University of California
Los Angeles

Virtual Reality Simulation of the Ophthalmoscopic Examination

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Daren Andrew Lee

1997

The thesis of Daren Andrew Lee is approved.

Alfonso F. Cardenas

Jack W. Carlyle

Donna L. Vredevoe

Mary A. Woo

Daniel J. Valentino

Walter J. Karplus, Committee Chair

University of California, Los Angeles

1997

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am grateful to my advisor, Dr. Walter Karplus, for his continued support and vision. Special thanks are extended to Dr. Daniel Valentino for his valuable input and for access to the computer facilities in the Medical Image Processing Laboratory. I would also like to thank Dr. Donna Vredevoe for her commitment to the project, and Professor Mary Woo and James Kimmick for their guidance and patience at every stage of the project. Also deserving thanks are Dr. Jack Carlyle and Dr. Alfonso Cardenas for serving on my committee and for reviewing this thesis.

Financial support for this research was provided by the Chancellorial Commitment to the School of Nursing at UCLA and the National Science Foundation under grant number IRI-9527128.

Abstract of the Thesis

Virtual Reality Simulation of the Ophthalmoscopic Examination

by

Daren Andrew Lee

Master of Science in Computer Science

University of California, Los Angeles, 1997

Professor Walter J. Karplus, Chair

Health care professionals perform ophthalmoscopic examinations to detect pathologies of the eye as well as other diseases, such as high-blood pressure and diabetes. The ophthalmoscopic examination is given using an ophthalmoscope, a hand-held instrument comprised of an adjustable lens and a focused beam of light. The difficulty of the procedure lies in positioning the ophthalmoscope accurately and correctly identifying the disease symptoms—skills that improve with experience.

To improve and accelerate the training of the student, this thesis develops the *Virtual Ophthalmoscopic Examination* simulation program, a three-dimensional real-time computer simulation of the ophthalmoscopic procedure using virtual reality. By navigating and manipulating the virtual ophthalmoscope in the simulation environment, the student learns how to properly position the instrument. Unlike other training aids which use photographic slides that show the full retina of the eye, the *Virtual Ophthalmoscopic Examination* program accurately simulates a more limited view of a three-dimensional eye. By increasing the realism

of the training, the transition to examining real patients will become less difficult. The program was evaluated by graduate nursing students was shown to be a promising training aid.

CHAPTER 1

Introduction

The ophthalmoscopic examination is a critical test given to many patients. In addition to ophthalmological pathologies such as retinal detachments and glaucoma, other diseases often develop symptoms visible in the retina, such as high-blood pressure and diabetes. On average, however, even the well-trained physician can only identify fifty percent of the pathologies [1].

The ophthalmoscopic examination is given using an ophthalmoscope, a hand-held instrument comprised of an adjustable lens and a focused beam of light. The two main difficulties in the procedure are positioning the ophthalmoscope accurately and correctly identifying the signs of disease, which can be very subtle. These skills are improved upon with experience. Moreover, because the spot light of the ophthalmoscope must be placed centimeters away from the eye, it can be highly uncomfortable for the patient when inexperienced students practice on their eyes. One teaching methods employs styrofoam heads to simulate real patients, but the two-dimensional slides of retinal images do not provide a realistic scenario for the students. In addition, the heads are static and the eyes do not blink, factors that make the examination more difficult on real patients.

To facilitate and accelerate the training of the student, this thesis develops the *Virtual Ophthalmoscopic Examination* simulation program, a three-dimensional computer simulation of the ophthalmoscopic procedure using virtual reality. This research was jointly funded by the Department of Computer Science and the School of Nursing at UCLA, while the computer equipment was provided by the Department of Radiological Sciences. By navigating and manipulating the virtual ophthalmoscope in the simulation environment, the student learns how to properly position the instrument and to identify the symptoms of numerous diseases. Using the virtual reality system as a training aid not only provides a realistic three-dimensional view of the eye but also eliminates the difficulty of performing the examination on real patients. In addition, unlike the traditional training with the styrofoam head where the instructor must convey the process through language, the virtual technology allows both the student and the instructor to view the simulation environment at the same time. Thus, the student learns effectively in a hands-on, dynamic, and realistic environment.

1.1 Examination Procedure

This section gives a brief overview of the ophthalmoscopic examination procedure. First, the student must learn how to position the ophthalmoscope at a distance of 4-6 inches from the eye to obtain the red-reflex, a phenomenon in which the normally dark inside of the iris becomes bright red. At certain angles, the light from the instrument is reflected off the retina causing the inside of the iris to flash red. Once this red-reflex behavior is seen, the student must then approach the eye of the patient, using the red-reflex landmark as a guide into the eye. As the student moves close enough, the back of the eye, called the fundus, will become visible. The steps involved in performing an ophthalmoscopic examination are:

1. Prepare instrument and room

(a) Check ophthalmoscope to ensure that the light is intense and bright.

(b) Dim the lights until the examination room is semi-dark.

2. Prepare the patient

   (a) If necessary, dilate eyes 15 minutes prior to examination.

   (b) If necessary, remove glasses from patient and seat patient comfortably.

   (c) Instruct patient to fixate on a target in the room and not to look at the ophthalmoscopic light beam.

3. Examine the fundus

   (a) Obtain red-reflex from 15° outward from patient's line of vision.

   (b) Maintain red-reflex and approach patient until fundus is visible.

   (c) Examine the disc, vessels, general background, and macula.

1.2 Previous Research

The use of virtual reality technology in the field of medicine has grown substantially over the past few years [2]. Although the application areas are quite diverse, a majority of the programs simulate invasive medical procedures. Because of the difficulty involved in teaching and learning invasive surgery, the virtual reality simulation is a powerful tool [3]. Instead of observing experienced surgeons and using cadavers, the student can practice on virtual patients which exhibit physical and physiological characteristics of living humans.

For example, Ziegler *et al.* have developed a virtual reality training program that simulates arthroscopic surgery of the knee joint. Using a synthetic knee, the exploratory probe is tracked during the procedure while the program simulates the corresponding view of the knee joint [4].

Similar to the surgical simulators, most of the virtual reality programs related to the eye have concentrated on surgical procedures or the optical properties of the eye. For example, Hunter *et al.* have developed an ophthalmic virtual environment that interfaces with a teleoperated microsurgical robot to perform surgery on the eye. The virtual environment for simulating the eye surgery provides both visual and tactile feedback for the user [5]. Both the surgical simulators developed by Ziegler and Hunter allow inexperienced surgeons to learn different procedures and experienced surgeons to sharpen their skills or experiment with rarely used or new procedures.

Mostafawy *et al.* have developed the Virtual Eye, a system that simulates human vision. Using ray tracing techniques, the program visualizes the image that is projected onto the retina under various physiological phenomena, such as myopia, hyperopia, or astigmatism. This type of human vision simulator has many potential clinical applications. For example, the effect of new lens combinations for eye glasses can be tested or laser surgical techniques can be planned and evaluated [6].

Most of these surgical and visualization simulators are geared for highly-trained, specialized medical professionals. Few programs have been developed that focus on the more common medical procedures that are vital to the health care of the general population. Therefore, improving the education of nurses would prove to be beneficial since their main responsibility is to oversee and supervise the day-to-day care of patients.

For example, the State University of New York at Plattsburgh collaborated with High Techsplanations, a computer firm that builds medical applications, to develop a virtual reality simulation that teaches nursing students how to insert intravenous needles. Using a specialized input sensor that gives tactile feedback, the student must place the needle at the correct position and penetrate the skin of the virtual patient. Along with various verbal cues that are given during the procedure to help the student, six different scenarios have been developed to test clinical decision making skills. This virtual reality prototype will soon be incorporated into the nursing curriculum at Plattsburgh and its effect carefully monitored [7].

The *Virtual Ophthalmoscopic Examination* program described in this thesis has been developed to help enhance the general education of nursing students. Because the ophthalmoscopic examination shares many characteristics of surgical procedures, it is a prime candidate for simulation. For example, performing the procedure on real patients is difficult for beginners and uncomfortable for patients. Moreover, experiencing the procedure first hand is the only suitable method of learning. Simulating the ophthalmoscopic examination, then, not only improves the education of the nursing students but also has the potential to help improve the quality of health care.

1.3 Objectives

The goal of this research is to provide an interactive training aid that allows users to conduct a virtual ophthalmoscopic examination in real-time. To be effective, the program should simulate the experience of a real examination procedure as closely as possible. Hence, an accurate model of the red-reflex behavior and a detailed visual representation of the eye are needed.

A primary objective is to teach the users the proper instrument positioning, which is the most difficult part of the examination procedure. Although a more conventional system that uses pop-up windows and a 2D mouse could have been used, such as VRML, it does not emphasize the physical aspect of the procedure. In a virtual reality environment, on the other hand, the interface is geared more to the physical movements of the user. For example, to move in the virtual environment, the user must physically walk to the destination rather than simply pressing a button to tell the program where to move. Hence, a virtual reality environment can simulate the physical aspects of the examination procedure more realistically.

Another key objective is to give students a more realistic view of the fundus. Although there are many good textbooks with pages of fundi images, these images are usually of fully dilated patients, shot with a special camera that captures the entire fundus. In reality, only a part of the fundus can be seen at a time during an exam. Hence, a student accustomed to viewing the entire fundus may not immediately pick out the pathology in a real patient. Therefore, by limiting the user's view and presenting a more realistic view of the fundus, the virtual reality simulation can help sharpen the student's diagnostic skills.

1.4 Organization

This thesis is divided into five chapters and includes two appendices. Chapter 2 presents an overview of the program's functionality and its overall structure. Chapter 3 describes the implementation details of the *Virtual Ophthalmoscopic Examination* program. Chapter 4 presents the results of the user survey, followed by the conclusions in chapter 5. A complete user manual for the program can be found in appendix A while appendix B contains the source code.

CHAPTER 2

Program Overview

In this chapter, we present an overview of the general functionality of the *Virtual Ophthalmoscopic Examination* program, the overall structure of the simulation, and discuss the design decisions made to improve performance.

2.1 Program Description

By using a *Logitech 3D Mouse* and a pair of *StereoGraphics CrystalEyes*, the user can conduct an ophthalmoscopic examination in a virtual environment. The 3D mouse gives the user the freedom to move anywhere in the three-dimensional space and the special shutter glasses produce a stereo, three-dimensional image that projects from the computer monitor.

At startup, the program positions the user a few feet from the virtual patient. The user then must maneuver around the virtual patient until the red-reflex behavior is seen. Maintaining the red-reflex in the virtual patient's eye, the user then must move forward until the fundus is visible. Once the interior of the eye is visible, the user can examine the disc, vessels, general background, and macula to make a diagnosis. The *Virtual Ophthalmoscopic Examination* contains the following features:

- Virtual ophthalmoscope with spotlight

- Virtual patient with texture mapped fundi images
- Red-reflex behavior for virtual patient
- Blinking
- Iris constriction and dilation
- Eye and head movement

Navigation in the simulation is analogous to the real world. Since the current viewpoint in the simulation is attached to the mouse sensor, if the user wishes to move forward, the user must simply move the 3D mouse forward. Likewise, if the user wishes to rotate the view to the right, the user can simply rotate the mouse sensor clockwise.

2.2 Program Structure

The underlying architecture of the *Virtual Ophthalmoscopic Examination* program is structured around many of the features of *Sense8's WorldToolKit* (*WTK*), a *C* graphics library for developing virtual worlds. These features include task functions, scene graphs, and the pre-defined simulation loop [8]. The high-level architecture of the program, on the other hand, uses an object hierarchy based on anatomy.

2.2.1 WorldToolKit

Sense8's WorldToolKit library provides high-level functions that handle the low-level details of a 3D simulation, such as creating geometries, changing viewpoints, updating sensors, specifying lights, and applying material properties to polygons.

The advantages of using the *WTK* software include built-in performance features, shorter development time, and portability. For example, the *WTK* library can automatically convert the polygons in a geometry into triangular strips for faster rendering speeds. Moreover, by using the pre-built *WTK* sensor drivers, the interaction between the 3D mouse and the simulation was easily accomplished. Furthermore, since the library is written on top of the *OpenGL* standard, all *WTK* applications are portable and platform independent. Hence, any computer supporting the *OpenGL* standard can run the *Virtual Ophthalmoscopic Examination* simulation program.

2.2.1.1 Task Functions

The *WTK* library allows *C* functions, called task functions, to be associated with any object or data structure. At each iteration of the simulation loop, the *WTK* simulation engine calls each object's task function. These task functions, then, can be used to update or assign behavior to the objects in the universe.

For the *Virtual Ophthalmoscopic Examination* program, the heart of the simulation engine lies within the task functions. The task functions simulate and define the program's entire behavior. For example, task functions are used to (1) update all the graphical objects in the universe, (2) simulate the red-reflex behavior of the eye, (3) produce human behavior such as blinking and eye or head movement, and (4) update the ophthalmoscope's viewpoint position and orientation.

2.2.1.2 Scene Graphs

A scene graph, a hierarchical arrangement of nodes used to create a scene, is used as the underlying graphics data structure by the *WTK* library. Some of these nodes include transformation, geometric, separator, and light nodes. The advantages of using a scene graph include object grouping, level of detail switching, powerful culling that increases the frame rate, geometry and sub-tree instancing for better memory usage, and localized lighting and environmental effects.

During each iteration of the simulation, the rendering engine traverses the scene graph and determines which graphical objects are to be included in the scene and how they will be rendered. For the *Virtual Ophthalmoscopic Examination* program, a small scene graph and medium resolution geometric models were used to ensure a real-time simulation.

2.2.1.3 Simulation Loop

To simplify the programming of the simulation, the *WTK* simulation loop was employed. Not only does it call each object's task function, but the *WTK* simulation loop also reads all sensor data, updates objects with sensor input, and renders the universe, as shown in Figure 2.1.

