Rochester Institute of Technology

RIT Scholar Works

Theses

9-1-1998

Computer synthesis of spectroradiometric images for color imaging systems analysis

Garrett M. Johnson

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

Johnson, Garrett M., "Computer synthesis of spectroradiometric images for color imaging systems analysis" (1998). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

COMPUTER SYNTHESIS OF SPECTRORADIOMETRIC IMAGES FOR COLOR IMAGING SYSTEMS ANALYSIS

Garrett M. Johnson
B.S. Rochester Institute of Technology
(1996)

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in the Chester F. Carlson Center for Imaging Science
of the College of Science
Rochester Institute of Technology

September 1998

Signature of Author________________________________

Accepted by Roy Berns ________________ 11/13/98
Coordinator M.S. Color Science Program Date

MUNSELL COLOR SCIENCE LABORATORY
CENTER FOR IMAGING SCIENCE
COLLEGE OF SCIENCE
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK

CERTIFICATE OF APPROVAL

M.S. DEGREE THESIS

The M.S. Degree Thesis of Garrett M. Johnson
has been examined and approved by the
thesis committee as satisfactory for the
thesis requirement for the
Master of Science degree

____

Dr. Mark D. Fairchild, Thesis Advisor

____

Dr. Ethan D. Montag

11/13/98
Date

THESIS RELEASE PERMISSION
ROCHESTER INSTITUTE OF TECHNOLOGY
COLLEGE OF SCIENCE
CHESTER F. CARLSON
CENTER FOR IMAGING SCIENCE
MUNSELL COLOR SCIENCE LABORATORY

COMPUTER SYNTHESIS OF SPECTRORADIOMETRIC IMAGES FOR COLOR IMAGING SYSTEMS ANALYSIS

I, Garrett M. Johnson, hereby grand permission to the Wallace Memorial Library of R.I.T. to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use of profit.

Signature:__________________________

Date: 11/13/98

COMPUTER SYNTHESIS OF SPECTRORADIOMETRIC IMAGES FOR COLOR IMAGING SYSTEMS ANALYSIS

Garrett M. Johnson

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in the Chester F. Carlson Center for Imaging Science
of the College of Science
Rochester Institute of Technology

ABSTRACT

A technique to perform full spectral based color calculations through an extension of OpenGL has been created. This method of color computations is more accurate than the standard RGB model that most computer graphics algorithms utilize. By maintaining full wavelength information in color calculations, it is also possible to interactively simulate and display many important color phenomena such as metamerism and fluorescence. This technique is not limited to creating simple images suitable for interactive display, however. Using this extension, it is also possible to synthesize spectroradiometric images of arbitrary spatial and spectral resolution, for use in color imaging system analysis.

Introduction ____ *8*

Background ____ *8*

Theory ____ *9*

Computer Generated Color ____ 9

Spectral Information ____ *12*

Wavelength Sampling ____ **14**

Box Sampling ____ 14
Riemann Sampling ____ 14
Gaussian Quadrature ____ 16
Adaptive Integration ____ 17
General Linear Models ____ 18

Choice of Weighting Functions ____ *20*

RGB functions ____ 20
XYZ functions ____ 21
LMS functions ____ 21
ACC ____ 21
CIELAB space ____ 22

Introduction to Computer Graphics ____ *23*

Modeling ____ 23
Geometric Transformations ____ 25

Translation ____ **26**

Rotation ____ **26**

Scaling ____ **27**

Lighting and "Camera" Placement ____ 28
Illumination and Color ____ 28

Local Illumination ____ **29**

Global Illumination ____ **30**

Shading 32
Rendering 34

OpenGL *34*

What is OpenGL? 34
Why OpenGL 35
Lighting and Color in OpenGL 35
Accumulation Buffer 36

Goals and Approach of Current Research *37*

The Scene Viewer *38*

File 39
Object 39
Ambient and Local 40
The Detector 41
Chromatic Adaptation 42
Ambient and Local Intensity 42
Spectral Properties Dialog 42

Viewing Spectral Images *43*

Color Calculations *45*

Spectral Radiance Calculations 45
Radiometric to Display 47

Human Observers **47**

Digital Camera and Photographic Film **49**

Chromatic Adaptation **51**

von Kries **51**

Fairchild Model **52**

Examples of Interactive Spectral Rendering *54*

Spectral Rendering of a Macbeth ColorChecker 54
Rendering and Modeling 55
Observer Metamerism 56

Illuminant Metamerism 58
Fluorescence 59
Metallic Surfaces 61
Texture Mapping 62
Chromatic Adaptation 63

Writing Full Spectral Images ***64***

The Save-As Option 64
Rendering Loop 65
Frame buffer Choices 65
Image File Format 67

More Advanced Scenes ***67***

Future Directions ***70***

Conclusion ***71***

References ***73***

A. Appendix A: Flow Chart of spectral Software A-1

B. Appendix B: Interactive Viewer Code. B-1

spectral.h B-1
spectral.c B-5
spect-read.c B-8
spect-init.c B-14
spect-gl.c B-27
spect-cb.c B-30
spect-gl.c B-44
spect-math.c B-56
shapes.c B-80
spect-write.c B-83
spect-gl.pasture.c B-98
spect-gl.table.c B-111
Makefile B-131

C. Appendix C: Example Material Properties File C-2

Introduction

The goal of this research was to develop a computer graphics algorithm that enables full-spectral color image rendering. This rendering was performed by extending the OpenGL graphics API, to accommodate a wide range of computing platforms, as well as hardware acceleration. By maintaining the wavelength information during the color calculations, it is possible to use this technique to create full spectral synthetic images of arbitrary spatial and spectral dimensions. This algorithm is also used to create an interactive, colorimetrically accurate scene for use in digital imaging simulations. This interactive scene is capable of demonstrating certain fundamental ideas of color science, such as fluorescence and metamerism.

Background

Color in the "real world" is a function of many factors including the spectral power distribution of the illumination, the spectral properties of the stimulus, and the spectral sensitivities of the observer. Accurate color synthesis for computer graphics needs to account for the full spectral properties of the objects being modeled. Researchers interested in photorealism have known this for years, as full-spectral information has been added to global illumination calculations for computer graphics.[1] Unfortunately, all current hardware acceleration exclusively use only the RGB model for color calculations.[1] This research hopes to bridge that gap, and form a model of full-spectral image synthesis utilizing current RGB hardware acceleration.

Theory

Computer Generated Color

The final perceived color of any image created using computer graphic routines depends upon the chosen output device. This might include Cathode Ray Tube (CRT) displays, continuous tone printers, LCD panels, film recorders or motion picture and photographic films. Since one of the primary goals of this research is to provide an interactive, colorimetrically correct scene, we shall focus first on the CRT. Color displayed on a CRT is the resultant of electron guns exciting three different types of phosphors. When these phosphors are excited, they emit light of a particular spectral power distribution. The typical monitor uses red, green, and blue phosphors, which together form an additive color reproduction system. The computer controls the color by modulating the excitation level of the phosphors. The number of colors available varies depending on the amount of memory dedicated to the display, the number of bits in the digital-to-analog converter, and other factors. A typical desktop computer stores 24 bits per pixel, or 8 bits for each of the red, green, and blue channels. High-end graphics workstations might store as many as 36 bits per pixel.

Another goal of this research is to generate an algorithm that is system independent. The end result should work on a variety of different computing platforms, independent of the number of colors available for display, or the chosen display device. The choice of color models used needs to be flexible enough to allow for this.

Today, most interactive graphics systems utilize the RGB color model.[1] This model is represented by a cube, as shown in Figure 1. The axes of the cube represent the red, green, and blue CRT phosphors, while the corners represent full "on" or "off" amounts of these phosphors.

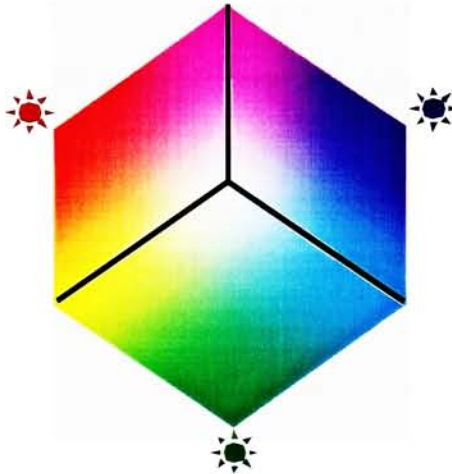


Figure 1. RGB Color-Cube rotated so that the viewer is looking down the achromatic axis.

To remain device independent, the coordinates are kept between 0 and 1. If the device uses 8 bits per pixel, the coordinates can be found by normalizing the digital counts ($1/(2^8-1)$, or 1/255). Likewise, if 12 bits are available, the coordinates are normalized by $1/(2^{12}-1)$, or 1/4095. The eight corners are represented by the following RGB triplets: Black [0,0,0], White [1,1,1], Red [1,0,0], Green [0,1,0], Blue[0,0,1], Cyan[0,1,1], Magenta[1,0,1], and Yellow[1,1,0]. It is important to remember that the perceived color at these points is device dependent, and varies as the phosphors vary.

This color model is an obvious and easy choice for computer graphics. Most commercial applications today exclusively use this model to describe the lights and surfaces in a scene.[1,2] The light sources and surfaces are described by their respective RGB triplets, and the interactions between the two are determined to be the product of the red, green, and blue values. These

products are then displayed directly. The result is color calculations that are strongly dependent on the display choices, and do not represent real world physical interactions. Hall[2] identified three strong causes for error in using RGB values for color calculations:

- *Information is lost when the spectral curves are sampled and represented by only 3 values.*
- *The RGB sampling functions are not selected on the basis of sampling theory, but determined by perceptual criteria. The perceptual criteria consider color reproduction, not color computation requirements.*
- *The RGB sampling functions are specific to the RGB primaries used. Computations in different RGB color spaces may not produce the same result(after color space compensation).*

It seems obvious that the RGB color model has many flaws. Another possibly better method of color calculation could be to use the CIEXYZ color system. The XYZ color model is more grounded in the fundamentals of perception. The XYZ model also has the benefit of being an international standard. By carefully calibrating and characterizing a CRT monitor, it is possible to convert between monitor RGB values, and CIEXYZ values. This conversion is generally a simple linear transform, with an exponential linearization term (gamma correction).[3] Color calculations can then be done in XYZ space and can be converted to RGB space for the display. This would eliminate the device dependency problem, since all the calculations will be done in a "device-independent" space. The end-user is ultimately held responsible for the final monitor characterization, to make sure that the desired color is properly displayed.

The XYZ space is probably a better space to perform color calculations, but it is still flawed. Information is still lost when the spectral curves are represented by only three values. Hall summarizes the basis of this problem as performing wavelength based computations after colors have been translated out of the wavelength domain, and into the perceptual domain.[2] Perhaps a good example of where these tristimulus based color calculations fail is the phenomena of metamerism. Metameric objects are objects that have different spectral properties, which appear to match in color. This color match often breaks down when any of the viewing conditions are changed, if the spectral properties differ greatly. Using a tristimulus based color representation, if two objects match under one condition, they must match under every condition. Thus, a tristimulus based color calculation method is unable to demonstrate metamerism. None of these tristimulus alternatives represent a good choice for this research, as one of the desired goals is to to write out synthetic images that contain the spectral information at every pixel.

Correctly nandling the color calculations during image synthesis ultimately requires preserving the wavelength dependence of both the light and the surfaces.[1] It is this preservation that makes up the goal of this research.

Spectral Information

Perceived color depends on many factors, including the spectral distribution of the light source, the spectral properties of the stimulus, and the spectral sensitivities of the observer viewing the stimulus. The goal of this research is to be able to model these factors accurately and quickly using computer graphics. The desired effect of color image synthesis is to take the surface

spectral reflectance functions and the light spectral power distribution and calculate the expected color signal. This signal could then be displayed such that it has the same effect on the human visual system as the actual color stimulus would.[4] This is generally the goal of most computer graphics algorithms, as the resultant images are usually viewed by a human observer. The technique presented in this research is not limited to this, however. The effect of the calculated color signal on any given imaging system can be simulated. This research also allows the signal itself to be recorded, as a spectroradiometric image.

Real world spectral reflectance functions are generally continuous functions. It is impossible to represent these continuous functions in a discrete computer graphics algorithm, so the need for wavelength sampling is an important issue. Wavelength selection is also an important decision, since often samples or light sources have energy in large wavelength regions. The human visual system is generally thought to be sensitive to energy between 380nm and 730nm. Many graphics systems that do full spectral rendering only consider these wavelengths. For accurate modeling of some stimuli, however, other wavelengths need to be considered. For example, to model a fluorescent stimulus (a stimulus that absorbs energy at a lower wavelength, and emits this energy at a higher wavelength). Since one of the goals of this research is to develop an algorithm that can handle the fluorescent properties of stimuli, it is necessary that a larger wavelength region is chosen.

Wavelength Sampling

Once the wavelength region has been selected, a method of sampling this region is necessary. There is much literature devoted to this subject.[5678] A brief summary of the most often applied methods follows.

Box Sampling

The Box method of sampling and reconstruction was first suggested by Hall.[2] This method uses boxes of varying height and width, but equal area to sample the spectral functions. They idea is to estimate the area of a given curve using simple boxes of known areas. The method utilized by Hall was a brute force approach of trial and error, in an attempt to accurately sample and reconstruct the spectral distributions. This technique is prone to large sampling errors when few boxes are chosen, but becomes increasingly accurate as the number of samples increases. This was by no means a mathematically elegant solution, but it was an attempt to begin using spectral data for color calculations.[2]

Riemann Sampling

This method utilizes evenly spaced out point samples, and then a numerical integration method. In utilizing this method, the spectral reflectance of the stimulus, the spectral power distribution of the light source, and the spectral sensitivities of the observer are all sampled at evenly spaced intervals. These (now discrete) functions are then multiplied, to perform the color calculations. Since the desired outcome of most computer graphics algorithms is a signal that can be displayed on an output device, these functions are usually weighted by an observer function such as the

1931 CIE Standard Observer. In this case the final color signals are then calculated using the Riemann summation, as follows. [5]

$$X = \int_\lambda \Phi(\lambda) R(\lambda) \bar{x}(\lambda) d\lambda \approx \Delta\lambda \sum_{i=0}^{N+1} \Phi(\lambda_i) R(\lambda_i) \bar{x}(\lambda_i)$$
$$Y = \int_\lambda \Phi(\lambda) R(\lambda) \bar{y}(\lambda) d\lambda \approx \Delta\lambda \sum_{i=0}^{N+1} \Phi(\lambda_i) R(\lambda_i) \bar{y}(\lambda_i)$$
$$Z = \int_\lambda \Phi(\lambda) R(\lambda) \bar{z}(\lambda) d\lambda \approx \Delta\lambda \sum_{i=0}^{N+1} \Phi(\lambda_i) R(\lambda_i) \bar{z}(\lambda_i) \quad (1)$$

Where λ represents a given wavelenght, $\Phi(\lambda)$ represents the spectral power distribution of the light source, $R(\lambda)$ is the spectral reflectance function of the surface, and $\bar{x}(\lambda)$, $\bar{y}(\lambda)$, and $\bar{z}(\lambda)$ are the spectral sensitivities of the observer. The accuracy of this method of wavelength sampling increases as the number of samples increase. When accuracy is essential, and time is plentiful, these calculations can be performed at 1nm intervals. This method is generally used as the benchmark for comparisons of other methods.[5] While the accuracy increases with the number of samples, so too does the computation time. With interactivity being a general concern, often there is the need to sacrifice some accuracy for quick rendering times. Peercy has shown that Riemann sampling with as little as four point samples is accurate, when the components in the scene can be well described with low-order Fourier components.[5] Unfortunately not all scenes can be well described in such a manner. This is especially true when the scene is illuminated by a light source with distinct emission lines, such as a fluorescent light source. When this is the case, either more samples need to be taken, or a different method of wavelength selection needs to be chosen. Since

the ultimate goal of this research is to utilize physically accurate color calculations, of arbitrary spectral dimensions, this techniques seems to be an appropriate choice.

Gaussian Quadrature

When both speed and accuracy are important, the method of gaussian quadrature sampling might be appropriate. Gaussian quadrature integration is numerically one of the most precise techniques, as well as a very efficient technique.[6,7,8] This is due to the small number of samples necessary to approximate an integral, with a sufficiently low error. Essentially the gaussian quadrature allows for the approximation of an integral by using a weighted sum over a set of sample points. Since the color calculation integrals are generally weighted by a function, such as a color matching function, this method of numerical approximation lends itself readily. The equations for this method are as follows:

$$\int_\lambda \Phi(\lambda) R(\lambda) W(\lambda) d\lambda \approx \sum_{i=0}^{n} H_i \Phi(\lambda) R(\lambda) \quad (2)$$

$$\begin{aligned} &where: \\ &H_i = \int_\lambda L_i(\lambda) W(\lambda) d\lambda \\ &L_i = \prod_{i=0, j \neq j}^{n} \frac{\lambda - \lambda_j}{\lambda_i - \lambda_j} \end{aligned}$$

The weighting function, W(λ) could be the color matching functions, the cone responsivities, or any other desired weights. Similarly to Equation 1, R(λ) represents the spectral reflectance of the material, and Φ(λ) is the spectral power distribution of the light source. The sample wavelengths chosen are roots of the second order of Chebychev polynomials to the nth degree, and can be found doing recursive calculations.[8] This method is the best technique, as a least-squares

minimizing errors technique, as long as the scene is continuous over the wavelengths sampled.[9] If this is not the case, as in a light source with discrete intervals such as a fluorescent light, two significant errors might occur:[8]

- If none of the wavelength samples are close to the spectrum peaks, then the emission peaks are not taken into account, and there is large error
- If one of the wavelength samples is exactly on an emission peak, then the integral estimation will over-weigh the sample, and significant error could occur.

This technique is very appropriate when the weighting functions are known, and are held fixed. This does not readily adapt to the interactive changing of weighting functions (changing from a human to a digital camera for instance). This technique also loses a large amount of spectral information, by representing material and light source properties with a small number of samples. Once the spectral information is sampled, it is impossible to recover. Thus, it is impossible to create a spectral image with more spectral dimensions than the number of samples.

Adaptive Integration

Deville *et al*[8] developed a method that allowed complex spectral characteristics to be modeled during image synthesis. This method involved a segmentation of the wavelength domain into smaller, continuous regions. These regions were then examined for emission peaks, using a simple variation-testing algorithm. This algorithm determines if the current sample is an emission peak by determining the variance between the current value, and the values of the neighbors. If the spectrum contains emission peaks, then more samples are automatically taken. For segmented areas where the spectrum is slowly varying, then less samples are taken, and gaussian quadrature

is used. This method utilizes the strength of both point sampling and gaussian quadrature. This method requires a large amount of pre-computing, however, and does not allow for ease of interactivity. All the elements of the scene must first be segmented, before any color calculations can be preformed.

General Linear Models

Much work has been done in the use of general linear models in the area of color calculations.[1,4,5,9,10,11] In these cases the light spectral power distribution and the surface spectral reflectance functions are expressed as expanded sums weighted over basis functions.

$$R(\lambda) \approx \sum_{i=1}^{n} \sigma_i R_i(\lambda)$$
$$\Phi(\lambda) \approx \sum_{i=1}^{n} \varepsilon_i \Phi_i(\lambda) \tag{3}$$

The basis functions R_i and Φ_i are chosen to minimize the error:

$$\sum_R \int \left[R(\lambda) - \sum_{i=0}^{n} \sigma_i R_i(\lambda) \right]^2 d\lambda \tag{4}$$

In the case of a single light source, and a single surface reflectance, these basis functions can simply be the actual spectral functions. In this case, the error is zero. If there are more than one light sources, or more than one surface reflections, then the basis functions are determined mathematically. This is typically done using a method such as principal component analysis. Principal component analysis is performed on spectral power distributions of all the light

sources, as well as all the surface reflections, and the observer sensitivities. The goal of this expansion is the hope that the illumination, surfaces, and observer sensitivities can be expressed by a small number of basis functions.[9] That is to say, the overall variance of the scene can be explained by a small number of orthogonal basis functions. These basis functions can then be expressed in matrix terms, and color calculation is simply matrix multiplication.

For a small number of light sources, the basis functions could be the spectral power distributions of the lights themselves. The total number of basis functions is then simply the total number of light sources. In this case, the basis functions are generally broken down into point samples of the original wavelength functions. However, as the number of light sources increase, a general linear model found using a technique such as principal component analysis will be more efficient.[9]

Though often computationally more efficient, this method also has several drawbacks. One of the major drawbacks, in respect to this research project, is the need to pre-compute the basis functions for the entire scene. One of the goals of the interactive scene is allow the user to load their own reflectance function for an object in the scene, or to alter the reflectance of a given object, and to see the resulting color. If the scene were represented by general basis functions, then these functions would have to be recomputed every time the scene changed. This would be the same with changing the lighting, or changing the observer sensitivities. While this would not be a problem if the user were forced to use a pre-selected scene, this would become a problem should the user desire more freedom. In addition, principal component analysis neglects the observer sensitivities during the selection of basis functions.[9] It is possible for the analysis to

give more weight to the wavelength regions where the observer is not sensitive, and less weight the regions of sensitivity. This becomes especially apparent when the observers are arbitrarily changed. Perhaps the biggest problem with this technique, for the current research, is the conversion of material and light source properties from physical reality, into statistical representations. A full spectral image created with this technique would have basis functions with no physical meaning, rather than having spectroradiometric signals at each pixel.

Choice of Weighting Functions

In order to be most efficient, many of the above methods of wavelength selection and sampling include weighting functions. These are usually in the form of observer color matching functions. The choice of these functions can greatly influence the efficiency and accuracy of the final model. Some chosen weighting functions include: device dependent RGB, CIE observer color matching functions $\bar{x}, \bar{y}, \bar{z}$, the cone responses of the human visual system $\bar{l}, \bar{m}, \bar{s}$, opponent spectral sensitivities AC_1C_2, and the CIELAB color space. Since the goal of this research is to maintain the full-spectral information throughout the color calculations, these weighting functions can be arbitrarily chosen, and can be used to convert the spectral signal into displayable information. In standard computer graphics algorithms, the choice of weighting functions becomes much more important, so a quick review is in order.

RGB functions

We have seen already how using the RGB color model to do color calculations could be the cause of large errors. This is generally the case when they are used as weighting functions as well. However, if the weighting functions are a linear transform between RGB primaries and human

response curves, such as XYZ, then it would be possible to do the color calculations directly into device RGB values. This would increase the efficiency of the spectral algorithm. In order to do this, the display device must be properly linearized, and characterized. If that is the case, then the monitor RGB space might be an acceptable choice as weighting functions.

XYZ functions

Using the CIE 1931 standard observer as the weighting functions would have the added benefit of having the results of the color calculations directly expressed as XYZ tristimulus values. These values could be quickly converted to monitor RGB values, through careful characterization of the display device. The end result would be similar to using linear transformed RGB weighting functions.

LMS functions

Using the sensitivities of the human visual system would allow the wavelength selection to be weighted by the sensitivity of the human eye. This is essentially similar to using the XYZ tristimulus space, but with a firmer basis in human physiology. Essentially the results would be similar to those found using the XYZ space, since the LMS space is a simple linear transform of XYZ. The disadvantage of using these color matching functions is the lack of a standard set of functions.

ACC

Meyer[7] suggested that to minimize the errors while synthesizing colors, the axes of the color space used to perform the color calculations should be oriented so that they pass through the

regions where tristimulus values are most likely to occur. Meyer points out that the LMS cone responsivities are not well spread out, and that there is a large degree of overlapping. This is also the case with the 1931 Standard Observer responsivities. A transform of the LMS coordinate system was sought that directed the axes through the densest regions of tristimulus values. Also desired was a priority on each axis, depending on the proportion of coordinates that lie along its direction.[7] The chosen space was the opponent sensitivity space AC_1C_2. In this space, the A axis represents the achromatic axis, while the C_1C_2 axes are the chromatic axes, roughly correlating to red-green, and blue-yellow regions of color space. This space yielded better results, with fewer samples for both gaussian quadrature, and adaptive integration.[9,8] Since this space has negative weights, representing the opponent channels, there is some difficulty with the gaussian quadrature method of wavelength sampling. There is not always a solution, for any given number of samples. For example, there might be a solution for 3, 6, 8 samples, but not for 4, 5, and 7 samples. Thus, this color space selection is slightly less robust. There is also more calculations needed to transform the results of the color calculation back into monitor RGB coordinates.

CIELAB space

Though not really a set of weighting functions on its own, another possible choice could be to use a color space such as CIELAB. This could allow for the minimization of perceptual error, rather than the minimization of tristimulus error. This method would essentially be using the XYZ tristimulus space as weights, though going a step further with the calculation of color differences as well. The weighting functions described above minimize the error in tristimulus space, which

has no direct correlation to perceived error. To determine perceptual color differences a color space based on uniform perceptual spacing is needed.

Introduction to Computer Graphics

Before moving on, a quick review of some of the basic ideas of computer graphics is in order. Some of the ideas that will be touched upon include basic modeling, rendering, lighting, shading, rasterization, rendering, and an introduction to some advanced topics such as ray tracing and radiosity.

Modeling

The first step in three dimensional computer graphics involves the selection of an appropriate World Model. This is usually chosen to be the Cartesian coordinate system, as shown in Figure 2. Geometric primitives are then placed into this world system, by their respective xyz position.

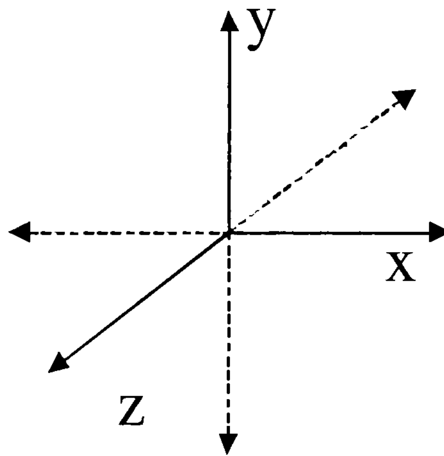


Figure 2 Cartesian world coordinate system

A geometric primitive can be a single point, a line connecting two points, or a polygon consisting of multiple points and lines. While many graphics systems support different kinds of polygons,

by far the most common is the triangle. The triangle is easy to implement in software, as 3 sets of

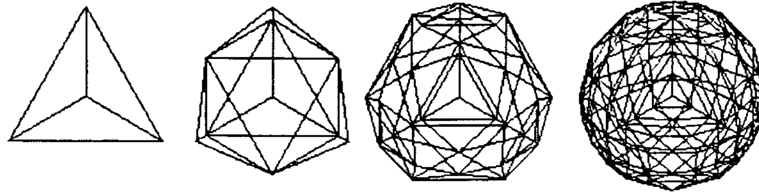


Figure 3. Example of simple triangle subdivision, to create a more complex geometry.

geometric coordinates, and easy to accelerate in hardware. For those reasons, we shall focus primarily on the triangle as the standard graphics primitive. To specify the location of a graphics primitive simply specify the Cartesian coordinates of each vertex. By building upon the simple graphics primitive, it is possible to create geometries that are much more complicated. It is this creation of more complex "models" that form the basis of computer graphics modeling. Several techniques can be used for graphics modeling. These include manually specifying the geometric coordinates, which can be done in a simple text file. A mathematical method of specifying a model can also be used, as in the case of triangle subdivision for sphere generation. In this technique, a simple geometric model is manually specified, and then repeatedly subdivided until a new geometric object is created. Figure 3 shows a tetrahedron subdivided repeatedly until it forms a sphere. The left most object is a simple tetrahedron, the next shows a tetrahedron subdivided one time, the next subdivided two times, and the last subdivided three times. With this technique it is possible to create complicated geometries, without having to manually specify the coordinates of each vertex. Triangles can also be used to create arbitrary, and quite complex, shapes. This is usually accomplished with a piece of software known specifically as a modeler, or a CAD tool. Figure 4 shows an example of a complicated geometric object created out of

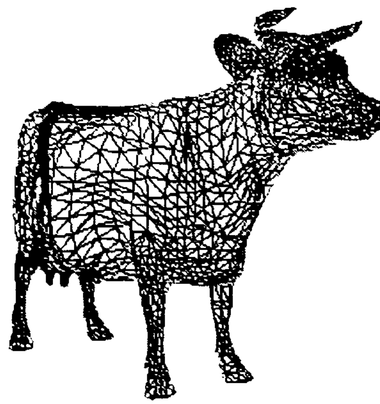


Figure 4. A complex geometry created using simple triangle primitives.

numerous triangles. Essentially, Geometric Modeling can be thought up as determining the spatial layout and shape of primitives (components), and the connectivity of these primitives to each other.[12]

Geometric Transformations

Often times it is necessary to move existing geometric models around in the world space. This is

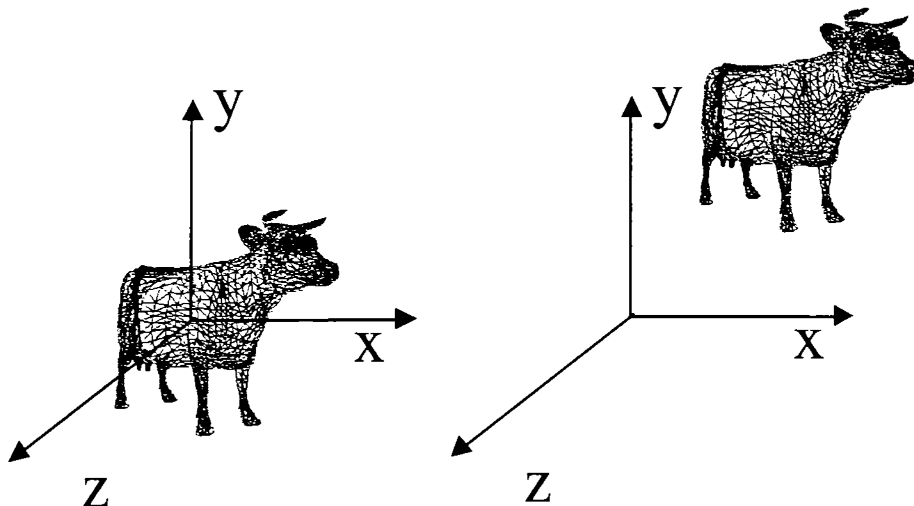


Figure 5. Example of a simple translation in two directions.

can be accomplished in a myriad of ways, but is most often accomplished using one of three techniques. These techniques make up the basis of most geometric transformations, and include translation, rotation, and scaling.

Translation

Translation is specifically an operation that moves a given point a fixed distance, in a given direction. Translation has three degrees of freedom, as an object can be moved in any of the three coordinate directions.[13] Figure 5 shows an example of a model translated in two directions.

Rotation

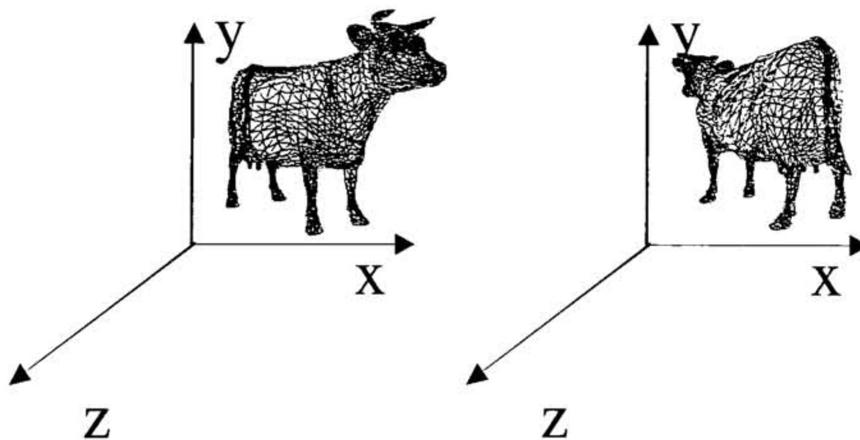


Figure 6. Example of fixed point rotation, 180 degrees about the y-axis.

Rotation involve one of two things: rotating about the origin of the world space, or rotating about the center of mass of the given object. The latter is often referred to as a fixed-point rotation.[13] Figure 6 demonstrates an example of a fixed point rotation as an object is rotated 180 degrees around the y-axis, . Figure 7 is an example of an object rotated 180 around the z-axis, with the rotation centered at the origin.

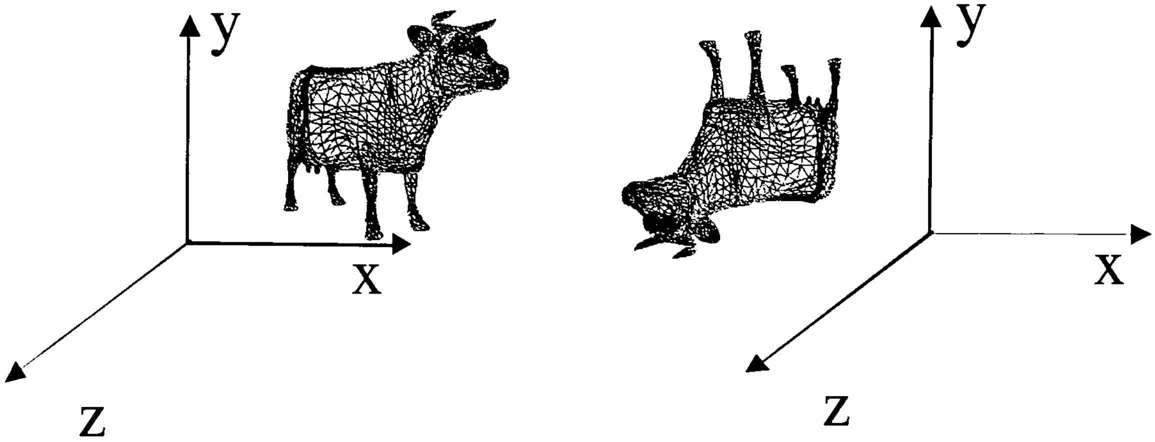


Figure 7. Example of rotation 180 degrees around the z-axis, relative to the origin.

Scaling

Scaling is known as a non-rigid-body transformation, as the dimensions and vertex relationships

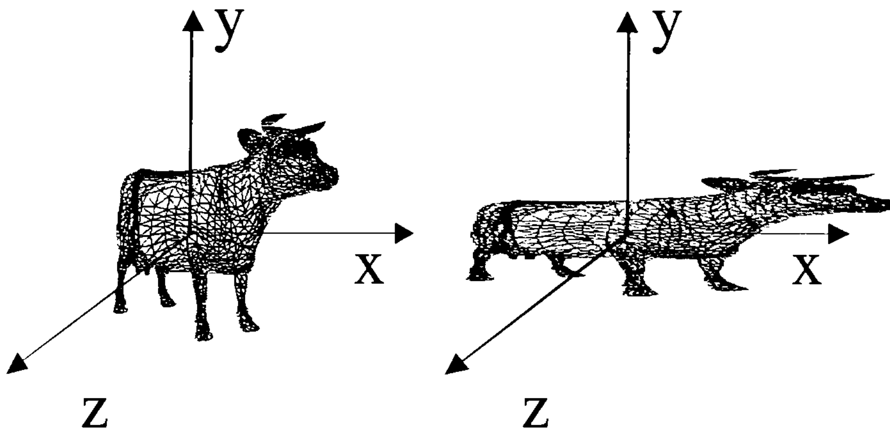


Figure 8. Example of scaling along the x and y directions.

of a model are altered. Scaling also has three degrees of freedom, as a model can be enlarged or shrunken in any of the three coordinate directions. Figure 8 is an example of an object enlarged in along the x-axis, and shrunken along the y-axis.

Lighting and "Camera" Placement

While we have dealt with the placement of objects in a scene, and the possible transformation of these objects, we have not dealt upon how we visualize the entire scene. Nor have we dealt with the scene illumination. To do this, we can use a camera light source analogy. It is easy to visualize taking a picture of a real scene with a camera. Extending upon this, if we consider the modeled objects to be a synthetic representation of a real scene, we can think placing a light source and a camera somewhere in the scene. What is ultimately presented to the end-user is a result of the position of the camera relative to the world coordinates, the position of the modeled objects, and the position of the light sources. Any and all of these positions can be altered, much like moving the lighting, camera, or objects in a real scene. Only the light that is reflected off the objects and into the camera is captured, so that is what ultimately determines what the final "picture" will look like. A more in-depth look at this interaction between light and matter is necessary to gain a better understanding of the final computer graphics image.

Illumination and Color

Color in computer graphics has already been touched upon above. Generally, there are two distinct ways for a model to be assigned colors. The first is to manually set the object, or model, to have a specific RGB triplet. This color remains fixed forever, once it is manually selected. This technique is often useful when creating business graphics, or eye-pleasing artwork. Far more common, objects are assigned material properties. These can be as simple as RGB reflectance properties, an object with a RGB material property of [0.9, 0.75, 0.6] would "reflect 90% of red light, 75% of green light, and 60% of blue light. When this is the case, a light source must be

present in the scene, which is also defined to have RGB material (radiance) values. It has been shown above how setting light sources and material properties with RGB values are generally not adequate for accurate color synthesis. To understand a little better as to what other methods there are, we first must examine the two different types of illumination models.

Local Illumination

Local illumination is the most common form of computer graphics lighting models. Effectively, local illumination places a light source (or multiple light sources) into the world system. Object colors are determined by the properties of the light source, the geometry of the model, and the material properties of the object itself. The color of one object in a scene is independent from the color of any other object in the scene. Thus, it is impossible to get any scene interactions such as shadows, mirrors, and caustics. While this does seem to accurately portray what is going on in the real world, for simple scenes this is often adequate. OpenGL, and thus the current research, use only local illumination to determine object color. Generally, the interactions between light and matter are separated into three or four distinct properties. A light source is usually defined to have a diffuse property, a specular property, and an ambient property. Materials can have different diffuse, specular, and ambient reflectances, as well as an emission property. Since local illumination models do not allow the interaction between two objects, an emissive material might appear to glow, but it would not illuminate any other object. The distinct lighting and material properties are then combined to form a given color, at a given vertex. Most often, this is all performed using the RGB color model. In the case of this current research, light sources are assigned a single full spectral power distribution, rather than three distinct RGB values. Any

given object can be assigned distinct full spectral diffuse and specular reflectances, as well as excitation and emission spectra. The ambient term in both the lighting and material properties can be considered the "global" contribution from the rest of the scene. There is no real scientific basis for this, as there is no way to know what the actual object interactions would result in. For that, a more advanced technique is needed.

Global Illumination

Global illumination accounts for the interactions between light sources and objects, as well as the interactions between the objects themselves. We will focus on the two main methods for calculating global illumination, ray tracing and radiosity.

Ray tracing is a method for the effect a light source has on a given object, as well as the effects one given object might have on another. Returning to the light source and camera analogy, only the light that reaches the camera is considered for determining the ultimate color of any given object. Thus, if we trace a path from the light source, to the camera, taking into account all the surfaces that it encounters along the way, we would be able to create a "globally illuminated" scene. Since a light source might have a huge number of path routes that might not make it to the camera, this technique is often performed in reverse. Starting from the camera position, a "ray" is cast into the scene at each given pixel. This ray is continued on into the scene until it intersects with a modeled object (surface). Once an intersection is detected, another ray is shot from that point to the light source, or sources as it might be. If this ray, often called a shadow ray, is not blocked by any other surface, then the interaction between the light source and the material is calculated. If the shadow ray is blocked, then it is known that the surface is in a shadow cast by

another object. This technique alone would create a scene with shadows as the only form of global illumination. While this is better than the local illumination method, ray tracing can also calculate the interactions between objects in a scene.

If the surface that the first ray intersects with is reflective, transparent, or translucent new rays can be generated. The paths of these new rays are determined by the material properties of the surface, and are followed until they intersect with another surface. From there, another shadow ray is cast, and the resulting color is factored into the first surface. This can be repeated endlessly, but is usually stopped after a few rays are cast. With this technique, it is possible to simulate many interesting interactions, such as mirror reflections and transparent refraction.

Ray tracing is generally very computationally expensive and thus time consuming. There are many methods of acceleration, and much research is being done to speed up this process. Another downfall of ray tracing is the inability to determine diffuse inter-reflections. The ray casting technique described above assumes that the reflection is only in the specular direction, and thus all interactions are specular in nature. To determine diffuse interactions, a technique such as radiosity needs to be utilized.

Radiosity is a method for calculating diffuse reflections from a surface by considering the radiant energy transfers between surfaces, subject to conservation of energy laws.[14] Radiosity algorithms divide scenes into many patches, and then determine the effect each patch has on the other. A general assumption is that each patch is perfectly diffuse, and that each patch has a constant shade. The reflections between patches and the light source, and the inter-reflection between patches are calculated using an iterative procedure. More accurate radiosity solutions

require highly complex bidirectional reflectance distribution functions, or BRDFs. The BRDF is a function of wavelength, surface properties, incoming light direction and outgoing light direction. The more details this model contains, the more accurate the radiosity solution.

The two methods discussed above are by no means a complete discussion on all the various global illumination techniques. There are many other hybrid techniques that blend the benefits of both ray tracing and radiosity together. These techniques are far beyond the scope of this research, as the current research deals strictly with local illumination. It should be noted that while global illumination methods can utilize the RGB model for light source and material descriptions, the desire for physically accurate light transport requires the use of full spectral information.

Shading

Returning to the local illumination paradigm, recall that the lighting and color calculations determine the color of a given vertex, not of a given polygon. Although the triangle modeling has produced wire frame objects such as those shown in Figures 3 and 4, the real world is not made up of wire frames. A method of shading the remainder of the triangle is needed. This is usually accomplished in one of three methods. These methods are known as flat, Gouraud, and Phong shading.[12]

Flat shading simply calculates the normal of a single vertex, to calculate the color of the entire polygon. This method often produces large jumps in colors between polygons, as polygons that share vertices might not have the same color. The result is often a model with large noticeable color differences between polygons. Interpolative shading attempts to alleviate this by

calculating the color at each vertex based upon its own vertex normal. The color of the polygon is then interpolated between the colors of all the vertices.

While Gouraud shading is by nature interpolative, this relationship is not true in reverse. Interpolative shading is not necessarily Gouraud shading.[13] Gouraud determined a method of defining the vertex normal in a way that it produced more accurate shading, utilizing the interpolation method. Using this method, any given vertex normal is determined to be the normalized average of all the polygon normals that share the vertex. Thus, if careful time is put in when modeling, it is possible to use the Gouraud shading method of interpolation. Figure 9.

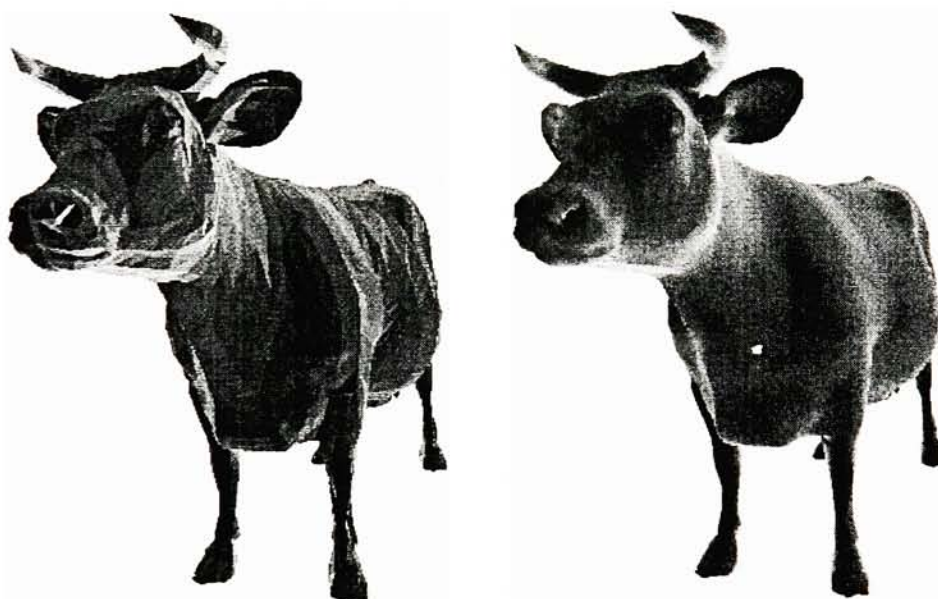


Figure 9. Examples of flat and Gouraud shading. The cow of the left was shaded with a flat shading algorithm, while the cow on the right was shaded using a Gouraud shading algorithm

Shows an example of Flat shading, and Gouraud shading.

Phong shading takes this idea one step further. Instead of interpolating vertex colors across the polygon, the Phong shading method interpolates normals across each polygon. Thus

once each pixel in the polygon has a normal, a color calculation can be performed for each pixel. This generally results in a much smoother shaded image, especially in the specular highlights. It should be noted that Gouraud shading can be implemented fairly easily in hardware, while Phong shading is almost always performed off-line. Thus, most interactive computer graphics applications rely on Gouraud, rather than Phong shading.[13] OpenGL utilizes hardware accelerated Gouraud shading, so that is a logical choice for this research.

Rendering

Once a scene is modeled, material properties are defined, and colors calculated, the computer graphics scene is ready to be rendered. Rendering is considered the conversion of the primitives in the object coordinates, to an image in the frame buffer.[15] This can be considered the conversion of a three-dimensional object scene into a two dimensional image. This conversion is accomplished through a process called rasterization. The research performed here is primarily concerned with the color calculations, rather than the modeling, or rendering. Thus, the light source properties and material properties are the main considerations. The software created is model independent, as many examples of different models are illustrated. The rendering aspects of the software are handled completely by OpenGL.

OpenGL

What is OpenGL?

OpenGL is an application programmers interface, or API. Essentially, it is a software interface to graphics hardware.[15] There are four major operations that OpenGL performs: [15]

- *Constructs shapes from geometric primitives (points, lines, polygons, images, and bitmaps), thereby creating mathematical descriptions of objects.*
- *Arrange the objects in three-dimensional space and select the desired vantage point for viewing the composed scene.*
- *Calculate the color of all the objects. The color might be explicitly assigned by the application, determined from specified lighting conditions, obtained by pasting a texture onto the objects, or some combination of these three actions.*
- *Convert the mathematical description of objects and their associated color information to pixels on the screen (rasterization).*

Why OpenGL

The main reasons for using the OpenGL API for the current research are speed, flexibility, availability of hardware acceleration, and standardization. The OpenGL API sits on top of the graphics hardware, and serves as the interface between the programmer and the hardware, allowing for great speed. The flexibility of OpenGL stems from being supported on many different platforms, and operating systems. Currently OpenGL is supported on most UNIX machines, as well as Windows 95/NT, and Macintosh. Many high-end graphics workstations have hardware acceleration designed specifically for OpenGL, with the goal of interactivity in mind.[16] Many PC makers are now starting to create hardware acceleration chips for both Windows and Macintosh users.

Lighting and Color in OpenGL

Color calculations in OpenGL are performed in the RGB color space. The final vertex color of a polygon depends upon the material properties of the polygon, and the properties of the light source. The programmer can specify the four aspects of material properties. These are the

diffuse, specular, ambient, and emissive properties. These are essentially the reflectance of the material, expressed in RGB terms. Another term is allowed to be specified, the shininess coefficient. This essentially defines the specular exponent of the material, or how large the specular reflection spot is.

The light source properties can be expressed by three terms, the diffuse, specular, and ambient terms. These terms are attenuated, depending on the selection of effects. The different effects include the spotlight effect, local point source effects, and infinite directional light source. Additionally there can be a global ambient light, used to simulate scattered light in a given scene. All lighting properties are expressed by their RGB triplets (or RGBA). Thus, the color produced by lighting, at a given vertex is as follows:[15]

Vertex color = the material emission at that vertex +

the global ambient light scaled by the materials ambient property at that vertex +

the ambient, diffuse, and specular contributions from all the light sources, scaled by the material properties, and properly attenuated.

Accumulation Buffer

Another important feature of OpenGL, is the Accumulation Buffer. This buffer is designed to integrate images that are rendered into the frame buffer.[17] The Accumulation Buffer provides 16 bits to store each red, green, blue, and alpha component, per pixel. The operations that may be applied to the buffer are as follows:

- Clear: The Accumulation Buffer is set to 0 for all pixels
- Add with weight: Each pixel in the drawing buffer is added to the Accumulation Buffer after being multiplied by a floating-point weight that may be positive or negative.
- Multiply by weight: Each pixel in the drawing buffer is scaled and then multiplied to the Accumulation Buffer.
- Return with scale: The contents of the Accumulation Buffer are returned to the drawing buffer after being scaled by a positive, floating-point constant.

Each of these above operations can be performed for all pixels in a viewport on the screen at very high rates, with today's hardware accelleration.[17] The Accumulation Buffer can allow for a multi-pass rendering technique needed to perform accurate color calculations, for multiple light sources.

Goals and Approach of Current Research

As already stated, the goal of the current research was to develop a computer graphics algorithm, based on an extension of the OpenGL API, that allows for physically accurate, interactive rendering based on full-spectral color calculations. The resultant renderings can be used to write out full-spectral images, which can be useful in the development, testing, and simulation of color imaging systems. This algorithm can also be used to generate an interactive scene for use in color imaging simulations, as well as for teaching and demonstration purposes. It is the interactive scene that will be discussed first, as it lays the foundation for the synthetic image creation. By maintaining full spectral information throughout all the color calculations, it is possible to

demonstrate such spectral properties as: illuminant and observer metamerism, fluorescence, spectral properties of gloss, and full-spectral texture mapping.

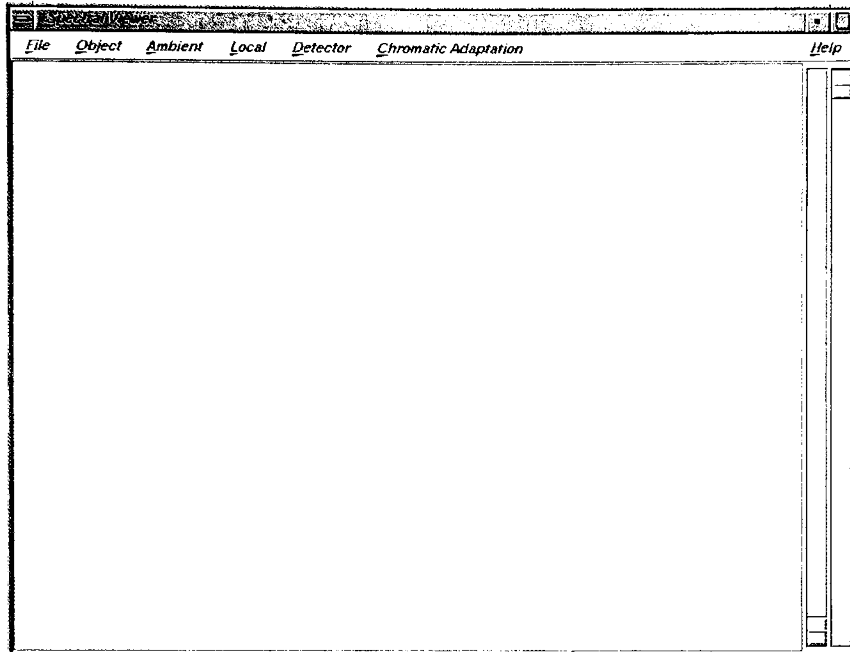


Figure 10. A screen capture of the spectral Scene Viewer.

The Scene Viewer

The Scene Viewer is the graphic user interface (GUI) that allows the end-user to interactively change the elements of the scene, and view the resultant color effect. The viewer was written in C, using the Motif toolkit to build the user interface. Although the use of Motif limits the viewer to the X-window environment, the GUI commands are easily separated from the OpenGL and color calculation algorithms. This should allow for easy porting into other environments. It is also possible to run the viewer from a remote Unix workstation, and display it on a PC running an X-server. Figure 10 shows a screen capture of the Scene Viewer, as seen when the user first launches the program. The drawing area takes up most of the viewer. This is the area in which the

rendered three dimensional scene is placed. The user is also presented with a series of menu choices. These menus are as follows: File, Object, Ambient, Local, Detector, Chromatic Adaptation, and Help.

File

The file menu is placed in the top left corner of the Menu Bar. It contains the options: Open, Save As, and Exit. The Open and Save As options will be discussed later, while the Exit option is self-evident.

Object

The Object menu allows the user to interactively choose the three-dimensional shape of the objects being displayed. The default shape is a torus, or doughnut. The Object menu contains the

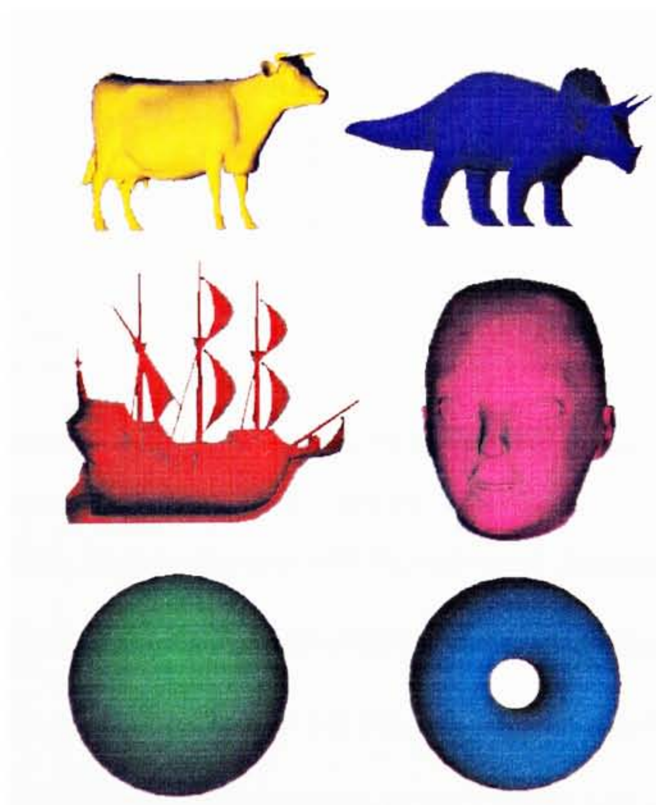


Figure 11. Screen capture of the different object options, clockwise from top left: cow, triceratops, head, torus, sphere, ship.

following options: Torus, Sphere, Cow, Triceratops, Head, and Ship. Figure 11 shows an example of each of these objects.

