THREE-DIMENSIONAL
DEPTH PERCEPTION ENHANCEMENT
BY DYNAMIC LIGHTING

Technical Report 85-011

by

David H. Holmes

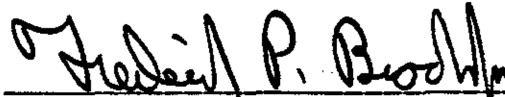
A Thesis submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill

1985

Approved by:

Advisor: Henry Fuchs



Reader: Frederick P. Brooks, Jr.

Reader: Stephen M. Pizer

DAVID HENDRICKSON HOLMES. Three-Dimensional Depth Perception Enhancement by Dynamic Lighting. (Under the direction of HENRY FUCHS.)

ABSTRACT

The production of computer-generated pictures of rendered 3-D objects with hidden surfaces removed is a time-consuming process on conventional computer graphics display systems. Thus, because of the computational complexity involved, it is very difficult to rotate or otherwise transform 3-D shaded images in real-time. Nevertheless, comprehension of 3-D objects from a single 2-D image is difficult. There have been many attempts to enhance the 2-D view for better 3-D comprehension, through such techniques as hidden surface removal and modeling of lighting and through depth cues such as rotations, depth intensity modulation, and stereo.

I have undertaken a project to explore the use of an actual light source as a light source orientation control to produce real-time changes in shading, or more appropriately, in "lighting". The user employs a hand-held light source to "light up" the scene from arbitrary orientations within a video camera's field of view. This use of an actual light source to control the light source orientation is a natural interface and does not have to be learned. The project shows that such dynamic changes in the shading (or "lighting") of an object provide useful depth cues for the viewer and enhance the realism of the display.

THREE-DIMENSIONAL DEPTH PERCEPTION ENHANCEMENT BY DYNAMIC LIGHTING

David H. Holmes

TABLE OF CONTENTS

1 INTRODUCTION 1
1.1 Background 3
1.1.1 Other Methods Of 3-D Depth Perception 3
1.1.2 Introduction To The Problem 4
1.1.3 Related Work 6
1.2 Objectives 7
1.2.1 Statement Of Thesis 7
1.2.2 Scope Of The Problem 8
1.3 Applications 9

2 MATHEMATICAL AND COMPUTER MODELS 11
2.1 The Reflectance Model 11
2.2 Surface Normal Encoding 13

3 THE OPERATIONAL ENVIRONMENT 15
3.1 Hardware 15
3.1.1 The Host Computer System 16
3.1.2 The Graphics System 16
3.1.2.1 The Graphics Display Device 18
3.1.2.2 The Bipolar Microprocessor 20
3.1.2.3 The Image Digitizing Subsystem 21
3.2 Software 21
3.2.1 Preparation Of The Image 22
3.2.1.1 Molecular Models 22
3.2.1.1.1 Sphere Prototype Generation 23
3.2.1.1.2 Z-buffer Algorithm For Molecules 24
3.2.1.2 General Polygonal Objects 24
3.2.1.3 Video Look-up Table Generation 25
3.2.2 Display And Dynamic Lighting 25
3.2.2.1 Light Source Detection 26
3.2.2.2 Video Look-up Table Manipulation 27

4 PRINCIPLES OF OPERATION 29
4.1 System Resource Requirements 29
4.2 Image Preparation 29
4.3 Image Display 30
4.3.1 Calibration Mode 30
4.3.2 Display Mode 30

5 RESULTS AND CONCLUSIONS 32
5.1 Evaluation Of The System 32
5.2 The Future 34
5.2.1 Improvements 34
5.2.2 Suggestions For Future Research 36
5.2.3 My Ideal System 37
5.3 Summary And Conclusions 38

Appendices

A SYSTEM USER'S MANUAL 40
A.1 Introduction - Purpose Of The System 40
A.2 System Initialization 40
A.3 System Execution 41
A.3.1 Calibration Mode 41
A.3.2 Display Mode 42
A.4 Demonstration Software 43
A.5 System Termination And Exit 43

B SYSTEM HARDWARE CONFIGURATION (WIRING DIAGRAM) . . 44

C PROGRAMMER'S MANUAL 45
C.1 Important Data Structures 45
C.2 Software Overview 45
C.3 Computer Program Listings 47

D BIBLIOGRAPHY 48

1 INTRODUCTION

The production of computer-generated pictures of shaded 3-D objects with hidden surfaces removed is a time-consuming process on conventional computer graphics display systems. The process requires a sequence of process steps similar to the following:

1. Model building. In this step, the viewer prepares mathematical models of the desired elements. The method employed for this work is the construction of object models from convex polygons and spheres in 3-D.

2. Rendering.

    1. All relevant geometric transformations of the object must be made. Such transformations typically include translations, rotations, scaling, clipping, and (optionally) perspective of the objects, which collectively transform the object models from 3-D object space into 3-D image space.

    2. Hidden surfaces must be determined and removed. In other words, one must determine what parts of the objects are visible and display only those parts.

    3. All visible surfaces must be shaded.

        1. The normal of the surface at each particular pixel of an object must be calculated. This normal may be represented as a triple of coordinate values for the unit normal (i.e., Cartesian coordinates) or as a pair of angles: an azimuth and an elevation (i.e., spherical coordinates).

        2. The color and intensity of each pixel must be calculated. That is, one must determine the appropriate intensity of each pixel associated with the visible objects, given the viewing angle, the normal of the surface at the particular pixel in question, and the direction of the light source. (For this application, as for many, all light rays from the single light source hit all objects

at the same angle, just as if the light source were an infinite distance away; thus, its orientation is relevant, but its position is not.) In addition, the viewer is assumed to be facing the screen head-on. The shading, then, becomes a function of two factors: the light source(s) that illuminate the scene and the surface properties of each object.

Figure 1-1 graphically depicts these processing steps and indicates the most common form of re-display: to repeat all steps in the sequence except model building.

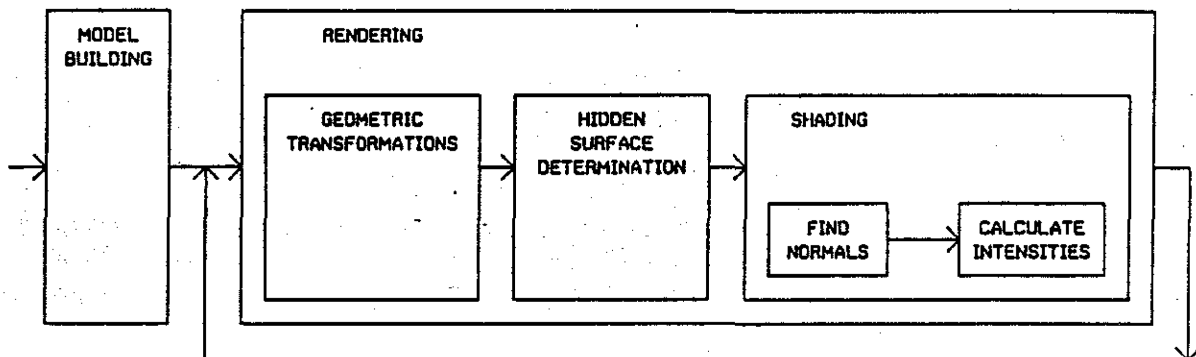


Figure 1-1.

Due to computational constraints, it is extremely difficult to perform the functions described above in real-time on a conventional graphics display system. Nevertheless, comprehension of 3-D objects from a single 2-D image is difficult. There have been many attempts to enhance the 2-D view for better 3-D comprehension, through such techniques as hidden surface removal and modeling of lighting and through depth cues such as rotations, depth intensity modulation, and stereo.

Some alternatives to these techniques exist which also provide the user with useful depth cues. This paper reports on a project which explores one of these alternatives.

I have undertaken a "dynamic lighting" project to explore the use of an actual light source as a light source orientation control to produce real-time changes in shading, or more appropriately, in "lighting". The user employs a hand-held light source to "light up" the scene from arbitrary orientations within a video camera's field of

view. This use of an actual light source to control the light source orientation is a natural interface and does not have to be learned. The project shows that such dynamic changes in the shading (or "lighting") of an object provide useful depth cues for the viewer and enhance the realism of the display, so that the viewer can understand better the structure of the 3-D objects being displayed.

1.1 Background

Before evalutating the success of the dynamic lighting technique, it is useful to explore related work as background material. In the following sections, both 3-D depth perception techniques and a dynamic lighting technique will be discussed.

1.1.1 Other Methods Of 3-D Depth Perception

James Lipscomb, in his PhD Dissertation "3-D Cues for a Molecular Computer Graphics System", evaluated a number of techniques for enhancing the depth perception of 3-D objects presented on a 2-D display, namely rotation, intensity modulation, and stereo. These techniques represent the more common methods for attempting to enhance depth perception. One of his conclusions is that smooth rotation appears to be the single best depth cue, as supported by the following statements that he cites:

> "It is apparent that the kinetic depth effect will readily yield a perception of a rigid spatial arrangement of unconnected objects." [8]

> "The human visual system has special processors for motion that help the user understand what he sees." [6]

These same "special processors for motion" may make dynamic lighting (with its "moving" illumination) a viable depth cue.

Most of the discussion of Lipscomb's work pertains to refresh line-drawing displays. The greater realism of the shaded hidden surface raster display enhances the overall comprehension of 3-D objects, and spatial arrangements of objects may be discerned through the added information provided by dynamic lighting. Moreover, in a conventional raster-scan display system, it is too computationally expensive to rotate a scene of arbitrary complexity in high resolution in order to achieve the kinetic depth effect as a 3-D depth cue.

1.1.2 Introduction To The Problem

Recall the sequence of processing steps to render a shaded geometric model with hidden surfaces removed:

1. All relevant geometric transformations of the object models must be made to transform them from 3-D object space to 2-D image space.

2. Hidden surfaces must be determined and removed.

3. All visible surfaces must be shaded.

    1. The normal of the surface at each particular pixel of an object must be calculated. (This normal is represented as a pair of angles: an azimuth and an elevation.)

    2. The color and intensity of each pixel must be calculated. That is, one must determine the appropriate intensity of each pixel associated with the visible objects, given the viewing angle (assumed to be constant), the normal of the surface at the particular pixel in question, and the direction of the light source.

In raster graphics, the frame buffer (a separate piece of memory hardware that can store and retrieve picture information one pixel at a time) must be loaded with all the individual pixel values that make up the displayed scene. The frame buffer can be quite large (e.g., the Ikonas system I used provided for a 512 by 512 (= 250 K pixels) and a 1024 by 1024 (= 1 Meg. pixels) resolution mode). Therefore, it may take a great deal of time to compute the intensity of each pixel. Thus, major changes to the frame buffer contents will not approach real-time on a conventional graphics display system.

In most raster-scan graphics display systems, the values stored in the frame buffer are addresses into the video look-up table (VLT) (also known as the color look-up table, compensation table, color map, or color table). The original intent of this mechanism was to provide a means of offsetting the non-linearity of the display device.

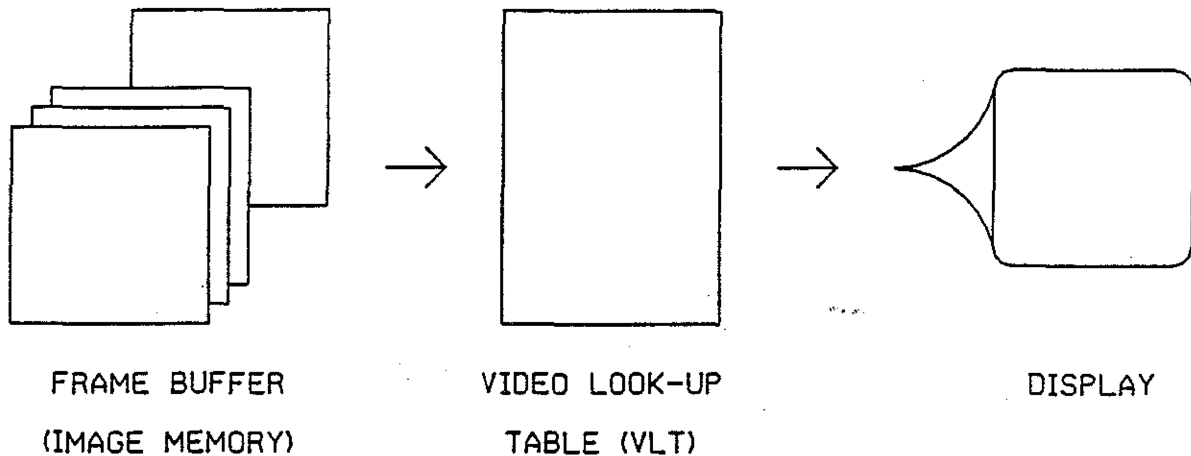


Figure 1-2.

However, the video look-up table may be used to accomplish any desired mapping. We can take advantage of this fact by placing the calculated pairs of angle values (encoded surface normal vectors) into the frame buffer instead of the intensity values. Then only the intensity associated with each normal must be determined to obtain a different display of the model under a different orientation of the point light source. This step may be accomplished very quickly and conveniently by mapping directly from an angle pair to an intensity value by virtue of the video look-up table and its addressing scheme.

The assumptions required for this mechanism to be effective are as follows:

- The frame buffer contents remain static (i.e., the surface normals don't change in time).

- The position of the observer is fixed.

- The single point light source is assumed to be located at an infinite distance (i.e., all light source rays are parallel).

Based on these assumptions, the shading becomes a function of the light source that illuminates the scene and the surface orientation properties of each object.

For any constant orientation of the user's eye, only the intensity calculation step must be repeated for a change in the orientation of the light source. Therefore, only the video look-up table values must be re-calculated to bring

about changes in shading as a function of the light source orientation. The much larger frame buffer need not be changed at all. This organization thus requires the calculation of only 3072 values (one video look-up table per primary color and 1024 entries per table) in the Ikonas raster display system for each new image as compared to the far greater number of values (262144) in the frame buffer. (Note: This work was accomplished on an Ikonas RDS2000; however, Ikonas has been bought by Adage and the successor to this machine is the Adage RDS3000.) Figure 1-3 graphically depicts these processing steps.

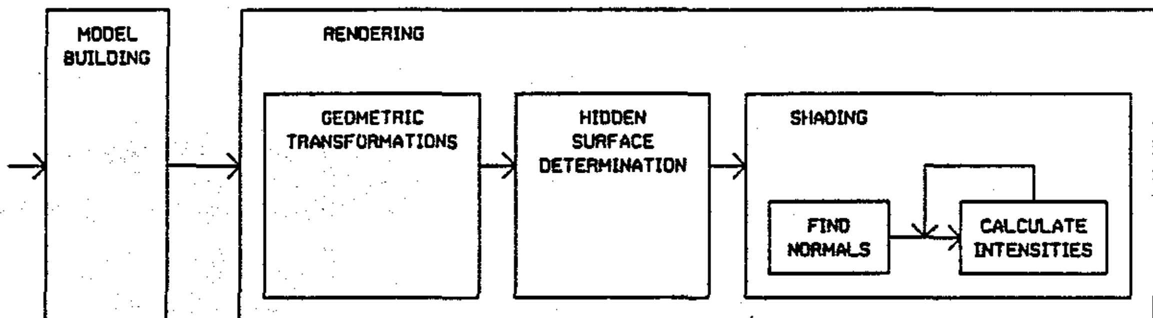


Figure 1-3.

This scheme has the effect of permitting the fast, dynamic display of a wide variety of lighting conditions for a static scene. It is hoped that the calculation of these video look-up table values can be performed quickly enough to bring about smooth changes in the shading in real-time.

1.1.3 Related Work

Previous work in the area of dynamic lighting has concentrated primarily on evaluating the use of video look-up tables to alter quickly the perceived light source orientation in a displayed scene.

Daniel Bass concentrated on comparing different surface normal quantization schemes and on investigating the display of objects under different lighting and reflectivity conditions. "We want to quantize the surface normals for two reasons:

1. to save time.

2. to take advantage of the use of the VLT to perform the mapping of normals to intensities; the VLT has only 256 discrete mappings possible." [1]

Because of the discretization caused by the limited number of entries in the video look-up table, the notion of equivalence classes of normals is a requirement. The challenge is to assign equivalence classes in an efficient fashion.

I spent a good deal of time trying to determine an efficient solution for this problem. This topic is dicussed in greater detail in section 2.2.

Bass lists the following drawbacks to the video look-up table scheme that he implemented:

1. "Quantizing surface normals produces artifacts in the final image unlike those resulting from the usual approach of quantizing the surface brightness.

2. Lighting effects must be independent of object position; therefore shadowing can't be done.

3. The process is still not quite fast enough to allow real-time rotating lights." [1]

Limitations in the display device posed some problems for Bass in his attempts to enhance the realism of the scene. Specifically, Bass had only one 256-color video look-up table, whereas the Ikonas system I used provided three separate video look-up tables, each consisting of 1024 entries. The 256-color limitation in Bass's system resulted in "banded" displays of curved surfaces. Also, my implementation of dynamic lighting can operate much faster (16 Hz.) than Bass's implementation, which did not approach real-time.

1.2 Objectives

The objectives of this work are presented as a statement of thesis and a specification of the scope of the problem. These objectives are defined below.

1.2.1 Statement Of Thesis

I have designed and implemented a system in which the direction of a light source can be modified dynamically to provide real-time changes in intensity based on the orientation of a hand-held light source. The thesis of this work is that such dynamic changes in the intensity and position of the shading (a more appropriate term might be

"lighting") will provide valuable depth cues, thus enhancing the viewer's comprehension of object structure. In addition, this method appears to provide a higher degree of realism for the viewer and the real-time nature of this technique is a fundamental part of the viewer's realistic perception of the modeled scene.

1.2.2 Scope Of The Problem

This work emphasizes the use of a hand-held light as an input device and fast response to provide realistic feedback of the user's orientation of the light source. Furthermore, this work addresses the first and third of the drawbacks listed by Bass: the quantization error and performance respectively. In addition, I have evaluated this technique on the basis of its potential use as a depth cue.

I improved on the artifacts resulting from quantization of normals by using larger video look-up tables and by trying to create a judicious set of normal equivalence classes. I addressed the performance drawback by using the bipolar microprocessor to detect the orientation of the light source in a small digitized region of the frame buffer and by using a VAX minicomputer and approximately 1 Meg. of main memory to allow the user to "poke" one of 33 by 33 = 1089 different pre-calculated video look-up tables dynamically (i.e., in real-time). This approach provides an apparently smooth transition between successive (adjacent) light source orientations.

In keeping with the goals of providing realism and fast response in the use of this system, a pen-light is used as the user's input device. A video camera digitizes the area in front of the display screen, placing the digitized image in a dedicated region of the frame buffer, and the Ikonas bipolar microprocessor quickly detects the orientation of the light source by searching that region.

Such added features as a calibration mode and a dynamic "out-of-bounds" indicator enhance the ease of use of the software. The calibration mode enables the user to calibrate the video camera lens settings, the distance of the point light source from the camera, the size of the point light source, and threshold values for light source detection. If the light source is out of the video camera's field of view, the out-of-bounds indicator, a border around the displayed image, is turned from black to green.

Originally, the driving problem for this work was a representation of a molecule in which each atom is represented as a shaded sphere. The results of this work eventually may be incorporated into a future GRIP system, a molecular graphics display system at the University of North

Carolina at Chapel Hill.

This project models objects as spheres or polygon-defined objects. In the initial implementation, I limited my attention to objects which consisted only of spheres that could intersect. Since spheres are easily represented as an x-, y-, and z-coordinate in space and a length (radius), the initial problem was simplified by restricting my attention to spheres. Furthermore, some additional simplifications were found to increase the speed of generating scenes consisting only of spherical objects. Since most objects can not be represented by spheres alone and are more appropriately represented by polygons, I extended the scope of the system to include provisions for polygons in later stages.

