left(x - \frac{c}{2} \right) d}{d - z} , \frac{yd}{d - z} \right)$$

As will be explained shortly in Section 2.4, the above two equations introduce negative parallax. The rule of thumb when creating a balanced and pleasing stereoscopic image is a balanced overall combination of negative, zero, and positive parallax. Thus the left and right eye equations have to be modified.

2.3 *Parallel Axis Asymmetric Frustum Perspective Projection*

As detailed in [A99], to correctly produce balanced parallax, the projected elements are post-projection shifted to the left for left camera projections and right for right camera projections. So the new equation for the left eye projections would be:

$$\left(\frac{\left(x + \frac{c}{2} \right) d}{d - z} - \frac{c}{2}, \frac{yd}{d - z} \right)$$

And the right eye projections would be:

$$\left(\frac{\left(x - \frac{c}{2} \right) d}{d - z} + \frac{c}{2}, \frac{yd}{d - z} \right)$$

Collectively, the above pair of equations is called the *parallel axis asymmetric frustum perspective projections* [A99] and these formulas form the basic model for creating stereoscopic display images. Thus, in order to implement asymmetric frustum perspective projections, the two required conditions are: pre-projection shift for **camera offset** and post-projection shift for **frustum asymmetry**. The camera offset, which is used to model the lateral interocular separation of the eyes, introduces negative parallax so the frustum asymmetry is used to balance the overall stereoscopic effect by introducing positive parallax to create an overall balanced parallax experience. By default, the scalar quantity for the pre-projection shift (*c/2*) is equal to that for the post-projection shift (*c/2*).

As explained in [SH97], there is another camera model that involves the use of rotation to generate stereoscopic pairs. But that alternative model produces distortions and will not be used in this project.

A pictorial representation of the viewing volumes created by the projection equations is shown below in Figure 2-2. (The differences between the asymmetric and symmetric viewing volumes are denoted by the solid and dashed lines).

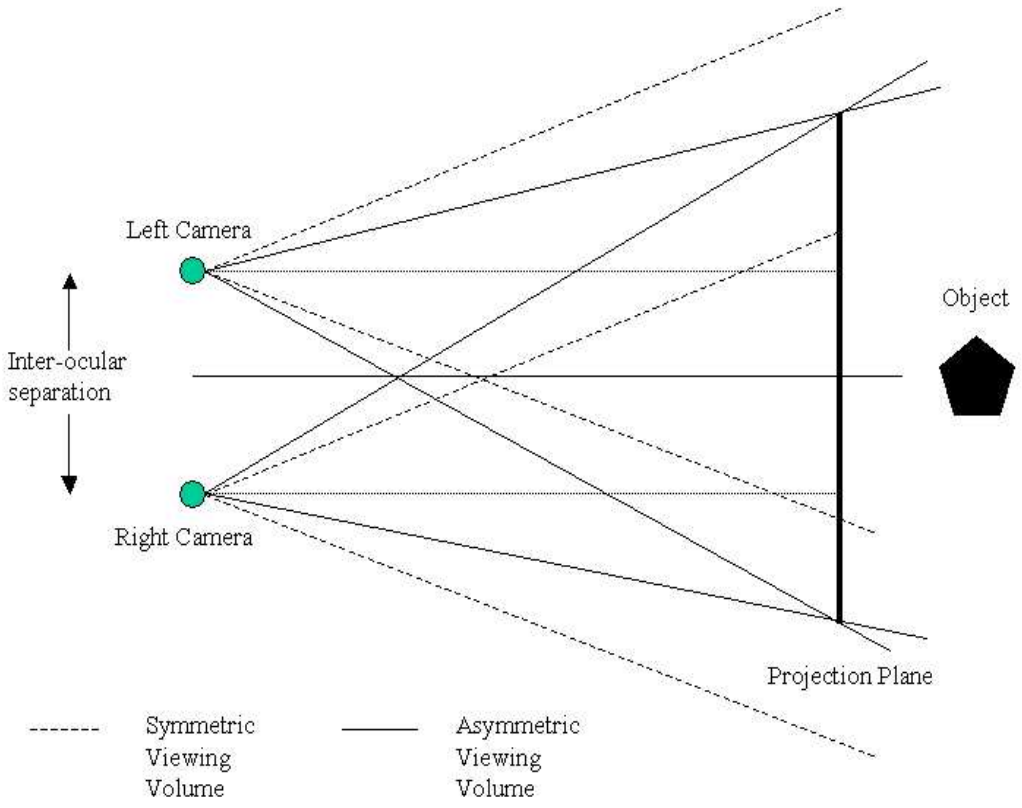


Figure 2-2. Asymmetric vs. Symmetric Viewing Volumes

2.4 Stereoscopic Projections

Figures 2-3a and 2-3b, as shown below, illustrate the effect of implementing the stereoscopic camera model with interocular camera separation and symmetric viewing volumes. (The diagrams below depict exaggerated image offsets for illustrative purposes only). Note also that the viewing frustums and camera placements in the following figures are done using the OpenGL `glFrustum` and `glTranslatef` functions. The `glFrustum` function allows the specification of an asymmetric viewing volume while the `glTranslatef` function allows the precise placement of the camera and its associated viewing volume.

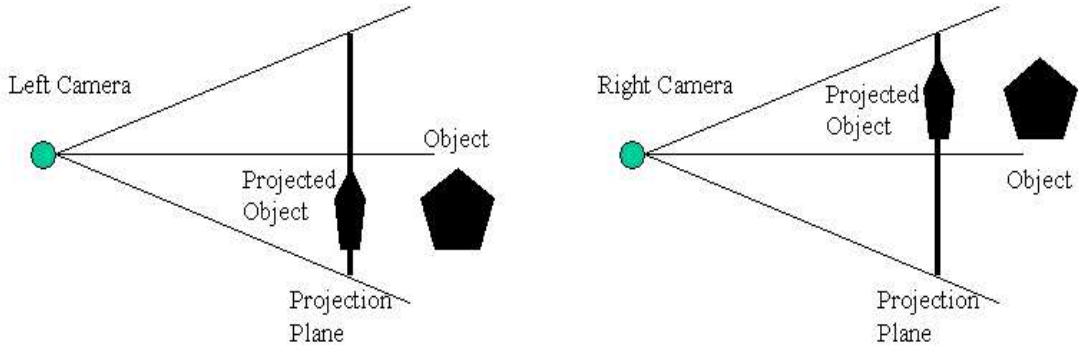


Figure 2-3a. Symmetric frustum projection with lateral camera offset

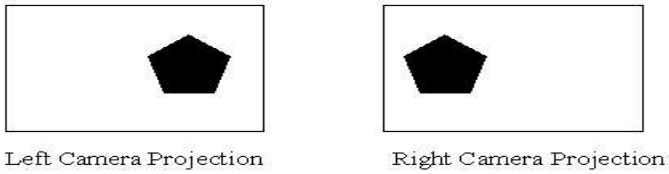


Figure 2-3b. Symmetric frustum projection with lateral camera offset

It can be seen that the left camera projection gets displayed farther to the right than the right camera projection, or put differently the right camera projection gets displayed farther to the left than the left camera projection. Such an arrangement is used to produce negative parallax. (The relationship between projections and parallax is detailed in Section 2.5). Thus, we can observe that the initial stereoscopic projection camera model from Section 2.2 is not quite sufficient.

Figures 2-4a and 2-4b show the effect of using perspective projection using an asymmetric frustum with no camera offset. Note that the asymmetric frustum projection produces an implicit lateral camera offset.

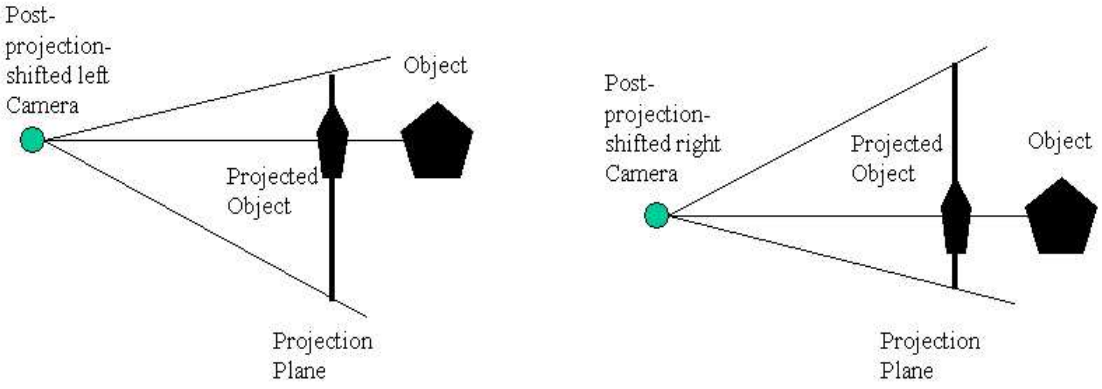


Figure 2-4a. Asymmetric frustum projection only with implicit camera offset

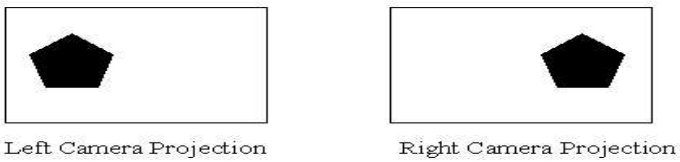


Figure 2-4b. Asymmetric frustum projection only with implicit camera offset

In this case, it can be seen that the left camera projection gets displayed to the left of the right camera projection. This arrangement is used to produce positive parallax. Likewise, asymmetric frustum projection with no camera offset is also not quite sufficient.

Figures 2-5a and 2-5b show the result of using the parallel axis asymmetric frustum perspective projection model from Section 2.3. This perspective projection model uses a combination of asymmetric frustum projection and camera offset for producing balanced parallax.

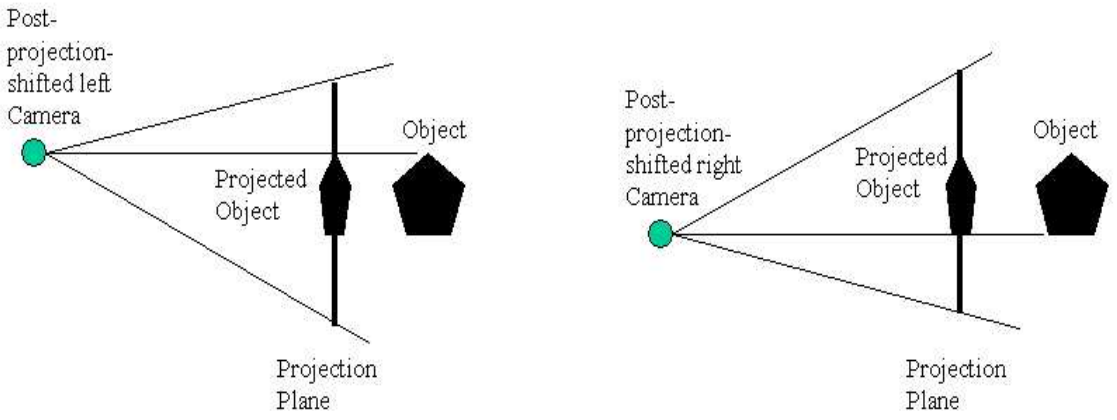


Figure 2-5a. Asymmetric frustum projection with lateral camera offset

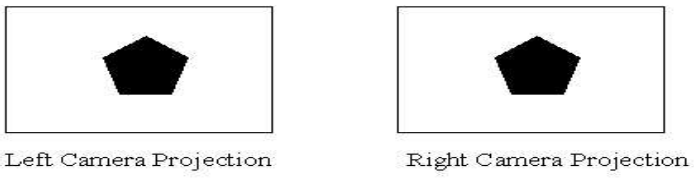


Figure 2-5b. Asymmetric frustum projection with lateral camera offset

It can be seen in Figure 2-5b that the left camera projection almost coincides with the right camera projection. When implemented, the two projection images are not identical but are different enough to produce an equal amount of negative and positive parallax. This is the correct stereoscopic camera model and is the one that will be used in this project.

2.5 Stereoscopic Settings

The stereoscopic settings refer to user adjustable parameters for manipulating the focal point or plane of zero parallax and varying the degree of stereoscopic strength. The interocular camera offset parameter is used to control the overall stereoscopic strength while the accommodation or focal point is used to control and vary between the types of parallax as described below.

It is important to note that parallax effects are induced by retinal disparity which refer to the disparate image projections as seen by the viewer on the CRT display. These disparate image projections in turn are the direct result of the asymmetric frustum stereoscopic camera model from Section 2.3. Also, note that when wearing stereo glasses, at any time only one eye is able to view the projected images. The viewer's mind fuses the image pair and hence perceives an optical illusion.

In general, control of parallax settings can be broadly categorized into the following three main types:

- Default Balanced Parallax

  Figure 2-6a shows balanced parallax as a result of the asymmetric frustum stereoscopic camera model from Section 2.3. Both the left and right eye projection images almost coincide with each other. When viewed using stereoscopic glasses, this induces the projected object to have equal amounts of positive and negative parallax. In other words, the front half of the object will be perceived to be in viewer space while the back half will be perceived to be in CRT space. The plane of zero parallax will directly coincide with the plane of the CRT.

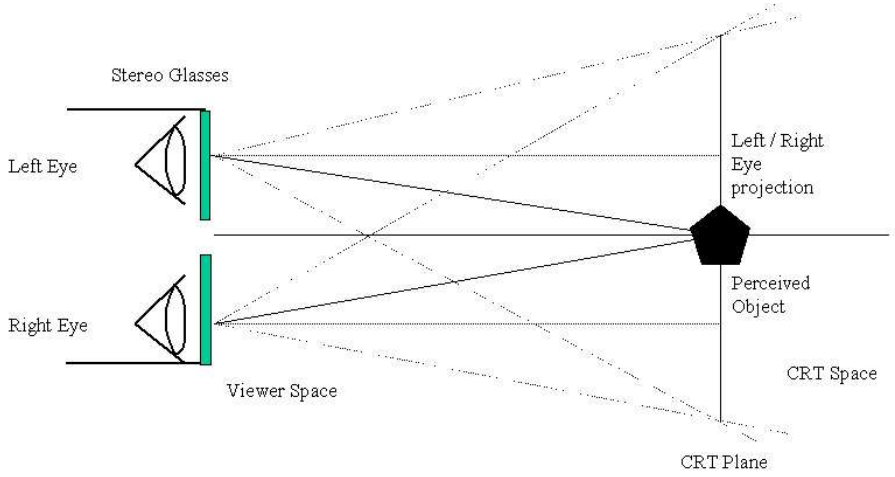


Figure 2-6a. Balanced Parallax

- Positive Parallax

  As shown in Figure 2-6b, the left eye projection is to the left of the right eye projection. In this case, when viewed using stereo glasses, the projected object will be perceived to have positive parallax. This will cause the viewer to perceive

an optical illusion of the object existing in CRT space. In general, positive parallax effects are much more viewer-friendly compared to negative parallax.

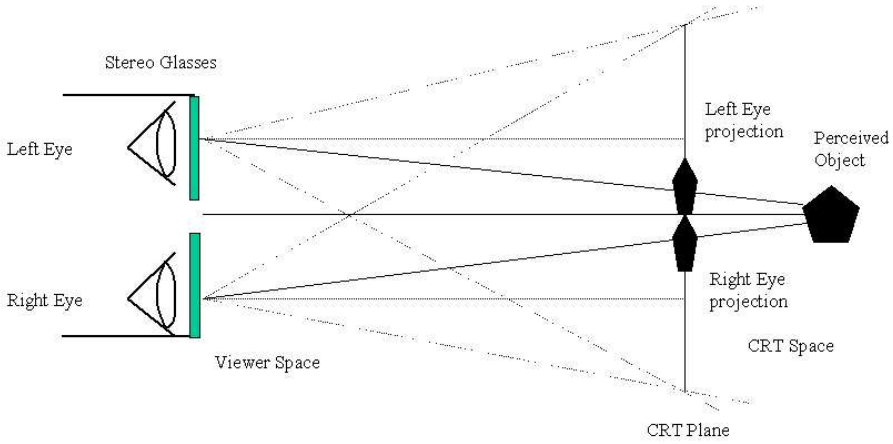


Figure 2-6b. Positive Parallax

- Negative Parallax

  As shown in Figure 2-6c, the left eye projection image is to the right of the right eye projection image. In this case, when viewed using stereo glasses, the projected object will be perceived to have negative parallax. This will cause the viewer to perceive an optical illusion of the object existing in viewer space. In general, negative parallax effects are much more spectacular compared to positive or balanced. However, in order to ensure comfortable viewing, negative parallax should be limited and carefully controlled.

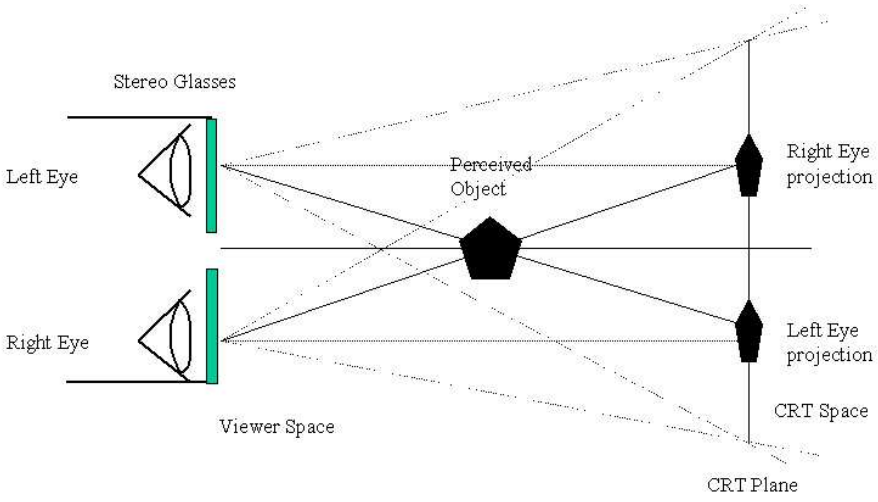


Figure 2-6c. Negative Parallax

Chapter 3

Implementation

In this chapter, the asymmetric frustum stereoscopic camera model from the previous chapter is applied in the implementation of a proof-of-concept stereo prototype using the *GLUT* [K96, KX96] library as well as in the implementation of the stereo extensions to the *Pop Framework* [R02].