Ambient and Local

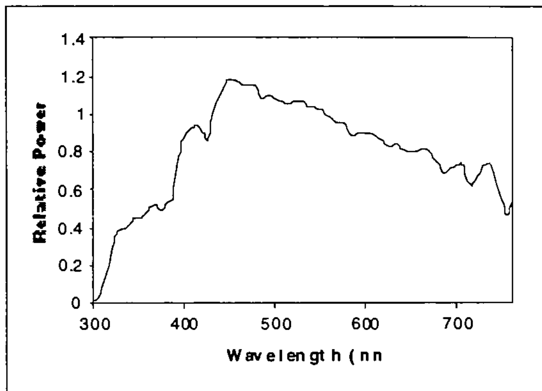


Figure 12. Relative Spectral Power Distribution of CIE D65

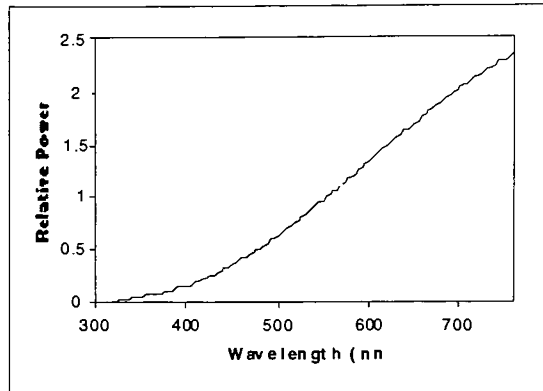


Figure 13. Relative Spectral Power Distribution of CIE Illuminant A.

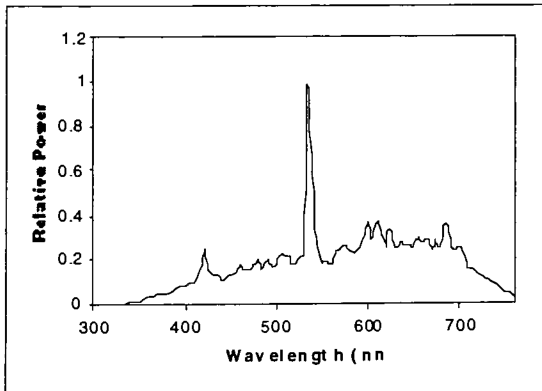


Figure 14. Relative Spectral Power Distribution of Fluorescent light source.

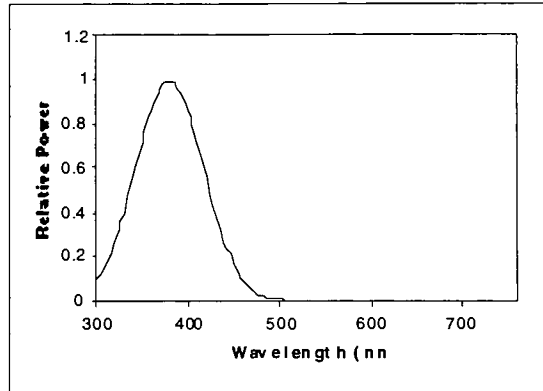


Figure 15. Spectral Power Distribution of the UV light source.

The Ambient and Local Menu presents the user with the options of changing the lighting. The user can change the ambient light source (the light coming from no particular direction), or the local light source (light with direction). The user can choose between the following light sources: CIE D65, CIE Illuminant A, a fluorescent light source, or a UV (black light) source.

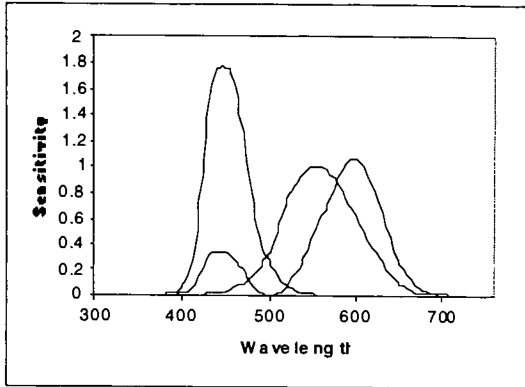


Figure 16. Spectral Sensitivities of CIE 1931 Standard Observer

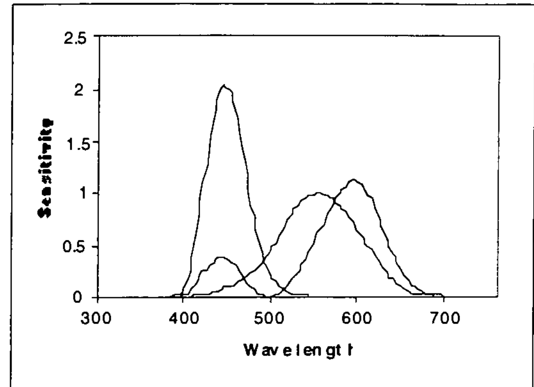


Figure 17. Spectral Sensitivities of CIE 1964 Standard Observer

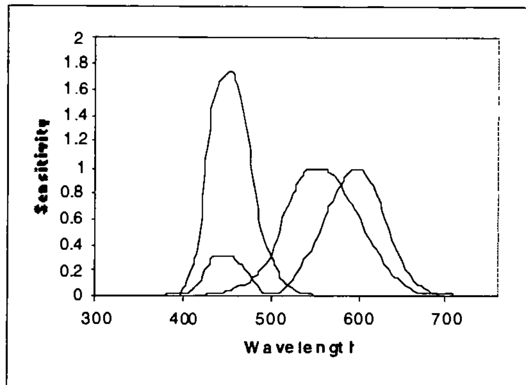


Figure 18. Spectral Sensitivities of CIE Standard Deviate Observer

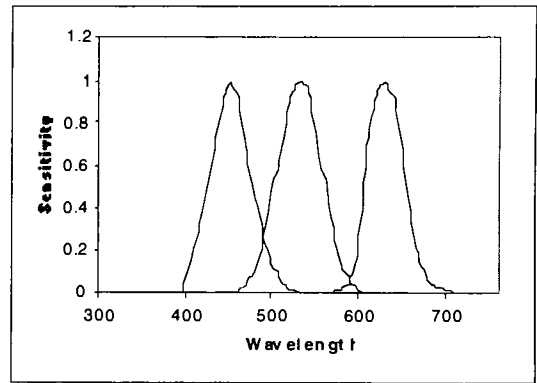


Figure 19. Spectral Sensitivities of a RGB digital camera

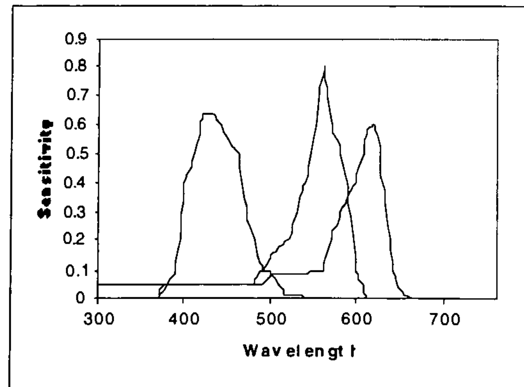


Figure 20. Spectral Sensitivities of a photographic transparency film

The Detector menu gives the end-user the option of choosing what imaging system is viewing the scene. This is a necessary step, since it is impossible to display a full-spectral image. The user has the option of choosing the following: CIE 1931 Standard Observer, CIE 1964 Standard Observer, CIE Standard Deviate Observer, an RGB Digital Camera, or Photographic Film. Figures 16-20 show the spectral sensitivities of these detectors.

Chromatic Adaptation

The chromatic adaptation menu allows the user to chose between three chromatic adaptation options: None, Von Kries, and Fairchild. For this to have an effect on the resultant image, one of the three human observers must be chosen.

Ambient and Local Intensity

The three slider bars on the right hand side of the drawing area allow the user to control the intensity of both the ambient and local illumination. This allows the user to vary the amount of energy from either light source, as they see fit. There are two default local light sources for the viewer. These are placed on the left and right sides of the viewer. These locations are easily altered, should other light positions be desired.

Spectral Properties Dialog

If the user selects any rendered object from the drawing area with the middle mouse button, a dialog pops open. This dialog displays the spectral properties of that object, as well as the interaction of the objects properties with the light source. This dialog also is capable of showing

the excitation and emission curves as well. The spectral plotting was written using the SciPlot widget library[18]. A screen capture of this is shown in Figure 21.

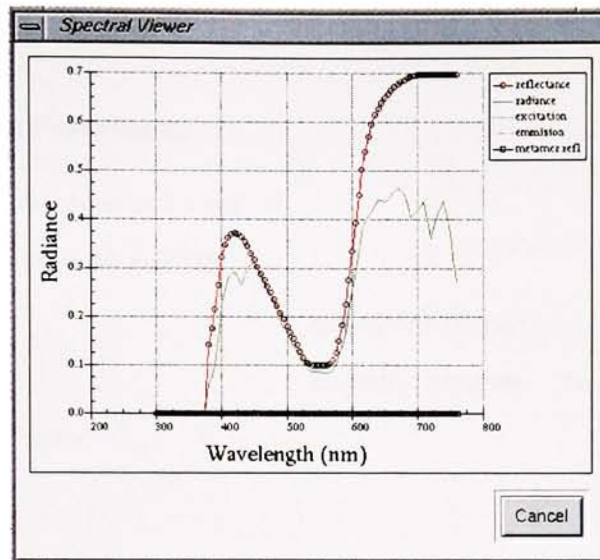


Figure 21. Screen capture of the spectral properties dialog.

Viewing Spectral Images

The scene viewer as describe above provides the end-user with an interactive environment in which they can manipulate many various aspects of the viewing environment. As stated earlier, the goal of this research is to perform all color calculations using the full-spectral information. Thus, all the above light sources are represented by their relative Spectral Power Distributions, from 300-760 nm, in 5 nm increments. The user must then input the material properties of the desired objects, before the rendering can begin. Material properties also must be defined using the spectral information between 300-760 nm, in 5 nm increments. A material may be defined to have any of the following information: diffuse reflectance, specular reflectance, shininess (gloss factor),

excitation spectrum and emission spectrum. All materials must have the diffuse reflectance defined, while all the other properties are optional. Each objects material properties are listed in an ASCII text file, in the following manner:

Table 1. Contents of a Material Properties file.

wavelength	*Diffuse reflectance between 0.0 and 1.0*		
g	*gloss factor between 0.0 and 128.0*		
s	*wavelength*	*specular reflectance*	
f	*wavelength*	*Excitation spectrum*	*Emission spectrum*
t	*texture map file name*		

Additionally, the user can define a metameric match for any given object. This is done by placing an *M* in front of the standard material property. Appendix A shows the material properties of many different objects.

Once the material properties are properly defined, the user must create a file listing the path names of the material files. This is a simple list of path names. The number of file names listed determines the number of objects in the final scene. Currently the Spectral Scene Viewer can display from 1 to 30 different objects, each with a different material file.

To view the scene, the user must first open the list of material file names. This can be accomplished using the Open dialog, selected from the File menu. Once the file are read into the Scene Viewer, the appropriate color calculations are performed, based upon the spectral

properties of the light sources, and the material properties. The final displayed image is then calculated, and rendered based upon the user's choices of detectors and chromatic adaptation. The user is then free to interactively alter any portion of the viewing environment.

Color Calculations

This section describes the computational procedure used in the bulk of this research. All color calculations are performed in the *spect-math.c* file, which can be found in Appendix B.

Spectral Radiance Calculations

The first, and most important, calculation is to determine the interaction between the light sources and the materials. In the simplest case there is a single local light source, and a single diffuse reflectance function. In this case, the interaction between these two is a simple multiplication of discrete spectral functions.

$$Radiance(\lambda) = \Phi(\lambda) * R(\lambda) \approx \Phi(\lambda_i) * R(\lambda_i) \tag{5}$$

Where $\Phi(\lambda)$ is the spectral power distribution of the light source, and $R(\lambda)$ is the spectral reflectance function of the object. This same equation can be used to calculate the ambient radiance, and the specular radiance as well.

To calculate the fluorescent contribution to the final radiance, it is necessary to first calculate the degree of excitation. This is accomplished by calculating the integral of the light source spectral power distribution and the excitation spectrum of the object.

$$exc_{\Phi} = \int_{300}^{760} \Phi(\lambda) Ex(\lambda) d? \approx 5 \sum_{i=300}^{N+1} \Phi(\lambda_i) Ex(\lambda_i) \tag{6}$$

Where Φ(λ) is the spectral power distribution of the light source, and Ex (λ) is the excitation spectrum. It is then necessary to calculate the total possible excitation. Essentially this is the excitation if the light source used had a relative power of 1 throughout the entire wavelength region of the excitation curve. This light source can be represented as the equal energy spectrum, with a value of 1.0.

$$exc_{equal} = \int_{300}^{760} equal(\lambda) Ex(\lambda) d\lambda \approx 5 \sum_{i=300}^{N+1} 1.0 * Ex(\lambda_i) \tag{7}$$

The ratio of these excitation integrals can be thought of as the amount of excitation caused by the given light source, or the excitation scale.

$$exc_{scale} = \frac{exc_{\Phi}}{exc_{equal}} \tag{8}$$

Finally, the fluorescent contribution to the radiance is the spectral emission weighted by the excitation scale.

$$Radiance_{fluor}(\lambda) = exc_{scale} * Em(\lambda) \tag{9}$$

The total radiance for a given object can then be calculated as follows.

$$\begin{aligned} Radiance_{total}(\lambda) &= Radiance_{local}(\lambda) + Radiance_{ambient}(\lambda) \\ &+ Radiance_{specular}(\lambda) + Radiance_{fluorescence}(\lambda) \end{aligned} \tag{10}$$

Once the total radiances for the scene are calculated, it is necessary to convert these values into values that can be displayed. In the interactive viewer, this is accomplished using the chosen detector sensitivities.

Radiometric to Display

Since no display device is capable of reproducing every spectroradiometric curve available, it is necessary to sample the spectral curve into displayable color coordinates. This can be accomplished in many different ways. The standard computer graphics algorithms attempt to sample and display the spectral signal such that it has the same effect of the human visual system as the actual color stimulus would. It is also possible to sample and display the signal so it is the same effect on *any* imaging system as the actual color stimulus would. The choice of sampling functions is determined by the choice of detector spectral sensitivities, in the interactive Scene Viewer.

Human Observers

The interactive Scene Viewer has three possible human observers, as a choice of imaging systems. These are the CIE 1931 Standard Observer, the CIE 1964 Standard Observer, and the CIE Standard Deviate Observer (Figures 16-18). If one of these detectors is chosen, the process to display involves several steps. First the spectral integral of the spectral sensitivities and the spectral radiances are calculated to form tristimulus values.

$$X = k\int_{300}^{760} Radiance(\lambda)\bar{x}(\lambda)d\lambda \approx \Delta\lambda \cdot k\sum_{i=0}^{N+1} Radiance(\lambda_i)\bar{x}(\lambda_i) \tag{11}$$

$$Y = k\int_{300}^{760} Radiance(\lambda)\bar{y}(\lambda)d\lambda \approx \Delta\lambda \cdot k\sum_{i=0}^{N+1} Radiance(\lambda_i)\bar{y}(\lambda_i)$$

$$Z = k\int_{300}^{760} Radiance(\lambda)\bar{z}(\lambda)d\lambda \approx \Delta\lambda \cdot k\sum_{i=300}^{N+1} Radiance(\lambda_i)\bar{z}(\lambda_i)$$

$$k = \frac{100}{\Delta\lambda\sum \bar{y}(\lambda)} \tag{12}$$

These tristimulus values can then be converted into monitor RGB values as follows, for a given monitor characterization.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 0.034 & -0.016 & -0.005 \\ -0.010 & 0.019 & 0.0002 \\ 0.0005 & -0.002 & 0.009 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{13}$$

$$dR = \frac{\left(R^{\frac{1}{1.534}} + 0.036 \right)}{1.035} \tag{14}$$

The methods for calculating dG and dB a similar. The numeric values found in Equations 13 and 14 are determined through the CRT characterization methods.[3] This process is done for the ambient, specular, and the combination of local and fluorescent radiances. The result is three groups of RGB triplets, representing the tristimulus values calculated. It is now a simple process to get OpenGL to render the scene, with the correct RGB values. The ambient and local light sources are set to the intensity level determined by the interactive slider bars, as follows

```
glLightf(GL_LIGHT0, GL_AMBIENT, [AmbScale, AmbScale, AmbScale])
glLightf(GL_LIGHT0, GL_DIFFUSE, [DiffScale, DiffScale, DiffScale])
glLightf(GL_LIGHT0, GL_SPECULAR, [DiffScale, DiffScale, DiffScale])
```

Where AmbScale, and DiffScale represent the ambient and local intensity scale, between 0.0 and 1.0. The Material properties are then set using the appropriate RGB triplets.

glMaterial(GL_FRONT, GL_AMBIENT, [$dR_{ambient}, dG_{ambient}, dB_{ambient}$*])*

glMaterial(GL_FRONT, GL_DIFFUSE, [$dR_{local+fluor}\ dG_{local+fluor}, dB_{local+fluor}$*])*

glMaterial(GL_FRONT, GL_SPECULAR, [$dR_{specular}\ dG_{specular}, dB_{specular}$*])*

OpenGL receives these material properties, in the correct monitor RGB values, and correctly renders the proper color. The default shading algorithm simply attenuates the RGB values, equally, as the distance from the local light source increases. This is essentially keeping the chromaticities constant, while decreasing the luminance level, thus creating colorimetrically correct shading. If the user alters anything in the scene, such as the light source, or the detector, the entire process is repeated, creating new material properties to get fed into the OpenGL rendering.

It should be noted that if the user does not input a specular reflectance curve, but does enter in a gloss factor, then the resulting material is similar to a shiny plastic. The specular contribution is then thought to be equivalent to the light source (usually thought of as white light). If this is the case, the specular radiance and resulting tristimulus values are calculated using just the light source spectral power distribution.

Digital Camera and Photographic Film

If the user chooses as the detector the Digital Camera, or the Photographic Transparency (Figures 11-12) a slightly different approach is necessary. First the radiances are integrated with the RGB spectral sensitivity curves, to form RGB tristimulus values.

$$R = \int_{300}^{760} Radiance(\lambda) r(\lambda) d\lambda \approx \Delta\lambda \sum_{i=0}^{N+1} Radiance(\lambda_i) r(\lambda_i) \quad (15)$$

$$G = \int_{300}^{760} Radiance(\lambda) g(\lambda) d\lambda \approx \Delta\lambda \sum_{i=0}^{N+1} Radiance(\lambda_i) g(\lambda_i)$$

$$B = \int_{300}^{760} Radiance(\lambda) b(\lambda) d\lambda \approx \Delta\lambda \sum_{i=300}^{N+1} Radiance(\lambda_i) b(\lambda_i)$$

These RGB tristimulus values are then converted into digital RGB monitor values, using the same technique as in Equation 14. The digital RGB values are then auto-white point adjusted by normalizing all values by the RGB triplet of a perfectly reflecting surface. This is accomplished as follows.

$$R_{white} = \int_{300}^{760} \Phi(\lambda) r(\lambda) d\lambda \approx \Delta\lambda \sum_{i=0}^{N+1} \Phi(\lambda_i) r(\lambda_i) \quad (15)$$

$$G_{white} = \int_{300}^{760} \Phi(\lambda) g(\lambda) d\lambda \approx \Delta\lambda \sum_{i=0}^{N+1} \Phi(\lambda_i) g(\lambda_i)$$

$$B_{white} = \int_{300}^{760} \Phi(\lambda) b(\lambda) d\lambda \approx \Delta\lambda \sum_{i=300}^{N+1} \Phi(\lambda_i) b(\lambda_i)$$

In this case, the spectral sensitivities are integrated with the light source spectral power distribution. The normalized digital RGB triplets are then calculated.

$$dR_{ambient} = \frac{dR_{ambient}}{dR_{white}} \quad (16)$$

$$dG_{ambient} = \frac{dG_{ambient}}{dG_{white}}$$

$$dB_{ambient} = \frac{dB_{ambient}}{dB_{white}}$$

The same procedure is done to calculate the contribution of the specular radiance and the combination of diffuse and fluorescent radiances. Once again that forms three sets of digital RGB triplets, which can be forced into the OpenGL shading algorithm by being assigned material properties. This allows for the accurate simulation of any given imaging system.

Chromatic Adaptation

If the user selects any of the three human observers, they also have the option of selecting a chromatic adaptation transform. The optional transforms are None (the default), the von Kries Model, and the Fairchild model.[19] The von Kries Model functions similarly to the white point adjustment used for the RGB imaging system, while the Fairchild Model is an incomplete adaptation transform.

von Kries

The von Kries transform is essentially thought of as proportionality law, where the individual components of vision are independently scaled, or fatigued according to its own function.[19] The adapted color signals in this research are calculated as follows. First, the XYZ tristimulus values are calculated for the radiances, as shown in Equation 11. Next, the XYZ_{white} tristimulus values are calculated. These are essentially calculated by integrating the light source with the spectral sensitivities of the observer, with a perfectly reflecting diffuser.

$$\begin{aligned}
X_{white} &= k\int_{300}^{760}\Phi(\lambda)\bar{x}(\lambda)d\lambda \approx \Delta\lambda \cdot k\sum_{i=0}^{N+1}\Phi(\lambda_i)\bar{x}(\lambda_i) \\
Y_{white} &= k\int_{300}^{760}\Phi(\lambda)\bar{y}(\lambda)d\lambda \approx \Delta\lambda \cdot k\sum_{i=0}^{N+1}\Phi(\lambda_i)\bar{y}(\lambda_i) \\
Z_{white} &= k\int_{300}^{760}\Phi(\lambda)\bar{z}(\lambda)d\lambda \approx \Delta\lambda \cdot k\sum_{i=300}^{N+1}\Phi(\lambda_i)\bar{z}(\lambda_i)
\end{aligned} \tag{17}$$

The XYZ tristimulus values are then converted into LMS cone responses using Equation 18.

$$\begin{bmatrix} L \\ M \\ S \end{bmatrix} = \begin{bmatrix} 0.400 & 0.707 & -0.080 \\ -0.023 & 1.165 & 0.046 \\ 0.000 & 0.000 & 0.912 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{18}$$

The adapted LMS cone signals are then calculated from the LMS values of the color signal, and the white value.

$$\begin{bmatrix} L_a \\ M_a \\ S_a \end{bmatrix} = \begin{bmatrix} 1.0/L_{white} & 0.0 & 0.0 \\ 0.0 & 1.0/M_{white} & 0.0 \\ 0.0 & 0.0 & 1.0/S_{white} \end{bmatrix} \begin{bmatrix} L_{ambient} \\ M_{ambient} \\ S_{ambient} \end{bmatrix} \tag{19}$$

The adapted signals are then converted back into XYZ tristimulus values, using the inverse matrix equation.

$$\begin{bmatrix} X_a \\ Y_a \\ Z_a \end{bmatrix} = \begin{bmatrix} 1.860 & -1.129 & 0.220 \\ 0.361 & 0.639 & -0.000 \\ 0.000 & 0.000 & 1.089 \end{bmatrix} \begin{bmatrix} L_a \\ M_a \\ S_a \end{bmatrix} \tag{20}$$

These adapted XYZ tristimulus values are then converted into RGB and digital RGB counts, using Equations 13 and 14. The final digital RGB counts are then forced into the OpenGL shading algorithm.

Fairchild Model

The Fairchild Chromatic Adaptation Model is implemented in this research as follows. First the XYZ tristimulus values of the color signal, the current light source, and the equal energy illuminant are calculated, using Equation 11 and 17 respectively. The XYZ tristimulus values are converted into LMS cone responses using Equation 18. Next, the degree in which the currently selected light source varies from the equal energy illuminant is calculated.

$$m_E = \frac{3\left(M_{light}/M_{equal}\right)}{L_{light}/L_{equal} + M_{light}/M_{equal} + S_{light}/S_{equal}} \tag{21}$$

This procedure is repeated for the L and S signals as well. The degree of incomplete adaptation is calculated next, using the values calculated in Equation 21, and the Luminance of the light source. Y_n is defined as the light source intensity as chosen using the interactive slider bars.

$$p_M = \frac{\left(1 + Y_n^{1/3} + m_E\right)}{\left(1 + Y_n^{1/3} + 1/m_E\right)} \tag{22}$$

The adapted LMS cone responses are then calculated using Equation 23.

$$\begin{bmatrix} L_a \\ M_a \\ S_a \end{bmatrix} = \begin{bmatrix} p_L/L_{light} & 0.0 & 0.0 \\ 0.0 & p_M/M_{light} & 0.0 \\ 0.0 & 0.0 & p_S/S_{light} \end{bmatrix} \begin{bmatrix} L_{ambient} \\ M_{ambient} \\ S_{ambient} \end{bmatrix} \tag{23}$$

The adapted LMS signals are then transformed back into adapted XYZ tristimulus values, using the inverse matrix of Equation 20. The XYZ tristimulus values are converted to RGB tristimulus values, and then into digital RGB counts using Equations 13 and 14. The resultant adapted digital RGB counts are then used as the material properties in the OpenGL shading algorithm.

Examples of Interactive Spectral Rendering

This sections shows examples of screen captured images rendered using the spectral Scene Viewer. Included are examples of simple scenes, as well as examples of metamerism, fluorescence, metallic surfaces, and texture maps.

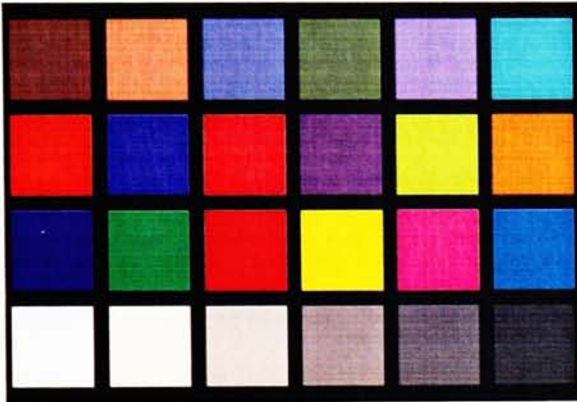


Figure 22. Rendered image of a Macbeth ColorChecker.

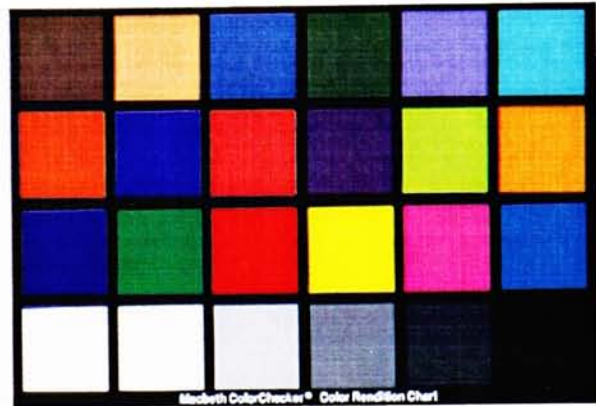


Figure 23. Digitized image of a real Macbeth ColorChecker.

Spectral Rendering of a Macbeth ColorChecker

Figures 22 and 23 show a side by side example of a Macbeth ColorChecker. The image on the right, Figure 23 shows a digitized image of a real ColorChecker, while Figure 22 shows a rendered image of the ColorChecker. The rendered image was computed using the measured diffuse spectral reflectance functions of the original Macbeth ColorChecker, CIE D65, and the 1931 Standard Observer. It can be seen that while the two images are slightly different, the colors themselves are comparable. The differences can be attributed to the fact that one of the rendering technique is not spectral, but rather a simple RGB digitization. This shows that the technique presented here is feasible, and can be extended to more complicated scenes.

Rendering and Modeling

While Figure 22 shows the result of the color calculation portion of this research, it does so only

Figure 24. Rendered Macbeth ColorChecker using toruses rather than squares.

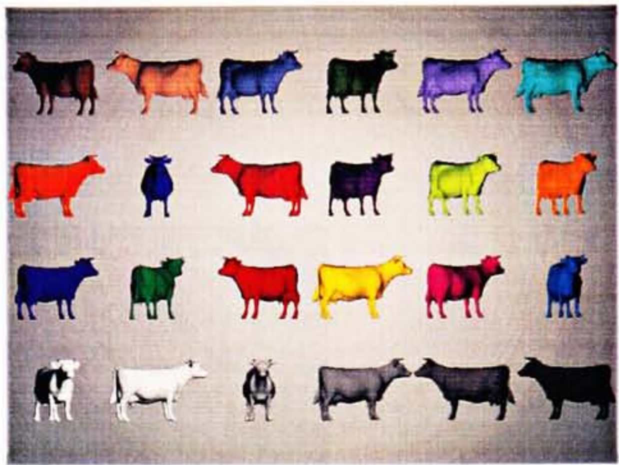


Figure 25. Rendered Macbeth ColorChecker using cows rather than squares.

with two-dimensional objects. While useful, the goal of this research is not limited to two dimensions. The same techniques can be used with more complicated objects, as shown in Figures 24 and 25. It should be noted that the rendering technique described in this research in independent of the ultimate three-dimensional models. The end-user is not limited to the default models as shown in Figure 11. Built into the software is the *glm* library, which is capable of reading in complicated Wavefront OBJ models.[20]

Observer Metamerism

Another important phenomena that can not be rendered accurately with out spectral information is that of metamerism. Metameric pairs are defined to be two objects with differing spectral properties that appear to match in color. The fundamental reason for this is the property of

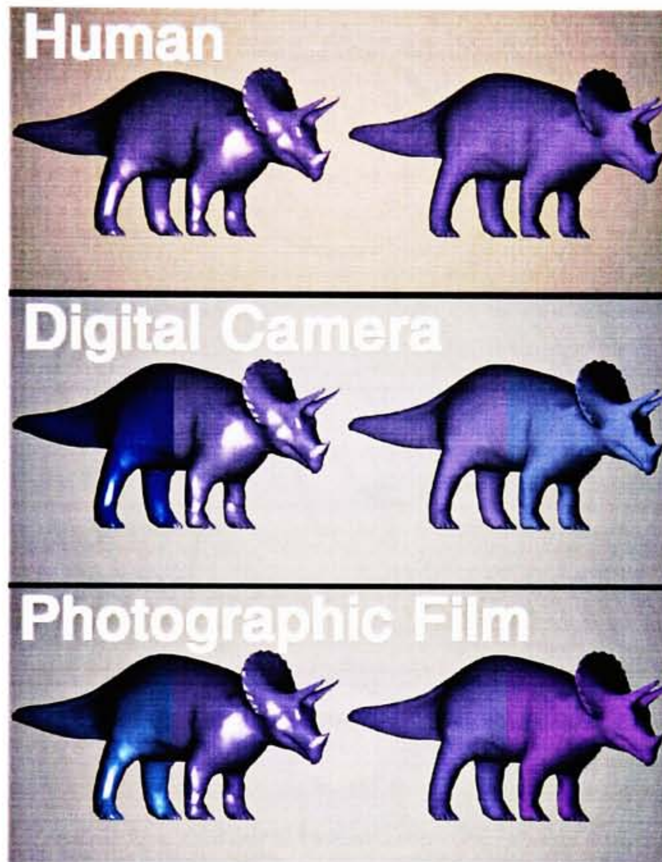


Figure 26. Example of Observer Metamerism

univariance, and the integration of the spectral power distributions into three cone responses. Since the spectral properties of a metameric pair vary, it is possible for a metameric pair to match under one condition, and to not match should that condition change. Thus, if two metameric objects match to a human observer, they may no longer match if a different imaging system is used. This is demonstrated in Figure 26. In this Figure, the top portion shows a scene as viewed

with the 1931 Standard Observer, under CIE Illuminant D65. The middle portions shows the same scene, with the detector being changed to the Digital Camera, while the bottom portion shows the scene with the detector being a photographic film. As mentioned above, the interactive Scene Viewer allows materials to be defined with a metameric match. When this is the case, an object is cut in half, with each half being a metameric match. In the case of Figure 26, there are three different spectral reflectances present. These are illustrated in Figure 27. The metameric

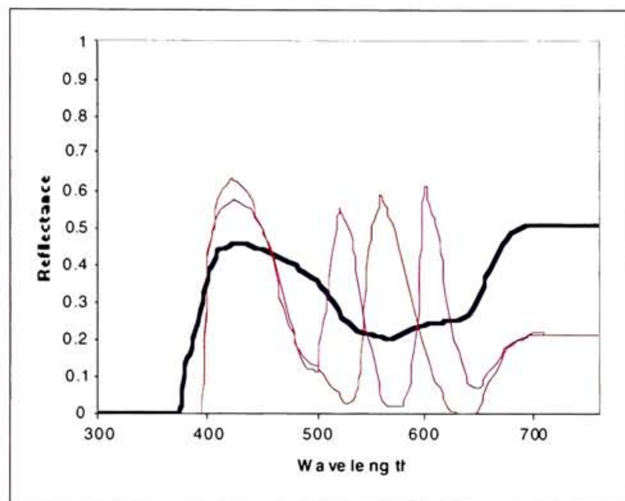


Figure 27. Spectral Reflectance Curves of the three metameric pairs.

pairs were calculated by adding metameric blacks onto the bold reflectance curve in Figure 27.[21] If the RGB color model was used to assign light sources and material properites, materials that matched under one condition, would inherently match under another. Full spectral information is needed to demonstrate observer metamerism, as above, as well as illuminant metamerism.

Illuminant Metamerism

As described above metameric pairs might often match under one given viewing condition, and

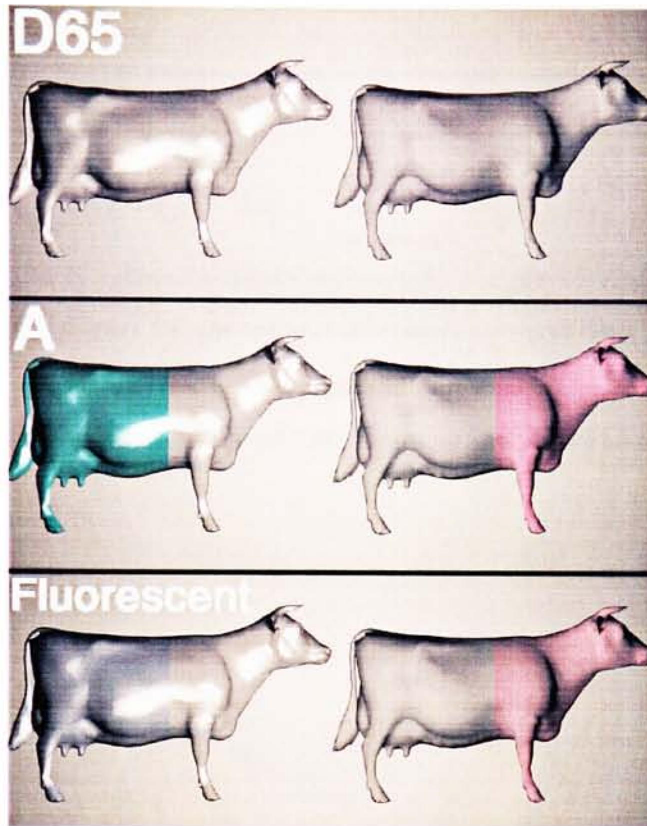


Figure 28. Example of Illuminant Metamerism.

not under another. This is not limited to changing the detector. If the light source were to change, often the metameric match breaks down as well. This is illustrated in Figure 28. The top portion of Figure 28 shows a scene illuminated by CIE Illuminant D65. The middle portion shows this same scene as illuminated with CIE Illuminant A. The bottom portion shows the scene Illuminated with a fluorescent light source. It can be seen that both halves of the cows match, and the cows match each other, under D65. These matches break down as the illumination changes. The spectral reflectance curves for these objects are shown in Figure 29. The bold curve

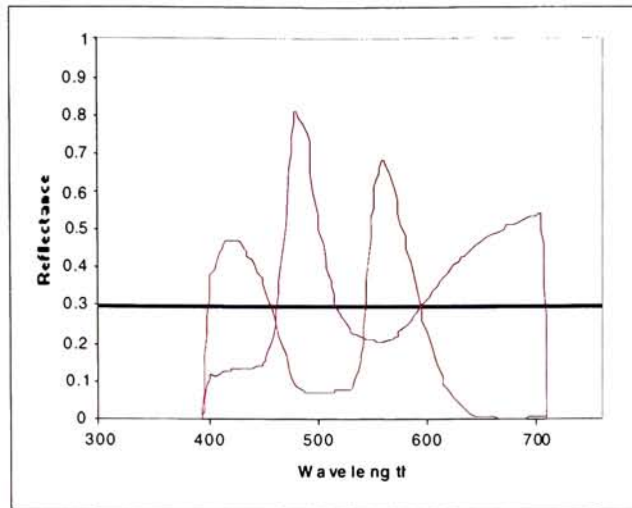


Figure 29. The spectral reflectance curves of the metamers shown in Figure 28.

represents a non-selective gray of 30% reflectance, while the other two curves are metameric blacks added onto the gray.

Fluorescence

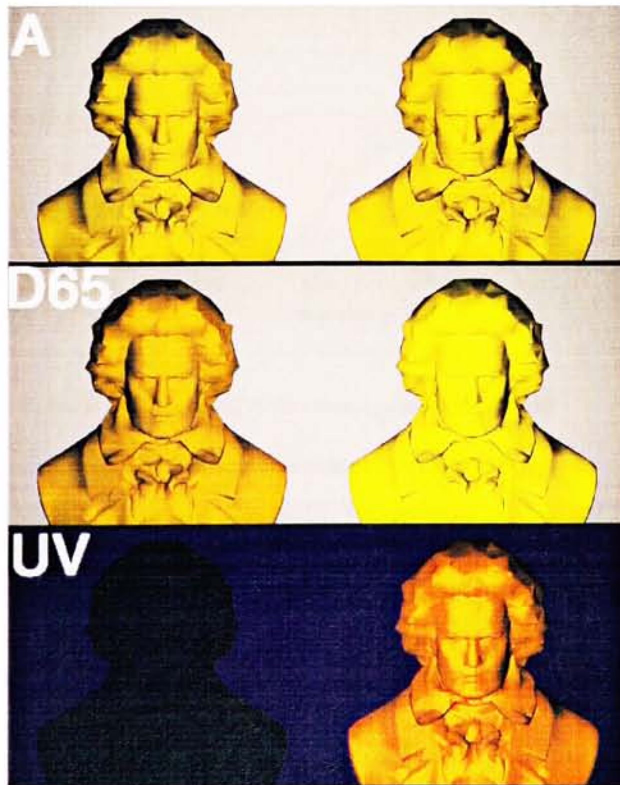


Figure 30. Example of Fluorescence.

Fluorescent materials are typically materials that absorb energy in one region of the spectrum, and re-emit this energy in a longer wavelength region of the spectrum. Part of the material descriptions for objects in this software can include both excitation and emission spectra. Therefore, it is possible to simulate and render fluorescent materials. This is demonstrated in Figure 30. The top portion of the figure illustrates two objects with identical spectral reflectance curves, illuminated under CIE Illuminant A. The middle portion of the figure shows these same objects illuminated by CIE Illuminant D65, while the bottom portion shows the objects under a black light (UV). It can be seen that the two objects appear to match under Illuminant A, while no longer matching under the other Illuminants. This apparent mismatch is cause by the

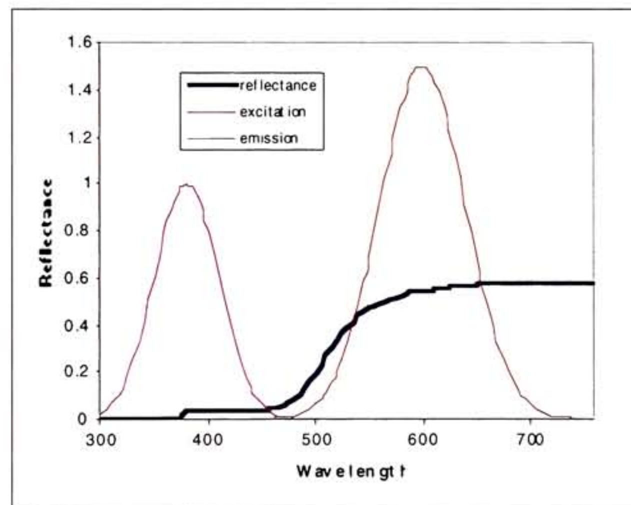


Figure 31. Spectral reflectance, excitation, and emission curve of the fluorescent sample.

fluorescent properties of the object on the right. Illuminant A has little energy in the shorter wavelength regions, where the excitation region is defined. Instead, most of the energy is situated in the longer wavelength regions where the two reflectance curves are identical. Illuminant D65 has energy in both the long wavelength region and in the excitation region. The UV light has little

energy in the visible regions, where the spectral reflectance is higher, so the object on the left all but disappears. The UV light does have a large amount of energy in the excitation region of the object on the right, resulting in great amount of emission in the visible region. Figure 31 shows the spectral reflectance and excitation and emission curves of the objects in Figure 30.

Metallic Surfaces

Most objects are not perfectly diffuse, and have some element of gloss. A typical paper or plastic might have first surface reflections, typically the specular reflection having the same spectral property as the light source. Metallic surfaces, on the other hand, actually demonstrate

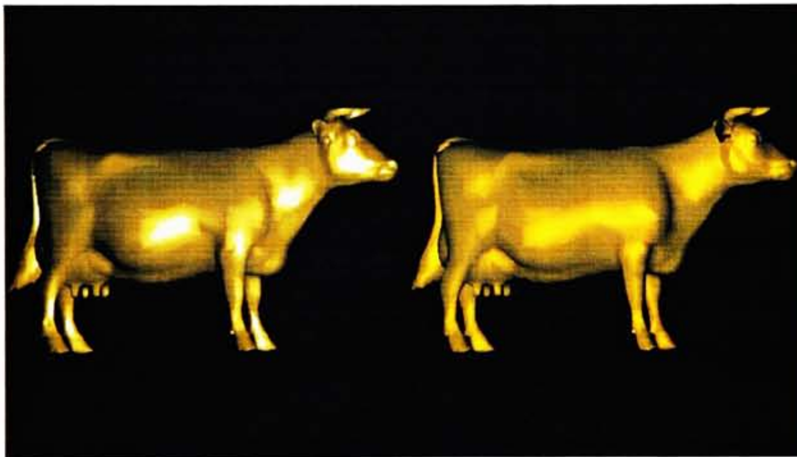


Figure 32. Example of metallic objects, with identical diffuse reflectance and differing specular reflectance.

interactions between the material and the specular component. Thus, the specular reflection is often a unique reflectance curve. Figure 32 illustrates an example of two objects with identical diffuse reflectance functions, but different specular reflectance functions.

Figure 33. An example of simulated full spectral texture mapping.

Texture Mapping

A common technique in computer graphics to increase the apparent realism of rendered images is the use of texture maps. Essentially this is applying an image like a decal around an object. OpenGL, as stated above, provides a mechanism for applying texture maps. The color calculation technique presented in this research includes the ability to attach full spectral texture maps to any object. Since full spectral images are difficult to obtain, it is possible to simulate a full spectral texture map by applying a monochromatic texture map to an object in Modulation mode. By placing a monochromatic image onto an object that has a predefined spectral material property, it is possible to simulate full spectral texture mapping. Figure 33 shows an example of this, using the Macbeth ColorChecker as the base material properties, and 24 different texture maps. An example of texture mapping using a full spectral texture map generated using the Scene Viewer itself is shown below.

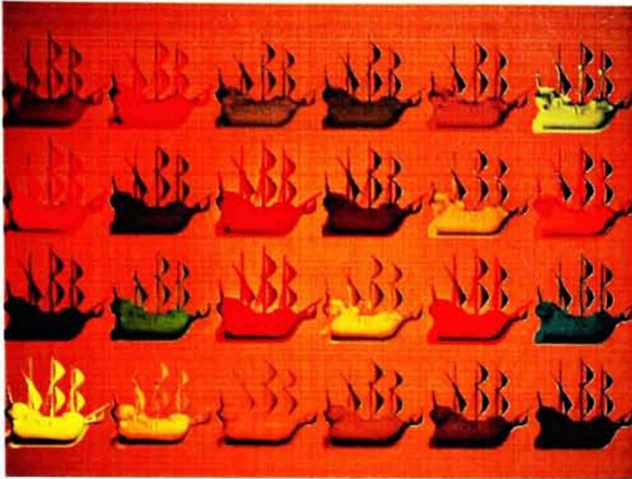


Figure 34. Macbeth ColorChecker under Illuminant A. no Chromatic Adaptation.

Figure 25. Macbeth ColorChecker under Illuminant A. von Kries Chromatic Adaptation.

Figure 36. Macbeth ColorChecker under Illuminant A. Fairchild Chromatic Adaptation.

Chromatic Adaptation

The methods for computing chromatic adaptation were discussed earlier. Figures 34-36 show examples of the three choices of chromatic adaptation, under CIE Illuminant A.

Writing Full Spectral Images

Up until now, the bulk of discussion has been on the interactive viewing of rendered scenes, after the spectral color calculations have been performed. Perhaps the most important aspect of this research is the ability to render full spectral images not for display purposes, but rather into an image file. The display device no longer is the limiting factor, and images can be created with arbitrary spatial and spectral resolution. The interactive Scene Viewer allows the user to interactively manipulate the viewing conditions and view the resulting effects on the colors. The Scene Viewer also contains a function for writing the current scene into a full spectral image.

The Save-As Option

Once the end-user is satisfied with the current scene, they are given the opportunity to save the scene into a full spectral image. That is to say, the final image will have spectroradiometric values

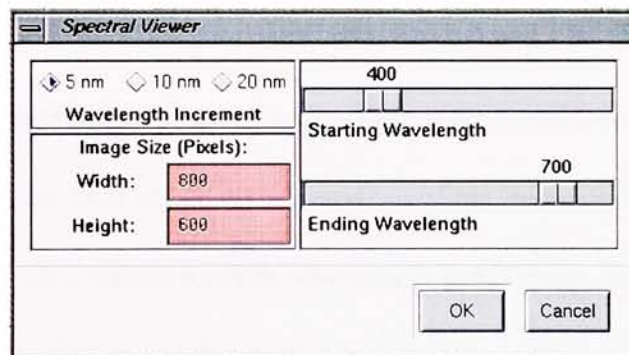


Figure 37. The Save-As dialog from the Spectral Scene Viewer.

of arbitrary wavelength resolution, at each pixel. The first step is to choose the Save As option from the File Menu. This brings up the dialog box shown in Figure 29. The user may chose the wavelength increments of 5, 10, or 20 nm. The user may also choose the desired image size, in pixels. The largest image size is only hardware limited, and is currently 2000x2000 on most

graphics workstations. The starting and ending wavelength scale can also be chosen. This allows for great flexibility, as the user can create images ranging from 1 to 93 dimensions (wavelength bands). Once the desired dimensions are selected, the image output begins. If necessary the software is easily expandable to include more wavelength bands.

Rendering Loop

The spectral image is written using a rendering loop, or a multi-pass technique. The first step of the interactive viewing process was to calculate the spectral interactions between the materials and the light sources, so the scene radiances are already computed. For each rendering pass, the material properties of the rendered objects are defined as follows:

glMaterial(GL_FRONT, GL_AMBIENT, [RadianceAmbient, 0.0, 0.0])

glMaterial(GL_FRONT, GL_DIFFUSE, [RadianceDiffuse + Radiance Fluorescent, 0.0, 0.0])

glMaterial(GL_FRONT, GL_SPECULAR, [RadianceSpecular, 0.0, 0.0])

Only the first of the material RGB triplet is defined, the R, while the G and B are defined to be 0.0. The light sources are defined as an intensity between 0.0 and 1.0. This allows each object to be rendered with the total radiance based upon the light source, and the material properties for a single wavelength. This rendered image is then read out of the frame buffer using the OpenGL glReadBuffer command. This creates a single wavelength band image, for each pass of the rendering loop. These wavelength bands can then be stacked together, to form a full spectral image.

Frame buffer Choices

The depth of the spectral image ultimately depends on the depth of the frame buffer, since each wavelength band image is read directly from the frame buffer. The Scene Viewer has two options

for determining how the spectral image is actually rendered, as well as the depth of the frame buffer. The first option is the standard default on-screen rendering. In this case, each wavelength band is rendered first to the display screen, and then the wavelength band image is read from the screen frame buffer. In this method, the Scene Viewer attempts to configure the frame buffer to a 12-bits/pixel resolution. On high-end workstations, such as the SGI InfiniteReality, this configuration is possible, and the resultant wavelength band images contain 12-bits/pixel. If the Scene Viewer is unable to configure the frame buffer at 12 bits/pixel, it attempts to configure in lower resolutions: either 10-bits/pixel or 8-bits/pixel. Using this on-screen rendering technique, it is possible to create a full spectral image with up to 12-bits per pixel, using special graphics hardware. The downfall of the on-screen rendering technique is that spatial resolution of the image is limited by the resolution of the screen. If the current screen resolution is set to 640x480 pixels, that is the largest spectral image that can be created. If the end-user is forced to use an on-screen rendering technique, a method of image tiling could be utilized to create larger images.

The Scene Viewer also attempts to configure an off screen rendering buffer, called the GLXPbuffer, or a GLX pixel buffer. These pixel buffers are generally only available on graphics workstations, such as the SGI InfiniteReality. There are several advantages to using an off-screen pixel buffer, rather than using the on-screen technique. The pixel buffer can be configured to any resolution, up to the total size of the frame buffer (usually 2000x2000). Thus if the screen resolution is set lower, the pixel buffer can still be used to create a larger image. The other advantage of using a pixel buffer is that some machines, such as the SGI InfiniteReality, are capable of configuring the frame buffer to use a 16-bit luminance channel, rather than multiple

12-bit RGB channels. The glReadBuffer command is capable of reading luminance information, allowing the creation of spectral images with 16-bit/pixel resolution.

Image File Format

Once the stack of wavelength-band images are created, it is necessary to convert the image array into a file. The HDF file format is used to save the resulting spectral images.[22] This format has several essential features worth noting. It is capable of saving N-dimension files, appropriate for saving spectral images of arbitrary wavelength depth. This format can also save images from ranging from 8 and 16 bit integers, up to 64-bit floating-point numbers. In addition, many of today's higher level languages such as Matlab and IDL are able to read and write HDF images.

More Advanced Scenes

While the Scene Viewer defaults to creating images much in the same layout as the Macbeth ColorChecker, it is a simple extension to create scenes that are more complicated. This is easily accomplished since the spectral color calculations are isolated from the rendering code. The following Figures 38-42 illustrate more complex images. It is important to note that while these images are far more complex, there is no loss in the interactive nature of the Scene Viewer.

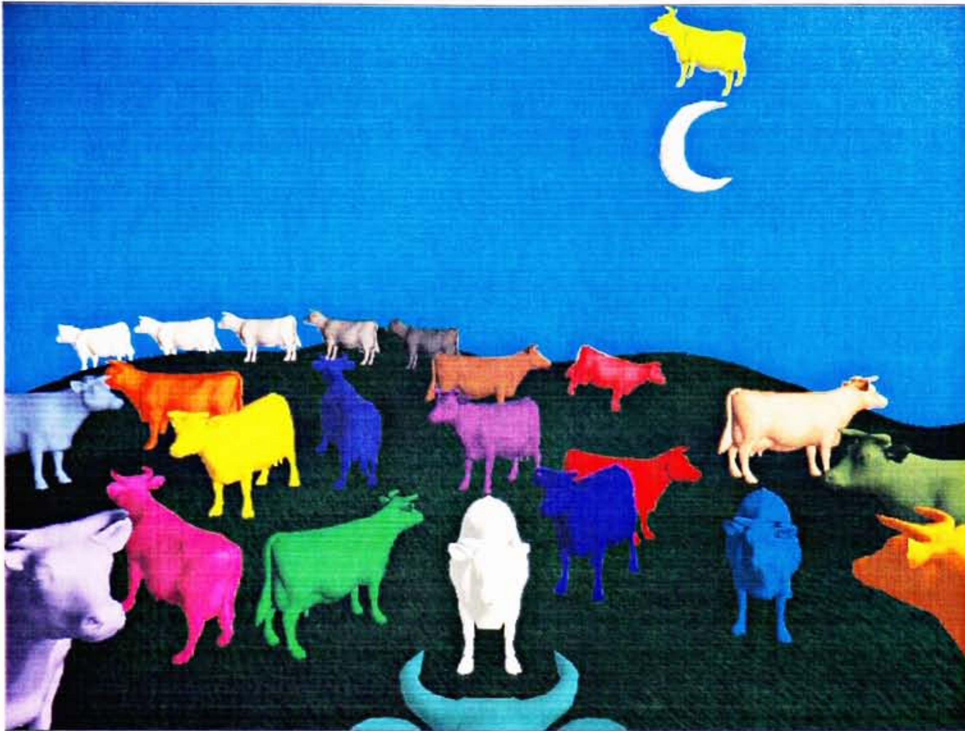


Figure 38. A more complex scene utilizing the Macbeth ColorChecker material properties and CIE Illuminant D65. The grass is actually a texture mapped surface.

Figure 39. A more complicated scene, including metameric, fluorescent, metallic, and texture mapped objects. The image on the wall is an actual full spectral texture map created using a multi-pass rendering technique. This scene is illuminated by CIE Illuminant D65.

Figure 40 The same scene as above, illuminated by CIE Illuminant A, using Fairchild Adaptation. Notice the metameric chairs no longer match.