Also in the initial implementation, I developed the software on a PDP 11/45; when a VAX became available, however, I converted the software to run there and performed the remaining software and hardware development on the VAX.

Initially, video look-up table calculations were performed "on the fly", but the performance considerations dictated that the VLT's be pre-calculated. This pre-calculation of VLT's resulted in an order of magnitude improvement in performance. However, a better resolution of light source orientation is possible if the VLT's are calculated after rather than before a specific light source orientation has been identified.

In the beginning, I used a black and white 30 Hz. monitor, but I extended the scheme to a color 30 Hz. monitor, in which eight colors (and up to 256 shades of each color) were provided. The reason for the eight color limitation had to do with the fact that originally the system had only 256-entry video look-up tables. I extended the scope of the system in later stages to support 64 colors by utilizing the Ikonas crossbar and 1024-entry video look-up tables.

Initially, I set the system up to accept keyboard commands for light source orientation changes. This mode is still supported as an alternative to the pen-light detection interface.

1.3 Applications

The dynamic lighting technique may be applied to other graphics software. In this manner, the GRIP-75 molecular graphics system at UNC may be extended to include a dynamic lighting feature. Furthermore, various realistic display features that are commonly included as part of the hidden surface rendering process may be included in the image

preparation phase of the dynamic lighting process. These extensions and various other applications are discussed in section 5.2.1.

2 MATHEMATICAL AND COMPUTER MODELS

The mathematical models and computer models required for the dynamic lighting technique are presented in this chapter. The basic concepts associated with the dynamic lighting technique depend on an understanding of the reflectance ("lighting") model. After the discussion of the reflectance model, the role of surface normals and the use of video look-up tables is discussed.

2.1 The Reflectance Model

A reflectance (or "lighting") model is used to determine appropriate intensities for each pixel once visible surfaces have been determined by a hidden surface algorithm. In this section, a description of the commonly-used reflectance model developed by Bui-Tuong Phong is presented.

> "The shading model has two ingredients, properties of the surface and properties of the illumination falling on it. The principal surface property is its "reflectance", which determines how much of the incident light is reflected...
>
> An object's illumination is as important as its surface properties in computing its intensity. The scene may have some illumination that is uniform from all directions, called "diffuse illumination". In addition, there may be "point sources" of light in the scene; they differ from diffuse lighting in that specular reflections, or "highlights", appear on surfaces.
>
> ...The shading model can be decomposed into three parts, a contribution from diffuse illumination, contributions for one or more specific light sources, and a transparency effect. Each of these effects contributes shading terms E, which are summed to find the total light energy coming from a point on an object. This is the energy a display should generate to present a realistic image of the object." [5]

For the purposes of this work, the contribution from transparency is ignored.

Diffuse illumination may be specified by the equation:

$$Epd = RpId$$

"where Epd is the energy coming from the point P due to diffuse illumination, Id is the diffuse illumination falling on the entire scene, and Rp is the reflectance coefficient at P, which ranges from 0 to 1. Thus the reflectance coefficient relates the energy leaving point P to that arriving. To model colored surfaces, the reflectance coefficient and illumination have separate components in a color coordinate system...

Shading contributions from specific light sources will cause the shade of a surface to vary as its orientation with respect to the light source changes and will also include specular reflection effects. The first of these effects is due to Lambert's law, which states that the energy falling on a surface varies as the cosine of the angle of incidence of the light...

...Treating specular reflection requires us to calculate the relationship between the observer, the light source, and the surface..." [5] The equation for the light source illumination (for each light source) is:

$$Eps = [Rpcos(i) + W(i)(cos(s))^{**}n]Ips$$

where: Rp = reflectance coefficient at P
i = angle of incidence (angle between incident ray and normal)
W(i) = specular reflection coefficient, a function of i
s = angle between reflected ray and viewer's line of sight
n = value in range 1-10; controls how "shiny" the surface appears
Ips = energy arriving from the light source

This formula contains a diffuse reflection component and a specular reflection component.

"The angles required in the shading model can be determined entirely from the "normal" vector for a surface... ..Angular calculations are simplified if we assume that the viewpoint and all light sources are infinitely far away from the object in the world coordinate system. Thus a vector to one of these points has a constant direction throughout the scene. For convenience, these vectors are normalized to have unit length." [5]

In the implementation of this dynamic lighting project, a shading parameters file is provided at image preparation time to enable the user to make changes in the relative contributions of specular and diffuse components and in the exponent (n) which controls how "shiny" the surface appears.

2.2 Surface Normal Encoding

Because of the discretization caused by the limited number of entries in the video look-up table, the notion of equivalence classes of normals is a requirement. The challenge lies in the efficient assignment of equivalent classes.

"... we must determine the set of vector equivalence classes into which all visible vectors may be grouped to quantize the infinite number of visible surface normals into a finite set of vector equivalence classes. Each equivalence class is represented by a single vector which is an approximation of those vectors in that equivalence class." [1]

The schemes for deriving equivalence classes of normals that Bass [1] tried included the following:

1. Choice of X and Y as independent variables; 6 bits for X and 6 bits for Y; therefore equal area patches for a sphere.

2. Choice of Z and theta (= arctan(Y/X)) as independent variables; 6 bits for each; therefore concentric rings and pie slices; problems: "pie-shaped" substructure, flat top.

3. Choice of phi and theta (spherical coordinates); 6 bits each; therefore flattening of sphere is removed, and surface of sphere is more evenly partitioned into equivalence classes.

4. Choice of phi and theta, but number of theta intervals per ring increases with phi. [1]

I explored both options 3 and 4 and found that visually, option 4 is a much better distribution of equivalence classes than option 3. After trying many schemes to optimize option 4, the method which emerges as the best is to calculate the actual normals that represent the equivalence classes based on some appropriate spacing criteria.

The most appropriate spacing criteria appear to be (a) concentric rings representing the visible side of a sphere, with "equal cosines" spacing between rings (i.e., the spacing is a function of the cosine of the angle between the normal at the center of the sphere and any normal along the ring in question), and (b) the calculation of the number of normals to be placed along the concentric ring as a function of the circumference of each particluar ring. The first criterion has the effect of generating concentric rings at the periphery of the sphere that are more closely spaced than at the center.

The question of generating a sphere prototype becomes one of establishing its equivalence classes. Once the spacing criteria for the actual normals that represent the equivalence classes have been determined, the equivalence classes themselves may be established by:

1. calculating all 256 normals that represent the equivalence classes,

2. determining the normal at the center of each pixel in the sphere,

3. determining the closest "equivalence class normal" to the given normal, and

4. using that equivalence class normal to represent the pixel in question.

To minimize computations, we may determine how close two normals are by determining their dot product instead of using the distance formula. In the case of general polyhedra, the same mechanism is used; however, instead of determining equivalence classes once for a prototype, the calculations must be performed "on the fly".

3 THE OPERATIONAL ENVIRONMENT

The system consists of hardware and software components. The hardware environment consists of a host computer system (VAX) and an Ikonas graphics display system. Within the graphics display system, we are primarily concerned with the graphics display device itself, a video digitizer, and a bipolar microprocessor.

The software component of the system consists of two sets of software:

- image preparation software, which models objects, performs hidden surface rendering, calculates normals, and generates video look-up tables in preparation for dynamic lighting, and

- dynamic lighting display software which dynamically changes the lighting of the scene by detecting light source orientation and manipulating the video look-up tables.

Details of the operational environment are discussed in the following sections.

3.1 Hardware

The hardware required for this work consisted of the following items:

* IKONAS advanced 24-bit color graphics system; includes:
  - frame buffer
  - video look-up table
  - cross-bar
  - raster-scan display logic

* 30 Hz. Color monitor (initial work was on a 30 Hz. b&w monitor)

* VAX 11/780 (initial work was on a PDP 11/45)

* User interface (to input light source orientation)
  - Video digitization subsystem
    - Digitizer
    - Bipolar microprocessor with scratchpad memory
    - Video camera
    - Pen-light
  - Alphanumeric terminal (keyboard commands)

The configuration of hardware is depicted in Figure 3-1.

I performed my early software development on a PDP 11/45 and converted the software to a VAX 11/780 when that system became available. In later stages, I implemented a more convenient, more "natural" light source direction indicator than keyboard commands: a pen-light, whose orientation is detected from its digitized image from a video camera. Furthermore, I extended the initial work on a 30 Hz. black and white monitor to a 30 Hz. color monitor.

3.1.1 The Host Computer System

A Vax 11/780 with 2 Meg. of main memory functioned as the host computer for this system. Whereas access to 33 x 33 x 1024 x 3 = 1.11 Meg. for video look-up tables was required, the large amount of main memory was extremely important to the real-time aspect of the dynamic lighting process (to inhibit page faults in the virtual memory system).

3.1.2 The Graphics System

The Ikonas graphics system at UNC-CH consists of an advanced 24-bit color graphics system, with a 24-bit frame buffer, a 60 Hz. high resolution monitor, a 1024-entry video look-up table for each primary color (red, green, and blue), a "cross-bar" (described below), a bipolar (i.e., bit-sliced) microprocessor with its own "scratchpad" memory, and a digitizer, which could digitize signals from a video camera at the rate of 30 Hz. Although the Ikonas system drives a 60 Hz. monitor, it must run in 30 Hz. mode, driving a 30 Hz. monitor, to allow continuous 30 Hz. video digitization.

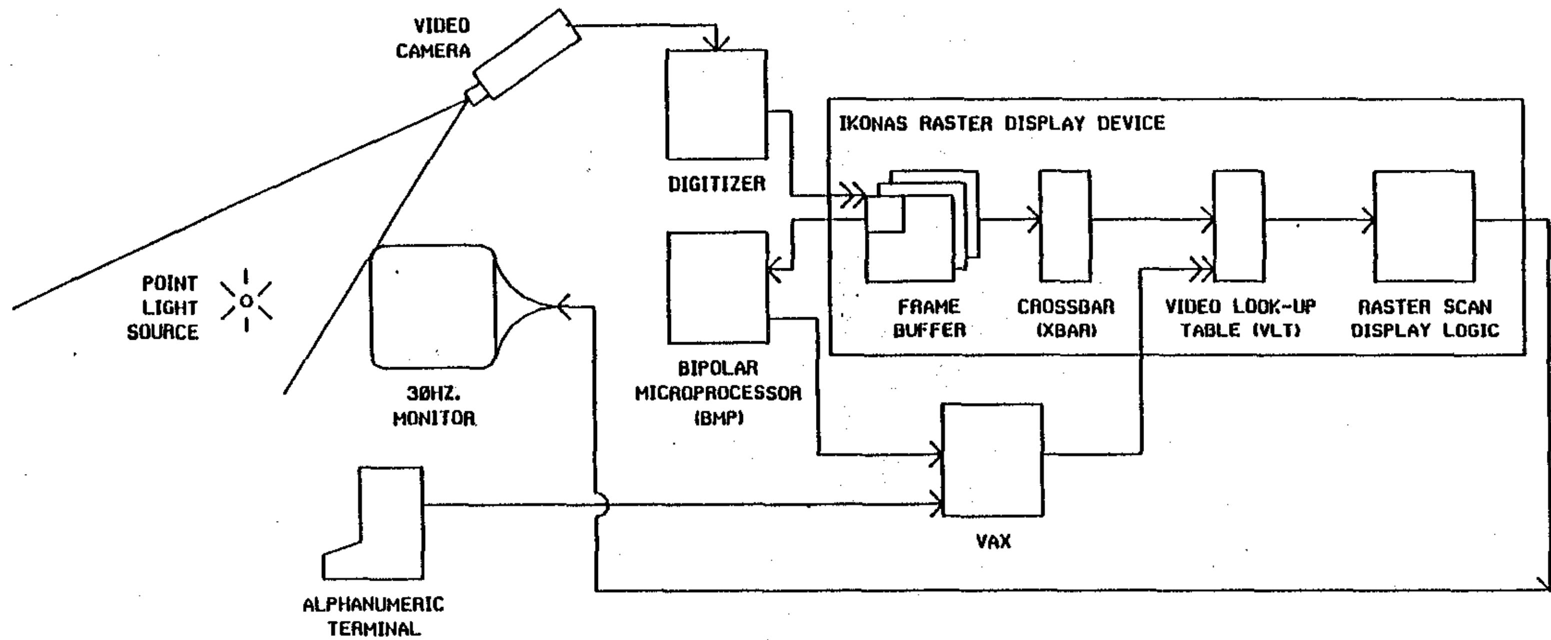


Figure 3-1.

3.1.2.1 The Graphics Display Device

The principal components of the graphics display device are the 24-bit frame buffer, the video look-up table, the cross-bar, and raster-scan display logic. The monitor and frame buffer memory were configurable to two distinct resolutions:

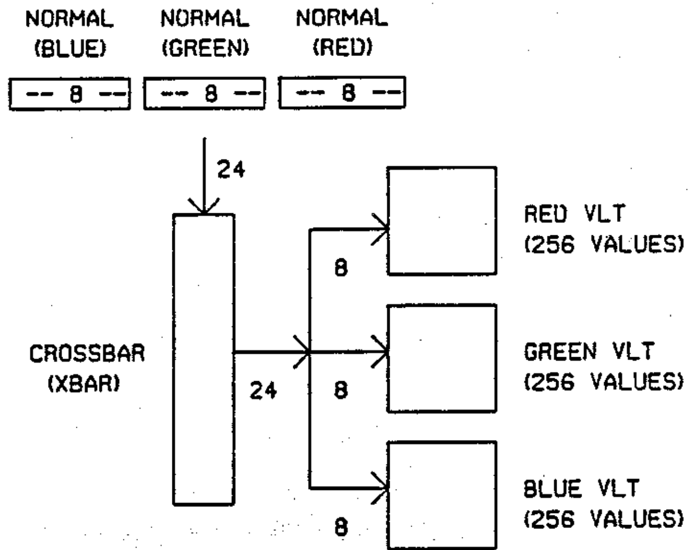
- low resolution: 512 by 512 display with 24 bits per pixel
- high resolution: 1024 by 1024 display with 6 bits per pixel

Due to the requirements for encoding normal information, color information, and digitized image information in the frame buffer, I performed all work pertaining to the dynamic lighting application in low resolution mode; hence each pixel may be considered to be 24 bits "deep".

For the initial implementation of color display, only 256 value video look-up tables were available. By using the normal value to specify the presence of each primary color and the value 0 to specify its absence, eight colors were provided: the three primaries, the 3 secondaries, black, and white. This mechanism is depicted in Figure 3-2.

A 1024-entry video look-up table was established for each primary color (red, green, and blue) when the hardware 1024-entry video look-up tables became available. Note that full addressability of the three video look-up tables requires 30 bits (10 bits for each primary color). Since the pixel is at most only 24 bits deep, a "cross-bar" is provided to allow arbitrary mapping of frame buffer bits into video look-up table addresses.

The cross-bar is used to advantage in this work by using 8 of the 24 bits to represent an encoded normal, using 3 pairs of bits to represent red, green, and blue contributions, and saving 4 bits for use in the light source digitization process. This video look-up table mapping is illustrated in Figure 3-2.



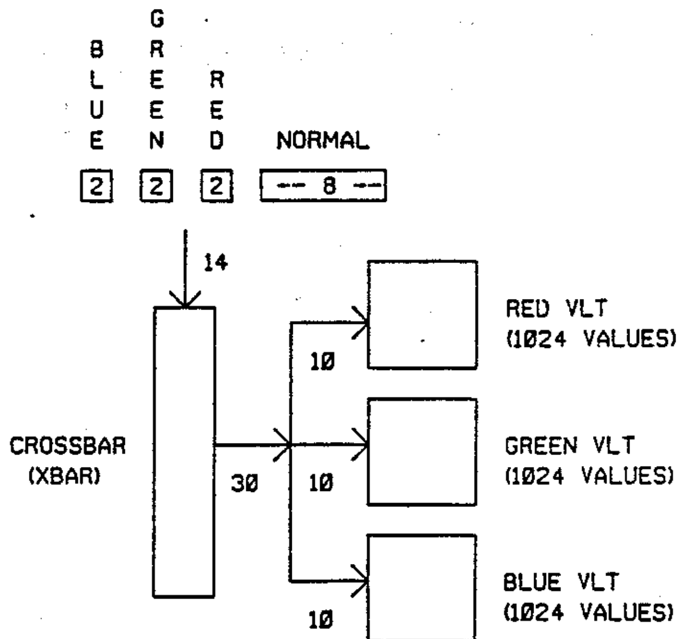


Figure 3-2.

Thus, each video look-up table may be subdivided into 4 sub-tables, one each to represent full intensity, 2/3 intensity, 1/3 intensity, and no intensity contributions of a primary color. One of these four intensities is assigned to each given object for each primary color at object creation time in order to specify the object's color. In this manner, we have increased the number of colors we may represent simultaneously to 64. These values are not the intensities that change as a function of the lighting, but are the encoded color information for each object. Figure 3-3 is an example of the variety of colors available with this method.

Figure 3-3.

One entry in each video look-up table is reserved for the screen border, which is toggled as feedback to the user for success or failure of the light source detection, as when the light source leaves the camera's field of view.

3.1.2.2 The Bipolar Microprocessor

In this application, the programmed bit-sliced bipolar microprocessor is used exclusively to track the hand-held light source by searching a portion of the frame buffer memory into which an image has been continuously digitized for the purpose of detecting the orientation of the light source. The bipolar microprocessor has proven to be critical to the goal of providing a real-time user interface that was easy to use and whose effects were easy to understand, providing timely and useful feedback.

For processing speed, the video camera image is digitized sparsely. Specifically, every eighth pixel in every eighth scan-line (resulting in a 64 by 64 pixel image)

is digitized in the frame buffer and searched by the bipolar microprocessor. Only four bits of intensity per pixel are used because a four-bit pixel depth is sufficient for the simple thresholding of the light source intensity if the room is darkened. Through the use of the cross-bar, this region may be made non-displayable so that it will not interfere with the dynamic lighting display, or displayable as in the case of the calibration mode for the digitization process.

3.1.2.3 The Image Digitizing Subsystem

In addition to the digitizer, the light source detection mechanism requires a video camera (to transmit the signals to be digitized) and a point light source of sufficient size to be detected by the digitizer in a reasonable amount of time, but not so large as to cause detection accuracy problems. A small hand-held fluorescent lamp with electrical tape covering all but a small square window proved to be the best point light source for this application.

3.2 Software

In this section, the organization of the software and many of the software details are described.

All host programs were written in C, since it tends to be the language of choice for Unix environments, and it is well-suited for the task in question. All Ikonas assembly programs (which run on the bipolar microprocessor) have been written in GIA, an assembly language developed by Gary Bishop at UNC.

For each pixel, an encoded normal is stored in the frame buffer instead of the intensity. The approach enables us to vary the intensity as a function of these angles with respect to the orientation of the light source. The mapping of these angles to intensities is accomplished by encoding the explicit intensity values in the video look-up table for each of the encoded normals. These video look-up table values are changed whenever the orientation of the light source changes.

I have written a dynamic lighting routine to calculate the intensity of a given pixel, given the orientation of the light source, for each azimuth/elevation value pair (i.e., encoded normal). A constant value may be added to the calculated intensity at this stage to simulate the effects of ambient light on the surfaces.

Methods for quickly changing the values of the video look-up table as a function of the light source orientation have been explored and have been implemented where feasible. (Such methods as "rolling up/down/left/right" the series of values stored in the video look-up table were found to be unnecessary, since I found that the update rate without such a change was sufficient.)