3.1 Project Statement

To experience a true stereoscopic sensation on an electro-stereoscopic display system, a stereo-ready workstation or a stereo-enabled graphics card is required. An example of a stereo-ready workstation would be the Octane2 [SG03] line of workstations from Silicon Graphics Inc. For PC users, there are stereo-enabled graphics cards from OEMs such as ELSA [E03], 3DLabs [3D03], and ASUS [A03]. As an alternative for PC users with NVIDIA-specific graphics processors, 3D Stereo [N01] and Detonator XP [N02] device drivers from NVIDIA are available that enables stereoscopic viewing on select chip sets. In all cases, to view stereo in any electro-stereoscopic systems, a shutter-type stereoscopic eyewear such as those from i-O Display Systems [I03] or StereoGraphics Corporation [S03] is required.

Stereo-ready solutions ranging from SGI workstations to OEM stereo cards are currently available in the market but the cost of such specialized hardware is usually exorbitantly high. Fundamentally, the underlying *frame buffer* of such graphics hardware is composed of four buffers: front and back, and left and right. This particular arrangement in graphics hardware is called *quad-buffering*.

In a quad-buffered scheme, the left eye view is rendered onto the left back buffer, and the corresponding right eye view is rendered onto the right back buffer. Whenever the graphics hardware receives the video refresh or sync signal, the back buffers are *page-flipped* or *page-swapped* with the front buffers. Thus the back buffers that were not visible prior to the refresh sync are displayed and the back buffer is again ready to be rendered awaiting the next refresh signal. This video-memory scheme involves four buffers and is thus called quad-buffering. However, when only two buffers (front and back) are used, it would be called *double-buffering*.

This project is based on the OpenGL [SA02, WNDS99] library. Since Version 1.1, OpenGL has native support for stereo rendering hardware. But I contend that it is possible to create stereoscopic optical illusions using non-stereo graphics cards. For purposes of optimal OpenGL performance, only hardware accelerated graphics card is required. The hardware graphics accelerator is required to maintain a high display refresh rate and to “drive” the shutter refresh. The graphics accelerator drives the shutter refresh by keeping the shutter refresh rate in sync with the frame buffer refresh rate.

Furthermore, different graphics hardware provides additional vendor-specific OpenGL extensions [O03, K03] that can be used to further enhance 3D performance.

Therefore, the purpose of this project is to functionally emulate a stereo-enabled graphics card using OpenGL for rendering and the Blue-Line-Code stereo display format to encode the left and right display fields. In the first part of this project, I explored the basics of stereo rendering and stereo viewing using the shutter-type glasses from StereoGraphics Corporation.

Fundamentally, the idea behind this project is to use the double-buffering capability in most commercially available hardware accelerated graphics cards to alternate between displaying the left and right viewpoints at every frame buffer refresh. By time-multiplexing the display of the left and right images into a single back buffer, this alternating pattern effectively “mimics” the real stereoscopic graphics hardware. However, the main limitation of such a scheme is that the graphics hardware has to work twice as hard for each scene and the effective scene refresh rate is half that of a true stereo graphics card. In addition, both the shutter refresh and frame buffer refresh have to stay in sync at all times to avoid eye-straining flicker.

While researching for this project, one of the sources that I studied in great detail was the SDK [SDK97] (Software Development Kit) from StereoGraphics. I used the sample source code that accompanied the SDK along with the technical documentation [SH97] as a guide for producing stereoscopic images.

During the investigative phase, I developed a proof-of-concept prototype application using code based on OpenGL and Win32. Building upon this, I further enhanced and refined my ideas and rewrote the prototype using the GLUT library. Once I understood the nuances of stereoscopic imaging, I incorporated and generalized the algorithms for use in extending the Pop Framework and incorporating object-oriented stereo rendering extensions into the core structures of the framework.

Also, most electro-stereoscopic display systems in the market only support full-screen stereoscopic viewing. However, by closely following the Blue-Line-Code display format specifications, I have developed the prototype to permit both full-screen as well as *stereo-in-a-window* [SS95, MGS95] viewing capabilities.

I based the implementation of this project on the following development platform:

- Windows XP Home Edition operating system
- Dell Inspiron 8100 with Pentium III processor clocked at 1.0 GHz
- 256 MB RAM memory
- NVIDIA GeForce2 Go graphics accelerator with 32 MB video RAM
- StereoEyes LCD (Liquid Crystal Display) shutter glasses from StereoGraphics

3.2 GLUT

A screenshot of the GLUT prototype application running in normal non-stereoscopic windowed mode is shown in Figure 3-1.

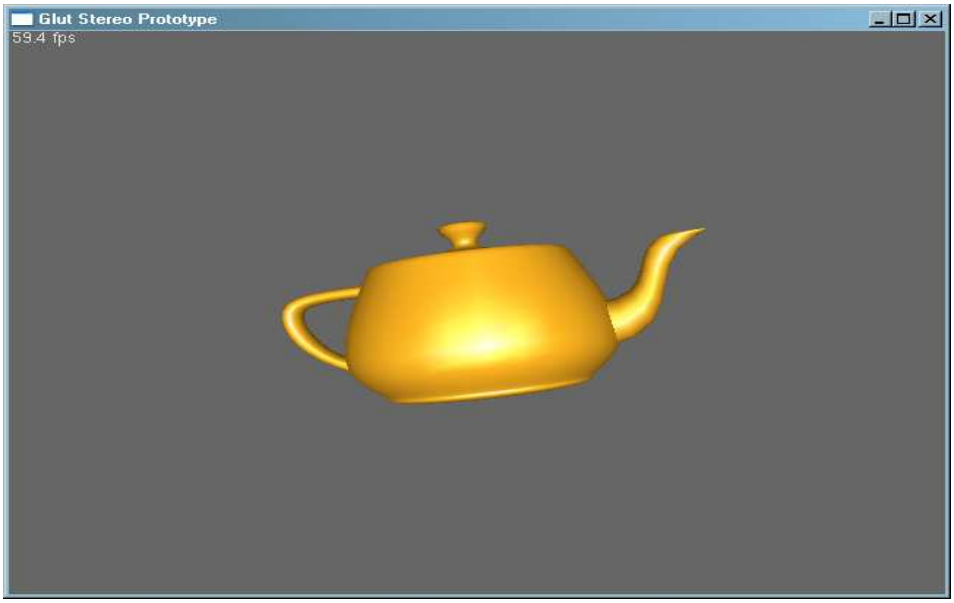


Figure 3-1. Screenshot from application

The following screenshots, as shown in Figures 3-2a and 3-2b, shows the application running in hardware accelerated stereo-in-a-window mode. The view setting of Figures 3-2a and 3-2b show extreme negative parallax.

As explained in the previous chapter, it is possible to tell that this is negative parallax by observing the horizontal offset of one view with respect to the other. Projections that produce negative parallax show the left eye projection horizontally offset to the right of the right eye view or the right eye projection is horizontally offset to the left of the left eye view. In positive parallax, the reverse is true. Therefore, it is possible to differentiate the parallax type by carefully observing the difference in projection offset.

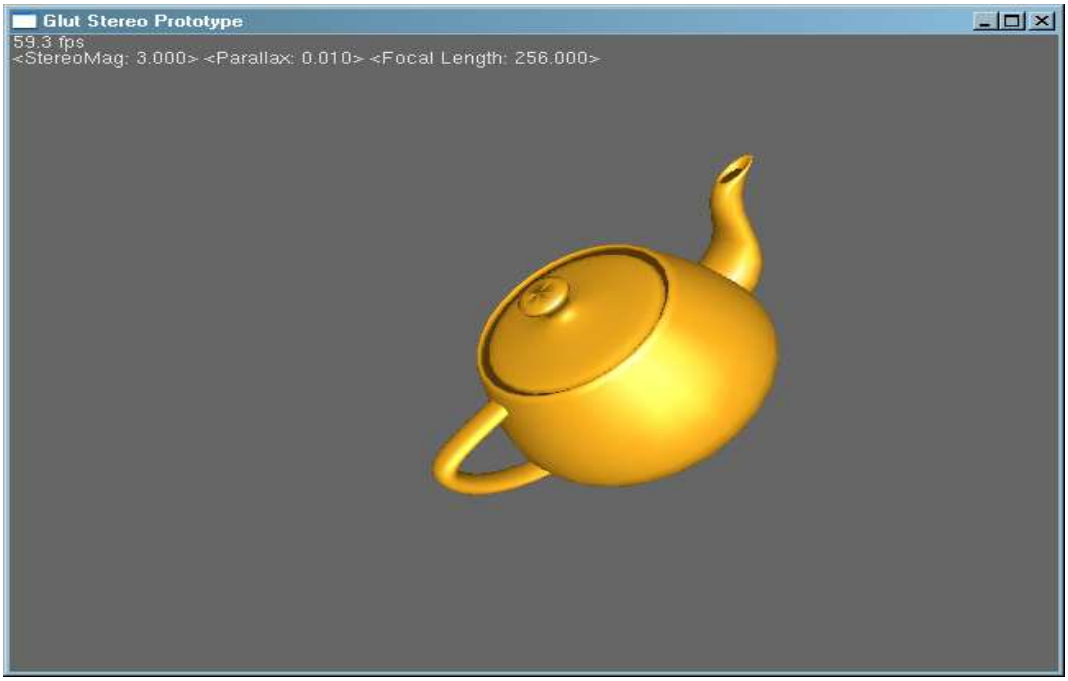


Figure 3-2a. Left Eye projection in extreme negative parallax

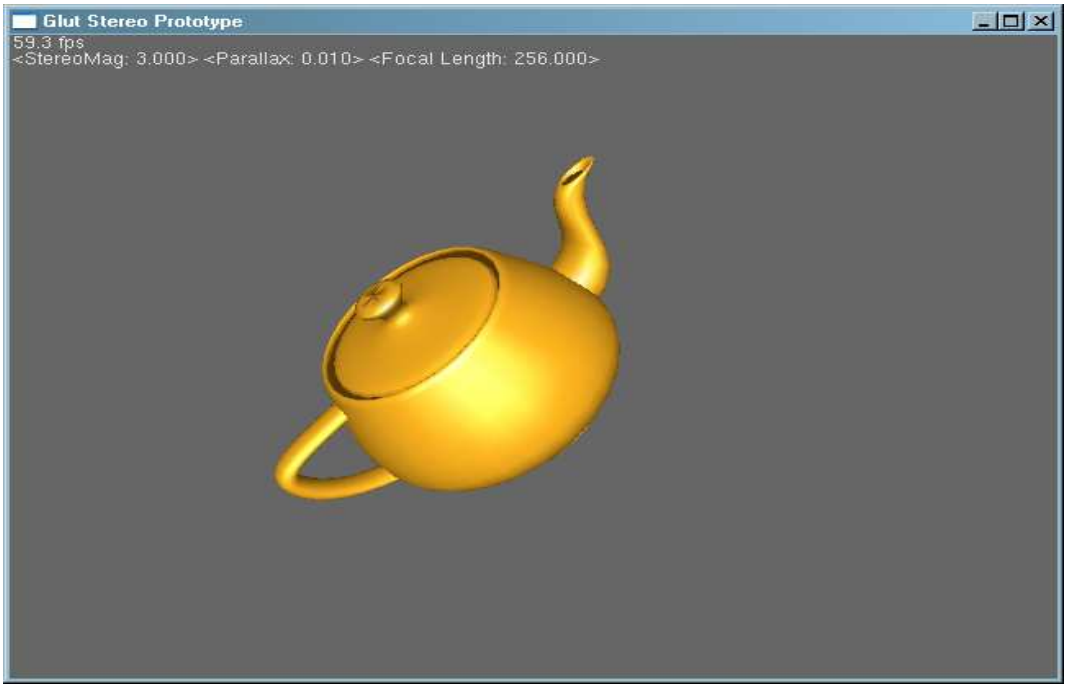


Figure 3-2b. Right Eye projection in extreme negative parallax

3.3 Pop Framework

This project is based on release Version 27.1 of the Pop Framework. Using the experimental code from the GLUT prototype, I extended the Pop Framework to be able to render stereoscopically using the stereoscopic camera model from the previous chapter.

3.3.1 Architecture

In order to better understand the extensions, the Pop Framework has to be examined in some detail. The Pop Framework is based on the Document-View (Model View Controller) [R02] and Multiple Document [R02] architectural design patterns that are prevalent in MFC (Microsoft Foundation Classes).

The triad of classes `CPopApp`, `CPopDoc`, and `CPopView` form the core foundation upon which the framework is built. The `CPopApp` is a class for the overall application. The `CPopDoc` is used as a container class for game specific data and data structures, and the `CPopView` is a class used for the GUI (Graphical User Interface). The following diagrams, Figures 3-3a and 3-3b, show the high-level UML schemas on which Pop Framework is based on. Specifically, Figure 3-3a shows the MVC / Document-View architecture and Figure 3-3b shows the core Pop Framework classes that are used in implementing the stereo extension.

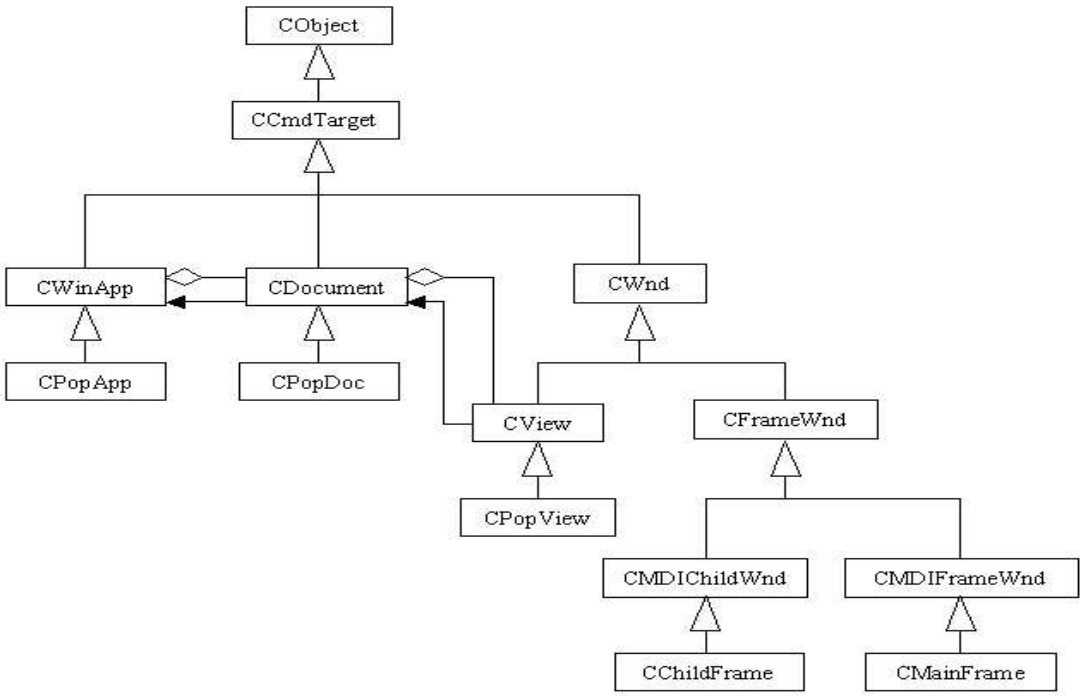


Figure 3-3a. High-level UML schema of Pop Framework

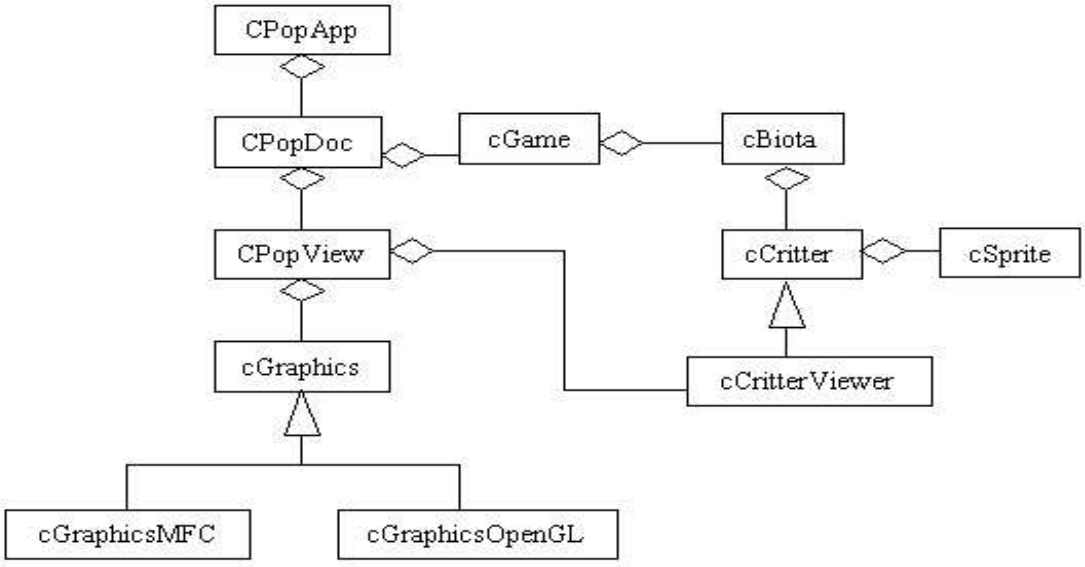


Figure 3-3b. UML schema of the core Pop Framework classes

3.4 Pop Framework Stereo Extension

In implementing the stereo extensions, a total of 6 classes are involved. 2 new classes were created and 4 of the existing framework classes were modified. The new classes are `cCritterViewerStereo` and `cStereoDialog`. The modified classes are `cCritterViewer`, `CPopView`, `cGraphics`, and `cGraphicsOpenGL`. Figures 3-4a and 3-4b highlight the required modifications.