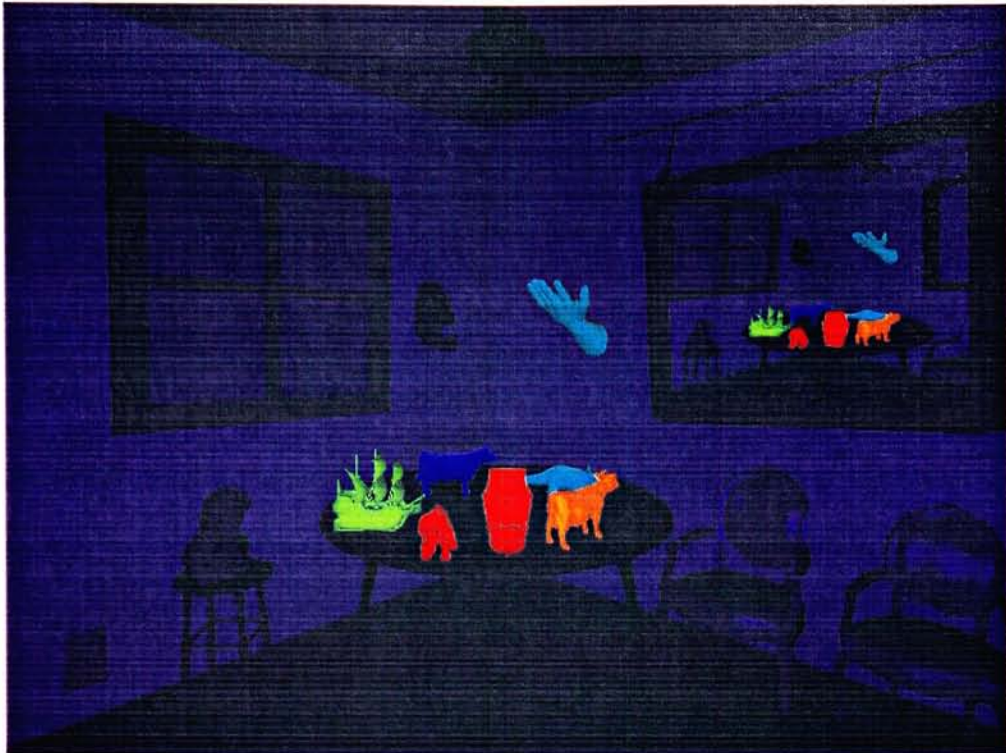


Figure 41. Same Scene as above, illuminated by the UV light source. Notice the Fluorescent items on the table and wall.

Future Directions

As can be seen from the above images, the software created for this research is capable of creating some rather complex images. There are still some possibilities for improvement, however. Currently each object can only be described with a single material file. In the future, this could be extended to allow a single object to have multiple material properties, for different regions of the object. This would allow for the creation of more realistic objects. Another downfall of using the OpenGL local illumination lighting model is the lack of interactions between the objects. This is most obvious in the lack of shadows cast from one object onto another. While the only truly accurate technique to account for interactions would be a global illumination algorithm such as

raytracing or radiosity, the interactive performance would inherently suffer. Perhaps a compromise could be reached, using a shadow casting algorithm within OpenGL.

Other improvements to the software could include the use of bump-mapping algorithms to allow the user to interactively alter the surface structure of the rendered objects. This could be useful for some imaging systems analysis and simulations. The software has been extended to handle multiple light sources, using the accumulation buffer. The accumulation buffer allows for a multi-pass rendering technique, maintaining full color bit-depth. Essentially the full scene is rendered once for each light source, and the resultant image is stored in the accumulation buffer. Once all the light sources have been accounted for, the accumulation buffer merges all the images together, and returns a single rendered image to the framebuffer. This image is then rendered to the screen, or to the full spectral image.

Finally, extensions to the current software could also be easily made to allow the creation of larger full spectral images. Currently the problem lies in the computer hardware, rather than the software, as creating larger images requires a huge amount of computing resources.

Conclusion

A technique to perform full spectral based color calculations through an extension of OpenGL has been created. With this extension it is possible to synthesize spectroradiometric images for use in color imaging systems analysis. These full spectral images can be created of arbitrary spatial and spectral resolution, up to 2000x2000x93, with 16-bits per pixel. This method of color

computations is also more accurate than the standard RGB model that most computer graphics algorithms utilize. This method also has some distinct advantages over other spectral rendering techniques, as it fully maintains the wavelength information during color calculations. By maintaining full spectral information, the method provides equal or more accurate color calculations than those with simplified wavelength information. Perhaps the most distinct advantage of maintaining full wavelength information is the ability to render full spectral images, whereas a mathematical simplification technique would be inadequate.

A Spectral Scene Viewer was also created that can interactively simulate and display important color phenomena such as metamerism and fluorescence. This viewer is a useful teaching tool for demonstrating the effects that wavelength manipulations and viewing condition changes have on resulting colors.

References

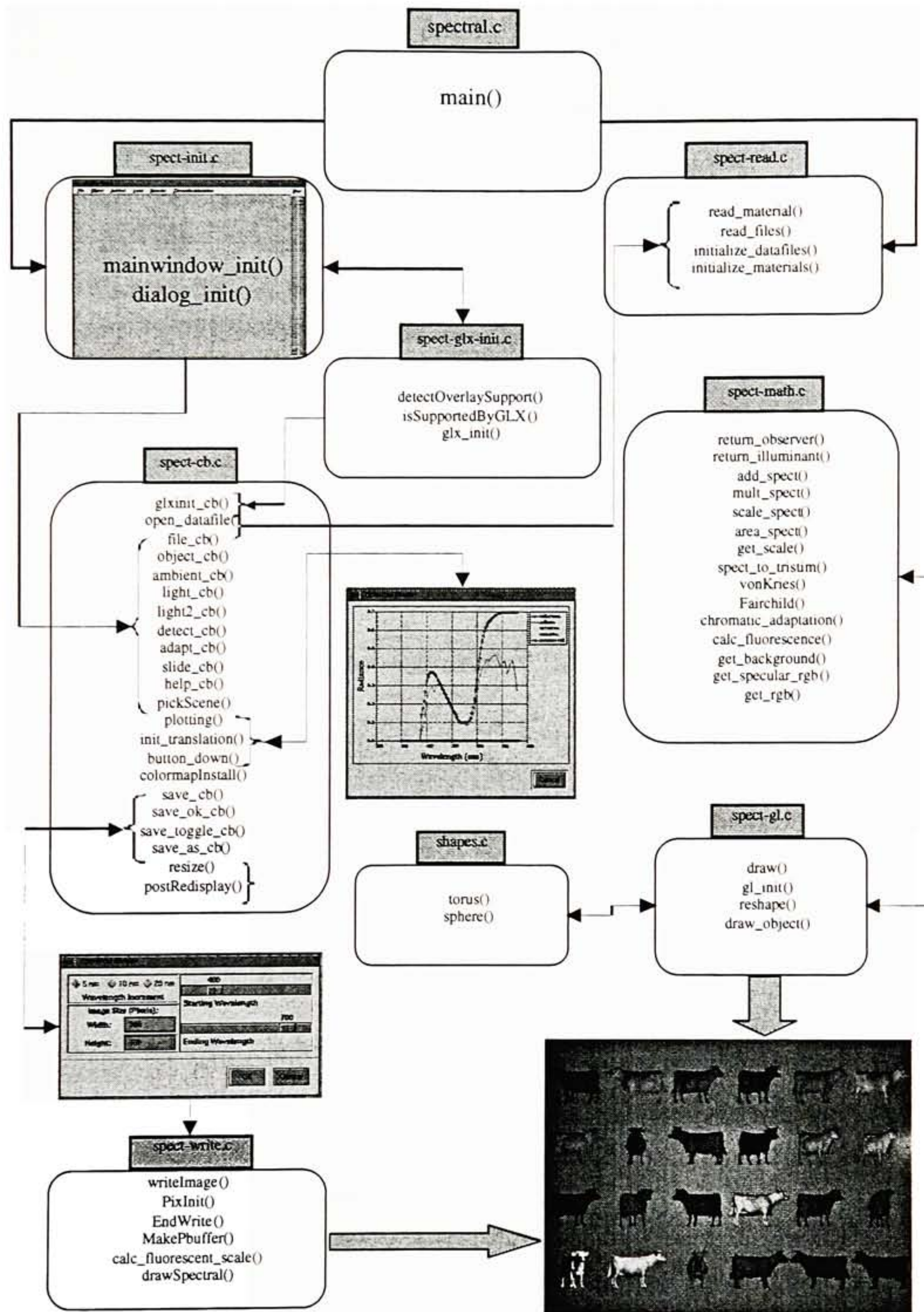
1. M.S. Peercy, B.M. Zhu, D.R. Baum, Interactive Full Spectral Rendering, *Proceedings 1995 Symposium on Interactive 3d Graphics.* 218, 67-68, 207 (1995).
2. R. Hall, *Illumination and Color in Computer Generated Imagery.* Springer, Berlin, (1989).
3. R.S. Berns, Methods for Characterizing CRT Displays, *Displays*, **16**, 173-182 (1996).
4. B.A. Wandell, The Synthesis and Analysis of Color Images, *IEEE Transactions of Pattern Analysis and Machine Intelligence.* 9, 2-13 (1987).
5. M.S.Peercy, Linear Color Representation for Full Spectral Rendering, *Computer Graphics (SIGGRAPH '93 Proceedings)*, **27**. 191-198, (1993).
6. R. Wallis, Fast Computation of Tristimulus Values by use of Gaussian Quadrature, *J. Opt. Soc. Am..*, **65**, 91-94 (1975).
7. G.W. Meyer, Wavelength Selection for Synthetic Image Generation, *Computer Vision, Graphics, and Image Processing*, **41**, 57-79 (1988).
8. P.M. Deville, S. Merzouk, D. Cazier, J.C. Paul, Spectral Data Modeling for Lighting Application, *Computer Graphics Forum*, **13**, C97-C106 (1994).
9. M.S. Peercy, F.R. Baum, B.M. Zhu, Linear color representations for efficient image synthesis, *Color Res. Appl.* **21**, 129-37 (1996).
10. L.T. Maloney, Evaluation of Linear Models of Surface Spectral Reflectance with Small Numbers of Parameters, *J. Opt. Soc. Am. A.*,**3**, 1673-1683 (1986).
11. D.H. Marimont, B.A. Wandell, Linear Models of Surface and Illuminant Spectra, *J. Opt. Soc. Am. A.* **9**, 1905-1913 (1992).
12. J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes, *Computer Graphics: Principles and pracitce,* Addison Wesley, Reading MA, (1992).
13. E. Angel, *Interactive Computer Graphics: A Top-Down Approach with OpenGL,* Addison-Wesley, Reading, MA (1997).
14. D. Hearn, and M.P. Baker, *Computer Graphics,* Prentice Hall, Englewood Cliffs, NJ (1994).
15. M.Woo, J. Neider, T. Davies, *OpenGL Programming Guide*, Addison-Wesley, Reading, MA (1997).
16. J.S. Montrym, D.R. Baum, D.L. Dignam, G.J. Migdal, InfiniteReality: A Real-Time Graphics System, *Computer Graphics (SIGGRAPH '97 Proceedings)*, 293-302, (1997).
17. P.Haeberli, K. Akeley, The Accumulation Buffer: Hardware Support for High-Quality Rendering, *Computer Graphics (SIGGRAPH '90 Proceedings)*, **4**, 309-318 (1990).
18. R. McMullen, The SciPlot Widget Library, *http://www.ae.utexas.edu/~rwmcm (1996).*
19. M.D. Fairchild, *Color Appearance Models*, Addison Wesley, Reading MA, (1998).

20. N. Robbins, The glm Rendering Library, *http://trant.sgi.com/opengl/examples/more_samples/smooth/* (1996).

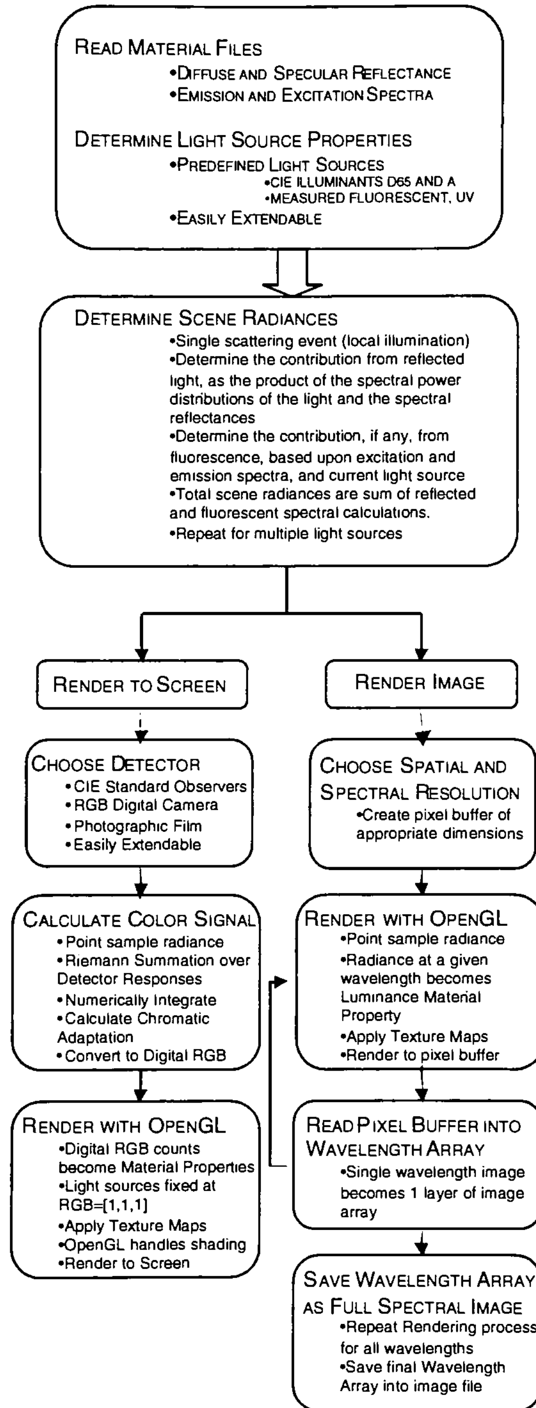
21. G. Wyszecki, and W.S. Stiles, *Color Science: Concepts and Methods, Quantitative Data and Formulae*, John Wiley & Sons, New York, NY (1986).

22. NCSA, HDF file format, *http://hdf.ncsa.uiuc.edu* (1997).

A. Appendix A: Flow Chart of spectral Software



Flow Chart of Software processes



B. Appendix B: Interactive Viewer Code.

spectral.h

```
/*
 * spectral.h
 *
 * Written By: Garrett M. Johnson
 *
 * Header file for the full spectral rendering package
 * using OpenGL, and Motif.
 *
 * This file contains all of the old global variables, but
 * please do not mock the author for using global stuff.
 * This file also contains function declarations, and tells
 * you where to find said functions.
 *
 * Last revised: 07/27/98
 *
 */


#include   <X11/Xlib.h>
#include   <X11/Intrinsic.h>
#include   <GL/glx.h>
#include   <sys/param.h>
#include   <GL/gl.h>
#include   <GL/glu.h>
#include   <stdio.h>
#include   "sovLayerUtil.h"


/* define the structure containing the observer response arrays */

typedef struct{ double s1[93], s2[93], s3[93] ;
       } RESPONSE   ;

/* define the structure which contains the material properties for
 * each object. this structure contains the spectral diffuse, specular
 * excitation, and emission arrays. this structure also contains the
 * shininess (gloss) factor, and the name of the texture file (if
 * applicable), as well as Booleans to determine if the object has
 * fluorescence, specular, metameric, or texture properties.
 */

typedef struct{ double diffuse[93],  spec[93],  exc[93], em[93] ;
       double shininess ;
       char *texture ;
       int  numTex ;
       Bool Fluor, Spec, Metameric, Texture ;
       } MATERIALS   ;


/* enumerate the variables for the user interface options */

enum { D65, A , F1, BLACK) ;
enum { XYZ2, XYZ10, STDEV, SCANNER, PHOTO) ;
enum { TORUS, SPHERE, COW, SHIP, WOMAN, TRICER) ;
enum { AMBIENT, LOCAL, LOCAL2 ) ;
enum { NONE, VONKRIES, FAIRCHILD ) ;

/*  standard malloc function   */

void   *malloc() ;

/* the following functions are the globalesque functions, or they need
```

```
 * to be seen outside their respective files. greater definition to what
 * each function actually does can be found in their files.
 */

/* functions in spect-read.c that need to be seen by other files */

void   read_material(FILE **file, MATERIALS *data, MATERIALS *met) ;
void   read_files(char **names, FILE **file) ;
FILE    *initialize_datafile(char *name) ;
void    initialize_materials(char **file_names) ;


/* functions in spect-math.c */

void   return_observer(int name, RESPONSE *response) ;
void   return_illuminant(int name, double *curve) ;
void   add_spect(double *curve1, double *curve2, double *sum) ;
void   mult_spect(double *curve1, double *curve2, double *product) ;
void   scale_spect(double *curve, double scale, double *scaled_curve) ;
double area_spect(double *curve) ;
double  get_scale(double *y) ;
void   spect_to_tristim(RESPONSE *xyz2, double *radiance, double *xyz) ;
void   vonKries(double *XYZ, double *XYZa, int light) ;
void   Fairchild(double *XYZ, double *XYZa, int light) ;
void    chromatic_adaptation(double *xyz, double *XYZa, int light) ;
void   calc_fluorescence(double *excite, double *emit, double *radiance, int light) ;
void   get_background(double *amb_rgb, double *diff_rgb) ;
void   get_rgb(double rgb[][3], double met_rgb[][3], int light) ;
void   get_specular_rgb(double spec_rgb[][3], double met_spec_rgb[][3], int light) ;


/* functions in shapes.c  */

void   sphere(GLdouble radius, GLint slices, GLint stacks) ;
void    Torus(GLfloat r, GLfloat R, GLint nsides, GLint rings) ;
void   rect(GLdouble length, GLdouble height) ;



/* functions in spect-init.c */

void    mainwindow_init(Widget toplevel) ;
void    dialog_init(Widget toplevel) ;


/* functions in spect-gl.c  */
void   draw(GLenum mode) ;
void    gl_init(void) ;
void   reshape(int width, int height) ;


/*  function in spect-glx-init.c   */

void   detectOverlaySupport(Display *dpy) ;


/* functions in spect-cb.c  */

void    postRedisplay(void) ;
void    colormapInstall(Widget w, XtPointer clientdata, XtPointer callData) ;
void    glxinit_cb(Widget w, XtPointer clientData, XtPointer callData) ;
void   file_cb(Widget w, XtPointer clientData, XtPointer callData) ;
void   object_cb(Widget w, XtPointer clientData, XtPointer callData) ;
void   ambient_cb(Widget w, XtPointer clientData, XtPointer callData) ;
void   light_cb(Widget w, XtPointer clientData, XtPointer callData) ;
void   light2_cb(Widget w, XtPointer clientData, XtPointer callData) ;
void   detect_cb(Widget w, XtPointer clientData, XtPointer callData) ;
void   adapt_cb(Widget w, XtPointer clientData, XtPointer callData) ;
void   slide_cb(Widget w, XtPointer clientData, XtPointer callData) ;
void   help_cb(Widget w, XtPointer clientData, XtPointer callData) ;
void    pickScene(int x, int y) ;
void   plotting(int name) ;
```

```
void    init_translation(void) ;
void    open_datafile(Widget w, XtPointer clientData, XtPointer callData) ;
void    button_down(Widget w, XEvent *event, String *params, Cardinal *num_params) ;
void    resize(Widget w, XtPointer clientData, XtPointer callData) ;
void    save_cb(void) ;
void    save_ok_cb(Widget w, XtPointer clientData, XtPointer callData) ;
void    save_toggle_cb(Widget w, XtPointer clientData, XtPointer callData) ;
void    save_as_cb(Widget w, XtPointer clientData, XtPointer callData) ;


/* function to write out the spectral image, in spectral-write.c    */

void                    writeImage(int width, int height, int incrementWL,
                                int startWL, int endWL, char *file) ;
int                     PixInit(int width, int height) ;
void                    EndWrite(void) ;
GLXPbufferSGIX          MakePbuffer(Display *Dpy,int screen, int width, int height) ;
GLfloat                 calc_fluorescent_scale(double *excite, double *illuminant) ;
void                    drawSpectral(int wavelength) ;



/* and now for the global variables...please do not cringe at them. */


/* Widgets and important variables declared in spectral.c   */

extern Widget             main_w ;      /* the main window  */
extern Widget             menubar ;     /* holds the file menus */
extern Widget             widget ;      /* a dummy widget */
extern Widget           file_menu ; /* the pull down file menu */
extern Widget             form ;        /* the container widget */
extern Widget             frame ;       /* another container type widget */
extern Widget             glxarea ;     /* the glx drawing area widget */
extern Widget             ob_menu ;     /* the objects menue */
extern Widget             am_menu ;     /* the ambient lighting menu */
extern Widget             li_menu ;     /* the local lighting menu */
extern Widget             li_menu2 ;    /* the local lighting menu */
extern Widget             det_menu ;   /* the detector menu */
extern Widget             adapt_menu ; /* the chromatic adaptation menu */
extern Widget             ambient_slide ; /* the ambient intensity slider */
extern Widget             local_slide ;   /* the local intensity slider */
extern Widget             local2_slide ;   /* the local intensity slider */
extern Widget             filebox ;        /* the pop-up filebox */
extern Widget             help_dialog ;    /* the help dialog box */
extern Widget             plot_dialog ;    /* the plot dialog box */
extern Widget             plot_widget ;    /* the plotting widget */


/* the following widgets make up the save_as option dialog */

extern Widget           dialog;
extern Widget           save_form;
extern Widget           increment_frame ;
extern Widget           increment_form ;
extern Widget           wl_label;
extern Widget           toggle_box ;
extern Widget           wl_frame ;
extern Widget           wl_form ;
extern Widget           begin_wl ;
extern Widget           end_wl ;
extern Widget           size_frame ;
extern Widget           size_form ;
extern Widget           size_form2;
extern Widget           size_form3;
extern Widget           size_label ;
extern Widget           x_label ;
extern Widget           y_label ;
extern Widget           x_text ;
extern Widget           y_text ;
extern Widget           save_filebox ;
```

```
extern XtAppContext   app ;                         /* the application context */
extern Display          *display ;                  /* the display variable */
extern Window           glxwin ;                    /* the glx window */
extern GLXContext       glxcx ;                          /* the glx context */
extern Dimension      viewWidth, viewHeight ;            /* the the drawing dimensions */
extern XVisualInfo      *vi ;                             /* the visual info  */
extern Visual           *overlayVisual ;            /* the visual */
extern int              overlayDepth ;                   /* the bit-depth of the overlay */
extern Colormap         overlayColormap ;           /* the overlay colormap */
extern Bool             doubleBuffer ;                   /* are we double-buffered? */


/* The following variables are first declared in spect-read.c  */

/* the variables to determine a given materials properties  */

extern Bool           RECALCULATE_COLORS ;                 /* if anything changes... */
extern Bool           METAMERIC[]  ;                /* is the object metameric?  */
extern Bool           FLUORESCENT[]  ;              /* is the object fluorescent */
extern Bool           SPECULAR[]  ;                 /* is the object specular */
extern Bool           TEXTURE[]  ;                       /* does the object have a tex? */
extern Bool           MET_FLUORESCENT[]  ;               /* if metameric, fluorescent? */
extern Bool           MET_SPECULAR[]  ;             /* if metameric, specular? */


/* global material properties data....  */
extern double         reflect[][93] ;       /* global reflection spectra */
extern double         excite[][93] ;        /* global excitation spectra */
extern double         emit[][93] ;          /* global emmision spectra */
extern double         specular[][93] ;      /* global specular spectra */
extern double         metamer[][93] ;       /* global metameric diffuse spectra */
extern double         met_excite[][93] ;         /* global metameric excitation spectra */
extern double         met_emit[][93] ;      /* global metameric emission spectra */
extern double         met_specular[][93] ;    /* global metameric specular spectra */
extern double         gloss[] ;             /* global gloss component */
extern double         met_gloss[] ;         /* global metameric gloss component */
extern int            NumTex[] ;            /* number and position of textures */


extern int            NUM_FILES ;           /* number of material files  */


/* global variables that determine the SaveAs properties */

extern int            incrementWL ;
extern int            startWL ;
extern int            endWL ;
extern int            Xsize ;
extern int            Ysize ;


/* State variables, to determine what the current user selection is
 * Most are declared in the spect-math.c file  */


extern int            CUR_OBSERVER ;        /* global observer state  */
extern int            CUR_ILLUMINANT ;      /* global illuminant state  */
extern int            CUR_ILLUMINANT2 ;     /* global illuminant state  */
extern int            CUR_AMBIENT ;         /* global ambient state */
extern int            CUR_OBJECT ;          /* current object (3D-model) */
extern int            CUR_ADAPT ;           /* current adaptation model */
extern int            INIT ;                /* have the materials been initialized? */

/* variables to determine the illuminant intensity for both the ambient
 * and local illumination. First declared in spect-cb.c
 */

extern GLfloat        AmbScale, DiffScale, DiffScale2 ;

extern XtTranslations trans  ;
```

spectral.c

```
/*
 * spectral.c
 *
 * Written By: Garrett M. Johnson
 *
 * main file for the full spectral rendering package
 * using OpenGL, and Motif.
 *
 * this file contains the main function for the spectral rendering
 * package. once the program is launched, this function is called.
 * this function sets up all necessary parameters, calls the widget
 * building functions, and then sacrifices control to the event loop.
 * eventually all control relinquished.
 *
 * Last revised: 07/27/98
 *
 */

#include <Xm/Xm.h>          /* we need this, to declare our widgets */
#include <stdio.h>
#include <sys/param.h>
#include <GL/glx.h>         /* this is necessary to declare the glx stuff */
#include <GL/gl.h>          /* for all the good OpenGL calls */
#include <GL/glu.h>         /* all glu's are good glu's will gary glu. */
#include "spectral.h"       /* the ever important spectral header file */

/* first we declare all the widgets necessary to build the cool user
 * interface. see the spectral.h and spect-init.c files for more details
 * on what each widget actually does.
 */
Widget main_w ;
Widget menubar ;
Widget widget ;
Widget form ;
Widget  frame ;
Widget  glxarea ;
Widget  file_menu ;
Widget ob_menu ;
Widget am_menu ;
Widget li_menu ;
Widget li_menu2 ;
Widget det_menu ;
Widget adapt_menu ;
Widget ambient_slide ;
Widget local_slide ;
Widget local2_slide ;
Widget popup ;
Widget filebox ;
Widget help_dialog ;
Widget plot_dialog ;
Widget plot_widget ;


/* more widgets, for the SaveAs dialog ... now that's allota widgets... */

Widget dialog;
Widget save_form;
Widget increment_frame ;
Widget increment_form ;
Widget wl_label;
Widget toggle_box ;
Widget wl_frame ;
Widget wl_form ;
Widget begin_wl ;
Widget end_wl ;
Widget size_frame ;
Widget size_form ;
Widget size_form2;
```

```
Widget size_form3;
Widget size_label ;
Widget x_label ;
Widget y_label ;
Widget x_text ;
Widget y_text ;
Widget save_filebox ;


XtAppContext    app ;                /* the ever important app context */
Display         *display ;           /* declare the display variable */
Window           glxwin ;            /* the glx window used to draw in */
GLXContext       glxcx ;             /* an then the glx context */
XVisualInfo      *vi = NULL ;              /* our visual info id. */
Dimension       viewHeight, viewWidth ;
Visual           *overlayVisual = NULL ; /* do we have overlay planes? */
int              overlayDepth ;             /* if so, what is the bit depth */
Colormap         overlayColormap ;   /* we need a colormap for said plane */
Bool doubleBuffer = True ;           /* display double buffered? */


static String fallbackResources[] = {
                 "*sgiMode: true",     /* Try to enable Indigo Magic look & feel */
                 "*useSchemes: all",   /* and SGI schemes. */
                 "*title: Spectral Viewer",
/*
                 "*filebox_popup*title: Open datafile...",
*/
                 "*glxarea*width: 800",
                 "*glxarea*height: 600",
                NULL
                };

/* we need an actions table, to determine where a mouse button was pressed */
static XtActionsRec actionsTable[] = {
       {"button_down", button_down},       /* record only if middle button */
       }  ;

/* main()
 *
 * here it is, the old main function. all sorts of anciified at that.
 * this function reads in the command line file, if offered one, and
 * then works on initializing everything it can possibly find. this
 * includes the gui stuff, as well as the material files.
 *
 * long live the main function, for it is the king.
 */
int
main(int argc, char **argv)
{

    FILE             *filenames ;          /* command line file name */
    char             names[24][120] ;      /* array holding material files */
    char             *name_array[24] ;     /* slick c-wizardry name array */
    Widget           toplevel ;            /* the old toplevel widget */
    int              glxtest = 0 ;         /* glx test variable */
    int              i ;                   /* sweet array counter */



    /* lets start by initializing the toplevel widget.
     * this gets the old ball rolling for our user interface.
     */
    toplevel = XtAppInitialize (&app, "Spectral",
              NULL, 0,
              &argc, argv,
              fallbackResources,
                NULL, 0);

    /* add the actions table, to record mouse events */
```

```
    XtAppAddActions(app, actionsTable, XtNumber(actionsTable) ) ;

    /* get the display variable from the toplevel widget */
    display = XtDisplay(toplevel) ;

    /* initialize the glx server stuff, using the glx_init() function
     * found in the spect-glx-init.c file.
     */
    glxtest = glx_init() ;

    /* let us know that the display is double buffered */
    if(doubleBuffer) fprintf(stderr, "the display is double buffered\n") ;

    /* if glx_init does not work right, it returns a 1, letting us know */
    if(glxtest == 1)
       XtAppError(app, "could not create rendering context") ;

    /* check if we have overlay planes. detectOverlaySupport() found in
     * spect-glx-init.c */
    detectOverlaySupport(display) ;

    /* initialize the main window for the gui. (spect-init.c) */
    mainwindow_init(toplevel) ;

    /* initialize the pop-up dialogs as well */
    dialog_init(toplevel) ;

    /* make the current glx context the current one */
    glXMakeCurrent(display, XtWindow(glxarea), glxcx);


    /* once we have set up gui stuff, we need to go one with the
     * rest of our life. */

    /* first set the name array so that it contains the addresses of
     * each element in the 2-D array.
     */
    for(i=0; i<24; i++){
        name_array[i] = &(names[i][0]) ;
    }


    /* now check to see if any command line file names were given. */
    if (argc == 2){
       /* apparently so, so initialize the material names file */
         filenames = initialize_datafile(argv[1])  ;

       /* get the file names using the read_file() function */
         read_files(name_array, &filenames) ;

       /* initialize the material files */
       initialize_materials(name_array) ;

       /* initialize the translation table */
       init_translation() ;
    }

    /* let the user know that there are overlay planes */
    if(overlayVisual)fprintf(stderr, "in main, i have overlay visual\n") ;

    /* here we go...time to sacrifice control over to X, using the main loop */
    XtAppMainLoop(app) ;

    return(0) ;
}
```

spect-read.c

```
/*
 * spect-read.c
 *
 * Written By: Garrett M. Johnson
 *
 *
 * This file contains all the functions used for reading the
 * material file names, as well as the material files themselves.
 * Essentially this file contains my somewhat limited version of
 * a file parser. Luckily, I am very strict as to the exact format
 * of the material files read, so my limited parsing skills do
 * a fine, if not downright good, job.
 *
 * Last revised: 07/27/98
 *
 */

#include <stdio.h>
#include <math.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include "spectral.h"
#include "gltx.h"

#define TRUE  1
#define FALSE 0

#if !defined(GL_VERSION_1_1) && !defined(GL_VERSION_1_2)
#define glBindTexture glBindTextureEXT
#endif

/* start with the global variables...most have been declared in the header */

int     INIT = FALSE ;              /* have we initialized materials? */


Bool    METAMERIC[24] ;             /* objects are metameric, true or false? */
Bool    FLUORESCENT[24] ;     /* objects fluoresce, true or false? */
Bool    SPECULAR[24] ;        /* objects have spectral gloss, true or false? */
Bool    TEXTURE[24] ;         /* objects are texture mapped, true or false? */
Bool    MET_FLUORESCENT[24] ; /* metameric objects fluoresce, true or false? */
Bool    MET_SPECULAR[24] ;    /* metameric objects specular.., true or false? */

double reflect[24][93] ;      /* global diffuse reflection */
double excite[24][93] ;       /* global excitation spectra */
double emit[24][93] ;         /* global emission spectra */
double specular[24][93] ;     /* global specular spectra */
double metamer[24][93] ;      /* global metameric diffuse reflection */
double met_excite[24][93] ;   /* metameric excitation spectra */
double met_emit[24][93] ;     /* metameric emission spectra */
double met_specular[24][93] ;       /* metameric specular spectra */
double  gloss[24] ;           /* global gloss factor (0-128) */
double  met_gloss[24] ;             /* global metameric gloss factor */
int     NumTex[24] ;          /* number of a given texture file (position) */

int    NUM_FILES = 0 ;              /* total number of different material files */
int    TOTAL_TEXTURE = 0 ;    /* total number of texture files */


/* read_material()
 *
 * this function reads the material information from a given
 * material file. it is passed an initiallized material file
 * as well as a MATERIALS structure to hold the information
 * read in. a metameric MATERIAL structure is also passed in
 * on the remote chance that the given material has a metameric
```

```
 * match. [ in future versions, this will probably be eliminated,
 * as the only real use is in the spectral viewer...]
 *
 */

void
read_material(
       FILE **file,
       MATERIALS *data,
       MATERIALS *met )
{
    char      data_in[400] ;         /* buffer to a given line */
    char      temp[20] ;             /* a temp var. to hold material type info */
    char      temp_name[400] ;       /* temp texture name file */
    int       wavelength[93] ;       /* array to hold wavelength info */
    int       i = 0, m = 0, test ;          /* array counters */
    int       s = 0, f = 0 ;         /* array counters */
    int       ms = 0, mf = 0 ;       /* metameric array counters */

    /* initialize all arrays in the structure to 0.0 */
    for(i=0;i<93;i++){
       met->diffuse[i] = 0.0 ;
       met->spec[i] = 0.0 ;
       met->exc[i] = 0.0 ;
       met->em[i] = 0.0 ;
       data->spec[i] = 0.0 ;
       data->exc[i] = 0.0 ;
       data->em[i] = 0.0 ;
    }

    i = 0 ;

    data->shininess = 0.0 ;  /* set the initial gloss to be 0 */
    data->Fluor = FALSE ;    /* not fluorescent until specifically told */
    data->Spec = FALSE ;     /* no specular spectra unless told */
    data->Metameric = FALSE ;       /* not metameric until told */
    data->Texture = FALSE ;  /* has no defined texture file */
    data->numTex = 100  ;    /* set texture position to absurd number */
    met->shininess = 0.0 ;   /* set the metameric gloss to 0.0 */
    met->Fluor = FALSE ;     /* metameric object does not fluoresce */
    met->Spec = FALSE ;             /* metameric object has no specular spectra */

    /* a big old while loop reads in all the information...
     * basically until fgets returns EOF.
     */
    while( fgets(data_in, 400, *file) ){
       switch(data_in[0]) {  /* first character of line controls the switch */

            case '#':  ;         /* a comment, ignore the line  */
               break  ;

            case 'g':  ;      /* the glossiness of the sample  */

               sscanf(data_in, "%s %lf", temp, &(data->shininess) ) ;
               break  ;

            case 't':  ;      /* the name of the texture file, if any */

               data->Texture = TRUE ;       /* we have a texture */
               sscanf(data_in, "%s %s", temp, temp_name) ;
               data->texture = temp_name ; /* the name of the texture */
               data->numTex = TOTAL_TEXTURE + 1;
               TOTAL_TEXTURE++ ;     /* increase the total number of textures */
               break  ;

            case 'f':  ;         /* the sample is fluorescent  */

               if(!data->Fluor) data->Fluor = TRUE ; /* fluorescent to true */
               if( f<93 && (sscanf(data_in, "%s %d %lf %lf", temp, &(wavelength[f]),
                                    &(data->exc[f]), &(data->em[f])) ==4)){
                     f++ ;
                }
```

```
            break ;

        case 's':  ;      /* the sample has a specular component */

           if(!data->Spec) data->Spec = TRUE ;
           if( s<93 && (sscanf(data_in, "%s %d %lf ", temp,
                              &(wavelength[s]),
                              &(data->spec[s])) ==3)){
                s++ ;
            }
           break ;


        case 'm':  ;      /* the given sample has a metameric match */

           if(!data->Metameric) data->Metameric = TRUE ;

           /* if the material is metameric, we now read the second character */
           switch(data_in[1]){

                case 'r':  ;           /* the standard metameric diffuse   */

                   if( m<93 && (sscanf(data_in, "%s %d %lf ", temp,
                              &(wavelength[m]),
                                 &(met->diffuse[m])) ==3)){
                       m++ ;
                   }
                break ;

                case 'f':  ;          /* the metamerics sample fluoresces   */

                   if(!met->Fluor) met->Fluor = TRUE ;

                   if(mf<93 && (sscanf(data_in, "%s %d %lf %lf",
                             temp, &(wavelength[mf]),
                             &(met->exc[mf]), &(met->em[mf])) ==4)){
                       mf++ ;
                   }
                break ;

                case 's':  ;          /* the metamerics specular component   */

                   if(!met->Spec) met->Spec = TRUE ;

                       if( ms<93 && (sscanf(data_in, "%s %d %lf ", temp,
                                 &(wavelength[ms]),
                                 &(met->spec[ms])) ==3)){
                           ms++ ;
                       }
                  break ;

                case 'g':  ;          /* the gloss factor of the metamer   */

                   sscanf(data_in, "%s %lf", temp, &(met->shininess) )   ;
                break ;
           }
        /* if there is no spectra type (t, f, s, m) default to standard
         * diffuse reflectance spectra */

        default:
           if( i<93 && (sscanf(data_in, "%d %lf", &(wavelength[i]),
                             &(data->diffuse[i])) ==2)   ){
                i++ ;
           }
            break  ;
    }

}
/* close the file */
test = fclose(*file);
if(test !=  0)
   fprintf(stderr, "error closing file\n" ) ;
```

```
}

/* read_files()
 *
 * this function reads the material names file and
 * places the name into the names file. this function
 * is passed a pointer to a the material file names array,
 * and an initiallized file.
 *
 */

void
read_files(char **names, FILE **file)
{
    char      buf[400] ;            /* buffer to hold fgets info */
    char      temp_name[20] ;               /* temporary 1-D name array */
    int i = 0 ;                             /* array counter */

    /* another funky while loop that quits when it hits an EOF */
    while( fgets(buf, 400, *file) ){
       sscanf(buf, "%s", &temp_name)  ;     /* place file name in temp array */
       strcpy(names[i], temp_name) ;               /* copy name to real name array */
       i++  ;                               /* count number of files */
    }

    /* close the file */
    (void)fclose(*file) ;

    NUM_FILES = i ;
    fprintf(stderr, "number of files=%d\n", i) ;
}

/* initialize_datafile()
 *
 * this function is used to initialize any given file.
 *
 */

FILE *
initialize_datafile(char *name)
{
    FILE *datafile ;                /* the master file */

    /* open the file using fopen, set to read */
    if((datafile = fopen(name, "r")) == NULL){
       fprintf(stderr, "file: %s is not valid\n", name) ;
       exit(-1) ;
    }

    /* return the now initialized data file */
    return datafile ;
}

/*
 * initialize_materials()
 *
 * once we have all the material file names, we have to do something
 * with them. this file takes all the files, initializes them, and
 * then fills all the appropriate global variable arrays.
 */

void
initialize_materials(char **file_names)
{
    GLTXimage   *image ;            /* a texture image pointer */
    FILE       *material  ;         /* a single material file pointer */
    MATERIALS *temp_data ;          /* a temporary MATERIALS structure */
    MATERIALS *temp_metamer ;       /* another temp MATERIALS structure */
    int        i, j ;               /* some array counters */
```

```
glEnable(GL_TEXTURE_2D) ;              /* first enable texture mapping */

/* allocate the memory for the MATERIALS structures */
temp_data = (MATERIALS *)malloc(sizeof(MATERIALS)) ;
temp_metamer = (MATERIALS *)malloc(sizeof(MATERIALS)) ;

/* we start a for loop, repeating through the total number of files */
for(i=0;i<NUM_FILES;i++){
   /* set the material file using the file_name array*/
   material = initialize_datafile(file_names[i]) ;

   /* call read_material, using the material file, and the temp structures */
   read_material(&material, temp_data, temp_metamer ) ;

   /* set all our booleans equal to the temporary ones */
   METAMERIC[i] = temp_data->Metameric ;
   FLUORESCENT[i] = temp_data->Fluor ;
   SPECULAR[i] = temp_data->Spec ;
   TEXTURE[i] = temp_data->Texture ;
   MET_FLUORESCENT[i] = temp_metamer->Fluor ;
   MET_SPECULAR[i] = temp_metamer->Spec ;

   /* now set our global spectra equal to the temporary spectra.
    * this is why we spent all that time setting all the spectra
    * to zero in the read_materials function.
    */
   for(j=0;j<93;j++){
        reflect[i][j] = temp_data->diffuse[j] ;
        specular[i][j] = temp_data->spec[j] ;
        excite[i][j] = temp_data->exc[j] ;
        emit[i][j] = temp_data->em[j] ;
        metamer[i][j] = temp_metamer->diffuse[j] ;
        met_specular[i][j] = temp_metamer->spec[j] ;
        met_excite[i][j] = temp_metamer->exc[j] ;
        met_emit[i][j] = temp_metamer->em[j] ;
   }

   /* set the gloss components */
   gloss[i] = temp_data->shininess ;
   met_gloss[i] = temp_metamer->shininess ;

   /* if the temp structure contains a texture file, set that up */
   if(temp_data->Texture == TRUE){
        NumTex[i] = temp_data->numTex ;

        /* initialize the texture file using the gltx.c libraries */
        image = gltxReadRGB(temp_data->texture) ;

        /* if the texture file initialized properly, set the OpenGL texture
         * properties.
         */
        if(image != NULL){
             /* first bind the texture to the proper position number */
           glBindTexture(GL_TEXTURE_2D, (GLuint)NumTex[i]) ;

           /* set the texture parameters to repeat rather than clamp */
           glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
           glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

           /* now build a mipmap using the texture image data */
             gluBuild2DMipmaps(GL_TEXTURE_2D, image->components, image->width,
                     image->height,  GL_RGB,
                     GL_UNSIGNED_BYTE, image->data) ;

           /* delete the gltx image, once we are done with it */
            gltxDelete(image) ;
        }
        /* if there is no texture file, set the boolean to false */
        else{
           TEXTURE[i] = FALSE ;
        }
   }
```

```
        /* set the material file name to NULL, before we loop through again */
         material = NULL ;
      }

      /* we have initialized all the global material spectra. */

      INIT = TRUE ;
}
```