Initially, I limited the scope of this work to spherical objects in order to determine feasibility of approaches with a simplified scheme. However, I extended this approach to include polygon-defined objects and such options as clipping and perspective.

Also, two input mechanisms are provided to orient the light source: (a) the use of a pen-light (or other point light source) and digitized images from a video camera and (b) simple keyboard commands which enable the user to move the light source up, down, left, or right by pressing the appropriate arrow keys.

3.2.1 Preparation Of The Image

Before the dynamic lighting process may be invoked, a scene must be prepared so that as much as possible of the rendering process has been performed. This requirement is in keeping with the idea that image data will be placed in the frame buffer only once (with the exception of the data from the digitization process) and changes will be limited to the video look-up tables, in order to achieve real-time updates in the display.

3.2.1.1 Molecular Models

Initially, only molecular models were studied in order to simplify the scope of the problem to dealing with the well-behaved nature of normals on the surfaces of spheres. In addition, I treated the GRIP-75 molecular graphics system as a potential driving problem for this work.

In order to display spheres realistically and yet simplify the process of displaying spheres, I divided the hidden surface rendering process into two stages: a sphere prototype generation stage and a simple "spheres only" z-buffer algorithm. These two stages are discussed in the following sections.

3.2.1.1.1 Sphere Prototype Generation

By limiting the scope of the problem to molecular models (initially), I was able to limit my attention to spheres. Instead of converting such spheres to polyhedral approximations, therefore, the problem could be solved once for a reasonably large sphere, with the result stored in a raster buffer, allowing the software to index into the sphere based on the scale of each new sphere to be displayed.

Many techniques for obtaining an optimal representation of the sphere were explored. The limitations for the spherical representation were such that only 256 unique normals were available to represent the visible portion of the sphere; hence the problem became one of determining the optimal placement of normal "equivalence classes".

The best scheme determined consisted of optimizing the placement of equivalence classes on concentric circles of the sphere's visible surface. I specified the number of equivalence classes on a concentric circle to be a function of the cosine of the angle between the normal at the center of the sphere and the normal at a particular point on the sphere's surface.

The exception to this rule occurs for equivalence classes along the perimeter of the sphere, relative to the line of sight. For these classes, the number of actual pixels represented by the equivalence class might be extremely small relative to the other classes, so by way of compensation, the farthest concentric circle from the center is placed far enough away from the perimeter of the sphere to ensure that a "reasonable number" of pixels is thereby represented. This technique is employed in the sphere prototype generation module.

I selected the sphere prototype size to cover 10% of the area of the screen, since in general, most molecular models (initially, the driving problem of this work) would not be viewed such that atoms would appear larger than that size. Figure 3-4 shows a sample 26 by 26 sphere prototype with three rings and a center.

```
 0  0  0  0  0  0  0  0  0 17 17 17 17 16 16 16 16  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0 18 17 17 17 17 17 16 16 16 16 16 15  0  0  0  0  0  0  0
 0  0  0  0  0 18 18 18  7 17 17 17  6  6  6  6  6 15 15 15 15  0  0  0  0  0
 0  0  0  0 18 18 18  7  7  7  7  6  6  6  6  6  6 15 15 15 15 15  0  0  0  0
 0  0  0 19 19 18  7  7  7  7  7  2  6  6  6  6  6  6 15 15 15  5 14  0  0  0
 0  0 19 19 19  7  7  7  7  7  7  2  2  2  6  6  6  5  5  5  5  5 14 14  0  0
 0  0 19 19 19 19  7  7  7  7  2  2  2  2  2  2  2  5  5  5  5  5 14 14  0  0
 0 19 19 19  8  8  7  7  7  7  2  2  2  2  2  2  2  5  5  5  5  5 14 14 14  0
 0 20  8  8  8  8  8  8  7  2  2  2  2  2  2  2  2  5  5  5  5  5 14 14 13  0
20 20  8  8  8  8  8  8  8  2  2  2  2  2  2  2  2  5  5  5  5  5 13 13 13 13
20 20  8  8  8  8  8  8  1  1  1  1  1  1  1  1  1  1  5  5  4  4  4 13 13 13
20 20 20  8  8  8  8  8  1  1  1  1  1  1  1  1  1  1  4  4  4  4  4  4 13 13
20 20 20  8  8  8  8  8  1  1  1  1  1  1  1  1  1  1  4  4  4  4  4  4  4 13
21 21 21  9  9  9  9  9  1  1  1  1  1  1  1  1  1  1  4  4  4  4  4  4  4 28
21 21 21  9  9  9  9  9  1  1  1  1  1  1  1  1  1  1  4  4  4  4  4  4 28 28
21 21  9  9  9  9  9  9  1  1  1  1  1  1  1  1  1  1 12 12  4  4  4 28 28 28
21 21  9  9  9  9  9  9  9  3  3  3  3  3  3  3  3 12 12 12 12 12 28 28 28 28
 0 21  9  9  9  9  9  9 10  3  3  3  3  3  3  3  3 12 12 12 12 12 27 27 28  0
 0 22 22 22  9  9 10 10 10 10  3  3  3  3  3  3  3 12 12 12 12 12 27 27 27  0
 0  0 22 22 22 22 10 10 10 10  3  3  3  3  3  3  3 12 12 12 12 12 27 27  0  0
 0  0 22 22 22 10 10 10 10 10 10  3  3  3 11 11 11 12 12 12 12 12 27 27  0  0
 0  0  0 22 22 23 10 10 10 10 10  3 11 11 11 11 11 11 26 26 26 12 27  0  0  0
 0  0  0  0 23 23 23 10 10 10 10 11 11 11 11 11 11 26 26 26 26 26  0  0  0  0
 0  0  0  0  0 23 23 23 10 24 24 24 11 11 11 11 11 26 26 26 26  0  0  0  0  0
 0  0  0  0  0  0  0 23 24 24 24 24 24 25 25 25 25 25 26  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0 24 24 24 24 25 25 25 25  0  0  0  0  0  0  0  0  0
```

Figure 3-4.

3.2.1.1.2 Z-buffer Algorithm For Molecules

I modified a simple z-buffer algorithm to index directly into the sphere prototype and obtain the specific normal equivalence class which represents each particular pixel and to store the resulting normal encoding in the frame buffer. The z-buffer is maintained as a separate internal data structure and is not part of the frame buffer. This software was a modification of software developed by Mike Pique at UNC-CH.

3.2.1.2 General Polygonal Objects

Eventually, the techniques for dynamic lighting were extended to handle any scene defined by polygons. I implemented this capability by modifying Eric Grant's implementation of the Binary Space Partitioning Tree (BSP Tree) algorithm, described in [3]. This visible surface algorithm enables the user to modify orientations conveniently and streamline the visible surface determination process.

The advantage of this particular algorithm is that it provides quick modification of the scene as part of the set-up sequence for the dynamic lighting procedure. For the purposes of this work, however, any hidden surface algorithm that deals with general geometric models would suffice.

I modified this software to permit encoding of surface normals and placement of the corresponding normal equivalence class directly into the frame buffer. In addition, I modified the software to support the 64-color encoding scheme and to accomodate variable parameters for lighting conditions.

3.2.1.3 Video Look-up Table Generation

In order to achieve real-time modification of the video look-up tables, values for the tables are calculated ahead of time and are loaded into the hardware video look-up table whenever needed. In order to accomplish a realistic transition between two adjacent lighting equivalence classes, at least 32 by 32 distinct video look-up tables were found to be necessary.

I established a selection of 33 by 33 video look-up tables in order to provide an extra measure of flexibility at the extremes. Specifically, I established an extreme lighting angle at the borders of light source detection to underscore the fact that the borders have been crossed. To further enhance the realism of the display, the video look-up table values have built-in compensation for the non-linearity of the display device.

3.2.2 Display And Dynamic Lighting

Once the hidden surface rendering process has completed, an "image" remains in the Ikonas frame buffer. Until the dynamic lighting process has begun, however, this image appears as an unusual rendering of the original scene (as shown in Figure 3-5 below), since the information stored in the frame buffer is not intensity information, but normal information.

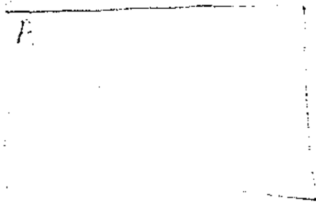


Figure 3-5.

3.2.2.1 Light Source Detection

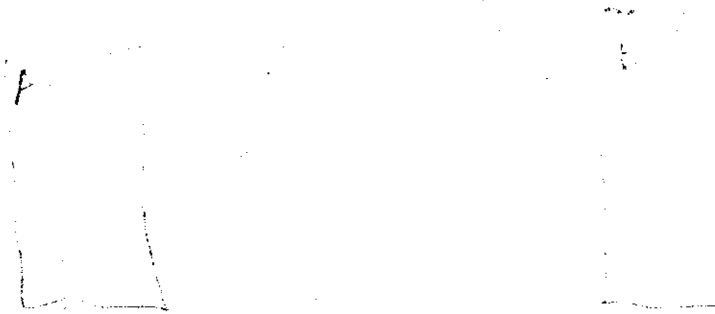
The algorithm employed in the search for the light source is as follows. The video camera image is digitized sparsely for processing speed. Specifically, every eighth pixel in every eighth scan-line (resulting in a 64 by 64 pixel image) is digitized in the frame buffer and searched by the bipolar microprocessor. Only four bits of intensity per pixel are used because a four-bit pixel depth is sufficient for the simple thresholding of the light source intensity if the room is darkened. Through the use of the cross-bar, this region may be made non-displayable so that it will not interfere with the dynamic lighting display, or displayable as in the case of the calibration mode for the digitization process. The light source detection code running in the bipolar microprocessor was provided by Gary Bishop, a former graduate student at UNC-CH.

A border is established around the displayed image which contains encoded normals that refer to a unique entry in the video look-up tables. In this manner, a special feedback mechanism is provided to indicate to the user that the light source is not being detected at a particular point in time. Specifically, only that unique video look-up table entry will be modified when the digitizer reports that the light source could not be detected. The most common reason for this "not detected" indication occurs when the light source is outside the field of view of the video camera.

3.2.2.2 Video Look-up Table Manipulation

Since the video look-up tables have already been calculated and are stored in a large array in main memory on the host computer system, the process is essentially one of address calculation and video look-up table loading once the address has been computed in the bipolar microprocessor. The bipolar microprocessor accomplishes this address calculation by scanning the digitized image of the light source in the frame buffer. The host computer maintains the large video look-up table array and loads a specific video look-up table associated with the location in the digitized image where the bipolar microprocessor found the light source.

Figure 3-6 illustrates the dynamic lighting system in operation.



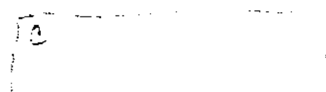


Figure 3-6.

4 PRINCIPLES OF OPERATION

In order to use the dynamic lighting system (DYNALITE), it is important for the user to grasp some preliminary points. The system is complex in its interrelationships and all components of the system depend heavily on the workings of other components. Thus, the most critical item is to verify that all aspects of the system are functional and able to communicate with the rest of the system.

This verification may be accomplished by means of the checklist that appears in section A.2 of the Appendices.

4.1 System Resource Requirements

The system requires at least 1.5 Meg. of main memory. In addition, the system requires dedicated access (or high priority access) on the host computer to be able to achieve a display rate and light source detection rate that approaches real-time.

4.2 Image Preparation

The user of the system may run either of two hidden surface programs to set up a frame buffer "image" for dynamic lighting: a "spheres only" z-buffer algorithm which indexes directly into a sphere prototype to obtain the desired normals, or an implementation of the Binary Space Partitioned Tree algorithm, in which the user writes a program based on the graphics package approach that is established by this implementation. (As an example of the latter graphics package approach, consider the "wellbench" program in Appendix C.)

Alternatively, if the user has a disk file of raster data which has already been generated in the proper dynamic lighting format (i.e., with encoded surface normals and color representing each pixel), then a utility may be called that merely transfers the data from the file to the frame buffer.

In addition, utilities are availale to generate a prototype for a sphere represented as encoded surface normals for use in the z-buffer algorithm, and to calculate a 33 by 33 grid of color tables based on normals spaced sixteen screen pixels apart (at 512 by 512 resolution).

4.3 Image Display

A calibration mode and a display mode enable the user to calibrate the light source detection process or to run the dynamic lighting display process, respectively. It is strongly recommended that the calibration mode be invoked before the display mode unless the system has been recently calibrated.

4.3.1 Calibration Mode

Due to the nature of the light source digitization process, it is necessary to invoke a calibration mode in order

1. to prevent the accidental detection as the light source of other illuminated or illuminating items in the room,

2. to guarantee the detection of the light source, and

3. to familiarize the user with the field of view limits of the video camera's lens.

When the calibration mode is invoked, a grid of dots is displayed which indicate the individual light source detection locations, or equivalence classes. The user must make sure that the video camera lens settings, the distance between the light source and the camera, the darkness of the room, and the threshold value in the light source detection process are calibrated so that only the light source is digitized above the threshold intensity and displayed. Furthermore, the user must ensure that the light source is always digitized into at least one grid dot (so that the light source won't "fall through the cracks" of the digitized image), but no more than four. If the light source is digitized into two to four adjacent grid dots, the closest grid dot to the previous value is chosen as the light source orientation.

The light source is being digitized at a coarse resolution to simplify the detection process, to optimize the speed of the detection process, and to minimize the impact on the frame buffer in terms of the number of bits that are required to digitize an image.

4.3.2 Display Mode

When the display mode is invoked, the Ikonas digitizer, bipolar microprocessor, video look-up table, and crossbar are initialized, an "out-of-bounds" frame is set up around

the image, and BMP and Vax software are invoked to perform the dynamic lighting display. Alternatively, the user may invoke Vax software to perform the dynamic lighting display based on keyboard "arrow keys" as input. The primary input technique emphasized in this work, however, is the digitization system, and the remainder of this section is devoted to its features.

In the display mode, the user is interacting with the dynamic lighting process in order to modify the display. It is therefore useful to discuss both the input required from the user and the kinds of output that the user may expect to see.

The user employs the hand-held light source to "light up" the scene from arbitrary orientations within the camera's field of view. The most important aspect of the system is simply what the user sees.

The reaction time of the system to detect the orientation of the light source and update the video look-up tables is approximately 1/16 of a second. Therefore, the feedback to the user is almost instantaneous and provides the user with a very natural mechanism for specifying light source orientations.

A border is established around the displayed image that contains encoded normals that refer to a unique entry in the video look-up tables. In this manner, a special feedback mechanism is provided to indicate to the user that the light source is not being detected at a particular point in time. Specifically, only that unique video look-up table entry will be modified when the digitizer reports that the light source could not be detected. Thus, the system will alter the color of the border (from black to green) to give the user feedback when the light source is "out of bounds", i.e., out of the field of view of the video camera.

5 RESULTS AND CONCLUSIONS

I have designed and implemented a system in which the direction of a light source can be modified dynamically to provide real-time changes in intensity based on the direction of the light source. My thesis is that such dynamic changes in the intensity and position of the shading (a more appropriate term might be "lighting") provide valuable depth cues, thus enhancing the depth perception of the viewer. In addition, this method appears to provide a higher degree of realism for the viewer and the real-time nature of this technique is a fundamental part of the viewer's realistic perception of the modeled scene.

5.1 Evaluation Of The System

In discussing the results of this implementation of dynamic lighting, several findings may be identified:

1. The kinetic depth effect (an extremely valuable depth cue) requires 3-D rotation of a scene. For scenes of arbitrary complexity, this task is too computationally expensive in a conventional shaded raster display environment to provide reasonable performance and therefore reasonable depth cues for the viewer.

2. A more practical approach is to leave the contents of the frame buffer as static and modify the much smaller video look-up table. This scheme allows significant mapping flexibility to alter the display of arbitrarily complex scenes. This work takes advantage of the mapping to enable dynamic modification of a light source's orientation to result in real-time changes in "lighting".

3. With this approach, an object must be stationary, but the user may vary the orientation of a light source about the object to discover information about the depth relationships within a scene. (See Figure 5-1.)

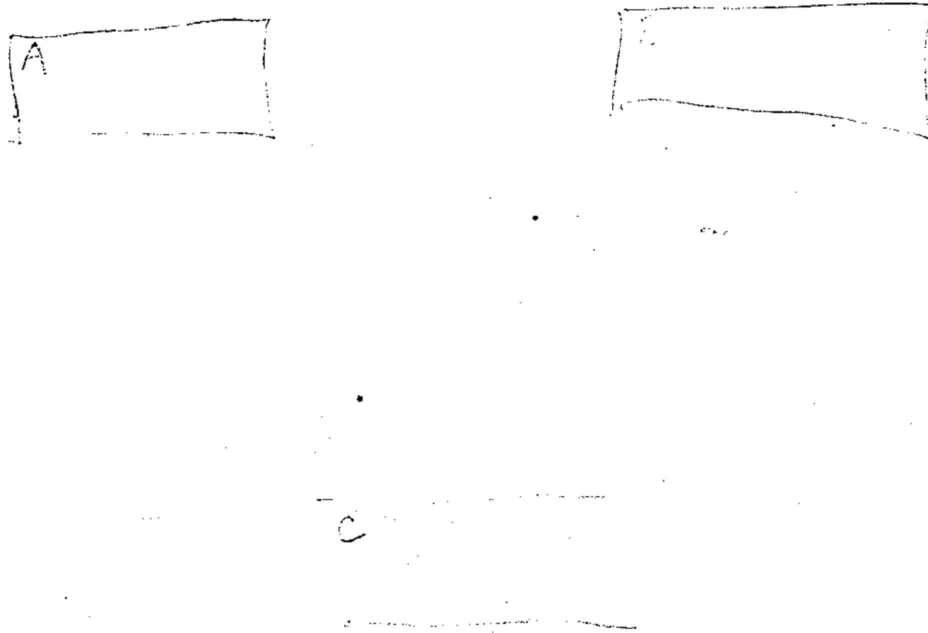


Figure 5-1.

4. Since depth information is not encoded in the frame buffer for this implementation, no shadows (which require depth information) may be generated dynamically using the video look-up table. However, even if depth information were maintained, the computational expense of calculating the shadows would degrade the real-time performance.

5. Real-time changes in the display (at a rate of 1/16 seconds per video look-up table update) dramatically improve the perception of depth relationships by the viewer.

6. The user interface employed in this work, involving a more natural mechanism for entering and detecting the orientation of a light source, provides a higher degree of realism for the viewer.

This project's results are an improvement over the previous state-of-the-art for the video look-up table mapping technique. The user interface is a significant improvement, since the user employs an actual hand-held light source as a light source orientation control to "light up" the scene in real-time from arbitrary orientations within a video camera's field of view. This use of an actual light source to control the light source orientation is a natural interface and does not have to be learned. By contrast, Daniel Bass did not employ a facile user interface to assist in the human factors aspect of the process. In addition, performance drawbacks in Bass's work detracted from the perception of the user, since slower feedback (1 sec. per VLT update as opposed to 1/16 sec. per VLT update in this work) exacerbates the problem the user has in conveniently perceiving differences in successive frames.

This work also extended the realism of the lighting model by utilizing larger video look-up tables. Limitations in the display device posed a problem for Bass in his attempts to enhance the realism of the scene. Specifically, the small color table he was forced to use resulted in large quantization errors in the establishment of surface normal equivalence classes. Bass's work was limited to monocolor due to the nature of his display device.

5.2 The Future

It is useful to discuss the problems that exist with the implemented system, the future changes that are possible as enhancements to or extensions of the existing system, and the areas of major new work.