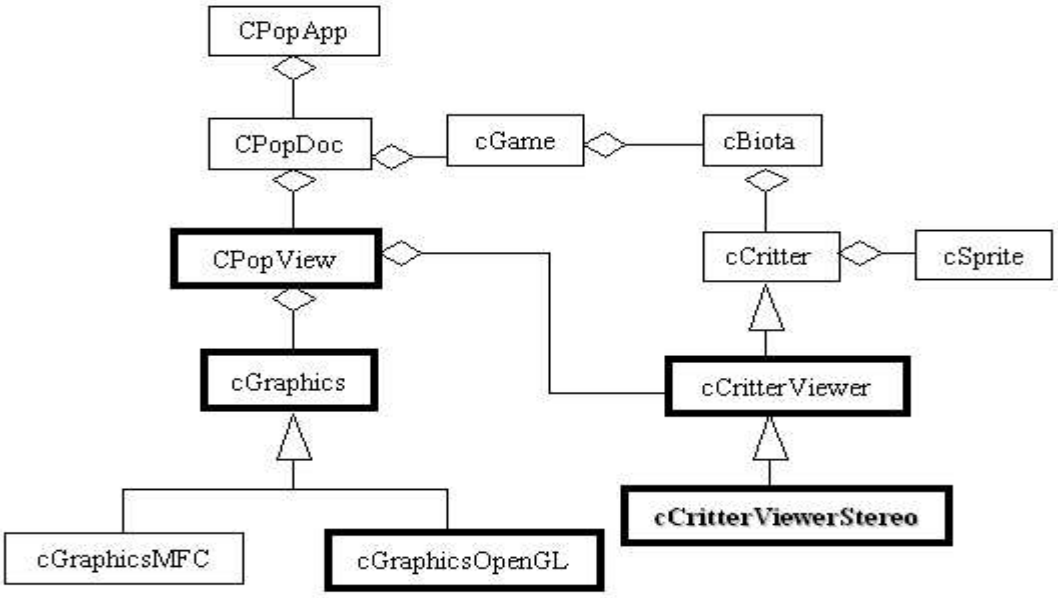


Figure 3-4a. Pop Framework Stereo Extension

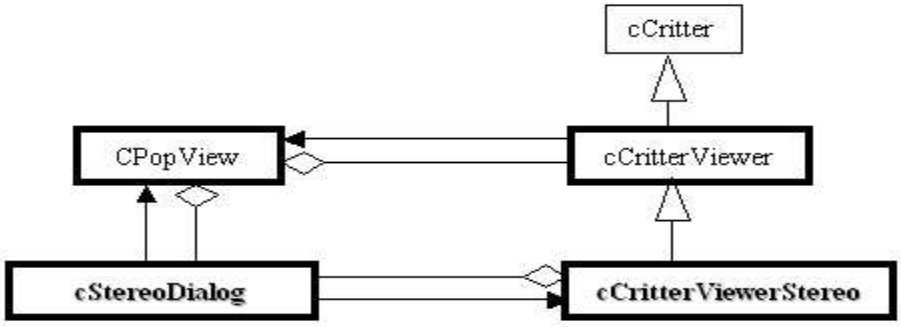


Figure 3-4b. Pop Framework Stereo Extension

In order to implement the stereo extensions in a clean and object-oriented manner, I had to closely study the function call cascade starting from `CPopApp::OnIdle` as shown in Figure 3-5. I narrowed down the relevant functions to be `cCritterViewer::update` and `CPopView::OnDraw`. These two methods warrant a detailed examination.

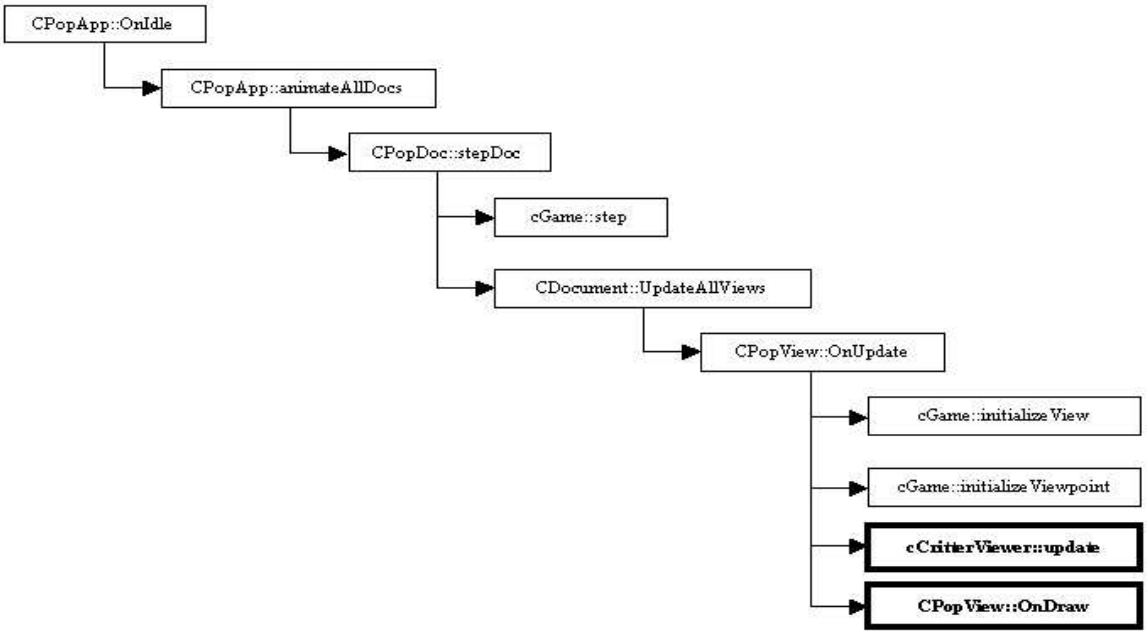


Figure 3-5. Refresh message pump

The `update` method is called on each refresh to set up and compute the clipping planes, aspect ratio, field of view, and view orientation or camera attitude, properties that pertain to the camera and its projection characteristics. As will be explained shortly in Section 3.4.5, since this method is called on each refresh just before the `OnDraw` display method, this method is ideal for overriding and setting the stereoscopic camera properties before each rendering.

The `OnDraw` method is used to initialize the rendering context, projection and model view transformation matrices, and display the world and critter objects. As shown in the code fragment below, after setting up the appropriate projection and model view transformations, the world and critters are rendered into OpenGL's back frame buffer and then finally displayed during the page-swap sequence.

```
void CPopView::OnDraw(CDC* pDC)
{
      …

      //Install the projection and view matrices.
      _pviewpointcritter->loadProjectionMatrix();
      //Initializes the MODELVIEW matrix
      _pviewpointcritter->loadViewMatrix();

      //Draw the world, by default as a background and a foreground
      //rectangle.
      pgame()->drawWorld(_pgraphics, _drawflags);

      //Draw the critters.
      pgame()->drawCritters(_pgraphics, _drawflags);

      // Display
      _pgraphics->display(this, pDC);
}
```

This display method calls two different transformation functions `cGraphicsOpenGL::loadProjectionMatrix` and `cGraphicsOpenGL::loadViewMatrix`. These methods correspond with the OpenGL's projection and model view transformations. The `loadProjectionMatrix` function is used for the selection of camera lens type while `loadViewMatrix` is used for placement of objects within the viewing volume. With regards to the stereoscopic extension, I discovered that only the projection function has to be modified. So an excerpt of the `loadProjectionMatrix` function is shown below.

```
void cCritterViewer::loadProjectionMatrix() const
{
```

```
        pgraphics()->matrixMode(cGraphics::PROJECTION);
        pgraphics()->loadIdentity();

        …
        {
              pgraphics()->perspective(fieldOfViewDegrees(),
                    _aspect, _znear, _zfar);
        }
        pgraphics()->matrixMode(cGraphics::MODELVIEW);
        pgraphics()->loadIdentity();
}
```

The `perspective` projection function is defined in `cGraphicsOpenGL::perspective`. An excerpt of this method is shown below.

```
virtual void perspective(Real fieldofviewangleindegrees,
            Real xtoyaspectratio,
            Real nearzclip,
            Real farzclip)
{
      ::gluPerspective(GLdouble(fieldofviewangleindegrees),
            GLdouble(xtoyaspectratio), GLdouble(nearzclip),
            GLdouble(farzclip));
}
```

As will be explained shortly in Sections 3.4.3 and 3.4.4, an additional projection method has to be written to facilitate the asymmetric frustum perspective projection. Note that by default this `perspective` function creates a symmetric viewing volume, which is not what we need.

3.4.1 `cCritterViewer`

This class is used in the framework for defining the position, projection and viewing properties of the camera. To implement the stereo extension, the first step I took was to use the Strategy Pattern and make `loadProjectionMatrix` into a strategy method to allow my stereoscopic camera subclass to inherit and redefine the projection behavior. So the modified `cCritterViewer` class looks something like this.

```
class cCritterviewer
{
public:
      …
      virtual void loadProjectionMatrix() const;
      …
};
```

3.4.2 `CPopView`

Then, in the `CPopView` class, I added a stereo viewing button on the toolbar as shown in Figure 3-6. This toolbar button is used to toggle stereo viewing mode on and off.

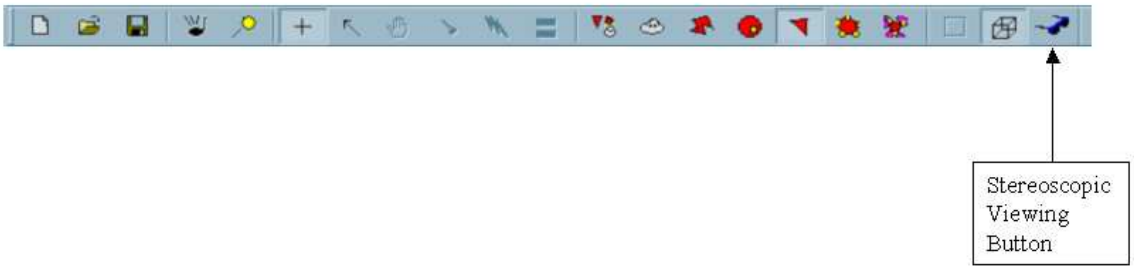


Figure 3-6. Stereo button

Associated with this button, I added the `CPopView::OnViewOpenglStereo` command handler. As shown in the fragment below, within this function, there is a `BOOL` variable for keeping track of the stereo status. When stereo mode is turned on, a `cStereoDialog` and `cCritterViewerStereo` objects are created dynamically. (`cCritterViewerStereo` and `cStereoDialog` objects are described in Sections 3.4.5 and 3.4.6.) However, when stereo mode is turned off the `cStereoDialog` object is deleted and destroyed. The `_pviewpointcritter` pointer variable in `CPopView` refers to a `cCritterViewerStereo` object when stereo is on and `cCritterViewer` object when stereo is off. The reason being that the default camera model in `cCritterViewer` does not allow for stereoscopic

projections and the new camera model in `cCritterViewerStereo` has the necessary implementation to facilitate stereo viewing. So the pointer reference to the camera object changes depending on whether stereo is on or off.

```
void CPopView::OnViewOpenglStereo()
{
	static BOOL stereoOn = TRUE;
	…

	delete _pviewpointcritter;

	if (stereoOn) {
		if (_pDlg == NULL) {
			_pDlg = new cStereoDialog(this);
			_pviewpointcritter = new cCritterViewerStereo(this,
				_pDlg);
			_pDlg->Create((cCritterViewerStereo *)
				_pviewpointcritter);
		}
	} else {
		_pviewpointcritter = new cCritterViewer(this);
		_pDlg->DestroyWindow();
		delete _pDlg;
		_pDlg = NULL;
	}

	stereoOn = !stereoOn;
	setGraphicsClass(RUNTIME_CLASS(cGraphicsOpenGL));
	OnUpdate(NULL, CPopDoc::VIEWHINT_STARTGAME, NULL);
}
```

Also, to assist in dynamically creating and maintaining a reference to the `cStereoDialog` object, I added a member variable in the `CPopView` class as shown below. The main reason the `cStereoDialog` object is created on the heap is because the stereo settings dialog box is modeless so we need to allocate that object from the heap instead of from the stack.

```
class CPopView
{
…

protected:
	cStereoDialog* _pDlg;
```

```
…
};
```

The `_pDlg` pointer variable is used to reference the modeless dialog box that contains the stereoscopic settings controls.

The next notable addition to `CPopView` was to update the message on the status bar of the application window to reflect the current viewing status. So the new `updateStatusBar` method would look like this.

```
void CPopView::updateStatusBar()
{
	…
	if (_pviewpointcritter->GetRuntimeClass() ==
		RUNTIME_CLASS(cCritterViewerStereo)) {
		statusmessage += " STEREO ON";
	} else {
		statusmessage += " STEREO OFF";
	}
	cMainFrame->SetMessageText(statusmessage); //Write to status bar
}
```

And finally, since the stereo settings dialog box is dynamically created, I had to make sure to release the memory for it in the class destructor and `OnDestroy` handler function like this.

```
CPopView::~CPopView()
{
	…
	if (_pDlg)
		delete _pDlg;
	…
	_pDlg = NULL;
}

void CPopView::OnDestroy()
{
	…
	if (_pDlg)
		delete _pDlg;
	…
```

```
      _pDlg = NULL;
      CWnd::OnDestroy();
}
```

3.4.3 `cGraphics`

In order to implement the asymmetric frustum camera model from the previous chapter, I discovered that the `cGraphics::perspective` projection function is not sufficient. I needed another method to allow the specification of an asymmetric viewing volume. So I added another strategy method called `frustum` to `cGraphics` to allow the subclass `cGraphicsOpenGL` to override and redefine with the correct behavior. So the new `cGraphics::frustum` method looks like this with a default empty function body.

```
class cGraphics
{
      …
      virtual void frustum(Real l, Real r,
            Real b, Real t, Real n, Real f) {}
      …
};
```

3.4.4 `cGraphicsOpenGL`

This class provides wrapper functions for the core OpenGL functions. It is a subclass of `cGraphics`. Here I redefined the `frustum` method to be a wrapper method for the `glFrustum` function, which by design allows the creation of an asymmetric viewing volume. This is the exact projection function I need. So the new `frustum` method looks like this.

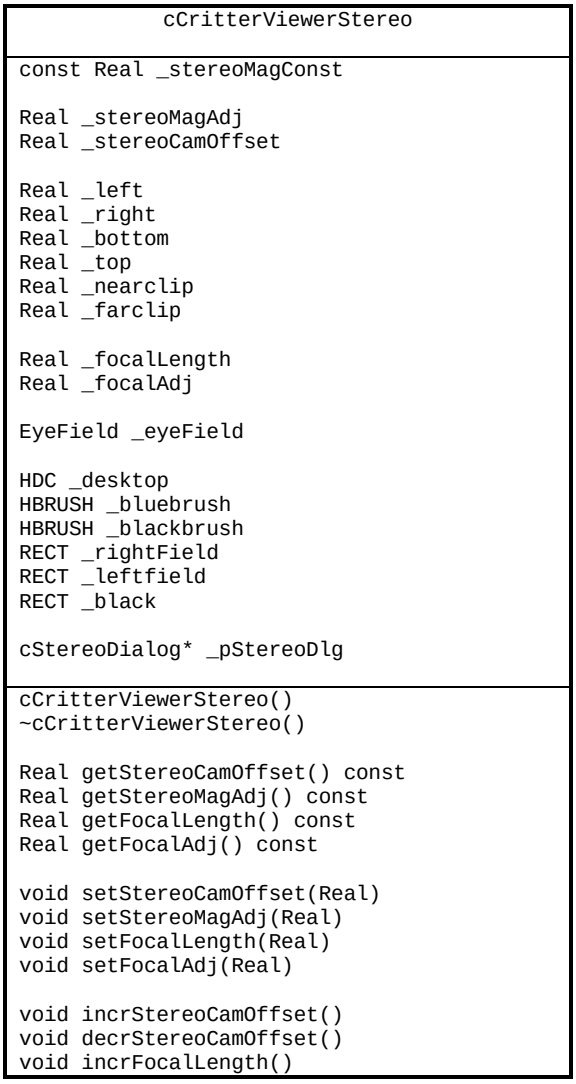
```
class cGraphicsOpenGL
{
      …
      virtual void frustum(Real l, Real r,
            Real b, Real t, Real n, Real f)
            {::glFrustum(l, r, b, t, n, f);}
      …
```

```
};
```

We have now made the necessary changes to allow the stereoscopic extensions to be integrated.

3.4.5 `cCritterViewerStereo`

This new class is used for setting the stereoscopic camera properties and implementation of the Blue Line Code display format. A UML diagram for this class is shown below.



```
void decrFocalLength()

void renderBlueLines() const
void toggleFields()
void releaseDC()

void loadProjectionMatrix() const
void update(CPopView*, Real)
void Serialize(Carchive&)
```

The member attributes are defined follows:

_stereoMagConst	This variable is a predefined constant with magnitude 0.07 for computing the stereo camera offset.
_stereoMagAdj _stereoCamOffset	_stereoMagAdj is a user changeable variable used to scale and modify the _stereoCamOffset variable. _stereoCamOffset = (_right - _left) * _stereoMagAdj * _stereoMagConst
_focalAdj _focalLength	_focalAdj is a user changeable variable used to scale and modify the _focalLength variable. _focalLength = _focalAdj * _farclip
_left, _right, _bottom, _top, _nearclip, _farclip	These refer to the dimensions of the camera viewing volume.
_eyeField	Enumerated type to denote left / right eye field. (FIELD_LEFT or FIELD_RIGHT)
_desktop _bluebrush _blackbrush _rightField _leftfield _black	GDI structures for implementing Blue Line Code format.
_pStereoDlg	Pointer to cStereoDialog object (modeless dialog box)

Similarly, the methods are defined as follows:

cCritterViewerStereo ~cCritterViewerStereo	Constructor and destructor. Explanation in later section.
getStereoCamOffset getStereoMagAdj getFocalLength getFocalAdj	Accessor functions for retrieving the stereoscopic camera parameters.
setStereoCamOffset setStereoMagAdj setFocalLength setFocalAdj	Mutator functions for setting the stereo camera parameters.
incrStereoCamOffset decrStereoCamOffset incrFocalLength decrFocalLength	Manipulator functions for stereo camera parameters.

renderBlueLines toggleFields releaseDC	Blue Line Code implementation functions.
loadProjectionMatrix update Serialize	Overrides from parent cCritterViewer class. loadProjectionMatrix for implementing stereoscopic projection. update for setting the stereoscopic camera parameters before performing projection. Serialize for serialization purposes.