spect-init.c

```
/*
 * spect-init.c
 *
 * Written By: Garrett M. Johnson
 *
 * source file for the full spectral rendering package
 * using OpenGL, and Motif.
 *
 * this file contains all the initialization routines for
 * the X-window/Motif user interface. it contains the functions
 * to initialize the main-window, as well as the pop-up dialog
 * windows. all of the nasty motif code lies in this file.
 * this code is not quite platform independent, as it requires
 * the system to run X and have Motif. perhaps a port to NT or
 * any other platform will be made in the future (hint hint).
 *
 * Last revised: 07/27/98
 *
 */


#include <X11/Intrinsic.h>
#include <X11/Xcms.h>
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/LabelG.h>
#include <Xm/ToggleB.h>
#include <Xm/MessageB.h>
#include <Xm/DialogS.h>
#include <Xm/PanedW.h>
#include <Xm/FileSB.h>
#include <Xm/Form.h>
#include <Xm/Frame.h>
#include <Xm/Scale.h>
#include <Xm/Text.h>
#include <Xm/Label.h>
#include <GL/glx.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <X11/GLw/GLwMDrawA.h>
#include "spectral.h"
#include "SciPlot.h"


/*
 * mainwindow_init() contains everything needed to build the user
 * interface. the callbacks for the widgets lie in the spect-cb.c
 * file.
 */

void
mainwindow_init(Widget toplevel)
{
    XmString file, object, ambient, light1, light2, detector, adapt, help ;
    XmString open, save, quit;
    XmString torus, sphere, cow, ship, woman, tricer ;
    XmString d65, ill_a, f1, black, red ;
    XmString std2, std10, scan, photo, stdev;
    XmString none, vonkries, fairchild ;
    XmString plot ;
    Arg args[10] ;
    int n ;
```

```
/* and so it begins... */

/* main window contains a MenuBar and then several container widgets. */

main_w = XtVaCreateManagedWidget ("main_window",
          xmMainWindowWidgetClass,    toplevel,
          NULL);


/* create the string names for the menubar */

file = XmStringCreateLocalized ("File");
object = XmStringCreateLocalized ("Object");
ambient = XmStringCreateLocalized ("Ambient");
light1 = XmStringCreateLocalized ("Light 1");
light2 = XmStringCreateLocalized ("Light 2");
detector = XmStringCreateLocalized ("Detector");
adapt = XmStringCreateLocalized ("Chromatic Adaptation");
help = XmStringCreateLocalized ("Help");

/* Create a simple MenuBar that contains the user option menus */

menubar = XmVaCreateSimpleMenuBar (main_w, "menubar",
            XmVaCASCADEBUTTON, file, 'F',
            XmVaCASCADEBUTTON, object, 'O',
            XmVaCASCADEBUTTON, ambient, 'A',
            XmVaCASCADEBUTTON, light1, 'L',
            XmVaCASCADEBUTTON, light2, '2',
            XmVaCASCADEBUTTON, detector, 'D',
            XmVaCASCADEBUTTON, adapt, 'C',
            XmVaCASCADEBUTTON, help, 'H',
           NULL);


/* free the enslaved strings...long live the strings */

XmStringFree (file);
XmStringFree (object);
XmStringFree (ambient);
XmStringFree (light1);
XmStringFree (detector);
XmStringFree (adapt);


/* Tell the menubar which button is the help menu, so at least someone knows   */

if (widget = XtNameToWidget (menubar, "button_7")){
    XtVaSetValues (menubar,
           XmNmenuHelpWidget, widget, NULL);
}


/* First menu is the File menu -- callback is file_cb() */

/* first create the strings for the file menu */

open = XmStringCreateLocalized ("Open...");
save = XmStringCreateLocalized ("Save As...");
quit = XmStringCreateLocalized ("Quit");


/*
 * if we have an overlay visual, lets use that for the pull down
 * menu (so we don't have to redraw everytime a menu is used)
 */

if(overlayVisual){
     file_menu = XmVaCreateSimplePulldownMenu (menubar, "file_menu",
          0, file_cb,          /* position and callback    */
          XmVaPUSHBUTTON, open, 'O', NULL, NULL,
```

```
            XmVaSEPARATOR,
            XmVaPUSHBUTTON, save, 'S', NULL, NULL,
            XmVaSEPARATOR,
            XmVaPUSHBUTTON, quit, 'Q', NULL, NULL,
            XmNvisual, overlayVisual,
            XmNdepth, overlayDepth,
            XmNcolormap, overlayColormap,
            NULL);
}

/* if we don't have an overlay visual...*/
else{
     file_menu = XmVaCreateSimplePulldownMenu (menubar, "file_menu",
            0, file_cb,              /* position and callback   */
             XmVaPUSHBUTTON, open, 'O', NULL, NULL,
            XmVaSEPARATOR,
             XmVaPUSHBUTTON, save, 'S', NULL, NULL,
            XmVaSEPARATOR,
             XmVaPUSHBUTTON, quit, 'Q', NULL, NULL,
            NULL);
}


/* just incase the colormap for the overlay isn't installed...do that */

if(overlayVisual){
   XtAddCallback(XtParent(file_menu), XmNpopupCallback,
           colormapInstall, NULL);
}


/* free 'em up!!!!  */

XmStringFree (open);
XmStringFree (save);
XmStringFree (quit);


/* Second menu is the object menu -- callback is object_cb() */

torus = XmStringCreateLocalized ("Torus");
sphere = XmStringCreateLocalized ("Sphere");
cow = XmStringCreateLocalized ("Cow");
ship = XmStringCreateLocalized ("Ship");
woman = XmStringCreateLocalized ("Woman");
tricer = XmStringCreateLocalized ("Triceratops");

ob_menu = XmVaCreateSimplePulldownMenu (menubar, "object_menu",
          1, object_cb,         /* Position and callback function   */
          XmVaRADIOBUTTON, torus, 'T', NULL, NULL,
          XmVaRADIOBUTTON, sphere, 'S', NULL, NULL,
          XmVaRADIOBUTTON, cow, 'C', NULL, NULL,
          XmVaRADIOBUTTON, ship, 'p', NULL, NULL,
          XmVaRADIOBUTTON, woman, 'W', NULL, NULL,
          XmVaRADIOBUTTON, tricer, 'r', NULL, NULL,
          XmNradioBehavior, True,      /* RowColumn resources to enforce */
          XmNradioAlwaysOne, True,     /* radio behavior in Menu */
          NULL);


/* oh yeah...free them baby */

XmStringFree (torus);
XmStringFree (sphere);
XmStringFree (cow);
XmStringFree (ship);
XmStringFree (woman);
XmStringFree (tricer);


/* Initialize menu so that the torus is selected. */
```

```
if (widget = XtNameToWidget (ob_menu, "button_0")){
    XtVaSetValues (widget, XmNset, True, NULL) ;
}

/* now we are going to work on the ambient light menu...callback is ambient_cb() */

d65 = XmStringCreateLocalized ("D65") ;
ill_a = XmStringCreateLocalized ("Illuminant A") ;
f1 = XmStringCreateLocalized ("F1") ;
black = XmStringCreateLocalized ("Black") ;
red = XmStringCreateLocalized ("Red") ;
none = XmStringCreateLocalized ("None") ;

am_menu = XmVaCreateSimplePulldownMenu (menubar, "ambient_menu",
          2, ambient_cb,        /* Position and callback function   */
          XmVaRADIOBUTTON, d65, 'D', NULL, NULL,
          XmVaRADIOBUTTON, ill_a, 'A', NULL, NULL,
          XmVaRADIOBUTTON, f1, 'F', NULL, NULL,
          XmVaRADIOBUTTON, black, 'B', NULL, NULL,
          XmVaRADIOBUTTON, red, 'R', NULL, NULL,
          XmNradioBehavior, True,      /* RowColumn resources to enforce */
          XmNradioAlwaysOne, True,     /* radio behavior in Menu */
          NULL);

/* Initialize menu so that "D65" is selected. */

if (widget = XtNameToWidget (am_menu, "button_0")){
    XtVaSetValues (widget, XmNset, True, NULL) ;
}

/* since the local lights will have the same options...we don't need
 * to free the strings just yet. let's build the light1 menu. callback
 * is light_cb().
 */

li_menu = XmVaCreateSimplePulldownMenu (menubar, "light_menu",
          3, light_cb,          /* Position and callback function   */
          XmVaRADIOBUTTON, d65, 'D', NULL, NULL,
          XmVaRADIOBUTTON, ill_a, 'A', NULL, NULL,
          XmVaRADIOBUTTON, f1, 'F', NULL, NULL,
          XmVaRADIOBUTTON, black, 'B', NULL, NULL,
          XmVaRADIOBUTTON, red, 'R', NULL, NULL,
          XmNradioBehavior, True,      /* RowColumn resources to enforce */
          XmNradioAlwaysOne, True,     /* radio behavior in Menu */
          NULL);

/* Initialize menu so that "D65" is selected. */

if (widget = XtNameToWidget (li_menu, "button_0")){
    XtVaSetValues (widget, XmNset, True, NULL);
}

/* now lets build the second light menu...quite similar to light1
 * but with "None" as an option. callback is light2_cb().
 */

li_menu2 = XmVaCreateSimplePulldownMenu (menubar, "light_menu",
          4, light2_cb,         /* Position and callback function   */
          XmVaRADIOBUTTON, none, 'N', NULL, NULL,
          XmVaRADIOBUTTON, d65, 'D', NULL, NULL,
          XmVaRADIOBUTTON, ill_a, 'A', NULL, NULL,
          XmVaRADIOBUTTON, f1, 'F', NULL, NULL,
          XmVaRADIOBUTTON, black, 'B', NULL, NULL,
          XmNradioBehavior, True,      /* RowColumn resources to enforce */
          XmNradioAlwaysOne, True,     /* radio behavior in Menu */
          NULL);


/* Initialize menu so that "None" is selected. */

if (widget = XtNameToWidget (li_menu, "button_0")){
    XtVaSetValues (widget, XmNset, True, NULL);
```

```
}

/* now free the string labels for all the light source menus. */

XmStringFree (d65);
XmStringFree (ill_a);
XmStringFree (fl);
XmStringFree (black);
XmStringFree (red);
XmStringFree (none);

/* Create the string labels for the detector menu.  */

std2 = XmStringCreateLocalized ("2 Degree Observer");
std10 = XmStringCreateLocalized ("10 Degree Observer ");
stdev = XmStringCreateLocalized ("Standard Deviate Observer");
scan = XmStringCreateLocalized ("RGB Scanner");
photo = XmStringCreateLocalized ("Photograpic Film");


/* now create the simple detector menu, callback detect_cb(). */

det_menu = XmVaCreateSimplePulldownMenu (menubar, "detect_menu",
            5, detect_cb,        /* Position and callback function  */
          XmVaRADIOBUTTON, std2, 'D', NULL, NULL,
          XmVaRADIOBUTTON, std10, 'O'. NULL, NULL,
          XmVaRADIOBUTTON, stdev, 'S', NULL, NULL,
          XmVaRADIOBUTTON, scan, 'R', NULL, NULL,
          XmVaRADIOBUTTON, photo, 'P', NULL, NULL,
          XmNradioBehavior, True,      /* RowColumn resources to enforce */
          XmNradioAlwaysOne, True,     /* radio behavior in Menu */
          NULL);

/* Initialize menu so that "std2" is selected. */

if (widget = XtNameToWidget (det_menu, "button_0")){
    XtVaSetValues (widget, XmNset, True, NULL);
}


/* Free the detector string labels.  */

XmStringFree (std2);
XmStringFree (std10);
XmStringFree (stdev);
XmStringFree (scan);
XmStringFree (photo);

/* now we have to work on the chromatic adaptation menu..callback adapt_cb(). */

none = XmStringCreateLocalized ("None");
vonkries = XmStringCreateLocalized ("Von Kries");
fairchild = XmStringCreateLocalized ("Fairchild");

adapt_menu = XmVaCreateSimplePulldownMenu (menubar, "detect_menu",
            6, adapt_cb,         /* Position and callback function  */
          XmVaRADIOBUTTON, none, 'N', NULL, NULL,
          XmVaRADIOBUTTON, vonkries, 'V', NULL, NULL,
          XmVaRADIOBUTTON, fairchild, 'F', NULL, NULL,
          XmNradioBehavior, True,      /* RowColumn resources to enforce */
          XmNradioAlwaysOne, True,     /* radio behavior in Menu */
          NULL);


/* Initialize menu so that "none" is selected. */
if (widget = XtNameToWidget (adapt_menu, "button_0")){
    XtVaSetValues (widget, XmNset, True, NULL);
}


/* Free the detector string labels.  */
```

```
XmStringFree (none);
XmStringFree (vonkries);
XmStringFree (fairchild);


/* eighth menu is the help menu -- callback is help_cb() */

XmVaCreateSimplePulldownMenu (menubar, "help_menu",
          7, help_cb,
          XmVaPUSHBUTTON, help, 'H', NULL, NULL,
          NULL);

XmStringFree (help); /* we're done with it; now we can free it */


/* once the menubar and all the menus are created, we can managed the
 * menubar (and thusly all the children...if only life were this easy */

XtManageChild (menubar);


/* Create the form widget to place the sliders and the GLX frame in  */

form = XtVaCreateManagedWidget( "form",
          xmFormWidgetClass, main_w,
          XmNleftAttachment, XmATTACH_FORM,
          XmNrightAttachment, XmATTACH_FORM,
          XmNtopAttachment, XmATTACH_FORM,
          XmNbottomAttachment, XmATTACH_FORM,
          NULL ) ;

/* Create the local2 illuminant intensity slider, place on right side... */

local2_slide = XtVaCreateManagedWidget("local2_slide",
            xmScaleWidgetClass, form,
            XmNminimum, 0,
            XmNmaximum, 100,
            XmNvalue, 0,
            XmNtopOffset, 5,
            XmNbottomOffset, 5,
          XmNscaleWidth, 25,
          XmNhighlightOnEnter, TRUE,
            XmNtopAttachment, XmATTACH_FORM,
            XmNbottomAttachment, XmATTACH_FORM,
          XmNrightAttachment, XmATTACH_FORM,
            NULL ) ;


XtAddCallback(local2_slide, XmNdragCallback,
                 slide_cb, (XtPointer)LOCAL2) ;
XtAddCallback(local2_slide, XmNvalueChangedCallback,
                 slide_cb, (XtPointer)LOCAL2) ;

/* create the local1 illuminant intensity slider. place in middle. */
local_slide = XtVaCreateManagedWidget("local_slide",
          xmScaleWidgetClass, form,
          XmNminimum, 0,
          XmNmaximum, 100,
          XmNvalue, 100,
          XmNtopOffset, 5,
          XmNbottomOffset, 5,
          XmNscaleWidth, 25,
          XmNhighlightOnEnter, TRUE,
          XmNtopAttachment, XmATTACH_FORM,
          XmNbottomAttachment, XmATTACH_FORM,
          XmNrightAttachment, XmATTACH_WIDGET,
          XmNrightWidget, local2_slide,
          NULL ) ;

XtAddCallback(local_slide, XmNdragCallback,
                 slide_cb, (XtPointer)LOCAL) ;
XtAddCallback(local_slide, XmNvalueChangedCallback,
```

```
                        slide_cb, (XtPointer)LOCAL) ;

      /* Create the ambient light intensity slider, place next to local... */

      ambient_slide = XtVaCreateManagedWidget("ambient_slide",
                xmScaleWidgetClass, form,
                XmNminimum, 0,
                XmNmaximum, 100,
                XmNvalue, 0,
                XmNtopOffset, 5,
                XmNbottomOffset, 5,
                XmNscaleWidth, 25,
                XmNhighlightOnEnter, TRUE,
                XmNtopAttachment, XmATTACH_FORM,
                XmNbottomAttachment, XmATTACH_FORM,
                XmNrightAttachment, XmATTACH_WIDGET,
                XmNrightWidget, local_slide,
                NULL ) ;

      XtAddCallback(ambient_slide, XmNdragCallback,
                        slide_cb, (XtPointer)AMBIENT) ;
      XtAddCallback(ambient_slide, XmNvalueChangedCallback,
                        slide_cb, (XtPointer)AMBIENT) ;

      /* now create a frame, in which to place the GLX drawing area.
       * basically, just another container widget.
       */
      frame = XtVaCreateManagedWidget( "frame",
                xmFrameWidgetClass, form,
                XmNtopAttachment, XmATTACH_FORM,
                XmNbottomAttachment, XmATTACH_FORM,
                XmNleftAttachment, XmATTACH_FORM,
                XmNrightAttachment, XmATTACH_WIDGET,
                XmNrightWidget, ambient_slide,
                NULL ) ;

      /* place the GLX drawing area widget inside the entire frame. */

      glxarea = XtVaCreateManagedWidget( "glxarea",
                glwMDrawingAreaWidgetClass, frame,
                GLwNvisualInfo, vi,
                NULL) ;


      /* add the expose, initialize, and resize callbacks:
       * draw(), glxinit_cb(), resize() */

      XtAddCallback(glxarea, XmNexposeCallback, draw, NULL) ;
      XtAddCallback(glxarea, GLwNginitCallback, glxinit_cb, NULL) ;
      XtAddCallback(glxarea, XmNresizeCallback, resize, NULL) ;


      /* have you realized your toplevel yet? */

      XtRealizeWidget (toplevel);
}

/* ok, now we create a really cheesy message for the help dialog to
 * display...this should probably be changed to something witty...
 * but i would need to hire an outside writer for the material.
 */

#define MSG \
"Use the File->Open dialog to open a material file.\n\
this will display in the drawing area in the main window.\n\
Use the other menus to play with the viewing conditions.\n\
Use the sliders on the right to control the local\n\
and ambient illuminant intensity.\n\
Have a nice day."
/*
*/
```

```
/* dialog_init()
 *
 * in this function we initialize the many fun flavors of dialog
 * boxes, which are essential to make this software cool. ok
 * maybe the dialogs don't make it cool, but they are pretty
 * handy for viewing the spectra, and for opening and saving files.
 */

void
dialog_init(Widget toplevel)
{
    Arg       fileargs[2], helpargs[5] ;
    Arg       plotargs[5] ;
    Arg args[5];
    int       n = 0 ;
    XmString  dialogTitle ;
    XmString  filePattern ;
    XmString    msg ;
    XmString begin_wl_label = XmStringCreateLocalized ("Starting Wavelength") ;
    XmString end_wl_label = XmStringCreateLocalized ("Ending Wavelength") ;
    XmString wl_label_st = XmStringCreateLocalized ("Wavelength Increment") ;
    XmString size_label_st = XmStringCreateLocalized ("Image Size (Pixels):") ;
    XmString x_label_st = XmStringCreateLocalized ("Width:") ;
    XmString y_label_st = XmStringCreateLocalized ("Height:") ;
    XmString btn1, btn2, btn3 ;



    /* First Create the Filebox for Opening New Files  */

    filePattern = XmStringCreateSimple("* *") ;
    dialogTitle = XmStringCreateSimple("Open New File" ) ;

    filebox = XmCreateFileSelectionDialog (toplevel,
              "filebox",
              (ArgList)fileargs,
              sizeof(fileargs)/sizeof(Arg) )   ;

    XtAddCallback (filebox, XmNokCallback, open_datafile, NULL);
    XtAddCallback (filebox, XmNcancelCallback, XtUnmanageChild, NULL);

    /* yeah, that's right...free them! */

    XmStringFree(dialogTitle) ;
    XmStringFree(filePattern) ;


    /* Next Create the oh-so usefull Help Dialog  */

    /* define the msg to be that mess we wrote above */
    msg = XmStringCreateLtoR (MSG, XmFONTLIST_DEFAULT_TAG);

    XtSetArg (helpargs[n], XmNmessageString, msg); n++;

    /* once again, if we have overlay visuals...use 'em! */
    if(overlayVisual){
       XtSetArg(helpargs[n], XmNvisual, overlayVisual); n++ ;
       XtSetArg(helpargs[n], XmNdepth, overlayDepth); n++ ;
       XtSetArg(helpargs[n], XmNcolormap, overlayColormap); n++ ;
    }

    help_dialog = XmCreateInformationDialog (toplevel, "help_dialog",
             helpargs, n) ;

    XmStringFree(msg) ;

    /* Now create the Dialog for the Plotting Widgets  */

    n = 0 ;  /* set n = 0, for the old SetArg commands  */

    dialogTitle = XmStringCreateSimple("Radiance Plots") ;
```

```
/* set a fixed height for the plot widget at 480x360 pixels */

XtSetArg(plotargs[n], XtNheight, 360); n++ ;
XtSetArg(plotargs[n], XtNwidth, 480); n++ ;

/* create a simple information dialog to hold the plot widget */

plot_dialog = XmCreateInformationDialog(toplevel,"plot_dialog",
          plotargs, n ) ;


/* disable all the normal buttons, symbols, and messages from the
 * info dialog. keep the cancel button. */

XtUnmanageChild(XmMessageBoxGetChild(plot_dialog, XmDIALOG_OK_BUTTON) ) ;
XtUnmanageChild(XmMessageBoxGetChild(plot_dialog, XmDIALOG_HELP_BUTTON) ) ;
XtUnmanageChild(XmMessageBoxGetChild(plot_dialog, XmDIALOG_SYMBOL_LABEL) ) ;
XtUnmanageChild(XmMessageBoxGetChild(plot_dialog, XmDIALOG_MESSAGE_LABEL) ) ;


/* now create a plotting widget, using the SciPlot widget, written
 * by Robert W. McMullen. it is a pretty cool widget set, by the
 * way...to get your very own copy goto
 * http://www.ae.utexas.edu/~rwmcm
 */

plot_widget = XtVaCreateWidget("plot",
          sciplotWidgetClass, plot_dialog,
          XtNheight, 340,
          XtNwidth, 460,
          XtNxLabel, "Wavelength (nm)",
          XtNyLabel, "Radiance",
          XtNdrawMinor, False,
          XtNshowTitle, False,
          XtNdefaultMarkerSize, 2,
          NULL)  ;

/* try to force the wavelength scale to be between 300 and 760 */

SciPlotSetXUserScale(plot_widget, 300.0, 760.0) ;


/* ain't no such thing as a free string...until now */

XmStringFree(dialogTitle) ;



/* Next we have to create the Save Options Dialog.
 * this time, starting with a prompt-dialog box.
 */

XtSetArg (args[n], XmNautoUnmanage, False); n++;

/* create the prompt dialog */

dialog = XmCreatePromptDialog (toplevel, "prompt", args, n);

/* unmanage the silly stuff we don't need */
XtUnmanageChild(XmSelectionBoxGetChild(dialog, XmDIALOG_TEXT) ) ;
XtUnmanageChild(XmSelectionBoxGetChild(dialog, XmDIALOG_SELECTION_LABEL) ) ;
XtUnmanageChild(XmSelectionBoxGetChild(dialog, XmDIALOG_HELP_BUTTON) ) ;
XtUnmanageChild(XmSelectionBoxGetChild(dialog, XmDIALOG_APPLY_BUTTON) ) ;


/* add a callback for the ok button */
XtAddCallback (dialog, XmNokCallback, save_ok_cb, widget);

/* If the user selects cancel, just destroy the dialog */
XtAddCallback (dialog, XmNcancelCallback, XtDestroyWidget, NULL);
```

```
/* create a container form to place inside the prompt dialog */

save_form = XmCreateForm(dialog, "save_form", NULL, 0) ;

XtManageChild (save_form) ;

/* place a container frame inside the form */

increment_frame = XtVaCreateManagedWidget( "increment_frame",
                    xmFrameWidgetClass, save_form,
                    XmNtopAttachment, XmATTACH_FORM,
                    XmNleftAttachment, XmATTACH_FORM,
                   NULL ) ;


/* now place a form inside the frame (can you say ungodly
 * repetative...what can i say...i am a slave to good layouts
 */
increment_form = XtVaCreateManagedWidget( "increment_form",
                    xmFormWidgetClass, increment_frame,
                   NULL) ;

/* inside the form inside the frame inside the form inside the dialog
 * we place the radio buttons, so that the user might be able to
 * select the wavelength increments for the final spectral image.
 * phew!
 */
btn1 = XmStringCreateLocalized ("5 nm");
btn2 = XmStringCreateLocalized ("10 nm");
btn3 = XmStringCreateLocalized ("20 nm");

toggle_box = XmVaCreateSimpleRadioBox (increment_form,
                    "radio_box", 0,
                    save_toggle_cb,
                    XmVaRADIOBUTTON, btn1, 0, NULL, NULL,
                    XmVaRADIOBUTTON, btn2, 0, NULL, NULL,
                    XmVaRADIOBUTTON, btn3, 0, NULL, NULL,
                    XmNorientation, XmHORIZONTAL,
                    XmNtopAttachment, XmATTACH_FORM,
                    XmNleftAttachment, XmATTACH_FORM,
                    XmNrightAttachment, XmATTACH_FORM,
                   NULL);

/* manage your children, or i will! */
XtManageChild (toggle_box);

/* we must be free! */
XmStringFree (btn1) ;
XmStringFree (btn2) ;
XmStringFree (btn3) ;



/* ok, now we have to create a label widget, to tell everyone what
 * that fine radio box above actually does...
 */

wl_label = XtVaCreateManagedWidget( "label", xmLabelGadgetClass,
                    increment_form,
                    XmNlabelString, wl_label_st,
                    XmNtopAttachment, XmATTACH_WIDGET,
                    XmNtopWidget, toggle_box,
                    XmNbottomAttachment, XmATTACH_FORM,
                    XmNleftAttachment, XmATTACH_FORM,
                    XmNrightAttachment, XmATTACH_FORM,
                   NULL)  ;

XmStringFree(wl_label_st) ;


/* uh-oh...looks like we are back on the container frame...inside
```

```
 * the first form...and to the right of the increment frame.
 */

wl_frame = XtVaCreateManagedWidget( "wl_frame",
                    xmFrameWidgetClass, save_form,
                    XmNrightAttachment, XmATTACH_FORM,
                    XmNleftAttachment, XmATTACH_WIDGET,
                    XmNleftWidget, increment_frame,
                   NULL ) ;

/* where there is a frame, there must be a form, right? */

wl_form = XtVaCreateManagedWidget( "wl_form",
                    xmFormWidgetClass, wl_frame,
                   NULL) ;


/* create a slide widget, to control the starting wavelength
 * point. somewhere between 300 and 760.
 */

begin_wl = XtVaCreateManagedWidget( "begin_wl",
             xmScaleWidgetClass, wl_form,
             XmNtitleString, begin_wl_label,
            XmNminimum, 300,            /* minimum wavelength */
            XmNmaximum, 760,            /* minimum wavelength */
             XmNorientation, XmHORIZONTAL,
            XmNscaleMultiple, 5,               /* multiples of 5nm */
            XmNshowValue, TRUE,
            XmNvalue, 400,                     /* default to 400nm */
             XmNtopAttachment, XmATTACH_FORM,
             XmNrightAttachment, XmATTACH_FORM,
             XmNleftAttachment, XmATTACH_FORM,
            XmNhighlightOnEnter, True,
            XmNscaleHeight, 23,
            XmNscaleWidth, 250,
           NULL) ;

XmStringFree (begin_wl_label) ;

/* since the starting wavelength was a smashing success, let us
 * create another slider for the ending wavelength.
 */

end_wl = XtVaCreateManagedWidget( "end_wl",
            xmScaleWidgetClass, wl_form,
            XmNtitleString, end_wl_label,
           XmNminimum, 300,
           XmNmaximum, 760,
            XmNorientation, XmHORIZONTAL,
           XmNscaleMultiple, 5,
           XmNshowValue, TRUE,
           XmNvalue, 700,
            XmNtopAttachment, XmATTACH_WIDGET,
           XmNtopWidget, begin_wl,
            XmNrightAttachment, XmATTACH_FORM,
            XmNleftAttachment, XmATTACH_FORM,
            XmNbottomAttachment, XmATTACH_FORM,
           XmNhighlightOnEnter, True,
           XmNscaleHeight, 23,
           XmNscaleWidth, 250,
          NULL) ;

XmStringFree (end_wl_label) ;


/* once again we are back on this form-frame-form kick.
 * this frame is to hold the image dimensions info...
 * it is placed below the increment frame, and to the
 * left of the slider frame.
 */
size_frame = XtVaCreateManagedWidget( "size_frame",
```

```
                    xmFrameWidgetClass, save_form,
                    XmNtopAttachment, XmATTACH_WIDGET,
                    XmNtopWidget, increment_frame,
                    XmNleftAttachment, XmATTACH_FORM,
                    XmNrightAttachment, XmATTACH_WIDGET,
                    XmNrightWidget, wl_frame,
                    NULL ) ;


size_form = XtVaCreateManagedWidget( "size_form",
                    xmFormWidgetClass, size_frame,
                    NULL) ;


/* now create a label for our image size info to go into */

size_label = XtVaCreateManagedWidget( "size_label", xmLabelGadgetClass,
                    size_form,
                    XmNlabelString, size_label_st,
                    XmNtopAttachment, XmATTACH_FORM,
                    XmNleftAttachment, XmATTACH_FORM,
                    XmNrightAttachment, XmATTACH_FORM,
                    NULL)  ;

XmStringFree (size_label_st) ;


/* just when you thought there were too many forms...another
 * container form!
 */

size_form2 = XtVaCreateManagedWidget( "size_form3",
                    xmFormWidgetClass, size_form,
                    XmNfractionBase, 10,
                    XmNtopAttachment, XmATTACH_WIDGET,
                    XmNtopWidget, size_label,
                    XmNrightAttachment, XmATTACH_FORM,
                    XmNleftAttachment, XmATTACH_FORM,
                    NULL) ;

/* now we have to create a label for the x-dimension aka width */
x_label = XtVaCreateManagedWidget( "x_label",
                    xmLabelGadgetClass, size_form2,
                    XmNlabelString, x_label_st,
                    XmNtopAttachment, XmATTACH_FORM,
                    XmNleftAttachment, XmATTACH_FORM,
                    XmNbottomAttachment, XmATTACH_FORM,
                    XmNrightAttachment, XmATTACH_POSITION,
                    XmNrightPosition, 4,
                    XmNalignment, XmALIGNMENT_END,
                    NULL) ;

/* create a text box, in which the user can type the width in pixels */
x_text = XtVaCreateManagedWidget( "x_text",
                    xmTextWidgetClass, size_form2,
                    XmNtraversalOn, True,
                    XmNvalue, "800",           /* default at 800 pixels wide */
                    XmNtopAttachment, XmATTACH_FORM,
                    XmNrightAttachment, XmATTACH_FORM,
                    XmNleftAttachment, XmATTACH_POSITION,
                    XmNleftPosition, 5,
                    XmNbottomAttachment, XmATTACH_FORM,
                    NULL) ;


/* just when you thought you couldn't get enough forms. */
size_form3 = XtVaCreateManagedWidget( "size_form3",
                    xmFormWidgetClass, size_form,
                    XmNfractionBase, 10,
                    XmNtopAttachment, XmATTACH_WIDGET,
                    XmNtopWidget, size_form2,
                    XmNrightAttachment, XmATTACH_FORM,
```

```
                        XmNleftAttachment, XmATTACH_FORM,
                        XmNbottomAttachment, XmATTACH_FORM,
                        NULL) ;

    /* create the label for the y-dimension...otherwise known as height */
    y_label = XtVaCreateManagedWidget( "y_label",
                        xmLabelGadgetClass, size_form3,
                        XmNlabelString, y_label_st,
                        XmNtopAttachment, XmATTACH_FORM,
                        XmNleftAttachment, XmATTACH_FORM,
                        XmNbottomAttachment, XmATTACH_FORM,
                        XmNrightAttachment, XmATTACH_POSITION,
                        XmNrightPosition, 4,
                        XmNalignment, XmALIGNMENT_END,
                        NULL) ;

    /* create a textbox widget to hold the users inputed height */
    y_text = XtVaCreateManagedWidget( "y_text",
                        xmTextWidgetClass, size_form3,
                        XmNtraversalOn, True,
                        XmNvalue, "600",
                        XmNtopAttachment, XmATTACH_FORM,
                        XmNrightAttachment, XmATTACH_FORM,
                        XmNbottomAttachment, XmATTACH_FORM,
                        XmNleftAttachment, XmATTACH_POSITION,
                        XmNleftPosition, 5,
                        NULL) ;

    /* free said labels */
    XmStringFree (x_label_st) ;
    XmStringFree (y_label_st) ;

/* wow, after all that we have the tiny little save-options dialog */


    /* now Create the Filebox for the save as option  */

    filePattern = XmStringCreateSimple("*.hdf") ;
    dialogTitle = XmStringCreateSimple("Save As" ) ;

    save_filebox = XmCreateFileSelectionDialog (toplevel,
              "filebox",
              (ArgList)fileargs,
              sizeof(fileargs)/sizeof(Arg) )   ;

    XtAddCallback (save_filebox, XmNokCallback, save_as_cb, NULL);
    XtAddCallback (save_filebox, XmNcancelCallback, XtUnmanageChild, NULL);

    /* last time, free those babies! */
    XmStringFree(dialogTitle) ;
    XmStringFree(filePattern) ;

}
```

spect-gl.c

```
/*
 * spect-gl.c
 *
 * Written By: Garrett M. Johnson
 *
 * OpenGL glx file for the full spectral rendering package
 * using OpenGL, and Motif.
 *
 * This file contains all the code for initializing the glx
 * stuff. Thus, this file is pretty darn platform dependant.
 * Basically the functions in this file just determine if
 * we have overlay planes, and if so enable them and their
 * fun loving colormap.
 */
#include <stdlib.h>
#include <stdio.h>
#include <GL/glx.h>
#include "spectral.h"
#include "sovLayerUtil.h"

/* this is where we place all the GLX window information. basically
 * when we initialize the GLXserver goodies, we make sure the frame
 * buffer is at least configured as so. GLX first attempts to meet
 * the minimum requirements, and then tries to improve upon those.
 * notice that the red, green, and blue buffers are at least 8 bits.
 * we need at least "true-color", since everyone involved with color
 * should have that. Note also that we are not limited to just 8 bits
 * per color channel. If your RealityMonster happens to support 12
 * bits, more power to you.
 */
int config[] = {
              None, None,           /* Space for multisampling GLX
                                       attributes if supported. */
              GLX_DOUBLEBUFFER, GLX_RGBA, GLX_DEPTH_SIZE, 12,
              GLX_RED_SIZE, 8, GLX_GREEN_SIZE, 8, GLX_BLUE_SIZE, 8,
              None
              };

/* we start the doublebuffered config at the third position
 * in the config array
 */
int *dblBuf = &config[2];

/* we start the singlebuffered config at the fourth position...*/
int *snglBuf = &config[3];

/* detectOverlaySupport()
 *
 * this function determines if the display is capable of overlay
 * planes. these planes come in handy with the pull down menus
 * of the GUI, since we would not have to redraw all the information
 * that would be normally be covered. This function is borrowed
 * from Mark Kilgard's Programming for the X Window System, Addison-
 * Wesley, (1996).
 *
 */
void
detectOverlaySupport(Display *dpy)
{
    sovVisualInfo template, *overlayVisuals;
    int layer, nVisuals, i, entries;
```

```
    /* Need more than two colormap entries for reasonable menus. */
    entries = 2;
    for (layer = 1; layer <= 3; layer++) {
        template.layer = layer;
        template.vinfo.screen = DefaultScreen(dpy);
        overlayVisuals = sovGetVisualInfo(dpy,
              VisualScreenMask | VisualLayerMask,
              &template, &nVisuals);

        if (overlayVisuals) {
            for (i = 0; i < nVisuals; i++) {
                if (overlayVisuals[i].vinfo.visual->map_entries > entries) {
                    overlayVisual = overlayVisuals[i].vinfo.visual;
                    overlayDepth = overlayVisuals[i].vinfo.depth;
                    entries = overlayVisual->map_entries;
                }
            }
            XFree(overlayVisuals);
        }
    }
    if (overlayVisual)
        overlayColormap = XCreateColormap(dpy, DefaultRootWindow(dpy),
                                          overlayVisual, AllocNone);
}


/* isSupportedByGLX()
 *
 * this function, also borrowed from Mark J. Kilgard queries the
 * GLX server, to determine what is actually supported, so we
 * don't overshoot the capabilities of the computer.
 *
 */
static int
isSupportedByGLX(Display * dpy, char *extension)
{
#if defined(GLX_VERSION_1_1)
    static const char *extensions = NULL;
    const char *start;
    char *where, *terminator;
    int major, minor;

    glXQueryVersion(dpy, &major, &minor);


    /* Be careful not to call glXQueryExtensionsString if it
     looks like the server doesn't support GLX 1.1.
     Unfortunately, the original GLX 1.0 didn't have the notion
     of GLX extensions. */

    if ((major == 1 && minor >= 1) || (major > 1)) {
        if (!extensions)
            extensions = glXQueryExtensionsString(dpy, DefaultScreen(dpy));

    /* It takes a bit of care to be fool-proof about parsing
       the GLX extensions string.  Don't be fooled by
       sub-strings,  etc. */

        start = extensions;
        for (;;) {
            where = strstr(start, extension);
            if (!where)
                return 0;
            terminator = where + strlen(extension);
            if (where == start || *(where - 1) == ' ') {
                if (*terminator == ' ' || *terminator == '\0') {
                    return 1;
                }
            }
            start = terminator;
        }
    }
```

```
#else

  /* No GLX extensions before GLX 1.1 */
#endif
  return 0;
}

/* glx_init()
 *
 * this function initializes our visuals, using the coolest
 * feature set that we are capable of using. once the visual
 * is configured, we use that to create a GLX rendering
 * context. Once again this function is a creative, or
 * perhaps not so creative, borrowing from Mark J. Kilgard.
 */
int
glx_init(void)
{

#if defined(GLX_VERSION_1_1) && defined(GLX_SGIS_multisample)

     /* if the GLX_SGIS multisampling is defined, by all means,
      * lets try and use it.
      */
     if (isSupportedByGLX(display, "GLX_SGIS_multisample")) {
          config[0] = GLX_SAMPLES_SGIS;
          config[1] = 4;
          vi = glXChooseVisual(display, DefaultScreen(display), config);
     }

#endif

     /* if we don't have a visual yet, first try and get a double
      * buffered one.
      */
     if (vi == NULL) {
          /* Find an OpenGL-capable RGB visual with depth buffer. */
          vi = glXChooseVisual(display, DefaultScreen(display), dblBuf);

        /* if that failed, now lets try and get a single buffered visual */
        if (vi == NULL) {
              vi = glXChooseVisual(display, DefaultScreen(display), snglBuf);

             /* if that failed, well then...we might just want to give
              * up and get ourselves a paper route so that we might
              * be able to save up and get a better graphics system
              */
             if (vi == NULL)
               return(1) ;

             /* set the global doubleBuffer boolean, for future reference */
             doubleBuffer = False;
        }
     }


   /* Create an OpenGL rendering context. */

     glxcx = glXCreateContext(display, vi,
                          None, True);

     /* determine if there are overlay planes */
     detectOverlaySupport(display) ;

     /* make sure that we do have a rendering context */
     if (glxcx == NULL)
        return(1) ;
     else
        return(0) ;
}
```

spect-cb.c

```
/*
 * spect-cb.c
 *
 * Written By: Garrett M. Johnson
 *
 * source file for the full spectral rendering package
 * using OpenGL, and Motif.
 *
 * this file contains all the callback routines for the
 * interactive spectral viewer. once the program is launched,
 * it is the functions in this file that handle most of the
 * work.
 *
 * Last revised: 07/27/98
 *
 */


#include <X11/Intrinsic.h>
#include <X11/Xcms.h>
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/Label.h>
#include <Xm/ToggleB.h>
#include <Xm/MessageB.h>
#include <Xm/FileSB.h>
#include <Xm/Form.h>
#include <Xm/Frame.h>
#include <Xm/Scale.h>
#include <GL/glx.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <X11/GLw/GLwMDrawA.h>
#include "spectral.h"
#include "SciPlot.h"

/* declare the window width and height, for resizing purposes */

static GLfloat winWidth, winHeight ;
static int            redisplayPending = 0 ;
static XtWorkProcId   redisplayID ;

XtTranslations        trans = NULL ;

/* declare the semi-global, perhaps hemispherical, variables
 * used in the save-as functions
 */
int                   incrementWL ;
int                   startWL ;
int                   endWL ;
int                   Xsize ;
int                   Ysize ;


/*
 * handleRedisplay()
 *
 * a simple function designed to redraw the screen after
 * changes are made (eg the viewing conditions are changed,
 * the window was covered and re-exposed, or the window was
 * resized.
 */
```

```
Boolean
handleRedisplay(XtPointer closure)
{
     draw(GL_RENDER) ;
     redisplayPending = 0 ;

     return TRUE ;
}

/*
 * postRedisplay()
 *
 * this function actually handles all the redisplay commands,
 * but first makes sure that all is well with X.
 *
 */
void
postRedisplay(void)
{
     if(!redisplayPending){
        redisplayID = XtAppAddWorkProc(app, handleRedisplay, 0) ;
        redisplayPending = 1 ;
     }
}



/*
 * glxinit_cb()
 *
 * this function is the initialize callback function of the
 * glx drawing area. it is called when the widget is first
 * created. it defaults to a 600x800 dimension, by calling the
 * reshape() function.
 * it also calls the gl_init() function, found in spect-gl.c,
 * to initialize the OpenGL calls.
 *
 */
void
glxinit_cb(Widget w, XtPointer clientData, XtPointer callData)
{
     glxwin = XtWindow(w);
     glXMakeCurrent(XtDisplay(w), XtWindow(w), glxcx);
     reshape(800,600) ;
     gl_init() ;
}


/*
 * quit()
 *
 * a simple dummy function that is called when the quit
 * option is selected from the file menu. simply exits
 * the program.
 *
 */
void
quit(void)
{
     exit(0) ;
}


/*
 * open_cb()
 *
 * this is the callback function called when the user selects
 * the "Open File" option from the File Menu. this function
 * simple manages the filebox, and pops it up.
 *
 */
void
```

```
open_cb()
{
     XtManageChild(filebox) ;
     XtPopup (XtParent(filebox), XtGrabNone);
}


/*
 * create a glxarea translations table, to determine when the
 * middle mouse button is pressed, inside the glxarea.
 */
static char *glxareaTranslations =
                        "#override\n\
                        <Btn2Down>:button_down()\n"    ;

/*
 * init_translation()
 *
 * this function initiallizes the translations table used to
 * determine when the mouse button is clicked inside the glx
 * drawing area widget. this will be used to determine where
 * the button was clicked, and then to determine the spectra
 * of the object at that position.
 *
 */
void
init_translation(void)
{

     if (trans == NULL){
        trans = XtParseTranslationTable(glxareaTranslations) ;
        XtOverrideTranslations(glxarea, trans) ;
     }
}


/*
 * button_down()
 *
 * this function is called when the middle mouse button (Btn2) is
 * clicked inside the glx drawing area, as called by the
 * glxareaTranslation table. essentially this function determines
 * the pointer position, inside the drawing area, and calls the
 * pickScene() function with the pointer coordinates.
 *
 */
void
button_down(Widget w, XEvent *event, String *params, Cardinal *num_params)
{
     int x, y ;

     x = event->xbutton.x ;
     y = event->xbutton.y ;

     pickScene(x, y) ;

}


/*
 * open_datafile()
 *
 * this function is called when a file name is selected from the
 * openFile filebox (can you use the word file more often in a
 * sentence?). This function reads the file name, and then
 * calls the necessary sequence of events to read the data
 * files. long live the data files.
 *
 */
void
open_datafile(Widget w, XtPointer clientData, XtPointer callData)
{
```

```
FILE                *filenames ;
char                names[24][120] ;
char                *name_array[24] ;
char                *file = NULL ;
int                 i ;

XmFileSelectionBoxCallbackStruct       *cbs =
   (XmFileSelectionBoxCallbackStruct *)callData ;



/* read the filebox selection using the get string Motif command.
 * if such a selection doesn't exist...get out of this function.
 */
if (cbs) {
     if (!XmStringGetLtoR (cbs->value, XmFONTLIST_DEFAULT_TAG, &file))
         return; /* internal error */
}


/* do some fancy schmancy array nonsense to stick the address of
 * the arrays which will hold the file names into a single array.
 * basically this just makes my life easier, so i can pass a single
 * array into the next function.
 */
for(i=0; i<24; i++){
     name_array[i] = &(names[i][0]) ;
}

/* let's initiallize the datafile, using the old initialize_datafile()
 * function found in spect-read.c
 */
filenames = initialize_datafile(file) ;
/* don't forget to free that data! */
XtFree (file); /* free allocated data from XmStringGetLtoR() */

/* since our first datafile basically holds the names of many other
 * data files, lets get those names now, using the read_files function
 * found in spect-read.c
 */
read_files(name_array, &filenames) ;

/* once we have our actual material file names, lets initialize those
 * files using the initialize_materials() function found in spect-read.c
 */
initialize_materials(name_array) ;

/* if all is well and good, we no longer need that pesky filebox, so
 * unmanage it, and it just goes away.
 */
XtUnmanageChild(filebox) ;

/* we better initialize our translations table, so that our user can
 * select anything their heart desires...including magic?
 */
init_translation() ;

/*
 * well, it looks like we have loaded some new material files into our
 * program. we had best tell the OpenGL draw() function, found in
 * spect-gl.c, that it needs to recalculate all the colors.
 */
RECALCULATE_COLORS = TRUE ;

/* now we simply postRedisplay, or tell the display to redisplay,
 * and all is good.
 */
postRedisplay() ;

}
```

```
/*
 * save_cb()
 *
 * this is the call back function which is activated when the user
 * selects the "SaveAs" option from the file menu. essentially, it
 * manages the save_options dialog, and then pops it open.
 *
 */
void
save_cb(void)
{
     XtManageChild(dialog) ;
     XtPopup (XtParent(dialog), XtGrabNone) ;
}



/*
 * save_ok_cb()
 *
 * this is the callback function for the save options dialog widget.
 * it is called when the user selects the ok button from the
 * save options widget.
 *
 */
void
save_ok_cb(Widget w, XtPointer clientData, XtPointer callData)
{
     char *text ;

     /* get the values from the x-dimensions (width) text box */
     XtVaGetValues (x_text, XmNvalue, &text, NULL) ;

     /* convert the string into and integer */
     Xsize = atoi(text ) ;

     /* get the values from the y-dimensions (height) text box */
     XtVaGetValues (y_text, XmNvalue, &text, NULL) ;

     /* convert the string into and integer */
     Ysize = atoi(text ) ;

     /* now get the values from the starting and finishing wavelegth
      * slider bars
      */
     XtVaGetValues (begin_wl, XmNvalue, &startWL, NULL) ;
     XtVaGetValues (end_wl, XmNvalue, &endWL, NULL) ;

     /* we need to figure out if the starting wavelength is divisble by
      * five, so we will use the % (mod) command, to see if there is a
      * remainder. if there is a remainder, we will subtract that from
      * the starting wavelength, to get one divisible by 5. eg. if the
      * user selects 363nm as the starting wavelength, 363 % 5 = 3, so
      * starting wavelength = 363   (3603 % 5) = 360.
      */
    if(startWL % 5 != 0){
        startWL = startWL - (startWL % 5) ;
    }

    /* we need to do a similar thing with the ending wavelength as well.
     * except this time we need to go in the other direction, so if the
     * user selects 702, the actual wavelength used would be 705nm.
     */
    if(endWL % 5 != 0){
        endWL = endWL + 5 - (endWL % 5) ;
    }

    /* once we have all the info we need to know, unmanage the save_options dialog */
    XtUnmanageChild(dialog) ;

    /* now manage the save as filebox, so the user can choose an output file name.
     */
```

```
	XtManageChild(save_filebox) ;

	/* popup said filebox, while we are at it. */
	XtPopup (XtParent(save_filebox), XtGrabNone) ;

}



/*
 * save_toggle_cb()
 *
 * this is the function called when the user toggles the wavelength
 * increment radio buttons, on the save options dialog. essentially
 * just sets the global incrementWL variable to 5, 10 or 20, based
 * on the selection.
 *
 */
void
save_toggle_cb(Widget w, XtPointer clientData, XtPointer callData)
{
	int button = (int)clientData ;

	if(button == 0){
		incrementWL = 5 ;
	}
	else if(button == 1){
		incrementWL = 10 ;
	}
	else{
		incrementWL = 20 ;
	}
}

/*
 * save_as_cb()
 *
 * this is the callback function attached to the ok button on
 * the save_as filebox.
 *
 */
void
save_as_cb(Widget w, XtPointer clientData, XtPointer callData)
{
	char *file = NULL ;

	XmFileSelectionBoxCallbackStruct	*cbs =
		(XmFileSelectionBoxCallbackStruct *)callData ;

	/* read the file name */
	if (cbs) {
		if (!XmStringGetLtoR (cbs->value, XmFONTLIST_DEFAULT_TAG, &file))
		    return; /* internal error */
	}


	/* once we have the file name, kill the filebox! death to filebox. */
	XtUnmanageChild(save_filebox) ;


	/* ok, use all the knowledge we have to write out the full spectral
	 * image. the writeImage() function is located in the spect-write.c
	 * file.
	 */
	writeImage(Xsize, Ysize, incrementWL, startWL, endWL, file) ;
}


/*
 * file_cb()
 *
 * this is the callback associated with the pull-down file menu.
```

```
 * essentially it figures out which selection you chose, and then
 * calls the corresponding function.
 *
 */
void
file_cb(Widget w, XtPointer clientData, XtPointer callData)
{
     int choice = (int)clientData  ;

     /* choice is the number of the selected menu item...
      * if you recall open was the first option, then save as,
      * and finally quit...the rest is simple
      */
     switch(choice){
        case(0):
            open_cb() ;
            break ;

        case(1):
            save_cb() ;
            break ;

        case(2):
            quit() ;
            break ;
     }

}


/*
 * object_cb
 *
 * this is the callback associated with the object menu. it basically just
 * sets the global CUR_OBJECT to the same number as the selection. piece
 * of cake, since we ennumerated all the objects way back when in the
 * header file.
 *
 */
void
object_cb(Widget w, XtPointer clientData, XtPointer callData)
(
     int choice = (int)clientData  ;

     CUR_OBJECT = choice ;

     /* now that our object has changed, we need to redisplay with the
      * the new object
      */
     postRedisplay() ;
}


/*
 * ambient_cb()
 *
 * this is the callback associated with the ambient menu. same case as
 * the object callback, just set the global variable CUR_AMBIENT to the
 * same number as the selection. all is good, thanks to sweet global
 * variables and ennumeration.
 */
void
ambient_cb(Widget w, XtPointer clientData, XtPointer callData)
{
     int choice = (int)clientData  ;

     CUR_AMBIENT = choice ;

     /* ah yes, we changed some sort of light source, so we are going to
      * have to recalculate all the colors...or this spectral nonsense will
      * not have any real effect.
      */
```

```
     RECALCULATE_COLORS = TRUE ;
     postRedisplay() ;
}


/*
 * light_cb()
 *
 * ah, the old callback for the light1 menu. same story as the ambient
 * callback, except replace the word CUR_AMBIENT with CUR_ILLUMINANT.
 * viva the globals!
 *
 */
void
light_cb(Widget w, XtPointer clientData, XtPointer callData)
{
     int choice = (int)clientData  ;

     CUR_ILLUMINANT = choice ;

     /* let us not forget to recalculate those colors */
     RECALCULATE_COLORS = TRUE ;
     postRedisplay() ;
}



/*
 * light2_cb()
 *
 * ah, the new callback for the light2 menu. same story as the light1
 * callback, except replace the word CUR_ILLUMINANT with CUR_ILLUMINANT2.
 * sweet sweet, precious globals!
 *
 */
void
light2_cb(Widget w, XtPointer clientData, XtPointer callData)
{
     int choice = (int)clientData  ;

     CUR_ILLUMINANT2 = choice ;

     /* did i mention recalculate the colors? */
     RECALCULATE_COLORS = TRUE ;
     postRedisplay() ;
}


/*
 * detect_cb()
 *
 * yet again, a simple callback function to handle when the user
 * selects from the detector/observer menu. simply define the
 * global variable CUR_OBSERVER to be the selection number, and
 * voila...you are done.
 *
 */
void
detect_cb(Widget w, XtPointer clientData, XtPointer callData)
{
     int choice = (int)clientData  ;

     CUR_OBSERVER = choice ;

     /* now don't forget, if any part of the scene change, ie light,
      * material properties or observer, you need to recalculate the
      * display colors.
      */
     RECALCULATE_COLORS = TRUE ;
     postRedisplay() ;
}
```

```
/*
 * adapt_cb()
 *
 * ah, the elusive chromatic adaptation callback.
 * once again this is performed using a simple trick
 * of smoke and mirrors. actually, it is simply the
 * global variable CUR_ADAPT, and some fance ennumeration.
 *
 */
void
adapt_cb(Widget w, XtPointer clientData, XtPointer callData)
{
    int choice = (int)clientData  ;

    CUR_ADAPT = choice ;

    RECALCULATE_COLORS = TRUE ;
    postRedisplay() ;
}


/*
 * help_cb()
 *
 * this callback is called when the user selects the help button
 * from the menubar. it simply manages the help_dialog box, and
 * pops it up. simple, yet elegant.
 *
 */
void
help_cb(Widget w, XtPointer clientData, XtPointer callData)
{

     XtManageChild(help_dialog) ;
     XtPopup(XtParent(help_dialog), XtGrabNone) ;
}



/*
 * processHits()
 *
 * this function is used to process the selection buffer, and
 * to determine if an object was picked. with the mouse that is.
 * most of this function is a direct hack from a similarly named function
 * written by Mark J. Kilgard, in his OpenGL Programming for the
 * X Window System book, Addison Wesley Developers Press, (1996).
 *
 */
int
processHits(GLint hits, GLuint buffer[])
{
    unsigned int i;
    GLint names ;
    GLuint *ptr ;
    GLuint minz, z, match ;
    int j ;

    if(hits < 0) {
       fprintf(stderr, "WARNING, Selection buffer overflow!\n") ;
        return NULL ;
    }

    if(hits == 0)
       return NULL ;

    minz = 0xffffffff ;

    ptr = (GLuint *)buffer ;
    for(i = 0; i<hits;i++){
```

```
        names = *ptr ;
        ptr++ ;
        ptr++ ;
        ptr++ ;
        for(j = 0; j < names; j++){
            match = *ptr ;
            ptr++ ;
        }
    }


    return  (int)match ;

}

#define BUFSIZE 1024

/* pickScene()
 *
 * this function sets up everything so we can use the OpenGL
 * picking functions, to determine if the user clicks the mouse
 * button on an object, and to determine just which object they
 * picked. loosely borrowed from Mark Kilgard.
 *
 */
void
pickScene(int x, int y)
{
    GLuint     selectBuf[BUFSIZE] ;  /* we need a picking buffer */
    GLfloat    width, height ;              /* dimensions of the glxdraw area */
    GLint        viewport[4] ;              /* the current GL viewport */
    GLint      hits ;                /* the hits variable */
    int        name ;                /* give each object a name, baby */

    /* the first thing we need to do is determine the current viewport */
    glGetIntegerv (GL_VIEWPORT, viewport) ;

    /* now we need to define the current OpenGL selection buffer */
    glSelectBuffer (BUFSIZE, selectBuf) ;

    /* we need to set the current mode so that selection is enabled */
    (void)glRenderMode(GL_SELECT) ;

/*
 *   uncomment this out, if the face culling is causing strange
 *   picking issues.
 *
 *   glDisable(GL_CULL_FACE) ;
*/


    /* we need to initialize the gl naming stuff */
    glInitNames() ;

    /* push name 0, so that we start out at name 1 */
    glPushName(0) ;

    /* make sure that we are in projectrion matrix mode */
    glMatrixMode(GL_PROJECTION) ;
    glPushMatrix() ;
    glLoadIdentity() ;

    /* set our pick matrix so that it is really precise */
    gluPickMatrix((GLdouble)x, (GLdouble)(viewport[3] - y),
                    1.0, 1.0, viewport) ;

    /* get the current dimensions of the glx drawing widget */
    XtVaGetValues(glxarea, XtNwidth, &viewWidth, XtNheight,
                &viewHeight, NULL) ;

    /* set the local dimension variables equal to the above dimensions */
    width = (GLfloat)viewWidth ;
```

```
    height = (GLfloat)viewHeight ;

    /* set our orthographic OpenGL viewing info */
    if (width <= (height * 2))
         glOrtho (-6.0, 6.0, -3.0*((GLfloat)height*2)/(GLfloat)width,
                  3.0*((GLfloat)height*2)/(GLfloat)width, -10.0, 10.0);
    else
         glOrtho (-6.0*(GLfloat)height/((GLfloat)height*2),
                  6.0*(GLfloat)width/((GLfloat)height*2), -3.0, 3.0, -10.0, 10.0);

    /* go back to the modelview matrix mode */
    glMatrixMode(GL_MODELVIEW) ;

    /* call the draw() function using the GL_SELECT command, so that
     * the function knows we are going to do some mad picking */
    draw(GL_SELECT) ;

    glMatrixMode(GL_PROJECTION) ;
    glPopMatrix() ;

    glMatrixMode(GL_MODELVIEW) ;

    /* go back into render mode, and get the hits */
    hits = glRenderMode(GL_RENDER) ;

/* uncomment as above
 *
 *  glEnable(GL_CULL_FACE) ;
 */

    /* use the hits, and the selection buffer to process through */
    name = processHits(hits, selectBuf) ;

    /* figure out the given name of the object picked. if there is
     * no name, then we know the user didn't press the mouse on any
     * of the rendered objects. if there is a name, we need to send
     * that name over to the plotting function, so that we can plot
     * the spectra of that object.
     */
    if(name != NULL) {
        plotting(name) ;
    }

}



/* plotting()
 *
 * this function handles all the necessary goods so that we can
 * plot the spectra of any given object, when that object is
 * selected using the middle mouse button.
 *
 */
void
plotting(int pick)
{
    /* we start out by declaring a bunch of temporary variables, since
     * the plotting widget can not handle the large multi-dimensional
     * arrays that are the global variables.
     */
    double    temp_wavelength[93] ;
    double    temp_reflect[93] ;
    double    temp_metamer[93] ;
    double    temp_excite[93] ;
    double    temp_emit[93] ;
    double    temp_amb_fluor[93] ;
    double    temp_ill_fluor[93] ;
    double    temp_fluor[93] ;
    double    illuminant[93] ;     /* the spectra of the current illuminant */
    double    ambient[93] ;       /* likewise for the current ambient light */
    double    equal[93] ;         /* the equal energy spectra */
```

```
double     temp_radiance[93] ;
double     total_radiance[93] ;
double     ill_reflect[93], amb_reflect[93] ;
double     ill_excite[93], equal_excite[93], amb_excite ;
double     scale_ill_reflect[93], scale_amb_reflect[93] ;
double     N ;
static     int line1 = 1267, line2 = NULL;     /* plot line variables */
static     int line3 = NULL, line4 = NULL;
static     int line5 = NULL ;
int        line1color, line2color ;     /* plot line colors */
int        line3color, line4color ;
int        line5color ;
int        rows, collumns ;
int        i = NULL, j, no_col;


 /* use the return_illuminant() function found in spect-math.c to
  * get the current local and ambient illuminants.
  */
 return_illuminant(CUR_ILLUMINANT, illuminant) ;
 return_illuminant(CUR_AMBIENT, ambient) ;

 /* set our temporary spectra equal to the spectra of the selected
  * object, where pick represents the name returned using the pickScene()
  * fuction.
  */
 for(i=0;i<93;i++){
    temp_wavelength[i] = (float)(i*5.0 + 300.0) ;
    temp_reflect[i] = reflect[pick-1][i] ;
    temp_metamer[i] = metamer[pick-1][i] ;
    temp_excite[i] = excite[pick-1][i] ;
    temp_emit[i] = emit[pick-1][i] ;
    temp_fluor[i] = 0.0 ;
    equal[i] = 1.0 ;
}

 /* we need to calculate the effect of fluorescence, if necessary */
 if(FLUORESCENT[pick-1] == TRUE) {
    /* calculate the fluorescence caused by the local illuminant*/
      calc_fluorescence(temp_excite, temp_emit, temp_ill_fluor, LOCAL) ;

    /* now calculate that cause by the ambient light */
      calc_fluorescence(temp_excite, temp_emit, temp_amb_fluor, AMBIENT) ;

    /* finally add these two together */
      add_spect(temp_ill_fluor,  temp_amb_fluor, temp_fluor) ;
}

 /* we need to multiply the reflectance by the illumination */
 mult_spect(illuminant, temp_reflect, ill_reflect) ;
 mult_spect(ambient, temp_reflect, amb_reflect) ;

 /* now scale these spectra, based upon the illuminant slider bars */
 scale_spect(amb_reflect, AmbScale, scale_amb_reflect)  ;
 scale_spect(ill_reflect, DiffScale, scale_ill_reflect) ;

 /* add the ambient and the local radiances to get the total radiance */
 add_spect(scale_amb_reflect, scale_ill_reflect, temp_radiance) ;
 add_spect(temp_radiance,  temp_fluor, total_radiance) ;

 /* manange the plot dialog widget, and the plotting widget itself */
 XtManageChild(plot_dialog) ;
 XtManageChild(plot_widget) ;

 /* if this is the first time the widget is called, we need to set up
  * the widget. 1267 is used as an absurd number that the
  * SciPlotListCreateFromDouble would not return. for some bizarre
  * reason, just setting line1 == NULL would not work....
  */
if(line1 == 1267){
   line1 = SciPlotListCreateFromDouble( plot_widget,
```

```
                 93, temp_wavelength, temp_reflect, "reflectance") ;
         line1color = SciPlotAllocRGBColor(plot_widget, 255, 0, 0) ;
         SciPlotListSetStyle(plot_widget, line1, 1, XtMARKER_CIRCLE,
                          line1color, XtLINE_SOLID) ;
      }
      if(!line2){
         line2 = SciPlotListCreateFromDouble( plot_widget,
                 93, temp_wavelength, total_radiance, "radiance") ;
         line2color = SciPlotAllocRGBColor(plot_widget, 100, 100, 0) ;
         SciPlotListSetStyle(plot_widget, line2, 1, XtMARKER_NONE,
                          line2color, XtLINE_SOLID) ;
      }
      if(!line3){
         line3 = SciPlotListCreateFromDouble( plot_widget,
                 93, temp_wavelength, temp_excite, "excitation") ;
         line3color = SciPlotAllocRGBColor(plot_widget, 0, 200, 0) ;
         SciPlotListSetStyle(plot_widget, line3, 1, XtMARKER_NONE,
                          line3color, XtLINE_DOTTED) ;
      }
      if(!line4){
         line4 = SciPlotListCreateFromDouble( plot_widget,
                 93, temp_wavelength, temp_emit, "emmision") ;
         line4color = SciPlotAllocRGBColor(plot_widget, 255, 0, 255) ;
         SciPlotListSetStyle(plot_widget, line4, 1, XtMARKER_NONE,
                          line4color, XtLINE_DOTTED) ;
      }
      if(!line5){
         line5 = SciPlotListCreateFromDouble( plot_widget,
                 93, temp_wavelength, temp_metamer, "metamer refl") ;
         line5color = SciPlotAllocRGBColor(plot_widget, 0, 0, 255) ;
         SciPlotListSetStyle(plot_widget, line5, 1, XtMARKER_SQUARE,
                          line5color, XtLINE_SOLID) ;
      }



      /* if all the lines have already been created, all we need to do
       * is update them with the spectra of the newly selected object.
       */
      SciPlotListUpdateDouble(plot_widget, line1, 93,
                              temp_wavelength, temp_reflect) ;
      SciPlotListUpdateDouble(plot_widget, line2, 93,
                              temp_wavelength, total_radiance) ;
      SciPlotListUpdateDouble(plot_widget, line3, 93,
                              temp_wavelength, temp_excite) ;
      SciPlotListUpdateDouble(plot_widget, line4, 93,
                              temp_wavelength, temp_emit) ;
      SciPlotListUpdateDouble(plot_widget, line5, 93,
                              temp_wavelength, temp_metamer) ;

      /* now we update the actual widget itself, so that it redraws correctly */
      SciPlotUpdate(plot_widget) ;

      /* finally we pop-up the widget, so the user can see the fruits of all
       * this labor.
       */
      XtPopup(XtParent(plot_dialog), XtGrabNone) ;
}

/* colormapInstall()
 *
 * this function is used to install a colormap for the overlay planes,
 * if such a thing is actually desired.
 */
void
colormapInstall(Widget w, XtPointer clientData, XtPointer callData)
{
      XInstallColormap(display, overlayColormap) ;
}

/* slide_cb()
 *
```

```
 * this function is the callback function for all the slider
 * bars. it determines which slider was moved, and sets the
 * appropriate global variable (good old global variables...)
 */
void
slide_cb(Widget w, XtPointer clientData, XtPointer callData)
{
    int                     slider = (int)clientData ;
    XmScaleCallbackStruct   *cbs = (XmScaleCallbackStruct*)callData ;


    switch(slider){
       case(AMBIENT):
            AmbScale = (float)(cbs->value)/100.0   ;
            break ;
       case(LOCAL):
            DiffScale = (float)(cbs->value)/100.0 ;
            break ;
       case(LOCAL2):
            DiffScale2 = (float)(cbs->value)/100.0 ;
            break ;
    }

    /* the amount of light has changed...so we need to recalculate */
    RECALCULATE_COLORS = TRUE ;
    postRedisplay() ;

}

/* resize()
 *
 * this function is the resize callback for the glwdrawing area.
 * when the window is resized, the drawing area is as well. basically
 * it makes sure the glx contex is current, waits for X to handle all
 * that it needs to do, and then calls the reshape function.
 *
 */
void
resize(Widget w, XtPointer clientData, XtPointer callData)
{
    GLwDrawingAreaCallbackStruct    *Resize =
                (GLwDrawingAreaCallbackStruct *)callData ;

    viewWidth = Resize->width ;
    viewHeight = Resize->height ;
    glXMakeCurrent(display, glxwin, glxcx) ;
    glXWaitX() ;
    reshape(viewWidth, viewHeight) ;
}
```