For the *Virtual Ophthalmoscopic Examination* program, the simulation engine first reads the input from the 3D mouse. The universe's task function `UniverseTaskFn()` is then called to simulate the virtual patient's behavior and handle any keyboard input. Once the universe completes its task, the simulation engine calls the other object's task functions in the following order:

1. `spotLightTaskFn()`—updates the viewpoint of the simulation to corre-

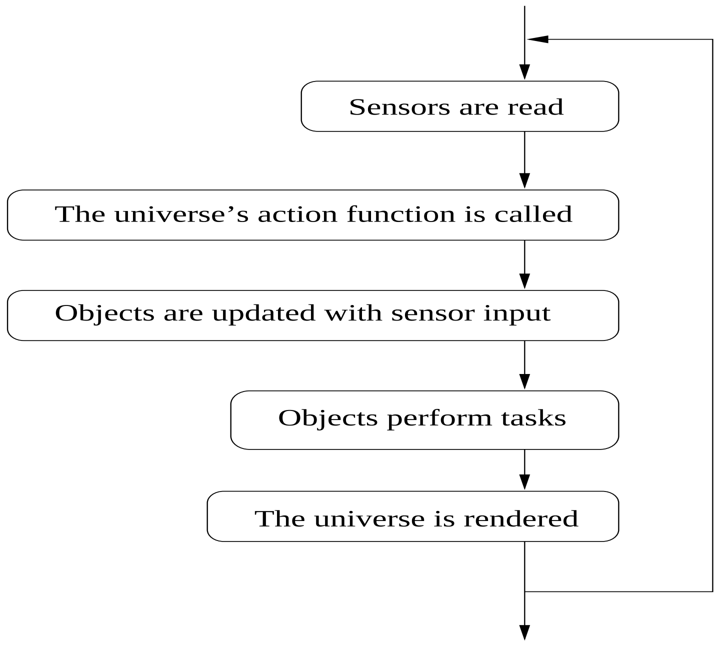


Figure 2.1: *WTK* Simulation Loop

spond with the virtual ophthalmoscope's position and orientation.

2. `RedFlashTaskFn()`—determines if the current position and orientation of the virtual ophthalmoscope would produce a red-reflex in the eye.

3. `TaskFn()`—updates all the transformation state variables of the graphical objects.

And finally, after the objects perform their tasks, the rendering engine traverses the scene graph and renders the current scene.

2.2.2 Object Hierarchy

Since the human anatomy can be regarded as a hierarchy of elements, it is natural to model the virtual patient using an object-oriented language, such as

C^{++}[9]. For example, the simulation universe is represented by a `VRUniverse` object that contains the two major objects of the simulation—a `Patient` and an `Ophthalmoscope` object. The `Patient` object handles all the behavior related to the virtual patient's face, eyes, and eyelids. The `Ophthalmoscope` object handles all the behavior related to the instrument's position, orientation, and spot light. Through the use of task functions, the behavior of the virtual patient and ophthalmoscope as well as their interaction are determined.

2.3 Performance Issues

To be an effective teaching tool, the *Virtual Ophthalmoscopic Examination* program must function in real-time. Since the simulation is a highly graphics intensive program, the performance of the simulation is mainly determined by the frame rate of the rendering engine. The design decisions made to improve the rendering performance of the program are described below.

2.3.1 Switch Nodes

During the course of a simulation, the appearance of the eye changes many times; the retina will appear black, red, or show the fundus image. Our first approach was to model the retinal behavior based on the actual red-reflex phenomenon. The program included a single retina geometry that dynamically changed during the simulation. Hence, if the retina needed to appear red, the program changed the color of each polygon in the retinal geometry from black to red. Similarly, if the fundus image needed to be shown, the appropriate image was texture-mapped onto each polygon. Although this method provided the correct behavior, there was a slight delay in the changing of the retina appearances since the program had to iterate through each polygon in the retina geometry.

With the help of the *WTK* switch node, the delay associated with the first approach was eliminated. The switch node allows multiple geometries, which represent the same object, to be grouped together and rendered one at a time. The switch nodes are intended to facilitate the animation of objects. Hence, each geometry attached to the switch node can be regarded as a single frame of an animation sequence.

Instead of using a single geometry to represent the retina, the current program creates three separate geometries—a black retina, a red retina, and a texture-mapped retina. To display a particular retina, the appropriate geometry to be rendered is selected by the switch node. Thus, to show the red retina, the switch node switches to the retina geometry model with red polygons. The switch node improves the performance by eliminating the need to recalculate the properties of the different retinas. Since the pre-constructed geometries remain in the program's memory, the only processing required is the time needed for the switch node to select the new geometry. Unfortunately, there is still a slight delay when switching to the texture-mapped fundus retina. This delay, however, is due to the implementation of the texture-mapping functions, which for the Solaris architectures are executed in software rather in hardware.

2.3.2 Single-Sided Polygons

In the *WTK* environment, geometries can be rendered with either single-sided or double-sided polygons. For single-sided polygons, only the side from which the polygon normal points is rendered. For double-sided polygons, both sides of the polygon are rendered. Thus, by specifying the use of single-sided polygons, only the front facing polygons of a geometry are rendered. Because the frame rate is largely dependent on the number of polygons in the scene, eliminating the back facing polygons substantially decreases the amount of work for the rendering engine. Moreover, since the ophthalmoscopic examination is a non-invasive procedure, the user never needs to view the interior of the geometries. Hence, it is sufficient to render only the front facing polygons. The only object, however, that needs to be rendered with double-sided polygons is the retina because it lies in the interior of the eye.

2.3.3 Geometry Resolutions and Triangle Strips

To further reduce the number of polygons in the scene, the geometry resolutions were kept to a minimum without losing any necessary details. For example, since the face and eye lids of the virtual patient do not play a major role in the visual experience of the simulation, their geometries were kept simple. On the other hand, since the eye is the focus of the simulation, the geometries of the eyeball and retina had to contain enough detail to look realistic. Hence, the resolution for these eye parts was reduced as far as possible without losing too much realism.

Another way in which we improved the frame rate performance of the *Virtual Ophthalmoscopic Examination* program was by using the *WTK* pre-build function to convert the polygons of the geometries into triangle strips. Because the rendering of triangle strips can be optimized fairly well, these pre-built geometries help increase the speed of the rendering engine.

2.3.4 Simulation Loop

The performance of the *Virtual Ophthalmoscopic Examination* program also relies on the efficiency of the main simulation loop. For example, a short, compact simulation loop decreases the time for each iteration, thus increasing the frame rate of the program. Since the implementation details of the built-in *WTK* simulation loop is hidden from the programmer, we originally developed our own simulation loop. In this way, we could better control the flow of the simulation loop.

After some experimentation, however, we discovered that using the pre-written simulation loop gave similar frame rates as those from our customized loop. Moreover, using task functions with the *WTK* simulation loop greatly simplified the program structure. Instead of traversing down the object hierarchy to invoke an update function, the task functions allow the update function to be attached directly to the object. Since the *WTK* simulation loop automatically manages the task functions, no further intervention from the program was needed. Hence, the *WTK* simulation loop provides real-time frame rates and leads to clean, simple source code.

CHAPTER 3

Implementation

In this chapter, we describe the implementation details of the *Virtual Ophthalmoscopic Examination* program. We present a description of the program's object hierarchy, the scene graph used in the simulation, and a detailed explanation of the functionality of each class.

3.1 Object Hierarchy

The object hierarchy of the *Virtual Ophthalmoscopic Examination* program is based on viewing the human anatomy and the real world as a hierarchy, as shown in Figure 3.1. At the top most level, there exists a `VRUniverse` object which represents the simulation universe. For our simulation, the universe contains two objects—a `Patient` and an `Ophthalmoscope` object. The `Patient` class contains a `Head` object with `Face`, `Eye`, and `EyeLid` objects. Moreover, the `Eye` object contains an `EyeBall`, `Iris`, and `Retina` object. The five stand-alone objects with geometric representations (`Face`, `EyeLid`, `EyeBall`, `Iris`, and `Retina`), all inherit from the `GraphicalObject` class, the base abstract class that handles all functions relevant to graphical objects. The `Ophthalmoscope` class, on the other hand, only contains a single object which inherits from the `GraphicalObject` class—an `EyePiece` object that handles the positioning of the

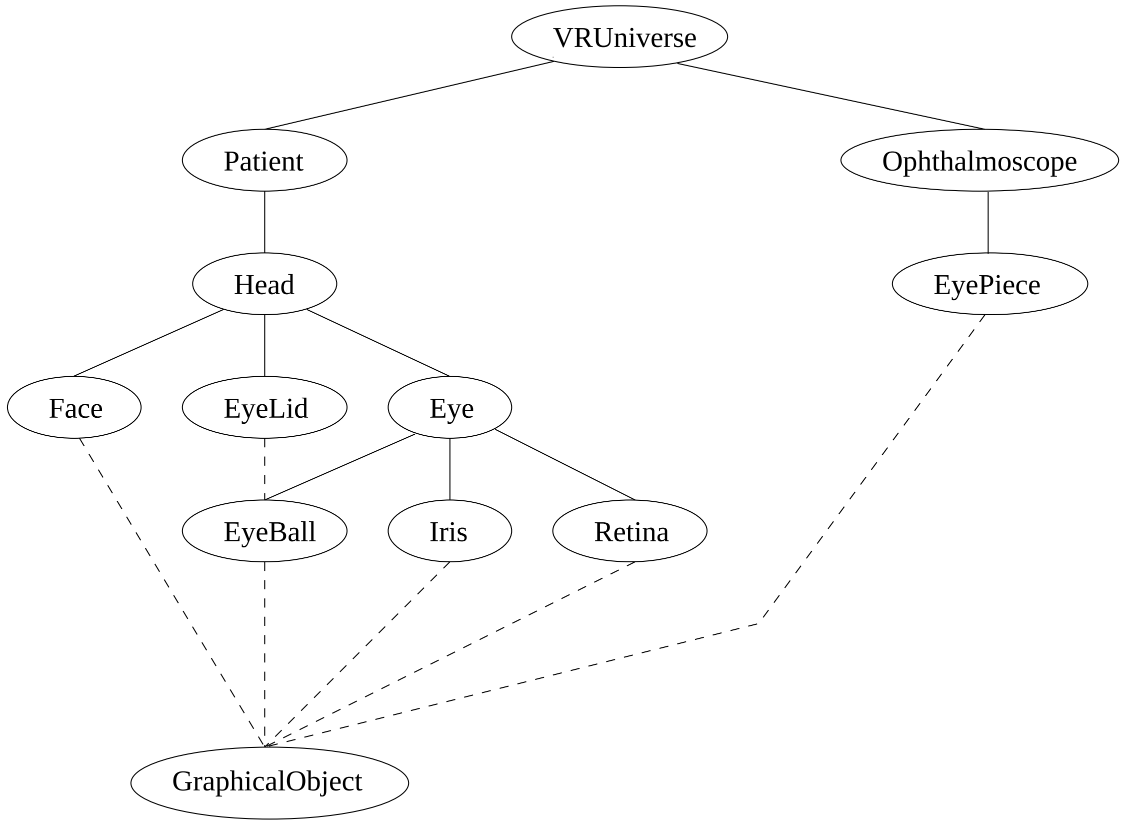


Figure 3.1: Object Hierarchy

instrument's spot light and updates the universe's viewpoint.

The `Patient` object determines how the parts of the `Head` will behave. For example, if the `Patient` objects decides that the virtual patient should blink, it informs the `Head` of the action. The `Head` object in turn tells the `EyeLid` object that it needs to move the eye lid geometry. Hence, when the scene is rendered, the eye lid geometry is in its new position.

Since the red-reflex behavior is determined by the relationship between the virtual ophthalmoscope and the eye geometry, the `Eye` object handles its subparts by itself. For example, if it has been determined that the red-reflex should be seen, the `Eye` object will tell the `Retina` object to display the retina geometry with a red-reflex. Similarly, if constriction of the iris is needed, the `Eye` object informs the `Iris` object to display an animation sequence that shows the iris geometry constricting.

3.2 Scene Graph

A brief description of the scene graph node types is presented followed by a description of the scene graph used in the simulation program.

3.2.1 Node Types

There exist three main categories of scene graph node types—geometry, attribute, and procedural nodes, as shown in Table 3.1. The geometry nodes contain the raw polygonal information as well as the specifications for the material properties of the polygons.

The attribute nodes govern the way the geometry nodes are rendered. For example, by setting the transform node, the given geometry can be translated, rotated, or scaled. In addition, lighting effects from ambient, directed, point, or spot lights can be applied to the object through light nodes.

The procedural nodes are used to control the way the scene graph is traversed or processed. For example, the group node allows several nodes to be grouped together and treated as a single entity for more efficient handling. Switch nodes determine which child should be traversed by the rendering engine. This is particularly useful when there are several representations of the same object at different times, such as during an animation sequence. Separator nodes prevent information from propagating throughout the scene graph. Similarly, transform separa-

Node Types		
Type	Name	Action
Geometry	Geometry	Displays a set of polygons with material properties
Attribute	Transform	Provides position and orientation information
	Light	Specifies a ambient, directed, point, or spot light source
Procedural	Group	Allows grouping of other nodes
	Switch	Controls which of its children are traversed
	Separator	Prevents state information from propagating from its descendent nodes to its sibling nodes
	Transform Separator	Prevents only the transformation information from propagating from its descendent nodes to its sibling nodes

Table 3.1: *WTK* scene graph node types.

tor nodes only prevent transformation information from propagating. These two types of separator nodes are useful when it is desired to isolate a particular node from the rest of the scene graph.

3.2.2 Scene Graph Hierarchy

Similar to the object hierarchy, the scene graph hierarchy is structured around the natural order of the human anatomy. The scene graph for the virtual ophthalmoscope with the spot light and the virtual patient with a face, two eyes, and two eyelids is shown in Figure 3.2.