The most important methods in this class are `loadProjectionMatrix` and `update`. In order to implement the asymmetric frustum perspective projection, I first had to override the default `loadProjectionMatrix` method and provide my implementation to match the stereo camera model. The asymmetric frustum perspective projection, as described in the previous chapter, can be implemented using OpenGL `glFrustum` and `glTranslatef` functions. For our stereo camera model, left eye projections are implemented using code that looks like this:

```
glFrustum(left + camoffset * near/focalLen,
      right + camoffset * near/focalLen,
      bottom, top, near, far);
glTranslatef(camoffset, 0.0f, 0.0f);
```

Likewise for right eye projections, the code looks like this:

```
glFrustum(left - camoffset * near/focalLen,
      right - camoffset * near/focalLen,
      bottom, top, near, far);
glTranslatef(-camoffset, 0.0f, 0.0f);
```

Therefore, in order to implement our stereo camera model, the redefined projection method will have to be written along the lines of the code fragment shown below in order to correctly reflect the properties of the asymmetric frustum stereo projection.

```
void cCritterViewerStereo::loadProjectionMatrix() const
{
```

```
	/* change MATRIXMODE to GL_PROJECTION */
	pgraphics()->matrixMode(cGraphics::PROJECTION);
	/* initialize by loading Identity matrix */
	pgraphics()->loadIdentity();

	…
	{
		pgraphics()->frustum(
			_left - _stereoCamOffset * _nearclip / _focalLength,
			_right - _stereoCamOffset * _nearclip / _focalLength,
			_bottom,
			_top,
			_nearclip,
			_farclip);

		if (_plistener->GetRuntimeClass() ==
			RUNTIME_CLASS(cListenerViewerRide))
		{
			pgraphics()->translate(cVector(- _stereoCamOffset,
				0.0f, - _znear /*0.0f*/));
		} else {
			pgraphics()->translate(cVector(- _stereoCamOffset,
				0.0f, 0.0f));
		}
	}
	/* change MATRIXMODE to GL_MODELVIEW */
	pgraphics()->matrixMode(cGraphics::MODELVIEW);

	/* initialize by loading Identity matrix */
	pgraphics()->loadIdentity();
}
```

With this, the next method that I had to redefine was the `update` method. The default behavior of this method is to compute the camera view settings and attitudes at each refresh time step, which are necessary to ensure proper viewing of the framework. I chose to reuse the existing behavior of `update` and augment it by adding the necessary steps to compute the stereo camera parameters. Therefore, the redefined method first calls the base class method to compute the necessary camera characteristics, and then computes the stereo-specific parameters: `_stereoCamOffset` and `_focalLength` for each refresh update just before the main display function calls the new stereoscopic `loadProjectionMatrix` method. So the new `update` method looks like the fragment below.

```
void cCritterViewerStereo::update(CPopView *activeview, Real dt)
{
	…

	/* call base class function first */
	cCritterViewer::update(activeview, dt);

	/* update data structures for viewing frustum */
	_left = - _aspect * _znear * tan(_fieldofviewangle/2.0f);
	_right = _aspect * _znear * tan(_fieldofviewangle/2.0f);
	_bottom = - _znear * tan(_fieldofviewangle/2.0f);
	_top = _znear * tan(_fieldofviewangle/2.0f);
	_nearclip = _znear;
	_farclip = _zfar;

	/* stereo camera offset */
	_stereoCamOffset = (_right - _left) * _stereoMagConst *
		_stereoMagAdj;

	if (_eyeField == FIELD_LEFT)
		_stereoCamOffset = -_stereoCamOffset;

	_focalLength = _focalAdj * _farclip;

	/* set fields in dialog box */
	temp.Format("%.5f", _focalLength);
	SetDlgItemText(hwnd, IDC_FOCALLEN, temp);
	temp.Format("%.5f", _stereoMagAdj);
	SetDlgItemText(hwnd, IDC_CAMOFFSET, temp);

	renderBlueLines();
	toggleFields();
}
```

Since this method is called on every refresh, I chose to let this method set and display the parameters in the stereoscopic modeless dialog box controls. Also, since this method is called just before the actual display function, I added function calls to `renderBlueLines` and `toggleFields` to implement the Blue Line Code format.

The Pop Framework runs in windowed mode only, therefore I had to use GDI (Graphics Device Interface) functions to implement BLC for stereo-in-a-window. Specific to this mode, the `renderBlueLines` method is used to display the black rectangle first followed by blue rectangle to the main desktop window while `toggleFields` is used to update the

eye field parameter to alternate between each eye for each refresh. So, the functions for `renderBlueLines` and `toggleFields` are shown below.

```
void cCritterViewerStereo::renderBlueLines() const
{
      // set black brush first
      ::FillRect(_desktop, &_black, _blackbrush);

      (_eyeField == FIELD_LEFT) ?
            ::FillRect(_desktop, &_leftField, _bluebrush) : \
            ::FillRect(_desktop, &_rightField, _bluebrush);
}

void cCritterViewerStereo::void toggleFields()
{
      _eyeField = (_eyeField==FIELD_LEFT) ? FIELD_RIGHT : FIELD_LEFT;
}
```

In order to correctly implement the Blue Line Code format, I put the initialization steps for this stereo format in the `cCritterViewerStereo` constructor. So the code to the constructor looks like the following.

```
cCritterViewerStereo::cCritterViewerStereo(CPopView *pview,
      cStereoDialog *dlg)
: cCritterViewer(pview)
{
      …

      /* initialize BLC format */
      _desktop = ::GetDC(NULL);

      /* set blue brush */
      _bluebrush = ::CreateSolidBrush(RGB(0,0,255));
      /* set black brush */
      _blackbrush = ::CreateSolidBrush(RGB(0,0,0));

      /* set dimensions of black and blue rectangles */
      ::SetRect(&_black, 0, ::GetSystemMetrics(SM_CYSCREEN)-3,
            ::GetSystemMetrics(SM_CXSCREEN),
            ::GetSystemMetrics(SM_CYSCREEN));
      ::SetRect(&_leftField, 0, ::GetSystemMetrics(SM_CYSCREEN)-3,
            ::GetSystemMetrics(SM_CXSCREEN) * 0.75f,
            ::GetSystemMetrics(SM_CYSCREEN));
      ::SetRect(&_rightField, 0, ::GetSystemMetrics(SM_CYSCREEN)-3,
            ::GetSystemMetrics(SM_CXSCREEN) * 0.25f,
            ::GetSystemMetrics(SM_CYSCREEN));
}
```

The `GetDC(NULL)` function is used to retrieve the `HDC` to the desktop window, the `CreateSolidBrush` function is used to create the blue / black colored rectangular regions, and the `SetRect` function is used to initialize the region dimensions in accordance to the format specification. Therefore, just before the left and right stereo camera projections are displayed by the display function, the stereo specific parameters are computed and updated, the blue / black rectangles are displayed on the desktop window using GDI functions, and eye field parameter updated ready for the next eye projection.

Finally, associated with this constructor is the destructor. For this function, I added the code to release the `HDC _desktop` variable to the main desktop window once the destruction sequence was started. So the code to the destructor is as follows.

```
cCritterViewerStereo::~cCritterViewerStereo()
{
      releaseDC();
}
```

And the `releaseDC` inline method is as follows.

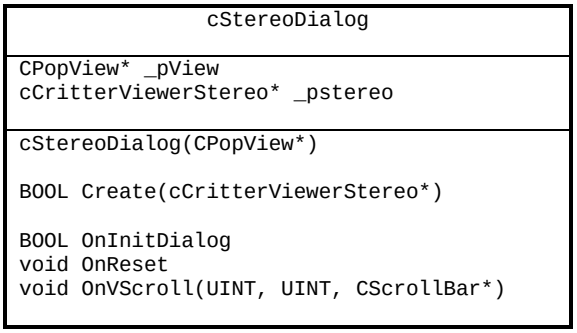
```
class cCritterViewerStereo
{
      …
      void releaseDC() { ::ReleaseDC(NULL, _desktop); }
      …
};
```

3.4.6 `cStereoDialog`

The next new class that I implemented is used for the creation of a modeless dialog box containing the controls that are specific to stereo viewing. A screenshot of this window is shown in Figure 3-7.

Figure 3-7. Stereo viewing controls

The UML diagram for this class is shown as follows.



Since this window is modeless, it has to be created dynamically and is created in the stereo button command handler as described earlier in Section 3.4.2. The `cStereoDialog` class maintains pointer references to its parent `CPopView` window object and the `cCritterViewerStereo` stereoscopic camera object.

The stereoscopic camera parameters are focal length and camera offset. The `Focal Length` parameter is used to control the exact placement of the plane of zero parallax and the `Cam Offset` parameter is used to control and fine-tune the overall stereoscopic strength. Figure 3-8 shows how these parameters are used in this application.

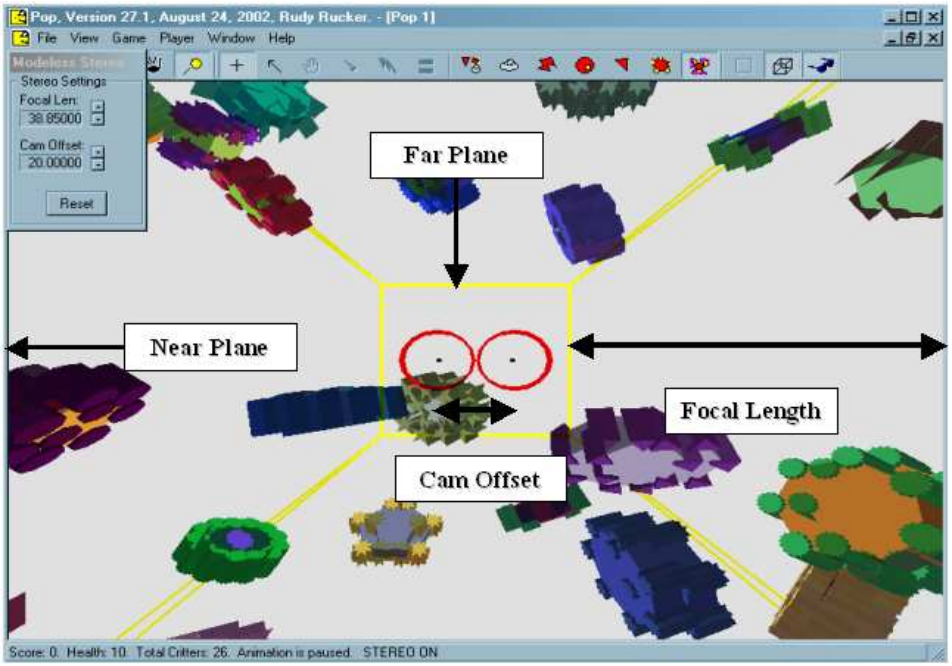


Figure 3-8. Stereo control parameters

The widgets for changing these parameters are implemented using spinner controls. The reason I chose spinners instead of sliders was to allow the user precise control. The parameter values are displayed in read-only text boxes and the reset button is used to set the parameters to their default values. By changing the parameters using the spin controls, I have made this application flexible enough to tune the stereoscopic experience to each viewer's comfort.

The following code fragments are taken from the reset button and spinner control handler functions.

```
void cStereoDialog::OnVScroll(UINT nSBCode, UINT nPos,
      CScrollBar* pScrollBar)
{
      …
      switch (pScrollBar->GetDlgCtrlID()) {
            case IDC_SPINFOCAL:
                  _pstereo->setFocalAdj(nPos/(Real)(150.0f));
                  break;
            case IDC_SPINCAM:
                  _pstereo->setStereoMagAdj(nPos/(Real)(5.0f));
                  break;
      }
      …
}

void cStereoDialog::OnReset()
{
      …
      _pstereo->setFocalAdj(1.0f);
      _pstereo->setStereoMagAdj(1.0f);
      …
}
```

By design, I have made the range of the camera offset parameter to be 0 ~ 20 with steps of 0.5. The default is 1.0, which results in comfortable but weak stereo. 0 means no stereo at all and 20 the strongest. The focal length parameter ranges from 0 ~ 2 * far plane distance. The default value for this parameter is 1.0 * far plane distance.

As shown in Figure 3-9, I designed the far plane to project to zero parallax by default. In this negative parallax configuration, when viewed using stereo glasses the critters will appear to be originating and flying away from the plane of the CRT toward the viewer into viewer space. To experience positive parallax, the focal length parameter is set to the near plane so that the near plane projects to zero parallax. In this case, the viewer will

experience the critters originating from deep CRT space flying toward the viewer who is positioned at the CRT plane.

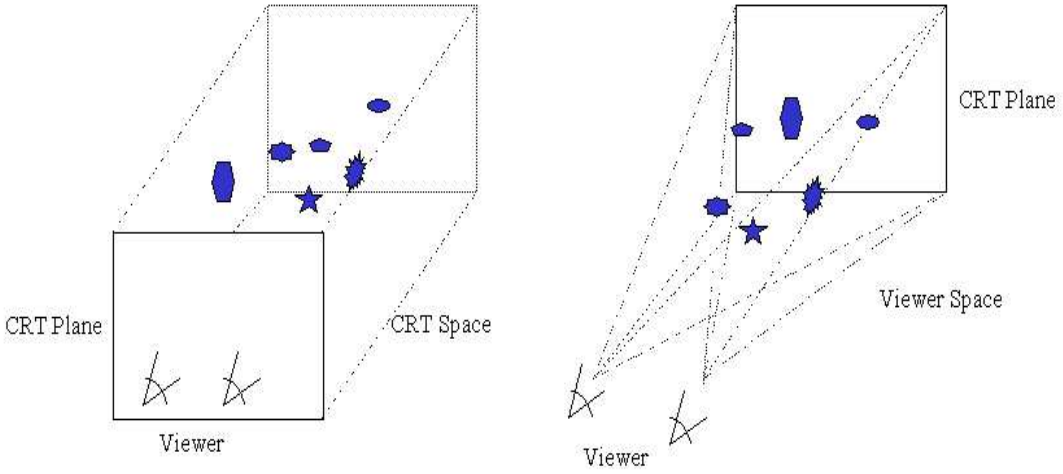


Figure 3-9. Focal length set to Near vs. Far planes

3.5 Summary / Screenshots

In summary, the Pop Framework stereo extension consists of the following files:

1. critterviewerstereo.h
2. critterviewerstereo.cpp
3. critterViewer.h
4. critterViewer.cpp
5. popview.h
6. popview.cpp
7. graphics.h
8. graphicsOpenGL.h
9. stereoDialog.h
10. stereoDialog.cpp
11. resource.h
12. resource.rc

Screenshots taken from the final stereo-extended Pop Framework application are shown below in the Figures 3-10a, b, c, and d.