spect-gl.c

```
/*
 * spect-gl.c
 *
 * Written By: Garrett M. Johnson
 *
 * OpenGL rendering file for the full spectral rendering package
 * using OpenGL, and Motif.
 *
 * This file contains all of the OpenGL rendering code used in
 * the spectral viewer. This code should be rather platform
 * independent, so a port to other windowing systems should be
 * pretty darn easy. The author will leave this task to any
 * other brave soul (porting anything over to windows should
 * constitute some sort of medal...at least in my book).
 * on another note, if you, said end-user, are not satisfied with
 * the rendering of the spectral viewer, this is the file that
 * needs to be altered. this is a fairly simple and painless
 * process. just replace the draw() and reshape() functions
 * with your functions. examples of this can be found in the
 * spect-gl-table.c, and the spect-gl-pasture.c files.
 *
 * Last revised: 07/27/98
 *
 */


#include <stdio.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>
#include <GL/glx.h>
#include "spectral.h"
#include "glm.h"

/* the first thing we need to do is figure out what version of
 * OpenGL we have running. if we have an older version we need
 * to define the bindtexture extension equal to that of the
 * original function.
 */

#if !defined(GL_VERSION_1_1) && !defined(GL_VERSION_1_2)
#define glBindTexture glBindTextureEXT
#endif

int     CUR_OBJECT = TORUS ;           /* set our current object to torus */
Bool    RECALCULATE_COLORS = TRUE ;    /* assume that we need to calc colors */
GLfloat AmbScale = 0.0 ;               /* start off with no ambient */
GLfloat DiffScale = 1.0 ;              /* start off with full local 1 */
GLfloat DiffScale2 = 0.0 ;             /* start off with no local 2 */
GLuint  spherelist ;                   /* sphere display list name */
GLuint  toruslist ;                    /* torus display list name */
GLuint  cow_list ;                     /* cow display list */
GLuint  ship_list ;                    /* spanish galleon list */
GLuint  woman_list ;                   /* woman head list */
GLuint  tricer_list ;                  /* triceratops list */
GLMmodel *model ;                      /* the glm model variable */
GLfloat scale ;                               /* a model scaling variable */

/* reshape()
 *
 * the reshape function is called when the from the resize function
 * in the spect-cb.c file. it is placed inside the spect-gl.c file
```

```
 * because it defines the gl viewpoint and all that good stuff.
 *
 */
void
reshape(int width, int height)
{
     int winWidth, winHeight ;


     winWidth = width ;              /* set the window dimensions */
     winHeight = height ;

     glViewport(0, 0, winWidth, winHeight) ;/* define the viewport */
     glMatrixMode(GL_PROJECTION);
     glLoadIdentity();

     /* set our orthographic projection matrix. if the desired goal is
      * to show some sort of perspective, don't use ortho. (duh?)
      */
     if (winWidth <= (winHeight * 2))
          glOrtho (-6.0, 6.0, -3.0*((GLfloat)winHeight*2)/(GLfloat)winWidth,
                   3.0*((GLfloat)winHeight*2)/(GLfloat)winWidth, -10.0, 10.0);
     else
          glOrtho (-6.0*(GLfloat)winHeight/((GLfloat)winHeight*2),
                   6.0*(GLfloat)winWidth/((GLfloat)winHeight*2), -3.0, 3.0, -10.0, 10.0);
     glMatrixMode(GL_MODELVIEW);
     glLoadIdentity();

}



/* gl_init()
 *
 * the gl_init function is used to initialize all the good wholesome
 * OpenGL states. This is where we define the light source properties,
 * including position. Since we are kludging the actual lighting
 * so that it simulates some full-spectral goodness, we simply set all
 * the light source RGBA variables equal to 1.0. we will deal with
 * the color calculations in spect-math.c. oh yeah, in this file we
 * also initialize all the models used in the spectral viewer. if you
 * as an enduser want new cooler models, simply declare said model
 * in this function.
 */
void
gl_init(void)
{

     /* set our light source variables */
     GLfloat diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
     GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
     GLfloat position[] = { 0.0, 0.0, 08.0, 1.0 };
     GLfloat position2[] = { 0.0, -6.0, 08.0, 1.0 };

     /* now define light source number one (local 1) */
     glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
     glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
     glLightfv(GL_LIGHT0, GL_POSITION, position);
     glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE) ;

     /* now define light source number two (local 2) */
     glLightfv(GL_LIGHT1, GL_DIFFUSE, diffuse);
     glLightfv(GL_LIGHT1, GL_SPECULAR, light_specular);
     glLightfv(GL_LIGHT1, GL_POSITION, position2);
     glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE) ;


     /* now that we have defined the light sources, lets enable them */
     glEnable(GL_LIGHTING);
     glEnable(GL_LIGHT0);
     glEnable(GL_LIGHT1);
```

```
/* enable the standard z-buffer depth testing, and back face culling */
glEnable(GL_DEPTH_TEST) ;
glEnable(GL_CULL_FACE) ;

/* since the spectral viewer scales the object sizes based on the
 * total number of material files present, we should set the auto
 * normalize function, so OpenGL handles our normals for us */
glEnable(GL_NORMALIZE) ;

/* generate the necessary display lists for the sphere and torus */
spherelist = glGenLists(1) ;
toruslist = glGenLists(1) ;

/* now it is time to define the sphere display list. the sphere()
 * function can be found in the shapes.c file.
 */
glNewList(spherelist, GL_COMPILE) ;
   sphere(2.8, 30, 30)  ;
glEndList() ;

/* well, we are going to define the display list for the torus, but
 * we are going to get a little fancier. since our torus is a simple
 * little mathematically defined model, we don't really have any
 * texture coordinates for it. so we will have OpenGL generate the
 * coordinates (and do a pretty lousy job at that) for us.
 */
glNewList(toruslist, GL_COMPILE) ;
     glEnable(GL_TEXTURE_GEN_S) ; /* enable texture coordinate generation */
     glEnable(GL_TEXTURE_GEN_T) ;  /* enable texture coordinate generation */

   /* now lets generate the texture coordinates, based upon a textured
    * spherical mapping
    */
   glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP) ;
     glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP) ;

   /* finally call the torus() function, defined in shapes.c */
   Torus(1.05, 1.65, 30, 30)  ;

   /* disable the texture generation, so GL doesn't go nuts and try to
    * define texture coordinates for everything.
    */
     glDisable(GL_TEXTURE_GEN_S) ;
     glDisable(GL_TEXTURE_GEN_T) ;
glEndList() ;

/* now the fun stuff, we get to define the display lists for all the
 * other cool models. this is done using wavefront OBJ model files,
 * and the glm Rendering library. This library was written by the
 * ever impressive Nate Robbins, and can be found at various web sites,
 * including http://trant.sgi.com/opengl/examples/more_samples/smooth.
 * all in all, this is a very cool little library, my thanks to nate.
 */
model = glmReadOBJ("obj/cow.obj") ;    /* give glm the name of the obj file */
scale = glmUnitize(model) ;            /* unitize, in case it is huge */
glmFacetNormals(model) ;        /* make sure the normals are correct */
glmVertexNormals(model, 90.0) ;
glmLinearTexture(model) ;              /* define a linear texture mapping */
glmScale(model, 2.8) ;          /* scale the model back up to size */

/* now we let glm define the display list, using smooth shading, and
 * texture coordinates.
 */
cow_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE) ;

/* basically we do the same things as above, for all the other models */
model = glmReadOBJ("obj/galleon.obj") ;
scale = glmUnitize(model) ;
glmFacetNormals(model) ;
glmVertexNormals(model, 90.0) ;
glmScale(model, 2.8) ;
glmLinearTexture(model) ;
```

```
    ship_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE) ;

    model : glmReadOBJ("obj/womanhead.obj") ;
    scale = glmUnitize(model) ;
    glmFacetNormals(model) ;
    glmVertexNormals(model, 90.0) ;
    glmScale(model, 2.5) ;
    glmLinearTexture(model) ;

    woman_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE) ;

    model = glmReadOBJ("obj/triceratops.obj") ;
    scale = glmUnitize(model) ;
    glmFacetNormals(model) ;
    glmVertexNormals(model, 90.0) ;
    glmScale(model, 3.5) ;
    glmLinearTexture(model) ;

    tricer_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE) ;

    /* set our texture mapping so that it modulates the color of the
     * object, rather than replaces. basically this is done so we
     * can hack some imitation spectral texture maps, using full-spectral
     * object colors, and non-spectrally selective texture maps.
     * basically that is just a fancy name for grey-scale textures
     */
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

    /* clear the background color to white, for the heck of it */
    glClearColor(1.0, 1.0, 1.0, 1.0)   ;
}

/* draw_object()
 *
 * this is a simple function designed to make it easy to switch
 * between objects in the spectral viewer. basically it allows
 * the drawing call to be the same, regardless of the current
 * model choice. the draw_object() function then uses the
 * (sweet sweet) global variable to determine the correct
 * display list to call. so simple, so sweet, so hackesque.
 */
void
draw_object(void)
{
    long random()   ; /* in case we want to randomly rotate the models */
    GLfloat i ;              /* our random number, of course */

    i = (GLfloat)random()   ;

    /* basically we use the CUR_OBJECT variable ,set in the object_cb()
     * function, to control the switch statement. remember, we enumerated
     * the object names in the header file.
     */
    switch(CUR_OBJECT){
       case  TORUS:
           glCallList(toruslist) ;
           break ;
        case  SPHERE:
            glCallList(spherelist) ;
            break ;
        case  COW:
           glPushMatrix() ;
               /* lets rotate the cows, for good measure... */
               glRotatef(i, 0.0, 1.0, 0.0) ;
                glCallList(cow_list) ;
           glPopMatrix() ;
            break ;
        case  SHIP:
            glCallList(ship_list) ;
            break ;
        case  WOMAN:
```

```
            glPushMatrix() ;
                glRotatef(i, 0.0, 1.0, 0.0) ;
                 glCallList(woman_list) ;
            glPopMatrix() ;
             break ;
        case  TRICER:
            glPushMatrix() ;
               glRotatef(-35.0, 0.0, 1.0, 0.0) ;
                 glCallList(tricer_list) ;
            glPopMatrix() ;
             break ;

    }

}


/* draw()
 *
 * oh yeah baby, the meat of all our rendering is done in the good
 * old draw() function. this is where it all comes together, and
 * makes our life fun and exciting. as an added bonus, if at any
 * time you, the user, get bored with the ColorChecker type layout
 * of the interactive view, this is the only function that really
 * needs to be changed. in this function we call out for the spectral
 * color calculations, calculate what the resultant spectra would
 * look like if imaged by our current observer, and then render our
 * objects using these "tristimulus" values. sounds simple enough.
 */
void
draw(GLenum mode)
{
    static double     amb_rgb[24][3] ;      /* ambient tristimulus values */
    static double     local_rgb[24][3] ;    /* local 1 tristim values */
    static double     local2_rgb[24][3] ;   /* local 2 tristim values */
    static double     amb_met_rgb[24][3] ;  /* metameric ambient trist */
    static double     local_met_rgb[24][3] ;       /* metameric local 1 trist */
    static double     local2_met_rgb[24][3] ;      /* metameric local 2 trist */
    static double     spec_rgb[24][3] ;     /* specular local 1 trist */
    static double     met_spec_rgb[24][3] ; /* metameric local 1 specular */
    static double     spec2_rgb[24][3] ;    /* specular local 2 trist */
    static double     met2_spec_rgb[24][3] ;       /* metameric local 2 specular */
    static double     amb_back_rgb[3] ;     /* ambient background trist */
    static double     local_back_rgb[3] ;   /* local background trist */

    /* defaul light and material properties */
    GLfloat      mat_diffuse[] = {1.0, 1.0, 1.0, 1.0} ;
    GLfloat      mat_specular[] = {1.0, 1.0, 1.0, 1.0} ;
    GLfloat      test_specular[] = {1.0, 1.0, 1.0, 1.0} ;
    GLfloat      mat_ambient[] = {0.0, 0.0, 0.0, 0.0} ;
    GLfloat      null_ambient[] = {0.0, 0.0, 0.0, 0.0} ;
    GLfloat      null_specular[] = {0.0, 0.0, 0.0, 0.0} ;
    GLfloat   light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
    GLfloat   light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat   light_specular[] = { 1.0, 1.0, 1.0, 1.0 };

    /* current gloss (shininess) component */
    GLfloat   cur_gloss ;

    /* we need to define the position and scaling of the individual objects */
    GLfloat   x, shift = 0.0;
    GLfloat   y , scale = 1.0 ;
    int       rows = 1;
    int       collumns, i ;
    int       no_col = 0 ;

    /* an easy method of simulating more than one light source, accomplished
     * by using the accumulation buffer, and a simple (but large) for loop.
     */
    int       light_number = 1 ;
    int       accum_pass ;
```

```
/* since we are all fancy with the metameric objects, we need to define
 * the right and left clipping planes. essentially this allows us to
 * draw half an object with one material property, and the other half
 * with a different property.
 */
GLdouble     right_clip[] = {-1.0, 0.0, 0.0, 0.0} ;
GLdouble     left_clip[] = {1.0, 0.0, 0.0, 0.0} ;



/* we want the disable texture mapping, until we need it. */
glDisable(GL_TEXTURE_2D) ;


/* we need to arbitrarily determine the scaling of the objects,
 * based upon the total number of different objects. this was
 * essentially figured out through a trial and error bases.
 * the number of rows is calculated so that there is a reasonable
 * number of objects in each row, and so that 24 objects would
 * be layed out in the style of the macbeth colorchecker.
 */
if(NUM_FILES>3){
    rows = 2 ;
    scale = 0.66 ;
}
if(NUM_FILES>8){
    rows = 3 ;
    scale = 0.5 ;
}
if(NUM_FILES>15){
    rows = 4 ;
    scale = 0.333 ;
}

/* Calculate the number of collumns based on the total number of
 * files, and rows. this determines the ultimate layout of the objects
 */
if(NUM_FILES % rows == 0){
   collumns = NUM_FILES/rows ;
}
else{
   collumns = NUM_FILES/rows + 1 ;
}

/* set the initial x and y position based on the total number of
 * collumns and rows. once again this was determined on a trial
 * and error basis.
 */
x = (GLfloat) -3.0*(collumns-1.0) ;
y = (GLfloat) (rows   1.0)*3.0 ;


/* Scale the ambient and diffuse lighting, according to sliders  */

light_ambient[0] = light_ambient[1] = light_ambient[2] = AmbScale ;
light_diffuse[0] = light_diffuse[1] = light_diffuse[2] = DiffScale ;
light_specular[0] = light_specular[1] = light_specular[2] = DiffScale ;
light_diffuse2[0] = light_diffuse2[1] = light_diffuse2[2] = DiffScale2 ;
light_specular2[0] = light_specular2[1] = light_specular2[2] = DiffScale2 ;

/* set up the light properties based upon the above calculations */
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient) ;
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);

/* disable the lights, unless we need them */
glDisable(GL_LIGHT0) ;
glDisable(GL_LIGHT1) ;

/* if the second light is turned on, set up the properties and enable it */
if(CUR_ILLUMINANT2 != NONE) {
```

```
        glLightfv(GL_LIGHT1, GL_DIFFUSE, light_diffuse2) ;
        glLightfv(GL_LIGHT1, GL_SPECULAR, light_specular2) ;
      light_number = 2 ;
   }

   /* now clear the accumulation buffer */
   glClear(GL_ACCUM_BUFFER_BIT) ;

   /* set our drawing and reading buffers to be the back buffer,
    * so we do not display until after we have calculated the
    * entire accumulation buffer.
    */
   glDrawBuffer(GL_BACK) ;
   glReadBuffer(GL_BACK) ;


/* fogive the poor indentation...the for loop was added later... */
for(accum_pass = 0 ; accum_passs < light_number ; accum_pass ++) {

   /* If the colors need to be recalculated do so now. first
    attempt to do it with the locall light source. */
   if(RECALCULATE_COLORS && accum_pass == 0){
      get_rgb(local_rgb, local_met_rgb, LOCAL) ;
      get_rgb(amb_rgb, amb_met_rgb, AMBIENT) ;
      get_specular_rgb(spec_rgb, met_spec_rgb, LOCAL2) ;
      get_background(amb_back_rgb, local_back_rgb) ;
      glEnable(GL_LIGHT0) ;
   } else

   /* if we are on the second pass of the big old (poorly indented)
    * for loop, we need to calculate the colors based on the
    * second light source */
   if(RECALCULATE_COLORS && accum_pass == 0) {
      get_rgb(local_rgb, local_met_rgb, LOCAL2) ;
      get_rgb(amb_rgb, amb_met_rgb, AMBIENT) ;
      get_specular_rgb(spec_rgb, met_spec_rgb, LOCAL2) ;
      get_background(amb_back_rgb, local_back_rgb, LOCAL2) ;

      /* disable local 1, and enable local 2 */
      glDisable(GL_LIGHT0) ;
      glEnable(GL_LIGHT1) ;
      RECALCULATE_COLORS = FALSE ;
   }

   /* clear our color and depth buffers */
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glLoadIdentity() ;

   /* set our ambient material to be NULL initially (or else
    * there will be a slight leak of ambient light */
   glMaterialfv(GL_FRONT, GL_AMBIENT, null_ambient) ;

   /* set the diffuse material properties to the tristimulus
    * values calculated using the get_rgb() function, based
    * upon the current observer.
    */
   mat_diffuse[0] = local_back_rgb[0] ;
   mat_diffuse[1] = local_back_rgb[1] ;
   mat_diffuse[2] = local_back_rgb[2] ;
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse) ;

   /* if the ambient light scale is actually not 0.0, then we
    * need to set the ambient material properties to the
    * the ambient tristimulus values calculated with the
    * get_rgb() function.
    */
   if(AmbScale != 0.0){
       mat_ambient[0] = amb_back_rgb[0] ;
       mat_ambient[1] = amb_back_rgb[1] ;
       mat_ambient[2] = amb_back_rgb[2] ;
       glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient) ;
```

```
}

/* default the specular tristimulus values to NULL as well */
glMaterialfv(GL_FRONT, GL_SPECULAR, null_ambient) ;

/* now we need to make our pretty background. basically it
 * is just a really really really big sphere, defaulted to
 * a non-selective 40% reflectance grey. essentially, this
 * provides a good mechanism to allow the user to determine
 * the color of the light source[s].
 */
glPushMatrix() ;
   glTranslated(0.0, 0.0, -16.0) ;
    sphere(14.0, 40, 40)  ;
glPopMatrix() ;

/* once again, default the ambient back to NULL */
glMaterialfv(GL_FRONT, GL_AMBIENT, null_ambient) ;


/* set the scale factor to that calculated based upon the
 * total number of material files entered.
 */
glScalef(scale, scale, scale) ;

/* now we begin the loop which creates our fun spectral
 * scene. basically, we do one loop for each material
 * file.
 */
for(i=0;i<NUM_FILES;i++){
   /* first we need to figure out if we are in selection or
    * rendering mode. if we are in selection, we need to load
    * the object names
    */
   if( mode == GL_SELECT){
       glLoadName(i+1) ;
   }
   /* now we need to determine if the object is a single object,
    * or a metameric pair. by including metamerism in the actual
    * material file, we can allow a single object to have multiple
    * spectral properties. basically this is for demonstration
    * purposes.
    */
   if (METAMERIC[i] == TRUE){
       glPushMatrix() ;
            /* set the diffuse and specular RGBs to the tristimulus
           * values calculated in the get_rgb() function */
         mat_diffuse[0] = local_rgb[i][0] ;
           mat_diffuse[1] = local_rgb[i][1] ;
           mat_diffuse[2] = local_rgb[i][2] ;
           mat_specular[0] = spec_rgb[i][0] ;
           mat_specular[1] = spec_rgb[i][1] ;
           mat_specular[2] = spec_rgb[i][2] ;

         /* set the gloss component */
         cur_gloss = (GLfloat)gloss[i] ;

         /* now set the diffuse material property */
            glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse) ;

         /* if the gloss is not 0 (perfectly diffuse) then we
          * need to set the specular material properties. if
          * the gloss is 0, we don't want to set any specular
          * properties, as that creates a really really really
          * really really washed out white spot on any object.
          */
         if(gloss[i] > 0.0){
               glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular) ;
         }
         else{
               glMaterialfv(GL_FRONT, GL_SPECULAR, null_specular) ;
```

```
}

/* set the material properties of shininess (gloss) */
   glMaterialf(GL_FRONT, GL_SHININESS, cur_gloss) ;

/* if the ambient light is turned on, using the slider bar,
 * then we need to set the material properties to those
 * calculated with get_rgb(), otherwise we want to keep
 * the ambient properties at NULL (to prevent ambient
 * light from leaking and causing washed out colors).
 */
  if(AmbScale != 0.0){
       mat_ambient[0] = amb_rgb[i][0] ;
       mat_ambient[1] = amb_rgb[i][1] ;
       mat_ambient[2] = amb_rgb[i][2] ;
        glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient) ;
  }

/* now we figure where exactly we want the objects to
 * be positioned. basically this was determined with
 * the row/collumn calculations.
 */
  glTranslatef(x+shift, y, 0.0) ;

/* now we get to actually do some rendering. first, though
 * we need to define the first clipping plane. the clipping
 * plane passes through the position of the object,
 * effectively cutting the object in half.
 */
glPushMatrix() ;
     /* define the clipping plane position */
     right_clip[0] = -3*(x+shift) ;

     /* declare, and enable the clipping plane */
     glClipPlane(GL_CLIP_PLANE1, right_clip) ;
     glEnable(GL_CLIP_PLANE1) ;

     /* call the draw_object() function to draw the
      * current object.
      */
       draw_object() ;

     /* disable the clipping plane. */
     glDisable(GL_CLIP_PLANE1) ;
glPopMatrix() ;

/* basically now we have half an object floating in
 * space. so now we need to draw the other half of said
 * object. this other half has different material p
 * material properties, so we need to set those up first.
 */
  mat_diffuse[0] = local_met_rgb[i][0] ;
  mat_diffuse[1] = local_met_rgb[i][1] ;
  mat_diffuse[2] = local_met_rgb[i][2] ;
  mat_specular[0] = met_spec_rgb[i][0] ;
  mat_specular[1] = met_spec_rgb[i][1] ;
  mat_specular[2] = met_spec_rgb[i][2] ;
  cur_gloss = (GLfloat)met_gloss[i] ;

/* once again declare the diffuse property */
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse) ;

/* if gloss is not 0, declare the specular property */
if(met_gloss[i] > 0.0)
       glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular) ;
else{
       glMaterialfv(GL_FRONT, GL_SPECULAR, null_specular) ;
}

/* now declare the shininess coefficient */
  glMaterialf(GL_FRONT, GL_SHININESS, cur_gloss) ;
```

```
        /* determine if we need to consider the ambient properties */
          if(AmbScale != 0.0)(
               mat_ambient[0] = amb_met_rgb[i][0] ;
               mat_ambient[1] = amb_met_rgb[i][1] ;
               mat_ambient[2] = amb_met_rgb[i][2] ;
                glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient) ;
         }

        /* now we get to do the other half of the rendering. once
         * again we need to set up the other clipping plane first.
         * this clipping plane is designed to cut the objects in
         * half opposite to the first clipping plane.
         */
          glPushMatrix() ;
            /* define, declare, and enable the clipping planes */
            left_clip[0] = 3.0*(x+shift) ;
            glClipPlane(GL_CLIP_PLANE2, left_clip) ;
              glEnable(GL_CLIP_PLANE2) ;

            /* draw the same object as before */
              draw_object() ;

            /* disable the clipping plane */
              glDisable(GL_CLIP_PLANE2) ;
          glPopMatrix() ;

        /* once we have drawn our now whole object, we need to
         * redetermine the position for the next object. basically
         * the x postition is shifted 6 units to the right (positive
         * x-direction.) this was determined thorough a trial and
         * error procedure.
         */

          shift = shift + 6.0 ;
          no_col++ ;
        /* if the no_col is greater than the total number of
         * collumns, then it is time to move to a new row.
         * in that case, we reset the x-shift back to 0, and
         * move 6 units in the negative y-direction.
         */
          if(no_col >= collumns){
               shift = 0.0 ;
               y = y - 6.0 ;
               no_col = 0 ;
         }

      glPopMatrix() ;
}
/* all of the above code is used if the object has a defined
 * metameric pair. in most cases the object has but a single
 * material property. basically we just repeat what we did
 * above, except only once instead of twice
 */
else{
      glPushMatrix() ;
         /* define the diffuse and specular properties */
         mat_diffuse[0] = local_rgb[i][0]  ;
         mat_diffuse[1] = local_rgb[i][1] ;
         mat_diffuse[2] = local_rgb[i][2] ;
          mat_specular[0] = spec_rgb[i][0] ;
          mat_specular[1] = spec_rgb[i][1] ;
          mat_specular[2] = spec_rgb[i][2] ;

        /* define the gloss component */
        cur_gloss = (GLfloat )gloss[i] ;

        /* declare the diffuse material property */
          glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse) ;

        /* determine if we need to declare the specular */
        if(gloss[i] > 0.0){
               glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
```

```
        }
        else{
               glMaterialfv(GL_FRONT, GL_SPECULAR, null_specular) ;
        }

        /* declare the shininess coefficient */
           glMaterialf(GL_FRONT, GL_SHININESS, cur_gloss) ;

        /* determine if we need to define and declare the
         * ambient properties */
         if(AmbScale != 0.0){
             mat_ambient[0] = amb_rgb[i][0] ;
             mat_ambient[1] = amb_rgb[i][1] ;
             mat_ambient[2] = amb_rgb[i][2] ;
              glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient) ;
         }

        /* move ourselfs into proper position */
           glTranslatef(x+shift, y, 0.0) ;

        /* if the current object has a texture file defined
         * we need to set that up */
        if(TEXTURE[i] == TRUE){
             /* enable texture mapping */
             glEnable(GL_TEXTURE_2D) ;

             /* bind the texture map to the object. this is
              * where the NumTex variable comes in handy. this
              * variable automatically tells us which texture
              * mip-map to use.
              */
             glBindTexture(GL_TEXTURE_2D, (GLuint)NumTex[i]) ;

             /* set up the mip-map texture parameters */
             glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                         GL_LINEAR_MIPMAP_LINEAR);
             glTexParameteri(GL_TEXTURE_2D,
                         GL_TEXTURE_MAG_FILTER, GL_LINEAR);

             /* finally draw the dang object */
              draw_object() ;

             /* disable texture mapping, so that all the other
              * objects aren't mapped with this texture */
             glDisable(GL_TEXTURE_2D) ;
        }
        /* if the object does not have a texture file, all we
         * need to do is call the draw_object() function.
         */
        else{
            draw_object() ;
        }


        /* recalculate the position for the next pass */
         shift = shift + 6.0 ;
         no_col++ ;
         if(no_col >= collumns){
             shift = 0.0 ;
             y = y   6.0 ;
            no_col = 0 ;
         }

      glPopMatrix() ;
 }

/* if we have more than one light source turned on, we need to
 * place what we have just drawn into the accumulation buffer,
 * and scale it so that the two slider bar scales add up to be
 * 1.0.
 */
if(accum_pass == 0 && light_number >1 ) {
```

```
            glAccum(GL_ACCUM,, DiffScale/(DiffScale + DIffScale2) ;
        }
        /* if only one light is on, set the accumulation buffer equal
         * to unity
         */
        else if {
            glAccum(GL_ACCUM, 1.0) ;
        }
        /* if both light sources are on, and we are on the second pass
         * of the accumulation loop, use this to scale the buffer */
        else {
            glAccum(GL_ACCUM, DiffScale2/(DiffScale + DiffScale2) ;
        }

    }
}


    /* get the values from the accumulation buffer, and set the drawing
     * buffer equal to those */
    glAccum(GL_RETURN, 1.0) ;

    /* if our display is double buffer, swap buffers */
    if (doubleBuffer) glXSwapBuffers(display, glxwin);

    /* flush the screen, so that all our hard work is displayed.
    glFlush() ;

}
```