Due to the processing nature of the rendering engine, the scene graph for the simulation has a more complex hierarchy than the object hierarchy. Beginning

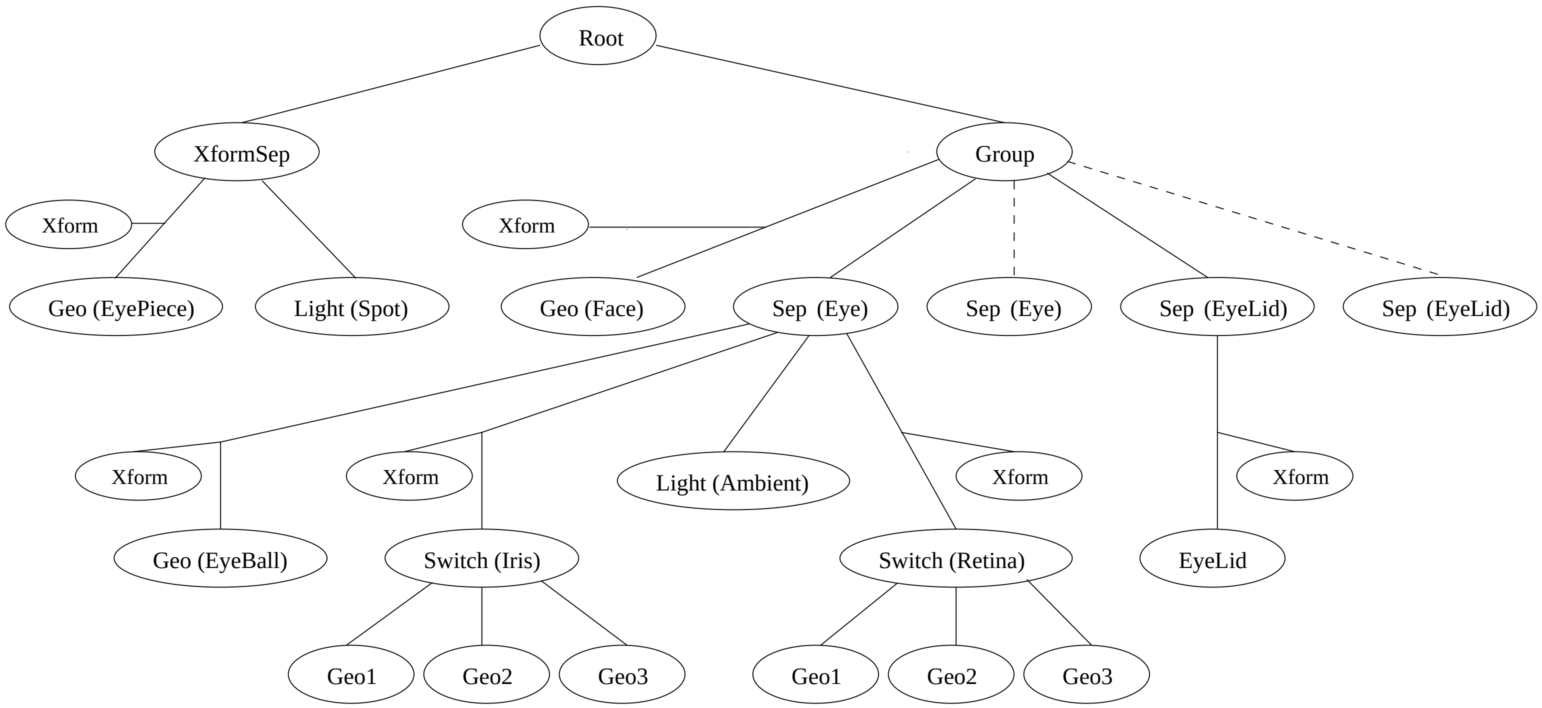


Figure 3.2: Scene Graph Hierarchy (dashed-lines indicate identical structures)

at the root node, the rendering engine traverses the scene graph in a depth-first manner from left to right. Moreover, because all state information is propagated along the traversal path, the order of the nodes becomes significant since all previous nodes can potentially affect all the following ones. For example, if an object needs to be lit by a light source, the light node must be placed before (to the left of) the node that represents the object. Conversely, if the state information of a node needs to be protected from the rest of the scene graph, a separator node must be inserted to prevent the information from propagating.

During the simulation, the rendering engine first processes the nodes representing the virtual ophthalmoscope. This small group of nodes to the left of the root contains a geometry and a light node to simulate the ophthalmoscope's spot light. Since the spot light should only affect the virtual patient, a transform separator is inserted into the scene graph. Hence, the light node is still able to affect the rest of the scene graph without propagating any other transformation information onto other nodes in the scene graph.

Once the nodes related to the ophthalmoscope have been traversed, the group of nodes to the right of the root node is processed. These nodes represent the virtual patient and include geometric representations for the face, the left and right eyes, and the left and right eyelids. Since the movement of the face affects the positioning of all the other components, the transform node associated with the face geometry is left unprotected. Hence, a rotation of the face causes a corresponding rotation in the eyes and eyelids. Since the remaining components of the face are independent, they need to be isolated using separator nodes.

After the face nodes have been traversed, the rendering engine handles the two eye nodes. Because the *WTK* lighting model does not support shadows, the eye is divided into two regions—an outer region and an inner region. The outer region is comprised of all the parts of the eye that are visible from the exterior, such as the front half of the eyeball and the iris. The inner region includes the part of the eye that can only be seen with the ophthalmoscope, the retina. To make the inner region appear dark, an ambient light with negative intensity was added to cancel out the lighting effects. And because the iris and retina representations can change during the simulation, switch nodes were used to select the appropriate geometric representation. To correctly simulate the lighting of the outer region components, the eyeball and iris geometries are placed before the negative ambient light. To the right of the negative ambient light, then, are placed the inner region geometries of the retina. Hence, the universe's ambient light only affects the outer geometries of the eyeball and iris.

Once the nodes of the two eyes have been traversed, the nodes representing the left and right eyelids are processed. The eyelid nodes contain a single transform node along with a geometric representation of the eyelid. To make the virtual patient blink, the eyelid geometries are rotated through the transform node.

3.3 Class Descriptions

For each of the classes in the hierarchy, we present a brief description of its functionality and explain the more subtle points of the implementation. A more detailed description of each function can be found in the source code documentation in appendix B.

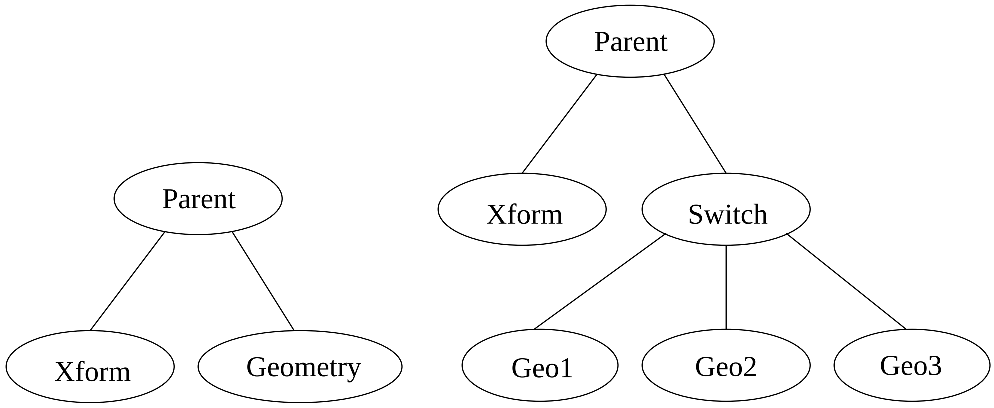
3.3.1 GraphicalObject class

The `GraphicalObject` class member functions handle (1) the creation of graphical objects in the universe, (2) the manipulation of an object's graphical attributes such as visibility or color, and (3) positional and rotational transformations.

The constructor for the class accomplishes the first task of creating a graphical object by adding the geometric representation of the object to the scene graph and initializing the object's transformation state. When adding the object to the scene graph, the constructor first creates a transform node for the object, loads the geometry into a geometry node, and then attaches these nodes to the parent node. The object's transformation state holds information regarding rotations or translations and is initialized to zero. However, because some objects have multiple geometric representations, two constructors for the `GraphicalObject` class are needed—one for single representations and another for multiple ones. The constructor for the multiple geometric objects is identical to the single case described above but instead of creating a single geometric node, a switch node is created with multiple geometric nodes. The two different scene graph structures are shown in Figure 3.3.

The second task of changing the object's attributes is mainly handled by wrapper functions that call the corresponding *WTK* library functions. For example, the member function `GeometrySetRGB()` is an inline function that simply calls the *WTK* function `WTgeometry_setrgb()`.

The last task is accomplished through a combination of the `GraphicalObject`'s member functions and a task function. First, the member function `SetAction()` determines what type of transformation is being requested and then stores the total amount and the incremental step value of the transformation in the ob-



(a) (b)

Figure 3.3: Node structure for (a) single geometries and (b) multiple geometries

ject's transformation state variables. Once the transformation information has been set, the `GraphicalObject` task function `TaskFn()` increases the current transformation variables by the incremental step value. At each iteration, after the task function has updated the object's current transformation state variables, the *WTK* rendering engine renders the universe. This update process is repeated until the object completes the transformation. By using this frame by frame update technique, all the graphical objects in the universe can move simultaneously, creating a much more realistic simulation.

3.3.2 VRUniverse class

The `VRUniverse` class has three main purposes. First, it uses wrappers to many of the *C* implemented functions of the *WTK* library to initialize the graphical universe. Second, after the simulation universe is initialized, the `VRUniverse` creates the `Patient` and `Ophthalmoscope` objects. Lastly, the universe's task function that updates the `Patient`'s behavior every iteration, `UniverseTaskFn()`, is attached to the universe.

3.3.3 Patient class

The `Patient` class contains a `Head` object and simulates the virtual patient's behavior. The class member function `SimulateBehavior()`, invoked by the universe task function, produces random head and eye movement, determines when the virtual patient blinks, and handles keyboard input from the user.

3.3.4 Head, Face, and EyeLid classes

The responsibilities of the `Head` class include initializing the scene graph hierarchy for the `Face`, `Eye`, and `EyeLid` objects and providing an interface to these components. To accomplish the first task, the constructor for the `Head` class creates a single group node with four separator nodes. The `Face` object adds its geometry node directly to the group node. The two `Eye` and two `EyeLid` objects, however, need to be added to separator nodes to prevent the objects from affecting each other. Hence, the `EyeLid` objects customize their attributes and then add their geometry nodes to the corresponding separator node. Due to its complexity of parts, the `Eye` object, on the other hand, creates and attaches a deeper hierarchy of nodes to its separator node. The second task of providing an interface is handled by wrapper functions that call the inherited `GraphicalObject` functions for the `Face`, `Eye`, and `EyeLid` objects.

3.3.5 Eye, EyeBall, Iris, and Retina classes

The `Eye` class forms the scene graph hierarchy for the `EyeBall`, `Iris`, and `Retina` objects and handles the eye's red-reflex behavior. Due to the lighting constraints of the scene graph, the `EyeBall` and `Iris` nodes are added first to the scene. An ambient light node with negative intensity is then added followed by the `Retina` nodes. In determining the red-reflex behavior, the `Retina` member function `CalcAngleBounds()` uses the ophthalmoscope-to-patient distance to calculate the lower and upper bounds of the red-reflex angle. The `Eye` task function `RedFlashTaskFn()` uses these angles to determine whether the current view position and orientation of the ophthalmoscope would produce a red-reflex. Moreover, depending on the distance to the patient, either the red-reflex or the actual retinal image is displayed.

3.3.6 Ophthalmoscope and EyePiece classes

The `Ophthalmoscope` class contains an `EyePiece` object to simulate the view from an ophthalmoscope. The `EyePiece` class adds a spot light node to the scene graph and attaches the 3D mouse sensor to the `EyePiece` geometry. In order to create the correct instrument movement, the `EyePiece` task function `spotLightTaskFn()` sets the simulation's viewpoint to correspond with the eyepiece's position and orientation.

CHAPTER 4

Validation

To evaluate the realism and effectiveness of the *Virtual Ophthalmoscopic Examination* program, two separate groups of students from the nursing program at UCLA were invited to use the simulation. The feedback from a small group of five students was first used to adjust the parameters of the system. Feedback from a second, larger group of thirteen students was then used to further refine the simulation parameters as well as to survey a larger population of students. In this chapter, we present the students' responses and the results of the user surveys.

4.1 User Background

Most of the students in both groups had used a mannequin with slides as a training aid in the classroom. Moreover, all of the students had previously performed an ophthalmoscopic examination, either on other students or on actual patients. Only 2 out of the 18 students, however, had previously used any type of virtual reality equipment. Although not familiar with virtual reality, all of the students had had some prior experience with educational software on the computer via multimedia programs, CDROMs, or the Internet. Hence, all of the students were not only well versed in the details of the examination procedure but also familiar with learning from and interacting with a computer.

4.2 User Surveys

A description of the questionnaire used, followed by a discussion of the survey results, is given below.

4.2.1 Questionnaire

The *Virtual Ophthalmoscopic Examination* program was tested on nursing students to assess student opinion on the ease of use and overall effectiveness of the simulation and to obtain suggestions from faculty and students on the user interface and options. A questionnaire was given to each participating student to evaluate the program in three areas—the accuracy of the eye behavior, the intuitiveness of the user interface, and the educational content of the program. The students scored these areas on a scale from 1 to 5. The questions asked, the range of answers (the first one representing 1 on the scale; the second representing 5), and the average scores for groups I and II are given below. For example, consider the first row in Table 4.1. The students were asked to judge the difficulty of obtaining the red-reflex. The possible answers ranged from easy (1 on the scale) to hard (5 on the scale). For group I, the average score was 3.40 while for group II it was 2.77.

4.2.2 Group I

Overall, the students in the first group thought that (1) the eye behavior in the simulation needed improvement, (2) the virtual reality components were fairly

Question	Score	
	I	II
Obtaining the red-reflex was easy/hard	3.40	2.77
The angle for the red-reflex was narrow/wide	2.20	2.62
Maintaining the red-reflex was easy/hard	4.00	4.08
Obtaining the fundus image was easy/hard	3.20	3.75
The angle for the fundus image was narrow/wide	2.20	2.73
Maintaining the fundus image was easy/hard	3.80	3.50

Table 4.1: Survey Results: Eye Behavior

easy to use, and (3) the educational content of the program was worthwhile.

4.2.2.1 Eye Behavior

Table 4.1 shows that the students in the first group felt that obtaining the red-reflex in the simulation was slightly more difficult than in real life and the angles for the red-reflex were too narrow. Maintaining the red-reflex in the eye, however, was hard for the students. Similar to the red-reflex, the students thought that obtaining the fundus image was a bit difficult and the angles for the fundus image were too narrow. Again, the students had difficulty maintaining the fundus image in the eye.