5.2.1 Improvements

The major problems with the existing implementation are the system usage requirements, the memory requirements, and the hardware interrelationships. Dedicated (or high-priority) system usage is required to insure dynamic (almost real-time) performance. Furthermore, large amounts of virtual memory and disk space are required to save pre-calculated video look-up tables (in excess of 1 Megabyte). Also, the dependence on so much interrelated hardware that is used for other purposes constitutes a problem. It is not uncommon, for example, for several of the system components to be cannibalized for other purposes, making progress extremely frustrating at times.

The main memory requirements for my implementation of the dynamic lighting system seem expensive. My interest was to emphasize human factors of depth perception and realism in visualization accomplished in real time, and not to build

a cost-effective system. Hence, no restrictions were imposed on the use of available resources. Some improvements in reducing resource utilization may be possible.

Extensions to improve the accuracy of the light source detection could be implemented. Specifically, dual video camera light source detection could be implemented to evaluate the precise location of a light source in the room.

Some experiments were undertaken to evaluate the feasibility of this approach. I determined that for dual camera tracking using the hardware available at the time, the upper limit on the speed of light source detection would be 7.5 Hz. This limit existed because there were two 30 Hz. cameras between which the system was switching and a frame (or cycle) was lost each time the cameras switched since the Vax was in charge of the video multiplexer and was running asynchronously to the digitizing system. A rate of 7.5 Hz. is much slower than the 16 Hz. rate that I achieved with a single video camera, so I abandoned the dual camera tracking approach.

In the area of depth cues, a number of additional techniques may be attempted to supplement the existing dynamic lighting depth cues. For example, intensity depth cues, where the intensity of objects is a function of their depth, as well as their illumination, could be implented. Under the current 64-color scheme, only 3 levels of intensity are allowed (since black would be excluded) and an alternative technique would need to be developed if finer intensity resolution than three values is required.

In addition, we could display orthogonal projections as a depth cue. More specifically, the display of several orthogonal parallel projections simultaneously allows the user to comprehend better the 3-D structure. This mechanism may be implemented in a straightforward manner in the current software. However, the light source orientation is screen-oriented, so simultaneously there will be four separate lighting conditions for four orthogonal views. Furthermore, the technique may have limited usefulness in the display of molecules, which consist of spherical components.

Mechanisms which enable the user to store multiple pre-calculated images in the frame buffer and to cycle quickly among the images might be employed to animate a scene. In this manner, we may provide some means of kinetic depth effects in addition to dynamic lighting. This technique seems like a viable consolidation of two depth perception techniques, although there is a notable trade-off in resolution in some instances for the sake of animated motion. Jim Lipscomb, a former graduate student at UNC, has

implemented such an approach at UNC, with some success.

In addition, the techniques employed in fractal surface generation might easily be extended to support the dynamic lighting technique. Another extension that would enhance the realism of the displayed scene and could be incorporated easily into this work is the idea of "texture mapping".

"Texture mapping is a technique which creates a shaded representation of a 3-D texture (leather, ceramic, fur, etc.) which is then applied to an object model. A variation on texture mapping, known as bump mapping, was invented by JPL's Blinn. This technique simulates low-relief wrinkles and bumps on a surface by means of lighting and shading effects. The bump map does not actually change the shape of the model; it merely stores normal vectors (the direction perpendicular to the surface at each point), which are compared with the light source to yield shading values. By "wiggling" the normal vectors to produce light and dark areas, the bump map creates the illusion of a 3-D texture. In contrast to a texture map, the shading values are not stored but must be recalculated when the model moves relative to the light source." [7]

The bump mapping technique can be incorporated into dynamic lighting by storing, for some of an object's pixels, the encoded normals from adjacent equivalent classes as opposed to the calculated encoded normal.

5.2.2 Suggestions For Future Research

Continuing research on new algorithms that combine image complexity with speed of generation will always be of high priority for much of the work in computer graphics. The following suggestions for major work as extensions of this work provide an insight into some of the remaining problems in this area.

An alternative to the scheme of using an array of pre-calculated video look-up tables would be to use a dedicated processor to calculate video look-up tables "on the fly". However, I implemented this scheme using the Vax as a dedicated processor, and the processing requirements were sufficient to degrade the performance. Nevertheless, alternative architectures for processors (e.g., parallel processors) might be developed to make this approach more viable.

A more accurate lighting model would improve the realism of the display. In this work, I modeled the light source as if it were infinitely far away, so that all of the incoming light rays are parallel to each other. This assumption greatly simplified the calculations but limited

the quality of the images.

Modeling the position of the light source within the object space, however, would enable better comprehension of 3-D models, by enabling the user to move the light source through a crevice, for example. However, the mechanism of using the video look-up tables to accomplish a mapping from a particular encoded normal to an intensity depends upon the notion that all normals of a common equivalence class will have the same intensity, which is not necessarily true if the light rays are not parallel. Hence, the combination of the video look-up table mapping technique with the mechanism of specifying a light source position (with non-parallel light rays) is not feasible.

Besides the hardware "frame cycling" technique discussed in the previous section, other techniques in the area of animation in conjunction with dynamic lighting might be an area for fruitful and interesting research. Brute force approaches, such as mapping frame buffer rasters from disk to the frame buffer, appear to be too slow for appropriate animation speeds. However, it may be possible to change only the portion of the frame buffer that has changed from the previous frame, so that we take advantage of frame coherence, and may provide significant performance gains.

For the dynamic lighting technique, shadows may be calculated relative to a particular light source at image preparation time, but those shadows may not be dynamically changed when the light source changes. A method might be established for generating shadows based on the opaque visible portions of the model if depth information is stored.

Currently, in the 64-color scheme employed in this work, there are 6 bits in the frame buffer that are not used. However, 6 bits worth of depth information (64 depth values) may not be sufficient to generate realistic looking shadow effects. Additional research is recommended in this area to see if the very useful addition of shadows as part of the dynamic lighting depth cues might be included.

5.2.3 My Ideal System

My ideal system would:

1. have a dedicated Vax with lots of main memory (possibly a micro-Vax) to support a larger number of color tables, enabling smoother lighting transitions or a wider range of lighting angles.

2. have an extra Megabyte of frame buffer memory in the Ikonas.

3. have a pair of dedicated 30 Hz. video cameras (running synchronously with the Vax so the digitizer can digitize a pair of images in 15 Hz.)

4. have a toggle switch to a dedicated set of cables for the video cameras (as opposed to re-wiring most of the cables in the video multiplexer each time).

5. have a high resolution 30 Hz. monitor or a 60 Hz. digitizer and video cameras.

6. have permanent wall mounts for the video cameras relative to the placement of the 30 Hz. monitor (as an official workstation).

5.3 Summary And Conclusions

The dynamic lighting (DYNALITE) system has the potential to be very useful in a viewer's visualization process to asses depth cues and to improve the realism of a scene. More specifically, the user may find the system especially useful in evaluating "crevices" and collections of objects that are oriented approximately along the line of sight.

The real-time changes in intensity provide the user almost instantaneous feedback and therefore more control over what he desires to see. In addition, the "natural movements" associated with the user's dynamic orientation of the light source directly correspond to real life movements and are comfortable and easy to comprehend.

This implementation of dynamic lighting (DYNALITE) (a) is useful in providing depth cues, (b) provides a higher degree of realism, and (c) is easy to use (it is "natural" in the sense that it models natural single light source lighting conditions in real-time). This claim is based in part on my own impressions after observing users of the system. The sequence of steps employed by a first-time user often proceeds as follows:

1. The user expresses some measure of enthusiasm at the "natural" manner in which the user interface works, the closeness of the model to the "real world", and the realism of the perceived display.

2. The user then becomes more exploratory in his efforts to see what he can learn about the model, e.g., crevices, and becomes immersed in the process

of working directly with the model. The user, in general, appears to be unaware of the user interface, and his adjustment time to the user interface seems very fast.

There is strong evidence that this dynamic lighting technique (including the "natural" user interface) provides a higher degree of realism for the user than a system without dynamic lighting and constitutes a powerful visualization and conceptualization tool. The real-time nature of this technique is a fundamental part of the viewer's realistic perception of the modeled scene.

Users of the dynamic lighting system do not have enough collective experience, however, to give a definitive answer as to its significance in providing depth cues. It appears that, while dynamic lighting is useful in that regard, the kinetic depth effect remains a more powerful technique for providing depth cues. The problem remains of how to accomplish such an effect on raster-scan devices in real-time for arbitrarily complex objects and scenes.

A SYSTEM USER'S MANUAL

The purpose of this section is to assist the user in understanding the options available with the system and the mechanisms to invoke them.

A.1 Introduction - Purpose Of The System

A system has been designed and implemented in which the direction of a light source can be modified dynamically to provide real-time changes in intensity based on the direction of the light source. The real-time nature of this technique is a fundamental part of the viewer's realistic perception of the modeled scene.

A.2 System Initialization

In order to use the dynamic lighting system effectively, the user must understand the basic characteristics of the system. The system is complex in its interrelationships and all components of the system depend heavily on the workings of other components. Thus, the most critical item is to verify that all aspects of the system are functional and able to communicate with the rest of the system.

This verification may be accomplished by means of the following checklist:

- a functional point light source is available and in useable condition (note: such a light source is provided with the system)

- a video camera is available and functioning, and wiring to the digitizer is intact (the wiring in the UNC graphics lab is such that it is often reconfigured to support other applications on an as-needed basis); lens settings: f-stop = 8, focus = infinity.

- the digitizer is working and able to communicate to the frame buffer

- the frame buffer is in working order

- the monitor is turned on

- the bipolar microprocessor is working, able to read the frame buffer memory, and able to report to the host

- Vax 11/780 with a main memory complement of at least 1.5 Meg. in working order. Need dedicated access (or high priority access) to be able to achieve a display rate and light source detection rate that approaches real-time.

After logging onto the system, the user must perform the partial rendering step in preparation of dynamic lighting display. More specifically, raster data with encoded surface normals and color representing each pixel must be downloaded to the frame buffer prior to execution of the dynamic lighting software. This partial rendering (i.e., performance of all steps except for the shading itself) may be accomplished by:

1. running the program "shade", which is the "spheres only" z-buffer implementation; it takes GRIP-75 MOLCARDS format as input and calculates a "normals" image,

2. running a program written by the user (such as "wellbench") based on the tree-structure graphics package implementation; it takes polygon data as input and calculates a "normals" image, or

3. running the program "ikrdwr", which transfers stored raster data to the frame buffer.

A.3 System Execution

Once the frame buffer has been loaded, the user runs the C shell file "dynalite" to invoke the dynamic lighting software. In the initial state of the system, a simple prompt indicating the two system modes, calibrate or display, is displayed. The user may select either mode, but it is strongly recommended that the calibration mode be invoked before the display mode unless the system has been recently calibrated.

A.3.1 Calibration Mode

Due to the nature of the light source digitization process, it is necessary to invoke a calibration mode in order

1. to prevent the accidental detection as the light source of other illuminated or illuminating items in the room,
2. to guarantee the detection of the light source, and
3. to familiarize the user with the field of view limits of the video camera's lens.

When the calibration mode is invoked, a grid of dots is displayed which indicate the individual light source detection locations, or equivalence classes. The user must make sure that the video camera lens settings, the distance between the light source and the camera, the darkness of the room, and the threshold value in the light source detection process are calibrated so that only the light source is digitized above the threshold intensity and displayed. Furthermore, the user must ensure that the light source is always digitized into at least one grid dot (so that the light source won't "fall through the cracks" of the digitized image), but no more than four. If the light source is digitized into two to four adjacent grid dots, the closest grid dot to the previous value is chosen as the light source orientation.

The light source is being digitized at a coarse resolution to simplify the detection process, to optimize the speed of the detection process, and to minimize the impact on the frame buffer in terms of the number of bits that are required to digitize an image.

A.3.2 Display Mode

In display mode the user is interacting with the dynamic lighting process in order to modify the display. It is therefore useful to discuss both the input required from the user and the kinds of output that the user may expect to see.

The user employs the hand held light source to "light up" the scene from arbitrary orientations within the camera's field of view. The most important aspect of the system is simply what the user sees.

The reaction time of the system is approximately 1/16 of a second to detect the orientation of the light source and update the video look-up tables. Therefore, the feedback to the user is almost instantaneous and provides the user with a very natural mechanism for specifying light source orientations.

In addition, the system will alter the color of the border (from black to green) to give the user feedback when the light source is "out of bounds", i.e., out of the field of view of the video camera.

A.4 Demonstration Software

The user may wish to run some demonstrations using pre-computed raster data to familiarize himself with the system. The user may invoke the C shell file "demo1" to run a demonstration on a neurotoxin molecule using a "script" of function keys as input. The C shell files "demo2a" and "demo2b" may be invoked to download an image of the UNC well and benches and perform dynamic lighting based on the pen-light input.

A.5 System Termination And Exit

The user must issue a break command (by depressing a break key at the terminal from which the software is running) to terminate the dynamic lighting software and allow the user to exit. The user should then log off the terminal, and turn off the following equipment:

- point light source
- video camera
- 30 Hz. monitor

B SYSTEM HARDWARE CONFIGURATION (WIRING DIAGRAM)

The diagram below illustrates the relative placement and wiring of the principal system hardware components.

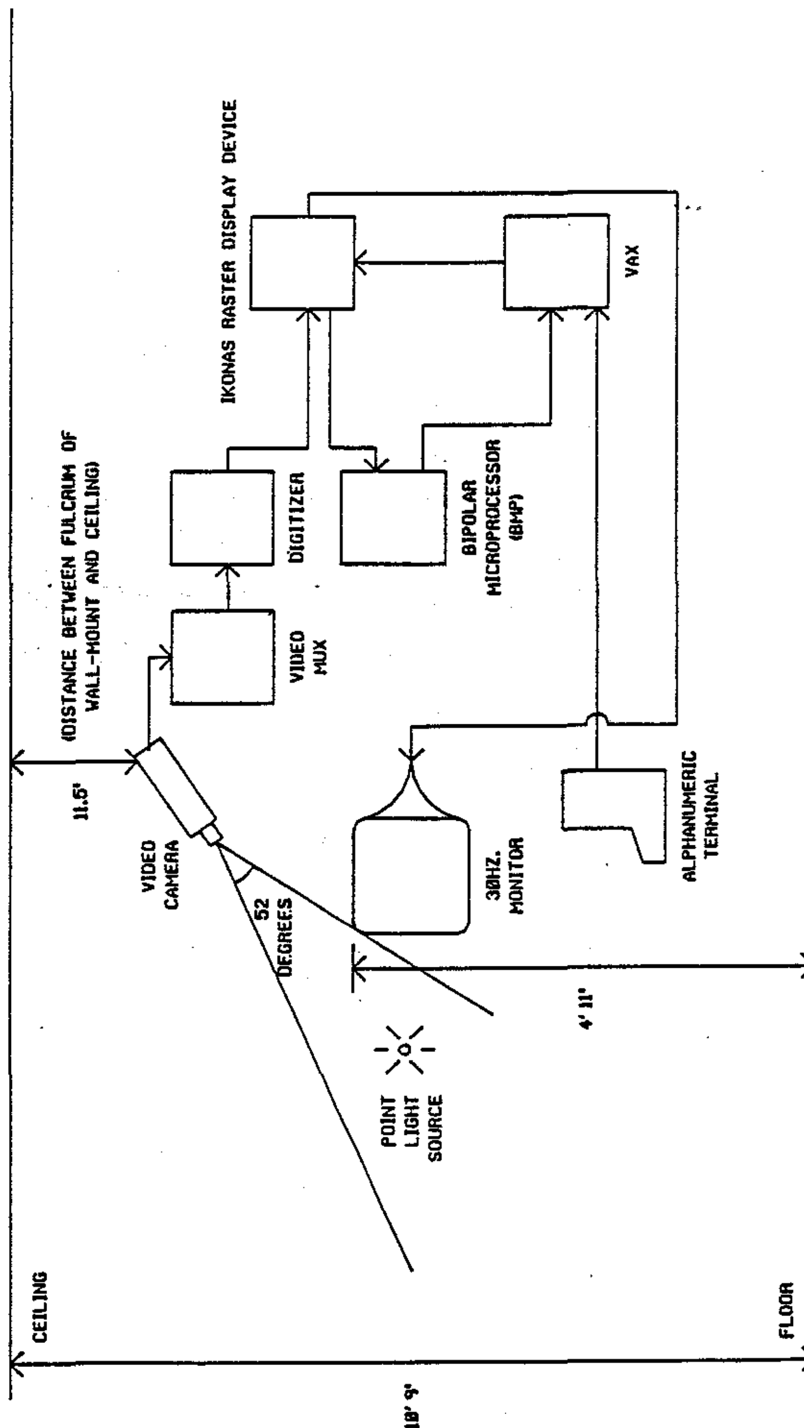


Figure B-1.

C PROGRAMMER'S MANUAL

The purpose of this section is to provide the reader with the information necessary to modify the software for the purpose of enhancing it.

C.1 Important Data Structures

The important data structures in this work include the following:

- frame buffer (pixel map of encoded normals)
- digitized raster (array of pixels containing digitized image)
- array of 33 by 33 complete video look-up tables (need 1.1 Meg. for storing video look-up tables array)
- array of pre-processed sphere prototype (with all normal equivalence classes assigned); dimensions are 162 by 162 pixels (10% of screen area)
- z-buffer (512 x 512 x 10) or tree for hidden surface processing

C.2 Software Overview

The "nlite2" program calculates and stores a 33 by 33 grid of color tables based on normals spaced at 16 pixels apart (at 512 by 512 screen resolution). The grid of color tables are stored in the file "nclrtbls.d". A shading parameters file is provided at color table calculation time to enable the user to make changes in the relative contributions of specular and diffuse components and in the exponent (n) which controls how "shiny" the surface appears. More specifically, the following values may be modified in the shading parameters file (shading.h):

- the diffuse illumination component, Id
- the illumination from the point light source, Ip

- the reflectance coefficient, Rp, at point P
- the exponent, n, of the cos(s) term in the intensity calculation
- the maximum possible intensity, MAXINTENS

To generate and store a prototype for a sphere as encoded normals for use in the "spheres only" z-buffer algorithm, the program "progen" is invoked. Parameters such as the total number of equivalence classes for normals and the number of rings for a prototype may be modified in "protoargs.h". The sphere prototype is stored in the file "proto.d".

The program "shade" (main program is cshade.c) samples into the sphere prototype array and stores sphere images as encoded surface normals in the frame buffer using a z-buffer technique. Input of molecules to the system consists of a specification in MOLCARDS format (standard GRIP-75 input), and sphere radii are calculated from a table of ideal bond lengths given the connectivity information of the atoms. The color of the atoms is determined by the type of atom; a simple table is used for this purpose.

Programs such as "wellbench" represent simple programs based on a graphics package. The hidden surface processing is accomplished by virtue of a tree-structure. See the documentation in the program listings for more details.

The utility "ikrdwr" is provided to save and restore the contents of a frame buffer. The raster data file is "pictfile".

The program "nflite3a" performs all dynamic lighting, whether from keyboard input or from light source detection through the digitization system.

The BMP program "box3asearch.g" works in conjunction with the C program "locate2.c" to search a portion of the frame buffer (into which the video camera is digitizing an image) in order to detect the location of the point light source.

Such C shell files as "calibrate" and "display" initialize the Ikonas digitizer, bipolar microprocessor, video look-up table, and crossbar for the calibration and display modes.