spect-math.c

```
/*
 * spect-math.c
 *
 * Written By: Garrett M. Johnson
 *
 * OpenGL spectral color calculation file for the full spectral
 * rendering package using OpenGL, and Motif.
 *
 * This file contains all of the spectral color calculations
 * stuff for this software. basically, this is the meat and
 * potatoes (potatos? tomatos?). in this file we declare the
 * default observer functions, and illuminant functions. if
 * new illuminants or detectors are desired, simply add them to
 * this file. in the future this file will be altered so that
 * it relies less on the sweet global variables, and will be
 * self contained. right now it relies heavily on the globals,
 * and therefore needs to be used in conjunction with the
 * interactive spectral viewer.
 *
 * Last revised: 07/27/98
 *
 */
#include <stdio.h>
#include <math.h>
#include "spectral.h"


int      CUR_OBSERVER = XYZ2 ;  /* default observer state */
int      CUR_ILLUMINANT = D65 ;  /* default illuminant state */
int      CUR_ILLUMINANT2 = NONE ;  /* default illuminant 2 state */
int      CUR_AMBIENT = D65 ;  /* default ambient state */
int      CUR_ADAPT = NONE ;  /* default adaptation state */

/* void return_observer()
 *
 * this function receives the name of the observer, and
 * three empty spectral curves. it fills the curves with
 * the desired responses, depending on the observer name
 * passed into the function. if new observer/detector
 * functions are desired, the current ones can be replaced,
 * or new ones can be added. easy as pie. at least easy as
 * a really easy pie. maybe one of those pre-baked pies...
 *
 */
void
return_observer(int name, RESPONSE *response)
{
    /* declare the variable for the CIE 1931 Standard Observer */
    double    ciexyz2[93][4] = {
       {300, 0.0000, 0.0000, 0.0000}, {305, 0.0000, 0.0000, 0.0000},
       {310, 0.0000, 0.0000, 0.0000}, {315, 0.0000, 0.0000, 0.0000},
       {320, 0.0000, 0.0000, 0.0000}, {325, 0.0000, 0.0000, 0.0000},
       {330, 0.0000, 0.0000, 0.0000}, {335, 0.0000, 0.0000, 0.0000},
       {340, 0.0000, 0.0000, 0.0000}, {345, 0.0000, 0.0000, 0.0000},
       {350, 0.0000, 0.0000, 0.0000}, {355, 0.0000, 0.0000, 0.0000},
       {360, 0.0000, 0.0000, 0.0000}, {365, 0.0000, 0.0000, 0.0000},
       {370, 0.0000, 0.0000, 0.0000}, {375, 0.0000, 0.0000, 0.0000},
       {380, 0.0014, 0.0000, 0.0065}, {385, 0.0022, 0.0001, 0.0105},
       {390, 0.0042, 0.0001, 0.0201}, {395, 0.0076, 0.0002, 0.0362},
       {400, 0.0143, 0.0004, 0.0679}, {405, 0.0232, 0.0006, 0.1102},
       {410, 0.0435, 0.0012, 0.2074}, {415, 0.0776, 0.0022, 0.3713},
       {420, 0.1344, 0.0040, 0.6456}, {425, 0.2148, 0.0073, 1.0391},
```

```
    {430, 0.2839, 0.0116, 1.3856}, {435, 0.3285, 0.0168, 1.6230},
    {440, 0.3483, 0.0230, 1.7471}, {445, 0.3481, 0.0298, 1.7826},
    {450, 0.3362, 0.0380, 1.7721}, {455, 0.3187, 0.0480, 1.7441},
    {460, 0.2908, 0.0600, 1.6692}, {465, 0.2511, 0.0739, 1.5281},
    {470, 0.1954, 0.0910, 1.2876}, {475, 0.1421, 0.1126, 1.0419},
    {480, 0.0956, 0.1390, 0.8130}, {485, 0.0580, 0.1693, 0.6162},
    {490, 0.0320, 0.2080, 0.4652}, {495, 0.0147, 0.2586, 0.3533},
    {500, 0.0049, 0.3230, 0.2720}, {505, 0.0024, 0.4073, 0.2123},
    {510, 0.0093, 0.5030, 0.1582}, {515, 0.0291, 0.6082, 0.1117},
    {520, 0.0633, 0.7100, 0.0782}, {525, 0.1096, 0.7932, 0.0573},
    {530, 0.1655, 0.8620, 0.0422}, {535, 0.2257, 0.9149, 0.0298},
    {540, 0.2904, 0.9540, 0.0203}, {545, 0.3597, 0.9803, 0.0134},
    {550, 0.4334, 0.9950, 0.0087}, {555, 0.5121, 1.0000, 0.0057},
    {560, 0.5945, 0.9950, 0.0039}, {565, 0.6784, 0.9786, 0.0027},
    {570, 0.7621, 0.9520, 0.0021}, {575, 0.8425, 0.9154, 0.0018},
    {580, 0.9163, 0.8700, 0.0017}, {585, 0.9786, 0.8163, 0.0014},
    {590, 1.0263, 0.7570, 0.0011}, {595, 1.0567, 0.6949, 0.0010},
    {600, 1.0622, 0.6310, 0.0008}, {605, 1.0456, 0.5668, 0.0006},
    {610, 1.0026, 0.5030, 0.0003}, {615, 0.9384, 0.4412, 0.0002},
    {620, 0.8544, 0.3810, 0.0002}, {625, 0.7514, 0.3210, 0.0001},
    {630, 0.6424, 0.2650, 0.0000}, {635, 0.5419, 0.2170, 0.0000},
    {640, 0.4479, 0.1750, 0.0000}, {645, 0.3608, 0.1382, 0.0000},
    {650, 0.2835, 0.1070, 0.0000}, {655, 0.2187, 0.0816, 0.0000},
    {660, 0.1649, 0.0610, 0.0000}, {665, 0.1212, 0.0446, 0.0000},
    {670, 0.0874, 0.0320, 0.0000}, {675, 0.0636, 0.0232, 0.0000},
    {680, 0.0468, 0.0170, 0.0000}, {685, 0.0329, 0.0119, 0.0000},
    {690, 0.0227, 0.0082, 0.0000}, {695, 0.0158, 0.0057, 0.0000},
    {700, 0.0114, 0.0041, 0.0000}, {705, 0.0081, 0.0029, 0.0000},
    {710, 0.0058, 0.0021, 0.0000}, {715, 0.0041, 0.0015, 0.0000},
    {720, 0.0029, 0.0010, 0.0000}, {725, 0.0020, 0.0007, 0.0000},
    {730, 0.0014, 0.0005, 0.0000}, {735, 0.0010, 0.0004, 0.0000},
    {740, 0.0007, 0.0002, 0.0000}, {745, 0.0005, 0.0002, 0.0000},
    {750, 0.0003, 0.0001, 0.0000}, {755, 0.0003, 0.0001, 0.0000},
    {760, 0.0003, 0.0001, 0.0000} };

/* phew...now declare the CIE 1964 Standard Observer */
double     ciexyz10[93][4] = {
     {300, 0.0000, 0.0000, 0.0000}, {305, 0.0000, 0.0000, 0.0000},
     {310, 0.0000, 0.0000, 0.0000}, {315, 0.0000, 0.0000, 0.0000},
     {320, 0.0000, 0.0000, 0.0000}, {325, 0.0000, 0.0000, 0.0000},
     {330, 0.0000, 0.0000, 0.0000}, {335, 0.0000, 0.0000, 0.0000},
     {340, 0.0000, 0.0000, 0.0000}, {345, 0.0000, 0.0000, 0.0000},
     {350, 0.0000, 0.0000, 0.0000}, {355, 0.0000, 0.0000, 0.0000},
     {360, 0.0000, 0.0000, 0.0000}, {365, 0.0000, 0.0000, 0.0000},
     {370, 0.0000, 0.0000, 0.0000}, {375, 0.0000, 0.0000, 0.0000},
     {380, 0.0002, 0.0000, 0.0007}, {385, 0.0007, 0.0001, 0.0029},
     {390, 0.0024, 0.0003, 0.0105}, {395, 0.0072, 0.0008, 0.0323},
     {400, 0.0191, 0.0020, 0.0860}, {405, 0.0434, 0.0045, 0.1971},
     {410, 0.0847, 0.0088, 0.3894}, {415, 0.1406, 0.0145, 0.6568},
     {420, 0.2045, 0.0214, 0.9725}, {425, 0.2647, 0.0295, 1.2825},
     {430, 0.3147, 0.0387, 1.5535}, {435, 0.3577, 0.0496, 1.7985},
     {440, 0.3837, 0.0621, 1.9673}, {445, 0.3867, 0.0747, 2.0273},
     {450, 0.3707, 0.0895, 1.9948}, {455, 0.3430, 0.1063, 1.9007},
     {460, 0.3023, 0.1282, 1.7454}, {465, 0.2541, 0.1528, 1.5549},
     {470, 0.1956, 0.1852, 1.3176}, {475, 0.1323, 0.2199, 1.0302},
     {480, 0.0805, 0.2536, 0.7721}, {485, 0.0411, 0.2977, 0.5701},
     {490, 0.0162, 0.3391, 0.4153}, {495, 0.0051, 0.3954, 0.3024},
     {500, 0.0038, 0.4608, 0.2185}, {505, 0.0154, 0.5314, 0.1592},
     {510, 0.0375, 0.6067, 0.1120}, {515, 0.0714, 0.6857, 0.0822},
     {520, 0.1177, 0.7618, 0.0607}, {525, 0.1730, 0.8233, 0.0431},
     {530, 0.2365, 0.8752, 0.0305}, {535, 0.3042, 0.9238, 0.0206},
     {540, 0.3768, 0.9620, 0.0137}, {545, 0.4516, 0.9822, 0.0079},
     {550, 0.5298, 0.9918, 0.0040}, {555, 0.6161, 0.9991, 0.0011},
     {560, 0.7052, 0.9973, 0.0000}, {565, 0.7938, 0.9824, 0.0000},
     {570, 0.8787, 0.9556, 0.0000}, {575, 0.9512, 0.9152, 0.0000},
     {580, 1.0142, 0.8689, 0.0000}, {585, 1.0743, 0.8256, 0.0000},
     {590, 1.1185, 0.7774, 0.0000}, {595, 1.1343, 0.7204, 0.0000},
     {600, 1.1240, 0.6583, 0.0000}, {605, 1.0891, 0.5939, 0.0000},
     {610, 1.0305, 0.5280, 0.0000}, {615, 0.9507, 0.4618, 0.0000},
     {620, 0.8563, 0.3981, 0.0000}, {625, 0.7549, 0.3396, 0.0000},
     {630, 0.6475, 0.2835, 0.0000}, {635, 0.5351, 0.2283, 0.0000},
     {640, 0.4316, 0.1798, 0.0000}, {645, 0.3437, 0.1402, 0.0000},
```

```
     {650, 0.2683, 0.1076, 0.0000}, {655, 0.2043, 0.0812, 0.0000},
     {660, 0.1526, 0.0603, 0.0000}, {665, 0.1122, 0.0441, 0.0000},
     {670, 0.0813, 0.0318, 0.0000}, {675, 0.0579, 0.0226, 0.0000},
     {680, 0.0409, 0.0159, 0.0000}, {685, 0.0286, 0.0111, 0.0000},
     {690, 0.0199, 0.0077, 0.0000}, {695, 0.0138, 0.0054, 0.0000},
     {700, 0.0096, 0.0037, 0.0000}, {705, 0.0066, 0.0026, 0.0000},
     {710, 0.0046, 0.0018, 0.0000}, {715, 0.0031, 0.0012, 0.0000},
     {720, 0.0022, 0.0008, 0.0000}, {725, 0.0015, 0.0006, 0.0000},
     {730, 0.0010, 0.0004, 0.0000}, {735, 0.0007, 0.0003, 0.0000},
     {740, 0.0005, 0.0002, 0.0000}, {745, 0.0004, 0.0001, 0.0000},
     {750, 0.0003, 0.0001, 0.0000}, {755, 0.0002, 0.0001, 0.0000},
     {760, 0.0001, 0.0000, 0.0000} }   ;

/* just for kicks, declare the CIE Standard Deviate Observer */
double  stdev[93][4] = {
     {300, 0.0000, 0.0000, 0.0000}, {305, 0.0000, 0.0000, 0.0000},
     {310, 0.0000, 0.0000, 0.0000}, {315, 0.0000, 0.0000, 0.0000},
     {320, 0.0000, 0.0000, 0.0000}, {325, 0.0000, 0.0000, 0.0000},
     {330, 0.0000, 0.0000, 0.0000}, {335, 0.0000, 0.0000, 0.0000},
     {340, 0.0000, 0.0000, 0.0000}, {345, 0.0000, 0.0000, 0.0000},
     {350, 0.0000, 0.0000, 0.0000}, {355, 0.0000, 0.0000, 0.0000},
     {360, 0.0000, 0.0000, 0.0000}, {365, 0.0000, 0.0000, 0.0000},
     {370, 0.0000, 0.0000, 0.0000}, {375, 0.0006, 0.0000, 0.0025},
     {380, 0.0014, 0.0000, 0.0065}, {385, 0.0022, 0.0001, 0.0105},
     {390, 0.0042, 0.0001, 0.0201}, {395, 0.0076, 0.0002, 0.0360},
     {400, 0.0140, 0.0004, 0.0669}, {405, 0.0223, 0.0005, 0.1066},
     {410, 0.0409, 0.0008, 0.1964}, {415, 0.0707, 0.0013, 0.3419},
     {420, 0.1210, 0.0025, 0.5898}, {425, 0.1951, 0.0054, 0.9571},
     {430, 0.2591, 0.0094, 1.3726}, {435, 0.3109, 0.0147, 1.5090},
     {440, 0.3220, 0.0213, 1.6392}, {445, 0.3265, 0.0289, 1.6954},
     {450, 0.3240, 0.0385, 1.7266}, {455, 0.3166, 0.0495, 1.7414},
     {460, 0.2944, 0.0608, 1.6863}, {465, 0.2603, 0.0736, 1.5623},
     {470, 0.2140, 0.0915, 1.3579}, {475, 0.1684, 0.1115, 1.1395},
     {480, 0.1212, 0.1354, 0.8989}, {485, 0.0805, 0.1633, 0.6803},
     {490, 0.0534, 0.2015, 0.5199}, {495, 0.0344, 0.2526, 0.4008},
     {500, 0.0236, 0.3185, 0.3117}, {505, 0.0191, 0.4042, 0.2442},
     {510, 0.0239, 0.4993, 0.1810}, {515, 0.0424, 0.6035, 0.1267},
     {520, 0.0751, 0.7041, 0.0899}, {525, 0.1190, 0.7872, 0.0669},
     {530, 0.1716, 0.8595, 0.0484}, {535, 0.2274, 0.9159, 0.0327},
     {540, 0.2871, 0.9545, 0.0208}, {545, 0.3512, 0.9792, 0.0122},
     {550, 0.4195, 0.9930, 0.0067}, {555, 0.4927, 0.9972, 0.0035},
     {560, 0.5698, 0.9911, 0.0015}, {565, 0.6498, 0.9742, 0.0003},
     {570, 0.7287, 0.9493, 0.0000}, {575, 0.7999, 0.9132, 0.0001},
     {580, 0.8646, 0.8627, 0.0002}, {585, 0.9220, 0.8036, 0.0000},
     {590, 0.9663, 0.7441, 0.0000}, {595, 0.9930, 0.6823, 0.0000},
     {600, 0.9966, 0.6148, 0.0000}, {605, 0.9818, 0.5472, 0.0000},
     {610, 0.9431, 0.4831, 0.0000}, {615, 0.8854, 0.4225, 0.0000},
     {620, 0.8096, 0.3640, 0.0000}, {625, 0.7168, 0.3065, 0.0000},
     {630, 0.6182, 0.2538, 0.0000}, {635, 0.5264, 0.2093, 0.0002},
     {640, 0.4394, 0.1702, 0.0000}, {645, 0.3564, 0.1357, 0.0000},
     {650, 0.2816, 0.1058, 0.0000}, {655, 0.2186, 0.0810, 0.0000},
     {660, 0.1659, 0.0610, 0.0000}, {665, 0.1228, 0.0449, 0.0000},
     {670, 0.0893, 0.0325, 0.0000}, {675, 0.0653, 0.0238, 0.0000},
     {680, 0.0481, 0.0176, 0.0000}, {685, 0.0338, 0.0125, 0.0000},
     {690, 0.0231, 0.0087, 0.0000}, {695, 0.0161, 0.0060, 0.0000},
     {700, 0.0116, 0.0043, 0.0000}, {705, 0.0082, 0.0030, 0.0000},
     {710, 0.0059, 0.0022, 0.0000}, {715, 0.0042, 0.0016, 0.0000},
     {720, 0.0029, 0.0010, 0.0000}, {725, 0.0020, 0.0007, 0.0000},
     {730, 0.0014, 0.0005, 0.0000}, {735, 0.0010, 0.0004, 0.0000},
     {740, 0.0007, 0.0002, 0.0000}, {745, 0.0005, 0.0002, 0.0000},
     {750, 0.0003, 0.0001, 0.0000}, {755, 0.0002, 0.0001, 0.0000},
     {760, 0.0002, 0.0001, 0.0000} }  ;

/* declare a simple RGB digital camera sensitivity (KODAK DCS200) */
double  rgb[93][4] = {
     {300, 0.0000, 0.0000, 0.0000}, {305, 0.0000, 0.0000, 0.0000},
     {310, 0.0000, 0.0000, 0.0000}, {315, 0.0000, 0.0000, 0.0000},
     {320, 0.0000, 0.0000, 0.0000}, {325, 0.0000, 0.0000, 0.0000},
     {330, 0.0000, 0.0000, 0.0000}, {335, 0.0000, 0.0000, 0.0000},
     {340, 0.0000, 0.0000, 0.0000}, {345, 0.0000, 0.0000, 0.0000},
     {350, 0.0000, 0.0000, 0.0000}, {355, 0.0000, 0.0000, 0.0000},
     {360, 0.0000, 0.0000, 0.0000}, {365, 0.0000, 0.0000, 0.0000},
```

```
    {370, 0.0000, 0.0000, 0.0000}, {375, 0.0000, 0.0000, 0.0000},
    {380, 0.0000, 0.0001, 0.0000}, {385, 0.0000, 0.0001, 0.0000},
    {390, 0.0001, 0.0002, 0.0000}, {395, 0.0003, 0.0002, 0.0065},
    {400, 0.0004, 0.0003, 0.0751}, {405, 0.0006, 0.0003, 0.1437},
    {410, 0.0007, 0.0004, 0.2123}, {415, 0.0008, 0.0004, 0.2892},
    {420, 0.0009, 0.0004, 0.3661}, {425, 0.0009, 0.0004, 0.4952},
    {430, 0.0009, 0.0005, 0.6243}, {435, 0.0009, 0.0006, 0.7281},
    {440, 0.0009, 0.0008, 0.8320}, {445, 0.0010, 0.0019, 0.9160},
    {450, 0.0011, 0.0031, 1.0000}, {455, 0.0010, 0.0080, 0.9579},
    {460, 0.0009, 0.0129, 0.9158}, {465, 0.0008, 0.0256, 0.7915},
    {470, 0.0007, 0.0384, 0.6673}, {475, 0.0007, 0.0720, 0.5614},
    {480, 0.0007, 0.1056, 0.4556}, {485, 0.0007, 0.1799, 0.3597},
    {490, 0.0006, 0.2542, 0.2639}, {495, 0.0006, 0.3673, 0.2001},
    {500, 0.0006, 0.4805, 0.1364}, {505, 0.0006, 0.5834, 0.0977},
    {510, 0.0007, 0.6863, 0.0591}, {515, 0.0008, 0.7987, 0.0409},
    {520, 0.0008, 0.9111, 0.0228}, {525, 0.0008, 0.9555, 0.0149},
    {530, 0.0008, 1.0000, 0.0070}, {535, 0.0008, 0.9836, 0.0043},
    {540, 0.0008, 0.9673, 0.0017}, {545, 0.0009, 0.8606, 0.0010},
    {550, 0.0009, 0.7539, 0.0004}, {555, 0.0016, 0.6457, 0.0004},
    {560, 0.0023, 0.5376, 0.0004}, {565, 0.0050, 0.4127, 0.0004},
    {570, 0.0077, 0.2878, 0.0004}, {575, 0.0145, 0.2095, 0.0004},
    {580, 0.0213, 0.1313, 0.0004}, {585, 0.0369, 0.0910, 0.0004},
    {590, 0.0526, 0.0508, 0.0004}, {595, 0.1404, 0.0339, 0.0004},
    {600, 0.2283, 0.0170, 0.0004}, {605, 0.4627, 0.0108, 0.0003},
    {610, 0.6971, 0.0047, 0.0003}, {615, 0.8276, 0.0027, 0.0003},
    {620, 0.9581, 0.0008, 0.0003}, {625, 0.9790, 0.0006, 0.0003},
    {630, 1.0000, 0.0004, 0.0003}, {635, 0.9582, 0.0003, 0.0003},
    {640, 0.9165, 0.0003, 0.0004}, {645, 0.7958, 0.0003, 0.0003},
    {650, 0.6752, 0.0003, 0.0002}, {655, 0.5298, 0.0002, 0.0002},
    {660, 0.3844, 0.0002, 0.0002}, {665, 0.2729, 0.0001, 0.0001},
    {670, 0.1615, 0.0001, 0.0001}, {675, 0.1136, 0.0001, 0.0001},
    {680, 0.0658, 0.0001, 0.0002}, {685, 0.0482, 0.0001, 0.0002},
    {690, 0.0306, 0.0002, 0.0003}, {695, 0.0253, 0.0001, 0.0002},
    {700, 0.0200, 0.0001, 0.0002}, {705, 0.0147, 0.0000, 0.0001},
    {710, 0.0094, 0.0000, 0.0000}, {715, 0.0041, 0.0000, 0.0000},
    {720, 0.0000, 0.0000, 0.0000}, {725, 0.0000, 0.0000, 0.0000},
    {730, 0.0000, 0.0000, 0.0000}, {735, 0.0000, 0.0000, 0.0000},
    {740, 0.0000, 0.0000, 0.0000}, {745, 0.0000, 0.0000, 0.0000},
    {750, 0.0000, 0.0000, 0.0000}, {755, 0.0000, 0.0000, 0.0000},
    {760, 0.0000, 0.0000, 0.0000} } ;

/* declare RGB photographic sensitivities. these were borrowed
 * from Digital Color Management: Encoding Solutions, by
 * E.J. Giorgianni and T.E. Madden, Addison-Wesley (1998).
 */
double   photo[93][4] = {
    {300, 0.0000, 0.0000, 0.0000}, {305, 0.0000, 0.0000, 0.0000},
    {310, 0.0000, 0.0000, 0.0000}, {315, 0.0000, 0.0000, 0.0000},
    {320, 0.0000, 0.0000, 0.0000}, {325, 0.0000, 0.0000, 0.0000},
    {330, 0.0000, 0.0000, 0.0000}, {335, 0.0000, 0.0000, 0.0000},
    {340, 0.0000, 0.0000, 0.0000}, {345, 0.0000, 0.0000, 0.0000},
    {350, 0.0000, 0.0000, 0.0000}, {355, 0.0000, 0.0000, 0.0000},
    {360, 0.0000, 0.0000, 0.0000}, {365, 0.0000, 0.0000, 0.0000},
    {370, 0.0000, 0.0000, 0.0000}, {375, 0.0000, 0.0000, 0.0250},
    {380, 0.0000, 0.0000, 0.0500}, {385, 0.0000, 0.0000, 0.0750},
    {390, 0.0000, 0.0000, 0.1000}, {395, 0.0000, 0.0000, 0.2500},
    {400, 0.0000, 0.0000, 0.4000}, {405, 0.0000, 0.0000, 0.4500},
    {410, 0.0000, 0.0000, 0.5000}, {415, 0.0000, 0.0000, 0.5700},
    {420, 0.0000, 0.0000, 0.6400}, {425, 0.0000, 0.0000, 0.6400},
    {430, 0.0000, 0.0000, 0.6400}, {435, 0.0000, 0.0000, 0.6200},
    {440, 0.0000, 0.0000, 0.6000}, {445, 0.0000, 0.0000, 0.5750},
    {450, 0.0000, 0.0500, 0.5500}, {455, 0.0000, 0.0500, 0.5250},
    {460, 0.0500, 0.0500, 0.5000}, {465, 0.0500, 0.0500, 0.4000},
    {470, 0.0500, 0.0500, 0.3000}, {475, 0.0500, 0.0500, 0.2500},
    {480, 0.0500, 0.0500, 0.2000}, {485, 0.0500, 0.0750, 0.1500},
    {490, 0.0500, 0.1000, 0.1000}, {495, 0.0650, 0.1250, 0.0900},
    {500, 0.0800, 0.1500, 0.0800}, {505, 0.0800, 0.1600, 0.0650},
    {510, 0.0800, 0.1700, 0.0500}, {515, 0.0800, 0.1850, 0.0300},
    {520, 0.0800, 0.2000, 0.0100}, {525, 0.0800, 0.2500, 0.0100},
    {530, 0.0800, 0.3000, 0.0100}, {535, 0.0800, 0.3500, 0.0050},
    {540, 0.0800, 0.4000, 0.0000}, {545, 0.0900, 0.5000, 0.0000},
    {550, 0.1000, 0.6000, 0.0000}, {555, 0.1000, 0.7000, 0.0000},
```

```
     {560, 0.1000, 0.8000, 0.0000}, {565, 0.1500, 0.7000, 0.0000},
     {570, 0.2000, 0.6000, 0.0000}, {575, 0.2500, 0.5500, 0.0000},
     {580, 0.3000, 0.5000, 0.0000}, {585, 0.3250, 0.4250, 0.0000},
     {590, 0.3005, 0.3500, 0.0000}, {595, 0.3850, 0.2250, 0.0000},
     {600, 0.4002, 0.1000, 0.0000}, {605, 0.5000, 0.0500, 0.0000},
     {610, 0.5008, 0.0000, 0.0000}, {615, 0.5900, 0.0000, 0.0000},
     {620, 0.6000, 0.0000, 0.0000}, {625, 0.5000, 0.0000, 0.0000},
     {630, 0.4000, 0.0000, 0.0000}, {635, 0.2500, 0.0000, 0.0000},
     {640, 0.1000, 0.0000, 0.0000}, {645, 0.0650, 0.0000, 0.0000},
     {650, 0.0003, 0.0000, 0.0000}, {655, 0.0150, 0.0000, 0.0000},
     {660, 0.0000, 0.0000, 0.0000}, {665, 0.0000, 0.0000, 0.0000},
     {670, 0.0000, 0.0000, 0.0000}, {675, 0.0000, 0.0000, 0.0000},
     {680, 0.0000, 0.0000, 0.0000}, {685, 0.0000, 0.0000, 0.0000},
     {690, 0.0000, 0.0000, 0.0000}, {695, 0.0000, 0.0000, 0.0000},
     {700, 0.0000, 0.0000, 0.0000}, {705, 0.0000, 0.0000, 0.0000},
     {710, 0.0000, 0.0000, 0.0000}, {715, 0.0000, 0.0000, 0.0000},
     {720, 0.0000, 0.0000, 0.0000}, {725, 0.0000, 0.0000, 0.0000},
     {730, 0.0000, 0.0000, 0.0000}, {735, 0.0000, 0.0000, 0.0000},
     {740, 0.0000, 0.0000, 0.0000}, {745, 0.0000, 0.0000, 0.0000},
     {750, 0.0000, 0.0000, 0.0000}, {755, 0.0000, 0.0000, 0.0000},
     {760, 0.0000, 0.0000, 0.0000} } ;

int i ;

/* now basically control the switch statement with the observer
 * name that was passed into the function. luckily for us, those
 * names were enumerated in the header file. once we decide which
 * observer/detector we want, a simple for loop fills the arrays
 * passed in (RESPONSE structure).
 */
switch(name){
   case XYZ2:
        for(i=0;i<93;i++){
           response->s1[i] = ciexyz2[i][1] ;
           response->s2[i] = ciexyz2[i][2] ;
           response->s3[i] = ciexyz2[i][3] ;
        }
        break;
   case XYZ10:
        for(i=0;i<93;i++){
             response->s1[i] = ciexyz10[i][1] ;
             response->s2[i] = ciexyz10[i][2] ;
             response->s3[i] = ciexyz10[i][3] ;
        }
        break;
   case STDEV:
        for(i=0;i<93;i++){
             response->s1[i] = stdev[i][1] ;
             response->s2[i] = stdev[i][2] ;
             response->s3[i] = stdev[i][3] ;
        }
        break;
   case SCANNER:
        for(i=0;i<93;i++){
             response->s1[i] = rgb[i][1] ;
             response->s2[i] = rgb[i][2] ;
             response->s3[i] = rgb[i][3] ;
        }
        break;
   case PHOTO:
        for(i=0;i<93;i++){
             response->s1[i] = photo[i][1] ;
             response->s2[i] = photo[i][2] ;
             response->s3[i] = photo[i][3] ;
        }
        break;

   default:
       fprintf(stderr, "that is not a valid observer choice.\n") ;
       exit(-1) ;
}
```

```
}

/*  return_illuminant()
 *
 *  this function is passed a name and an illuminant array. the
 *  function determine which light source is desired, and fills
 *  the array with the necessary values.
 */
void
return_illuminant(int name, double *curve)
{

    /* declare CIE Illuminant D65 */
    double d65[93][2] = {
            {300, 0.000341}, {305, 0.016643}, {310, 0.032945},
            {315, 0.117652}, {320, 0.202360}, {325, 0.286447},
            {330, 0.370535}, {335, 0.385011}, {340, 0.399488},
            {345, 0.424302}, {350, 0.449117}, {355, 0.457750},
            {360, 0.466383}, {365, 0.493637}, {370, 0.520891},
            {375, 0.510323}, {380, 0.499755}, {385, 0.523118},
            {390, 0.546482}, {395, 0.687015}, {400, 0.827549},
            {405, 0.871204}, {410, 0.914860}, {415, 0.924589},
            {420, 0.934318}, {425, 0.900570}, {430, 0.866823},
            {435, 0.957736}, {440, 1.048650}, {445, 1.109360},
            {450, 1.170080}, {455, 1.174100}, {460, 1.178120},
            {465, 1.163360}, {470, 1.148610}, {475, 1.153920},
            {480, 1.159230}, {485, 1.123670}, {490, 1.088110},
            {495, 1.090820}, {500, 1.093540}, {505, 1.085780},
            {510, 1.078020}, {515, 1.062960}, {520, 1.047900},
            {525, 1.062390}, {530, 1.076890}, {535, 1.060470},
            {540, 1.044050}, {545, 1.042250}, {550, 1.040460},
            {555, 1.020230}, {560, 1.000000}, {565, 0.981671},
            {570, 0.963342}, {575, 0.960611}, {580, 0.957880},
            {585, 0.922368}, {590, 0.886856}, {595, 0.893459},
            {600, 0.900062}, {605, 0.898026}, {610, 0.895991},
            {615, 0.886489}, {620, 0.876987}, {625, 0.854936},
            {630, 0.832886}, {635, 0.834939}, {640, 0.836992},
            {645, 0.818630}, {650, 0.800268}, {655, 0.801207},
            {660, 0.802146}, {665, 0.812462}, {670, 0.822778},
            {675, 0.802810}, {680, 0.782842}, {685, 0.740027},
            {690, 0.697213}, {695, 0.706652}, {700, 0.716091},
            {705, 0.729790}, {710, 0.743490}, {715, 0.679765},
            {720, 0.616040}, {725, 0.657448}, {730, 0.698856},
            {735, 0.724863}, {740, 0.750870}, {745, 0.693398},
            {750, 0.635927}, {755, 0.550054}, {760, 0.464182} }  ;


    /* declare CIE Illuminant A */
    double a[93][2] = {
              {300, 0.009305}, {305, 0.011282}, {310, 0.013577},
              {315, 0.016222}, {320, 0.019251}, {325, 0.022698},
              {330, 0.026598}, {335, 0.030986}, {340, 0.035897},
              {345, 0.041365}, {350, 0.047424}, {355, 0.054107},
              {360, 0.061446}, {365, 0.069472}, {370, 0.078214},
              {375, 0.087698}, {380, 0.097951}, {385, 0.108996},
              {390, 0.120853}, {395, 0.133543}, {400, 0.147080},
              {405, 0.161480}, {410, 0.176753}, {415, 0.192907},
              {420, 0.209950}, {425, 0.227883}, {430, 0.246709},
              {435, 0.266425}, {440, 0.287027}, {445, 0.308508},
              {450, 0.330859}, {455, 0.354068}, {460, 0.378121},
              {465, 0.403002}, {470, 0.428693}, {475, 0.455174},
              {480, 0.482423}, {485, 0.510418}, {490, 0.539132},
              {495, 0.568539}, {500, 0.598611}, {505, 0.629320},
              {510, 0.660635}, {515, 0.692525}, {520, 0.724959},
              {525, 0.757903}, {530, 0.791326}, {535, 0.825193},
              {540, 0.859470}, {545, 0.894124}, {550, 0.929120},
              {555, 0.964423}, {560, 1.000000}, {565, 1.035820},
              {570, 1.071840}, {575, 1.108030}, {580, 1.144360},
              {585, 1.180800}, {590, 1.217310}, {595, 1.253860},
              {600, 1.290430}, {605, 1.326970}, {610, 1.363460},
              {615, 1.399880}, {620, 1.436180}, {625, 1.472350},
              {630, 1.508360}, {635, 1.544180}, {640, 1.579790},
```

```
            {645, 1.615160}, {650, 1.650280}, {655, 1.685100},
            {660, 1.719630}, {665, 1.753830}, {670, 1.787690},
            {675, 1.821180}, {680, 1.854290}, {685, 1.887010},
            {690, 1.919310}, {695, 1.951180}, {700, 1.982610},
            {705, 2.013590}, {710, 2.044090}, {715, 2.074110},
            {720, 2.103650}, {725, 2.132680}, {730, 2.161200},
            {735, 2.189200}, {740, 2.216670}, {745, 2.243610},
            {750, 2.270000}, {755, 2.295850}, {760, 2.321150} }  ;

/* declare a fluorescent light source. this is named f1, but
 * is currently a real measured light source */
double  f1[93][2] = {
          {300, 0.013004}, {305, 0.013004}, {310, 0.013004},
          {315, 0.013004}, {320, 0.013004}, {325, 0.013004},
          {330, 0.013004}, {335, 0.013004}, {340, 0.047088},
          {345, 0.081172}, {350, 0.115255}, {355, 0.149339},
          {360, 0.183423}, {365, 0.217507}, {370, 0.251591},
          {375, 0.285675}, {380, 0.314013}, {385, 0.365338},
          {390, 0.382181}, {395, 0.441275}, {400, 0.491692},
          {405, 0.555776}, {410, 0.574264}, {415, 0.868258},
          {420, 1.452787}, {425, 0.889922}, {430, 0.740203},
          {435, 0.740543}, {440, 0.610787}, {445, 0.639766},
          {450, 0.749220}, {455, 0.831226}, {460, 1.002665},
          {465, 0.844213}, {470, 0.872115}, {475, 0.964385},
          {480, 1.155107}, {485, 0.891113}, {490, 1.147054},
          {495, 0.956332}, {500, 1.030341}, {505, 1.288777},
          {510, 1.237963}, {515, 1.247774}, {520, 1.017864},
          {525, 1.139057}, {530, 1.267396}, {535, 5.671185},
          {540, 2.536551}, {545, 1.521409}, {550, 1.012477},
          {555, 1.112289}, {560, 1.000000}, {565, 1.347587},
          {570, 1.349402}, {575, 1.501786}, {580, 1.371689},
          {585, 1.304826}, {590, 1.342256}, {595, 1.665797},
          {600, 2.111439}, {605, 1.683605}, {610, 2.115012},
          {615, 1.868996}, {620, 1.486644}, {625, 1.895764},
          {630, 1.435831}, {635, 1.541882}, {640, 1.513412},
          {645, 1.498214}, {650, 1.442069}, {655, 1.737991},
          {660, 1.548120}, {665, 1.639936}, {670, 1.380593},
          {675, 1.590030}, {680, 1.439403}, {685, 2.077582},
          {690, 1.452787}, {695, 1.344071}, {700, 1.477741},
          {705, 1.206771}, {710, 0.929621}, {715, 0.877899},
          {720, 0.774627}, {725, 0.705722}, {730, 0.628225},
          {735, 0.550729}, {740, 0.473232}, {745, 0.395735},
          {750, 0.318239}, {755, 0.240742}, {760, 0.163245} } ;

/* declare a black, or UV light source. currently the UV light
 * source is a simulated source, essentially a gaussian function
 * that peaks at 380nm.
 */
double  black[93][2] = {
            {300, 0.09657}, {305, 0.12817}, {310, 0.16702},
            {315, 0.21372}, {320, 0.26852}, {325, 0.33127},
            {330, 0.40129}, {335, 0.47731}, {340, 0.55746},
            {345, 0.63928}, {350, 0.71985}, {355, 0.79591},
            {360, 0.86408}, {365, 0.92111}, {370, 0.96414},
            {375, 0.99091}, {380, 1.00000}, {385, 0.99091},
            {390, 0.96414}, {395, 0.92111}, {400, 0.86408},
            {405, 0.79591}, {410, 0.71985}, {415, 0.63928},
            {420, 0.55746}, {425, 0.47731}, {430, 0.40129},
            {435, 0.33127}, {440, 0.26852}, {445, 0.21372},
            {450, 0.16702}, {455, 0.12817}, {460, 0.09657},
            {465, 0.07145}, {470, 0.05190}, {475, 0.03702},
            {480, 0.02593}, {485, 0.01783}, {490, 0.01204},
            {495, 0.00799}, {500, 0.00520}, {505, 0.00332},
            {510, 0.00209}, {515, 0.00129}, {520, 0.00078},
            {525, 0.00046}, {530, 0.00027}, {535, 0.00015},
            {540, 0.00000}, {545, 0.00000}, {550, 0.00000},
            {555, 0.00000}, {560, 0.00000}, {565, 0.00000},
            {570, 0.00000}, {575, 0.00000}, {580, 0.00000},
            {585, 0.00000}, {590, 0.00000}, {595, 0.00000},
            {600, 0.00000}, {605, 0.00000}, {610, 0.00000},
            {615, 0.00000}, {620, 0.00000}, {625, 0.00000},
            {630, 0.00000}, {635, 0.00000}, {640, 0.00000},
```

```
                {645, 0.00000}, {650, 0.00000}, {655, 0.00000},
                {660, 0.00000}, {665, 0.00000}, {670, 0.00000},
                {675, 0.00000}, {680, 0.00000}, {685, 0.00000},
                {690, 0.00000}, {695, 0.00000}, {700, 0.00000},
                {705, 0.00000}, {710, 0.00000}, {715, 0.00000},
                {720, 0.00000}, {725, 0.00000}, {730, 0.00000},
                {735, 0.00000}, {740, 0.00000}, {745, 0.00000},
                {750, 0.00000}, {755, 0.00000}, {760, 0.00000} } ;

    int i ;

    /* now use the name to control the switch function, and
     * fill the illuminant array with the appropriate values.
     */
    switch(name){
      case D65:      /* D65 Dag Nabbit!!!!   */
           for(i=0;i<93;i++){
              curve[i] = d65[i][1] ;
           }
           break ;
      case A:      /* Illuminant A   */
            for(i=0;i<93;i++){
                curve[i] = a[i][1] ;
            }
            break ;
       case F1:      /* Fluorescent   */
            for(i=0;i<93;i++){
                curve[i] = f1[i][1] ;
            }
            break ;
       case BLACK:      /* Black Light   */
            for(i=0;i<93;i++){
                curve[i] = black[i][1] ;
            }
            break ;


      default:
            for(i=0;i<93;i++){
                curve[i] = a[i][1] ;
            }
            break ;

    }


}

/* add_spect()
 *
 * this function makes it easy to add two spectra together.
 * it is passed the two spectra, and a container array that
 * will hold the added spectra.
 */
void
add_spect(double *curve1, double *curve2, double *sum)
{
    int i ;

    for(i=0;i<93;i++){
       sum[i] = curve1[i] + curve2[i] ;
    }
}


/* mult_spect()
 *
 * this function makes it easy to multiply two spectra together.
 * it is passed the two spectra, and a container array that
 * will hold the product of the two spectra.
 */
void
```

```
mult_spect(double *curve1, double *curve2, double *product)
{
    int i ;

     for(i=0;i<93;i++){
        product[i] = curve1[i] * curve2[i] ;
     }
}

/* scale_spect()
 *
 * this function makes it easy to scale a spectra with a scalar.
 * it is passed the spectra, the scalar, and a container array that
 * will hold the product of the two.
 */
void
scale_spect(double *curve, double scale, double *scaled_curve)
{
    int i;

     for(i=0;i<93;i++){
        scaled_curve[i] = scale * curve[i] ;
     }
}

/* area_spect()
 *
 * this function makes it easy to find the area of a spectra
 * using a simple integration type technique. this function
 * does not count the spacing between array units, aka the
 * delta-x unit.
 */
double
area_spect(double *curve)
{
    double area = 0.0 ;
    int i ;

     for(i=0;i<93;i++){
        area = area + curve[i] ;
     }
    return area;
}

/* get_scale()
 *
 * this is a simple function designed to get the tristimulus
 * scaling function (for relative colorimetry). it is passed
 * the luminance, y-hat.
 */
double
get_scale(double *y)
{
    double   k ;

    k = 100 / (5*area_spect(y) ) ;

    return k ;
}


/* spect_to_tristim()
 *
 * this function is passed the observer response functions,
 * as well as a radiance function, and a container array.
 * basically this function uses the detector responses and
 * the radiances to determine some sort of tristimulus values.
 */
void
spect_to_tristim(RESPONSE *detector ,
                double *radiance, double *tristim)
{
```

```
    double  tempcurve[93] ;  /* a temp holding curve */
    double  temptristim ;    /* a temp tristimulus value */
    double  k ;                      /* the scaling function */

    /* if CUR_OBSERVER is less than 3, that means it is a
     * CIE Standard Observer, and we can scale the resulting
     * tristimulus values so that a perfect reflector
     * has a Y value of 100
     */
    if(CUR_OBSERVER < 3){
        k = get_scale(detector->s2) ;
    }
    else{
       k = 1.0 ;
    }

    /* multiply the observer sensitivity curve with the radiance
     * curve.
     */
    mult_spect(detector->s1, radiance, tempcurve) ;

    /* get the area of the product, scaled by k and 5 (nm spacing) */
    temptristim = k * 5 * area_spect(tempcurve) ;
    tristim[0] = temptristim ;

    /* now do the same for the other two observer sensitivities */
    mult_spect(detector->s2, radiance, tempcurve) ;
    tristim = k * 5 * area_spect(tempcurve) ;
    tristim[1] = temptristim ;

    mult_spect(detector->s3, radiance, tempcurve) ;
    tristim = k * 5 * area_spect(tempcurve) ;
    tristim[2] = temptristim ;

}

/* vonKries()
 *
 * this function contains the code necessary to perform a
 * von Kries chromatic adaptation transform on the tristimulus
 * values, before they are displayed. this function is passed
 * the XYZ values of the object, a container for the adapted
 * XYZ values, and the name of the current light source.
 */
void
vonKries(double *XYZ, double *XYZa, int light)
{
    /* the matrix to go from XYZ to LMS. these values are taken
     * from Color Appearance Models, by M.D. Fairchild,
     * Addison-Wesley (1998).
     */
    double    M[3][3] = {    { 0.400200, 0.707600, -0.08080} ,
                             {-0.226300, 1.165300, 0.045700} ,
                             { 0.000000, 0.000000, 0.918200} } ;

    /* the inverse of the above matrix, also taken from Color
     * Appearance Models, by M.D. Fairchild.
     */
    double    invM[3][3] ={         {1.860070, -1.12948, 0.2198980} ,
                             {0.361223, 0.638804, -0.0000007} ,
                             {0.000000, 0.000000, 1.0890900} } ;

    double    XYZw[3] ;      /* XYZ values of the current light source */
    double    LMSw[3] ;      /* LMS values of the current light source */
    double    LMSa[3] ;      /* adapted LMS values of the object */
    double    LMS[3] ;       /* LMS values of the object */
    double    illuminant[93], white[93] ;  /* illuminant spectra */
    double    ill_reflect[93] ;     /* radiance spectra */
    int       i ;
    RESPONSE  *observer ;    /* current observer sensitivities */

    /* basically we are setting the white to be a perfect diffuser.
```

```
     * in this case, we could just use the light source to determine
     * the adaptation state, but we go through the loops, in case
     * some future user might want to measure a real white object
     * to set the adaptation state
     */
    for(i=0;i<93;i++){
       white[i] = 1.0 ;
    }

    /* allocate memory for the observer sensitivity functions */
    observer = (RESPONSE *)malloc(sizeof(RESPONSE)) ;

    /* get the current light source */
    if(light == AMBIENT){
         return_illuminant(CUR_AMBIENT, illuminant) ;
    }
    else if(light == LOCAL2) {
       return_illuminant(CUR_ILLUMINANT2, illuminant) ;
    }
    else{
       return_illuminant(CUR_ILLUMINANT, illuminant) ;
    }

    /* get the current observer functions */
    return_observer(CUR_OBSERVER, observer) ;

    /* multiply the current light source with the reflection spectra
     * of the white sample (PRD in this case).
     */
    mult_spect(illuminant, white, ill_reflect) ;

    /* get the tristimulus values of the white */
    spect_to_tristim(observer, ill_reflect, XYZw) ;

    /* convert the XYZ values of the object into LMS values. */
    LMS[0] = M[0][0]*XYZ[0] + M[0][1]*XYZ[1] + M[0][2]*XYZ[2] ;
    LMS[1] = M[1][0]*XYZ[0] + M[1][1]*XYZ[1] + M[1][2]*XYZ[2] ;
    LMS[2] = M[2][0]*XYZ[0] + M[2][1]*XYZ[1] + M[2][2]*XYZ[2] ;

    /* convert the XYZ values of the white into LMS values. */
    LMSw[0] = M[0][0]*XYZw[0] + M[0][1]*XYZw[1] + M[0][2]*XYZw[2] ;
    LMSw[1] = M[1][0]*XYZw[0] + M[1][1]*XYZw[1] + M[1][2]*XYZw[2] ;
    LMSw[2] = M[2][0]*XYZw[0] + M[2][1]*XYZw[1] + M[2][2]*XYZw[2] ;

    /* do the actual von Kries transform (L/Lw, M/Mw, S/Sw). we
     * need to scale by 100, since the LMS values are between
     * 0 and 1.
     */
    LMSa[0] = 100*LMS[0]/LMSw[0] ;
    LMSa[1] = 100*LMS[1]/LMSw[1] ;
    LMSa[2] = 100*LMS[2]/LMSw[2] ;

    /* convert the adapted LMS values back into XYZ adapted values */
    XYZa[0] = invM[0][0]*LMSa[0] + invM[0][1]*LMSa[1] + invM[0][2]*LMSa[2] ;
    XYZa[1] = invM[1][0]*LMSa[0] + invM[1][1]*LMSa[1] + invM[1][2]*LMSa[2] ;
    XYZa[2] = invM[2][0]*LMSa[0] + invM[2][1]*LMSa[1] + invM[2][2]*LMSa[2] ;

    /* free our allocated memory */
    free(observer) ;

}


/* Fairchild()
 *
 * this fuction handles the calculation of the Fairchild
 * chromatic adaptation. These calculations were taken
 * from the book Color Appearance Models, by M.D. Fairchild.
 */
void
Fairchild(double *XYZ, double *XYZa, int light)
{
```

```
/* the matrix to go from XYZ to LMS */
double     M[3][3] = {    { 0.400200, 0.707600, -0.08080} ,
                          {-0.226300, 1.165300, 0.045700} ,
                          { 0.000000, 0.000000, 0.918200} } ;

/* the matrix to go from LMS to XYZ */
double     invM[3][3] = {        {1.860070, -1.12948, 0.2198980} ,
                          {0.361223, 0.638804, -0.0000007} ,
                          {0.000000, 0.000000, 1.0890900} } ;

double     XYZw[3] ;      /* XYZ of the adapting white */
double     XYZe[3] ;      /* XYZ of the equal energy spectrum */
double     LMSw[3] ;      /* LMS of the adapting white */
double     LMSe[3] ;      /* LMS of the equal energy spectrum */
double     LMSa[3] ;      /* LMS of the adapted stimulus */
double     LMS[3] ;       /* LMS of the stimulus */
double     Yn ;           /* total luminance */
double     le, me, se ;   /* incomplete adaptation terms */
double     pl, pm, ps ;   /* incomplete/coginitve terms */
double     illuminant[93], white[93] ;  /* illuminant spectra */
double     ill_reflect[93] ;            /* white reflectance */
int        i ;
RESPONSE   *observer ;    /* observer sensitivities */

/* once again we use a perfect reflecting diffuser to set the
 * the adaptation state. essentially this is adding an extra
 * step to the calculations, but it will make things easier
 * should a different adaptation point is desired.
 */
for(i=0;i<93;i++){
   white[i] = 1.0 ;
}

/* allocate the necessary memory for the detector curves */
observer = (RESPONSE *)malloc(sizeof(RESPONSE)) ;

/* determine the current illuminant */
if(light == AMBIENT){
     return_illuminant(CUR_AMBIENT, illuminant) ;
}
else if(light == LOCAL2) {
   return_illuminant(CUR_ILLUMINANT2, illuminant) ;
}
/* default to the locall illumination */
else{
   return_illuminant(CUR_ILLUMINANT, illuminant) ;
}

/* get the current observer */
return_observer(CUR_OBSERVER, observer) ;

/* get the product of the light source and the reflecting white */
mult_spect(illuminant,white, ill_reflect) ;

/* calculate the XYZ tristimulus values of the illuminant and white */
spect_to_tristim(observer, ill_reflect, XYZw) ;

/* get the XYZ values of the equal energy spectrum. we can use
 * the white[] variable, since we set that to 1.0 above.
 */
spect_to_tristim(observer, white, XYZe) ;

/* calculate the LMS values for the stimulus */
LMS[0] = M[0][0]*XYZ[0] + M[0][1]*XYZ[1] + M[0][2]*XYZ[2] ;
LMS[1] = M[1][0]*XYZ[0] + M[1][1]*XYZ[1] + M[1][2]*XYZ[2] ;
LMS[2] = M[2][0]*XYZ[0] + M[2][1]*XYZ[1] + M[2][2]*XYZ[2] ;

/* calculate the LMS values for the adapting white */
LMSw[0] = M[0][0]*XYZw[0] + M[0][1]*XYZw[1] + M[0][2]*XYZw[2] ;
LMSw[1] = M[1][0]*XYZw[0] + M[1][1]*XYZw[1] + M[1][2]*XYZw[2] ;
LMSw[2] = M[2][0]*XYZw[0] + M[2][1]*XYZw[1] + M[2][2]*XYZw[2] ;
```

```
    /* calculate the LMS values for the equal energy spectrum */
    LMSe[0] = M[0][0]*XYZe[0] + M[0][1]*XYZe[1] + M[0][2]*XYZe[2] ;
    LMSe[1] = M[1][0]*XYZe[0] + M[1][1]*XYZe[1] + M[1][2]*XYZe[2] ;
    LMSe[2] = M[2][0]*XYZe[0] + M[2][1]*XYZe[1] + M[2][2]*XYZe[2] ;

    /* determine how far away from the equal energy spectrum the
     * current light source is. the further away we are, the less
     * complete the total adaptation is. another way of saying that
     * is the more incomplete the adaptation is.
     */
    le = (3*(LMSw[0]/LMSe[0]))/
              ( LMSw[0]/LMSe[0] + LMSw[1]/LMSe[1] + LMSw[2]/LMSe[2] ) ;

    me = (3*(LMSw[1]/LMSe[1]))/
              ( LMSw[0]/LMSe[0] + LMSw[1]/LMSe[1] + LMSw[2]/LMSe[2] ) ;

    se = (3*(LMSw[2]/LMSe[2]))/
              ( LMSw[0]/LMSe[0] + LMSw[1]/LMSe[1] + LMSw[2]/LMSe[2] ) ;

    /* determine the luminance (in a hacking sort of way). basically
     * the more light there is, the more complete the adaptation.
     */
    Yn = 1.0 ;

    if(light == AMBIENT && AmbScale > 1){
       Yn = pow((10*AmbScale), (1.0/3.0)) ;
    }
    else if(light == LOCAL && DiffScale > 1){
       Yn = pow((10*DiffScale), (1.0/3.0)) ;
    }
    else {
       Yn = pow((10*DiffScale2), (1.0/3.0)) ;
    }


    /* now determine the extent of incomplete adaptation.
     * the smaller the p_ value, the less complete the
     * adaptation. if adaptation were 100% complete, like
     * in von Kries, the p_ values would all equal 1.0.
     */
    pl = (1 + Yn + le) / (1 + Yn + (1.0/le)) ;
    pm = (1 + Yn + me) / (1 + Yn + (1.0/me)) ;
    ps = (1 + Yn + se) / (1 + Yn + (1.0/se)) ;

    /* now do the von Kries type chromatic adaptation scaling */
    LMSa[0] = 100*LMS[0]*pl/LMSw[0] ;
    LMSa[1] = 100*LMS[1]*pm/LMSw[1] ;
    LMSa[2] = 100*LMS[2]*ps/LMSw[2] ;

    /* convert the adapted LMS values into XYZ values */
    XYZa[0] = invM[0][0]*LMSa[0] + invM[0][1]*LMSa[1] + invM[0][2]*LMSa[2] ;
    XYZa[1] = invM[1][0]*LMSa[0] + invM[1][1]*LMSa[1] + invM[1][2]*LMSa[2] ;
    XYZa[2] = invM[2][0]*LMSa[0] + invM[2][1]*LMSa[1] + invM[2][2]*LMSa[2] ;

    /* free the memory */
    free(observer) ;

}

/* scan_to_rgb()
 *
 * this function takes the tristimulus values of the scanner and
 * does a non-linear gamma correction, as a real scanner might do.
 * the result is gamma corrected RGB values.
 *
 */
void
scan_to_rgb(double *trist, double *rgb)
{
/*******************************************************************
*       uncomment out this section for a colorimetrically calibrated   *
*       scanner function.                                         *
```

```
*                                                                           *
*
*       double      scan[3][3] = {                                          *
*                            { 0.691740,  0.236361,  0.1975240},            *
*                            { 0.253273,  0.959515, -0.0928043},            *
*                            {-0.247167, -0.177031,  1.6734700} } ;         *
*                                                                           *
****************************************************************************/

/* perform the non-linear "gamma" correction. */

     rgb[0] =  ((pow(trist[0],(1.0/1.534))+0.0)/1.035) ;
     rgb[1] = ((pow(trist[1],(1.0/1.435))+0.0)/1.047) ;
     rgb[2] =((pow(trist[2],(1.0/1.393))+0.0)/1.053) ;

/*****************************************************************************
*    uncomment out this section for the matrix multipication...
*
*    xyz[0] = scan[0][0]*trist[0] + scan[0][1]*trist[1] + scan[0][2]*trist[2] ;
*    xyz[1] = scan[1][0]*trist[0] + scan[1][1]*trist[1] + scan[1][2]*trist[2] ;
*    xyz[2] = scan[2][0]*trist[0] + scan[2][1]*trist[1] + scan[2][2]*trist[2] ;
*
*****************************************************************************/

}

/* photo_to_xyz()
 *
 * this is a largely ignored function. it can be used if the user
 * wants colorimetrically calibrated results...
 *
 */
void
photo_to_xyz(double *trist, double *xyz)
{
    double    photo[3][3] = {
                    { 0.539577,  0.179218, -0.165438},
                    { 0.175847,  0.764411, -0.424421},
                    {-0.104830, -0.317030,  0.930255} } ;

     xyz[0] = photo[0][0]*trist[0] + photo[0][1]*trist[1] + photo[0][2]*trist[2] ;
     xyz[1] = photo[1][0]*trist[0] + photo[1][1]*trist[1] + photo[1][2]*trist[2] ;
     xyz[2] = photo[2][0]*trist[0] + photo[2][1]*trist[1] + photo[2][2]*trist[2] ;


}


/* xyz_to_rgb()
 *
 * this function performs the XYZ to digital RGB conversions so that the
 * calculated XYZ tristimulus values can be accurately displayed. the
 * default monitor is a standard SGI monitor. if the end-user wants
 * decent color, they are highly encourage to calibrate their own monitor
 * and to place the results of the calibration/characterization into this
 * fuction.
 *
 */
void
xyz_to_rgb(double *xyz, double *rgb)
{
    /* default 3x3 XYZ-RGB matrix. Note that this matrix it technically
     * only valid for one monitor. it does do a reasonable job for most
     * other (non-microsoft?) monitors. replace this matrix with your
     * own, if you are feeling frisky.
     */
    double    xyz_rgb[3][3] = {
                    { 0.0342600, -0.016330, -0.0053650} ,
                    {-0.0102600,  0.019110,  0.0002522} ,
                    { 0.0005290, -0.001811,  0.0086700} } ;

    /* our RGB, and digital RGB variables */
```

```
    double      red, green, blue ;
    double      d_red, d_green, d_blue ;

    /* initialize the variables to 0.0 */
    red = green = blue = 0.0 ;
    d_red = d_green = d_blue = 0.0 ;

    /* use the xyz to rgb matrix calculation to get RGB counts */
    red =    xyz_rgb[0][0]*xyz[0]  + xyz_rgb[0][1]*xyz[1]  + xyz_rgb[0][2]*xyz[2]  ;
    green = xyz_rgb[1][0]*xyz[0]  + xyz_rgb[1][1]*xyz[1]  + xyz_rgb[1][2]*xyz[2]  ;
    blue =   xyz_rgb[2][0]*xyz[0]  + xyz_rgb[2][1]*xyz[1]  + xyz_rgb[2][2]*xyz[2]  ;

    /* do some high-tech gamut mapping (otherwise known as clipping in
     * RGB space, of all spaces...). this is an area of potential
     * improvement.
     */
    if(red < 0.0)
        red = 0.0;
    if(green < 0.0)
        green = 0.0;
    if(blue < 0.0)
        blue = 0.0;

    /*
     * The nonlinear conversion between RGB to the calibrated digital
     * counts. once again, this is only truly valid for one monitor,
     * sitting in our darkend lab. if you are really interested in
     * accuracy, replace this with your own Gamma-Offset-Gain numbers.
     */

/*  the offset, unless you are really using your own calibrated
 *  monitor, causes a boost in all the colors...so ignore it for now.
 *
 *  d_red =   ((pow(red,(1.0/1.534))+0.036)/1.035) ;
 *  d_blue =  ((pow(blue,(1.0/1.435))+0.045)/1.047) ;
 *  d_green =((pow(green,(1.0/1.393))+0.049)/1.053) ;
 */
    d_red =   ((pow(red,(1.0/1.534))+0.0)/1.035) ;
    d_green =((pow(green,(1.0/1.393))+0.0)/1.053) ;
    d_blue =  ((pow(blue,(1.0/1.435))+0.0)/1.047) ;

    /* once again, some more high-tech gamut mapping */
    if(d_red <= 0.0)
        d_red = 0.0 ;
    if(d_red >= 1.0)
        d_red = 1.0 ;

    if(d_blue <= 0.0)
        d_blue = 0.0 ;
    if(d_blue >= 1.0)
        d_blue = 1.0 ;

    if(d_green <= 0.0)
        d_green = 0.0 ;
    if(d_green >= 1.0)
        d_green = 1.0 ;

    /* finally, set our rgb array that was passed into the
     * function equal to these new digital counts. when the
     * material properties of OpenGL are set to these digital
     * counts, the result will be a display that "closely"
     * resembles what an average observer would see if they
     * looked at an object, with those given spectral properties.
     * basically, this is a little hack, since it is impossible
     * to display full spectral information anyway.
     */
    rgb[0] = d_red ;
    rgb[1] = d_green ;
    rgb[2] = d_blue ;

}
```

```
/* chromatic_adaptation()
 *
 * this is a simple function that determines the current choice
 * of chromatic adaptation transforms, and then calls the correct
 * function.
 */
void
chromatic_adaptation(double *xyz, double *XYZa, int light)
{

     /* first we need to make sure we are dealing with a human
      * observer...as most other systems have their own special
      * brand of "white-point adjustment.
      */
     if(CUR_OBSERVER < 3){
          switch(CUR_ADAPT){
            /* if the user does not want a chromatic adaptation
             * transform, then the adapted tristimulus values are
             * equal to the non-adapted.
             */
            case(NONE):
                XYZa[0] = xyz[0] ;
                XYZa[1] = xyz[1] ;
                XYZa[2] = xyz[2] ;
                break;

            /* otherwise call the right transform */
            case(VONKRIES):
                vonKries(xyz, XYZa, light) ;
                break;

            case(FAIRCHILD):
                Fairchild(xyz, XYZa, light) ;
                break;
          }
     }
     /* if the observer isn't "human" then the adapted values
      * are equal to the original
      */
     else{
        XYZa[0] = xyz[0] ;
         XYZa[1] = xyz[1] ;
         XYZa[2] = xyz[2] ;
     }
}

/* calc_fluoresence()
 *
 * one of the huge fundamental weaknesses with the standard RGB
 * color model used in most computer graphics applications,
 * including OpenGL, is the inability to demonstrate fluoresence.
 * since we do all the color calculations ourselves, and since
 * we maintain full spectral informatin including excitation and
 * emission specta, we are quite capable of doing fluorescent
 * calculations. this function is passed the excitation, emission,
 * and radiance spectra, as well as the name of the current light
 * source.
 *
 */
void
calc_fluorescence(double *excite, double *emit, double *radiance, int light)
{
    double     equal[93] ;           /* the equal energy spectrum */
    double     illumination[93] ;    /* current illuminant */
    double     scale_illum[93] ;     /* the scaled illuminant */
    double     illum_equal[93] ;     /* product of equal and excite */
    double     illum_excite[93] ;    /* product of illumination + excite */
    double     scale ;               /* the light scalar */
    double     N ;                   /* the "amount" of fluoresence */
    int        i ;
```

```
    /* if we are intrested in the ambient light
     * go out and get the the illuminant, and get
     * the scale
     */
    if(light == AMBIENT) {
         return_illuminant(CUR_AMBIENT, illumination) ;
       scale = AmbScale ;
    }
    /* perhaps we are interested in the local 2 illumination... */
    else if(light == LOCAL2) {
         return_illuminant(CUR_LOCAL2, illumination) ;
       scale = DiffScale2 ;
    }
    /* otherwise, use the local illuminant*/
    else{
         return_illuminant(CUR_ILLUMINANT, illumination) ;
       scale = DiffScale ;
    }


    /* scale the light by the scale factor */

    scale_spect(illumination, scale, scale_illum) ;


    /* set the radiance = 0, and define the equal energy spectrum  */

    for(i=0;i<93;i++){
       equal[i] = 1.0 ;
       radiance[i] = 0.0 ;
    }


    /* multiply the scaled illuminant by the excitation spectra  */

    mult_spect(scale_illum, excite, illum_excite) ;

    /* multiply the equal enegry spectra by the excitation spectra */

    mult_spect(equal, excite, illum_equal) ;


    /* determine the ratio between the areas of the equal energy
     * (maximum energy at all bands) and the current illuminant
     * this determines the number of quanta absorbed.
     */
    N = area_spect(illum_excite) / area_spect(illum_equal) ;


    /* the radiant output caused by fluoresence is then the
     * emission curve, scaled by the number of quanta absorbed.
     */
    for(i=0;i<93;i++){
       radiance[i] = emit[i] * N ;
    }
}

/* trist_to_rgb()
 *
 * this function simply determines which function
 * converts the tristimulus values into digital RGB
 * values.
 */
void
trist_to_rgb(double *trist, double *rgb)
{
    /* use the current observer/detector to control the switch */
    switch(CUR_OBSERVER){
       /* if it is "human" use the xyz_to_rgb() function */
       case XYZ2:
            xyz_to_rgb(trist, rgb) ;
```

```
            break ;
        case XYZ10:
            xyz_to_rgb(trist, rgb) ;
            break ;
        case STDEV:
            xyz_to_rgb(trist, rgb) ;
            break ;

        /* if it isn't human use the scan_to_rgb() function. if
         * a new detector is chosen, rather than the default ones
         * offered, it is up to the user to figure out the
         * transform. luckily, both the scanner and photographic
         * film have RGB sensitivities. usually a simple linear
         * regression provides a good estimation, otherwise.
         */
        case SCANNER:
            scan_to_rgb(trist, rgb) ;
            break ;
        case PHOTO:
            scan_to_rgb(trist, rgb) ;
            break ;

    }
}

/* get_background()
 *
 * this is just a relatively simple function used to get the
 * digital RGB counts of a non-selective 40% grey background.
 * basically this was just a hack to allow the background to
 * better illustrate the color of the light source.
 *
 */
void
get_background(double *amb_rgb, double *diff_rgb, int light)
{
    RESPONSE    *observerp ;        /* the detector response curves */
    double      ambient[93] ;   /* the ambient illuminant */
    double      local[93] ;         /* the local illuminant */
    double      scale_ambient[93] ;    /* the scaled ambient */
    double      scale_local[93] ;      /* the scaled local */
    double     XYZamb[3], XYZloc[3] ;   /* the XYZ of ambient and local */
    double     sXYZamb[3], sXYZloc[3] ; /* the temp scaled XYZ */
    double     aXYZamb[3], aXYZloc[3] ; /* the adapted XYZ */
    int        i ;


    /* first we need to allocate memory for the observer functions */
    observerp = (RESPONSE *)malloc(sizeof(RESPONSE))   ;

    /* now we need to get the ambient illuminant */
    return_illuminant(CUR_AMBIENT, ambient) ;

    /* we need to determine which light source we are dealing
     * with, and get the correct information */
    if(light == LOCAL2) {
        return_illuminant(CUR_ILLUMINANT2, local) ;
    }
    else {
       return_illuminant(CUR_ILLUMINANT, local) ;
    }

    /* get the current observer/detector responses */
    return_observer(CUR_OBSERVER, observerp) ;

    /* scale our ambient and local illuminants to 0.4, to represent
     * a non-selective grey with a 40% reflectance.
     */
    scale_spect(ambient, 0.4, scale_ambient) ;

    scale_spect(local, 0.4, scale_local) ;
```

```
    /* get the tristimulus values of the scaled spectra */
    spect_to_tristim(observerp, scale_ambient, sXYZamb) ;
    spect_to_tristim(observerp, scale_local, sXYZloc) ;

    /* get the tristimulus values of the non-scaled illuminants */
    spect_to_tristim(observerp, ambient, XYZamb) ;
    spect_to_tristim(observerp, local, XYZloc) ;

    /* perform the chromatic adaptation transforms, if necessary */
    chromatic_adaptation(sXYZamb, aXYZamb, AMBIENT) ;
    chromatic_adaptation(sXYZloc, aXYZloc, light) ;

    /* this is where the two non human detectors do some sort of
     * auto white-point adjustment. basically the tristimulus
     * values are scaled by the tristimulus values of the illuminant.
     */
    if(CUR_OBSERVER > 2){
       for(i=0;i<3;i++){
             if(XYZloc[i] > 0.0) aXYZloc[i] = aXYZloc[i]/XYZloc[i] ;
             if(XYZamb[i] > 0.0) aXYZamb[i] = aXYZamb[i]/XYZamb[i] ;
       }
    }

    /* convert the adapted xyz values into rgb values */
    trist_to_rgb(aXYZloc, diff_rgb ) ;
    trist_to_rgb(aXYZamb, amb_rgb ) ;

    /* free the memory */
    free(observerp) ;


}

/* get_rgb()
 *
 * this function is called to convert the spectral properties into
 * digital rgb counts, to feed into the OpenGL material properties.
 *
 * it expects to receive the digital rgb arrays for both the regular
 * and metameric patches. It also receives the desired light with
 * which to calculate the rgb values. This can be either AMBIENT or
 * local. This function determines the current observer from the state
 * variable CUR_OBSERVER, and calculates the appropriate RGB values.
 *
 * basically this function is needed to control the interactive
 * spectral display. this is necessary, since the average computer
 * graphics display in not capable of reproducing full spectra.
 * instead, we need a way to reproduce the spectra so that the
 * reproduction envokes the same response from the detector, as
 * the real spectra would have. this is why we need the detector
 * functions, as they all sample the spectra down into just 3 numbers,
 * which can (and will) be reproduced on a display.
 *
 */
void
get_rgb(double rgb[][3], double met_rgb[][3], int light)
{
    double     illuminant[93] ;      /* the current illuminant */
    RESPONSE   *observerp ;          /* the current detector  */
    double     xyz[24][3] ;          /* xyz tristimulus values */
    double     trist[24][3] ;        /* other tristimulus values */
    double     temp_reflect[93] ;    /* temp reflectance curve */
    double     temp_excite[93] ;     /* temp excitation */
    double     temp_emit[93] ;              /* temp emission */
    double     temp_fluor[93] ;      /* temp fluoresence curve */
    double     ill_reflect[93] ;     /* product of illuminant+reflectance */
    double       total_reflect[93] ; /* total reflectance */
    double       temp_trist[3] ;            /* temp tristimulus values */
    double       XYZa[3] ;           /* adapted tristimulus values */
    double     XYZillum[3] ;         /* illuminant tristimulus values */
    double     temp_rgb[3] ;         /* temp digital rgb values */
    int        i, j ;
```

```
/* allocate memory for the detectors */
observerp = (RESPONSE *)malloc(sizeof(RESPONSE))   ;


/* determine which light source we are dealing with */
if(light == AMBIENT){
   return_illuminant(CUR_AMBIENT, illuminant) ;
}
else if(light == LOCAL2){
   return_illuminant(CUR_ILLUMINANT2, illuminant) ;
}
else {
     return_illuminant(CUR_ILLUMINANT, illuminant) ;
}

/* get the current observer/detector responses */
return_observer(CUR_OBSERVER, observerp) ;


/* get the tristimulus values of the illuminant */
spect_to_tristim(observerp, illuminant, XYZillum) ;

/* go through the loop and calculated digital RGB values for
 * each of the spectral material files. these RGB values will
 * be used as OpenGL material properties, essentially forcing
 * OpenGL to use our color calculations.
 *
for(i=0;i<NUM_FILES;i++){

   /* set our temporary variables from the global variables */
   for(j=0;j<93;j++){
        temp_reflect[j] = reflect[i][j] ;
        temp_excite[j] = excite[i][j] ;
        temp_emit[j] = emit[i][j] ;
        temp_fluor[i] = 0.0 ;
   }


   /* if the current material has fluorescent properties, calculate
    * the effect of the fluorescence.
    */
   if(FLUORESCENT[i] == TRUE){
        calc_fluorescence(temp_excite, temp_emit, temp_fluor, light) ;
   )

   /* multiply the illuminant by the diffuse reflection curve */
   mult_spect(illuminant, temp_reflect, ill_reflect) ;

   /* add the fluorescent contribution if it exists */
   add_spect(ill_reflect, temp_fluor, total_reflect) ;


   /* if there is fluorescence, use the total_reflect spectrum
    * to determine the tristimulus values, otherwise just use
    * the ill_reflect spectrum.
    */
   if(FLUORESCENT[i] == TRUE){
         spect_to_tristim(observerp, total_reflect, temp_trist) ;
   }
   else{
         spect_to_tristim(observerp, ill_reflect, temp_trist) ;
    }

   /* if the observer is human, determine (if any) the effects
    * of chromatic adaptation.
    */
   chromatic_adaptation(temp_trist, XYZa, light) ;

   /* if the observer is not human, so some fancy white point
    * adjustment
```

```
     */
    if(CUR_OBSERVER > 2){
        if(XYZillum[0] > 0.0) XYZa[0] = XYZa[0]/XYZillum[0] ;
        if(XYZillum[1] > 0.0) XYZa[1] = XYZa[1]/XYZillum[1] ;
        if(XYZillum[2] > 0.0) XYZa[2] = XYZa[2]/XYZillum[2] ;
    }

    /* convert the adapted xyz values into rgb values */
     trist_to_rgb(XYZa, temp_rgb ) ;

    /* now set the rgb array equal to the temp digital rgb counts */
    for(j=0;j<3;j++){
        rgb[i][j] = temp_rgb[j] ;
    }
}


/* now go through the whole dang process for the metameric values */
for(i=0;i<NUM_FILES;i++){

    /* first let's check to see if we need to bother...
     * if the sample is not metameric, then don't calculate
     * anything....
     */

    if(METAMERIC[i] == TRUE){

        /* set our temp variables equal to the global metameric
         * variables.
         */
         for(j=0;j<93;j++){
              temp_reflect[j] = metamer[i][j] ;
              temp_excite[j] = met_excite[i][j] ;
              temp_emit[j] = met_emit[i][j] ;
              temp_fluor[i] = 0.0 ;
         }

        /* calculate fluorescence, if necessary */
        if(MET_FLUORESCENT[i] == TRUE){
              calc_fluorescence(temp_excite, temp_emit, temp_fluor, light) ;
         }

        /* multiply the illuminant with the reflectance (diffuse) */
        mult_spect(illuminant, temp_reflect, ill_reflect) ;

        /* add in the fluorescent contribution, if it exists */
        add_spect(ill_reflect, temp_fluor, total_reflect) ;

        /* calculate the tristimulus values, accounting for
         * fluorescent contributions if necessary.
         */
        if(MET_FLUORESCENT[i] == TRUE){
              spect_to_tristim(observerp, total_reflect, temp_trist) ;
         }
         else{
              spect_to_tristim(observerp, ill_reflect, temp_trist) ;
        }

        /* do the chromatic adaptation transform */
        chromatic_adaptation(temp_trist, XYZa, light) ;

        /* if the detector is not human, than do the white
         * point adjustment.
         */
        if(CUR_OBSERVER > 2){
            if(XYZillum[0] > 0.0) XYZa[0] = XYZa[0]/XYZillum[0] ;
            if(XYZillum[1] > 0.0) XYZa[1] = XYZa[1]/XYZillum[1] ;
            if(XYZillum[2] > 0.0) XYZa[2] = XYZa[2]/XYZillum[2] ;
        }

        /* transform the adapted tristimulus values into
         * digital RGB counts.
```

```
			 */
			 trist_to_rgb(XYZa, temp_rgb ) ;

			/* set the metameric rgb array equal to the temp rgb values */
			 for(j=0;j<3;j++){
				met_rgb[i][j] = temp_rgb[j] ;
			 }
		}

		/* if the current sample does not have a metameric
		 * partner, then simply set the met_rgb = rgb.
		 */
		else{
			for(j=0;j<3;j++){
				met_rgb[i][j] = rgb[i][j] ;
			 }
		}

	}

	/* do not forget to free our allocated memory */
	free(observerp) ;
}


/*  get_specular_rgb()
 *
 *  essentially the same function as get_rgb(), except we are now
 *  dealing with the specular component.
 *
 */
void
get_specular_rgb(double spec_rgb[][3], double met_spec_rgb[][3], int light)
{
	double		illuminant[93] ;	/* the current illuminant */
	RESPONSE	*observerp ;		/* the current detector   */
	double		temp_specular[93] ;	   /* temporary specular curve */
	double		ill_reflect[93] ;	/* product of illuminant+reflect */
	double		total_reflect[93] ;	/* total reflectance (specular) */
	double		temp_trist[3] ;			/* temp tristimulus values */
	double		XYZa[3] ;		/* adapted tristimulus values */
	double		XYZillum[3] ;		/* illuminant tristim */
	double		temp_rgb[3] ;		/* temp digital rgb counts */
	int		i, j ;


	/* allocate memory */
	observerp = (RESPONSE *)malloc(sizeof(RESPONSE))   ;

	/* determine what light source we need */
	if(light == LOCAL2) {
		return_illuminant(CUR_ILLUMINANT2, illuminant) ;
	}
	else {
	   return_illuminant(CUR_ILLUMINANT, illuminant) ;
	}


	/* get the observer response functions */
	return_observer(CUR_OBSERVER, observerp) ;

	/* calculate tristimulus values of the light source */
	spect_to_tristim(observerp, illuminant, XYZillum) ;



	/* loop through the total number of spectral material files */
	for(i=0;i<NUM_FILES;i++){

	   /* set the temp specular curve equal the the global */
	   for(j=0;j<93;j++){
		temp_specular[j] = specular[i][j] ;
```

```
    }

    /* multiply the illuminant and specular reflectance curves */
    mult_spect(illuminant, temp_specular, ill_reflect) ;


    /* if the material has a specular curve, then use that to
     * determine the tristimulus values.
     */
    if(SPECULAR[i] == TRUE){
          spect_to_tristim(observerp, ill_reflect, temp_trist) ;
    }

    /* if the material does not have a supplied specular curve,
     * then we assume the material is diffuse, and all specular
     * highlights should be the color of the light source. so
     * we get the tristimulus values of the light source.
     */
    else{
          spect_to_tristim(observerp, illuminant, temp_trist) ;
     }


    /* if the observer is human, determine (if any) the effects
     * of chromatic adaptation.
     */
    chromatic_adaptation(temp_trist, XYZa, light) ;

    /* if the observer is not human, do the white point stuff */
    if(CUR_OBSERVER > 2){
        if(XYZillum[0] > 0.0) XYZa[0] = XYZa[0]/XYZillum[0] ;
        if(XYZillum[1] > 0.0) XYZa[1] = XYZa[1]/XYZillum[1] ;
        if(XYZillum[2] > 0.0) XYZa[2] = XYZa[2]/XYZillum[2] ;
    }


    /* convert the adapted xyz values into rgb values */
     trist_to_rgb(XYZa, temp_rgb ) ;


    /* set the passed in rgb values equal to the temporary rgb */
    for(j=0;j<3;j++){
        spec_rgb[i][j] = temp_rgb[j] ;
    }
}

/* now go through the whole dang process for the metameric values */
for(i=0;i<NUM_FILES;i++){

    /* first let's check to see if we need to bother...
     * if the sample is not metameric, then don't calculate
     * anything....
     */
    if(METAMERIC[i] == TRUE){

        /* set our temp specular curve equal to the global curve */
         for(j=0;j<93;j++){
              temp_specular[j] = met_specular[i][j] ;
         }

        /* multiply the illuminant and the reflectance curve */
        mult_spect(illuminant, temp_specular, ill_reflect) ;

        /* get the tristimulus values either based upon the given
         * specular curves, or the light source itself.
         */
        if(SPECULAR[i] == TRUE){
              spect_to_tristim(observerp, ill_reflect, temp_trist) ;
         }
         else{
              spect_to_tristim(observerp, illuminant, temp_trist) ;
        }
```

```
        /* determine the effects of chromatic adaptation */
        chromatic_adaptation(temp_trist, XYZa, light) ;

        /* if the detector is not human, auto-white point adjust */
        if(CUR_OBSERVER > 2){
            if(XYZillum[0] > 0.0) XYZa[0] = XYZa[0]/XYZillum[0] ;
            if(XYZillum[1] > 0.0) XYZa[1] = XYZa[1]/XYZillum[1] ;
            if(XYZillum[2] > 0.0) XYZa[2] = XYZa[2]/XYZillum[2] ;
        }

        /* get the digital rgb values */
         trist_to_rgb(XYZa, temp_rgb ) ;

        /* set our real variables equal to the temporary ones */
         for(j=0;j<3;j++){
              met_spec_rgb[i][j] = temp_rgb[j] ;
         }
    }

    /* if the current sample does not have a metameric
     * partner, then simply set the met_rgb = rgb.
     */
    else{
         for(j=0;j<3;j++){
              met_spec_rgb[i][j] = spec_rgb[i][j] ;
         }
    }

  }

}
```