4.2.2.2 VR Components

Overall, the students thought the simulation was realistic and accurate. As shown in Table 4.2, the virtual environment felt fairly natural and realistic for most

Question	Score	
	I	II
Using the 3D mouse to navigate is easy/hard	3.40	3.69
Using the Stereo CrystalEyes is distracting/not distracting	3.60	4.00
The virtual patient is not realistic/realistic	2.60	3.69
The 3D effect made the simulation not realistic/realistic	4.20	3.69
The virtual environment felt awkward/natural	3.60	3.25
The virtual environment felt disorienting/realistic	3.50	3.50

Table 4.2: Survey Results: VR Components

Question	Score	
	I	II
Compared to real patients, the program is not realistic/realistic	3.20	3.09
Viewing a 3D model of the eye parts is not helpful/helpful	3.60	3.90
My knowledge of the exam was improved not at all/greatly	2.80	3.42
Using the simulation in classrooms will be not helpful/helpful	4.20	4.42

Table 4.3: Survey Results: Educational Content

students. The students were comfortable wearing the *StereoGraphics CrystalEyes* and moving around in the virtual environment. The students felt, however, that the 3D flying mouse was a bit difficult to use and the virtual patient was not realistic enough.

4.2.2.3 Educational Content

Table 4.3 shows that the majority of students believe that using the simulation in the classroom would be beneficial for future students. The program contains the major features of a real examination and the 3D modeling increases the effectiveness of the simulation. But, since most students in group I were already well versed in the procedure, most did not learn anything new.

4.2.2.4 Summary

For the first group, we were pleased to see that the students were not intimidated by the virtual reality technology and reacted positively to the potential use of the program in the classroom. The biggest problem, however, was maintaining the red-reflex and fundus image in the eye. This was probably due to the fact that the 3D mouse was too sensitive and the angles were too narrow. Many students commented that learning how to use the flying mouse was slightly frustrating as it took some time for them to adjust to the 3D interface.

4.2.3 Group II

Based on the feedback from the first group, the angles for the red-reflex and fundus image were widened and the sensitivity of the mouse was decreased. The second group of students was then asked to test the simulation program and fill out the same user survey as the first group.

4.2.3.1 Eye Behavior

As shown in Table 4.1, the students from the second group felt that the angles for the red-reflex were accurate and obtaining the red-reflex was slightly too easy. Many students also commented that the view of the fundus was too restrictive; on real patients, the view of the fundus is a bit wider. Most students still, however, had difficulty maintaining the red-reflex and fundus image as they approached the virtual patient's eye.

4.2.3.2 VR Components

The students in group II thought that the 3D effect made the simulation more realistic, the virtual environment felt natural, and the virtual patient was fairly realistic, as shown in Table 4.2. The students also felt that wearing the *StereoGraphics CrystalEyes* was not distracting, but as with the first group, many students complained about the sensitivity of the mouse sensor. Because of this, navigating in the virtual environment was difficult and many students had problems adjusting to the 3D interface.

4.2.3.3 Educational Content

Similar to the first group, the majority of students believe that the simulation program would be helpful in the classroom. The students in the second group also felt that, in general, the simulation was realistic. As shown in Table 4.3, viewing the 3D model of the eye made a bigger impact on the second group and a few students thought the simulation improved their knowledge of the examination procedure.

4.2.3.4 Summary

As with the first group, the students from the second group reacted positively to the simulation and felt that the program is a vast improvement over the styrofoam mannequin and would be helpful in the classroom. We are also pleased to see that the scores related to the sensitivity of the mouse and the width of the angles improved after our adjustments to the program. Many students still, however, had slight difficulty with maintaining the red-reflex and fundus image in the eye. Hence, a few more adjustments need to be made in these areas.

CHAPTER 5

Conclusion

In this chapter, we present our conclusions of the *Virtual Ophthalmoscopic Examination* program. We discuss the user interface issues of the simulation and present the future work to be done on the program.

5.1 User Interface

The strength of the virtual reality program lies in the program's ability to teach the physical aspects of the examination procedure. Using the 3D mouse interface to navigate in the virtual environment helps train the students in the physical movements of the exam.

From the user surveys, however, it was shown that the interface was difficult to learn. Because of the sensitivity of the mouse, navigation near the eye was difficult. Moreover, because we are using a fully immersive style interface with non-immersive equipment, the students were easily disoriented. For example, when users rotate the viewpoint to the right, they think that the virtual patient moves left when in reality the virtual patient remains fixed and the user's viewpoint changes. Furthermore, since people have a natural tendency to stand directly in front of the stationary computer monitor, the students began moving only the mouse rather than moving their entire body around the monitor, thus eliminating the physical nature of the program.

The sensitivity of the mouse can easily be adjusted but the other problems with the user interface are related more directly to the virtual reality equipment. In order to eliminate the shortcomings of the 3D mouse, several other interfaces should be investigated, such as a wired ophthalmoscope, a head tracker, or a head mounted display.

5.1.1 Wired Ophthalmoscope

By attaching a few sensors to the body of the instrument, a wired ophthalmoscope could be used in place of the flying mouse. The wired ophthalmoscope would have the same functionality as the 3D mouse but instead of holding the 3D mouse, the student would be holding the much more familiar ophthalmoscope.

Introducing an actual ophthalmoscope into the simulation may increase the intuitiveness of the interface. Since the students will probably be more comfortable with an ophthalmoscope than the 3D mouse, it may feel more natural to hold the wired ophthalmoscope at eye level and walk around. Similar to the 3D mouse, however, the non-immersive nature of the wired ophthalmoscope may still cause some students to lose their orientation in the simulation.

5.1.2 Head Tracker

Another approach would be to simplify the interface by eliminating all hand-held input sensors and use a head tracking device. By tracking the *StereoGraphics CrystalEyes* on the head, the position and orientation of the user could be calculated. In order to move in the virtual environment, then, the user will be forced to physically move around. Hence, using a head tracker would definitely emphasize the physical aspect of the examination procedure. However, the disorientation caused by the non-immersive equipment still remains.

5.1.3 Head Mounted Display

The best interface would be to use a head mounted display that creates a fully immersive environment. When wearing the head mounted display, the user becomes immersed in the virtual environment. Hence, the disorientation problems associated with the non-immersive sensors are non-existent since there is no frame of reference to confuse the user.

If the hardware were available, a virtual reality lab with sensors to track the head mounted display could be used to update the user's position to create a true virtual simulation. However, due to the high cost of such a lab, a data glove or joystick would be more appropriate as an input device to change the user's position in the simulation.

5.1.4 Summary

Although a head mounted display would provide the most realistic simulation, the wired ophthalmoscope or the head tracking device would be just as adequate. The only drawbacks to the non-immersive interfaces are that they are slightly counter-intuitive and it may take some time for the user to adjust. However, from the trials with the nursing students, many adapted to the interface after a few minutes of experimenting.

5.2 Future Work

The *Virtual Ophthalmoscopic Examination* simulation program described in this thesis has successfully modeled the examination process. To be a more effective teaching tool, however, additional features can be added to enhance the student's learning experience, such as a larger fundus database, instructional cues, different skill levels, ophthalmoscope adjustments, and various training aids.

5.2.1 Fundus Database

The simulation program contains five retinal conditions: (1) normal, (2) glaucomatous cup, (3) papilledema, (4) macular scar, and (5) exudates. To increase the student's knowledge of eye diseases, the program should be able to expose the student to as many fundi images as possible. Thus, the database of fundi images could be increased to 150 or more images.

5.2.2 Instructional Cues

For the beginning student, adding instructional cues into the program would be extremely helpful. For example, if the student had difficulty obtaining the red-reflex, the program could tell the student in which direction to move. Alternatively, a path showing the correct route to take could be displayed for the student to use as a guide. Once the students became comfortable with the examination procedure, the instructional cues could be turned off so that the students can perform the exam on their own.

5.2.3 Skill Levels

The virtual patient in the *Virtual Ophthalmoscopic Examination* program remains still during the simulation. In future versions, it would be beneficial to include various behavior patterns for different student skill levels. For example, for beginners, the virtual patient would remain still with occasional blinking. For intermediate students, the virtual patient may blink faster, shift his eyes more during the simulation, or jerk his head away from the light. To challenge the advanced students, more random behavior would be added to the simulation and the virtual patient would have a lower tolerance of pain. Hence, it would be more difficult for the student to maintain the red-reflex in the advanced mode since the patient moves more frequently.

5.2.4 Ophthalmoscope Adjustments

The current simulation stresses only the positioning of the ophthalmoscope. During a real examination, however, the student must also adjust the focus of the ophthalmoscope. In addition to these lenses, different light colors and beam shapes are available on the ophthalmoscope to examine other characteristics of the eye. Hence, a higher level of realism can be obtained if the program allows the student to change the diopters or the type and color of the light beam on the ophthalmoscope.

5.2.5 Training Aids

Additionally, some more advanced training aids could be added to help the student. For example, the student may ask for (1) cross-sectional views of the eye,

(2) explanations of the parts and handling techniques of the ophthalmoscope, (3) explanations of the diseased states of the eye, or (4) a glossary of terms.

5.3 Conclusion

The *Virtual Ophthalmoscopic Examination* simulation program has shown to be a promising training aid for students. By displaying a three-dimensional view of the eye, the simulation program provides a much more realistic experience as compared to the mannequin. Moreover, whereas the photographic slides used in the mannequin show the full fundus, the simulation accurately simulates a more limited view. By adding realism to the training, the students learn more and the transition to examining real patients will be less difficult. With a few improvements to the user interface, the *Virtual Ophthalmoscopic Examination* program can prove to be quite an effective teaching tool in the classroom.

APPENDIX A

User Manual

In this appendix, we describe the peripheral equipment needed for the virtual environment and the operating instructions for the *Virtual Ophthalmoscopic Examination* simulation program.

A.1 Peripheral Equipment

In addition to a computer installed with the *Sense8's WorldToolKit* libraries, the two components needed for the system are a *Logitech 3D Mouse* and the *StereoGraphics CrystalEyes*. The *Logitech 3D Mouse* works in conjunction with an infrared head tracker to triangulate the position of the user in three-dimensional space. The *StereoGraphics CrystalEyes* are polarized lenses that are synchronized with a stereo graphics display to create the three-dimensional effect for the virtual environment.

A.2 Getting Started

The *Virtual Ophthalmoscopic Examination* program can be run with or without stereo support. The program executable (`vr`) takes a single optional argument (`-useCrystal`) that specifies if stereo images are to be displayed. To start the program with stereo support the following steps should be followed:

1. Configure the computer's display for stereo imaging. On systems running Solaris with the Creator3D graphics, type the following on the command line:

   `/usr/sbin/ffbconfig -res 960x680x108s`

2. Turn on the *CrystalEyes* and the *CrystalEyes* receiver

3. Turn on the *Logitech 3D Mouse* receiver and position the *Logitech 3D Mouse* transmitter on top of the computer monitor

4. Enter the directory where the simulation executable is kept

5. Begin the program by typing the following at the command line:

   `vr -useCrystal`

To start the program without stereo support, skip the first two steps above and simply run the executable with no arguments. To put the computer back into mono mode, type:

1. `/usr/sbin/ffbconfig -res 1280x1024x76`

A.3 User Interface Commands

Although most of interaction between the user and the *Virtual Ophthalmoscopic Examination* program involves moving the *Logitech 3D Mouse* , a few other mouse and keyboard commands are provided for the user.

A.3.1 Mouse Commands

The two buttons on the base of the 3D flying mouse are used to reposition the mouse in the real world without affecting the state of the user's position in the simulation. For example, during the course of the simulation, the user will move toward the *Logitech 3D Mouse* transmitter on top of the computer monitor to get closer to the eye of the virtual patient. It may turn out, however, that the user can no longer move any farther forward since the computer monitor is in the way. To reposition the mouse, the user can press one of the buttons on the base of the 3D flying mouse to freeze the current viewpoint in the simulation. While the button on the base of the mouse is depressed, the position and orientation of the 3D flying mouse is ignored by the simulation program. In this way, the user can move the mouse away from the computer monitor and then resume the simulation by releasing the button on the base of the 3D mouse.

A.3.2 Keyboard Commands

The keyboard commands allow the user to change the fundus image and return the user to the starting position, as shown in Table A.1.

Key	Description
"1"	Display "normal" fundus image
"2"	Display "glaucoma" fundus image
"3"	Display "papilledema" fundus image
"4"	Display "macular scar" fundus image
"5"	Display "exudates" fundus image
"h"	Return user to starting position (home)

Table A.1: Keyboard commands

APPENDIX B

Source Code

In this appendix, the complete source code for the *Virtual Ophthalmoscopic Examination* program is presented. Sections B.1 to B.25 list the class definitions that are archived into the library `libOphthal.a`. Section B.26 presents the *C* task functions used by the simulation and section B.27 contains the the code for the `main()` procedure.