C.3 Computer Program Listings

```
/* makefile */

CFLAGS = -O

/unc/dh/surf2/proto.d : /unc/dh/surf2/prodesc.h progen.o
	cc progen.o -lm -o progen
	progen

clean:
	/bin/rm -f *.o a.out core mon.out
```

```
/* progen.c */

#define DBG 0 /* off */
#define DEBUG if (DBG) printf
#include <stdio.h>
#define PI 3.14159265
#define      PZERO    0
#define BLACK 1
#define X (int) (x+offset)
#define Y (int) (y+offset)
main (){ /* progen - shading and depth prototype generator. Holmes. 7 Nov 80*/

#include <math.h>
#define ACOS(A) ((y)>=(0) ? (acos(A)) : (2*PI-acos(A)))
#define MIN(A,B) ( (A)<(B) ? (A) : (B) )
#define MAX(A,B) ( (A)>(B) ? (A) : (B) )
#include "/unc/dh/surf2/prodesc.h"
FILE *pfile, *nfile, *fopen();
register int          i,j,k;
int rho, rhosq,
     ring;
static int   numpts[NUMRINGS+1],
               ptnum[NUMRINGS+1][MAXPTS+1],
               pointnum[3];
float        offset,
     x, y;
double       avg,
     temp, tempstep,
     maxangle,
     angle, anglestep,
     rad, radsq;
static double        r[NUMRINGS+3],
               xa[NUMRINGS+1][MAXPTS+1],
               ya[NUMRINGS+1][MAXPTS+1],
               theta[NUMRINGS+1][MAXPTS+1],
               phi[NUMRINGS+1],
               distsq[3],
               angdiff[3];
ZTYPE          za[NUMRINGS+1];
double       sinphi,
     np[3];
int sum;

/* initialize constants */
rho=PSIZE/2;
rhosq=rho*rho;
offset=rho+1;          /* used to convert from origin at center of array */
               /* (conceptual) to origin at lower left corner (actual) */
if (NULL == (pfile = fopen(PNAME, "w")))
     fprintf(stderr, "prototype file open err.\n");
if (NULL == (nfile = fopen(NORMLTBL, "w")))
     fprintf(stderr, "normal table file open err.\n");

/* zero prototype s and z */
```

```
/* progen.c */

for(i=0; i<PSIZE; i++)
     for(j=1; j<=PSIZE; j++)
               proto[i][j].ps = proto[i][j].pz = PZERO;

/* set up sphere prototypes for angles and depths */

/* For the angle prototype, the no. of "actual" pts. for which angles
     are specified may be less than the number of array elements;
     thus an algorithm which maps array elements to the "actual" pts.
     is used.                                                          */

/* The no. of "actual" pts. on a ring differs:  the inner rings have the
     fewest pts. and the outer rings have the most pts.  The algorithm
     starts with 1/4*(ave. no. of pts./ring) for the 1st (innermost)
     ring and procedes to 7/4*(ave. no. of pts./ring) for the last ring
     in (NUMRINGS-1) equal intermediate steps.                       */
numpts[0]=1;
avg=1.*TOTALPTS/NUMRINGS;
temp=(avg/4.)+.5;
tempstep=1.5*(avg/(NUMRINGS-1));
sum = 0;
for (i=1; i<NUMRINGS; i++) {
     numpts[i]=(int) (temp);
     sum = sum + numpts[i];
     temp=temp+tempstep;
     }
numpts[NUMRINGS] = TOTALPTS - sum;  /* put remaining pts. on last ring */

/* determine the radii for the rings as a function of the sine of the angles
     given by equal angle intervals                                    */
r[0]=0.0;
maxangle=asin((rho-.5)/rho);          /* last ring at r=rho-.5 */
anglestep=maxangle/NUMRINGS;
angle=anglestep;
DEBUG("angle1=%f\n", angle);
for (i=1; i<=NUMRINGS; i++) {
     r[i]=(((double)(i)/(double)(NUMRINGS)) * ((double)(rho) - .5) +
                         ((double)(rho) * sin(angle)))/2.0;
     DEBUG("r[%d]=%f\n", i, r[i]);
     angle=angle+anglestep;
     }
r[NUMRINGS+1]=rho;
r[NUMRINGS+2]=(rho*sqrt(2.))+1.;      /* max. dist.=dist. from center to a */
                                      /* corner of the square */
DEBUG("last 2 r[]'s:  %f %f\n", r[NUMRINGS+1], r[NUMRINGS+2]);

/* determine characteristics of "actual" pts.                          */
k=1;
for (i=0; i<=NUMRINGS; i++) {
     anglestep=2*PI/numpts[i];
     if (i/2!=i/2.) angle=anglestep/2.;       /* 1st pt. NOT on y=0 */
     else angle=0.0; /* 1st pt. on y=0 */
     DEBUG("angle2=%f anglestep2=%f\n", angle, anglestep);
     for (j=1; j<=numpts[i]; j++) {
```

```
/* progen.c */

                xa[i][j]=r[i]*cos(angle);
                ya[i][j]=r[i]*sin(angle);
                theta[i][j]=angle;
                ptnum[i][j]=k;
                angle=angle+anglestep;
                k++;
                }
        phi[i]=asin(r[i]/rho);
        za[i]=(ZTYPE)(rho*cos(phi[i])+.5);
        }

/* determine the closest "actual" pt. to ea. of the prototype pixels        */
for (y=rho-.5; y>-rho; y--) {
        for (x=.5-rho; x<rho; x++) {
                radsq=(x*x)+(y*y);
                rad=sqrt(radsq);
                for (i=1; rad>r[i]; i++);
                /* pixel center lies between rings i-1 and i          */
                if (i<NUMRINGS+2) {      /* pixel center is in circle */
                        angle=ACOS(x/rad);
                        /* determine 2 closest pts. on the 2 nearest rings */
                        ring=i-1;
                        for (k=1; k<=2; k++) {   /* for ea. of the 2 rings */
                                /* find the closest 2 "actual" angles    */
                                for (j=1; j<=numpts[ring]; j++) {
                                        if (angle<=theta[ring][j]) break;
                                        }

                                /* determine which angle is closest */
                                if (j==1) {
                                   distsq[k]=(x-xa[ring][1])*(x-xa[ring][1])
                                        +(y-ya[ring][1])*(y-ya[ring][1]);
                                   pointnum[k]=ptnum[ring][1];
                                   }
                                else if (j==numpts[ring]+1) {
                                   angdiff[1]=(theta[ring][1]+(2*PI))-angle;
                                   angdiff[2]=angle-theta[ring][numpts[ring]];
                                   if (angdiff[1]<=angdiff[2]) {
                                      distsq[k]=(x-xa[ring][1])*(x-xa[ring][1])
                                           +(y-ya[ring][1])*(y-ya[ring][1]);
                                      pointnum[k]=ptnum[ring][1];
                                      }
                                   else {
                                      distsq[k]=(x-xa[ring][numpts[ring]])*
                                                (x-xa[ring][numpts[ring]])+
                                                (y-ya[ring][numpts[ring]])*
                                                (y-ya[ring][numpts[ring]]);
                                      pointnum[k]=ptnum[ring][numpts[ring]];
                                      }
                                   }
                                else {
                                   angdiff[1]=theta[ring][j]-angle;
                                   angdiff[2]=angle-theta[ring][j-1];
                                   if (angdiff[1]<=angdiff[2]) {
```

```
                                        distsq[k]=(x-xa[ring][j])*(x-xa[ring][j])
                                                +(y-ya[ring][j])*(y-ya[ring][j]);
                                        pointnum[k]=ptnum[ring][j];
                                        }
                                      else {
                                        distsq[k]=(x-xa[ring][j-1])*
                                                  (x-xa[ring][j-1])+
                                                  (y-ya[ring][j-1])*
                                                  (y-ya[ring][j-1]);
                                        pointnum[k]=ptnum[ring][j-1];
                                        }
                                      }
                                    ring++;
                                    }
                            if (distsq[1]<=distsq[2])
                                    proto[Y][X].ps=pointnum[1];
                            else proto[Y][X].ps=pointnum[2];
                            proto[Y][X].pz=(ZTYPE)(sqrt(rhosq-radsq)+.5);
                            DEBUG("x=%f y=%f z=%f s=%d\n",x,y,proto[Y][X].pz,
                                    proto[Y][X].ps);
                            }       /* if (i<NUMRINGS+2) */
                    }       /* for (x=...) */
    }           /* for (y=...) */

/* set bounds marking elements used */
for(i=1; i<=PSIZE; i++) {
    pbounds[i].low  = PSIZE+1;
    pbounds[i].high = 0;
    for( j=1; j<=PSIZE; j++) if (proto[i][j].ps > (unsigned char)(0))  {
            pbounds[i].low  = MIN(j, pbounds[i].low);
            pbounds[i].high = MAX(j, pbounds[i].high);
            }
    }
/* write in lines to standard output */

fprintf(pfile, "%4d%4d%4d\n", PSIZE, BLACK, PSIZE);
for(i=1; i <= PSIZE; i++){
    fprintf(pfile, "%d %d\n", pbounds[i].low, pbounds[i].high);
    for(j=1; j <= PSIZE; j++){
            fprintf(pfile, (j==0 ?"%d":" %d"), (unsigned)(proto[i][j].ps));
            }
    putc('\n', pfile);
    for(j=0; j < PSIZE; j++){
            fprintf(pfile, (j==0 ?"%d":" %d"), (ZTYPE)(proto[i][j].pz));
            }
    putc('\n', pfile);
    }
fprintf(nfile, "%d\n", TOTALPTS+1);
for (i=0; i<=NUMRINGS; i++) {
    sinphi = sin(phi[i]);
    np[2] = cos(phi[i]);
    for (j=1; j<=numpts[i]; j++) {
            np[0] = sinphi * cos(theta[i][j]);
            np[1] = sinphi * sin(theta[i][j]);
```

```
/* progen.c */

                fprintf(nfile, "%d %.8f %.8f %.8f\n", ptnum[i][j],
                                np[0], np[1], np[2]);
                }
        }
exit(0);
}
```

```
/* prodesc.h */

#include "protoargs.h"
#define PXOK(x) (x>=0 && x<PSIZE)
#define PYOK(y) (y>=0 && y<PSIZE)
#define STYPE unsigned char
#define ZTYPE unsigned char

struct {
     STYPE ps;
     ZTYPE pz;
     } proto [PSIZE+1][PSIZE+1];
struct{
     int low;
     int high;
     } pbounds [PSIZE+1];
```

```
/* protoargs.h */

#define PSIZE 162
#define TOTALPTS 252          /* no. "actual" pts.; excludes center & background */
#define NUMRINGS 8
#define MAXPTS 150
```

```
/* makefile */

CFLAGS = -O -p

nflite3a: nflite3a.o locate2.o startik.o
	cc nflite3a.o locate2.o startik.o -lk ${CFLAGS} -o nflite3a

nlite2: nlcalc2.o normltbl.o getang.o mathf.o
	cc nlcalc2.o normltbl.o getang.o mathf.o -lm -lk ${CFLAGS} -o nlite2

frame: frame.o
	cc frame.o -lk ${CFLAGS} -o frame

redout: redout.o
	cc redout.o -lk ${CFLAGS} -o redout

litecalc2.o			: color3.h

litecalc2.o nflite3a.o normltbl.o	: normal.h

normltbl.o	: /unc/dh/surf2/prodesc.h

litecalc2.o			: shading.h

clean:
	/bin/rm -f *.o a.out core mon.out
```

```
/* nflite3a.c */

/* continuously read a light orientation angle & use it to set the color map */
/*   - dh, 12/80                                                              */
/* this routine currently runs at 30 updates/sec. */
#define DEBUG if(0) printf
#define DBUG if(0) printf
#include <math.h>
#include "normal.h"
#include <stdio.h>
#include <ikdefs.h>
#define DIGWIDTH 63 /* width of digitized image */
#define PSPACING 2  /* no. of pixels betw. grid pts. in digitized image */
#define HLFPSPAC PSPACING/2

typedef struct {
     unsigned rcmapi : 10;
     unsigned gcmapi : 10;
     unsigned bcmapi : 10;

     } CMAPITEM;
CMAPITEM (*clrmap)[TOTALPTS+3];
CMAPITEM (*clrbuf)[TOTALPTS+3];      /* clrbuf is buffer for calculating middle */
                                     /*  2 quadrants of color table, given the  */
                                     /*  last (full intensity) quadrant         */
CMAPITEM *mptr,      /* pointer to a location in clrmap */
      *bptr; /* pointer to a location in clrbuf */
CMAPITEM outbnds,    /* color when out-of-bounds border is turned "on"  */
      inbnds;        /* color when out-of-bounds border is turned "off"*/

quit () {
     fprintf (stderr, "Normal termination.\n");
     system ("date");
     exit(0);
}

main () {
#include <signal.h>
#include <sgtty.h>

register int x, y;
/*register int newx, newy;*/
int foundflag,
     xfound, yfound; /*vals[2],*/
int zero;

signal (SIGINT, quit);
setcmask (INVIO | RUNPROC | IKRESET);       /* invisible i/o:  set to inhibit */
setcmask (INVIO | RUNPROC);
     /*    transfers except during non-visible picture time (blanking)      */
     /* - setcmask (INVIO | RUNPROC | 02000) also allows processor to run */
/*startik ();*/      /* start ikonas - from gda, 3/27/82 */
setorig (TORIG);
```

```
/* nflite3a.c */


zero = 0;
x = y = 31;
xfound = yfound = 31;
foundflag = 0;
outbnds.rcmapi = 0;
outbnds.gcmapi = 0x0ff;       /* 1/4 of full intensity green (511) */
outbnds.bcmapi = 0;
inbnds.rcmapi = inbnds.gcmapi = inbnds.bcmapi = 0;  /* inbnds always off */
ctinit();
msg();
lightcalc (x, y);
system ("date");
for (;;) {
     /* read found flag from scratchpad */
     /*
     ikread (&foundflag, 1, 0106, 020200);
     */

#ifdef DBG
     fprintf(stderr, "Calling locate ... \n");
#endif

     locate(&xfound, &yfound, &foundflag);

#ifdef DBG
     fprintf(stderr, "xf,yf,foundflag: %d %d %d\n",xfound,yfound,foundflag);
#endif

     /* if found */
     if (foundflag != 0) {
               /* read xfound & yfound, & reset found flag */
               /*
               ikread (vals, 2, 0107, 020200);
               ikwrite (&zero, 1, 0106, 020200);
               newx = vals[0]/2;
               newy = vals[1]/2;
               */

               /* determine which color map to "poke" into VLT */
               if ((xfound != x) || (yfound != y)) {
                         x = xfound;
                         y = yfound;
#ifdef DBG
                         fprintf (stderr, "x,y: %d %d\n", x, y);
#endif
                         lightcalc (x, y);
                         }
               }  /* if (foundflag...) */

     else {  /* turn on "out of range" border */
               ikwrite (&outbnds, 4, IKADDR(CMAPADDR, 1023));
               }  /* else */

     }  /* for (;;) */
```

```
/* nflite3a.c */


}   /* main() */

lightcalc(x, y)
     register int x, y;
{

#ifdef OLDSTUFF
x = DIGWIDTH - x;     /* correction due to video camera pointing toward */
                  /*    user, not away from user */
x = (x+HLFPSPAC)/PSPACING;   /* floor of ... (integer divide) */
y = (y+HLFPSPAC)/PSPACING;
ikwrite( clrmap+(y*33 + x), sizeof(*clrmap)/sizeof(CMAPITEM), 0, CMAPADDR);
#endif

#define OFFSET 0

x = (OFFSET + DIGWIDTH) - x;           /* correction due to video camera pointing */
                                 /*    toward user, not away from user */
y = y - OFFSET;

x = (x+HLFPSPAC)/PSPACING;   /* floor of ... (integer divide) */
y = (y+HLFPSPAC)/PSPACING;

#ifdef DBGLOC
     fprintf(stderr,"x,y: %d %d \n",x,y);
#endif

/*
ikwrite( clrmap+(y*33 + x), sizeof(*clrmap)/sizeof(CMAPITEM), 0, CMAPADDR);
*/
#ifdef DBGW
     fprintf(stderr,"writing ... \n");
     fprintf(stderr,"clrmap,x,y,clrmap+(y*33+x): %x %d %d %x \n",
               clrmap,x,y,clrmap+(y*33+x));
     fprintf(stderr,"sizeof(*clrmap),IKADDR(CMAPADDR,0): %d %x \n",
               sizeof(*clrmap),IKADDR(CMAPADDR,0));
#endif

/* poke the "full intensity" colormap into the top quadrant of the */
/*   physical Ikonas color map                                      */
ikwrite( clrmap+(y*33 + x), sizeof(*clrmap), IKADDR(CMAPADDR,768));

/* calculate the "1/3 intensity" colormap & poke it into the 2nd     */
/*   quadrant of the physical Ikonas color map                       */
mptr = &clrmap[(y*33 + x)][0];
bptr = &clrbuf[0][0];
while (bptr < &clrbuf[0][TOTALPTS+3]) {
     bptr->rcmapi = bptr->gcmapi = bptr->bcmapi = (mptr->rcmapi / 3);
     ++bptr;
     ++mptr;
     }
ikwrite( clrbuf, sizeof(*clrbuf), IKADDR(CMAPADDR,256));

/* calculate the "2/3 intensity" colormap & poke it into the 3rd     */
```

```
/* nflite3a.c */


/*  quadrant of the physical Ikonas color map                              */
bptr = &clrbuf[0][0];
while (bptr < &clrbuf[0][TOTALPTS+3]) {
     bptr->rcmapi = bptr->gcmapi = bptr->bcmapi = (2 * bptr->rcmapi);
     ++bptr;
     }
ikwrite( clrbuf, sizeof(*clrbuf), IKADDR(CMAPADDR,512));

/* the "no intensity" (i.e., black) colormap has already been poked */
/*  by a shell file & will never be changed, except the out-of-      */
/*  bounds border, which is turned off here                          */
ikwrite (&inbnds, 4, IKADDR(CMAPADDR, 1023));
}

msg () {

printf("\nUse penlight to specify the orientation of the light source.\n");
printf("Hit SHIFT & DEL key to quit.\n");
}

ctinit() {
#define CTNAME "/unc/dh/cmap8/n3clrtbls"
CMAPITEM *malloc();
FILE *ctfile, *fopen();
register int x, y;

fprintf(stderr, "Loading color tables into core . . .\n");
if (NULL == (ctfile = fopen(CTNAME, "r")))
               fprintf(stderr, "nclrtbls file open error.\n");
setbuf(ctfile, malloc(BUFSIZ));
clrmap = (CMAPITEM **) malloc (33 * 33 * sizeof(*clrmap));
for (y=0; y<=32; y++) {
     for (x=0; x<=32; x++) {
               fread (clrmap + (y * 33 + x), sizeof(CMAPITEM),
                              sizeof(*clrmap)/sizeof(CMAPITEM), ctfile);

               /* set location 768 in r, g, and b color maps (0th entry */
               /* in top quadrant) to full intensity (255); this assign-*/
               /* ment allows a range of 64 colors (2 bits r, 2 bits g, */
               /* 2 bits b) for the background                          */
               clrmap[(y*33 + x)][0].rcmapi = 0x3ff;
               clrmap[(y*33 + x)][0].gcmapi = 0x3ff;
               clrmap[(y*33 + x)][0].bcmapi = 0x3ff;
               }
     }

clrbuf = (CMAPITEM **) malloc (sizeof(*clrbuf));     /* allocate clrbuf */
}
```

```c
/* locate2.c */

struct { int mask,
         Xloc,
         Yloc,
         found; } interface;

#define SP(a) ((0202<<16)|a)
#define LO(a) ((a) & 0x3ff)
#define HI(a) ((a) >> 10)
#define IA SP(0100)
#define DONE SP(0200)
#include <ikdefs.h>
#define ENPROC 02000
#define MASK 0x80;
#include <stdio.h>

locate (x, y, f)
int *x, *y, *f;
{
    int done, reads;

    setcmask(0);

#ifdef DEBUG
    fprintf(stderr, "Beginning locate ...; x,y,f: %d, %d, %d\n", *x, *y, *f);
#endif

    /* initialize interface */
    interface.mask = MASK;
    interface.Xloc = *x;
    interface.Yloc = *y;
    ikwrite(&interface, sizeof(interface), IA);
    done = 0;
    ikwrite(&done, sizeof(done), DONE);

#ifdef DEBUG
    fprintf(stderr, "Resetting & starting processor ...\n");
#endif

    /* reset and start processor */
    setcmask(IKRESET|RUNPROC);
    setcmask(RUNPROC|INVIO);

#ifdef DEBUG
    fprintf(stderr, "Beginning wait for search complete signal ...\n");
#endif

    /* wait for search complete signal */
    reads=0;
    do {
            ikread (&done, sizeof(done), DONE);

#ifdef DEBUG
```

```
/* locate2.c */

    fprintf(stderr, "reads = %d, done = %d\n", reads, done);
#endif

            reads++; }
    while (done == 0);

#ifdef DEBUG
    fprintf(stderr, "Finishing wait ...; reads = %d\n", reads);
#endif

    /* report location and value found */
    ikread(&interface, sizeof(interface), IA);
    *x = interface.Xloc;
    *y = interface.Yloc;
    *f = interface.found;
#ifdef DEBUG
    fprintf(stderr, "Finishing locate ...; x,y,f: %d, %d, %d\n", *x, *y, *f);
#endif
    return(reads);
}
```

```
/* startik.c */

/* start ikonas bmp at location 0 */

#include <ikdefs.h>

startik() {
     int null;

     setcmask(IKRESET|RUNPROC); /* reset processor */
     ikwrite(&null, 0, 0);
     setcmask(RUNPROC); /* enable processor and let it go */
     ikwrite(&null, 0, 0);
}
```

```
/* normal.h */

/* there are TOTALPTS+2 normals of dimension VECTSIZE */
#include "/unc/dh/surf2/protoargs.h"
#define VECTSIZE 3

double          np[TOTALPTS+2][VECTSIZE],
     nptemp[VECTSIZE];
```

```
/* nlcalc2.c */

/* continuously read a light orientation angle & use it to set the color map */
/*  - dh, 12/80                                                                */
/* this routine currently runs at ~3.0 updates/sec.; 30 updates/sec. ideal    */
#define DEBUG if(0) printf
#define DBUG if(0) printf
#define DISKWR 1
#include <math.h>
#include "normal.h"
#include <stdio.h>

#ifdef DISKWR
#define CTNAME "/unc/dh/cmap8/n3clrtbls"
char *malloc();
FILE *ctfile, *fopen();
#endif

static int compens[TOTALPTS+2][3];

main () {
register double lightang[4];
#include <ikconsts.h>
#include <sgtty.h>
#define MAXC 511
#define MINC 0
#define DELTAC 16      /* ideally, 4 for 1 degree resolution */
                      /* currently set to ~(30/3.0)*4 for speed */
register int x, y;

normlinit();
compinit();
#ifdef DISKWR
     if (NULL == (ctfile = fopen(CTNAME, "w")))
                        fprintf(stderr, "nclrtbls file open error.\n");
     setbuf(ctfile, malloc(BUFSIZ));
#endif
for (y=0; y<=MAXC; y=incr(y)) {
     for (x=0; x<=MAXC; x=incr(x)) {
              getang (lightang, x, y);
              lightcalc (lightang);
              }
     }
}

incr(i)
     register int i;
{
if (i+DELTAC == MAXC+1) return (MAXC);
else return (i+DELTAC);
}

decr(i)
     register int i;
```

```
/* nlcalc2.c */

{
if (i-DELTAC >= MINC) return (i - DELTAC);
else return (MINC);
}

lightcalc(lightang)
     register double *lightang;
{
#include "shading.h"
#include "color3.h"
#include <ikconsts.h>
#define HALFPI 1.57079632
#define CORRECT(C) ((C >= 0.0) ? (C) : (0.0))
static double           l[VECTSIZE],
                npprime[VECTSIZE],
                spprime[VECTSIZE],
                vprime[VECTSIZE];
register int            ptnum,
                k;
register double         cosi,
                coss;
typedef struct {
     unsigned rcmapi : 10;
     unsigned gcmapi : 10;
     unsigned bcmapi : 10;

     } CMAPITEM;
CMAPITEM clrmap[TOTALPTS+3];
register int intens;
double normalp ();
double          w(),
     powr();

/* cos(theta) = lightang[0] */
/* sin(theta) = lightang[1] */
/* cos(phi) = lightang[2] */
/* sin(phi) = lightang[3] */

l[0] = lightang[3] * lightang[0];
l[1] = lightang[3] * lightang[1];
l[2] = lightang[2];

vprime[0] = -lightang[3];
vprime[1] = 0.0;
vprime[2] = lightang[2];

#if (REDON)
clrmap[0].rcmapi = compens[BACKGR][0];
clrmap[TOTALPTS+2].rcmapi = compens[BACKGR][0];      /* reserved for "out of */
                                                     /*    range" border */
#else
clrmap[0].rcmapi = compens[RDEFAULT][0];
clrmap[TOTALPTS+2].rcmapi = compens[RDEFAULT][0];
#endif
```

```
/* nlcalc2.c */

#if (GREENON)
clrmap[0].gcmapi = compens[BACKGR][1];
clrmap[TOTALPTS+2].gcmapi = compens[BACKGR][1];
#else
clrmap[0].gcmapi = compens[GDEFAULT][1];
clrmap[TOTALPTS+2].gcmapi = compens[GDEFAULT][1];
#endif
#if (BLUEON)
clrmap[0].bcmapi = compens[BACKGR][2];
clrmap[TOTALPTS+2].bcmapi = compens[BACKGR][2];
#else
clrmap[0].bcmapi = compens[BDEFAULT][2];
clrmap[TOTALPTS+2].bcmapi = compens[BDEFAULT][2];
#endif

for (ptnum=1; ptnum<=TOTALPTS+1; ptnum++) {
    for (k=0; k<VECTSIZE; k++) {
            nptemp[k] = normalp(ptnum, k);
            }

    npprime[0] = (nptemp[0]*lightang[0]*lightang[2]) +
                 (nptemp[1]*lightang[1]*lightang[2]) -
                 (nptemp[2]*lightang[3]);
    npprime[1] = -(nptemp[0]*lightang[1]) +
                 (nptemp[1]*lightang[0]);
    npprime[2] = (nptemp[0]*lightang[0]*lightang[3]) +
                 (nptemp[1]*lightang[1]*lightang[3]) +
                 (nptemp[2]*lightang[2]);

    spprime[0] = 2.*npprime[2]*npprime[0];
    spprime[1] = 2.*npprime[2]*npprime[1];
    spprime[2] = (2.*npprime[2]*npprime[2]) - 1.;

#ifdef DBG
    for (k=0; k<VECTSIZE; k++) {
            printf("%.8f %.8f %.8f %.8f %.8f\n", l[k],nptemp[k],npprime[k],
                        spprime[k], vprime[k]);
            }
#endif

    /* calculate dot products */
    cosi = (l[0] * nptemp[0]) + (l[1] * nptemp[1]) + (l[2] * nptemp[2]);
    /* i = acos(cosi); */
    coss = (spprime[0] * vprime[0]) + (spprime[1] * vprime[1]) +
           (spprime[2] * vprime[2]);

    cosi = CORRECT(cosi);
    coss = CORRECT(coss);

    DEBUG("** %f %f\n", cosi, coss);

    intens = ((int)((R*ID)+((R*cosi + w(cosi)*powr(coss,N)) * IP) + .5));
    DEBUG("* %d\n", intens);
```

```
/* nlcalc2.c */

#if (REDON)
     clrmap[ptnum].rcmapi = compens[intens][0];
#else
     clrmap[ptnum].rcmapi = compens[RDEFAULT][0];
#endif
#if (GREENON)
     clrmap[ptnum].gcmapi = compens[intens][1];
#else
     clrmap[ptnum].gcmapi = compens[GDEFAULT][1];
#endif
#if (BLUEON)
     clrmap[ptnum].bcmapi = compens[intens][2];
#else
     clrmap[ptnum].bcmapi = compens[BDEFAULT][2];
#endif

     }
#ifdef IKWR
ikwrite( clrmap, sizeof(clrmap)/sizeof (CMAPITEM), 0, CMAPADDR);
#endif
#ifdef DISKWR
fwrite (clrmap, sizeof(CMAPITEM), sizeof(clrmap)/sizeof(CMAPITEM), ctfile);
#endif

}

double w(i)
     register double i;
{
if (i >= 0.0) return (1.0 - (0.5 * i));
else return (0.0);
}

compinit () {
#define CNAME "/unc/dh/cmap4/lin.ctbl"
register int i, j;
int temp;
FILE *cfile, *fopen();

if (NULL == (cfile = fopen (CNAME, "r")))
     fprintf (stderr, "compensation table file open err.\n");

for (i=0; i<=TOTALPTS+1; i++) {
     for (j=0; j<3; j++) {
               fscanf (cfile, "%d", &temp);
               DBUG ("%d ", temp);
               compens[i][j] = temp;
               DBUG ("%d ", compens[i][j]);
               }
     DBUG ("\n");
     }
}
```

```
/* normltbl.c */

/* maintain prototype normal table */
#include <stdio.h>
#include "/unc/dh/surf2/prodesc.h"
#include "normal.h"

normlinit () {
FILE *nfile, *fopen();
int tblsize;
int ptnum;
double      x,
     y,
     z;
register int i;

if (NULL == (nfile = fopen (NORMLTBL, "r")))
               fprintf (stderr, "normal table open error.\n");
fscanf (nfile, "%d", &tblsize);
if (tblsize != TOTALPTS+1) {
     fprintf (stderr, "normal table size error.\n");
     return;
     }
for (i=1; i<=TOTALPTS+1; i++) {
     fscanf (nfile, "%d %F %F %F", &ptnum, &x, &y, &z);
     if (i != ptnum) {
               fprintf (stderr, "normal table read error at ptnum = %d.\n",
                              ptnum);
               return;
               }
     np[ptnum][0] = x;
     np[ptnum][1] = y;
     np[ptnum][2] = z;
     }
}

double normalp (i, j)
     register int i,j;
{
if ((i>=1) && (i<=TOTALPTS+1)) return (np[i][j]);
else return (0.0);  /* ???? */
}
```

```
/* getang.c */

/* return a value for the current light orientation angle */
#include <math.h>
getang (lightang, x, y)
     register double *lightang;
     register int      x, y;
{
#define Z 128.
#define ZSQUARE 16384.
#define OFFSET 255

register double r;
register double rho;

x = x - OFFSET;
y = OFFSET - y;
r = sqrt((double)(x*x + y*y));
rho = sqrt((double)(r*r + ZSQUARE));
lightang[0] = ((r>0.0) ? ((double)(x)/r) : (1.0));
lightang[1] = ((r>0.0) ? ((double)(y)/r) : (0.0));
lightang[2] = ((rho>0.0) ? (Z/rho) : (1.0));
lightang[3] = ((rho>0.0) ? (r/rho) : (0.0));

}
```

```
/* mathf.c */

double
powr (base, expon)
     register double base;
     register int expon;
{
register double      temp1,
                     temp2;
switch (expon) {
     case 0:
               return (1.0);
               break;
     case 1:
               return (base);
               break;
     case 2:
               return (base * base);
               break;
     case 3:
               return (base * base * base);
               break;
     case 4:
               temp1 = base * base;
               return (temp1 * temp1);
               break;
     case 5:
               temp1 = base * base;
               return (temp1 * temp1 * base);
               break;
     case 6:
               temp1 = base * base;
               return (temp1 * temp1 * temp1);
               break;
     case 7:
               temp1 = base * base;
               return (temp1 * temp1 * temp1 * base);
               break;
     case 8:
               temp1 = base * base;
               temp2 = temp1 * temp1;
               return (temp2 * temp2);
               break;
     case 9:
               temp1 = base * base;
               temp2 = temp1 * temp1;
               return (temp2 * temp2 * base);
               break;
     case 10:
               temp1 = base * base;
               temp2 = temp1 * temp1;
               return (temp2 * temp2 * temp1);
               break;
     default:
```