shapes.c

```
/*
 * shapes.c
 *
 * Written By: Garrett M. Johnson
 *
 * Geometric shapes file for the full spectral rendering package
 * using OpenGL, and Motif.
 *
 * This file contains code for generating simple geometric shapes
 * such as spheres, cones, rectangles, and toruses. It is designed
 * just to make any attempt to draw these objects much nicer.
 *
 *
 * Last revised: 07/27/98
 *
 */

#include <stdio.h>
#include <math.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include "spectral.h"
#include "glm.h"

/* make sure we have a definition of π (pi) */
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

/* our GLU quadric object */
static     GLUquadricObj  *quadObj ;

/* some of the following code was borrowed from the glut
 * (GL utitity toolkit) written by Mark J. Kilgard.
 */
#define QUAD_OBJ_INIT() { if(!quadObj) initQuadObj(); }

/* initQuadObj()
 *
 * this function just makes sure the GLU quadric is
 * initialized, so we can play with it.
 */
static void
initQuadObj(void)
{
  quadObj = gluNewQuadric();
  if (!quadObj){
       fprintf(stderr, "out of memory, bozo") ;
       exit(-1) ;
   }

}

/* sphere()
 *
 * this function draws a simple sphere. It is passed a radius
 * (overall size), total number of vertical slices, and total
 * number of horizontal stacks. the more slices and stacks, the
 * smoother the sphere will appear (more triangles).
 */
void
sphere(GLdouble radius, GLint slices, GLint stacks)
```

```
{
    /* make sure the quadric object is initialized. */
    QUAD_OBJ_INIT() ;

    /* we want a filled in sphere */
    gluQuadricDrawStyle(quadObj, GLU_FILL);

    /* we want smooth shaded normals */
    gluQuadricNormals(quadObj, GLU_SMOOTH);

    /* we want texture coordinates, just in case */
    gluQuadricTexture(quadObj, GL_TRUE) ;

    /* now draw that baby, using the glu library standard sphere call */
    gluSphere(quadObj, radius, slices, stacks);
}

/* cone()
 *
 * this function draws a simple cone. it is passed a base radius, the
 * height, and the number of vertical and horizontal slices.
 */
void
cone(GLdouble base, GLdouble height,
  GLint slices, GLint stacks)
{
    /* make sure our quadric is initialized */
    QUAD_OBJ_INIT() ;

    /* fill in the cone */
    gluQuadricDrawStyle(quadObj, GLU_FILL);

    /* smooth shading */
    gluQuadricNormals(quadObj, GLU_SMOOTH);

    /* now call the glu cylinder command, giving the base
     * radius, and a radius of 0.0 (for the point of the
     * cone
     */
    gluCylinder(quadObj, base, 0.0, height, slices, stacks);
}


/* Torus()
 *
 * this function draws a torus, aka a doughnut, aka a donut.
 * it is passed an inner radias, an outer radius, and a number
 * of vertical and horizontal slices. This function was borrowed
 * from the glut library.
 */
void
Torus(GLfloat r, GLfloat R, GLint nsides, GLint rings)
{
    int i, j;
    GLfloat theta, phi, thetal;
    GLfloat cosTheta, sinTheta;
    GLfloat cosThetal, sinThetal;
    GLfloat ringDelta, sideDelta;
    GLfloat cosPhi, sinPhi, dist;


    ringDelta = 2.0 * M_PI / (GLfloat)rings;
    sideDelta = 2.0 * M_PI / (GLfloat)nsides;

    theta = 0.0;
    cosTheta = 1.0;
    sinTheta = 0.0;
    for (i = rings   1; i >= 0; i--) {
        thetal = theta + ringDelta;
        cosThetal = cos(thetal);
        sinThetal = sin(thetal);
        glBegin(GL_QUAD_STRIP);
```

```
        phi = 0.0;
        for (j = nsides; j >= 0; j--) {

            phi += sideDelta;
            cosPhi = cos(phi);
            sinPhi = sin(phi);
            dist = R + r * cosPhi;

            glNormal3f(cosTheta1 * cosPhi, -sinTheta1 * cosPhi, sinPhi);
            glVertex3f(cosTheta1 * dist, -sinTheta1 * dist, r * sinPhi);
            glNormal3f(cosTheta * cosPhi, -sinTheta * cosPhi, sinPhi);
            glVertex3f(cosTheta * dist, -sinTheta * dist,  r * sinPhi);
        }
        glEnd();
        theta = theta1;
        cosTheta = cosTheta1;
        sinTheta = sinTheta1;
    }
}
```

spect-write.c

```
/*
 * spect-write.c
 *
 * Written By: Garrett M. Johnson
 *
 * the image writing file for the full spectral rendering package
 * using OpenGL, and Motif.
 *
 * This file contains all the functions necessary for writing out
 * the full spectral images. These images are written out using
 * the NCSA HDF image file library. I guess this is the cool part
 * of the spectral rendering library. The program defaults to writing
 * out in the style of the interactive viewer. Basically, this function
 * calls the same draw() function as that called in the spect-gl.c
 * file. If a different scene is desired, the drawing function should
 * be altered.
 *
 * Last revised: 07/27/98
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>
#include <X11/Intrinsic.h>
#include <GL/glx.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include "/remote/user2/users/garrett/tmp/hdf/include/hdf.h"
#include "/remote/user2/users/garrett/tmp/hdf/include/mfhdf.h"
#include "pbutil.h"
#include "spectral.h"


static Pixmap XPix = 0  ;    /* declare an Xpixmap to render in */
static GLXPixmap  GPix = 0 ; /* declare a GLXpixmap to place in the Xpix */

static GLXFBConfigSGIX gFBconfig = 0  ;     /* framebuffer configuration */
static GLXPbufferSGIX  gPBuffer = 0 ;       /* pixelbuffer, in case we case use it */
static GLXContext  oldCtx, glCtx ;  /* rendering contexts */
static Display *Dpy ;               /* display variable */
static int gHeight, gWidth ;        /* rendering buffer dimensions */
static int gScreen = 0 ;            /* pixel buffer screen */
static Bool  OutOfMemory ;          /* make sure we have enough memory */


/*  xErrorHandler()
 *
 * this function is used to handle any error event that might occur if
 * we try to create too big of a Pixmap, or pixel buffer */
 *
 */
static int
xErrorHandler(Display *Dpy, XErrorEvent *evtP)
{
  OutOfMemory = True;
  return 0;
}

/* destroyPixmap()
 *
 * this function destroys and free the memory of any Pixmap, when
```

```
 * we have had our way with it.
 */
static void
destroyPixmap(void)
{
     glXDestroyGLXPixmap(Dpy, GPix);
     GPix = 0;
     XFreePixmap(Dpy, XPix);
     XPix = 0;
}

/* endWrite()
 *
 * this function is called when we are done writing to an offscreen
 * Pixmap. it returns the rendering context to the original (the screen)
 * and then destroys the pixmap.
 */
void
endWrite(void)
{
     (void) glXMakeCurrent(Dpy, XtWindow(glxarea), oldCtx) ;
     glXDestroyContext(Dpy, glCtx) ;

}


/*
 * MakePbuffer code snarfed from Brian Paul's pbdemo written for
 * the "OpenGL and Window System Integration" course presented at
 * SIGGRAPH '97.
 *
 * the pixel buffer is a far cooler beast than a pixmap. basically
 * a pixel buffer is an offscreen rendering buffer, but unlike a
 * Pixmap, it can be configured to have more than 8 bits per
 * color channel. it also has the added benefit of using
 * hardware acceleration. unfortunately, not all machines can
 * handle pixelbuffers. currently SGI O2s, Octanes, RealityEngines,
 * and InfiniteRealitys support them.
 */

GLXPbufferSGIX
MakePbuffer(Display *Dpy, int screen, int width, int height)
{
     /* set up the frame buffer attributes, making sure we set
      * it up to have minimal configurations, as any machine
      * that can handle pixel buffers will by default choose
      * as many color-bits as it can handle.
      */
     int fbAttribs[100] = {
                    GLX_RENDER_TYPE_SGIX, GLX_RGBA_BIT_SGIX,
                    GLX_DRAWABLE_TYPE_SGIX, GLX_PIXMAP_BIT_SGIX,
                    GLX_RED_SIZE, 1,
                    GLX_GREEN_SIZE, 1,
                    GLX_BLUE_SIZE, 1,

                    /* Infinite Realities can define the
                     * frame buffer so that it is a static
                     * gray, with a higher bit depth. uncomment
                     * this out to give that a whirl
                     *
                     * GLX_X_VISUAL_TYPE_EXT, GLX_STATIC_GRAY_EXT,
                     *
                     */

                    GLX_DEPTH_SIZE, 1,
                    GLX_DOUBLEBUFFER, 1,
                    GLX_STENCIL_SIZE, 0,
                    None} ;

     /* now we need to specifically set up the pixelbuffer attributes */
     int pbAttribs[] = {
                    /* use as big a pixelbuffer as we can */
```

```
                    GLX_LARGEST_PBUFFER_SGIX, True,

                    /* don't try to preserve the buffer when we are done */
                    GLX_PRESERVED_CONTENTS_SGIX, False,
                    None
                    };

    GLXFBConfigSGIX *fbConfigs;             /* declare the frame buffer config */
    GLXPbufferSGIX pBuffer = None;   /* don't set up the pBuffer yet */

    int nConfigs;
    int i;


    /* set up the frame buffer, using our chosen attributes *
    fbConfigs = glXChooseFBConfigSGIX(Dpy, screen, fbAttribs, &nConfigs);

    /* glXChooseFBConfigSGIX returns as many different configurations as
     * it can. it places the total number of different configs into the
     * variable nConfigs. first make sure we have at least 1 framebuffer.
     */
    if (nConfigs==0 || !fbConfigs) {
        printf("Error: glxChooseFBConfigSGIX failed\n");
        return 0;
    }



    /* if we have at least one framebuffer then go through until we get
     * the coolest/best framebuffer that is available.
     */
    for (i=0;i<nConfigs;i++) {
        pBuffer = CreatePbuffer(Dpy, fbConfigs[i], width, height, pbAttribs);
        if (pBuffer) {
            gFBconfig = fbConfigs[i];
            gWidth = width;
            gHeight = height;
            break;
        }
    }

    /* no free the framebuffer config memory */
    XFree(fbConfigs);

    /* print out the total number of different configuration
     * possibilites, just so the user knows how great their
     * computer really is */
    fprintf(stderr, "number of fbConfigs = %d\n", nConfigs) ;

    /* return the best pixel buffer possible */
    return pBuffer ;

}

/* PbResize()
 *
 * we need to have a resize function so that when the user selects
 * a different sized image than that of the spectral viewer itself,
 * we don't end up with a little tiny rendering in the corner,
 * or a huge rendering in a time image...
 */
void
PbResize(int width, int height)
{
     glViewport(0, 0, width, height) ;
     glMatrixMode(GL_PROJECTION);
     glLoadIdentity();
     if (width <= (height * 2))
          glOrtho (-6.0, 6.0, -3.0*((GLfloat)height*2)/(GLfloat)width,
                   3.0*((GLfloat)height*2)/(GLfloat)width, -10.0, 10.0);
     else
          glOrtho (-6.0*(GLfloat)height/((GLfloat)height*2),
```

```
                6.0*(GLfloat)width/((GLfloat)height*2), -3.0, 3.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

}

/* pbinit()
 *
 * this function initializes the pixelbuffer for use in offscreen
 * rendering. first it checks to see if pbuffers are available, and
 * if so, then it creates one, and sets the rendering context.
 */
static int
pbInit(int width, int height)
{

/* the first thing we need to do is make sure our computer has both
 * configurable framebuffers, and pixel buffers
 */
#if defined(GLX_SGIX_fbconfig) && defined(GLX_SGIX_pbuffer)

    XVisualInfo *visP ;

    /* Open the X display */
    Dpy = XOpenDisplay(NULL);
    if (!Dpy) {
       fprintf(stderr, "Error: couldn't open default X display.\n");
       return 0;
    }

    /* Get default screen */
    gScreen = DefaultScreen(Dpy);

    /* Test that pbuffers are available */
    if (!QueryPbuffers(Dpy, gScreen)) {
       fprintf(stderr, "Error: pbuffers not available on this screen\n");
       return 0;
    }

    /* Create Pbuffer */

    gPBuffer = MakePbuffer( Dpy, gScreen, width, height );
    if (gPBuffer==None) {
       fprintf(stderr, "Error: couldn't create pbuffer\n");
       return 0 ;
    }

    /* try to get the visual, from the the framebuffer configuration */
    visP = glXGetVisualFromFBConfigSGIX(Dpy, gFBconfig) ;
    if (!visP){
       fprintf(stderr, "Error: couldn't create visual from FBconfig\n");
       return 0 ;
    }

    /* try to create a GLX context with direct rendering AKA hardware....   */

    glCtx = glXCreateContext(Dpy, visP, NULL, True);

    /* if that doesn't work, try indirect.... AKA software....    */
    if (!glCtx){
       glCtx = glXCreateContext(Dpy, visP, NULL, False) ;
       if(!glCtx){
            fprintf(stderr, "Error: Could not create GLXContext!\n") ;
            return 0 ;
       }
       else{
            fprintf(stderr, "Warning...using indirect GLXContext...could be slow!\n") ;
       }
    }

    /* get the current context, so that we can go back to on screen rendering
     * when we are done with it.
```

```
     */
    oldCtx = glXGetCurrentContext() ;

    /* make the pixel buffer the current rendering context */
    if(!glXMakeCurrent(Dpy, gPBuffer, glCtx)) {
       fprintf(stderr, "Error: glXMakeCurrent failed\n");
       glXMakeCurrent(Dpy, XtWindow(glxarea), oldCtx) ;
       XFree(visP);
       return 0;
    }

    return 1 ;   /* could it be true that this entire process worked?!!! yeah!!! */

/* if we don't have cofigurable frame and pixel buffers, let the user know */
#else
     fprintf(stderr, "Error: GLX+SGIX_fbconfig or GLX_SGIX_pbuffer just don't work!\n") ;
    return 0 ;

#endif
}


/* calc_fluorescent_scale()
 *
 * since we are now dealing with the full spectral information, we
 * need to calculate the contribution of fluorescence ourselves. this
 * function determines amount of fluorescent excitation caused by
 * the current light source, and the total possible amount (caused
 * by the equal energy spectrum). the ratio of the two is determined
 * to be the fluorescent scalar.
 */
GLfloat
calc_fluorescent_scale(double *excite, double *illuminant)
{
    double      equal[93] ;         /* equal energy spectrum */
    double      illum_equal[93] ;   /* excitation caused by equal energy */
    double      illum_excite[93] ,  /* excitation caused by illuminant */
    double      N ;                 /* fluorescence scalar */
    int         i ;


    /* define the equal energy spectrum */
    for(i=0;i<93;i++){
        equal[i] = 1.0 ;
    }


    /* multiply the scaled illuminant by the excitation spectra  */

    mult_spect(illuminant, excite, illum_excite) ;

    /* multiply the equal enegry spectra by the excitation spectra */

    mult_spect(equal, excite, illum_equal) ;


    /* determine the ratio between the areas of the equal energy
     * (maximum energy at all bands) and the current illuminant
     * this determines the number of quanta absorbed.
     */

    N = area_spect(illum_excite) / area_spect(illum_equal) ;

    /* return the scalar */
    return N ;
}



/* drawSpectral()
 *
 * this function is more or less equivalent to the draw() function in
```

```
 * the spect-gl.c file. basically this function takes care of most of
 * the rendering, as well as the colors. since we are now dealing with
 * specific wavelengths, rather than an RGB approximation, this function
 * is passed in the wavelength. the material properties are then set to
 * be the spectral radiances at that wavelength (taking into account
 * both diffuse and specular reflectance, the spectral properties of
 * the current light source[s], and the contribution from fluorescence.
 *
 */
void
drawSpectral(int wavelength)
{
    double    current_local[93] ;   /* current light source 1 */
    double    current_local2[93] ;  /* current light source 1 */
    double    current_ambient[93] ; /* current ambient light source */
    double    scale_local[93] ;     /* scaled local 1 */
    double    scale_local2[93] ;    /* scaled local 2 */
    double    scale_ambient[93] ;   /* scaled ambient */
    double    temp_excite[93] ;     /* temporary excitation spectra */
    GLfloat     mat_diffuse[] = {1.0, 0.0, 0.0, 1.0} ;     /* diffuse material */
    GLfloat     mat_specular[] = {1.0, 0.0, 0.0, 1.0} ;    /* diffuse specular */
    GLfloat     test_specular[] = {1.0, 0.0, 0.0, 1.0} ;  /* temp specular */
    GLfloat     mat_ambient[] = {0.0, 0.0, 0.0, 0.0} ;      /* ambient material */
    GLfloat     null_ambient[] = {0.0, 0.0, 0.0, 0.0} ;     /* NULL material */
    GLfloat     null_specular[] = {0.0, 0.0, 0.0, 0.0} ;  /* NULL material */
    GLfloat     light_ambient[] = { 0.0, 0.0, 0.0, 1.0 }; /* ambient light */
    GLfloat     light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 }; /* diffuse light */
    GLfloat     light_specular[] = { 1.0, 1.0, 1.0, 1.0 }; /* specular light */
    GLfloat     light_diffuse2[] = { 1.0, 1.0, 1.0, 1.0 }; /* diffuse light */
    GLfloat     light_specular2[] = { 1.0, 1.0, 1.0, 1.0 }; /* specular light */
    GLfloat     cur_gloss ;          /* current gloss/shininess */
    GLfloat     x, shift = 0.0;             /* position variable */
    GLfloat     y , scale = 1.0 ;   /* position variable */
    GLdouble    right_clip[] = {-1.0, 0.0, 0.0, 0.0} ;      /* clipping plane */
    GLdouble    left_clip[] = {1.0, 0.0, 0.0, 0.0} ;        /* other clipping */
    int         rows = 1;            /* number of rows */
    int         collumns, i ;               /* other position varibles */
    int         no_col = 0 ;         /* number of collumns */
    int       wl_index ;             /* wavelength index */
    int       j ;                    /* loop counter */
    GLfloat     neutral = 0.2 ;             /* nonselective neutral reflectance */
    GLfloat   FluorScale, FluorScale_amb ;          /* fluorescents scale */

       /* set up the variables for the accumulation loop */
    int             light_number = 1 ;
       int                    accum_pass ;



    /* first off disable texturing until we need it. */
    glDisable(GL_TEXTURE_2D) ;

    /* Calculate the number of rows and collumns, to determine
     * the layout of the objects
     */

    if(NUM_FILES>3){
        rows = 2 ;
        scale = 0.66 ;
    }
    if(NUM_FILES>8){
        rows = 3 ;
        scale = 0.5 ;
    }
    if(NUM_FILES>15){
        rows = 4 ;
        scale = 0.333 ;
    }


    if(NUM_FILES % rows == 0){
        collumns = NUM_FILES/rows ;
```

```
}
else{
    collumns = NUM_FILES/rows + 1 ;
}

x = (GLfloat) -3.0*(collumns-1.0) ;
y = (GLfloat) (rows   1.0)*3.0 ;


/* Scale the ambient and diffuse lighting, according to sliders  */

   /*
light_ambient[0] = AmbScale ;
light_diffuse[0] = DiffScale ;
light_specular[0] = DiffScale ;
light_diffuse2[0] = DiffScale2 ;
light_specular2[0] = DiffScale2 ;
   */


glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient) ;
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT1, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT1, GL_SPECULAR, light_specular);


/* disable the lights until we need them */
glDisable(GL_LIGHT0) ;
glDisable(GL_LIGHT1) ;

   if(CUR_ILLUMINANT2 != NONE) {
          light_number = 2 ;
   }


/* If the colors need to be recalculated do so now  */
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glClear(GL_ACCUM_BUFFER_BIT) ;


glLoadIdentity() ;

/* set our drawing and reading buffers to be the back buffer,
 * so we do not display until after we have calculated the
 * entire accumulation buffer.
 */
glDrawBuffer(GL_BACK) ;
glReadBuffer(GL_BACK );


   /* start our big accumulation buffer loop. we will go through this
    * loop twice, if both local light sources are turned on. each
    * pass will be rendered into the accumulation buffer, and then
    * the combined two renderings will be rendered back into the
    * normal drawing buffer, creating the simulation of two light
    * sources.
    */
for(accum_pass = 0; accum_pass < light_number ; accum_pass ++) {

    glMaterialfv(GL_FRONT, GL_AMBIENT, null_ambient) ;

       /* convert the Wavelength into the Array Index */

    wl_index = (wavelength   300)/5 ;

          /* deterimine which illuminant we need to get, and then
           * scale that mother
           */
          if(accum_pass > 0) {
                 return_illumiant(CUR_ILLUMINANT2, current_local) ;
```

```
        scale_spect(current_local, DiffScale2, scale_local) ;
          glEnable(GL_LIGHT1) ;
          glDisable(GL_LIGHT0) ;
    }
    else {
          return_illuminant(CUR_ILLUMINANT, current_local) ;
          scale_spect(current_local, DiffScale, scale_local) ;
          glEnable(GL_LIGHT0) ;
          }

  return_illuminant(CUR_AMBIENT, current_ambient) ;
  scale_spect(current_ambient, AmbScale, scale_ambient) ;



    /* set our diffuse material property to be the scaled illuminant
     * at the current wavelength, scaled by our 0.2 neutral gray reflectance.
     */
  mat_diffuse[0] = scale_local[wl_index]*neutral ;
  glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse) ;

    /* if the ambient is turned on, we need to set the ambient material
     * property equal to the scaled ambient radiance, at the current
     * wavelength, scaled by the gray reflectance of 0.2.
     */
  if(AmbScale != 0.0){
      mat_ambient[0] = scale_ambient[wl_index]*neutral ;
      glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient) ;
  }


    /* now draw our really really big sphere, as the background */
  glPushMatrix() ;
      glTranslated(0.0, 0.0, -16.0) ;
      sphere(14.0, 40, 40)  ;
  glPopMatrix() ;

    /* set the ambient back to NULL */
  glMaterialfv(GL_FRONT, GL_AMBIENT, null_ambient) ;

    /* set our scale, depending upon the number of total objects being
     * rendered */
  glScalef(scale, scale, scale) ;

  /* now we go through the other big old for loop. this one loops
     * through once for every material file that has been initialized.
     * basically what we are doing is setting the material property for
     * each object to be the total spectral radiance for a single given
     * wavelength.
     */
  for(i=0;i<NUM_FILES;i++){

          /* our first task is to deterimine whether we are dealing with
           * metameric material files...if so, we must deal with them
           * accordingly.
           */
      if (METAMERIC[i] == TRUE){
          glPushMatrix() ;

                  /* now we need to see if the object is fluorescent */
                  if(FLUORESCENT[i] == TRUE){
                        /* if so, we need to grab the excitation spectra */
                      for(j = 0; j<93; j++){
                        temp_excite[j] = excite[i][j] ;
                      }

                        /* now we need to calculate the fluorescent scaling
factor */
                      FluorScale =
                                    calc_fluorescent_scale(temp_excite,
scale_local) ;
                      FluorScale_amb =
```

```
                                                    calc_fluorescent_scale(temp_excite,
scale_ambient) ;
                              }
                              /* if not, then set the scaling factors equal to zero */
                              else{
                                   FluorScale = FluorScale_amb = 0.0 ;
                              }


                              /* now we set the diffuse material properties. basically what
                               * we are dealing with is grabbing the diffuse reflectance
                               * at the current wavelength, scaling that by the scaled
                               * local illuminant, and then adding in the fluorescent
                               * contribution. this should be the total diffuse radiance,
                               * for the current wavelength.
                               */
                  mat_diffuse[0] = reflect[i][wl_index]*scale_local[wl_index]
                                           + FluorScale*emit[i][wl_index] ;

                              /* now we set the diffuse material property */
                  glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse) ;

                              /* next we have to set the specular component. if there is
                               * as specular component, then this is simple the spectral
                               * specular reflectance multiplied by the scaled local
                               * illuminant. if there is no defined specular component,
                               * then this should just be the scaled local illuminant.
                               */
                              if(SPECULAR == TRUE) {
                                     mat_specular[0] =
specular[i][wl_index]*scale_local[wl_index] ;
                              }
                              else {
                                     mat_specular[0] = scale_local[wl_index] ;
                              }

                              /* determine what the gloss component is */
                  cur_gloss = (GLfloat)gloss[i] ;

                              /* if there is some gloss, we need to set the specular
                               * material property
                               */
                  if(gloss[i] > 0.0){
                       glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular) ;
                  }
                  else{
                       glMaterialfv(GL_FRONT, GL_SPECULAR, null_specular) ;
                  }

                              /* now we need to set the shininess coefficient */
                  glMaterialf(GL_FRONT, GL_SHININESS, cur_gloss) ;

                              /* if there is an ambient light turned on, we need to set the
                               * ambient component. this is considered to be the diffuse
                               * spectral reflectance, scaled by the current ambient
                               * illuminant, with the fluorescent contribution added.
                               */
                  if(AmbScale != 0.0){
                       mat_ambient[0] = reflect[i][wl_index]*scale_ambient[wl_index]
                                                 + FluorScale_amb*emit[i][wl_index] ;

                       glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient) ;
                  }


                              /* position ourselves in the correct place */
                  glTranslatef(x+shift, y, 0.0) ;

                  glPushMatrix() ;
                                     /* set the necessary clipping plane */
                      right_clip[0] = -3*(x+shift) ;
                      glClipPlane(GL_CLIP_PLANE1, right_clip) ;
```

```
glEnable(GL_CLIP_PLANE1) ;

                    /* now draw the appropriate object */
draw_object() ;

                    /* disable the clipping plane */
glDisable(GL_CLIP_PLANE1) ;
glPopMatrix() ;


        /* now we have to worry whether the assigned metameric
         * pair also has fluorescent tendencies (sounds so darn
         * psychological).
         */
        if(MET_FLUORESCENT[i] == TRUE){
                /* if so, lets get the excitation spectra */
            for(j = 0; j<93; j++){
                temp_excite[j] = met_excite[i][j] ;
            }

                /* calculate the fluorescent scaling factors */
            FluorScale =
                                    calc_fluorescent_scale(temp_excite,
scale_local) ;

            FluorScale_amb =
                                    calc_fluorescent_scale(temp_excite,
scale_ambient) ;
        }
        /* if we don't need to worry about this fluorescence stuff,
         * then just set the scales to be zero.
         */
        else{
            FluorScale = FluorScale_amb = 0.0 ;
        }

        /* we set up the diffuse material components identical to
         * what we did above.
         */
mat_diffuse[0] = metamer[i][wl_index]*scale_local[wl_index]
         + FluorScale*met_emit[i][wl_index] ;

glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse) ;

        /* we set the specular component as above */

        if(MET_SPECULAR == TRUE) {
                mat_specular[0] = met_specular[i][wl_index]
                                                                    *
scale_local[wl_index] ;
        }
        else {
                mat_specular[0] = scale_local[wl_index] ;
        }

        /* get the current gloss component */
cur_gloss = (GLfloat)met_gloss[i] ;

        /* determine if we even need to bother with the specular stuff
*/
if(met_gloss[i] > 0.0)
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular) ;
else{
    glMaterialfv(GL_FRONT, GL_SPECULAR, null_specular) ;
}

        /* set the shininess/gloss */
glMaterialf(GL_FRONT, GL_SHININESS, cur_gloss) ;

        /* now we need to determine whether or not the ambient light
         * is going to come into play. aka, is the ambient light turned
         * on? if so, then set up the ambient properties as above.
         */
```

```
 if(AmbScale != 0.0){
      mat_ambient[0] = metamer[i][wl_index]*scale_ambient[wl_index]
                   + FluorScale_amb*met_emit[i][wl_index] ;
      glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient) ;
 }
              /* get ready to render, with the opposite clipping plane as
               * above.
               */
 glPushMatrix() ;
                       /* set up the clipping plane */
      left_clip[0] = 3.0*(x+shift) ;
      glClipPlane(GL_CLIP_PLANE2, left_clip) ;
      glEnable(GL_CLIP_PLANE2) ;

                       /* draw the appropriate object */
      draw_object() ;

                       /* turn off the clipping plane */
      glDisable(GL_CLIP_PLANE2) ;
 glPopMatrix() ;

              /* determine the position for the next loop around */
 shift = shift + 6.0 ;
 no_col++ ;
 if(no_col >= collumns){
      shift = 0.0 ;
      y = y - 6.0 ;
      no_col = 0 ;
 }
     glPopMatrix() ;
}

      /* well, if we aren't dealing with metameric pairs, then all
       * that work was for not...but nevertheless we must press on.
       * basically we need to repeat all that was going on up there,
       * down here.
       */
else{
     glPushMatrix() ;

              /* determine if we are dealing with fluorescent objects...
               * and get the scale if we are.
               */
              if(FLUORESCENT[i] == TRUE){
                       /* get the flourescent excitation spectra */
                   for(j = 0; j<93; j++){
                             temp_excite[j] = excite[i][j] ;
                   }
                       /* calculate the scaling factor */
                   FluorScale =
                             calc_fluorescent_scale(temp_excite, scale_local) ;
                   FluorScale_amb =
                             calc_fluorescent_scale(temp_excite, scale_ambient)
;

              }
              /* if not, just set the scale to be zero. */
              else{
                   FluorScale = FluorScale_amb = 0.0 ;
              }

              /* now we set the diffuse material properties. basically what
               * we are dealing with is grabbing the diffuse reflectance
               * at the current wavelength, scaling that by the scaled
               * local illuminant, and then adding in the fluorescent
               * contribution. this should be the total diffuse radiance,
               * for the current wavelength.
               */
 mat_diffuse[0] = reflect[i][wl_index]*scale_local[wl_index]
               + FluorScale*emit[i][wl_index] ;

 glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse) ;
```

```
                    /* now we have to determine the specular component...this
                     * should be either the product of the specular reflectance
                     * and the scaled local illuminant, or just the scaled
                     * local illuminant
                     */
                    if(SPECULAR == TRUE ) {
                          mat_specular[0] =
specular[i][wl_index]*scale_local[wl_index] ;
                    }
                    else {
                          mat_specular[0] = scale_local[wl_index] ;
                    }

                    /* get the current gloss factor */
             cur_gloss = (GLfloat)gloss[i] ;

                    /* if this gloss is bigger than 0.0, set our specular material
                     * properties
                     */
             if(gloss[i] > 0.0){
                  glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
             }
             else{
                  glMaterialfv(GL_FRONT, GL_SPECULAR, null_specular) ;
             }

                    /* set our shininess/gloss material property */
             glMaterialf(GL_FRONT, GL_SHININESS, cur_gloss) ;

                    /* if we are dealing with ambient light, as in the the
                     * ambient light is turned on, we need to calculate that
                     * contribution. same way as above.
                     */
             if(AmbScale != 0.0){
                  mat_ambient[0] = reflect[i][wl_index]*scale_ambient[wl_index]
                          + FluorScale_amb*emit[i][wl_index] ;

                  glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient) ;
             }

                    /* position ourselves in the correct spot */
             glTranslatef(x+shift, y, 0.0) ;

                    /* set our texture mapping properties, if necessary */
             if(TEXTURE[i] == TRUE){
                  glEnable(GL_TEXTURE_2D) ;
                  glBindTexture(GL_TEXTURE_2D, (GLuint)NumTex[i]) ;

                  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                          GL_LINEAR_MIPMAP_LINEAR);
                  glTexParameteri(GL_TEXTURE_2D,
                          GL_TEXTURE_MAG_FILTER, GL_LINEAR);

                          /* call and draw the correct object */
                  draw_object() ;

                          /* disable texturing, so that every thing doesn't
                           * get covered with the same texture.
                           */
                  glDisable(GL_TEXTURE_2D) ;
             }
                    /* if texturing isn't really necessary, just draw baby */
             else{
                  draw_object() ;
             }

                    /* set up our positioning for the next run through the
                     * loop
                     */
             shift = shift + 6.0 ;
             no_col++ ;
             if(no_col >= collumns){
```

```
                  shift = 0.0 ;
                  y = y - 6.0 ;
                  no_col = 0 ;
              }
          glPopMatrix() ;
              }

                   /* if we have more than one light source turned on, we need
                    * to place what we have just drawn into the accumulation
                    * buffer, and scale it so taht the two slider bar scales
                    * add up to be 1.0.
                    */
                   if(accum_pass == 0 && light_number >1 ) {
                glAccum(GL_ACCUM,, DiffScale/(DiffScale + DIffScale2) ;
             }
                   /* if only one light is on, set the accumulation buffer equal
                    * to unity
                    */
                   else if {
                       glAccum(GL_ACCUM, 1.0) ;
             }
                   /* if both light sources are on, and we are on the second pass
                    * of the accumulation loop, use this to scale the buffer */
             else {
                       glAccum(GL_ACCUM, DiffScale2/(DiffScale + DiffScale2) ;
             }
          }

       /* get the combined images from the accumulation buffer, and transfer
        * them to the drawing buffer.
        */
       glAccum(GL_RETURN, 1.0) ;

       /* flush all the OpenGL calls to the screen */
    glFlush() ;

}


/* writeImage()
 *
 * this function controls all of the steps needed to write out a full
 * spectral image. for that reason, it needs to be passed a whole lotta
 * info. specifically, it needs to be passed the image dimensions, in
 * pixels, the wavelength increment, the starting and finishing wavelengths,
 * and the file name. from there, this function makes all the right
 * calls, and we end up with a (usually large) full spectral image.
 *
 */
void
writeImage(int width, int height, int incrementWL, int startWL, int endWL, char *file )
{

    GLfloat    *pixel ;                     /* frame buffer pixel data */
    int32      sd_id, sds_id, status;       /* HDF scientific data set variables */
    int32      dims[3], start[3], edges[3], rank;  /* more HDF goodies */
    int        i, j, k;                            /* random letters and/or loop control */
    int        x, y, z ;                           /* image dimensions */
    float      *image ;                            /* HDF image array */
    int        depth  ;                            /* total number of wavelength bands */
    int        PBinit = 0 ;                 /* pixelbuffer init stuff */
    char       *hdf_file ;                         /* name of the output file */


       /* the first step is to determine the total number of wavelength
        * band that we are dealing with. this is determined based upon
        * the chosen starting, finishing, and wavelength increments.
        */
    depth = (endWL - startWL)/incrementWL ;


       /* we need to allocate the memory for the hdf image. WARNING>>>>
```

```
	 * this might require much memory. please do not hold me
	 * responsible for overloading your computer. a good idea might
	 * be to start small...and to work the image size up incrementally.
	 */
	image = (float *)malloc(height*width*depth * sizeof(float **)) ;

	   /* next we need to allocate the memory for the pixel information */
	pixel = (GLfloat *) malloc(width * sizeof(GLfloat)) ;

	   /* initialize the pixel buffer. if we don't have a pixel buffer,
	    * then we are going to default to reading the screen frame buffer.
	    */
	PBinit = pbInit(width, height) ;


	   /* if we have a pixel buffer, then we need to re-initialize all the
	    * previous OpenGL calls (big pain, i know...but it is worth it).
	    */
	if(PBinit != 0){
	    gl_init() ;
	}

	   /* resize the the appropriate image sizes */
	PbResize(width, height) ;

	   /* set up our pixel storage, so they are packed nicely */
	glPixelStorei(GL_PACK_ALIGNMENT, 1) ;

	   /* start our big loop for drawing single wavelength images, and
	    * then reading them out of the frame buffer...basically, the
	    * z dimension controls the wavelength, while the x-y control
	    * screen coordinates.
	    */
	for(z = 0; z<depth; z++){

		   /* draw spectral, at the wavelength of z*increment+startingWL.
		    * Example: z = 1, increment = 10nm, start = 400...wavelength = 410
		    */
	    drawSpectral(z*incrementWL + startWL) ;

		   /* start at the bottom of the image, and take a 1 pixel high
		    * strip of pixel information (1 x Image Height).
		    */
	    for(v = 0; y < height; y++){
		glReadPixels(0, y, width, 1, GL_LUMINANCE, GL_FLOAT, pixel) ;
				/* read each pixel, starting at the far left, moving right */
			for(x = 0; x < width ; x++){
				/* the hdf image, equals that pixel value. since we can
				 * only deal with a single pointer *image, we have to
				 * do a little bit of fancy pointer math to pretend that
				 * this is really a three-dimensional array.
				 */
			    image[z + (x)*depth + y*width*depth] = pixel[x] ;
			}
	    }
	}


	/* Create and open the file and initiate the SD interface. */
	hdf_file = file ;

	   /* start the scientific data set (see HDF documentation for more
	    * details
	    */
	sd_id = SDstart(hdf_file, DFACC_CREATE);

	   /* let the user know the sd_id, for later reference */
	fprintf(stderr, "sd_id = %d\n", sd_id) ;


	/* Define the rank and dimensions of the data set to be created. */
```

```
       /* how many dimensions are we dealing with? */
    rank = 3;

       /* what the the dimensions of the three dimensions (is it possible
        * to say dimensions more time in once sentence?)
        */
    dims[0] = height;
    dims[1] = width;
    dims[2] = depth;


    /* Create the array data set. */
    sds_id = SDcreate(sd_id, "test_array", DFNT_FLOAT32, rank, dims);


    /* Define the location, pattern, and size of the data set */
    for (i = 0; i < rank; i++) {
        start[i] = 0;
        edges[i] = dims[i];
    }


    /* Write the stored data to the "Ex_Array_3" data set. The fifth
     * argument must be explicitly cast to a generic pointer to conform
     * to the HDF API definition for SDwritedata. */

    status = SDwritedata(sds_id, start, NULL, edges, (VOIDP)image);


    /* Terminate access to the array. */

    status = SDendaccess(sds_id);


    /* Terminate access to the SD interface and close the file. */

    status = SDend(sd_id);

       /* let the user know that the file has be written */
    fprintf(stderr, "HDFwrite finished \n") ;

       /* if we have a pixelbuffer, we need to destroy it */
    if(PBinit != 0){
        endWrite() ;
    }

       /* that is it, we have just written out a full spectral image. cool,
        * eh?
        */
}
```

spect-gl.pasture.c

```
/*
 * spect-gl.pasture.c
 *
 * Written By: Garrett M. Johnson
 *
 * OpenGL rendering file for the full spectral rendering package
 * using OpenGL, and Motif.
 *
 * This file is a simple replacement for the regular spect-gl.c file.
 * Basically, it replaces the MacBeth ColorChecker style layout with
 * a nice rolling pasture scene. Essentially, this file replaces the
 * reshape(), gl_init(), and draw() functions. To use this file, simply
 * rename the original spect-gl.c, and then rename this file "spect-gl.c"
 * Then just run the make file, and all is good.
 *
 * Last revised: 07/27/98
 *
 */


#include <stdio.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>
#include <GL/glx.h>
#include "spectral.h"
#include "glm.h"

/* the first thing we need to do is figure out what version of
 * OpenGL we have running. if we have an older version we need
 * to define the bindtexture extension equal to that of the
 * original function.
 */

#if !defined(GL_VERSION_1_1) && !defined(GL_VERSION_1_2)
#define glBindTexture glBindTextureEXT
#endif

int     CUR_OBJECT = TORUS ;            /* set our current object to torus */
Bool    RECALCULATE_COLORS = TRUE ;     /* assume that we need to calc colors */
GLfloat AmbScale = 0.0 ;                /* start off with no ambient */
GLfloat DiffScale = 1.0 ;               /* start off with full local 1 */
GLfloat DiffScale2 = 0.0 ;              /* start off with no local 2 */
GLuint  spherelist ;                    /* sphere display list name */
GLuint  cow_list ;                      /* cow display list */
GLuint  pasture_list ;                          /* pasture display list */
GLuint  moon_list ;                     /* moon display list */
GLuint  toruslist ;                     /* torus display list name */
GLuint  cow_list ;                      /* cow display list */
GLuint  ship_list ;                     /* spanish galleon list */
GLuint  woman_list ;                    /* woman head list */
GLuint  tricer_list ;                   /* triceratops list */
GLMmodel *model ;                       /* the glm model variable */
GLfloat scale ;                                 /* a model scaling variable */

/* reshape()
 *
 * the reshape function is called when the from the resize function
 * in the spect-cb.c file. it is placed inside the spect-gl.c file
 * because it defines the gl viewpoint and all that good stuff.
 *
 */
```

```
void
reshape(int width, int height)
{
    int winWidth, winHeight ;


    winWidth = width ;              /* set the window dimensions */
    winHeight = height ;

    glViewport(0, 0, winWidth, winHeight) ;/* define the viewport */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    /* set our orthographic projection matrix. if the desired goal is
     * to show some sort of perspective, don't use ortho. (duh?)
     */
    if (winWidth <= (winHeight * 2))
        glOrtho (-6.0, 6.0, -3.0*((GLfloat)winHeight*2)/(GLfloat)winWidth,
                3.0*((GLfloat)winHeight*2)/(GLfloat)winWidth, -10.0, 10.0);
    else
        glOrtho (-6.0*(GLfloat)winHeight/((GLfloat)winHeight*2),
                6.0*(GLfloat)winWidth/((GLfloat)winHeight*2), -3.0, 3.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

}



/* gl_init()
 *
 * the gl_init function is used to initialize all the good wholesome
 * OpenGL states. This is where we define the light source properties,
 * including position. Since we are kludging the actual lighting
 * so that it simulates some full-spectral goodness, we simply set all
 * the light source RGBA variables equal to 1.0. we will deal with
 * the color calculations in spect-math.c. oh yeah, in this file we
 * also initialize all the models used in the spectral viewer. if you
 * as an enduser want new cooler models, simply declare said model
 * in this function.
 */
void
gl_init(void)
{

    /* set our light source variables */
    GLfloat diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat ambient[] = { 0.2, 0.2, 0.2, 1.0} ;
    GLfloat position[] = { 4.0, 2.0, 0.0, 1.0 };


    /* now define light source number one (local 1) */
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE) ;


    /* now that we have defined the light sources, lets enable them */
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);


    /* enable the standard z-buffer depth testing, and back face culling */
    glEnable(GL_DEPTH_TEST) ;
    glEnable(GL_CULL_FACE) ;

    /* since the spectral viewer scales the object sizes based on the
     * total number of material files present, we should set the auto
     * normalize function, so OpenGL handles our normals for us */
    glEnable(GL_AUTO_NORMAL) ;
```

```
glEnable(GL_NORMALIZE) ;

/* generate the necessary display lists for the sphere and torus */
spherelist = glGenLists(1) ;
toruslist = glGenLists(1) ;



/* now it is time to define the sphere display list. the sphere()
 * function can be found in the shapes.c file.
 */
glNewList(spherelist, GL_COMPILE) ;
   sphere(2.8, 30, 30)  ;
glEndList() ;

  /* well, we are going to define the display list for the torus, but
 * we are going to get a little fancier. since our torus is a simple
 * little mathematically defined model, we don't really have any
 * texture coordinates for it. so we will have OpenGL generate the
 * coordinates (and do a pretty lousy job at that) for us.
 */
glNewList(toruslist, GL_COMPILE) ;
     glEnable(GL_TEXTURE_GEN_S) ; /* enable texture coordinate generation */
     glEnable(GL_TEXTURE_GEN_T) ;  /* enable texture coordinate generation */

   /* now lets generate the texture coordinates, based upon a textured
    * spherical mapping
    */
   glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP) ;
     glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP) ;

   /* finally call the torus() function, defined in shapes.c */
   Torus(1.05, 1.65, 30, 30)  ;

   /* disable the texture generation, so GL doesn't go nuts and try to
    * define texture coordinates for everything.
    */
     glDisable(GL_TEXTURE_GEN_S) ;
     glDisable(GL_TEXTURE_GEN_T) ;
glEndList() ;



/* now the fun stuff, we get to define the display lists for all the
 * other cool models. this is done using wavefront OBJ model files,
 * and the glm Rendering library. This library was written by the
 * ever impressive Nate Robbins, and can be found at various web sites,
 * including http://trant.sgi.com/opengl/examples/more_samples/smooth.
 * all in all, this is a very cool little library, my thanks to nate.
 */
model = glmReadOBJ("obj/cow.obj") ;     /* give glm the name of the obj file */
scale = glmUnitize(model) ;             /* unitize, in case it is huge */
glmFacetNormals(model) ;        /* make sure the normals are correct */
glmVertexNormals(model, 90.0) ;
glmLinearTexture(model) ;               /* define a linear texture mapping */
glmScale(model, 0.8) ;          /* scale the model back up to size */

/* now we let glm define the display list, using smooth shading, and
 * texture coordinates.
 */
cow_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE) ;

/* basically we do the same things as above, for all the other models */
model = glmReadOBJ("obj/pasture2.obj") ;
scale = glmUnitize(model) ;
glmFacetNormals(model) ;
glmVertexNormals(model, 90.0) ;
glmScale(model, 6.0) ;
glmLinearTexture(model) ;

pasture_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE) ;
```

```
    model = glmReadOBJ("obj/moon.obj") ;
    scale = glmUnitize(model) ;
    glmFacetNormals(model) ;
    glmVertexNormals(model, 90.0) ;
    glmScale(model, 0.6) ;
    glmLinearTexture(model) ;

    moon_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE) ;

      /* basically we do the same things as above, for all the other models */
    model = glmReadOBJ("obj/galleon.obj") ;
    scale = glmUnitize(model) ;
    glmFacetNormals(model) ;
    glmVertexNormals(model, 90.0) ;
    glmScale(model, 2.8) ;
    glmLinearTexture(model) ;

    ship_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE) ;

    model = glmReadOBJ("obj/womanhead.obj") ;
    scale = glmUnitize(model) ;
    glmFacetNormals(model) ;
    glmVertexNormals(model, 90.0) ;
    glmScale(model, 2.5) ;
    glmLinearTexture(model) ;

    woman_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE) ;

    model = glmReadOBJ("obj/triceratops.obj") ;
    scale = glmUnitize(model) ;
    glmFacetNormals(model) ;
    glmVertexNormals(model, 90.0) ;
    glmScale(model, 3.5) ;
    glmLinearTexture(model) ;

    tricer_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE) ;

    /* set our texture mapping so that it modulates the color of the
     * object, rather than replaces. basically this is done so we
     * can hack some imitation spectral texture maps, using full-spectral
     * object colors, and non-spectrally selective texture maps.
     * basically that is just a fancy name for grey-scale textures
     */
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

    /* clear the background color to white, for the heck of it */
    glClearColor(1.0, 1.0, 1.0, 1.0)  ;
}

/* draw_object()
 *
 * this is a simple function designed to make it easy to switch
 * between objects in the spectral viewer. basically it allows
 * the drawing call to be the same, regardless of the current
 * model choice. the draw_object() function then uses the
 * (sweet sweet) global variable to determine the correct
 * display list to call. so simple, so sweet, so hackesque.
 */
void
draw_object(void)
{
    long random()  ; /* in case we want to randomly rotate the models */
    GLfloat i ;              /* our random number, of course */

    i = (GLfloat)random()  ;

    /* basically we use the CUR_OBJECT variable ,set in the object_cb()
     * function, to control the switch statement. remember, we enumerated
     * the object names in the header file.
     */
    switch(CUR_OBJECT){
      case  TORUS:
```

```
            glCallList(toruslist) ;
            break ;
        case  SPHERE:
             glCallList(spherelist) ;
             break ;
        case  COW:
            glPushMatrix() ;
                /* lets rotate the cows, for good measure... */
                glRotatef(i, 0.0, 1.0, 0.0) ;
                 glCallList(cow_list) ;
            glPopMatrix() ;
             break ;
        case  SHIP:
             glCallList(ship_list) ;
             break ;
        case  WOMAN:
            glPushMatrix() ;
                glRotatef(i, 0.0, 1.0, 0.0) ;
                 glCallList(woman_list) ;
            glPopMatrix() ;
             break ;
        case  TRICER:
            glPushMatrix() ;
               glRotatef(-35.0, 0.0, 1.0, 0.0) ;
                 glCallList(tricer_list) ;
            glPopMatrix() ;
             break ;

    }

}


/* draw()
 *
 * oh yeah baby, the meat of all our rendering is done in the good
 * old draw() function. this is where it all comes together, and
 * makes our life fun and exciting. as an added bonus, if at any
 * time you, the user, get bored with the ColorChecker type layout
 * of the interactive view, this is the only function that really
 * needs to be changed. in this function we call out for the spectral
 * color calculations, calculate what the resultant spectra would
 * look like if imaged by our current observer, and then render our
 * objects using these "tristimulus" values. sounds simple enough.
 */
void
draw(GLenum mode)
{
    static double     amb_rgb[24][3] ;       /* ambient tristimulus values */
    static double     local_rgb[24][3] ;     /* local 1 tristim values */
    static double     local2_rgb[24][3] ;    /* local 2 tristim values */
    static double     amb_met_rgb[24][3] ;  /* metameric ambient trist */
    static double     local_met_rgb[24][3] ;        /* metameric local 1 trist */
    static double     local2_met_rgb[24][3] ;       /* metameric local 2 trist */
    static double     spec_rgb[24][3] ;      /* specular local 1 trist */
    static double     met_spec_rgb[24][3] ; /* metameric local 1 specular */
    static double     spec2_rgb[24][3] ;     /* specular local 2 trist */
    static double     met2_spec_rgb[24][3] ;        /* metameric local 2 specular */
    static double     amb_back_rgb[3] ;      /* ambient background trist */
    static double     local_back_rgb[3] ;    /* local background trist */

    /* defaul light and material properties */
    GLfloat      mat_diffuse[] = {1.0, 1.0, 1.0, 1.0} ;
    GLfloat      mat_specular[] = {1.0, 1.0, 1.0, 1.0} ;
    GLfloat      test_specular[] = {1.0, 1.0, 1.0, 1.0} ;
    GLfloat      mat_ambient[] = {0.0, 0.0, 0.0, 0.0} ;
    GLfloat      null_ambient[] = {0.0, 0.0, 0.0, 0.0} ;
    GLfloat      null_specular[] = {0.0, 0.0, 0.0, 0.0} ;
    GLfloat    light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
    GLfloat    light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat    light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
```

```
/* current gloss (shininess) component */
GLfloat   cur_gloss ;



/* we want the disable texture mapping, until we need it. */
glDisable(GL_TEXTURE_2D) ;




/* Scale the ambient and diffuse lighting, according to sliders  */

light_ambient[0] = light_ambient[1] = light_ambient[2] = AmbScale ;
light_diffuse[0] = light_diffuse[1] = light_diffuse[2] = DiffScale ;
light_specular[0] = light_specular[1] = light_specular[2] = DiffScale ;


/* set up the light properties based upon the above calculations */
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient) ;
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);


/* now clear the accumulation buffer */
glClear(GL_ACCUM_BUFFER_BIT) ;


/* If the colors need to be recalculated do so now. first
 attempt to do it with the locall light source. */
if(RECALCULATE_COLORS) {
    /* get the digital rgb values */
   get_rgb(local_rgb, local_met_rgb, LOCAL) ;
   get_rgb(amb_rgb, amb_met_rgb, AMBIENT) ;
   get_specular_rgb(spec_rgb, met_spec_rgb, LOCAL2) ;
   glEnable(GL_LIGHT0) ;
} else


/* clear our color and depth buffers */
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glLoadIdentity() ;

/* set our ambient material to be NULL initially (or else
 * there will be a slight leak of ambient light */
glMaterialfv(GL_FRONT, GL_AMBIENT, null_ambient) ;

/* set the diffuse material properties to the tristimulus
 * values calculated using the get_rgb() function, based
 * upon the current observer.
 */
mat_diffuse[0] = local_rgb[0][0] ;
mat_diffuse[1] = local_rgb[0][1] ;
mat_diffuse[2] = local_rgb[0][2] ;
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse) ;

/* if the ambient light scale is actually not 0.0, then we
 * need to set the ambient material properties to the
 * the ambient tristimulus values calculated with the
 * get_rgb() function.
 */
if(AmbScale != 0.0){
     mat_ambient[0] = amb_rgb[0][0] ;
     mat_ambient[1] = amb_rgb[0][1] ;
     mat_ambient[2] = amb_rgb[0][2] ;
     glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient) ;
}


/* default the specular tristimulus values to NULL as well */
glMaterialfv(GL_FRONT, GL_SPECULAR, null_ambient) ;
```

```
/* now we need to make our pretty background. basically it
 * is just a really really really big sphere, defaulted to
 * a non-selective 40% reflectance grey. essentially, this
 * provides a good mechanism to allow the user to determine
 * the color of the light source[s].
 */
glPushMatrix() ;
   glTranslated(0.0, 0.0, -10.0) ;
   glScalef(1.0, 1.0, 0.5) ;
    sphere(12.0, 80, 80)  ;
glPopMatrix() ;

/* once again, default the ambient back to NULL */
glMaterialfv(GL_FRONT, GL_AMBIENT, null_ambient) ;



/* set up our material properties for the rolling pasture, which
 * is a fine work of art modelled by none other than
 * Brad Ayers. my thanks to him for this wonderful hillside.
 */
mat_diffuse[0] = local_rgb[1][0] ;
mat_diffuse[1] = local_rgb[1][1] ;
mat_diffuse[2] = local_rgb[1][2] ;
mat_ambient[0] = local_amb[1][0] ;
mat_ambient[1] = local_amb[1][1] ;
mat_ambient[2] = local_amb[1][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

/* enable the grass texture */
glEnable(GL_TEXTURE_2D) ;
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);

glPushMatrix() ;
   glTranslatef(1.0, -0.9, -7.2) ;
   glRotatef(10.0, 1.0, 0.0, 0.0) ;
   glRotatef(23.0, 0.0, 1.0, 0.0) ;
   glRotatef(16.0, 0.0, 0.0, 1.0) ;
   glBindTexture(GL_TEXTURE_2D, (GLuint)NumTex[0]) ;
     glCallList(pasture_list) ;
glPopMatrix() ;

glDisable(GL_TEXTURE_2D) ;


/* the next bunch of lines are basically just setting the material
 * properties, positioning, and drawing the remainder of the objects.
 */
mat_diffuse[0] = local_rgb[21][0] ;
mat_diffuse[1] = local_rgb[21][1] ;
mat_diffuse[2] = local_rgb[21][2] ;
mat_ambient[0] = local_amb[21][0] ;
mat_ambient[1] = local_amb[21][1] ;
mat_ambient[2] = local_amb[21][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, mat_emit) ;

glPushMatrix() ;
   glTranslatef(3.0, 3.5, -9.0) ;
   glRotatef(100.0, 1.0, 0.0, 0.0) ;
   glCallList(moon_list)  ;
glPopMatrix() ;

mat_diffuse[0] = local_rgb[2][0] ;
mat_diffuse[1] = local_rgb[2][1] ;
mat_diffuse[2] = local_rgb[2][2] ;
mat_ambient[0] = local_amb[2][0] ;
mat_ambient[1] = local_amb[2][1] ;
mat_ambient[2] = local_amb[2][2] ;
```

```
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(0.0, -0.15, -8.0) ;
   draw_object() ;
glPopMatrix() ;

mat_diffuse[0] = local_rgb[3][0] ;
mat_diffuse[1] = local_rgb[3][1] ;
mat_diffuse[2] = local_rgb[3][2] ;
mat_ambient[0] = local_amb[3][0] ;
mat_ambient[1] = local_amb[3][1] ;
mat_ambient[2] = local_amb[3][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(3.0, -0.55, -6.0) ;
   draw_object() ;
glPopMatrix() ;

mat_diffuse[0] = local_rgb[4][0] ;
mat_diffuse[1] = local_rgb[4][1] ;
mat_diffuse[2] = local_rgb[4][2] ;
mat_ambient[0] = local_amb[4][0] ;
mat_ambient[1] = local_amb[4][1] ;
mat_ambient[2] = local_amb[4][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(-4.0, -0.55, -5.5) ;
   draw_object() ;
glPopMatrix() ;

mat_diffuse[0] = local_rgb[5][0] ,
mat_diffuse[1] = local_rgb[5][1] ;
mat_diffuse[2] = local_rgb[5][2] ;
mat_ambient[0] = local_amb[5][0] ;
mat_ambient[1] = local_amb[5][1] ;
mat_ambient[2] = local_amb[5][2] ;

glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(2.5, -0.80, -3.2) ;
   glRotatef(180.0, 0.0, 1.0, 0.0) ;
   draw_object() ;
glPopMatrix() ;

mat_diffuse[0] = local_rgb[6][0] ;
mat_diffuse[1] = local_rgb[6][1] ;
mat_diffuse[2] = local_rgb[6][2] ;
mat_ambient[0] = local_amb[6][0] ;
mat_ambient[1] = local_amb[6][1] ;
mat_ambient[2] = local_amb[6][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(-1.65, -0.90, -2.0) ;
   glRotatef(-30.0, 0.0, 1.0, 0.0) ;
   draw_object() ;
glPopMatrix() ;

mat_diffuse[0] = local_rgb[7][0] ;
mat_diffuse[1] = local_rgb[7][1] ;
mat_diffuse[2] = local_rgb[7][2] ;
mat_ambient[0] = local_amb[7][0] ;
mat_ambient[1] = local_amb[7][1] ;
```

```
mat_ambient[2] = local_amb[7][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(0.00, -2.10, -1.0) ;
   glRotatef(90.0, 0.0, 1.0, 0.0) ;
   glScalef(2.1, 2.1, 2.1) ;
   draw_object() ;
glPopMatrix() ;

mat_diffuse[0] = local_rgb[8][0] ;
mat_diffuse[1] = local_rgb[8][1] ;
mat_diffuse[2] = local_rgb[8][2] ;
mat_ambient[0] = local_amb[8][0] ;
mat_ambient[1] = local_amb[8][1] ;
mat_ambient[2] = local_amb[8][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(-3.50, -0.40, -7.0) ;
   glRotatef(190.0, 0.0, 1.0, 0.0) ;
   draw_object() ;
glPopMatrix() ;


mat_diffuse[0] = local_rgb[9][0] ;
mat_diffuse[1] = local_rgb[9][1] ;
mat_diffuse[2] = local_rgb[9][2] ;
mat_ambient[0] = local_amb[9][0] ;
mat_ambient[1] = local_amb[9][1] ;
mat_ambient[2] = local_amb[9][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(-1.50, -0.50, -6.5) ;
   glRotatef(120.0, 0.0, 1.0, 0.0) ;
   glRotatef(20.0, 0.0, 0.0, 1.0) ;
   draw_object() ;
glPopMatrix() ;


mat_diffuse[0] = local_rgb[10][0] ;
mat_diffuse[1] = local_rgb[10][1] ;
mat_diffuse[2] = local_rgb[10][2] ;
mat_ambient[0] = local_amb[10][0] ;
mat_ambient[1] = local_amb[10][1] ;
mat_ambient[2] = local_amb[10][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(1.70, -0.10, -8.5) ;
   glRotatef(-20.0, 0.0, 1.0, 0.0) ;
   glRotatef(-20.0, 0.0, 0.0, 1.0) ;
   glScalef(0.8, 0.8, 0.8) ;
   draw_object() ;
glPopMatrix() ;

mat_diffuse[0] = local_rgb[11][0] ;
mat_diffuse[1] = local_rgb[11][1] ;
mat_diffuse[2] = local_rgb[11][2] ;
mat_ambient[0] = local_amb[11][0] ;
mat_ambient[1] = local_amb[11][1] ;
mat_ambient[2] = local_amb[11][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(0.00, -0.65, -6.0) ;
```

```
   glRotatef(230.0, 0.0, 1.0, 0.0) ;
   draw_object() ;
glPopMatrix() ;

mat_diffuse[0] = local_rgb[12][0] ;
mat_diffuse[1] = local_rgb[12][1] ;
mat_diffuse[2] = local_rgb[12][2] ;
mat_ambient[0] = local_amb[12][0] ;
mat_ambient[1] = local_amb[12][1] ;
mat_ambient[2] = local_amb[12][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(4.10, 4.3, -9.5) ;
   glRotatef(190.0, 0.0, 1.0, 0.0) ;
   glRotatef(45.0, 0.0, 0.0, 1.0) ;
   draw_object() ;
glPopMatrix() ;

mat_diffuse[0] = local_rgb[13][0] ;
mat_diffuse[1] = local_rgb[13][1] ;
mat_diffuse[2] = local_rgb[13][2] ;
mat_ambient[0] = local_amb[13][0] ;
mat_ambient[1] = local_amb[13][1] ;
mat_ambient[2] = local_amb[13][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(1.8, -1.00, -1.7) ;
   glRotatef(120.0, 0.0, 1.0, 0.0) ;
   draw_object() ;
glPopMatrix() ;

mat_diffuse[0] = local_rgb[14][0] ;
mat_diffuse[1] = local_rgb[14][1] ;
mat_diffuse[2] = local_rgb[14][2] ;
mat_ambient[0] = local_amb[14][0] ;
mat_ambient[1] = local_amb[14][1] ;
mat_ambient[2] = local_amb[14][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(0.75, -1.15, -4.7) ;
   glRotatef(240.0, 0.0, 1.0, 0.0) ;
   draw_object() ;
glPopMatrix() ;


mat_diffuse[0] = local_rgb[15][0] ;
mat_diffuse[1] = local_rgb[15][1] ;
mat_diffuse[2] = local_rgb[15][2] ;
mat_ambient[0] = local_amb[15][0] ;
mat_ambient[1] = local_amb[15][1] ;
mat_ambient[2] = local_amb[15][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(-0.90, -1.25, -4.0) ;
   glRotatef(50.0, 0.0, 1.0, 0.0) ;
   glScalef(0.8, 0.8, 0.8) ;
   draw_object() ;
glPopMatrix() ;


mat_diffuse[0] = local_rgb[16][0] ;
mat_diffuse[1] = local_rgb[16][1] ;
mat_diffuse[2] = local_rgb[16][2] ;
mat_ambient[0] = local_amb[16][0] ;
```

```
mat_ambient[1] = local_amb[16][1] ;
mat_ambient[2] = local_amb[16][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(1.40, -1.25, -6.5) ;
   glRotatef(-40.0, 0.0, 1.0, 0.0) ;
   draw_object() ;
glPopMatrix() ;


mat_diffuse[0] = local_rgb[17][0] ;
mat_diffuse[1] = local_rgb[17][1] ;
mat_diffuse[2] = local_rgb[17][2] ;
mat_ambient[0] = local_amb[17][0] ;
mat_ambient[1] = local_amb[17][1] ;
mat_ambient[2] = local_amb[17][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(-2.1, -0.70, -5.3) ;
   glRotatef(-100.0, 0.0, 1.0, 0.0) ;
   glRotatef(-10.0, 0.0, 0.0, 1.0) ;
   draw_object() ;
glPopMatrix() ;


mat_diffuse[0] = local_rgb[18][0] ;
mat_diffuse[1] = local_rgb[18][1] ;
mat_diffuse[2] = local_rgb[18][2] ;
mat_ambient[0] = local_amb[18][0] ;
mat_ambient[1] = local_amb[18][1] ;
mat_ambient[2] = local_amb[18][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(-2.2, -1.30, -4.3) ;
   glRotatef(150.0, 0.0, 1.0, 0.0) ;
   glRotatef(15.0, 0.0, 0.0, 1.0) ;
   glScalef(0.9, 0.9, 0.9) ;
   draw_object() ;
glPopMatrix() ;


mat_diffuse[0] = local_rgb[19][0] ;
mat_diffuse[1] = local_rgb[19][1] ;
mat_diffuse[2] = local_rgb[19][2] ;
mat_ambient[0] = local_amb[19][0] ;
mat_ambient[1] = local_amb[19][1] ;
mat_ambient[2] = local_amb[19][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(2.0, -1.40, -4.6) ;
   glRotatef(-115.0, 0.0, 1.0, 0.0) ;
   glRotatef(-5.0, 0.0, 0.0, 1.0) ;
   draw_object() ;
glPopMatrix() ;


mat_diffuse[0] = local_rgb[20][0] ;
mat_diffuse[1] = local_rgb[20][1] ;
mat_diffuse[2] = local_rgb[20][2] ;
mat_ambient[0] = local_amb[20][0] ;
mat_ambient[1] = local_amb[20][1] ;
mat_ambient[2] = local_amb[20][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
```

```
glPushMatrix() ;
   glTranslatef(0.0, -1.35, -4.0) ;
   glRotatef(-90.0, 0.0, 1.0, 0.0) ;
   glRotatef(-5.0, 0.0, 0.0, 1.0) ;
   draw_object() ;
glPopMatrix() ;


mat_diffuse[0] = local_rgb[21][0] ;
mat_diffuse[1] = local_rgb[21][1] ;
mat_diffuse[2] = local_rgb[21][2] ;
mat_ambient[0] = local_amb[21][0] ;
mat_ambient[1] = local_amb[21][1] ;
mat_ambient[2] = local_amb[21][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(-5.3, 0.30, -8.5) ;
   glRotatef(-190.0, 0.0, 1.0, 0.0) ;
   glRotatef(-8.0, 0.0, 0.0, 1.0) ;
   glScalef(0.7, 0.7, 0.7) ;
   draw_object() ;
glPopMatrix() ;


mat_diffuse[0] = local_rgb[22][0] ;
mat_diffuse[1] = local_rgb[22][1] ;
mat_diffuse[2] = local_rgb[22][2] ;
mat_ambient[0] = local_amb[22][0] ;
mat_ambient[1] = local_amb[22][1] ;
mat_ambient[2] = local_amb[22][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(-4.2, 0.40, -8.5) ;
   glRotatef(-180.0, 0.0, 1.0, 0.0) ;
   glRotatef(-5.0, 0.0, 0.0, 1.0) ;
   glScalef(0.7, 0.7, 0.7) ;
   draw_object() ;
glPopMatrix() ;


mat_diffuse[0] = local_rgb[23][0] ;
mat_diffuse[1] = local_rgb[23][1] ;
mat_diffuse[2] = local_rgb[23][2] ;
mat_ambient[0] = local_amb[23][0] ;
mat_ambient[1] = local_amb[23][1] ;
mat_ambient[2] = local_amb[23][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

glPushMatrix() ;
   glTranslatef(-3.1, 0.45, -8.5) ;
   glRotatef(-180.0, 0.0, 1.0, 0.0) ;
   glRotatef(-3.0, 0.0, 0.0, 1.0) ;
   glScalef(0.7, 0.7, 0.7) ;
   draw_object() ;
glPopMatrix() ;


mat_diffuse[0] = local_rgb[24][0] ;
mat_diffuse[1] = local_rgb[24][1] ;
mat_diffuse[2] = local_rgb[24][2] ;
mat_ambient[0] = local_amb[24][0] ;
mat_ambient[1] = local_amb[24][1] ;
mat_ambient[2] = local_amb[24][2] ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
```

```
    glPushMatrix() ;
       glTranslatef(-2.0, 0.45, -8.5) ;
       glRotatef(-190.0, 0.0, 1.0, 0.0) ;
       glRotatef(5.0, 0.0, 0.0, 1.0) ;
       glScalef(0.7, 0.7, 0.7) ;
       draw_object() ;
    glPopMatrix() ;


    mat_diffuse[0] = local_rgb[20][0] ;
    mat_diffuse[1] = local_rgb[20][1] ;
    mat_diffuse[2] = local_rgb[20][2] ;
    mat_ambient[0] = local_amb[20][0] ;
    mat_ambient[1] = local_amb[20][1] ;
    mat_ambient[2] = local_amb[20][2] ;
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;

    glPushMatrix() ;
       glTranslatef(-0.9, 0.35, -8.5) ;
       glRotatef(-190.0, 0.0, 1.0, 0.0) ;
       glRotatef(4.0, 0.0, 0.0, 1.0) ;
       glScalef(0.7, 0.7, 0.7) ;
       draw_object() ;
    glPopMatrix() ;

    if (doubleBuffer) {
         glXSwapBuffers(display, glxwin);
    }

    else {
        glFlush() ;
    }

}
```