B.1 GraphicalObject.h

```
/*******************************************************
// GraphicalObject class
//
// Handles the creation of graphical objects in the
// universe, the manipulation of an object's graphical
// attributes, and position and rotational transformations.
// Also is attached to the task function, TaskFn(). See
// TaskFn.cc for more details.
/*******************************************************/
#ifndef GRAPHICALOBJECT_H
#define GRAPHICALOBJECT_H
extern "C" {
#include "wt.h"
}

class GraphicalObject {
public:
    GraphicalObject(WTnode *parent,char *fileName,float scale);
    GraphicalObject(WTnode *parent,char **fileName,int numSwitchNodes,float scale);
    GraphicalObject(WTnode *obj);
    ~GraphicalObject();
    void SetVisibility(FLAG token);
    void GeometryTranslate(WTp3 p);
```

```
    void GeometrySetRGB(int r, int g, int b);
    void GeometrySetMidPoint(WTp3 p);
    void SetWhichChild(int which);
    void GeometryPreBuild();
    void GeometryChangeTexture(char *bitmap,FLAG shading,FLAG transparent);
    void SwapColor(unsigned char r, unsigned char g, unsigned char b);
    void SetAction(short action,float span,float incr);
    void SetPivot(WTp3 pivot);

    /* Transformation state variables */
    float currVal[2][3];
    float totVal[2][3];
    float incrVal[2][3];
    int signVal[2][3];
    short axisVal[2][3];
    short dirty;

    /* Scene graph variables */
    WTnode *node;
    WTnode *xformNode;
    WTp3 p;
    WTp3 midPoint;
    WTp3 centerLine;
};


inline void
GraphicalObject::SetVisibility(FLAG flag)
{
    WTnode_enable(node,flag);
    WTnode_enable(xformNode,flag);
}

inline void
GraphicalObject::SwapColor(unsigned char r, unsigned char g, unsigned char b)
{
    WTgeometry_setrgb(WTnode_getgeometry(node),r,g,b);
}

inline void
GraphicalObject::GeometryTranslate(WTp3 p)
{
    WTgeometry_translate(WTnode_getgeometry(node),p);
}

inline void
GraphicalObject::GeometrySetRGB(int r, int g, int b)
```

```
{
    WTgeometry_setrgb(WTnode_getgeometry(node),r,g,b);
}

inline void
GraphicalObject::GeometrySetMidPoint(WTp3 p)
{
    WTp3_copy(p,midPoint);
}

inline void
GraphicalObject::SetWhichChild(int which)
{
    WTswitchnode_setwhichchild(node,which);
}

inline void
GraphicalObject::GeometryPreBuild()
{
    // Note: can't change colors or textures after prebuild (5-30)

    if (!WTgeometry_prebuild(WTnode_getgeometry(node)))
        WTerror("prebuild failed for %s\n",WTnode_getname(node));
}

inline void
GraphicalObject::GeometryChangeTexture(char *bitmap,FLAG shading,FLAG transparent)
{
    WTgeometry_changetexture(WTnode_getgeometry(node),bitmap,shading,transparent);
}

#endif /* !__GraphicalObject_h */
```

B.2 GraphicalObject.cc

```
#include <iostream.h>
#include "GraphicalObject.h"
#include "actions.h"
#include "math.h"

/****************************************************
// GraphicalObject::GraphicalObject()
// Constructor for single geometry graphical object.
/****************************************************/
GraphicalObject::GraphicalObject(WTnode *parent, char *fileName, float scale)
{
```

```
    /* Create Xform and Object node */
    xformNode = WTxformnode_new(parent);
    WTnode_setname(xformNode,fileName);

    if (!(node = WTgeometrynode_load(parent,fileName,scale)))
        WTmessage("Couldn't load %s\n",fileName);
    else {
        WTnode_setname(node,fileName);
    }

    /* Init data structures */
    for(int j=0;j<2;j++) {
        for(int i=0;i<3;i++) {
            currVal[j][i] = 0;
            totVal[j][i] = 0;
            incrVal[j][i] = 0;
            signVal[j][i] = 1;
            axisVal[j][i] = 0;
        }
    }
    dirty = FALSE;
    WTp3_init(p);
    WTgeometry_getmidpoint(WTnode_getgeometry(node),midPoint);
    WTp3_init(centerLine);
    centerLine[Z]= -1.0;
}


/****************************************************
// GraphicalObject::GraphicalObject()
// Constructor for multiple geometry graphical object.
/****************************************************/
GraphicalObject::GraphicalObject(WTnode *parent, char **fileName, int
numSwitchNodes,float scale)
{
    /* Create Xform and Object node */
    xformNode = WTxformnode_new(parent);
    WTnode_setname(xformNode,fileName[0]);

    /* Create switch node */
    node = WTswitchnode_new(parent);
    for (int i=0;i<numSwitchNodes;i++) {
        if (!WTgeometrynode_load(node,fileName[i],scale))
            WTmessage("Couldn't load %s\n",fileName[i]);
    }

    /* Init data structures */
```

```
    for(int j=0;j<2;j++) {
        for(int i=0;i<3;i++) {
            currVal[j][i] = 0;
            totVal[j][i] = 0;
            incrVal[j][i] = 0;
            signVal[j][i] = 1;
            axisVal[j][i] = 0;
        }
    }
    dirty = FALSE;
    WTp3_init(p);
    WTgeometry_getmidpoint(WTnode_getgeometry(node),midPoint);
    WTp3_init(centerLine);
    centerLine[Z]= -1.0;
}


/****************************************************
// GraphicalObject::GraphicalObject()
// Constructor for graphical object that already exists
/****************************************************/
GraphicalObject::GraphicalObject(WTnode *obj)
{
    /* Create Object */
    node = obj;

    /* Init data structures */
    for(int j=0;j<2;j++) {
        for(int i=0;i<3;i++) {
            currVal[j][i] = 0;
            totVal[j][i] = 0;
            incrVal[j][i] = 0;
            signVal[j][i] = 1;
            axisVal[j][i] = 0;
        }
    }
    dirty = FALSE;
    WTp3_init(p);
    WTgeometry_getmidpoint(WTnode_getgeometry(node),midPoint);
    WTp3_init(centerLine);
    centerLine[Z]= -1.0;
}


/****************************************************
// GraphicalObject::~GraphicalObject()
// Destructor for class
```

```
/****************************************************/
GraphicalObject::~GraphicalObject()
{
    WTnode_remove(xformNode);
    WTnode_remove(node);

    /* delete WTpq ?? */
}


/****************************************************
// GraphicalObject::SetAction()
// Function that sets the transformation state
// variables for translations and rotations.
/****************************************************/
void
GraphicalObject::SetAction(short action,float span,float incr)
{
    short axis,actionType;

    /* Handle Object Translations */
    if ((action & TRANSLATE) == TRANSLATE) {
        actionType = TRANS;

        /* Determine Axis of Translation */
        if ((action & X_AXIS) == X_AXIS)
            axis = X;
        if ((action & Y_AXIS) == Y_AXIS)
            axis = Y;
        if ((action & Z_AXIS) == Z_AXIS)
            axis = Z;

        /* Assign Directional information */
        if ((action & POS) == POS )
            signVal[TRANS][axis] = 1;
        else
            signVal[TRANS][axis] = -1;

    }
    /* Handle Object Rotations */
    else {
        actionType = ROT;
        /* Determine Axis of Rotation */
        if ((action & X_AXIS) == X_AXIS)
            axis = X;
        if ((action & Y_AXIS) == Y_AXIS)
            axis = Y;
```

```
        if ((action & Z_AXIS) == Z_AXIS)
            axis = Z;

        /* Assign Directional Information */
        if ((action & POS) == POS)
            signVal[ROT][axis] = 1;
        else
            signVal[ROT][axis] = -1;
    }

    axisVal[actionType][axis] = 1;
    totVal[actionType][axis] = span;
    incrVal[actionType][axis] = incr;
    currVal[actionType][axis] = 0;
}


/*****************************************************
// GraphicalObject::SetPivot()
// Function to set object's pivot point
*****************************************************/
void
GraphicalObject::SetPivot(WTp3 pivot)
{
    WTp3 invpivot;

    WTp3_invert(pivot,invpivot);
    WTgeometry_translate(WTnode_getgeometry(node),invpivot);
    WTnode_translate(xformNode,pivot,WTFRAME_LOCAL);
}
```

B.3 VRUniverse.h

```
/********************************************************
// VRUniverse class
//
// Contains wrappers to the WTK library functions to
// initialize the universe, creates the virtual patient and
// opthalmoscope objects, and is attached to the universe
// task function, UniverseTaskFn(). See TaskFn.cc for more
// details.
********************************************************/
#ifndef VRUNIVERSE_H
#define VRUNIVERSE_H
#include "Patient.h"
#include "Ophthalmoscope.h"
```

```
#include "Banner.h"

class VRUniverse {
public:
    VRUniverse(int argc, char **argv,char *eyeParts[4][3],float scale);
    ~VRUniverse();
    void Ready();
    void ZoomAll();
    void Go();
    void Go1();
    void Delete();
    short GetKey(short *key);
    FLAG isBannerBusy();
    void EnableSensors();
    WTnode *GetRootNode();
    WTnode *GetBannerRootNode();

    Patient *patient;
    Ophthalmoscope *ophthal;
    Banner *banner;

private:

    /* Command Line variable */
    short useTexture;
    short useCrystalEyesVR;
    long displayConfig;
    long windowConfig;
    char *lightsFile;

    /* Scene graph root nodes */
    WTnode *rootNode;
    WTnode *bannerRootNode;
};


inline void
VRUniverse::Ready()
{
    WTuniverse_ready();
}

inline void
VRUniverse::ZoomAll()
{
    WTviewpoint_zoomall(WTuniverse_getviewpoint());
}
```

```
inline void
VRUniverse::Go()
{
    WTuniverse_go();
}

inline void
VRUniverse::Go1()
{
    WTuniverse_go1();
}

inline void
VRUniverse::Delete()
{
    WTuniverse_delete();
}

inline short
VRUniverse::GetKey(short *key)
{
    *key = WTkeyboard_getkey();
    if (*key == ' ')
        return TRUE;
    else
        return FALSE;
}

inline WTnode *
VRUniverse::GetRootNode()
{
    return rootNode;
}

inline WTnode *
VRUniverse::GetBannerRootNode()
{
    return bannerRootNode;
}

inline FLAG
VRUniverse::isBannerBusy()
{
    return banner→isBusy();
}
```

```
inline void
VRUniverse::EnableSensors()
{
    ophthal→EnableMouse();
    ophthal→EnableBaron();
}

#endif /* !__VRUniverse_h */
```

B.4 VRUniverse.cc

```
#include "VRUniverse.h"
extern void actions(void);
extern void BannerTaskFn(Banner *banner);
extern void TaskFn(GraphicalObject *obj);

/****************************************************
// VRUniverse::VRUniverse()
// Constructor that parses the command line arguments,
// initializes the universe, and creates the virtual
// patient and ophthalmoscope
/****************************************************/

VRUniverse::VRUniverse(int argc, char **argv,char *eyeParts[4][3],float scale)
{
    WTwindow *window;

    /* Initialize universe state */
    useTexture = TRUE;
    useCrystalEyesVR = FALSE;
    displayConfig = WTDISPLAY_DEFAULT;
    windowConfig = WTWINDOW_DEFAULT;
    lightsFile = "lights";

    /* Parse command line arguments */
    for (int i = 1; i < argc; i++) {
        if (!strcmp(argv[i], "-useCrystal")) {
            useCrystalEyesVR = TRUE;
            displayConfig = WTDISPLAY_CRYSTALEYES;
        }
        else if (!strcmp(argv[i], "-useTexture")) {
            useTexture = TRUE;
        }
        else {
            printf("usage:  vr -useCrystal -useTexture\n");
        }
```

```
    }

    /* Initialize Defaults */
    WTinit_defaults(&argc,argv);

    /* Create Universe */
    WTuniverse_new(displayConfig,windowConfig);
    window = WTuniverse_getwindows();
    if (WTscreen_load("/export/home/dalee/VR/bin/.data/images/homer_1.tga"))
        printf("load screen failed\n");

    /* Open the keyboard */
    WTkeyboard_open();

    /* Enable texturing */
    if (useTexture) {
        WTuniverse_setoption(WTOPTION_3DSCHGTEXEXT,TRUE);
    } else {
        WTuniverse_setrendering(WTRENDER_GOURAUD);
    }

    /* Load background */
    WTuniverse_setbgrgb(0x00,0x00,0x00);

    /* Enable CrystalEyes */
    if (useCrystalEyesVR) {
        WTviewpoint_setparallax(WTuniverse_getviewpoint(), 1.3);
    }
    WTwindow_sethithervalue(window,0.50);

    rootNode = WTuniverse_getrootnodes();
    if (!WTlightnode_load(rootNode,".data/lights"))
        WTerror("Couldn't read lights\n");

    /* Initialize Universe objects */
    ophthal = new Ophthalmoscope(rootNode,".data/path",scale);
    patient = new Patient(rootNode,"face",eyeParts,"eyelid",scale);

    bannerRootNode = WTrootnode_new();
    if (!WTlightnode_load(bannerRootNode,".data/bannerlights"))
        WTerror("Couldn't read lights\n");
    banner = new Banner(bannerRootNode,"banner",1);
    WTtask_new(banner,BannerTaskFn,0.1);
    WTtask_new(banner,TaskFn,1);
}

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

```
// VRUniverse::~VRUniverse()
// Destructor for class
/***************************************************/
VRUniverse::~VRUniverse()
{
    WTuniverse_delete();
}
```

B.5 Patient.h

```
/*******************************************************
// Patient class
//
// Contains a Head object and simulates the virtual
// patient's behavior. Also handles keyboard input from
// the user.
/*******************************************************/
#ifndef PATIENT_H
#define PATIENT_H

#include "Head.h"

class Patient {
public:
    Patient(WTnode *parent,char *face,char *eyeParts[4][3],char *eyeLid,float scale);
    ~Patient();
    void SimulateBehavior(short key);
    void SimulateSymptoms();

private:
    Head *head;
    short doBlink;
    float time;
    short renderingStyle;
    float currTime;
    float startTime;
    int sleep;
    int random;
    short event;
    short lastEvent;
    short eyeRotate;
    short action;
    short rAction;
    float faceAngleSpan;
    short doBehavior;
    float eyeAngle;
```

```
    WTp3 pos;
    WTq dir;

};

#endif /* !__Patient_h */
```

B.6 Patient.cc

```
#include <iostream.h>
#include <stdlib.h>
#include "Patient.h"
#include "actions.h"

/*********************************************************
// Patient::Patient
// Constructor for virtual patient; currently virtual
// patient only consists of a Head object.
/*********************************************************/
Patient::Patient(WTnode *parent,char *face,char *eyeParts[4][3],char *eyeLid,float scale)
{
    head = new Head(parent,face,eyeParts,eyeLid,scale);
    doBlink = FALSE;
    time = WTuniverse_time();
    renderingStyle = 0;
    event = CHOOSE;
    lastEvent = 0;
    eyeRotate = FALSE;
    faceAngleSpan = 0;
    eyeAngle = 20;
    doBehavior = FALSE;

    pos[X] = 5.5;
    pos[Y] = -34;
    pos[Z] = -366.7;

    dir[X] = dir[Y] = dir[Z] = 0;
    dir[3] = 1;
}


/*********************************************************
// Patient::~Patient()
// Destructor for class.
/*********************************************************/
Patient::~Patient()
```

```
{
    delete head;
}


/*******************************************************
// Patient::SimulateBehavior()
// Simulates the behavior of the virtual patient,
// including eye lid blinking and eye and head movement.
// Uses random number generator to simulate random eye
// and head movement. Also handles keyboard input from the
// user.
/*******************************************************/
void
Patient::SimulateBehavior(short key)
{
    float angle = 10;
    float incr = 2;
    float blinkAngle = 40;
    float faceAngle = 60;
    int pause;
    float theta = 0;

    /* Make eyelids blink every 4 secs */
    currTime = WTuniverse_time();
    if (currTime-time > 4 && !doBlink) {
        head->SetAction(EYELID,ROTATEDOWN,blinkAngle,blinkAngle);
        doBlink = TRUE;
    }
    else if (doBlink) {
        time = currTime;
        head->SetAction(EYELID,ROTATEUP,blinkAngle,blinkAngle);
        doBlink = FALSE;
    }