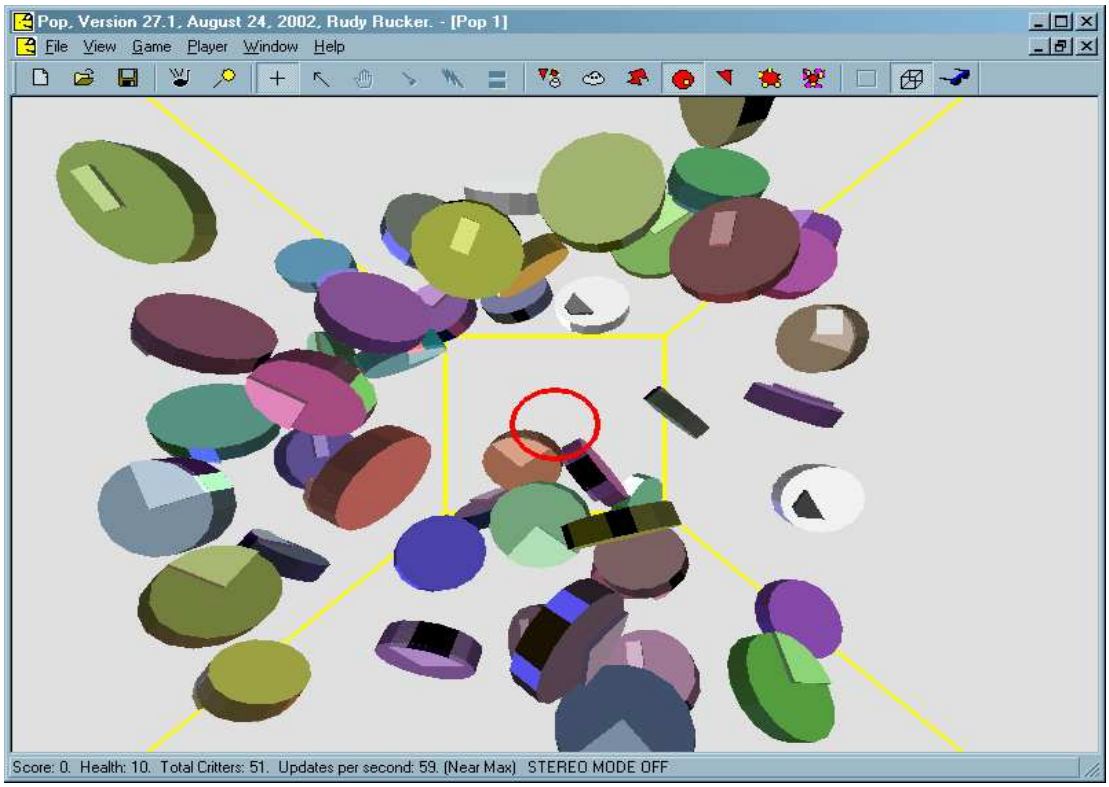


Figure 3-10a. Defender 3D in default non-stereo mode

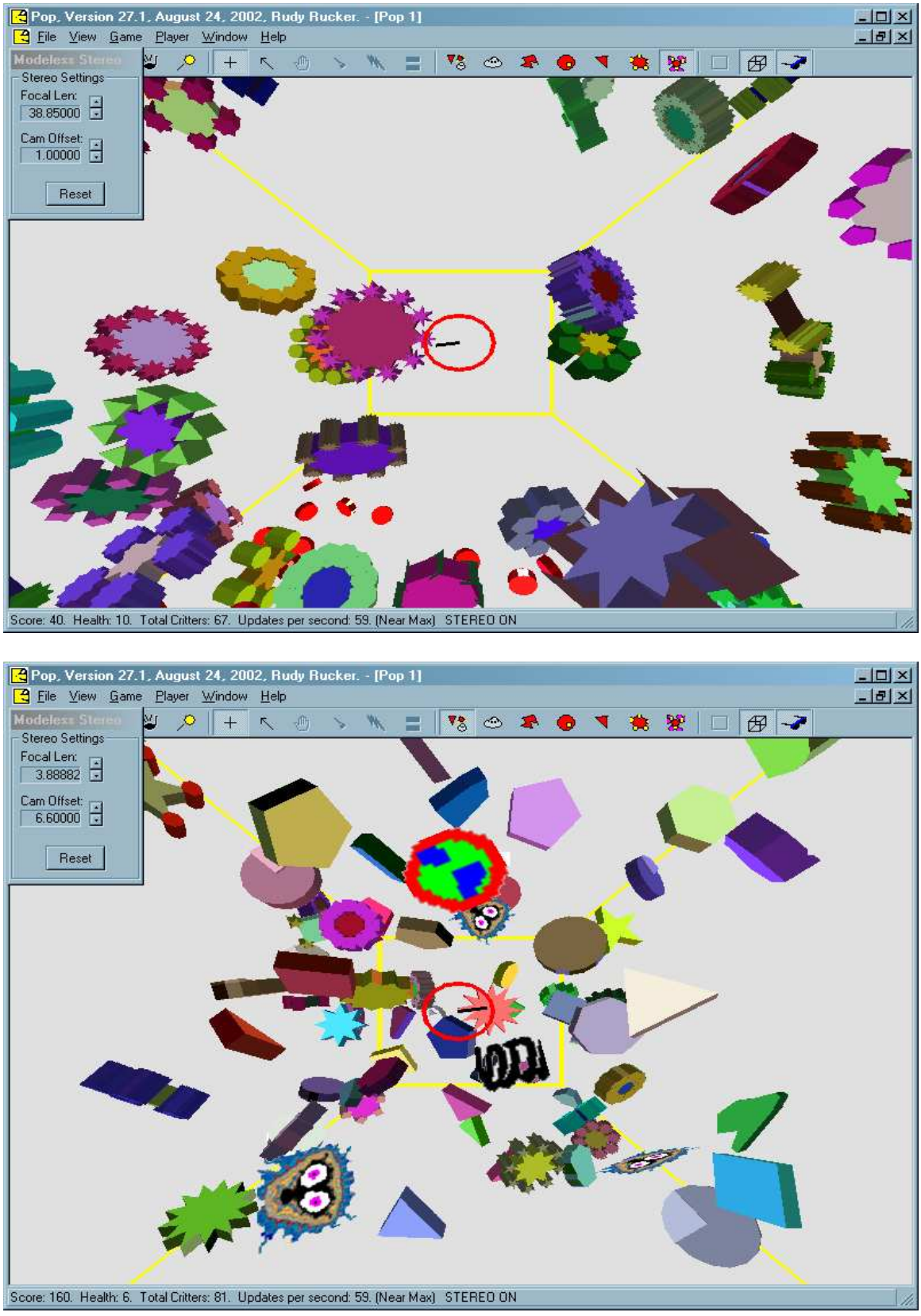
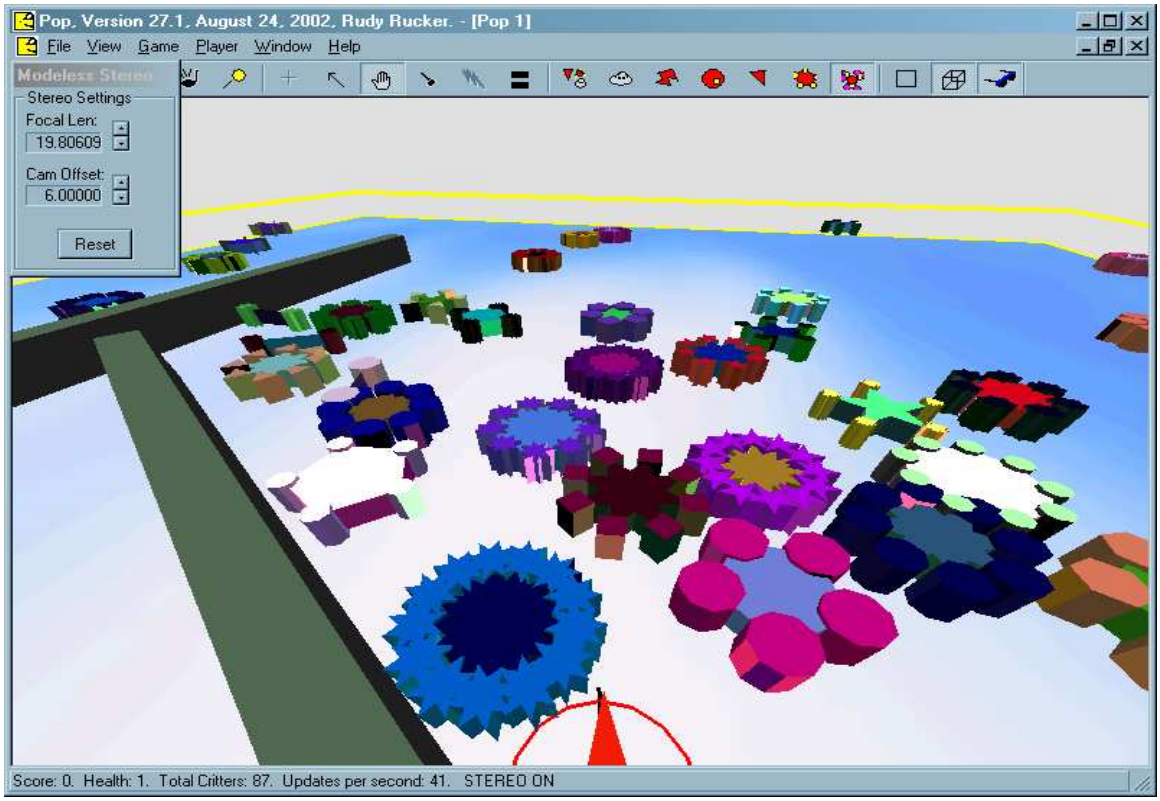


Figure 3-10b. Defender 3D in stereoscopic mode



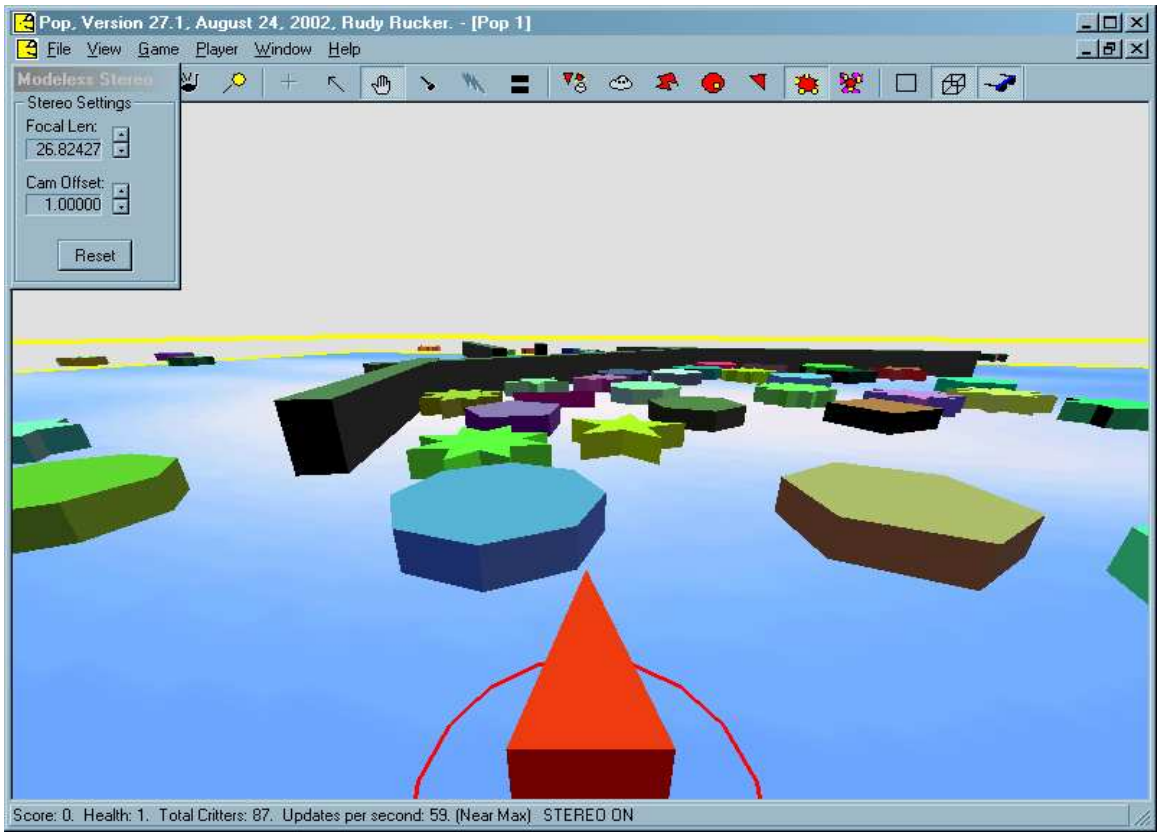
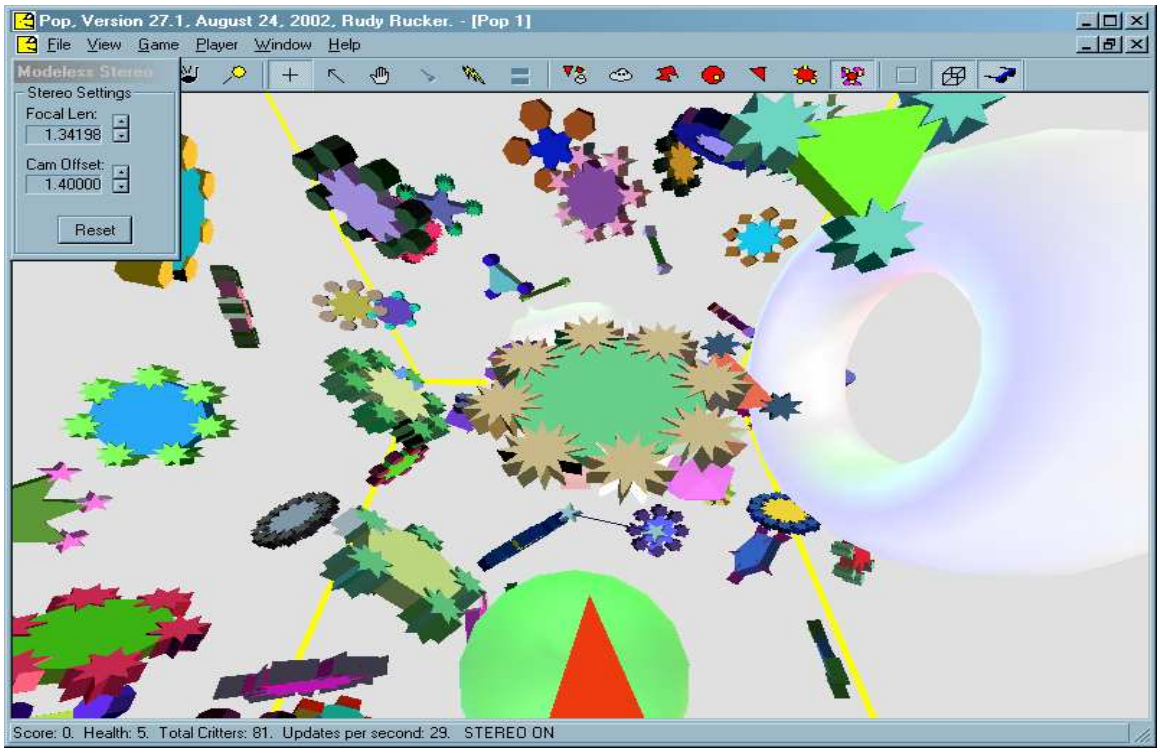


Figure 3-10c. Dam Builder in stereoscopic mode



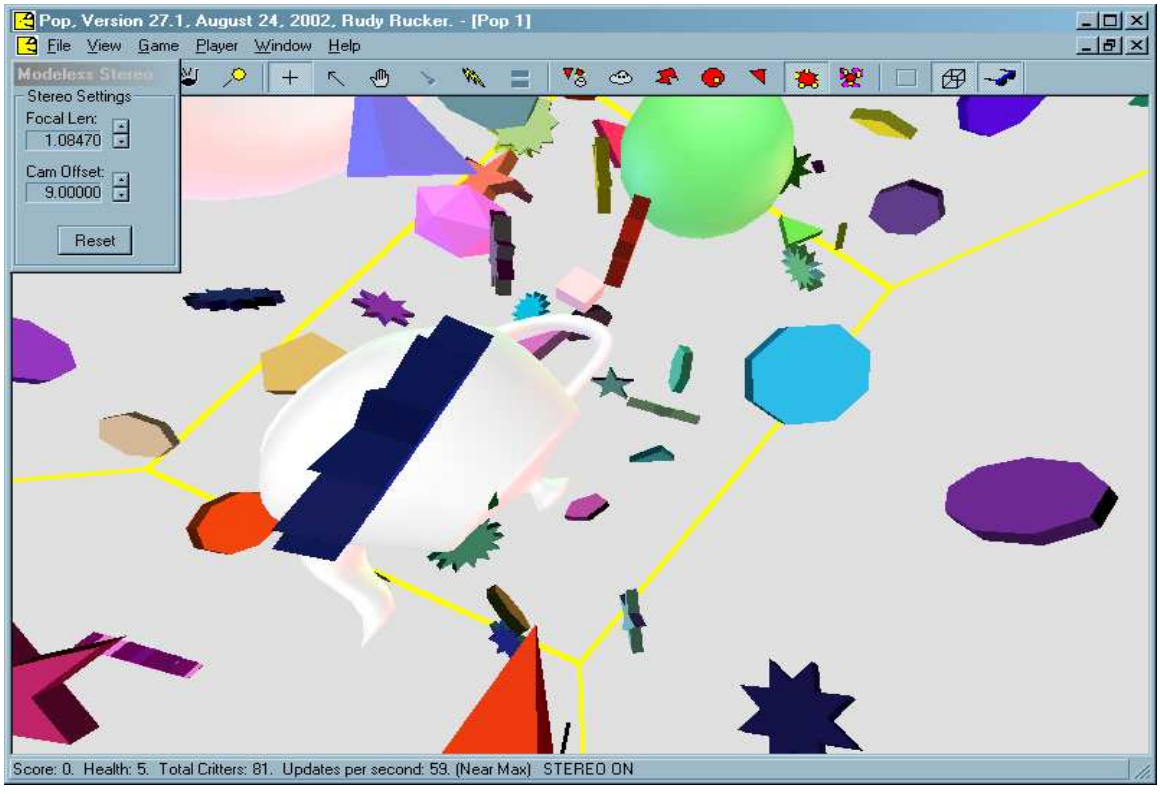


Figure 3-10d. 3D Stub in stereoscopic mode

Chapter 4

Conclusion

In conclusion, I have shown that it is possible to recreate the stereoscopic experience on non-stereoscopic consumer-class graphics hardware. This project was developed and tested using a first-generation mobile GPU from NVIDIA, GeForce2 Go. It is therefore safe to say that 3D graphics performance will only be better on desktop PC or workstations equipped with either NVIDIA or ATI based GPU hardware. Also, I have deduced that the primary reason graphics accelerators are required is to prevent display flicker when electro-stereoscopically viewing using CRTs.

The different steps that need to be taken into consideration when implementing stereoscopic software have been carefully documented and presented. I have presented a proof-of-concept stereoscopic application implemented using the GLUT library. From this, I generalized and developed the algorithms to implement stereo specific extensions for the Pop Framework.

In order to obtain the best possible OpenGL graphics rendering performance, hardware specific OpenGL extensions should be used.

References

[3D03] **3DLabs**: http://www.3dlabs.com

[A03] **Asus**: http://www.asus.com

[A99] Bob Akka: **Writing stereoscopic software for StereoGraphics systems using Microsoft Windows OpenGL**. (1999)

[B87] John Baker: **Generating images for a time-multiplexed stereoscopic computer graphics system**. Proc. Of SPIE Vol.0761, True Three-Dimensional Imaging Techniques and Display Technologies. (Jan 1987)

[E03] **ELSA**: http://www.elsa.com

[I03] **i-O Display Systems**: http://www.i-glassesstore.com

[K03] Mark Kilgard: **NVIDIA OpenGL Extensions Specifications**. http://developer.nvidia.com/view.asp?IO=nvidia_opengl_specs. (Jan 2003)

[K96] Mark Kilgard: **The OpenGL Utility Toolkit (GLUT) Programming Interface: API Version 3**. (1996)

[KX96] Mark Kilgard: **OpenGL programming for the X Window System**. Addison Wesley (1996)

[L87] Lenny Lipton: **Factors affecting "ghosting" in time-multiplexed plano-stereoscopic CRT display systems**. Proc. Of SPIE Vol.0761, True Three-Dimensional Imaging Techniques and Display Technologies. (Jan 1987)

[L97] Lenny Lipton: **Stereo-vision formats for video and computer graphics**. Proc. Of SPIE Vol.3012, Stereoscopic displays and virtual reality systems. (May 1997)

[MGS95] Jeffrey S. McVeigh, Victor S. Grinberg, M.W. Siegel: **Double Buffering Technique for Binocular Imaging in a Window**. Proc. SPIE Inter. Conf. On Stereoscopic Displays and Applications. (Feb 1995)

[N01] **NVIDIA 3D Stereo User's Guide for Detonator XP (Revision 2.0)**: http://www.nvidia.com/docs/lo/1403/supp/NVDetXP3DstereoUG_20.pdf (Nov 2001)

[N02] **NVIDIA Display Properties User's Guide (Driver Version Release 40)**: http://download.nvidia.com/windows/40.72/Detonator_40_Users_Guide.pdf (Oct 2002)

[O03] **OpenGL Extension Registry**: http://oss.sgi.com/projects/ogl-sample/registry/

[R02] Rudy Rucker: **Software Engineering and Computer Games**. Addison Wesley (2002)

[S03] **StereoGraphics Corporation**: http://www.stereographics.com

[SA02] Mark Segal, Kurt Akeley: **The OpenGL Graphics System: A Specification (Version 1.4)**. (2002)

[SDK97] StereoGraphics Corporation: **CrystalEyes Software Development Kit - A StereoGraphics Product**. (1997)

[SH97] StereoGraphics Corporation: **StereoGraphics Developers' Handbook**. (1997)