/* mathf.c */

```
            return (base);
    }
}
```

```
/* color3.h */

/* intensity and point number constants                                          */
/*  REDON, etc. :  indicate whether the particular color is turned "on" */
/*  PTBACKGR      :  background value for point number                          */
/*  BACKGR        :  background value for intensity                             */
/*  RDEFAULT, etc. :  indicate the default value for intensity for the  */
/*              particular color                                                 */

/*******                                                                          */
/**note*  last 4 values have NOT been shifted left twice                   */
/*******                                                                          */

#define REDON 1
#define GREENON 1
#define BLUEON 1
#define PTBACKGR 0
#define BACKGR 1      /* 40 originally */
#define RDEFAULT 0
#define GDEFAULT 0
#define BDEFAULT 0
#define ZERO 0
```

```
/* shading.h */

#define R .7          /* reflectance coefficient; ranges from 0 to 1 */
#define N 3 /* exponent of cos(s) term in intensity calculation */
#define MAXINTENS 220.        /* 255 originally */
/* ID + 2*IP = MAXINTENS */
#define ID 20.  /* diffuse illumination; MAXINTENS/11. */
#define IP 100.       /* illumination from point source; 5*MAXINTENS/11. */
```

```
/* box3asearch.g */

search {
     origin 10;
     register mcsi=r0, left, right, mask, ystep, x, y, control, doneflag;
     register yoffset;
     constant NXTCOM=2, STEP=2, MASK=0, XLOC=1, YLOC=2, FLAG=3;
     constant FND=0xffff, NOTFND=0, DONE=03, DONELFT=01, DONERGT=02;
     constant LFTLMT=2, RGTLMT=126, HEIGHT=63, YOFFSET=2;

     # get address of control words from mcsi parameter
     mar = mcsi++;
     mcsi++;
     read control = bus;

     # load control words
     load(mask, control+MASK);
     load(right, control+XLOC);

     # initialize left and right search addresses
     right SLL= right;
     left = right;
     left += STEP;
     ystep = STEP;
     ystep = ystep.hr;

     # initialize yoffset
     yoffset = YOFFSET;
     yoffset = yoffset.hr;

     # search columns progressing to the left and to the right
     doneflag = 0;
     loop
               # search left
               mar = left -= STEP;
               left - LFTLMT;
               mar += yoffset
               if(pos)
                         lread mash
                         mar += ystep;
                         mdr = bus;
                         mdr & mask
                         do HEIGHT times
                                   lread mash
                                   mar += ystep
                                   if(! zero)
                                             goto found;
                                   fi;
                                   mdr = bus;
                                   mdr & mask
                         repeat;
               else
                         doneflag |= DONELFT;
               fi;
```