    /* Random Behavior */
    if (doBehavior) {
        if ((event & CHOOSE)==CHOOSE) {
            do {
                event = rand() % NUMEVENTS + 1;
            } while (lastEvent == event);

            lastEvent = event;
        }

        /* Random Head movement */
        if ((event & HEADROTATE)==HEADROTATE) {
```

```
if ((event & SLEEP)==SLEEP) {
    if (currTime-startTime > sleep) {
        event = HEADROTATE | RESET;
        sleep = 2;
        startTime = currTime;
        if (eyeRotate) {
            head→SetAction(EYEBALL,action,angle,angle/5);
            eyeRotate = FALSE;
        }
        if (rAction == ROTATELEFT || rAction == ROTATERIGHT) {
            if (rand() % 3 > 0) {
                head→SetAction(FACE,rAction,angle,angle/5);
                if (rAction == ROTATERIGHT)
                    faceAngleSpan += angle;
                else
                    faceAngleSpan -= angle;
            }
        } else {
            head→SetAction(FACE,rAction,angle,angle/5);
        }
    }
}
else if ((event & RESET)==RESET) {
    if (currTime-startTime > sleep) {
        event = CHOOSE;
    }
}
else {
    /* Rotate left/right 2 times more probable */
    if (rand() % 3 > 0) {
        if (rand() % 4 > 1) {
            action = ROTATELEFT;
            rAction = ROTATERIGHT;
        } else {
            action = ROTATERIGHT;
            rAction = ROTATELEFT;
        }
    }
    else {
        if (rand() % 2 == 0) {
            action = ROTATEUP;
            rAction = ROTATEDOWN;
        } else {
            action = ROTATEDOWN;
            rAction = ROTATEUP;
        }
    }
```

```
            if (rand() % 2 == 0) {
                head→SetAction(EYEBALL,rAction,angle,angle/5);
                eyeRotate = TRUE;
            }
            angle = rand() % 5 + 5;

            if (action == ROTATELEFT || action == ROTATERIGHT) {
                if (action == ROTATERIGHT)
                    theta = angle;
                else
                    theta = -angle;
                if (fabs(faceAngleSpan+theta) < 35) {
                    head→SetAction(FACE,action,angle,angle/5);
                    faceAngleSpan += theta;
                }
            } else {
                head→SetAction(FACE,action,angle,angle/5);
            }

            sleep = rand() % 3 + 2;
            startTime = currTime;
            event |= SLEEP;
        }
    }
    /* Random Eye Movement */
    else if ((event & EYEROTATE)==EYEROTATE) {
        if ((event & SLEEP)==SLEEP) {
            if (currTime-startTime > sleep) {
                event = EYEROTATE | RESET;
                sleep = 2;
                startTime = currTime;
                if (eyeRotate) {
                    head→SetAction(EYEBALL,rAction,eyeAngle,eyeAngle/5);
                    if (rAction == ROTATEUP || rAction == ROTATEDOWN) {
                        head→SetAction(EYELID,rAction,eyeAngle,eyeAngle/5);
                    }
                    eyeRotate = FALSE;
                }
            }
        }
        else if ((event & RESET)==RESET) {
            if (currTime-startTime > sleep)
                event = CHOOSE;
        }
        else {
            random = rand() % 4;
```

```
                if (random == 0) {
                    action = ROTATELEFT;
                    rAction = ROTATERIGHT;
                    eyeAngle = 20;
                } else if (random == 1){
                    action = ROTATERIGHT;
                    rAction = ROTATELEFT;
                    eyeAngle = 20;
                } else if (random == 2){
                    action = ROTATEUP;
                    rAction = ROTATEDOWN;
                    eyeAngle = 10;
                } else if (random == 3){
                    action = ROTATEDOWN;
                    rAction = ROTATEUP;
                    eyeAngle = 10;
                }

                head→SetAction(EYEBALL,action,eyeAngle,eyeAngle/5);
                if (action == ROTATEUP || action == ROTATEDOWN) {
                    head→SetAction(EYELID,action,eyeAngle,eyeAngle/5);
                }
                eyeRotate = TRUE;

                sleep = rand() % 3 + 2;
                startTime = currTime;
                event |= SLEEP;
            }
        }
    }

    /* Handle keyboard input */
    if (key == 'q') {
        WTuniverse_stop();
    }
    else if (key == 'b') {
        doBehavior = !doBehavior;
        head→SetFlash(!doBehavior);
    }
    else if (key == WTKEY_RIGHTARROW) {
        head→SetAction(FACE,ROTATERIGHT,angle,incr);
    }
    else if (key == WTKEY_LEFTARROW) {
        head→SetAction(FACE,ROTATELEFT,angle,incr);
    }
    else if (key == WTKEY_UPARROW) {
        head→SetAction(FACE,ROTATEUP,angle,incr);
```

```
    }
    else if (key == WTKEY_DOWNARROW) {
        head→SetAction(FACE,ROTATEDOWN,angle,incr);
    }
    else if (key == '0') {
        head→SetAction(FACE,YAWRIGHT,angle,incr);
    }
    else if (key == '9') {
        head→SetAction(FACE,YAWLEFT,angle,incr);
    }
    else if (key == 'i') {
        head→SetAction(EYEBALL,MOVEBACK,angle/20,incr/10);
    }
    else if (key == 'm') {
        head→SetAction(EYEBALL,MOVEFORWARD,angle/20,incr/10);
    }
    else if (key == 'j') {
        head→SetAction(EYEBALL,MOVERIGHT,angle/20,incr/10);
    }
    else if (key == 'l') {
        head→SetAction(EYEBALL,MOVELEFT,angle/20,incr/10);
    }
    else if (key == 'I') {
        head→SetAction(EYELID,ROTATEUP,angle*2,angle*2);
    }
    else if (key == 'M') {
        head→SetAction(EYELID,ROTATEDOWN,angle*2,angle*2);
    }

    else if (key == 'Q') {
        if (!renderingStyle)
            WTuniverse_setrendering(WTRENDER_WIREFRAME);
        else
            WTuniverse_setrendering(WTRENDER_GOURAUD |
WTRENDER_TEXTURED);

        renderingStyle = !renderingStyle;
    }

    /* Eye Actions */
    else if (key == 'a') {
        head→SetAction(EYEBALL,ROTATERIGHT,eyeAngle,incr);
    }
    else if (key == 'd') {
        head→SetAction(EYEBALL,ROTATELEFT,eyeAngle,incr);
    }
    else if (key == 'w') {
```

```
        head→SetAction(EYEBALL,ROTATEUP,eyeAngle,incr);
    }
    else if (key == 'x') {
        head→SetAction(EYEBALL,ROTATEDOWN,eyeAngle,incr);
    }
    else if (key == '!') {
        head→SwapEyeTexture("normal");
    }
    else if (key == '@') {
        head→SwapEyeTexture("homer");
    }

    /* Home key */
    else if (key == 'h') {
        WTviewpoint_setposition(WTuniverse_getviewpoint(),pos);
        WTviewpoint_setorientation(WTuniverse_getviewpoint(),dir);
    }

    /* User Interface */
    else if (key == '[') {// decrease parallax

        WTviewpoint_setparallax(WTuniverse_getviewpoint(),
                                WTviewpoint_getparallax(WTuniverse_getviewpoint())
                                * 1.0/1.2);
    }
    else if (key == ']') {// increase parallax

        WTviewpoint_setparallax(WTuniverse_getviewpoint(),
                                WTviewpoint_getparallax(WTuniverse_getviewpoint())
                                * 1.2);
        printf("parallax:  %f\n",WTviewpoint_getparallax(WTuniverse_getviewpoint()));

    }
    else if (key == '{') {// decrease convdistance

        WTviewpoint_setconvdistance(WTuniverse_getviewpoint(),
                                WTview-
point_getconvdistance(WTuniverse_getviewpoint())*
1.0/1.2);
        printf("convdistance:
%f\n",WTviewpoint_getconvdistance(WTuniverse_getviewpoint()));
    }
    else if (key == '}') {// increase convdistance

        WTviewpoint_setconvdistance(WTuniverse_getviewpoint(),
                                WTview-
point_getconvdistance(WTuniverse_getviewpoint())
```

```
                    * 1.2);
        printf("convdistance:
%f\n",WTviewpoint_getconvdistance(WTuniverse_getviewpoint()));
    }

    else if (key == '!') {
        head->IrisAnimate(0);
    }
    else if (key == '@') {
        head->IrisAnimate(1);
    }
    else if (key == '#') {
        head->IrisAnimate(2);
    }

    else if (key == '1') {
        head->SwapEyeTexture("normal");
    }
    else if (key == '2') {
        head->SwapEyeTexture("glaucoma");
    }
    else if (key == '3') {
        head->SwapEyeTexture("papilledema");
    }
    else if (key == '4') {
        head->SwapEyeTexture("macularscar");
    }
    else if (key == '5') {
        head->SwapEyeTexture("exudates");
    }

}


/*********************************************************
// Patient::SimulateSymptoms()
// Not implemented this version yet.
/*********************************************************/
void
Patient::SimulateSymptoms()
{
}
```

B.7 Head.h

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

```
// Head class
// Contains a Face, a left and right Eye, and a left and
// right EyeLid objects. Initializes these parts of the
// virtual patient's head and provides the interface to
// these components.
/*******************************************************/
#ifndef HEAD_H
#define HEAD_H

#include "Face.h"
#include "Eye.h"
#include "EyeLid.h"

class Head {
public:
    Head(WTnode *parent,char *face, char *eyeParts[4][3], char *eyeLid,float scale);
    ~Head();
    void SetAction(short type,short action,float span, float incr);
    void SwapEyeTexture(char *bitmap);
    void IrisAnimate(int switchVal);
    void SetFlash(FLAG flag);

private:
    /* Scene graph variables */
    WTnode *groupHeadNode;
    WTnode *xformSepLeftEyeNode;
    WTnode *xformSepRightEyeNode;

    /* Head parts */
    Face *face;
    Eye *leftEye;
    Eye *rightEye;
    EyeLid *leftEyeLid;
    EyeLid *rightEyeLid;
};


inline void
Head::IrisAnimate(int switchVal)
{
    leftEye->IrisAnimate(switchVal);
    rightEye->IrisAnimate(switchVal);
}

inline void
Head::SwapEyeTexture(char *bitmap)
{
```

```
    char bitmapLeft[20];
    char bitmapRight[20];

    sprintf(bitmapLeft,"%s%s",bitmap,"_l");
    sprintf(bitmapRight,"%s%s",bitmap,"_r");

    leftEye->SwapEyeTexture(bitmapLeft,TRUE,FALSE);
    rightEye->SwapEyeTexture(bitmapRight,TRUE,FALSE);
}

void inline
Head::SetFlash(FLAG flag)
{
    leftEye->SetFlash(flag);
    rightEye->SetFlash(flag);
}

#endif /* !__Head_h */
```

B.8 Head.cc

```
#include <iostream.h>
#include <string.h>
#include "Head.h"
#include "actions.h"

/* Task Functions (TaskFn.cc) */
extern void TaskFn(GraphicalObject *obj);
extern void RedFlashTaskFn(Eye *eye);

/* EyeBall positions */
WTp3 left = {48.5134, -41.9048,-72.6525};
WTp3 right = {-29.7258, -41.9048,-75.9525};
WTp3 leftLid = {48.5134, -41.9048,-75.6525};
WTp3 rightLid = {-29.7258, -41.9048,-78.9525};


/*******************************************************
// Head::Head
// Constructor for class that takes a list of file names
// that contain the geometric model data for the head parts
// and creates the associated scene graph hierarchy.
*******************************************************/
Head::Head(WTnode *parent,char *faceFile, char *eyeParts[4][3],char *eyeLidFile,float
scale)
{
```

```
    WTnode *xform;

    /* Create group Head node */
    groupHeadNode = WTgroupnode_new(parent);

    /* Create hierarchy */
    face = new Face(groupHeadNode,faceFile,scale);

    /* Create Eyes */
    xformSepLeftEyeNode = WTsepnode_new(groupHeadNode);
    leftEye = new Eye(xformSepLeftEyeNode,eyeParts,left,scale);
    xformSepRightEyeNode = WTsepnode_new(groupHeadNode);
    rightEye = new Eye(xformSepRightEyeNode,eyeParts,right,scale);

    /* Create EyeLids */
    xform = WTsepnode_new(groupHeadNode);
    leftEyeLid = new EyeLid(xform,eyeLidFile,leftLid,scale);
    xform = WTsepnode_new(groupHeadNode);
    rightEyeLid = new EyeLid(xform,eyeLidFile,rightLid,scale);

    /* Assign Task Functions */
    WTtask_new(face,TaskFn,1);
    WTtask_new(leftEye,RedFlashTaskFn,0.5);
    WTtask_new(rightEye,RedFlashTaskFn,0.5);
    WTtask_new(leftEyeLid,TaskFn,1);
    WTtask_new(rightEyeLid,TaskFn,1);

    SwapEyeTexture("normal");
}


/*********************************************************
// Head::~Head()
// Destructor for class
*********************************************************/
Head::~Head()
{
    delete face;
    delete leftEye;
    delete rightEye;

    /* Delete object hierarchy ?? */
}


/*********************************************************
// Head::SetAction()
```

```
// A wrapper function that sets the action for the
// particular head component. Types of actions are defined
// in actions.h.
*******************************************************/
void
Head::SetAction(short type,short action,float span, float incr )
{
    switch (type) {
      case FACE:
          face→SetAction(action,span,incr);
          break;
      case EYEBALL:
          leftEye→SetEyeBallAction(action,span,incr);
          rightEye→SetEyeBallAction(action,span,incr);
          break;
      case EYELID:
          leftEyeLid→SetAction(action,span,incr);
          rightEyeLid→SetAction(action,span,incr);
          break;
    }
}
```

B.9 Face.h

```
/*******************************************************
// Face class
// Inherits from the GraphicalObject class and simply
// handles the graphical representation of the virtual
// patient's face.
/*******************************************************/
#ifndef FACE_H
#define FACE_H

#include "GraphicalObject.h"

class Face : public GraphicalObject {
public:
    Face(WTnode *parent,char *faceFileName,float scale);
    ~Face();
};