[SG03] **Octane2, Silicon Graphics Inc.**: http://www.sgi.com/workstation/octane2/

[SS95]: Scott A. Safier, M.W. Siegel: **3D-Stereoscopic X Windows**. Proc. SPIE Inter. Conf. On Stereoscopic Displays and Applications. (Feb 1995)

[WNDS99] Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner: **OpenGL Programming Guide, Third Edition, OpenGL Version 1.2**. Addison Wesley (1999)

[WT02] Andrew Woods, Stanley S.L. Tan: **Characterizing Sources of Ghosting in Time-Sequential Stereoscopic video displays**. Proc. Of SPIE Vol.4660, Stereoscopic displays and virtual reality systems. (Jan 2003)

Appendix

Code Listings

1) Pop Framework Stereo Extension

i. critterviewerstereo.h

```
//
// CritterViewerStereo.h
// CS 298
// Spring 2003
// Kenji Tan
//


#ifndef CRITTERVIEWERSTEREO_H
#define CRITTERVIEWERSTEREO_H

#include "critter.h"
#include "critterviewer.h"
#include "listener.h"



/*
 * Enumerated type for eye fields (L or R)
 */
typedef enum EyeField {
        FIELD_LEFT     = 0,
        FIELD_RIGHT = 1,
} EyeField;


/*
 * forward declaration of cStereoDialog class
 */
class cStereoDialog;


/*
 * cCritterViewerStereo class
 */
class cCritterViewerStereo : public cCritterViewer
{
DECLARE_SERIAL(cCritterViewerStereo)

private:

        /*
         * Predefined constant for comfortable stereo viewing (recommended value 0.07)
         */
        const static Real _stereoMagConst;
```

```
        /*
         * User changeable variable for strength of stereo effect
         */
        static Real _stereoMagAdj;

        /*
         * Lateral offset for modeling interocular separation (stereo viewing)
         */
        static Real _stereoCamOffset;

        /*
         * User changeable variable for balance of parallax effect (positive, zero,
              negative)
         */
        //Real _parallaxAdj;

        /*
         * Offset for creating an asymmetric frustum projection volume
         */
        //Real _frustumAsym;

        /*
         * Dimensions of the frustum projection volume. Left, Right, Bottom, Top,
         * Near, and Clip clipping planes.
         */
        static Real _left, _right, _bottom, _top, _nearclip, _farclip;

        /*
         * Scalar multiplier for the focal length parameter
         */
        static Real _focalAdj;

        /*
         * Focal length of camera. This point projects to plane of zero parallax.
         */
        static Real _focalLength;

        /*
         * Class variable for keeping track of eye field state
         */
        static EyeField _eyeField;

        /*
         * Win32 data structures for implementing Blue Line Code stereo format
         */
        static HDC _desktop;
        static HBRUSH _bluebrush;
        static HBRUSH _blackbrush;
        static RECT _rightField;
        static RECT _leftField;
        static RECT _black;

        /*
         * Reference to the cStereoDialog class
         */
        cStereoDialog* _stereoDlg;

public:
        /* Contructors */
        cCritterViewerStereo() : cCritterViewer() {};
        cCritterViewerStereo(CPopView *pview, cStereoDialog *dlg);

        /* Destructor */
        virtual ~cCritterViewerStereo();

        /* accessors */
        Real getStereoCamOffset() const { return _stereoCamOffset; }
        Real getStereoMagAdj() const { return _stereoMagAdj; }
        //Real getFrustumAsym() const { return _frustumAsym; }
        //Real getParallaxAdj() const { return _parallaxAdj; }
        Real getFocalLength() const { return _focalLength; }
```

```
        Real getFocalAdj() const { return _focalAdj; }

        /* mutators */
        void setStereoCamOffset(Real camoffset) { _stereoCamOffset = camoffset; }
        void setStereoMagAdj(Real stereomagadj) { _stereoMagAdj = stereomagadj; }
        //void setFrustumAsym(Real frustumasym) { _frustumAsym = frustumasym; }
        //void setParallaxAdj(Real parallaxadj) { _parallaxAdj = parallaxadj; }
        void setFocalLength(Real focal) { _focalLength = focal; }
        void setFocalAdj(Real focal) { _focalAdj = focal; }

        /* increment or decrement user variables */
        void incrStereoCamOffset() { _stereoMagAdj *= 1.1f;
               _stereoMagAdj = (_stereoMagAdj > 30.000f) ? 30.000f : _stereoMagAdj; };
        void decrStereoCamOffset() { _stereoMagAdj /= 1.1f;
               _stereoMagAdj = (_stereoMagAdj < 0.005f) ? 0.005f : _stereoMagAdj; };
        //void incrFrustumAsym() { _parallaxAdj *= 1.1f;
        //      _parallaxAdj = (_parallaxAdj > 30.000f) ? 30.000f : _parallaxAdj; };
        //void decrFrustumAsym() { _parallaxAdj /= 1.1f;
        //      _parallaxAdj = (_parallaxAdj < 0.005f) ? 0.005f : _parallaxAdj; };
        void incrFocalLen() { _focalLength += _nearclip;
               _focalLength = (_focalLength > _farclip) ? _farclip : _focalLength; };
        void decrFocalLen() { _focalLength -= _nearclip;
               _focalLength = (_focalLength < _nearclip) ? _nearclip : _focalLength; };

        /* special stereo mode functions */
        void renderBlueLines() const;
        void toggleFields() { _eyeField = (_eyeField==FIELD_LEFT) ? FIELD_RIGHT :
               FIELD_LEFT; };

        /* clean up function */
        void releaseDC() { ::ReleaseDC(NULL, _desktop); }


/* overloads from cCritterViewer */

        /*
         * Override setViewpoint
         */
        /*
        virtual void setViewpoint(cVector toviewer = cVector3::ZAXIS, cVector lookatpoint
               = cVector::ZEROVECTOR, BOOL trytoseewholeworld = TRUE);
        */

        /*
         * Loads Projection Matrix using glFrustum
         */
        virtual void loadProjectionMatrix() const;

        /*
         * Updates class variables between each invocation
         */
        virtual void update(CPopView *pactiveview, Real dt);

        /*
         * Serialization function
         */
        virtual void Serialize(CArchive &ar) { cCritterViewer::Serialize(ar); }
};

#endif // CRITTERVIEWERSTEREO_H
```

ii. crittervieweerstereo.cpp

```
//
// CritterViewerStereo.cpp
// CS 298
// Spring 2003
// Kenji Tan
```

```
//


#include "stdafx.h"
#include "critterviewerstereo.h"
#include "game.h"
#include "listener.h"
#include "Pop.h"
#include "PopView.h"
#include "PopDoc.h"
#include "graphicsopengl.h"
#include "graphicsmfc.h"
#include "StereoDialog.h"

IMPLEMENT_SERIAL(cCritterViewerStereo, cCritterViewer, 0)


/*
 * Initialize class variables to default values
 */
const Real cCritterViewerStereo::_stereoMagConst = 0.07f; // recommended value is 0.07


/* Experimental Code */
Real cCritterViewerStereo::_stereoMagAdj = 1.0f;
Real cCritterViewerStereo::_stereoCamOffset = 0.0f;
Real cCritterViewerStereo::_focalAdj = 1.0f;
Real cCritterViewerStereo::_focalLength = 0.0f;
//Real cCritterViewerStereo::_parallaxAdj = 1.0f;
//Real cCritterViewerStereo::_frustumAsym = 0.0f;

Real cCritterViewerStereo::_left = 0.0f;
Real cCritterViewerStereo::_right = 0.0f;
Real cCritterViewerStereo::_bottom = 0.0f;
Real cCritterViewerStereo::_top = 0.0f;
Real cCritterViewerStereo::_nearclip = 0.0f;
Real cCritterViewerStereo::_farclip = 0.0f;

EyeField cCritterViewerStereo::_eyeField = FIELD_LEFT;

HDC cCritterViewerStereo::_desktop = NULL;
HBRUSH cCritterViewerStereo::_bluebrush = ::CreateSolidBrush(RGB(0,0,255));
HBRUSH cCritterViewerStereo::_blackbrush = ::CreateSolidBrush(RGB(0,0,0));
RECT cCritterViewerStereo::_leftField = {0};
RECT cCritterViewerStereo::_rightField = {0};
RECT cCritterViewerStereo::_black = {0};




/*
 * cCritterViewerStereo(CPopView *pview)
 * Purpose: Constructor
 */
cCritterViewerStereo::cCritterViewerStereo(CPopView *pview, cStereoDialog *dlg)
: cCritterViewer(pview)
{
        /* initialize viewing dimensions */
        _left = - _aspect * _znear * tan(_fieldofviewangle/2.0f);
        _right = _aspect * _znear * tan(_fieldofviewangle/2.0f);
        _bottom = - _znear * tan(_fieldofviewangle/2.0f);
        _top = _znear * tan(_fieldofviewangle/2.0f);
        _nearclip = _znear;
        _farclip = _zfar;

        /* initialize eye field */
        _eyeField = FIELD_LEFT;

        /* set default focal length and camera offset */
        _focalLength = 0.0f;
        _stereoCamOffset = 0.0f;
```

```
        _stereoMagAdj = 1.0f;
        _focalAdj = 1.0f;

        /* set modeless dialog reference */
        _stereoDlg = dlg;

        /* set default camera offset */
        _stereoCamOffset = (_right - _left) * _stereoMagConst * _stereoMagAdj;
        if (_eyeField == FIELD_LEFT) _stereoCamOffset = -_stereoCamOffset;

        //_focalLength = _focalAdj * _farclip;

        /* set default frusutm asymmetry */
        //_frustumAsym = - _stereoCamOffset * _parallaxAdj;

        /* initialize BLC format */
        _desktop = ::GetDC(NULL);

        /*
        _bluebrush = ::CreateSolidBrush(RGB(0,0,255));
        _blackbrush = ::CreateSolidBrush(RGB(0,0,0));
        */

        ::SetRect(&_black, 0, ::GetSystemMetrics(SM_CYSCREEN)-3,
                ::GetSystemMetrics(SM_CXSCREEN),
                ::GetSystemMetrics(SM_CYSCREEN));
        ::SetRect(&_leftField, 0, ::GetSystemMetrics(SM_CYSCREEN)-3,
                ::GetSystemMetrics(SM_CXSCREEN) * 0.75f,
                ::GetSystemMetrics(SM_CYSCREEN));
        ::SetRect(&_rightField, 0, ::GetSystemMetrics(SM_CYSCREEN)-3,
                ::GetSystemMetrics(SM_CXSCREEN) * 0.25f,
                ::GetSystemMetrics(SM_CYSCREEN));

}


/*
 * ~cCritterViewerStereo()
 * Purpose: Destructor (release HDC)
 */
cCritterViewerStereo::~cCritterViewerStereo()
{
        releaseDC();
}


/*
 * setViewpoint()
 * Experimental Code
 * Purpose: Use this function to set stereo magnitude appropriately. If riding the
player, stereo magnitude
 *          can be larger. if not set stereo magnitude to be small.
 */
//void cCritterViewerStereo::setViewpoint(cVector toviewer, cVector lookatpoint, BOOL
//trytoseewholeworld)
//{
//      /* call base class function first */
//      cCritterViewer::setViewpoint(toviewer, lookatpoint, trytoseewholeworld);
//
//      /* depending on the Listener attached to this object, set the _stereoMagAdj
//              variable appropriately.
//       * if riding the viewer, then set _stereoMagAdj to be higher ~ 2.5f or more. else
//              if not riding viewer,
//       * set _stereoMagAdj to be lower.
//       */
//      /*
//      if (_plistener->GetRuntimeClass() == RUNTIME_CLASS(cListenerViewerRide)) {
//              cCritterViewerStereo::_stereoMagAdj = 2.5f;
//      } else {
//              cCritterViewerStereo::_stereoMagAdj = 0.2f;
```

```
//      }
//      */
//}


/*
 * LoadProjectionMatrix()
 * Purpose: Loads the perpective projection matrix using glFrustum
 *          If pgraphicsclass returns cGraphicsMFC, game is played in 2D ortho mode.
 *          Else, game is played in 3D stereo mode
 */
void cCritterViewerStereo::loadProjectionMatrix() const
{
        /* change GL_MATRIXMODE to GL_PROJECTION */
        pgraphics()->matrixMode(cGraphics::PROJECTION);
        /* initialize by loading Identity matrix */
        pgraphics()->loadIdentity();

        if (!_perspective ||
                pownerview()->pgraphicsclass()==(RUNTIME_CLASS(cGraphicsMFC)))
        // We don't have perspective implemented for cGraphicsMFC yet.
        // use Ortho view. ortho call takes (left, right, bottom, top, nearzclip,
        // farzclip).
        {
                cRealBox2 orthoviewrect = orthoViewRect();
                /* The call to ortho() becomes a _realpixelconverter.setRealWindow call
                        in cGraphicsMFC, replacing our old call to
                        _pgraphics->setRealBox(border()); */
                pgraphics()->ortho(orthoviewrect.lox(), orthoviewrect.hix(),
                        orthoviewrect.loy(), orthoviewrect.hiy(), _znear, _zfar);
        }
        else
        // _perspective is TRUE. frustum call takes (_left, _right,_bottom, _top,
        // _nearclip, _farclip)
        {
                //pgraphics()->frustum(_left, _right, _bottom, _top, _nearclip, _farclip);

                pgraphics()->frustum(_left - _stereoCamOffset * _nearclip / _focalLength,
                        _right - _stereoCamOffset * _nearclip / _focalLength,
                        _bottom,
                        _top,
                        _nearclip,
                        _farclip);

                /*
                pgraphics()->frustum(- _aspect * _znear * tan(_fieldofviewangle/2.0f) +
                        _frustumAsym * _nearclip / _focalLength,
                        _aspect * _znear * tan(_fieldofviewangle/2.0f) + _frustumAsym *
                        _nearclip / _focalLength,
                        - _znear * tan(_fieldofviewangle/2.0f),
                        _znear * tan(_fieldofviewangle/2.0f),
                        _znear,
                        _zfar);
                */

                /* determine if Viewer Ride mode is on. If on, then need to translate
                   viewing volume in negative z axis. Else, then no need to translate
                   viewing volume.
                 */
                if (_plistener->GetRuntimeClass() == RUNTIME_CLASS(cListenerViewerRide)) {
                        pgraphics()->translate(cVector(- _stereoCamOffset, 0.0f, - _znear
                                /*0.0f*/));
                } else {
                        pgraphics()->translate(cVector(- _stereoCamOffset, 0.0f, 0.0f));
                }

                /*
                renderBlueLines();
                toggleFields();
                */
```

```
        }

        /* change GL_MATRIXMODE to GL_MODELVIEW */
        pgraphics()->matrixMode(cGraphics::MODELVIEW);
        /* initialize by loading Identity matrix */
        pgraphics()->loadIdentity();
}


/*
 * RenderBlueLines()
 * Purpose: Renders blue lines to desktop (global) DC using Win32 functions
 */
void cCritterViewerStereo::renderBlueLines() const
{
        ::FillRect(_desktop, &_black, _blackbrush); // set black brush first

        (_eyeField == FIELD_LEFT) ? ::FillRect(_desktop, &_leftField, _bluebrush) : \
                ::FillRect(_desktop, &_rightField, _bluebrush);
}


/*
 * update(CPopView *activeview, Real dt)
 * Purpose: This function is called at every "time step" originating from
 *          CPopApp::OnIdle and cascading to CPopView::OnDraw. Class variables
 *          that are updated at each time step are: _left, _right, _bottom, _top,
 *          _nearclip, _farclip, _stereoCamOffset, _frustumAsym, _focalLength.
 *          Also, the all fields in the modeless stereo dialog box are updated from
 *          here.
 *
 *          This is done at each step because _aspect, _fieldofviewangle, _znear,
 *          _zfar may change due to the user performing zoom ins or outs.
 *          Finally, blue lines are rendered on the global DC and the eye fields
 *          are toggled to implement time-division multiplexed double buffering.
 */
void cCritterViewerStereo::update(CPopView *activeview, Real dt)
{
        static CString temp;

        /* alias for fast access */
        HWND hwnd = _stereoDlg->GetSafeHwnd();

        /* call base class function first */
        cCritterViewer::update(activeview, dt);

        /* update data structures for viewing frustum */
        _left = - _aspect * _znear * tan(_fieldofviewangle/2.0f);
        _right = _aspect * _znear * tan(_fieldofviewangle/2.0f);
        _bottom = - _znear * tan(_fieldofviewangle/2.0f);
        _top = _znear * tan(_fieldofviewangle/2.0f);
        _nearclip = _znear;
        _farclip = _zfar;

        /* stereo camera offset */
        _stereoCamOffset = (_right - _left) * _stereoMagConst * _stereoMagAdj;

        if (_eyeField == FIELD_LEFT) _stereoCamOffset = -_stereoCamOffset;

        /* similar code for frustum asym */
        //_frustumAsym = - _stereoCamOffset * _parallaxAdj;

        _focalLength = _focalAdj * _farclip;

        /* set fields in dialog box */
        /*
        temp.Format("%.5f", _left);
        SetDlgItemText(hwnd, IDC_LEFT, temp);

        temp.Format("%.5f", _right);
        SetDlgItemText(hwnd, IDC_RIGHT, temp);
```

```
        temp.Format("%.5f", _bottom);
        SetDlgItemText(hwnd, IDC_BOTTOM, temp);

        temp.Format("%.5f", _top);
        SetDlgItemText(hwnd, IDC_TOP, temp);

        temp.Format("%.5f", _nearclip);
        SetDlgItemText(hwnd, IDC_NEAR, temp);

        temp.Format("%.5f", _farclip);
        SetDlgItemText(hwnd, IDC_FAR, temp);

        temp.Format("%.5f", _fieldofviewangle);
        SetDlgItemText(hwnd, IDC_FOV, temp);

        temp.Format("%.5f", _aspect);
        SetDlgItemText(hwnd, IDC_ASPECT, temp);
        */

        //temp.Format("%.5f", _focalAdj);
        temp.Format("%.5f", _focalLength);
        SetDlgItemText(hwnd, IDC_FOCALLEN, temp);

        temp.Format("%.5f", _stereoMagAdj);
        //temp.Format("%.5f", _stereoCamOffset);
        SetDlgItemText(hwnd, IDC_CAMOFFSET, temp);

        /*
        temp.Format("%.5f", _parallaxAdj);
        //temp.Format("%.5f", _frustumAsym);
        SetDlgItemText(hwnd, IDC_FRUSTASYM, temp);
        */

        renderBlueLines();
        toggleFields();
}
```

iii. stereodialog.h

```
//
// StereoDialog.h
// CS 298
// Spring 2003
// Kenji Tan
//


#if !defined(AFX_STEREODIALOG_H__DA393ABF_84A1_42FF_BDDF_1FE7B8AF3F01__INCLUDED_)
#define AFX_STEREODIALOG_H__DA393ABF_84A1_42FF_BDDF_1FE7B8AF3F01__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// StereoDialog.h : header file
//


/* Forward declarations for CPopView and cCritterViewerStereo */
class CPopView;
class cCritterViewerStereo;


/////////////////////////////////////////////////////////////////////////////
// cStereoDialog dialog

class cStereoDialog : public CDialog
{
```

```
// Construction
public:
	//cStereoDialog(CWnd* pParent = NULL);   // standard constructor
	/* Contructor */
	cStereoDialog(CPopView* pView);
	BOOL Create(cCritterViewerStereo *pstereo);

// Dialog Data
	//{{AFX_DATA(cStereoDialog)
	enum { IDD = IDD_StereoDialog };
	// NOTE: the ClassWizard will add data members here
	//}}AFX_DATA


// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(cStereoDialog)
	protected:
	virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
	//}}AFX_VIRTUAL

// Implementation
protected:
	CPopView* m_pView;
	cCritterViewerStereo *m_pstereo;

	// Generated message map functions
	//{{AFX_MSG(cStereoDialog)
	virtual BOOL OnInitDialog();
	afx_msg void OnReset();
	afx_msg void OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar);
	//}}AFX_MSG

	afx_msg void OnCancel();
	afx_msg void OnOK();
	DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif // !defined(AFX_STEREODIALOG_H__DA393ABF_84A1_42FF_BDDF_1FE7B8AF3F01__INCLUDED_)
```

iv. stereodialog.cpp

```
//
// StereoDialog.cpp
// CS 298
// Spring 2003
// Kenji Tan
//

// StereoDialog.cpp : implementation file
//

#include "stdafx.h"
#include "pop.h"
#include "StereoDialog.h"
#include "Popview.h"
#include "critterviewerstereo.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