/* box3asearch.g */

```
			# search right
			mar = right += STEP;
			RGTLMT - right;
			mar += yoffset
			if (pos)
				lread mash
				mar += ystep;
				mdr = bus;
				mdr & mask
				do HEIGHT times
					lread mash
					mar += ystep
					if(! zero)
						goto found;
					fi;
					mdr = bus;
					mdr & mask
				repeat;
			else
				doneflag |= DONERGT;
			fi;

			# check for completion
			doneflag ^ DONE;
	while(! zero);

	# if we arrive here the point was not found

	store(NOTFND, control+FLAG);
	goto NXTCOM;

found:
	# we come here to report the location of the dot
	x SRL= mar.low;
	y = mar;
	y SRL= y.hs;
	y -= 2;

	store(x, control+XLOC);
	store(y, control+YLOC);
	store(FND, control+FLAG);
	goto NXTCOM;
}
```

```
/* frame.c */

/* before Mar. '82 ...
#define TWIDTH 56
#define LWIDTH 28
#define RWIDTH 1
#define BWIDTH 48
*/
#ifdef OLDSTUFF
#define TWIDTH 36 /* 76 without moned settings in NEWRUN */
#define LWIDTH 46
#define RWIDTH 30
#define BWIDTH 78 /* 38 without moned */
#endif
#define TWIDTH 36
#define LWIDTH 42
#define RWIDTH 30
#define BWIDTH 40

#include <ikdefs.h>
#include <stdio.h>
main () {
     pixel_t dot;
     pixel_t line[512];
     register int x, y;

setorig (TORIG);
dot.r = 255;
dot.g = dot.b = 255;
for (x=0; x<512; x++) {
     line[x] = dot;
     }
for (y=0; y<TWIDTH; y++) {   /* top edge */
               ikpwrite(line,512*4,0,y);
     }
for (y=TWIDTH; y<512-BWIDTH; y++) {
               ikpwrite(line,LWIDTH*4,0,y);      /* left edge */
               ikpwrite(line,RWIDTH*4,512-RWIDTH,y);    /* right edge */
     }
for (y=512-BWIDTH; y<512; y++) {     /* bottom edge */
               ikpwrite(line,512*4,0,y);
     }
}
```

```
/* mcsi.k */

default nanop ccnop ldnop sb aluz yd car0 ss0 alubr mdrikd marika

org 0

;    mcsi - microprogram command string interpreter

;    this routine manages microprogram execution and parameter
;    transferal.  the routine uses the scratchpad memory (loc
;    202,0-202,1777) as a command/parameter list.  register 0
;    is used as the command list pointer, and should not be destroyed
;    by user routines.  the command pointer points to the location
;    in the scratch pad containing the address of the
;    next microprogram to execute.  the scratchpad locations follow-
;    ing this are the parameters for the given routine.  each
;    microprogram is responsible for updating the command pointer
;    for the next routine (when necessary).  the last routine
;    to execute should be either 'nulproc', which loops on the
;    current command address (its own), or 'mcsi' (resulting in
;    continuous passes through the command list).
;         to break out of 'nulproc', the host may either change
;    the current address, or pulse the mps16 reset line
;    (resulting in another pass through the command list).

     global mcsi, nxtcom, nulproc, NXTCOM

mcsi:        ldudr   0202                          ; scratchpad high address
     rimm 0 pr bd b0                         ; set r0 (command pointer)

;    nulproc - null process; if next command address is this,
;                  routine will loop here until interrupted
nulproc:
NXTCOM:
nxtcom:      b0 ps alumar                          ; get next command

go: ikrd
     jmpib

end
```

```
/* waitsig.k */

; Wait for location addressed by parameter 1 to equal parameter 2.
;
; Gary Bishop 01/26/81.
;
; Called by MCSI

org 3000

default nanop ccnop ldnop sb aluz yd car0 ss0 alubr mdrikd marika

NXTCOM = 2                                  ; MCSI next routine address

wait:       ldudr 0202
     incrs b0 alumar                        ; get flag address
     ikrd ikbr b1 bd                        ;  into reg 1
     incrs b0 alumar                        ; get flag value
     ikrd ikbr b2 bd                        ;  into reg 2
     incrs b0                               ; inc b0 to point at next routine

     pr ra1 alumar                          ; setup mar to read flag

loop:       ikrd ikbr b3 bd                 ; read flag into reg 3
     reos ra2 b3                            ; compare flag with value given
     ncczero jmpdf loop                     ; loop until flag is equal

     jmpdf NXTCOM                           ; Return to MCSI


; Signal by writing value in parameter 2 into address given in parameter 1
;
; Gary Bishop 01/26/81
;
; Called by MCSI

org 3100

signal:     ldudr 0202
     incrs b0 alumar                        ; Get flag address
     ikrd ikbr b1 bd                        ;  into reg 1
     incrs b0 alumar
     ikrd ikmdr                             ; Get flag value into MDR
     incrs b0                               ; inc reg 0 to point to next routine
     pr ra1 alumar                          ; Put flag value into flag
     ikwr

     jmpdf NXTCOM                           ; Return to MCSI
end
```

```
/* makefile */

CFLAGS = -O

dots: dots.o
	cc dots.o -lk -o dots

clean:
	/bin/rm -f *.o a.out core mon.out
```

```
/* dots.c */

#include <ikdefs.h>
#include <stdio.h>
#define DSPACING 8   /* no. pixels between dots */
#define NUMDOTS 64   /* no. dots per line */
main () {
     pixel_t zilch, dot;
     pixel_t blank[NUMDOTS*DSPACING], grid[NUMDOTS*DSPACING];
     register int x, y;

zilch.r = zilch.g = zilch.b = 0;
dot.r = dot.g = dot.b = 255;
for (x=0; x<512; x++) {
     blank[x] = zilch;
     if (x%DSPACING == 0) {
               grid[x] = dot;
               }
     else {
               grid[x] = zilch;
               }
     }
for (y=0; y<512; y++) {
     if (y%DSPACING == 0) {
               ikpwrite(grid,NUMDOTS*DSPACING*4,0,y);
               }
     else {
               ikpwrite(blank,NUMDOTS*DSPACING*4,0,y);
               }
     }
}
```

```
/* calibrate */

/unc/dh/bin/fixik
/usr/ikonas/moned << EOF > /dev/null
4 454 1014
5 0 62
s
q
EOF
# /usr/ikonas/color
/unc/dh/video/dots
/usr/ikonas/wmask f0
/usr/lib/iktest/imed << EOG > /dev/null
4 4 0
# 5 62 20
2 200 200
1 1 2
0 33 0
s
q
EOG
/usr/ikonas/color r
# /usr/ikonas/cmap < /unc/dh/video/cmapc
# /usr/ikonas/xbar < /unc/dh/video/xbarc
# *** the following is inserted just to use left wall-mount camera ***
/unc/gb/bin/ldvm.tst << EOH > /dev/null
v0
q
EOH
```

```
/* display */

/usr/ikonas/xbar < xbard
/usr/ikonas/imed << EOF > /dev/null
4 0 0
s
q
EOF
/usr/ikonas/wmask ffffffff
/usr/ikonas/cmapsetup | /usr/lib/iktest/cmap
/usr/ikonas/black
/unc/dh/surf4/ikrdwr2 1
/usr/ikonas/wmask f
/usr/ikonas/imed << EOG > /dev/null
0 33 0
1 1 2
2 200 200
4 4 0
5 62 20
s
q
EOG
/unc/dh/bmp/poker
/unc/dh/bmp/il /unc/dh/bmp/srch.obj
/unc/dh/bmp/go
/unc/dh/cmap6/nflite2
```

```
/* demo1 */

/unc/dh/cmap8/NEWRUN
```

```
/* NEWRUN */

/usr/ikonas/ikreset
/usr/ikonas/moned << EOF > /dev/null
4 454 1014
5 0 62
s
q
EOF
# /usr/ikonas/ikset 30hz > /dev/null
# /usr/ikonas/color b
/unc/dh/surf4/ikrdwr 3
/unc/dh/cmap8/frame
/unc/dh/cmap8/blackout
/usr/ikonas/wmask f0
/usr/lib/iktest/imed << EOG > /dev/null
0 0 0
1 7 16
2 0 0
4 4 0
# 5 62 20
s
q
EOG
# *** the following is included just to use left wall-mount camera ***
/unc/gb/bin/ldvm.tst << EOH > /dev/null
v0
q
EOH
/unc/dh/cmap8/load3a
/usr/ikonas/ikgo
# sdb /unc/dh/cmap8/nflite3a
/unc/dh/cmap8/nflite3a
```

```
/* demo2a */

/unc/dh/surfgp/wellbench << EOF
20
8
100 150 150
0 0 50
.5
0 0 100
5
0
EOF
```

```
/* demo2b */

/unc/dh/surfgp/LIGHT
```

```
/* LIGHT */

/usr/ikonas/moned << EOF > /dev/null
4 454 1014
5 0 62
s
q
EOF
# /usr/ikonas/ikset 30hz > /dev/null
# /usr/ikonas/color b
/unc/dh/cmap8/frame
/unc/dh/cmap8/blackout
/usr/ikonas/wmask f0
/usr/lib/iktest/imed << EOG > /dev/null
0 0 0
1 7 16
2 0 0
4 4 0
# 5 62 20
s
q
EOG
# initialize lowest quadrant of physical Ikonas color map to 0's - won't change
/usr/ikonas/ikset "cmap[0-255] = 0"
# set up xbar; 0-7 in frame buffer represent normal; 8-9 select a quadrant in
#  the red color map; 10-11 for green; 12-13 for blue
/usr/ikonas/ikset << EOI > /dev/null
xbar[0-9] = 0 1 2 3 4 5 6 7 8 9
xbar[10-19] = 0 1 2 3 4 5 6 7 10 11
xbar[20-29] = 0 1 2 3 4 5 6 7 12 13
q
EOI
# *** the following is included just to use left wall-mount camera ***
/unc/gb/bin/ldvm.tst << EOH > /dev/null
v0
q
EOH
/unc/dh/cmap8/load3a
/usr/ikonas/ikgo
# sdb /unc/dh/cmap8/nflite3a
/unc/dh/new.work/nflite3a
```

```
/* makefile */

CFLAGS = -O

shade1: cshade.o zbputs1.o zbik1.o proto.o
	cc  cshade.o zbputs1.o zbik1.o proto.o -li -o shade1

shade2: cshade.o zbputs2.o zbik2.o proto.o
	cc  cshade.o zbputs2.o zbik2.o proto.o -lk -o shade2

shade3: cshade.o zbputs3.o zbik3.o proto.o
	cc  cshade.o zbputs3.o zbik3.o proto.o -lk -o shade3

ikrdwr: ikrdwr.o
	cc  ikrdwr.o -lk -o ikrdwr

ikrdwr2: ikrdwr.o
	cc  ikrdwr.o -li -o ikrdwr2

zbik1.o zbputs1.o    : /unc/dh/cmap4/color1.h

zbik2.o zbputs2.o    : /unc/dh/cmap5/color2.h

zbik3.o zbputs3.o    : /unc/dh/cmap8/color3.h

proto.o   : prodesc.h

prodesc.h   : protoargs.h

cshade.o zbputs1.o zbputs2.o zbputs3.o : sphere.h

clean:
	/bin/rm -f *.o a.out core mon.out
```

```
/* cshade.c */

#define DEBUG if(0)

#define ZMAX 255

#include <stdio.h>
#include "sphere.h"
main (argc, argv)
int argc;
char *argv[];
{
int i;
struct sphere spherein;
static int snum; /* sphere number */
register int xlim, ylim;
register struct sphere *sp;
double origin[6];  /* x, y, z z-buffer origin, scale (input parameters ) */
#define scale origin[4]
#define shear origin[5]
double atof();

sp = &spherein;
for (i=1; i<=3; i++) origin[i] = 0.0;
scale= 1.0;
shear= 0.0;
for( i=1; i<argc; i++)
     origin[i] = atof( argv[i]);  /* set origin from argument: default zero */

printf("Scale %.8f, shear %.8f\n", scale, shear);


/* read prototype angle and depth file */
pinit();
mess("Pinit done\n");
/* clear depth buffer to background 'color' */
zbinit();
mess("zbinit done\n");

/* ask size of z-buffer */
xlim = xzbsize();
ylim = yzbsize();

/* read spheres until end of file */
snum= 0;
while( scanf("%d%d%d%d%d", &sp->spx, &sp->spy, &sp->spz, &sp->spr,
     &sp->spcolor) != EOF ) {
     snum++;
DEBUG printf("SPHERE %4d %4d %4d %4d %4d (original) \n",
     sp->spx, sp->spy, sp->spz, sp->spr, sp->spcolor);
     sp->spx = ( (double)sp->spx - origin[1]) * scale;
     sp->spy = ( (double)sp->spy - origin[2]) * scale;
     sp->spz = ( (double)sp->spz - origin[3]) * scale;
     sp->spr = (double) sp->spr * scale;
```

```
/* cshade.c */

          if((sp->spx >= - sp->spr) && ((sp->spx - sp->spr) <=xlim) &&
              (sp->spy >= - sp->spr) && ((sp->spy - sp->spr) <=ylim) &&
         (sp->spz >= 0)            && ((sp->spz + sp->spr) <=ZMAX )  ) {
         /* apply stereo shear and draw it : */
         sp->spx += (sp->spz -128)*shear;
         zbputs( sp);
                }
DEBUG printf("SPHERE %4d %4d %4d %4d (shifted) \n",
     sp->spx, sp->spy, sp->spz, sp->spr);
}

/* synchronize z buffer ( tell it we want to read it back ) */
mess("Shade calling zbsync\n");
zbsync();

/* write z buffer to standard output. */
 /*
mess("Shade calling zbdump\n");
zbdump(stdout);
 */
exit(0);
} /* end main */
mess(s)
char *s;
{
 DEBUG
 while(*s) putc( *s++, stderr);
}
```

```
/* zbputs3.c */

    /* zbputs.c —— z-buffer put sphere. Floating point version. M Pique. */
#define DEBUG if(0) printf
#define NEWCLR 1
#define RTHRESH 50
#define GTHRESH 50
#define BTHRESH 50
#include "sphere.h"
#include "ikdefs.h" /* definition of color passed to coverzb */
#include "/unc/dh/cmap8/color3.h"
zbputs (ps)
register struct sphere *ps;
{
register int xoffset, yoffset;
int protorad;
register int xproto;
register int yproto;
register double ratio;    /* prototype sphere radius / argument sphere radius */
int news, newz;
#ifdef NEWCLR
int redi, greeni, bluei;  /* working vars for unpacking from decimal */
#endif
pixel_t color;
register int pbx; /* prototype x-bound for this y-value */

if(ps->spr <= 0) ps->spr = 1;

protorad = pradius();

ratio =  (double)(protorad ) / (double)(ps->spr);

#ifdef DBG
printf("Sphere %4d %4d %4d %4d ratio %4f\n"
    , ps->spx, ps->spy, ps->spz, (ps->spr), ratio);
#endif


#ifdef NEWCLR
/* set color fractions for this sphere */
redi= ps->spcolor/10000;
greeni= (ps->spcolor/100) - redi*100;  /* middle two digits */
bluei= (ps->spcolor - greeni*100 - redi*10000);
#endif

/* Fill in frame buffer circle by sampling into prototype : */

for( yoffset = -ps->spr; yoffset < ps->spr; yoffset++){
    yproto = (int) (ratio * yoffset);
    pbx = pbound(yproto);
    for( xoffset = -ps->spr; xoffset < ps->spr; xoffset++) {
            xproto = (int)(ratio * xoffset);
            if( xproto > pbx - 1 ) break;    /* do not process the 0 */
            if (xproto >= -pbx) {    /* elements of the prototype array */
```

```
/* zbputs3.c */

#ifdef DBG
                        printf("   xo=%3d yo=%3d", xoffset, yoffset); /* debug */
#endif
                        news = protos( xproto, yproto);
                        DEBUG("news %d\n", news);
                        /* build color from news */
#ifdef OLDCLR
                color.r = ((REDON==1) ? (unsigned char)(news) : (RDEFAULT));
                color.g = ((GREENON==1) ? (unsigned char)(news) : (GDEFAULT));
                color.b = ((BLUEON==1) ? (unsigned char)(news) : (BDEFAULT));
#endif
#ifdef NEWCLR
                color.r = ((redi >= RTHRESH) ? (unsigned char)(news) : (ZERO));
                color.g = ((greeni >= GTHRESH) ? (unsigned char)(news) : (ZERO));
                color.b = ((bluei >= BTHRESH) ? (unsigned char)(news) : (ZERO));
#endif

                        newz = ps->spz + (int)(protoz(xproto,yproto)/ratio);
                        /* select z-buffer blend for the outer edge */
                        /*      blendzb( (xproto>= pbx) );        */

                coverzb( ps->spx + xoffset, ps->spy + yoffset, newz, &color);

                        }
                }
    } /* end yoffset loop */

} /* end zbputs */
```

```
/* zbik3.c */

 /* ZBIK - Z-buffer management:
    in-core version with intens in core,
    but written to Ikonas from time to time.
  */
#define XZBSIZE 512
#define YZBSIZE 512
#define ZMAX 255
#include <ikdefs.h>
#include <stdio.h>
#define I(C) ( 0377 & (C))  /* integer from character, unsigned */
#define SQR(x) ((x)*(x))  /* square of x */
#define OK(x,y) ((x>=0 && x<XZBSIZE) && (y>=0 && y<YZBSIZE))
int blendsw =0; /* blend new intensity into old if ==1 */
static pixel_t *fb; /* pointer to beginning of in-core frame buffer */
#define fbaddr(x,y) (fb+((x)+(YZBSIZE-1-(y))*XZBSIZE))
#define z fill
#define FBSIZE (XZBSIZE*YZBSIZE*sizeof(*fb))
static updtecount; static char rowused[YZBSIZE];
#define IKBATCH 25000  /* pixels changed between ikonas updates */
#include "/unc/dh/cmap8/color3.h"
char *malloc();

zbinit()
{
register int x,y;
register pixel_t *pixel;
    fb= (pixel_t *) malloc( FBSIZE);
    if(fb==NULL){fprintf(stderr,"zbinit: not enough core");exit(-1);}
    for (y=0; y<YZBSIZE; y++) {
            for(x=0; x< XZBSIZE; x++) {
                    pixel= fbaddr(x,y);
                    pixel->r = ((REDON==1) ? (PTBACKGR) : (RDEFAULT));
                    pixel->g = ((GREENON==1) ? (PTBACKGR) : (GDEFAULT));
                    pixel->b = ((BLUEON==1) ? (PTBACKGR) : (BDEFAULT));
                    pixel->z = 0;  /* depth (0=far) */
                    }
            }
    zbsync();
}

putzb( x, y, z, s)
register int x,y;
char z, s;
{
    if( OK(x,y) ){
            fbaddr(x,y)->z = z;
            fbaddr(x,y)->r = s;
            }
    }

coverzb(x, y, z, colorp)
int x,y;
```

```
/* zbik3.c */


char z;
pixel_t *colorp;
{
static int oldy;
register int zval, oldzval;
register pixel_t *pixel;
register float zfract; /* fraction 1.0 to 0.0 normalized distance from rear*/
pixel_t new;

     if(! OK(x,y)) return;

     pixel= fbaddr(x,y);
     oldzval= I( pixel->z );
     zval = I(z);


     if ( zval > oldzval) {   /* closer          */

               new = *colorp;

#ifdef MPBLEND
               /* color depth cueing: */
               zfract= (float) zval / (float) ZMAX;
               new.r = (int) ( (float) I(new.r) * zfract);
               new.g = (int) ( (float) (I(new.g)) * zfract);

               if( blendsw ) oldzval = zval;

               switch( zval - oldzval ) {

               case  0:
                         pixel->g = ( new.g + I(pixel->g) ) / 2;
                         pixel->b = ( new.b + I(pixel->b) ) / 2;
                         pixel->r = ( new.r + I(pixel->r) ) / 2;
                         pixel->z =  zval;
                         break;
               case -2:
                /* new is somewhat farther */
                         pixel->r = ( new.r + 7 * I(pixel->r) ) / 8;
                         pixel->g = ( new.g + 7 * I(pixel->g) ) / 8;
                         pixel->b = ( new.b + 7 * I(pixel->b) ) / 8;
                         pixel->z = oldzval;
                         break;
               case  2:
                /* new is somewhat closer */
                         pixel->r = ( 7 * new.r + I(pixel->r) ) / 8;
                         pixel->g = ( 7 * new.g + I(pixel->g) ) / 8;
                         pixel->b = ( 7 * new.b + I(pixel->b) ) / 8;
                         pixel->z = zval;
                         break;
               case -1:
                /* new is slightly farther */
                         pixel->r = ( new.r + 3 * I(pixel->r) ) / 4;
                         pixel->g = ( new.g + 3 * I(pixel->g) ) / 4;
```

```
                        pixel->b = ( new.b + 3 * I(pixel->b) ) / 4;
                        pixel->z = oldzval;
                        break;
                case  1:
                        pixel->r = ( 3 * new.r + I(pixel->r) ) / 4;
                        pixel->g = ( 3 * new.g + I(pixel->g) ) / 4;
                        pixel->b = ( 3 * new.b + I(pixel->b) ) / 4;
                        pixel->z = zval;
                        break;
                default:
#endif
                        pixel->r = new.r;
                        pixel->g = new.g;
                        pixel->b = new.b;
/*###121 [cc] member of structure or union required %%%*/
/*###121 [cc] warning: illegal combination of pointer and integer, op = %%%*/
                        pixel->z = zval;

                rowused[y] = 1;

                if( ++updtecount > IKBATCH ) {
                        int firstyrow;
                        y=YZBSIZE-1;

                        while(y>=0){
                                /* Skip unused (unchanged) rows */
                                while( y>=0 && ! rowused[y]) y--;

                                /* Search over used rows for next unused */
                                firstyrow=y;
                                while(y>=0 && rowused[y] && firstyrow-y<8)
                                        {rowused[y--]= 0; }
#ifdef DEBUG
                                fprintf(stderr,"wr rows %d to %d %d bytes\n",
                                        firstyrow, y+1,
                                        (firstyrow-y)*sizeof(pixel_t)*XZBSIZE);
#endif
                                /* Write out the used block */
                                ikpwrite( fbaddr(0, firstyrow),
                                 (firstyrow-y)*sizeof(pixel_t)*XZBSIZE,
                                  0, firstyrow);
                                }
                        updtecount = 0;
                        }

        }
} /* end coverzb */
getzb(x, y, zp, sp)
register int x,y;
char *zp, *sp;
{
        if( OK(x,y) ){
                *zp = fbaddr(x,y)->z;
                *sp = fbaddr(x,y)->r;
```

```
/* zbik3.c */

        }
}   /* end getzb */

xzbsize ()
{
        return( XZBSIZE );
}   /* end xzbsize */

yzbsize ()
{
        return( YZBSIZE );
}   /* end yzbsize */
blendzb(sw)
int sw;
{
/*   blendsw= sw;       set blend switch on or off... see coverzb */
}
zbsync() {
register int y;
        fprintf(stderr,"sync\n");
        for(y=YZBSIZE-1; y>=0; y--) {
                        updterow( y);
                        }
}

 static
updterow(y)
register int y;
{
        ikpwrite( fbaddr(0,y), XZBSIZE*sizeof(pixel_t), 0, y);
        }

zbdump(ofile)
/**
FILE *ofile;
 **/
{
} /* end zbdump */
```

```c
/* proto.c */

#define DEBUG if(0) printf
#define DBG if(0) printf
#include <stdio.h>
#include "prodesc.h"
#define I(C) ( 0377 & (C))  /* integer from character, unsigned */
#define abs(a) ((a)<0? -(a):(a))
int prad;
prad = PSIZE/2;

pinit() {
FILE *pfile, *fopen();
int pblack, pzmax,i, j, temp;
int psize;
/* read prototype angle and depth file */
/* format is header line: size, blackness, max depth, followed by */
/* 'size' repetitions of:  (
       lowbound, highbound of data,
              ( PSIZE (angle), PSIZE(depth)) */

DEBUG("pinit started\n");
if( NULL == (pfile = fopen( PNAME, "r")) )
  mess( "Prototype file open err.\n");
fscanf( pfile, "%d%d%d", &psize, &pblack, &pzmax);
if( psize != PSIZE) {printf("Prototype file size error.\n"); return;}
DEBUG("starting to read...\n");
for (i=0;i<PSIZE; i++){
     fscanf( pfile, "%d %d", &pbounds[i].low, &pbounds[i].high);
     for (j=0; j<PSIZE; j++){
               fscanf( pfile, "%d", &temp);  proto[i][j].ps = temp; }
     for (j=0; j<PSIZE; j++){
               fscanf( pfile, "%d", &temp);  proto[i][j].pz = temp; }
     }
DEBUG("angle and depth prototypes read\n");
fclose(pfile);
} /* end pinit */
pbound (y)
register int y;
{

y = y + prad;
if ( PYOK(y) ) return ( pbounds[y].high - prad );
else return ( -1 );
} /* end pbound */

psize () {
return ( PSIZE );
}  /* end psize */

pradius () {
return ( prad );
}  /* end pradius */
```

```
/* proto.c */

protoz ( x, y)
register int x, y;
{

x = x+(prad);
y = y+(prad);
if( PXOK(x) && PYOK(y) ) return ( I(proto[y][x].pz ));
else return ( 0 );
}

protos ( x, y)
register int x, y;
{

x = x+(prad);
y = y+(prad);
DBG("protos:x %d; y %d; prad %d; ps %d\n", x, y, prad, proto[y][x].ps);
DBG("ps++ %d\n", proto[y+1][x+1].ps);
if( PXOK(x) && PYOK(y) ) return ( I(proto[y][x].ps ));
else return ( 0 );
} /* end protos */
```

```
/* sphere.h */

/* sphere.h  Version of 30 Oct 80 */
struct sphere{
     int spx, spy, spz, spr, spcolor;
     };
```

```
/* ikrdwr.c */

/* IKRDWR - Ikonas frame buffer management:
     read or write
   */
#define XFBSIZE 512
#define YFBSIZE 512
#include <ikdefs.h>
#include <stdio.h>
static pixel_t *fb; /* pointer to beginning of in-core frame buffer */
#define fbaddr(x,y) (fb+((x)+(YFBSIZE-1-(y))*XFBSIZE))
#define FBSIZE (XFBSIZE*YFBSIZE*sizeof(*fb))
#define PNAMED "/unc/dh/surf4/pictfile"
#define PNAME1 "/unc/dh/surf4/pictfil1"
#define PNAME2 "/unc/dh/surf4/pictfil2"
#define PNAME3 "/unc/dh/surf4/pictfil3"
char *malloc();
FILE *pfile, *fopen();

main (argc, argv)
     int argc;
     char *argv[];
{
int arg;

arg = atoi(argv[1]);
fb = (pixel_t *) malloc (FBSIZE);
if (fb==NULL) {fprintf(stderr,"fbinit: not enough core.\n"); exit(-1);}
if (arg >= 1) wrsync(arg);
else rdsync();
}


rdsync() {
register int y;

     if (NULL == (pfile = fopen(PNAMED, "w")))
                         fprintf(stderr, "pictfile open error.\n");
     setbuf(pfile, malloc(BUFSIZ));
     fprintf(stderr,"Reading from frame buffer . . .\n");
     for(y=YFBSIZE-1; y>=0; y--) {
               ikpread( fbaddr(0,y), XFBSIZE*sizeof(* fb), 0, y);
/*
               fwrite( fbaddr(0,y), XFBSIZE*sizeof(* fb), 1, pfile);
*/
               }
     fprintf(stderr,"Writing to disk . . .\n");
     fwrite (fb, XFBSIZE*sizeof(* fb), YFBSIZE, pfile);
}

wrsync(fnum)
     int fnum;
{
register int y;
```