#endif /* !__Face_h */
```

B.10 Face.cc

```
#include <iostream.h>
#include "Face.h"

/*******************************************************
// Face::Face()
// Constructor for class that first calls the GraphicalObject
// constructor to load the geometry data and then pre-builds
// the geometry for better rendering performance.
/*******************************************************/
Face::Face(WTnode *parent,char *faceFileName,float scale)
    : GraphicalObject(parent,faceFileName,scale)
{
    GeometryPreBuild();
}


Face::~Face()
{
}
```

B.11 EyeLid.h

```
/*******************************************************
// EyeLid class
// Inherits from the GraphicalObject class and simply
// handles the graphical representation of the virtual
// patient's eye lids.
/*******************************************************/
#ifndef EYELID_H
#define EYELID_H

#include "GraphicalObject.h"

class EyeLid : public GraphicalObject {
public:
    EyeLid(WTnode *parent,char *eyeLidFileName,WTp3 pos,float scale);
    ~EyeLid();
};

#endif /* !__EyeLid_h */
```

B.12 EyeLid.cc

```
#include <iostream.h>
#include "EyeLid.h"

/********************************************************
// EyeLid::EyeLid()
// Constructor for class that first calls the GraphicalObject
// constructor to load the geometry data and then positions
// the eye lid at the proper position (eyelid geometry model
// initially centered around origin). Finally, sets the pivot
// of the eye lid object to correspond with its new position.
/********************************************************/
EyeLid::EyeLid(WTnode *parent,char *eyeLidFileName,WTp3 pos,float scale)
    : GraphicalObject(parent,eyeLidFileName,scale)
{
    GeometryTranslate(pos);
    GeometrySetMidPoint(pos);
    SetPivot(midPoint);
}

EyeLid::~EyeLid()
{
}
```

B.13 Eye.h

```
/********************************************************
// Eye class
// Contains the EyeBall, Iris, and Retina objects.
// Member functions handle the changing of the iris
// and retina graphical representations. Uses task function
// RedFlashTaskFn() to simulate red-reflex behavior.
/********************************************************/
#ifndef EYE_H
#define EYE_H

#include "EyeBall.h"
#include "Iris.h"
#include "Retina.h"

class Eye {
public:
    Eye(WTnode *parent,char *parts[4][3],WTp3 pos,float scale);
    ~Eye();
    void SetRedFlash(int whichRetina);
```

```
    void IrisAnimate(int switchVal);
    void SetEyeBallAction(short action,float span,float incr);
    void SwapEyeTexture(char *bitmap,FLAG shading,FLAG transparent);
    void GetMidPoint(WTp3 p);
    void GetLineOfSight(WTp3 p);
    void CalcAngleBounds(float dist,float *lower,float *upper,float *thres);
    short DoFlash();
    void SetFlash(FLAG flag);

private:
    /* Scene graph components */
    WTnode *ambientLight;
    WTnode *xformSepEyeLid;

    /* Eye parts */
    EyeBall *eyeBall;
    Iris *iris;
    Retina *retina;

    /* Flag for enabling red-flash behavior */
    short doFlash;
    char *texture;
};

inline void
Eye::SetRedFlash(int whichRetina)
{
    retina->SetWhichChild(whichRetina);
}

inline void
Eye::IrisAnimate(int switchVal)
{
    iris->SetWhichChild(switchVal);
}

inline void
Eye::SetEyeBallAction(short action,float span,float incr)
{

    eyeBall->SetAction(action,span,incr);
}

inline void
Eye::SwapEyeTexture(char *bitmap,FLAG shading,FLAG transparent)
{
    retina->SwapEyeTexture(bitmap,shading,transparent);
```

```
}

inline void
Eye::GetMidPoint(WTp3 p)
{
    WTp3_copy(eyeBall→midPoint,p);
}

inline void
Eye::GetLineOfSight(WTp3 p)
{
    WTp3_copy(eyeBall→centerLine,p);
}

inline void
Eye::CalcAngleBounds(float dist,float *lower,float *upper,float *thres)
{
    retina→CalcAngleBounds(dist,lower,upper,thres);
}

inline short
Eye::DoFlash()
{
    return doFlash;
}

inline void
Eye::SetFlash(FLAG flag)
{
    doFlash = flag;
}

#endif /* !__Eye_h */
```

B.14 Eye.cc

```
#include <iostream.h>
#include "Eye.h"
#include "actions.h"
extern void TaskFn(GraphicalObject *obj);

/********************************************************
// Eye::Eye()
// Constructor for class that loads the EyeBall and Iris
// objects into the scene graph, followed by the negative
// ambient light. Finally, the retina object is added last.
```

```
/*********************************************************/
Eye::Eye(WTnode *parent,char *parts[4][3],WTp3 pos,float scale)
{
    WTp3 p;
    scale *= 0.96;

    /* Create EyeBall and Iris */
    eyeBall = new EyeBall(parent,parts[EYE_LIST][0],pos,scale);
    iris = new Iris(parent,parts[IRIS_LIST],pos,scale);

    /* Create negative ambient light */
    ambientLight = WTlightnode_newambient(parent);
    WTlightnode_setintensity(ambientLight,-0.4);

    /* Create Retina */
    retina = new Retina(parent,parts[RETINA_LIST],pos,scale);

    /* Move Iris to correct position */
    WTp3_invert(eyeBall->midPoint,p);
    iris->GeometryTranslate(p);

    /* Initialize red-flash variables */
    doFlash = TRUE;
    texture = new char[50];

    /* Assign Task Function */
    WTtask_new(eyeBall,TaskFn,1);
}

Eye::~Eye()
{
}
```

B.15 EyeBall.h

```
/*********************************************************
// EyeBall class
// Inherits from the GraphicalObject class and simply
// handles the graphical representation of the virtual
// patient's eyeball.
/*********************************************************/
#ifndef EYEBALL_H
#define EYEBALL_H

#include "GraphicalObject.h"
```

```
class EyeBall : public GraphicalObject {
public:
    EyeBall(WTnode *parent,char *eyeFileName, WTp3 pos,float scale);
    ~EyeBall();
};


#endif /* !__EyeBall_h */
```

B.16 EyeBall.cc

```
#include <iostream.h>
#include "EyeBall.h"
#include "actions.h"

/*******************************************************
// EyeBall::EyeBall()
// Constructor for class that first calls the GraphicalObject
// constructor to load the geometry data and then positions
// the eyeball at the proper position (eyeball geometry model
// initially centered around origin). Finally, sets the pivot
// of the eyeball object to correspond with its new position.
/*******************************************************/
EyeBall::EyeBall(WTnode *parent,char *eyeBallFileName, WTp3 pos,float scale)
    : GraphicalObject(parent,eyeBallFileName,scale)
{
    GeometryTranslate(pos);
    GeometrySetMidPoint(pos);
    SetPivot(midPoint);
}

EyeBall::~EyeBall()
{
}
```

B.17 Iris.h

```
/*******************************************************
// Iris class
// Inherits from the GraphicalObject class and simply
// handles the graphical representation of the virtual
// patient's iris.
/*******************************************************/
#ifndef IRIS_H
#define IRIS_H
```

```
#include "GraphicalObject.h"

class Iris : public GraphicalObject {
public:
    Iris(WTnode *parent,char **irisFileName, WTp3 pos,float scale);
    ~Iris();

private:
    WTnode *switchNode;
};


#endif /* !__Iris_h */
```

B.18 Iris.cc

```
#include <iostream.h>
#include "Iris.h"
#include "actions.h"

/*******************************************************
// Iris::Iris()
// Constructor for class that first calls the GraphicalObject
// constructor to load the geometry data for multiple
// representations of the iris. Then switches to initial
// iris representation.
/*******************************************************/
Iris::Iris(WTnode *parent,char **irisFileName,WTp3 pos,float scale)
    : GraphicalObject(parent,irisFileName,NUM_IRISES,scale)
{
    SetWhichChild(IRIS_50);
}

Iris::~Iris()
{
}
```

B.19 Retina.h

```
/*******************************************************
// Retina class
// Inherits from the GraphicalObject class and handlesn
// the graphical representation of the virtual patient's
// retina. Also is responsible for calculating the angles
```

```
// for which the red-reflex/fundus image should be seen.
/*******************************************************/
#ifndef RETINA_H
#define RETINA_H

#include "GraphicalObject.h"

class Retina : public GraphicalObject {
public:
    Retina(WTnode *parent,char **retinaFileName,WTp3 pos,float scale);
    ~Retina();
    void CalcAngleBounds(float dist,float *lower, float *upper,float *thres);
    void SwapEyeTexture(char *bitmap,FLAG shading,FLAG transparent);

private:

    /* Variables for interpolation */
    float lowerSlope;
    float lowerIntercept;
    float upperSlope;
    float upperIntercept;

};

#endif /* !__Retina_h */
```

B.20 Retina.cc

```
#include <iostream.h>
#include "Retina.h"
#include "actions.h"

/*******************************************************
// Retina::Retina()
// Constructor that first calls the GraphicalObject
// constructor to load the multiple representation of the
// retina. Then calculate values for linear interpolation
// of the boundary angles for the red-reflex behavior.
/*******************************************************/
Retina::Retina(WTnode *parent,char **retinaFileName,WTp3 pos,float scale)
    : GraphicalObject(parent,retinaFileName,NUM_RETINAS,scale)
{
    SetWhichChild(RETINA_BLACK);

    lowerSlope = (lower1[Y]-upper1[Y])/(lower1[X]-upper1[X]);
    lowerIntercept = lower1[X] - lowerSlope*lower1[Y];
```

```
    upperSlope = (lower2[Y]-upper2[Y])/(lower2[X]-upper2[X]);
    upperIntercept = lower2[X] - upperSlope*lower2[Y];

}


/*******************************************************
// Retina::~Retina
// Destructor for class
/*******************************************************/
Retina::~Retina()
{
}


/*******************************************************
// Retina::CalcAngleBounds()
// Calculates the bounds for the red-flex/fundus image
// angles based on the current distance. Current, only
// three ranges: extremely close to eye to see fundus image
// (RETINATHRESHOLD), close but not close enough to see fundus
// image (FLASHTHRESHOLD), and far but still able to produce
// red-reflex (DIRTHRESHOLD).
/*******************************************************/
void
Retina::CalcAngleBounds(float dist,float *lower, float *upper,float *thres)
{
    /* Angles not interpolated yet; just hardcode for now */
    /* *lower = lowerSlope*dist + lowerIntercept; */
    /* *upper = upperSlope*dist + upperIntercept; */

    /* Default angles; normal flash */
    *lower = 10.0*PI/180.0;
    *upper = 15.0*PI/180.0;
    *thres = DIRTHRESHOLD;

    /* Angles for fundus image */
    if (dist < RETINATHRESHOLD) {
        *lower = 0;
        *upper = 65.0*PI/180.0;
        *thres = 35.0*PI/180.0;
    }
    /* Angles for close red-reflex */
    else if (dist < FLASHTHRESHOLD) {
        *lower = 8.0*PI/180.0;
        *upper = 20.0*PI/180.0;
```

```
        *thres = 6.5*PI/180.0;
    }

}


/*****************************************************
// Retina::SwapEyeTexture()
// Changes the current fundus image to a new image
// specified by "bitmap".
/*****************************************************/
void
Retina::SwapEyeTexture(char *bitmap,FLAG shading,FLAG transparent)
{
    WTnode *flashNode = WTnode_getchild(node,RETINA_FLASH);
    WTgeometry_changetexture(WTnode_getgeometry(flashNode),
                             bitmap,shading,transparent);
}
```

B.21 Ophthalmoscope.h

```
/*****************************************************
// Ophthalmoscope class
// Contains an EyePiece object that represents the
// viewpoint of the user.
/*****************************************************/
#ifndef OPHTHALMOSCOPE_H
#define OPHTHALMOSCOPE_H

#include "EyePiece.h"

class Ophthalmoscope {
public:
    Ophthalmoscope(WTnode *parent,char *pathFileName,float scale);
    ~Ophthalmoscope();
    void Simulate(short key);
    void EnableMouse();
    void EnableBaron();

private:
    EyePiece *eyePiece;
};


inline void
Ophthalmoscope::EnableMouse()
```

```
{
    eyePiece→EnableMouse();
}

inline void
Ophthalmoscope::EnableBaron()
{
    eyePiece→EnableBaron();
}

#endif /* !__Ophthal_h */
```

B.22 Ophthalmoscope.cc

```
#include <iostream.h>
#include "Ophthalmoscope.h"

/*******************************************************
// Ophthalmoscope::Ophthalmosopce()
// Constructor for the class that creates the EyePiece obj.
/*******************************************************/
Ophthalmoscope::Ophthalmoscope(WTnode *parent,char *pathFileName,float scale)
{
    eyePiece = new EyePiece(parent,"eyepiece",scale);
}


Ophthalmoscope::~Ophthalmoscope()
{
    delete eyePiece;
}


void
Ophthalmoscope::Simulate(short key)
{

}
```

B.23 EyePiece.h

```
/*******************************************************
// EyePiece class
// Controls the user's viewpoint during the simulation
```

```
// via the spotLightTaskFn (see TaskFn.cc). Also handles
// the enabling of the input sensors and inherits from
// GraphicalObject class.
/******************************************************/
#ifndef EYEPIECE_H
#define EYEPIECE_H

#include "GraphicalObject.h"

class EyePiece : public GraphicalObject {
public:
    EyePiece(WTnode *parent,char *eyePieceFileName,float scale);
    ~EyePiece();
    void EnableMouse();
    void EnableBaron();

private:
    /* Scene graph variables */
    WTnode *spotLightSepNode;
    WTnode *spotLightXformNode;
    WTnode *spotLightNode;

    /* Input sensors */
    WTsensor *mouse;
    WTsensor *baron;

};

#endif /* !__EyePiece_h */
```

B.24 EyePiece.cc

```
#include <iostream.h>
#include "EyePiece.h"
void spotLightTaskFn(WTnode *spotLightNode);

/*******************************************************
// EyePiece::EyePiece()
// Constructor for class that first calls the constructor
// of the GraphicalObject class and then creates the
// associated scene graph.
/******************************************************/
EyePiece::EyePiece(WTnode *parent,char *eyePieceFileName,float scale)
    : GraphicalObject(parent,eyePieceFileName,scale)
{
    /* Create XformSeparator for light node */
```

```
    spotLightSepNode = WTxformsepnode_new(parent);