```
/////////////////////////////////////////////////////////////////////////////
// cStereoDialog dialog


//cStereoDialog::cStereoDialog(CWnd* pParent /*=NULL*/)
//      : CDialog(cStereoDialog::IDD, pParent)
//{
        //{{AFX_DATA_INIT(cStereoDialog)
        // NOTE: the ClassWizard will add member initialization here
        //}}AFX_DATA_INIT
//}


cStereoDialog::cStereoDialog(CPopView* pView)
{
        ASSERT("cStereoDialog(CPopView *pView)" && pView != NULL);

        m_pView = pView;

        m_pView->SetActiveWindow();
        m_pView->SetFocus();
}


BOOL cStereoDialog::Create(cCritterViewerStereo *ps)
{
        BOOL retcode = CDialog::Create(cStereoDialog::IDD);
        ASSERT("Create(cCritterViewerStereo *ps)" && ps != NULL);

        m_pstereo = ps;

        m_pView->SetActiveWindow();
        m_pView->SetFocus();

        //m_pView->SetActiveWindow(); //SetActiveView(m_pView);
        return retcode;
}


void cStereoDialog::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(cStereoDialog)
                // NOTE: the ClassWizard will add DDX and DDV calls here
        //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(cStereoDialog, CDialog)
        //{{AFX_MSG_MAP(cStereoDialog)
        ON_BN_CLICKED(IDC_Reset, OnReset)
        ON_WM_VSCROLL()
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()


/////////////////////////////////////////////////////////////////////////////
// cStereoDialog message handlers


BOOL cStereoDialog::OnInitDialog()
{
        CDialog::OnInitDialog();

        // TODO: Add extra initialization here
        CSpinButtonCtrl *pSpin = (CSpinButtonCtrl *)GetDlgItem(IDC_SPINFOCAL);
        pSpin->SetRange(0,300);
        pSpin->SetPos(150);

        pSpin = (CSpinButtonCtrl *)GetDlgItem(IDC_SPINCAM);
        pSpin->SetRange(0,150);
```

```
        pSpin->SetPos(5);

        /*
        pSpin = (CSpinButtonCtrl *)GetDlgItem(IDC_SPINFRUST);
        pSpin->SetRange(0,100);
        pSpin->SetPos(5);
        */

        return TRUE;  // return TRUE unless you set the focus to a control
                      // EXCEPTION: OCX Property Pages should return FALSE
}


void cStereoDialog::OnReset()
{
        // TODO: Add your control notification handler code here
        m_pstereo->setFocalAdj(1.0f);
        m_pstereo->setStereoMagAdj(1.0f);
        //m_pstereo->setParallaxAdj(1.0f);

        OnInitDialog();

        m_pView->SetActiveWindow();
        m_pView->SetFocus();
}


void cStereoDialog::OnCancel()
{
        // Do nothing here
}


void cStereoDialog::OnOK()
{
        // Do nothing here
}


void cStereoDialog::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
        // TODO: Add your message handler code here and/or call default

        CDialog::OnVScroll(nSBCode, nPos, pScrollBar);

        switch (pScrollBar->GetDlgCtrlID()) {
                case IDC_SPINFOCAL:
                        m_pstereo->setFocalAdj(nPos/(Real)(150.0f));
                        break;
                case IDC_SPINCAM:
                        m_pstereo->setStereoMagAdj(nPos/(Real)(5.0f));
                        break;
                /*
                case IDC_SPINFRUST:
                        m_pstereo->setParallaxAdj(nPos/(Real)(5.0f));
                        break;
                */
        }

        m_pView->SetActiveWindow();
        m_pView->SetFocus();
}
```

2) GLUT Prototype

i. glut_demo.cpp

```
//
```

```
// File: glut_demo.cpp
// CS 298
// Spring 2003
// Kenji Tan
//


#pragma comment(lib, "glut32.lib")
#pragma comment(lib, "glu32.lib")


#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include "FrameTimer.h"


/*
 * Window dimensions
 */
#define WINWIDTH 640
#define WINHEIGHT 480

/*
 * Predefined stereo magnitude constant
 */
#define STEREO_MAGNITUDE_CONSTANT 0.07

/*
 * Dimensions of viewing frustum
 */
#define LEFT_EDGE      -6.4f
#define RIGH_EDGE       6.4f
#define BOT_EDGE       -4.8f
#define TOP_EDGE        4.8f
#define NEAR_EDGE       12.5f
#define FAR_EDGE        256.0f


/*
 * Eye fields - left or right
 */
typedef enum {
        E_LEFT  = 1,
        E_RIGHT = 2,
} Eye_field;

Eye_field g_field = E_LEFT;

/*
 * boolean full screen flag
 */
bool g_fullscreen = false;

/*
 * boolean stereo flag
 */
bool g_stereo = false;

/*
 * user adjustable parameters for stereo and parallax settings
 */
GLfloat g_stereomag = 1.0f, g_focalLen = NEAR_EDGE;

/*
 * viewport dimensions
```

```
 */
int g_vport[4] = {0};

/*
 * Data structure for implementing BLC
 */
HDC g_hdc = NULL; // handle to desktop
HBRUSH g_bluebrush = CreateSolidBrush(RGB(0,0,255)); // blue brush
HBRUSH g_blakbrush = CreateSolidBrush(RGB(0,0,0)); // black brush
RECT g_right, g_left, g_black;

/*
 * Display object settings
 */
#define SOLID_OBJECT           1
#define WIREFRAME_OBJECT       2
GLuint g_list = SOLID_OBJECT;

/*
 * Display object characteristics
 */
#define ROTATION_SPEED 0.60f
GLfloat g_rotate = 0.0f;
bool g_pause = false;


/*
 * output
 * - outputs character string using GL raster position
 */
void output(GLfloat x, GLfloat y, const char *format, ...)
{
        va_list args;
        char buffer[200], *p;

        /* use stdarg.h macros for variable argument list processing */
        va_start(args, format);
                vsprintf(buffer, format, args);
        va_end(args);
        glColor3f(0.9f, 0.9f, 0.9f);
        glRasterPos2f(x, y);
        for (p=buffer; *p; p++) {
                glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, *p);
        }
}

/*
 * glInfo
 * - prints graphic card's capability
 */
void glInfo()
{
        printf("OpenGL implementation: \n");
        printf("  Version:    %s\n", glGetString(GL_VERSION));
        printf("  Vendor:     %s\n", glGetString(GL_VENDOR));
        printf("  Renderer:   %s\n", glGetString(GL_RENDERER));
        printf("  Extensions: %s\n", glGetString(GL_EXTENSIONS));
}

/*
 * renderFPS
 * - renders frames per second
 */
void renderFPS()
{
        glPushAttrib(GL_DEPTH_BUFFER_BIT | GL_LIGHTING_BIT);
        //glPushAttrib(GL_ENABLE_BIT);
        glDisable(GL_DEPTH_TEST);
        glDisable(GL_LIGHTING);

        glMatrixMode(GL_PROJECTION);
```

```
        glPushMatrix();
        glLoadIdentity();
        gluOrtho2D(0, g_vport[2], 0, g_vport[3]);

        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
        glLoadIdentity();
        output(2.0f, g_vport[3]-10.0f, "%.1f fps", GetFPS());
        glPopMatrix();

        glMatrixMode(GL_PROJECTION);
        glPopMatrix();

        glPopAttrib();
}

/*
 * renderStereoSettings
 * - renders stereo settings on GL raster
 */
void renderStereoSettings()
{
        glPushAttrib(GL_DEPTH_BUFFER_BIT | GL_LIGHTING_BIT);
        //glPushAttrib(GL_ENABLE_BIT);
        glDisable(GL_DEPTH_TEST);
        glDisable(GL_LIGHTING);

        glMatrixMode(GL_PROJECTION);
        glPushMatrix();
        glLoadIdentity();
        gluOrtho2D(0, g_vport[2], 0, g_vport[3]);

        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
        glLoadIdentity();
        output(2.0f, g_vport[3]-22.0f, "<StereoMag: %.3f> <Focal Length: %.3f>",
                g_stereomag, g_focalLen);
        glPopMatrix();

        glMatrixMode(GL_PROJECTION);
        glPopMatrix();

        glPopAttrib();
}

/*
 * renderBlueLines
 * - renders blue lines using OpenGL when in fullscreen mode
 *   otherwise using Win32 in windowed mode
 */
void renderBlueLines()
{
        if (g_fullscreen) {
                // render blue lines using OpenGL for fullscreen mode
                glPushAttrib(GL_DEPTH_BUFFER_BIT | GL_LIGHTING_BIT);
                //glPushAttrib(GL_ENABLE_BIT);
                glDisable(GL_DEPTH_TEST);
                glDisable(GL_LIGHTING);

                glMatrixMode(GL_PROJECTION);
                glPushMatrix();
                glLoadIdentity();
                gluOrtho2D(0, g_vport[2], 0, g_vport[3]);
                glMatrixMode(GL_MODELVIEW);
                glPushMatrix();
                glLoadIdentity();

                // render black rectangle first
                glColor3f(0.0f, 0.0f, 0.0f);
                glRectf(0.0f, 0.0f, g_vport[2], 3.0f);
                // render blue rectangle next
```

```
                glColor3f(0.0f, 0.0f, 1.0f);
                (g_field == E_LEFT) ? glRectf(0.0f, 0.0f, (float)(g_vport[2])*0.25f, 3.0f)
                        : \
                        glRectf(0.0f, 0.0f, (float)(g_vport[2])*0.75f, 3.0f);
                glPopMatrix();

                glMatrixMode(GL_PROJECTION);
                glPopMatrix();

                glPopAttrib();
        } else {
                // render blue lines using Win32 for windowed mode
                FillRect(g_hdc, &g_black, g_blakbrush); // set black brush first

                (g_field == E_LEFT) ? FillRect(g_hdc, &g_left, g_bluebrush) : \
                        FillRect(g_hdc, &g_right, g_bluebrush);
        }
}

/*
 * StereoProjection
 * - sets up stereo projection matrix
 * - for "nose-bleed" stereo set FocalLength to Far. This effectively
 *   causes everything in the scene to have negative parallax. The
 *   Far clipping plane will project to zero parallax.
 * - for "gentle" stereo set FocalLength to Near. The Near clipping
 *   plane will project to zero parallax.
 */
void StereoProjection (GLfloat LeftBorder, GLfloat RightBorder,
                       GLfloat BottomBorder, GLfloat TopBorder,
                       GLfloat Near, GLfloat Far,
                       GLfloat FocalLength, GLfloat CameraDistance,
                       GLfloat StereoMagnitudeAdj,
                       Eye_field WhichEyeProjection)
{
        // the X & Y axis ranges
    GLfloat XRange = RightBorder - LeftBorder;
    GLfloat YRange = TopBorder - BottomBorder;

    // midpoints of the X & Y axis ranges
    GLfloat XMidpoint = (RightBorder + LeftBorder) / 2.0f;
    GLfloat YMidpoint = (TopBorder + BottomBorder) / 2.0f;

    GLfloat StereoCameraOffset = XRange * STEREO_MAGNITUDE_CONSTANT *
        StereoMagnitudeAdj;

    StereoCameraOffset /= 2.0f;  // offset each camera by half the overall sep
    if (WhichEyeProjection == E_LEFT) // left cam has neg offset
        StereoCameraOffset = -StereoCameraOffset;

    GLfloat FrustumTop = YRange / 2.0f;
    GLfloat FrustumBottom = -YRange / 2.0f;
    GLfloat FrustumRight = (XRange / 2.0f) - StereoCameraOffset * Near / FocalLength;
    GLfloat FrustumLeft = (-XRange / 2.0f) - StereoCameraOffset * Near / FocalLength;

    // set matrix mode to GL_PROJECTION
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // create viewing frustum and set position of camera
    glFrustum(FrustumLeft, FrustumRight, FrustumBottom, FrustumTop, Near, Far);
    glTranslatef(-XMidpoint - StereoCameraOffset, -YMidpoint, -CameraDistance);

    // set matrix mode to GL_MODELVIEW
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

/*
//
// Sample Projection Code from StereoGraphics
```

```
//
void StereoProjection (double dfLeftBorder, double dfRightBorder,
                       double dfBottomBorder, double dfTopBorder,
                       double dfNearBorder, double dfFarBorder,
                       double dfTargetPlane, double dfCameraToTargetDistance,
                       double dfStereoMagnitudeAdj, double dfParallaxBalanceAdj,
                       int WhichEyeProjection)
// Perform the asymmetric frustum perspective projection for one eye's
//      subfield.
// The projection is in the direction of the negative z axis.
//
// dfLeftBorder, dfRightBorder, dfBottomBorder, dfTopBorder =
//      The coordinate range, in the z-axis target plane, which will be
//      displayed on the screen. The ratio between (dfRightBorder-dfLeftBorder)
//      and (dfTopBorder-dfBottomBorder) should equal the aspect ratio of the
//      scene. Also, dfLeftBorder must be less than dfRightBorder, and
//      dfTopBorder must be less than dfBottomBorder.
//
// dfNearBorder, dfFarBorder =
//      The z-coordinate values of the clipping planes. Since the projection is
//      in the direction of the negative z axis, dfNearBorder needs to be
//      greater than dfFarBorder. Any element with a z-coordinate value greater
//      than dfNearBorder, or less than dfFarBorder, will be clipped out.
//
// dfTargetPlane =
//      The z-coordinate value of the mid-target plane that will, by default,
//      project to zero parallax. This value should reside somewhere between
//      dfNearBorder and dfFarBorder.
//
// dfCameraToTargetDistance =
//      The distance from the center of projection to the plane of zero
//      parallax. This distance needs to be greater than the difference between
//      dfNearBorder and dfTargetPlane, in order for the near clipping plane
//      to lie in front of the camera.
//
// dfStereoMagnitudeAdj =
//      The desired magnitude of the stereo effect. 0.0 would result in no
//      stereo effect at all, 1.0 would be a good default value, 2.0 would
//      be a very strong (perhaps uncomfortable) stereo effect. This value
//      should never be less than 0.0.
//
// dfParallaxBalanceAdj =
//      The amount by which to affect the asymmetry of the projection frustum,
//      effectively adjusting the stereo parallax balance. 0.0 would result
//      in extreme negative parallax (with objects at infinite distance
//      projecting to display surface), 1.0 would be a good default value
//      (dfTargetPlane will project to zero parallax at the display surface),
//      2.0 would result in considerable positive parallax (most of the scene
//      projecting behind the display surface). This value should never be less
//      than 0.0. When this value equals 0.0, the projection frustum is
//      perfectly symmetrical.
//
// WhichEyeProjection =
//             Equals LEFT_EYE_PROJECTION or RIGHT_EYE_PROJECTION.
{
    // the X & Y axis ranges, in the target Z plane
    double dfXRange = dfRightBorder - dfLeftBorder;
    double dfYRange = dfTopBorder - dfBottomBorder;

    // midpoints of the X & Y axis ranges
    double dfXMidpoint = (dfRightBorder + dfLeftBorder) / 2.0;
    double dfYMidpoint = (dfTopBorder + dfBottomBorder) / 2.0;

    // convert clipping plane positions to distances in front of camera
    double dfCameraPlane = dfTargetPlane + dfCameraToTargetDistance;
    double dfNearClipDistance = dfCameraPlane - dfNearBorder;
    double dfFarClipDistance  = dfCameraPlane - dfFarBorder;


    // Determine the stereoscopic camera offset. A good rule of thumb is
    //      for the overall camera separation to equal about 7% of the
```

```
    //      window's X-axis range in the XY-plane of the target
    //      ("target" being mid-object or the center of interest in the
    //      scene).
    double dfStereoCameraOffset = dfXRange * STEREO_MAGNITUDE_CONSTANT  *
        dfStereoMagnitudeAdj;
    //dfStereoCameraOffset /= 2.0;  // offset each camera by half the overall sep
    if (WhichEyeProjection == LEFT_EYE_PROJECTION) // left cam has neg offset
        dfStereoCameraOffset = -dfStereoCameraOffset;