spect-gl.table.c

```
/*
 * spect-gl.table.c
 *
 * Written By: Garrett M. Johnson
 *
 * OpenGL rendering file for the full spectral rendering package
 * using OpenGL, and Motif.
 *
 *
 * This file contains a replacement for the default spect-gl.c It
 * still contains all the OpenGL rendering code, except now we are
 * dealing with a more complicated scene. basically, everything
 * is the same as the original spect-gl.c, except the draw() and
 * reshape functions are different. To render this image,
 * simple rename the original spect-gl.c (spect-gl.c.old etc.)
 * and rename this file spect-gl.c. Then simply re-make the
 * package, using the included Makefile, and off you go. This
 * particular image is much less flexible than the standard
 * default renderings, as it expects a specific number of
 * material files and textures.
 *
 * Last revised: 07/27/98
 *
 */
#include <stdio.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glx.h>
#include <math.h>
#include "spectral.h"
#include "glm.h"

/* the first thing we need to do is figure out what version of
 * OpenGL we have running. if we have an older version we need
 * to define the bindtexture extension equal to that of the
 * original function.
 */
#if !defined(GL_VERSION_1_1) && !defined(GL_VERSION_1_2)
#define glBindTexture glBindTextureEXT
#endif


Bool   RECALCULATE_COLORS = TRUE ;  /* recalculate the colors */
int    CUR_OBJECT = TORUS ;
static  GLfloat winWidth, winHeight ;       /* windows dimensions */
GLfloat AmbScale = 0.0 ;            /* ambient light scale */
GLfloat DiffScale = 1.0 ;           /* local light scale */

/* the following variables are all displays list variable names */
GLuint  spherelist ;
GLuint  toruslist ;
GLuint  tealist ;
GLuint  cow_list ;
GLuint  table_list ;
GLuint  cube_list ;
GLuint  l_wall_list ;
GLuint  r_wall_list ;
GLuint  b_wall_list ;
GLuint  ceiling_list ;
GLuint  floor_list ;
GLuint  fan_list ;
GLuint  outlet_list ;
```

```
GLuint  track_list ;
GLuint  window_list ;
GLuint  plane_list ;
GLuint  frame_list ;
GLuint  beet_list ;
GLuint  tric_list ;
GLuint  dragon_list ;
GLuint  ship_list ;
GLuint  stool_list ;
GLuint  chair_list ;
GLuint  hand_list ;
GLuint  face_list ;

/* the glm model */
GLMmodel *model ;

/* the scaling factor */
GLfloat scale ;


/* reshape()
 *
 * the reshape function is called when the from the resize function
 * in the spect-cb.c file. it is placed inside the spect-gl.c file
 * because it defines the gl viewpoint and all that good stuff.
 * note, this reshape function contains perspective and gluLookAt
 * (camera position) code, rather than the standard orthographic
 * projections of the default viewer. this is why the reshape()
 * function is placed inside the spect-gl.c file.
 *
 */
void
reshape(int width, int height)
{
    winWidth = width ;
    winHeight = height ;

    /* set up the viewport */
    glViewport(0, 0, winWidth, winHeight) ;
    glMatrixMode(GL_PROJECTION);

    glLoadIdentity() ;

    /* set up the perspective viewing matrix */
    gluPerspective(60.0, (GLfloat)winWidth / (GLfloat)winHeight, 1.0, 64.0);

    /* stick the camera so that it is centered slightly above the middle
     * of the scene, looking directly at the center.
     */
    gluLookAt(0.0, 1.0, 0.0, 0.0, 0.0, -10.0, 0.0, 1.0, 0.0) ;

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -3.0) ;

}

/* gl_init()
 *
 * the gl_init function is used to initialize all the good wholesome
 * OpenGL states. This is where we define the light source properties,
 * including position. Since we are kludging the actual lighting
 * so that it simulates some full-spectral goodness, we simply set all
 * the light source RGBA variables equal to 1.0. we will deal with
 * the color calculations in spect-math.c. oh yeah, in this file we
 * also initialize all the models used in the spectral viewer. if you
 * as an enduser want new cooler models, simply declare said model
 * in this function. in this particular image, we have many many
 * different models, so most of our effort is placed in initiallizing
 * all the models.
 */
 */
```

```
void
gl_init(void)
{


     /* lets turn lighting on */
     glEnable(GL_LIGHTING);
     glEnable(GL_LIGHT0);

     /* enable all the standard depth testing, face culling, and what not */
     glEnable(GL_DEPTH_TEST) ;
     glEnable(GL_CULL_FACE) ;
     glEnable(GL_AUTO_NORMAL) ;

     /* make sure we are smooth, rather than flat, shading */
     glShadeModel(GL_SMOOTH) ;

     /* generate some display lists, just in case */
     spherelist = glGenLists(1) ;
     toruslist = glGenLists(1) ;
     tealist = glGenLists(1) ;

     /* now we spend a good deal of time initializing the object models..
      * once again, this is a good time to plug Nate Robbins very
      * impressive glm Rendering library.
      */
     model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/cow.obj") ;
     scale = glmUnitize(model) ;
     glmFacetNormals(model) ;
     glmVertexNormals(model, 90.0) ;
     glmScale(model, 0.8) ;
     glmLinearTexture(model) ;

     cow_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE) ;

     model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/table.obj")   ;
     scale = glmUnitize(model) ;
     glmFacetNormals(model) ;
     glmVertexNormals(model, 90.0) ;
     glmScale(model, 3.5) ;

     table_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE ) ;


     model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/l_wall.obj")   ;

     glmFacetNormals(model) ;
     glmVertexNormals(model, 90.0) ;
     glmScale(model, 1.2) ;

     l_wall_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE ) ;

     /* still initializing. */
     model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/r_wall.obj")   ;
     glmFacetNormals(model) ;
     glmVertexNormals(model, 90.0) ;
     glmScale(model, 1.2) ;

     /* got to get a little tricky for a minute...the normals were facing the
      * wronge way on this model, so lets reverse wind them to get them facing
      * the right way.
      */
     glmReverseWinding(model) ;

     r_wall_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE ) ;


     /* still initializing */
     model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/b_wall.obj")   ;
     glmFacetNormals(model) ;
     glmVertexNormals(model, 90.0) ;
     glmScale(model, 1.2) ;
```

```
b_wall_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE ) ;

model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/ceiling.obj")   ;
glmFacetNormals(model) ;
glmVertexNormals(model, 90.0) ;
glmScale(model, 1.2) ;
glmReverseWinding(model) ;

ceiling_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE ) ;

model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/floor.obj")   ;
glmFacetNormals(model) ;
glmScale(model, 1.2) ;
glmVertexNormals(model, 90.0) ;

floor_list = glmList(model, GLM_SMOOTH | GLM_TEXTURE ) ;

model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/outlet.obj")   ;
scale = glmUnitize(model) ;
glmFacetNormals(model) ;
glmVertexNormals(model, 90.0) ;
glmScale(model, .5) ;

outlet_list = glmList(model, GLM_SMOOTH ) ;

model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/fan.obj")   ;
scale = glmUnitize(model) ;
glmFacetNormals(model) ;
glmVertexNormals(model, 90.0) ;
glmScale(model, 2.0) ;

fan_list = glmList(model, GLM_SMOOTH ) ;

model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/trackLights.obj")   ;
scale = glmUnitize(model) ;
glmFacetNormals(model) ;
glmVertexNormals(model, 90.0) ;
glmScale(model, 3.0) ;

track_list = glmList(model, GLM_SMOOTH ) ;


model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/plane.obj")   ;
scale = glmUnitize(model) ;
glmFacetNormals(model) ;
glmVertexNormals(model, 90.0) ;
glmScale(model, 2.5) ;

plane_list = glmList(model, GLM_SMOOTH ) ;

model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/window.obj")   ;
scale = glmUnitize(model) ;
glmFacetNormals(model) ;
glmVertexNormals(model, 90.0) ;
glmScale(model, 2.7) ;

window_list = glmList(model, GLM_SMOOTH ) ;


model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/frame3.obj")   ;
scale = glmUnitize(model) ;
glmFacetNormals(model) ;
glmVertexNormals(model, 90.0) ;
glmScale(model, 2.7) ;

frame_list = glmList(model, GLM_SMOOTH ) ;

model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/beethoven.obj")   ;
scale = glmUnitize(model) ;
glmFacetNormals(model) ;
glmVertexNormals(model, 90.0) ;
```

```
    glmScale(model, 0.8) ;

    beet_list = glmList(model, GLM_SMOOTH ) ;

    model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/galleon.obj")   ;
    scale = glmUnitize(model) ;
    glmFacetNormals(model) ;
    glmVertexNormals(model, 90.0) ;
    glmScale(model, 0.8) ;

    ship_list = glmList(model, GLM_SMOOTH ) ;

    model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/triceratops.obj")   ;
    scale = glmUnitize(model) ;
    glmFacetNormals(model) ;
    glmVertexNormals(model, 90.0) ;
    glmScale(model, 0.8) ;

    tric_list = glmList(model, GLM_SMOOTH ) ;

    /* like that silly pink bunny...we are still in the old model
     * initialization mode.
     */
    model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/dragon.obj")   ;
    scale = glmUnitize(model) ;
    glmFacetNormals(model) ;
    glmVertexNormals(model, 90.0) ;
    glmScale(model, 0.8) ;
    glmReverseWinding(model) ;

    dragon_list = glmList(model, GLM_SMOOTH ) ;

    model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/stool.obj")   ;
    scale = glmUnitize(model) ;
    glmFacetNormals(model) ;
    glmVertexNormals(model, 90.0) ;
    glmScale(model, 1.3) ;
    glmReverseWinding(model) ;

    stool_list = glmList(model, GLM_SMOOTH ) ;

    model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/Chair.obj")   ;
    scale = glmUnitize(model) ;
    glmFacetNormals(model) ;
    glmVertexNormals(model, 90.0) ;
    glmScale(model, 2.0) ;

    chair_list = glmList(model, GLM_SMOOTH ) ;

    model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/face.obj")   ;
    scale = glmUnitize(model) ;
    glmFacetNormals(model) ;
    glmVertexNormals(model, 90.0) ;
    glmScale(model, 0.8) ;

    face_list = glmList(model, GLM_SMOOTH ) ;

    model = glmReadOBJ("/remote/user2/users/garrett/spectral/obj/Hand.obj")   ;
    scale = glmUnitize(model) ;
    glmFacetNormals(model) ;
    glmVertexNormals(model, 90.0) ;
    glmScale(model, 1.4) ;

    hand_list  = glmList(model, GLM_SMOOTH ) ;

    /* phew...after all that we are finally done initializing. clear
     * the color now.
     */
    glClearColor(0.0, 0.0, 0.0, 0.0)   ;
    glColor3f(1.0, 0.0, 0.0)   ;

}
```

```
/* draw_object()
 *
 * this is actually not used in this file, but it could be left over
 * from the original spect-gl.c, so it is left in as a place holder
 * for the benefit of all who happen to be reading.
 */
void
draw_object(void)
{
}

/* draw()
 *
 * as you might recall from the original draw() function, this is
 * where the meat of the rendering is done. ok, this is where all
 * of the rendering is done. basically, it is the same idea as
 * what what going on in the original. luckily, now that we are
 * dealing with more realistic scenes, we don't have to worry
 * about those dang metameric pairs (aka a single object getting
 * cut in half). as in the real world, if two objects are truely
 * metameric, then they would have two distinct material properties..
 * and two distinct material property files. yeeee-haw.
 */
void
draw(GLenum mode)
{
    static    double       amb_rgb[24][3] ;
    static    double       local_rgb[24][3] ;
    static    double       spec_rgb[24][3] ;
    static    double       null_rgb[24][3] ;
    GLfloat   ambient[] = { 0.2, 0.2, 0.2, 1.0 };
    GLfloat   diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat   specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat   position[] = { 0.0, 0.0, 0.0, 1.0 };
    GLfloat     mat_diffuse[] = {1.0, 1.0, 1.0, 1.0} ;
    GLfloat     mat_specular[] = {1.0, 1.0, 1.0, 1.0} ;
    GLfloat     null_specular[] = {0.0, 0.0, 0.0, 1.0} ;
    GLfloat     mat_ambient[] = {1.0, 1.0, 1.0, 1.0} ;
    GLfloat     mat_emit[] = {1.0, 1.0, 1.0, 1.0} ;
    GLfloat     cur_gloss ;
    GLfloat   *pic_tex = 0 ; /* ah, could it be a texture map? */
    int       i, Redraw ;


    /* default light and material properties */
    ambient[0] = ambient[1] = ambient[2] = AmbScale ;
    diffuse[0] = diffuse[1] = diffuse[2] = DiffScale ;
    specular[0] = specular[1] = specular[2] = DiffScale  ;

    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
    glLightfv(GL_LIGHT0, GL_POSITION, position);


    /* we better allocate some memory, in case we want to fill up
     * that "texture" later, hint hint hint.
     */
    pic_tex = (GLfloat *)malloc(512 * 512 * 3*sizeof(GLfloat) ) ;

    /* make sure we really need to recalculate all the
     * colors.
     */
    if(RECALCULATE_COLORS){
        get_rgb(local_rgb, null_rgb, LOCAL) ;
        get_rgb(amb_rgb, null_rgb, AMBIENT) ;
        get_specular_rgb(spec_rgb, null_rgb, LOCAL) ;
        RECALCULATE_COLORS = FALSE ;
    }
```

```
/* clear all the necessary buffers */
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glLoadIdentity() ;


/* make sure we aren't emitting any stray light */
mat_emit[0] = mat_emit[1] = mat_emit[2] = 0.0 ;


/* set the diffuse, specular, ambient, and gloss material properties */
mat_diffuse[0] = local_rgb[0][0] ;
mat_diffuse[1] = local_rgb[0][1] ;
mat_diffuse[2] = local_rgb[0][2] ;
mat_specular[0] = spec_rgb[0][0] ;
mat_specular[1] = spec_rgb[0][1] ;
mat_specular[2] = spec_rgb[0][2] ;
mat_ambient[0] = amb_rgb[0][0] ;
mat_ambient[1] = amb_rgb[0][1] ;
mat_ambient[2] = amb_rgb[0][2] ;
cur_gloss = (GLfloat)gloss[0] ;

/* declare some of those material properties */
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

/* declare the ambient, if we need it */
if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}

/* declare emission NULL */
glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, mat_emit) ;


/* declare the specular, if there is any */
if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}


/* lets calculate this so the lighting is slightly more accurate,
 * aka, based up a local viewer rather than an infinite viewer.
 */
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);


/* enable texture mapping, for now */
glEnable(GL_TEXTURE_2D) ;
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE) ;


/* load the name, incase we want to pick */
if(mode == GL_SELECT){
   glLoadName(1) ;
}

/* move ourselves into position for the floor, with a nice
 * marble texture mapped onto it.
 */
glPushMatrix() ;
   glTranslatef(0.0, 0.0, -9.8) ;
   glRotatef(48.0, 0.0, 1.0, 0.0) ;

   /* bind the texture, and then draw the floor */
   glBindTexture(GL_TEXTURE_2D, (GLuint) NumTex[0]) ;
```

```
   glCallList(floor_list)   ;
glPopMatrix() ;


/* set up the new material properties for the walls */
mat_diffuse[0] = local_rgb[1][0] ;
mat_diffuse[1] = local_rgb[1][1] ;
mat_diffuse[2] = local_rgb[1][2] ;
mat_specular[0] = spec_rgb[1][0] ;
mat_specular[1] = spec_rgb[1][1] ;
mat_specular[2] = spec_rgb[1][2] ;
mat_ambient[0] = amb_rgb[1][0] ;
mat_ambient[1] = amb_rgb[1][1] ;
mat_ambient[2] = amb_rgb[1][2] ;
cur_gloss = (GLfloat)gloss[1] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}


if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}

if(mode == GL_SELECT){
   glLoadName(2) ;
}

/* bind a new texture, and draw the walls and ceiling */
glPushMatrix() ;
   glTranslatef(0.0, 0.0, -9.8) ;
   glRotatef(-42.0, 0.0, 1.0, 0.0) ;
        glBindTexture(GL_TEXTURE_2D, (GLuint) NumTex[1]) ;
        glCallList(l_wall_list)   ;
        glCallList(r_wall_list)   ;
        glCallList(b_wall_list)   ;
        glCallList(ceiling_list)   ;
glPopMatrix() ;



/* new material properties, for a new object */
mat_diffuse[0] = local_rgb[2][0] ;
mat_diffuse[1] = local_rgb[2][1] ;
mat_diffuse[2] = local_rgb[2][2] ;
mat_specular[0] = spec_rgb[2][0] ;
mat_specular[1] = spec_rgb[2][1] ;
mat_specular[2] = spec_rgb[2][2] ;
mat_ambient[0] = amb_rgb[2][0] ;
mat_ambient[1] = amb_rgb[2][1] ;
mat_ambient[2] = amb_rgb[2][2] ;
cur_gloss = (GLfloat)gloss[2] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}
```

```
if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}



if(mode == GL_SELECT){
    glLoadName(3) ;
}


/* new texture, move into position, and then draw the table */
glPushMatrix() ;
   glTranslatef(-0.2, -4.4, -12.9) ;
   glRotatef(-90.0, 1.0, 0.0, 0.0) ;
   glRotatef(-48.0, 0.0, 0.0, 1.0) ;
   glBindTexture(GL_TEXTURE_2D, (GLuint) NumTex[2]) ;
    glCallList(table_list) ;
glPopMatrix() ;

/* we are done with textures for now */
glDisable(GL_TEXTURE_2D) ;


/* new material properties, for the little cow */
mat_diffuse[0] = local_rgb[3][0] ;
mat_diffuse[1] = local_rgb[3][1] ;
mat_diffuse[2] = local_rgb[3][2] ;
mat_specular[0] = spec_rgb[3][0] ;
mat_specular[1] = spec_rgb[3][1] ;
mat_specular[2] = spec_rgb[3][2] ;
mat_ambient[0] = amb_rgb[3][0] ;
mat_ambient[1] = amb_rgb[3][1] ;
mat_ambient[2] = amb_rgb[3][2] ;
cur_gloss = (GLfloat)gloss[3] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}


if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}

if(mode == GL_SELECT){
    glLoadName(4) ;
}


/* position and draw the cow */
glPushMatrix() ;
   glTranslatef(-1.0, -2.0, -12.5) ;
   glRotatef(0.0, 1.0, 0.0, 0.0) ;
   glCallList(cow_list)  ;
glPopMatrix() ;


/* new material properties for the triceratops */
mat_diffuse[0] = local_rgb[4][0] ;
mat_diffuse[1] = local_rgb[4][1] ;
```

```
mat_diffuse[2] = local_rgb[4][2] ;
mat_specular[0] = spec_rgb[4][0] ;
mat_specular[1] = spec_rgb[4][1] ;
mat_specular[2] = spec_rgb[4][2] ;
mat_ambient[0] = amb_rgb[4][0] ;
mat_ambient[1] = amb_rgb[4][1] ;
mat_ambient[2] = amb_rgb[4][2] ;
cur_gloss = (GLfloat)gloss[4] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}


if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else(
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}

if(mode == GL_SELECT){
    glLoadName(5) ;
}


/* position and draw the triceratops */
glPushMatrix() ;
   glTranslatef(1.0, -2.0, -11.5) ;
   glRotatef(0.0, 1.0, 0.0, 0.0) ;
   glCallList(tric_list)   ;
glPopMatrix() ;


/* new material properties for the boat */
mat_diffuse[0] = local_rgb[5][0] ;
mat_diffuse[1] = local_rgb[5][1] ;
mat_diffuse[2] = local_rgb[5][2] ;
mat_specular[0] = spec_rgb[5][0] ;
mat_specular[1] = spec_rgb[5][1] ;
mat_specular[2] = spec_rgb[5][2] ;
mat_ambient[0] = amb_rgb[5][0] ;
mat_ambient[1] = amb_rgb[5][1] ;
mat_ambient[2] = amb_rgb[5][2] ;
cur_gloss = (GLfloat)gloss[5] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}


if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}

if(mode == GL_SELECT){
    glLoadName(6) ;
}
```

```
/* position and draw the boat */
glPushMatrix() ;
   glTranslatef(-2.2, -2.0, -11.0) ;
   glRotatef(0.0, 1.0, 0.0, 0.0) ;
   glCallList(ship_list)   ;
glPopMatrix() ;


/* new material properties for the cow, again */
mat_diffuse[0] = local_rgb[6][0] ;
mat_diffuse[1] = local_rgb[6][1] ;
mat_diffuse[2] = local_rgb[6][2] ;
mat_specular[0] = spec_rgb[6][0] ;
mat_specular[1] = spec_rgb[6][1] ;
mat_specular[2] = spec_rgb[6][2] ;
mat_ambient[0] = amb_rgb[6][0] ;
mat_ambient[1] = amb_rgb[6][1] ;
mat_ambient[2] = amb_rgb[6][2] ;
cur_gloss = (GLfloat)gloss[6] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}


if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}

if(mode == GL_SELECT){
    glLoadName(7) ;
}


/* position and draw this new cow */
glPushMatrix() ;
   glTranslatef(1.2, -2.0, -10.0) ;
   glRotatef(40.0, 0.0, 1.0, 0.0) ;
   glCallList(cow_list)   ;
glPopMatrix() ;


/* material properties for yet another boat */
mat_diffuse[0] = local_rgb[7][0] ;
mat_diffuse[1] = local_rgb[7][1] ;
mat_diffuse[2] = local_rgb[7][2] ;
mat_specular[0] = spec_rgb[7][0] ;
mat_specular[1] = spec_rgb[7][1] ;
mat_specular[2] = spec_rgb[7][2] ;
mat_ambient[0] = amb_rgb[7][0] ;
mat_ambient[1] = amb_rgb[7][1] ;
mat_ambient[2] = amb_rgb[7][2] ;
cur_gloss = (GLfloat)gloss[7] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}
```

```
if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}

if(mode == GL_SELECT){
    glLoadName(8) ;
}


/* position and draw the boat */
glPushMatrix() ;
   glTranslatef(0.0, -2.0, -10.0) ;
   glRotatef(-90.0, 0.0, 1.0, 0.0) ;
   glCallList(ship_list)   ;
glPopMatrix() ;


/* new material properties for yet another triceratops */
mat_diffuse[0] = local_rgb[8][0] ;
mat_diffuse[1] = local_rgb[8][1] ;
mat_diffuse[2] = local_rgb[8][2] ;
mat_specular[0] = spec_rgb[8][0] ;
mat_specular[1] = spec_rgb[8][1] ;
mat_specular[2] = spec_rgb[8][2] ;
mat_ambient[0] = amb_rgb[8][0] ;
mat_ambient[1] = amb_rgb[8][1] ;
mat_ambient[2] = amb_rgb[8][2] ;
cur_gloss = (GLfloat)gloss[8] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}


if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}

if(mode == GL_SELECT){
    glLoadName(9) ;
}


/* position and draw the triceratops */
glPushMatrix() ;
   glTranslatef(-1.0, -2.0, -9.0) ;
   glRotatef(-90.0, 0.0, 1.0, 0.0) ;
   glCallList(tric_list)   ;
glPopMatrix() ;


/* material properties for the stool */
mat_diffuse[0] = local_rgb[9][0] ;
mat_diffuse[1] = local_rgb[9][1] ;
mat_diffuse[2] = local_rgb[9][2] ;
mat_specular[0] = spec_rgb[9][0] ;
mat_specular[1] = spec_rgb[9][1] ;
mat_specular[2] = spec_rgb[9][2] ;
mat_ambient[0] = amb_rgb[9][0] ;
mat_ambient[1] = amb_rgb[9][1] ;
```

```
mat_ambient[2] = amb_rgb[9][2] ;
cur_gloss = (GLfloat)gloss[9] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}


if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}

if(mode == GL_SELECT){
    glLoadName(10) ;
}


/* position and draw the stool */
glPushMatrix() ;
   glTranslatef(-4.5, -4.4, -10.0) ;
   glRotatef(-90.0, 1.0, 0.0, 0.0) ;
   glCallList(stool_list)   ;
glPopMatrix() ;


/* material properties for the bust of beethoven */
mat_diffuse[0] = local_rgb[10][0] ;
mat_diffuse[1] = local_rgb[10][1] ;
mat_diffuse[2] = local_rgb[10][2] ;
mat_specular[0] = spec_rgb[10][0] ;
mat_specular[1] = spec_rgb[10][1] ;
mat_specular[2] = spec_rgb[10][2] ;
mat_ambient[0] = amb_rgb[10][0] ;
mat_ambient[1] = amb_rgb[10][1] ;
mat_ambient[2] = amb_rgb[10][2] ;
cur_gloss = (GLfloat)gloss[10] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}


if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}


if(mode == GL_SELECT){
    glLoadName(11) ;
}


/* position and draw good old beethoven */
glPushMatrix() ;
   glTranslatef(-4.5, -2.5, -10.0) ;
   glRotatef(70.0, 0.0, 1.0, 0.0) ;
```

```
      glCallList(beet_list)   ;
   glPopMatrix() ;


   /* material properties for a chair */
   mat_diffuse[0] = local_rgb[11][0] * 0.5;
   mat_diffuse[1] = local_rgb[11][1] * 0.5;
   mat_diffuse[2] = local_rgb[11][2] * 0.5;
   mat_specular[0] = spec_rgb[11][0] ;
   mat_specular[1] = spec_rgb[11][1] ;
   mat_specular[2] = spec_rgb[11][2] ;
   mat_ambient[0] = amb_rgb[11][0] ;
   mat_ambient[1] = amb_rgb[11][1] ;
   mat_ambient[2] = amb_rgb[11][2] ;
   cur_gloss = (GLfloat)gloss[11] ;
   glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

   if(AmbScale != 0.0){
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
   }
   else{
      glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
   }


   if(cur_gloss > 0.0){
        glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
   }
   else(
        glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
   }

   if(mode == GL_SELECT){
       glLoadName(12) ;
   }


   /* position and draw the first chair */
   glPushMatrix() ;
      glTranslatef(4.5, -4.0, -12.0) ;
      glRotatef(-42.0, 0.0, 1.0, 0.0) ;
      glCallList(chair_list)   ;
   glPopMatrix() ;


   /* material properties of the hand...talk to the hand... */
   mat_diffuse[0] = local_rgb[12][0] ;
   mat_diffuse[1] = local_rgb[12][1] ;
   mat_diffuse[2] = local_rgb[12][2] ;
   mat_specular[0] = spec_rgb[12][0] ;
   mat_specular[1] = spec_rgb[12][1] ;
   mat_specular[2] = spec_rgb[12][2] ;
   mat_ambient[0] = amb_rgb[12][0] ;
   mat_ambient[1] = amb_rgb[12][1] ;
   mat_ambient[2] = amb_rgb[12][2] ;
   cur_gloss = (GLfloat)gloss[12] ;
   glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
   glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

   if(AmbScale != 0.0){
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
   }
   else{
      glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
   }


   if(cur_gloss > 0.0){
        glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
   }
   else{
```

```
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}

if(mode == GL_SELECT){
    glLoadName(13) ;
}


/* position and draw the hand, man */
glPushMatrix() ;
   glTranslatef(1.5, 1.0, -15.0) ;
   glRotatef(48.0, 0.0, 1.0, 0.0) ;
   glRotatef(70.0, 0.0, 0.0, 1.0) ;
   glCallList(hand_list)  ;
glPopMatrix() ;


/* material properties of the face, in the wall...ah to be a face
 * in a wall
 */
mat_diffuse[0] = local_rgb[13][0]*.5 ;
mat_diffuse[1] = local_rgb[13][1]*.5 ;
mat_diffuse[2] = local_rgb[13][2]*.5 ;
mat_specular[0] = spec_rgb[13][0] ;
mat_specular[1] = spec_rgb[13][1] ;
mat_specular[2] = spec_rgb[13][2] ;
mat_ambient[0] = amb_rgb[13][0] ;
mat_ambient[1] = amb_rgb[13][1] ;
mat_ambient[2] = amb_rgb[13][2] ;
cur_gloss = (GLfloat)gloss[13] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}


if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}

if(mode == GL_SELECT){
    glLoadName(14) ;
}


/* position and draw the face */
glPushMatrix() ;
   glTranslatef(-1.5, 1.0, -14.8) ;
   glRotatef(48.0, 0.0, 1.0, 0.0) ;
   glCallList(face_list)  ;
glPopMatrix() ;


/* material properties of the two wall outlets */
mat_diffuse[0] = local_rgb[14][0] ;
mat_diffuse[1] = local_rgb[14][1] ;
mat_diffuse[2] = local_rgb[14][2] ;
mat_specular[0] = spec_rgb[14][0] ;
mat_specular[1] = spec_rgb[14][1] ;
mat_specular[2] = spec_rgb[14][2] ;
mat_ambient[0] = amb_rgb[14][0] ;
mat_ambient[1] = amb_rgb[14][1] ;
mat_ambient[2] = amb_rgb[14][2] ;
cur_gloss = (GLfloat)gloss[14] ;
```

```
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else(
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}


if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}

if(mode == GL_SELECT){
    glLoadName(15) ;
}

/* position and draw the first outlet */
glPushMatrix() ;
   glTranslatef(-7.2, -4.9, -10.6) ;
   glRotatef(48.0, 0.0, 1.0, 0.0) ;
   glCallList(outlet_list)   ;
glPopMatrix() ;

/* now position and draw the second outlet */
glPushMatrix() ;
   glTranslatef(7.2, -4.9, -10.6) ;
   glRotatef(-48.0, 0.0, 1.0, 0.0) ;
   glCallList(outlet_list)   ;
glPopMatrix() ;


/* material properties for the hanging wall fan */
mat_diffuse[0] = local_rgb[15][0] * 0.5;
mat_diffuse[1] = local_rgb[15][1] * 0.5;
mat_diffuse[2] = local_rgb[15][2] * 0.5;
mat_specular[0] = spec_rgb[15][0] ;
mat_specular[1] = spec_rgb[15][1] ;
mat_specular[2] = spec_rgb[15][2] ;
mat_ambient[0] = amb_rgb[15][0] ;
mat_ambient[1] = amb_rgb[15][1] ;
mat_ambient[2] = amb_rgb[15][2] ;
cur_gloss = (GLfloat)gloss[15] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0)(
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}


if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}

if(mode == GL_SELECT){
    glLoadName(16) ;
}


/* position and draw the fan */
```

```
glPushMatrix() ;
   glTranslatef(0.0, 4.9, -10.0) ;
   glRotatef(-90.0, 1.0, 0.0, 0.0) ;
   glCallList(fan_list) ;
glPopMatrix() ;


/* material properties of the track lighting. */
mat_diffuse[0] = local_rgb[16][0] ;
mat_diffuse[1] = local_rgb[16][1] ;
mat_diffuse[2] = local_rgb[16][2] ;
mat_specular[0] = spec_rgb[16][0] ;
mat_specular[1] = spec_rgb[16][1] ;
mat_specular[2] = spec_rgb[16][2] ;
mat_ambient[0] = amb_rgb[16][0] ;
mat_ambient[1] = amb_rgb[16][1] ;
mat_ambient[2] = amb_rgb[16][2] ;
cur_gloss = (GLfloat)gloss[16] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}


if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}


if(mode == GL_SELECT){
    glLoadName(17) ;
}


/* position and draw the track lighting */
glPushMatrix() ;
   glTranslatef(3.6, 3.3, -9.5) ;
   glRotatef(-20.0, 1.0, 0.0, 0.0) ;
   glRotatef(-48.0, 0.0, 1.0, 0.0) ;
   glRotatef(-16.0, 0.0, 0.0, 1.0) ;
   glCallList(track_list)   ;
glPopMatrix() ;


/* material properties of a simple plane (not airplane) */
mat_diffuse[0] = local_rgb[17][0] ;
mat_diffuse[1] = local_rgb[17][1] ;
mat_diffuse[2] = local_rgb[17][2] ;
mat_specular[0] = spec_rgb[17][0] ;
mat_specular[1] = spec_rgb[17][1] ;
mat_specular[2] = spec_rgb[17][2] ;
mat_ambient[0] = amb_rgb[17][0] ;
mat_ambient[1] = amb_rgb[17][1] ;
mat_ambient[2] = amb_rgb[17][2] ;
cur_gloss = (GLfloat)gloss[17] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}
```

```
if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}


if(mode == GL_SELECT){
    glLoadName(18) ;
}


/* position and draw the plane */
glPushMatrix() ;
   glTranslatef(-4.8, 1.5, -13.4) ;
   glRotatef(48.0, 0.0, 1.0, 0.0) ;
   glRotatef(-2.0, 0.0, 0.0, 1.0) ;
   glCallList(plane_list)   ;
glPopMatrix() ;


/* material properties for the window */
mat_diffuse[0] = local_rgb[18][0] ;
mat_diffuse[1] = local_rgb[18][1] ;
mat_diffuse[2] = local_rgb[18][2] ;
mat_specular[0] = spec_rgb[18][0] ;
mat_specular[1] = spec_rgb[18][1] ;
mat_specular[2] = spec_rgb[18][2] ;
mat_ambient[0] = amb_rgb[18][0] ;
mat_ambient[1] = amb_rgb[18][1] ;
mat_ambient[2] = amb_rgb[18][2] ;
cur_gloss = (GLfloat)gloss[18] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0)(
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}


if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}

if(mode == GL_SELECT){
    glLoadName(19) ;
}


/* position and draw the window */
glPushMatrix() ;
   glTranslatef(-4.8, 1.5, -13.4) ;
   glRotatef(48.0, 0.0, 1.0, 0.0) ;
   glRotatef(-2.0, 0.0, 0.0, 1.0) ;
   glScalef(1.2, 1.0, 1.0) ;
   glCallList(window_list)   ;
glPopMatrix() ;


/* material properties for the window frame */
mat_diffuse[0] = local_rgb[19][0] ;
mat_diffuse[1] = local_rgb[19][1] ;
mat_diffuse[2] = local_rgb[19][2] ;
```

```
mat_specular[0] = spec_rgb[19][0] ;
mat_specular[1] = spec_rgb[19][1] ;
mat_specular[2] = spec_rgb[19][2] ;
mat_ambient[0] = amb_rgb[19][0] ;
mat_ambient[1] = amb_rgb[19][1] ;
mat_ambient[2] = amb_rgb[19][2] ;
cur_gloss = (GLfloat)gloss[19] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}


if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}

if(mode == GL_SELECT){
    glLoadName(20) ;
}


/* position and draw the window frame */
glPushMatrix() ;
   glTranslatef(5.6, 1.5, -12.7) ;
   glRotatef(-42.0, 0.0, 1.0, 0.0) ;
   glRotatef(4.0, 0.0, 0.0, 1.0) ;
   glScalef(1.2, 1.0, 1.0) ;
   glCallList(frame_list)   ;
glPopMatrix() ;


/* material properties of the other chair. note, this chair
 * is designed to be a metameric match to the first chair
 */
mat_diffuse[0] = local_rgb[20][0] * 0.5;
mat_diffuse[1] = local_rgb[20][1] * 0.5;
mat_diffuse[2] = local_rgb[20][2] * 0.5;
mat_specular[0] = spec_rgb[20][0] ;
mat_specular[1] = spec_rgb[20][1] ;
mat_specular[2] = spec_rgb[20][2] ;
mat_ambient[0] = amb_rgb[20][0] ;
mat_ambient[1] = amb_rgb[20][1] ;
mat_ambient[2] = amb_rgb[20][2] ;
cur_gloss = (GLfloat)gloss[20] ;
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, cur_gloss) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse) ;

if(AmbScale != 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient) ;
}
else{
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, null_specular) ;
}


if(cur_gloss > 0.0){
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular) ;
}
else{
     glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, null_specular) ;
}

if(mode == GL_SELECT){
```

```
        glLoadName(21) ;
    }


    /* position and draw this new chair */
    glPushMatrix() ;
       glTranslatef(4.5, -4.0, -12.0) ;
       glRotatef(-42.0, 0.0, 1.0, 0.0) ;
       glCallList(chair_list)    ;
    glPopMatrix() ;



    /* now we get into some good computing stuff...or color...i guess. */

    /* first enable texturing */
    glEnable(GL_TEXTURE_2D) ;

    /* set the texture mapping mode to decal (replace color) */
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL) ;

    /* set up the mipmap parameters */
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR_MIPMAP_LINEAR) ;

    /* set the pixel storage */
    glPixelStorei(GL_PACK_ALIGNMENT, 1) ;

    /* now read the current buffer into the texture array */
    glReadPixels(80, 20, 512, 512, GL_RGB, GL_FLOAT, pic_tex) ;

    /* bind that texture array into a texture map */
    glBindTexture(GL_TEXTURE_2D, (GLint)NumTex[3]) ;

    /* build a mip-map using the data from the frame buffer */
    gluBuild2DMipmaps(GL_TEXTURE_2D, 3, 512,
                      512,  GL_RGB,
                      GL_FLOAT, pic_tex) ;

    /* now position ourselves where we need to be */
    glPushMatrix() ;
       glTranslatef(5.2, 1.5, -12.3) ;
       glRotatef(-42.0, 0.0, 1.0, 0.0) ;
       glScalef(0.5, 0.4, 0.1) ;

       /* bind that texture we just made to a new object, and draw */
         glBindTexture(GL_TEXTURE_2D, (GLuint)NumTex[3]) ;
       glCallList(b_wall_list)    ;
    glPopMatrix() ;

    /* disable the texturing. in case it wasn't too obvious, we just
     * create a real full spectral texture map. since a full spectral
     * texture isn't too easy to come by (hence in part this particular
     * rendering package...) the easiest way to get one was to create our
     * own..which is just what we did above.
     */
    glDisable(GL_TEXTURE_2D) ;

    glFlush() ;

}
```

Makefile

```
all: spectral

OBJS = spect-init.o spect-glx-init.o spect-gl.o spect-read.o spect-cb.o\
       spect-math.o spect-write.o shapes.o spectral.o glm.o sovlayerutil.o\
       gltx.o SciPlot.o pbutil.o

spectral : $(OBJS)
       cc -ansi -s -o spectral $(OBJS) -I"/remote/user2/users/garrett/tmp/hdf/include"\
                      -L"/remote/user2/users/garrett/tmp/hdf/lib" \
                      -lmfhdf -ldf -lz -lGLw -lGLU -lGL -lXm -lXt -lXext -lX11 -lm

#      cc -o spectral $(OBJS)
# The molview_sgi version uses SGI's nicer file chooser widget by linking
# with -lSgm
#

spectral_sgi : $(OBJS)
       cc -ansi -s -o spectral $(OBJS) -I"/remote/user2/users/garrett/tmp/hdf/include"\
                            -L"/remote/user2/users/garrett/tmp/hdf/lib" \
                            -lmfhdf -ldf -lz -lSgm -lGLw -lGLU -lGL -lXm -lXt\
                             -lXext -lX11 -lm

clean:
       -rm -f *.o
       -rm spectral spectral_sgi
```

C. Appendix C: Example Material Properties File

```
#
#   Material:       Macbeth ColorCheckser - path 1   dark skin
#   Source:
#      reflectance:  McCamy, C. S., H. Marcus, and J. G. Davidson (1976)
#                         ``A Color Rendition Chart,''
#                         Journal of Applied Photographic Engineering,
#                        vol. 11, no. 3, pp. 95-99.
#                    Scanned data from article by:
#                        Program of Computer Graphics
#                        Cornell University
#
#t rgb/wood0.rgb
#g 128.0
300 0.0000
305 0.0000
310 0.0000
315 0.0000
320 0.0000
325 0.0000
330 0.0000
335 0.0000
340 0.0000
345 0.0000
350 0.0000
355 0.0000
360 0.0000
365 0.0000
370 0.0000
375 0.0000
380 0.0573
385 0.0625
390 0.0671
395 0.0707
400 0.0730
405 0.0739
410 0.0736
415 0.0723
420 0.0705
425 0.0682
430 0.0659
435 0.0636
440 0.0615
445 0.0597
450 0.0582
455 0.0573
460 0.0569
465 0.0569
470 0.0574
475 0.0582
480 0.0593
485 0.0607
490 0.0623
495 0.0641
500 0.0661
505 0.0681
510 0.0702
515 0.0724
520 0.0747
525 0.0769
530 0.0791
535 0.0811
```

```
540  0.0830
545  0.0844
550  0.0861
555  0.0888
560  0.0929
565  0.0979
570  0.1035
575  0.1094
580  0.1154
585  0.1214
590  0.1274
595  0.1334
600  0.1393
605  0.1451
610  0.1507
615  0.1561
620  0.1615
625  0.1669
630  0.1725
635  0.1783
640  0.1845
645  0.1913
650  0.1988
655  0.2070
660  0.2158
665  0.2251
670  0.2348
675  0.2446
680  0.2546
685  0.2646
690  0.2746
695  0.2847
700  0.2947
705  0.2947
710  0.2947
715  0.2947
720  0.2947
725  0.2947
730  0.2947
735  0.2947
740  0.2947
745  0.2947
750  0.2947
755  0.2947
760  0.2947
```