    /* Create spot light transform node */
    spotLightXformNode = WTxformnode_new(spotLightSepNode);
    WTnode_setname(spotLightXformNode,"spotLightXformNode");

    /* Create spot light */
    spotLightNode = WTlightnode_newspot(spotLightSepNode);
    WTnode_setname(spotLightNode,"spotLightNode");
    WTlightnode_setangle(spotLightNode,PI/9.0);
    WTlightnode_setattenuation(spotLightNode,1.0,0.005,0.0);

    /* Add task to spot light */
    WTtask_new(spotLightNode,spotLightTaskFn,0.1);

    /* Make "view object" invisible */
    SetVisibility(FALSE);
}


EyePiece::~EyePiece()
{
}


/*********************************************************
// EyePiece::EnableMouse()
// Enables the 2D mouse and attaches it to the viewpoint
/*********************************************************/
void
EyePiece::EnableMouse()
{
    /* Initialize mouse sensor */
    mouse = WTmouse_new();

    if (!mouse)
        WTerror("Couldn't find mouse\n");
    else
        WTsensor_setangularrate(mouse,PI/90.0);

    /* Attatch sensor to viewpoint */
    WTviewpoint_addsensor(WTuniverse_getviewpoint(), mouse);
    WTsensor_setsensitivity(mouse,
                            0.01f*WTnode_getradius(WTuniverse_getrootnodes()));
}
```

```
/*******************************************************
// EyePiece::EnableBaron()
// Enables the 3D mouse and attaches it to the viewpoint
/*******************************************************/
void
EyePiece::EnableBaron()
{

    /* Initialize flying baron sensor */
    baron = WTbaron_new(SERIAL1);
    if (!baron)
        WTerror("Couldn't find 3Dmouse\n");

    /* Add sensor to viewpoint */
    WTviewpoint_addsensor(WTuniverse_getviewpoint(), baron);
    WTsensor_setsensitivity(baron,
                            0.04f*WTnode_getradius(WTuniverse_getrootnodes()));
}
```

B.25 actions.h

```
#ifndef ACTIONS_H
#define ACTIONS_H
#include "wt.h"

/* Eye component definitions */
const unsigned short FACE = 0;
const unsigned short EYEBALL = 1;
const unsigned short EYELID = 2;

/* Axis definitions */
const unsigned short TRANSLATE =16; /* 10000 */
const unsigned short X_AXIS = 8; /* 01000 */
const unsigned short Y_AXIS = 4; /* 00100 */
const unsigned short Z_AXIS = 2; /* 00010 */
const unsigned short POS = 1; /* 00001 */
const unsigned short NEG = 0; /* 00000 */

/* Rotation definitions */
const unsigned short ROTATERIGHT = Y_AXIS | POS;
const unsigned short ROTATELEFT = Y_AXIS | NEG;
const unsigned short ROTATEDOWN = X_AXIS | POS;
const unsigned short ROTATEUP = X_AXIS | NEG;
const unsigned short YAWLEFT = Z_AXIS | POS;
const unsigned short YAWRIGHT = Z_AXIS | NEG;
```

```
/* Translation definitions */
const unsigned short MOVEFORWARD = TRANSLATE | Z_AXIS | POS;
const unsigned short MOVEBACK = TRANSLATE | Z_AXIS | NEG;
const unsigned short MOVERIGHT = TRANSLATE | X_AXIS | POS;
const unsigned short MOVELEFT = TRANSLATE | X_AXIS | NEG;
const unsigned short MOVEDOWN = TRANSLATE | Y_AXIS | POS;
const unsigned short MOVEUP = TRANSLATE | Y_AXIS | NEG;

/* Action definitions */
const short ROT = 0;
const short TRANS = 1;

/* File Editor constants */
const unsigned short SEARCH = 0;
const unsigned short MALLOC = 1;
const unsigned short PARSE = 2;

/* Eyeball Constants */
const unsigned short EYE_LIST = 0;
const unsigned short RETINA_LIST = 1;
const unsigned short IRIS_LIST = 2;
const unsigned short EYELID_LIST = 3;

/* Animation Constants */
const unsigned short NUM_RETINAS = 3;
const unsigned short NUM_IRISES = 3;
const unsigned short NUM_EYELIDS = 1;
const unsigned short RETINA_BLACK = 0;
const unsigned short RETINA_RED = 1;
const unsigned short RETINA_FLASH = 2;
const unsigned short IRIS_75 = 0;
const unsigned short IRIS_50 = 1;
const unsigned short IRIS_25 = 2;

/* Angle Cone for Flash */
const float lower1[2] = {400*PI/180.0,10*PI/180.0};
const float lower2[2] = {400*PI/180.0,15*PI/180.0};
const float upper1[2] = {40*PI/180.0,5*PI/180.0};
const float upper2[2] = {40*PI/180.0,50*PI/180.0};
const float FLASHTHRESHOLD = 500;
const float RETINATHRESHOLD = 20;
const float DIRTHRESHOLD = 4.0*PI/180.0;

/* Behavior constants */
const unsigned short CHOOSE =16; /* 10000 */
const unsigned short SLEEP = 8; /* 01000 */
const unsigned short RESET = 4; /* 00100 */
```

```
const unsigned short HEADROTATE = 1; /* 00001 */
const unsigned short EYEROTATE = 2; /* 00010 */
const unsigned short NUMEVENTS = 2;

#endif /* !__actions_h */
```

B.26 TaskFn.cc

```
/*******************************************************
// This file contains all the C task functions for the
// entire simulation.
/*******************************************************/
#include "TaskFn.h"
#include "GraphicalObject.h"
#include "Eye.h"
#include "VRUniverse.h"
#include "Banner.h"
#include "math.h"

/*******************************************************
// TaskFn()
// Task Function that update a GraphicalObject's
// transformation state variables.
/*******************************************************/
void
TaskFn(GraphicalObject *obj)
{
    short axis;
    float thetaX=0,thetaY=0,thetaZ=0;
    WTq q;
    WTp3 tmp;
    for (axis = X; axis <= Z; axis++) {
        /* Update Obj Translations */
        if (obj->axisVal[TRANS][axis]) {
            obj->currVal[TRANS][axis] += obj->incrVal[TRANS][axis];

            /* Calculate translation */
            obj->p[axis] = obj->signVal[TRANS][axis]*obj->incrVal[TRANS][axis];

            /* Object needs updating */
            obj->dirty = TRUE;

            /* Terminate translations when completed */
            if (obj->currVal[TRANS][axis] >= obj->totVal[TRANS][axis])
                obj->axisVal[TRANS][axis] = 0;
```

```
        }

        /* Update Object Rotations */
        if (obj→axisVal[ROT][axis]) {
            obj→currVal[ROT][axis] += obj→incrVal[ROT][axis];

            /* Calculate rotations */
            if (axis == X)
                thetaX = obj→signVal[ROT][X]*obj→incrVal[ROT][X];
            else if (axis == Y)
                thetaY = obj→signVal[ROT][Y]*obj→incrVal[ROT][Y];
            else if (axis == Z)
                thetaZ = obj→signVal[ROT][Z]*obj→incrVal[ROT][Z];

            /* Object needs updating */
            obj→dirty = TRUE;

            /* Terminate rotations when completed */
            if (obj→currVal[ROT][axis] ≥ obj→totVal[ROT][axis])
                obj→axisVal[ROT][axis] = 0;
        }
    }

    /* Only update object if needed */
    if (obj→dirty) {
        WTnode_rotate(obj→xformNode,thetaY,thetaX,thetaZ,WTFRAME_LOCAL);
        WTnode_translate(obj→xformNode,obj→p,WTFRAME_LOCAL);
        obj→dirty = FALSE;

        /* Update orientation vector */
        WTeuler_2q(thetaX*PI/180.0,thetaY*PI/180.0,thetaZ*PI/180.0,q);
        WTp3_rotate(obj→centerLine,q,tmp);
        WTp3_copy(tmp,obj→centerLine);
    }

}


/*******************************************************
// RedFlashTaskFn()
// Task Function that simulates the behavior of the
// red-reflex and fundus image. "theta" is the angle between
// the viewpoint position and the eye's line of sight. "dir"
// is the angle between the viewpoint position vector and the
// the viewpoint direction vector. "dir" is necessary in order
// to make sure the viewpoint is pointing in the general
// direction of the viewpoint position vector.
```

```
/*****************************************************/
void
RedFlashTaskFn(Eye *eye)
{
    WTp3 p1,p2,viewPointPos,midPoint,viewPointDir;
    double theta,dir;
    float lower,upper,thres,dist;

    if (eye->DoFlash()) {
        /* Get line of sight (p2) */
        eye->GetLineOfSight(p2);

        /* Get viewpoint direction vector (viewPointDir) */
        WTviewpoint_getdirection(WTuniverse_getviewpoint(),p1);
        WTp3_invert(p1,viewPointDir);

        /* Calculate vector from viewpoint position to middle of eye */
        WTviewpoint_getposition(WTuniverse_getviewpoint(),viewPointPos);
        eye->GetMidPoint(midPoint);
        WTp3_subtract(viewPointPos,midPoint,p1);

        /* Calculate angle between viewpoint position vector (p1)
           and line of sight (p2) */
        theta = acos(WTp3_dot(p1,p2)/((WTp3_mag(p1)*(WTp3_mag(p2)))));

        /* Calculate angle between viewpoint position vector (p1)
           and viewpoint direction */
        dir =
acos(WTp3_dot(viewPointDir,p1)/((WTp3_mag(viewPointDir)*(WTp3_mag(p1)))));
        dist = WTp3_distance(midPoint,viewPointPos);

        /* Get angle bounds */
        eye->CalcAngleBounds(dist,&lower,&upper,&thres);

        /* Switch retina representation */
        eye->SetRedFlash(RETINA_BLACK);
        if (dir < thres && theta < upper && theta > lower) {
            if (dist < RETINATHRESHOLD) {
                eye->SetRedFlash(RETINA_FLASH);
            }
            else if (dist < FLASHTHRESHOLD) {
                eye->SetRedFlash(RETINA_RED);
            }
        }
    } else {
        eye->SetRedFlash(RETINA_BLACK);
    }
```

```
}


/*******************************************************
// UniverseTaskFn()
// Task Function that queries the keyboard and updates
// the patient's behavior.
/*******************************************************/
void
UniverseTaskFn(VRUniverse *universe)
{
    short key = WTkeyboard_getkey();

    universe->patient->SimulateBehavior(key);
}


/*******************************************************
// spotLightTaskFn()
// Task Function that updates the spotlight to
// correspond with the user's viewpoint.
/*******************************************************/
void
spotLightTaskFn(WTnode *spotLightNode)
{
    WTp3 p,q;

    /* Position spotlight with viewpoint */
    WTviewpoint_getposition(WTuniverse_getviewpoint(),p);
    WTlightnode_setposition(spotLightNode,p);

    /* Orient spotlight with viewpoint */
    WTviewpoint_getorientation(WTuniverse_getviewpoint(),q);
    WTq_2dir(q,p);
    WTlightnode_setdirection(spotLightNode,p);
}


/*******************************************************
// BannerTaskFn()
// Task Function that updates the opening sequence.
/*******************************************************/
void
BannerTaskFn(Banner *banner)
{
    banner->Update();
}
```

B.27 main.cc

```
#include <iostream.h>
#include "main.h"
#include "VRUniverse.h"
#include "actions.h"
extern void UniverseTaskFn(VRUniverse *universe);

int
main(int argc, char **argv)
{
    short mode;
    char *configFile = ".data/config";
    WTnode *bannerRoot;
    WTwindow *window;
    float scale = 10;
    char *eyeParts[4][3] =
{{"eyeball"},{"eye_black","eye_red","eye_flash"},{"iris_75","iris_50","iris_25"},{"eyelid"}};

    /* Create Universe */
    VRUniverse *universe = new VRUniverse(argc,argv,eyeParts,scale);
    mode = PHASEII;
    WTtask_new(universe,UniverseTaskFn,0.1);

    /* Display Title Sequence */
    window = WTuniverse_getwindows();
    WTwindow_setrootnode(window,universe->GetBannerRootNode());
    universe->Ready();
    while (universe->isBannerBusy()) {
        universe->Go1();
    }

    /* Display Virtual Ophthalmoscope Environment */
    WTwindow_setrootnode(window,universe->GetRootNode());
    WTwindow_zoomviewpoint( WTuniverse_getwindows());

    /* Run Simulation */
    universe->EnableSensors();
    universe->Go();

    /* Clean up */
    WTuniverse_delete();
    cout << "That's all folks\n";
}
```

References

[1] David Paton, Barry Hyman, Johnny Justice. *Introduction to Ophthalmoscopy.* Upjohn Company, Kalamazoo, Michigan (1976).

[2] K.Pimentel and K.Teixeira, *Virtual Reality:Through the Looking Glass.* McGraw-Hill, New York (1993).

[3] Michael L. Rhodes. "Computer Graphics and Medicine: A Complex Partnership". *IEEE Computer Graphics and Applications*, 1997 January, 17(1):22-28.

[4] Rolf Ziegler, Georg Fischer, Wolfgang Muller, and Martin Gobel. "Virtual Reality Arthroscopy Training Simulator". Fraunhofer Institute for Computer Graphics, Germany. *Computers in Biology and Medicine*, 1995, (25)2: 193-203.

[5] I.W.Hunter, L.A.Jones, M.A.Sagar, S.R.LaFontaine, and P.J.Hunter. "Ophthalmic Microsurgical Robot and Associated Virtual Environment". Massachusetts Institute of Technology. *Computers in Biology and Medicine*, 1995, (25)2:173-182.

[6] Sina Mostafawy, Omid Kermani, and Holger Lubatschowski. "Virtual Eye: Retinal Image Visualization of the Human Eye". German National Research Center for Information Technology. *IEEE Computer Graphics and Applications*, 1997 January, 17(1):8-12.

[7] Gregory L. Merril, Virginia L. Barker. "Virtual Reality Debuts in the Teaching Laboratory in Nursing". Nursing Department, State University of New York. *Journal of Intravenous Nursing*, 1996, (19)4:182-187.

[8] *WorldToolKit Reference Manual: Release 6.* Sense8 Corporation, Mill Valley, CA. July 1996.

[9] Grady Booch. *Object Oriented Design: With Applications.* Benjamin/Cummings Publishing Company, Inc., Redwood City, California (1991).