    // Determine the amount by which the projection frustum will be made
    //      asymmetrical in order to affect a nice parallax balance between
    //      negative parallax (popping out of the display) and positive
    //      parallax (residing behind the display surface). With no frustum
    //      asymmetry, everything resides in negative parallax.
    double dfFrustumAsymmetry = -dfStereoCameraOffset * dfParallaxBalanceAdj;

    // determine the shape of the projection frustum; note that if
    //      FrustumRight doesn't equal -FrustumLeft, that makes this an
    //      asymmetric frustum projection
    double FrustumTop = dfYRange / 2.0;
    double FrustumBottom = -dfYRange / 2.0;
    double FrustumRight = (dfXRange / 2.0) + dfFrustumAsymmetry;
    double FrustumLeft = (-dfXRange / 2.0) + dfFrustumAsymmetry;

    // since glFrustum() maps the window borders based on the near clipping
    //      plane rather than the target plane, X and Y range values need
    //      to be adjusted by the ratio of those two distances
    double n_over_d = dfNearClipDistance / dfCameraToTargetDistance;
    dfXRange *= n_over_d;
    dfYRange *= n_over_d;
    dfFrustumAsymmetry *= n_over_d;

    // glMatrixMode(GL_PROJECTION) needs to have been called already
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();  // obtain a vanilla trans matrix to modify

    // this matrix transformation performs the actual persp projection
    glFrustum(FrustumLeft, FrustumRight, FrustumBottom, FrustumTop,
        dfNearClipDistance, dfFarClipDistance);
    //glTranslatef(-dfXMidpoint-dfStereoCameraOffset, -dfYMidpoint, -dfCameraPlane);
    glTranslatef(0.0f-dfStereoCameraOffset, 0.0f, -dfCameraPlane);
    //glTranslatef(0.0f, 0.0f, -dfCameraPlane);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    // this matrix transformation does two things: Translates the stereo
    //      camera towards the left (left camera) or the right (right
    //      camera), and also offsets the entire geometry such that the
    //      virtual camera is at (0.0, 0.0, 0.0) where glFrustum() expects
    //      it to be
    //glTranslated(-dfXMidpoint - dfStereoCameraOffset, -dfYMidpoint, -dfCameraPlane);
}
*/

/*
 * update
 * - updates rotation variable
 */
void update()
{
       if (!g_pause) {
              g_rotate += ROTATION_SPEED;
              g_rotate = (g_rotate>=360.0f) ? 0.0f : g_rotate;
       }
}

/*
 * display
 * - main display routine
 */
void display(GLvoid)
```

```
{
	glDrawBuffer(GL_BACK);
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	// flush out remaining commands in OpenGL hardware
	glFlush();
	// start frame counter for current frame
	PingFrameCounter();

	glPushMatrix();

	// sets up the projection for stereo
	if (!g_stereo) {
		StereoProjection(LEFT_EDGE, RIGH_EDGE, BOT_EDGE, TOP_EDGE, NEAR_EDGE,
			FAR_EDGE,
			g_focalLen, 18.0f,
			0.0f,
			E_LEFT);
	} else {
		(g_field == E_LEFT) ? \
			StereoProjection(LEFT_EDGE, RIGH_EDGE, BOT_EDGE, TOP_EDGE,
				NEAR_EDGE, FAR_EDGE,
				g_focalLen, 18.0f,
				g_stereomag,
				E_LEFT) : \
			StereoProjection(LEFT_EDGE, RIGH_EDGE, BOT_EDGE, TOP_EDGE,
				NEAR_EDGE, FAR_EDGE,
				g_focalLen, 18.0f,
				g_stereomag,
				E_RIGHT);
	}

	glPushMatrix();
		// move object closer toward camera away from focal point
		// this can be set to any arbitrary value suit according to
		// taste
		//glTranslatef(0.0f, 0.0f, 1.5f);
		glRotatef(g_rotate, 1.0f, 0.0f, 0.0f);
		glRotatef(g_rotate, 0.0f, 1.0f, 0.0f);
		glRotatef(g_rotate, 0.0f, 0.0f, 1.0f);

		// tilt camera view a little for easy viewing
		glRotatef(-10.0f, 1.0f, 0.0f, 0.0f);
		glCallList(g_list);
	glPopMatrix();

	glPopMatrix();

	// render blue lines and stereo settings only if in stereo mode
	if (g_stereo) {
		renderBlueLines();
		renderStereoSettings();
	}


	// toggle eye fields
	g_field = (g_field == E_RIGHT) ? E_LEFT : E_RIGHT;


	// flush out command buffer
	glFlush();
	update();

	// render frame counter frames per second
	renderFPS();

	// swap back buffer to front
	glutSwapBuffers();
	glutPostRedisplay();
}
```

```
/*
 * keyboard
 * - keyboard handling routine
 */
void keyboard(unsigned char key, int x, int y)
{
        static bool list = true;
        static bool lighting = true;
        static bool smoothshade = false;

        switch (key) {
                case 27:        // exit
                        exit(0);
                        break;

                case 'l':       // lighting
                        lighting = !lighting;
                        (lighting) ? glEnable(GL_LIGHTING) : glDisable(GL_LIGHTING);
                        break;

                case ' ':       // change display list
                        list = !list;
                        g_list = (list) ? SOLID_OBJECT : WIREFRAME_OBJECT;
                        break;

                case 's':       // stereo
                        g_stereo = !g_stereo;
                        break;

                case 'r':       // reset stereo settings
                        g_stereomag = 1.0f;
                        g_focalLen = NEAR_EDGE;
                        break;

                case 't':       // toggle shade model
                        smoothshade = !smoothshade;
                        (smoothshade) ? glShadeModel(GL_SMOOTH) : glShadeModel(GL_FLAT);
                        break;

                case 'p':       // toggle pause
                        g_pause = !g_pause;
                        break;

                default:
                        break;
        }

        glutPostRedisplay();
}


/*
 * Predefined macros for parallax and stereo settings
 */
#define PARALLAX_MAX   30.0000f
#define PARALLAX_MIN   0.01f
#define STEREO_MAX     30.0000f
#define STEREO_MIN     0.01f


/*
 * specialkey
 * - keyboard handling routine
 */
void specialkey(int key, int x, int y)
{
        switch (key) {
                case GLUT_KEY_UP:
                        g_focalLen += NEAR_EDGE;
                        // clamp value
                        g_focalLen = (g_focalLen > FAR_EDGE) ? FAR_EDGE : g_focalLen;
```

```
                        break;

                case GLUT_KEY_DOWN:
                        g_focalLen -= NEAR_EDGE;
                        // clamp value
                        g_focalLen = (g_focalLen < NEAR_EDGE) ? NEAR_EDGE : g_focalLen;
                        break;

                case GLUT_KEY_LEFT:
                        g_stereomag /= 1.1f;
                        // clamp value
                        g_stereomag = (g_stereomag < STEREO_MIN) ? STEREO_MIN :
                                g_stereomag;
                        break;

                case GLUT_KEY_RIGHT:
                        g_stereomag *= 1.1f;
                        // clamp value
                        g_stereomag = (g_stereomag > STEREO_MAX) ? STEREO_MAX :
                                g_stereomag;
                        break;

                case GLUT_KEY_F1:
                        g_fullscreen = !g_fullscreen;
                        if (g_fullscreen) {
                                glutFullScreen();
                        } else {
                                glutPositionWindow(20, 60);
                                glutReshapeWindow(WINWIDTH, WINHEIGHT);
                        }
                        break;

                default:
                        break;
        }

        glutPostRedisplay();
}

/*
 * reshape
 * - handles WM_SIZE message
 */
void reshape(int width, int height)
{
        // set up viewport
        glViewport(0, 0, width, height);

        // retrieve client window dimensions
        glGetIntegerv(GL_VIEWPORT, g_vport);
}

/*
 * setupMaterial
 * - sets up material property and display lists
 */
void setupMaterial()
{
        GLfloat brassAmbient[] = {0.33f, 0.22f, 0.03f, 1.0f},
                        brassDiffuse[] = {0.78f, 0.57f, 0.11f, 1.0f},
                        brassSpecular[] = {0.99f, 0.91f, 0.81f, 1.0f},
                        brassShininess = 27.8f;

        glNewList(SOLID_OBJECT, GL_COMPILE);
                glColor3f(0.4f, 0.4f, 0.2f);
                glMaterialfv(GL_FRONT, GL_AMBIENT, brassAmbient);
                glMaterialfv(GL_FRONT, GL_DIFFUSE, brassDiffuse);
                glMaterialfv(GL_FRONT, GL_SPECULAR, brassSpecular);
                glMaterialf(GL_FRONT, GL_SHININESS, brassShininess);
                //glutSolidCube(2.7f);
                //glutSolidTeapot(2.7f);
```

```
                glutSolidDodecahedron();
        glEndList();

        glNewList(WIREFRAME_OBJECT, GL_COMPILE);
        glColor3f(0.4f, 0.4f, 0.2f);
                glMaterialfv(GL_FRONT, GL_AMBIENT, brassAmbient);
                glMaterialfv(GL_FRONT, GL_DIFFUSE, brassDiffuse);
                glMaterialfv(GL_FRONT, GL_SPECULAR, brassSpecular);
                glMaterialf(GL_FRONT, GL_SHININESS, brassShininess);
                //glutWireCube(2.7f);
                //glutWireTeapot(2.7f);
                glutWireDodecahedron();
        glEndList();
}

/*
 * setupLighting
 * - sets up OpenGL lighting parameters
 */
void setupLighting()
{
        GLfloat light_position[] = {0.0f, 0.0f, 1.0f, 0.0f},
                        white_light[] = {1.0f, 1.0f, 1.0f, 1.0f};

        glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_FALSE);
        glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_FALSE);
        glLightModelfv(GL_LIGHT_MODEL_AMBIENT, white_light);
        glLightfv(GL_LIGHT0, GL_POSITION, light_position);
        glLightfv(GL_LIGHT0, GL_DIFFUSE, white_light);
        glLightfv(GL_LIGHT0, GL_SPECULAR, white_light);
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
}

/*
 * setupGeometry
 * - manually set up cube geometry without using GLUT
 *   for benchmarking purposes
 */
void setupGeometry()
{
        glNewList(SOLID_OBJECT, GL_COMPILE);
        glBegin(GL_QUADS);
                glColor3f(0.0f, 1.0f, 0.0f);
                glVertex3f(-1.0f, -1.0f,  1.0f);
                glVertex3f( 1.0f, -1.0f,  1.0f);
                glVertex3f( 1.0f,  1.0f,  1.0f);
                glVertex3f(-1.0f,  1.0f,  1.0f);

                glColor3f(0.0, 0.0f, 1.0f);
                glVertex3f(-1.0f, -1.0f, -1.0f);
                glVertex3f(-1.0f,  1.0f, -1.0f);
                glVertex3f( 1.0f,  1.0f, -1.0f);
                glVertex3f( 1.0f, -1.0f, -1.0f);

                glColor3f(1.0f, 0.0f, 0.0f);
                glVertex3f(-1.0f,  1.0f, -1.0f);
                glVertex3f(-1.0f,  1.0f,  1.0f);
                glVertex3f( 1.0f,  1.0f,  1.0f);
                glVertex3f( 1.0f,  1.0f, -1.0f);

                glColor3f(1.0f, 1.0f, 0.0f);
                glVertex3f(-1.0f, -1.0f, -1.0f);
                glVertex3f( 1.0f, -1.0f, -1.0f);
                glVertex3f( 1.0f, -1.0f,  1.0f);
                glVertex3f(-1.0f, -1.0f,  1.0f);

                glColor3f(0.0f, 1.0f, 1.0f);
                glVertex3f( 1.0f, -1.0f, -1.0f);
                glVertex3f( 1.0f,  1.0f, -1.0f);
                glVertex3f( 1.0f,  1.0f,  1.0f);
```

```
                glVertex3f( 1.0f, -1.0f,  1.0f);

                glColor3f(1.0f, 0.0f, 1.0f);
                glVertex3f(-1.0f, -1.0f, -1.0f);
                glVertex3f(-1.0f, -1.0f,  1.0f);
                glVertex3f(-1.0f,  1.0f,  1.0f);
                glVertex3f(-1.0f,  1.0f, -1.0f);
        glEnd();
        glEndList();
}

/*
 * setupBLC
 * - implement BLC format
 */
void setupBLC()
{
        // get DC to global desktop
        g_hdc = GetDC(NULL);

        // set up Win32 data structures for implementing BLC
        SetRect(&g_black, 0, GetSystemMetrics(SM_CYSCREEN)-3,
                GetSystemMetrics(SM_CXSCREEN),
                GetSystemMetrics(SM_CYSCREEN));
        SetRect(&g_right, 0, GetSystemMetrics(SM_CYSCREEN)-3,
                GetSystemMetrics(SM_CXSCREEN) * 0.25f,
                GetSystemMetrics(SM_CYSCREEN));
        SetRect(&g_left, 0, GetSystemMetrics(SM_CYSCREEN)-3,
                GetSystemMetrics(SM_CXSCREEN) * 0.75f,
                GetSystemMetrics(SM_CYSCREEN));
}

/*
 * init
 * - initialize GL system
 */
void init()
{
        glClearColor(0.4f, 0.4f, 0.4f, 1.0f);
        glPointSize(2.0f);
        glLineWidth(2.5f);

        glCullFace(GL_BACK);
        glFrontFace(GL_CCW);
        //glEnable(GL_CULL_FACE);
        glEnable(GL_DEPTH_TEST);
        //glDepthFunc(GL_ALWAYS);
        //glDepthMask(GL_FALSE);
        glDepthFunc(GL_LESS);
        //glDisable(GL_DEPTH_TEST);

        //glShadeModel(GL_SMOOTH);
        glShadeModel(GL_FLAT);
        glPolygonMode(GL_FRONT, GL_FILL);
        //glPolygonMode(GL_FRONT, GL_LINE);
        glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_FASTEST);

        // Disable all graphically expensive operations
        glDisable(GL_DITHER);
        glDisable(GL_BLEND);
        glDisable(GL_ALPHA_TEST);
        glDisable(GL_STENCIL_TEST);
        glDisable(GL_COLOR_MATERIAL);
        glDisable(GL_NORMALIZE);

        // Antialiasing mode - huge performance hit
        //glEnable(GL_BLEND);
        //glEnable(GL_LINE_SMOOTH);
        //glEnable(GL_POLYGON_SMOOTH);
        //glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        //glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
```

```
        setupBLC();
        setupLighting();
        setupMaterial();
        //setupGeometry();
        StartFrameCounter(); // init frame counter
}

/*
 * release
 * - release hDC to desktop
 */
void release()
{
        glInfo();
        printf("\n*** Releasing HDC to global desktop. *** \n\n");
        if (ReleaseDC(NULL, g_hdc)) {
                printf("Release success!\n");
        } else {
                printf("Release fail!\n");
        }
}

/*
 * main
 * - function main
 */
int main(int argc, char* argv[])
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
        glutInitWindowSize(WINWIDTH, WINHEIGHT);
        glutInitWindowPosition(20, 60);
        glutCreateWindow("Glut Stereo Prototype");

        glutReshapeFunc(reshape);
        glutDisplayFunc(display);

        glutKeyboardFunc(keyboard);
        glutSpecialFunc(specialkey);

        init();
        atexit(release);

        glutMainLoop();

        return 0;
}
```

ii. FrameTimer.h

```
//
// FrameTimer.h - A simple frame-rate utility
// CS 298
// Spring 2003
// Kenji Tan
// Original code by Evan Hart - ATI Research - January 6,1998
//
// This set of functions averages the frame
// times to keep a smoothed frame-rate.
// WARNING: it is not reentrant
//

#ifndef FRAME_TIMER_H
#define FRAME_TIMER_H

#include <windows.h>
```

```
// number of frames used in smoothing
#define FRAME_HISTORY 10 /* 30 */


// start keeping the counter
void StartFrameCounter();

// update the counter for this frame
void PingFrameCounter();

// get the current smoothed FPS (rolling average of FRAME_HISTORY frames)
float GetFPS();


#endif
```

ii. FrameTimer.cpp

```
//
// FrameTimer.cpp
// CS 298
// Spring 2003
// Kenji Tan
//

#include "FrameTimer.h"


#define SPEEDFREAK
#ifdef SPEEDFREAK
# define double_t float
#else
# define double_t double
#endif

/*
 * Global Data
 */
LARGE_INTEGER last;
double TimerResolution;
double_t frameTimes[FRAME_HISTORY];
int frameLocation;
unsigned int framesProcessed;


/*
 * StartTimer - (private) Performance Counter based timing function
 */
static void StartTimer()
{
        LARGE_INTEGER num;

        QueryPerformanceFrequency(&num);
        TimerResolution = (double)num.HighPart * 4294967296.0;
        TimerResolution += (double)num.LowPart;
        TimerResolution = 1.0f/TimerResolution;
        QueryPerformanceCounter(&last);
}

/*
 * PingTimer - (private) Performance Counter based timing function
 */
static double_t PingTimer()
{
        LARGE_INTEGER t;
        double_t elapsed;

        QueryPerformanceCounter(&t);
```

```
        elapsed = ((double_t)t.HighPart - (double_t)last.HighPart);
        elapsed += t.LowPart - last.LowPart;
        elapsed *= (double_t)TimerResolution;

        last.HighPart = t.HighPart;
        last.LowPart = t.LowPart;

        return elapsed;
}

/*
 * StartFrameCounter - (public) Starts the frame counter
 */
void StartFrameCounter()
{
        int i;

        for (i=0; i<FRAME_HISTORY; i++) {
                frameTimes[i] = 0.0f;
        }
        StartTimer();
        frameLocation = 0;
        framesProcessed = 0;
}

/*
 * PingFrameCounter - (public) update frame counter for the current frame
 */
void PingFrameCounter()
{
        frameTimes[frameLocation] = PingTimer();
        frameLocation = (frameLocation+1)%FRAME_HISTORY;
        framesProcessed++;
}

/*
 * GetFPS - (public) update the frame counter for the current frame and get fps
 *      frames per second
 * returns a rolling average of FRAME_HISTORY frames fps
 */
float GetFPS()
{
        int i, top;
        float total=0.0f;

        top = (FRAME_HISTORY>framesProcessed) ? framesProcessed : FRAME_HISTORY;

        for (i=0; i<top; i++)
                total += (float)frameTimes[i];

        total /= (float)FRAME_HISTORY;

        total = 1.0f/total;
        return (float)total;
}
```