/* ikrdwr.c */

```
if (fnum == 1) {
     if (NULL == (pfile = fopen(PNAME1, "r")))
                         fprintf(stderr, "pictfile open error.\n");
     }
else { if (fnum == 2) {
     if (NULL == (pfile = fopen(PNAME2, "r")))
                         fprintf(stderr, "pictfile open error.\n");
     }
else { if (fnum == 3) {
     if (NULL == (pfile = fopen(PNAME3, "r")))
                         fprintf(stderr, "pictfile open error.\n");
     }
else {fprintf(stderr,"Wrong filenum: %d\n", fnum); exit(-1);} } }
     setbuf(pfile, malloc(BUFSIZ));
     fprintf(stderr,"Reading from disk . . .\n");
     fread (fb, XFBSIZE*sizeof(* fb), YFBSIZE, pfile);
#ifdef SWAPRB
     /* exchange red & blue pixel values */
     swaprb(fb, XFBSIZE, YFBSIZE);
#endif
     fprintf(stderr,"Writing to frame buffer . . .\n");
     for(y=YFBSIZE-1; y>=0; y--) {
               ikpwrite( fbaddr(0,y), XFBSIZE*sizeof(* fb), 0, y);
               }
}

#ifdef SWAPRB
/* exchange red & blue pixel values */
swaprb (fb, xsize, ysize)
     pixel_t * fb;
     int       xsize, ysize;
{
register pixel_t *p;
pixel_t tpix;

for (p=fb; p < fb+(xsize*ysize); p++) {
     tpix.r = p->r;
     p->r = p->b;
     p->b = tpix.r;
     }
}
#endif
```

```
/* makefile */

CFLAGS = -O

3cubes: 3cubes.o clip.o gp.o gpak.o hidden.o init.o movie.o normltbl.o \
		shade.o shapes.o
	cc 3cubes.o clip.o gp.o gpak.o hidden.o init.o movie.o normltbl.o \
			shade.o shapes.o -lk -lm ${CFLAGS} -o 3cubes

boxes: boxes.o clip.o gp.o gpak.o hidden.o init.o movie.o normltbl.o \
		shade.o shapes.o
	cc boxes.o clip.o gp.o gpak.o hidden.o init.o movie.o normltbl.o \
			shade.o shapes.o -lk -lm ${CFLAGS} -o boxes

cylinder: cylinder.o clip.o gp.o gpak.o hidden.o init.o movie.o normltbl.o \
		shade.o shapes.o
	cc cylinder.o clip.o gp.o gpak.o hidden.o init.o movie.o normltbl.o \
			shade.o shapes.o -lk -lm ${CFLAGS} -o cylinder

logo: logo.o clip.o gp.o gpak.o hidden.o init.o movie.o normltbl.o \
		shade.o shapes.o
	cc logo.o clip.o gp.o gpak.o hidden.o init.o movie.o normltbl.o \
			shade.o shapes.o -lk -lm ${CFLAGS} -o logo

sphere: sphere.o clip.o gp.o gpak.o hidden.o init.o movie.o normltbl.o \
		shade.o shapes.o
	cc sphere.o clip.o gp.o gpak.o hidden.o init.o movie.o normltbl.o \
			shade.o shapes.o -lk -lm ${CFLAGS} -o sphere

wellbench: wellbench.o clip.o gp.o gpak.o hidden.o init.o movie.o normltbl.o \
		shade.o shapes.o
	cc wellbench.o clip.o gp.o gpak.o hidden.o init.o movie.o normltbl.o \
			shade.o shapes.o -lk -lm ${CFLAGS} -o wellbench

oldcylinder: cylinder.o /unc/eric/lib/libg.a
	cc cylinder.o /unc/eric/lib/libg.a -lk -lm ${CFLAGS} -o cylinder

normltbl.o : normal.h

clip.o gp.o gpak.o hidden.o init.o movie.o shade.o shapes.o cylinder.o : gp.h

clean:
	/bin/rm -f *.o a.out core mon.out
```

```
/* cylinder.c */

/*
#include <gp.h>
*/
#include "gp.h"
#include <math.h>

main()
{
int r,g,b,nsides;
char s[132];
init();
translate(0.0,0.0,-5.0);
rotate(XAXIS,4*PI/7);
r = 255; g = b = 0;
printf("Enter color [%d %d %d]: ",r,g,b);
if (gets(s)&&s[0])
    sscanf(s,"%d %d %d",&r,&g,&b);
set_color(r,g,b);
nsides = 10;
printf("Enter nsides [%d]: ",nsides);
if (gets(s)&&s[0])
    sscanf(s,"%d",&nsides);
cylinder(3.0,10.0,nsides);
movie();
}
```

```
/* wellbench.c */

#include <math.h>
#include <stdio.h>
/*
#include "/unc/eric/include/gp.h"
*/
#include "gp.h"

/* Draws a picture of the "Old Well" on the UNC campus. */
/* Written by Eric Grant 9-6-81. */

#define COLUMNHEIGHT 70.0
#define BASEHEIGHT 6.0
#define TILES 20
#define BASERADIUS 45.0

#define DEGCON (3.141592654/180.0)


/* Draws a column with specified number of sides.  Also includes
   the square ends on the columns. */
column(nsides)
int nsides;
{
set_color(255,255,255);
push_m();
push_m();
translate(-3.0,-3.0,0.0);
rect(6.0,6.0,1.0);
pop_m();
translate(0.0,0.0,1.0);
cylinder(3.0,COLUMNHEIGHT-2.0,nsides);
translate(0.0,0.0,COLUMNHEIGHT-2.0);
push_m();
translate(-3.0,-3.0,0.0);
rect(6.0,6.0,1.0);
pop_m();
pop_m();
}

/* Draws the eight columns equally spaced apart in a circle. */
columns(nsides)
int nsides;
{
int angle;
float x,y,theta;

push_m();
translate(0.0,0.0,BASEHEIGHT);
for (angle=0; angle<360; angle += 45)
    {
    theta = (float)angle * DEGCON;
    x = (BASERADIUS-15.0)*sin(theta);
```

```
/* wellbench.c */

      y = (BASERADIUS-15.0)*cos(theta);
      push_m();
      translate(x,y,0.0);
      column(nsides);
      pop_m();
      }
pop_m();
}

/* Draws the two cylinders which make up the "base" or steps. */
well_base(nsides)
int nsides;
{
set_color(200,200,200);
push_m();
cylinder(BASERADIUS,BASEHEIGHT/2.0,nsides);
translate(0.0,0.0,BASEHEIGHT/2.0);
cylinder(BASERADIUS-5.0,BASEHEIGHT/2.0,nsides);
pop_m();
}

/* Draws the drinking fountain in the center of the structure. */
fountain()
{
set_color(175,175,175);
push_m();
translate(-9.0,-9.0,BASEHEIGHT);
rotate(ZAXIS,PI/2.0);
rect(18.0,18.0,2.0);
pop_m();
set_color(200,200,200);
push_m();
translate(-6.0,-6.0,BASEHEIGHT+2.0);
rect(12.0,12.0,20.0);
pop_m();
}

/* Draws the roof.  The roof consists of a couple of white
   cylinders, and then the actual blue roof "tiling" itself. */
roof(nsides)
int nsides;
{
float theta,theta2,phi,phi2,radius;
float incr,x1,y1,x2,y2,x3,y3,x4,y4,z1,z2;
int i,j;

set_color(255,255,255);
push_m();
translate(0.0,0.0,BASEHEIGHT+COLUMNHEIGHT);
cylinder(BASERADIUS-10.0,2.0,nsides);
translate(0.0,0.0,2.0);
cylinder(BASERADIUS-8.0,2.0,nsides);
pop_m();
set_color(0,200,255);
```

```
/* wellbench.c */

 push_m();
 translate(0.0,0.0,BASEHEIGHT+COLUMNHEIGHT-8.0);
 incr = 2.0*PI/TILES;
 phi = 0.0;
 radius = BASERADIUS-6.0;
 for (i=0; i<TILES/4-1; i++)
      {
      phi2 = phi+incr;
      z1 = cos(phi)*radius;
      z2 = cos(phi2)*radius;
      theta = 0.0;
      for (j=0; j<TILES; j++)
      {
           theta2 = theta+incr;
           x1 = (float) (cos(theta)*sin(phi)*radius);
           y1 = (float) (sin(theta)*sin(phi)*radius);
           x2 = (float) (cos(theta)*sin(phi2)*radius);
           y2 = (float) (sin(theta)*sin(phi2)*radius);
           x3 = (float) (cos(theta2)*sin(phi)*radius);
           y3 = (float) (sin(theta2)*sin(phi)*radius);
           x4 = (float) (cos(theta2)*sin(phi2)*radius);
           y4 = (float) (sin(theta2)*sin(phi2)*radius);
           theta = theta2;
      plate(x3,y3,z1,x4,y4,z2,x2,y2,z2,x1,y1,z1);
           }
      phi = phi2;
      }
pop_m();
}

well()
{
int bsides,csides;
char s[80];


bsides = 20;
fprintf(stderr,"Enter number of sides for base [%d]: ",bsides);
if (gets(s)&&s[0])sscanf(s,"%d",&bsides);
well_base(bsides);
csides = 8;
fprintf(stderr,"Enter number of sides for columns [%d]: ",csides);
if (gets(s)&&s[0])sscanf(s,"%d",&csides);
roof(20);
columns(csides);
fountain();
}

bench()
{
set_color(255,200,200);
push_m();
translate(-2.0,-4.0,0.0);
scale(2.5,5.0,2.5);
```

/* wellbench.c */

```
push_m();
translate(0.0,0.0,4.0);
rect(4.0,8.0,1.0);
pop_m();
push_m();
translate(.5,0.5,0.0);
rect(3.0,0.5,5.0);
pop_m();
push_m();
translate(.5,7.0,0.0);
rect(3.0,0.5,5.0);
pop_m();
pop_m();
}

main()
{
init();
well();
push_m();
translate(75.0,0.0,0.0);
bench();
pop_m();
push_m();
translate(75.0,0.0,0.0);
rotate(ZAXIS,PI/2.0);
bench();
pop_m();
push_m();
translate(75.0,0.0,0.0);
rotate(ZAXIS,PI);
bench();
pop_m();
movie();
dump_data();
}
```

```
/* gp.c */

#include <stdio.h>
/*
#include <gp.h>
*/
#include "gp.h"
#include <math.h>

#define EPS .0001

/*

                                    Graphics Package
                                  Written by Eric Grant

This package is based on a graphics package written by Bob Sproull for 15-462.
Some of the documentation contained in this file is taken from class handouts.

This package provides functions for 3-dimensional viewing and modeling
transformations. It has built in functions for vector drawing ('line_to' and
'move_to'). Additionally, the caller may transform individual points by
calling 'vec_mul'. Thus this package may be used to transform faces and, in
conjunction with a shading package, may be used to produce three dimensional
images on a raster display device.

The package uses a matrix 'm', which may be used to transform coordinates from
one coordinate system to another. Additionally, this matrix may perform the
perspective transformation.

For vector applications, 'm' will generally be set up to achieve both modeling
and viewing transformations; thus m = MV, where M is the modeling
transformation and V is the viewing transformation. It is up to the client of
the package to arrange that this concatenation is performed.

For other applications, the client may wish to have an intermediate step in the
transformation. It may be necessary to first transform points to world
coordinates, and then perform the transformation to screen coordinates. A
number of routines have been provided to facilitate these transformations.

The modeling transformation M is generally composed by concatenating primitive
transformations that describe the instance transformation. This is the
instance transformation that transforms points to world coordinates. For more
information see Newman and Sproull, sections 9-4 and 10-1.

The matrix V is usually the concatenation of two parts: W, the transformation
from world to eye or camera coordinates, and P, the perspective transformation
(see Newman&Sproull eq. 23-4). The world-to-eye transformation is usually also
a concatenation of several primitive transformations; usually the last such
transformation forms the left-handed eye coordinate system (Newman&Sproull eq.
22-12). If you want to create a non-square viewport for viewing, this can be
done by inserting a scaling transformation just before P that has the inverse
effect of the viewport scaling. For example, suppose the height (y) of the
```

```
viewport is twice the width (x). In addition to setting the viewport
parameters appropriately in the geometry pipeline, we insert before P a matrix
that scales all x values by 2, while leaving y and z values untouched.

*/
static float
     vxl,     /* Left edge of viewport */
     vxr,     /* Right edge of viewport */
     vyb,     /* Bottom edge of viewport */
     vyt,     /* Top edge of viewport */
     vsx,     /* Viewport x-scaling factor */
     vcx,     /* Viewport x center coordinate */
     vsy,     /* Viewport y-scaling factor */
     vcy;     /* Viewport y center coordinate */
static float
     cx,      /* Current position - untransformed */
     cy,
     cz,
     cw;
static float
     cxo,     /* Current position - in screen coordinate system */
     cyo,
     czo;
static s_transform
     *m_stack,          /* Pointer to top of stack */
     *m_free;           /* Pointer to beginning of free list */
static transform
     q,
     m,
     camera,            /* World to screen transformation */
     preper;
static boolean
     q_identity;        /* True if q is identity matrix */

static inquiry_response
     v;                 /* Display characteristics */


/*
This package may contain certain flags associated with a transformation.
If the client makes changes to elements in the transformation matrix,
the flags may need to be updated. This routine will examine the
transformation and update the flags. Currently there is only one
flag used. This package can be tailored for two dimensional use.
In that case, another flag can be used to indicate if the transformation

*/
chk_transform(t)
transform *t;
{
t->w_identity = FALSE;
if ((t->m[0][3]==0.0) && (t->m[1][3]==0.0) && (t->m[2][3]==0.0) &&
     (t->m[3][3]==1.0))t->w_identity = TRUE;
}
```

```
/* gp.c */


/*
This procedure sets the supplied matrix 't' to the identity matrix.
*/
make_identity(t)
transform *t;
{
int i,j;

for (i=0; i<=3; i++)
     for (j=0; j<=3; j++)
     t->m[i][j] = (i==j ? 1.0 : 0.0);
}

/*
This procedure sets the supplied matrix 't' to the identity matrix,
and also sets any flags that may be associated with the matrix.
*/
idn_transform(t)
transform *t;
{
make_identity(t);
chk_transform(t);
}

/*
This procedure sets 't' to a scaling transform. 'sx','sy', and 'sz'
correspond to the scaling factors for the x,y, and z axes.
*/
scl_transform(sx,sy,sz,t)
float sx,sy,sz;
transform *t;
{
make_identity(t);
t->m[0][0] = sx;
t->m[1][1] = sy;
t->m[2][2] = sz;
chk_transform(t);
}

/*
This procedure sets 't' to a transformation transform. 'tx','ty', and
'tz' correspond to translations in the x,y, and z directions.
*/
trn_transform(tx,ty,tz,t)
float tx,ty,tz;
transform *t;
{
make_identity(t);
t->m[3][0] = tx;
t->m[3][1] = ty;
t->m[3][2] = tz;
chk_transform(t);
}
```

```
/* gp.c */


/*
This procedure sets 't' to a primitive transformation of 'theta'
radians about the axis specified by 'axis'.  Axis is either XAXIS,
YAXIS, or ZAXIS, as defined in gp.h.
*/
rot_transform(axis,theta,t)
axis_id axis;
float theta;
transform *t;
{
int a,b;

switch (axis)
     {
     case XAXIS: a=2; b=1; break;
     case YAXIS: a=0; b=2; break;
     case ZAXIS: a=1; b=0; break;
     };
make_identity(t);
t->m[a][a] = (float) cos((double) theta);
t->m[b][b] = t->m[a][a];
t->m[a][b] = (float) sin((double) theta);
t->m[b][a] = -t->m[a][b];
chk_transform(t);
}

/*
Sets 't' to a perspective transformation (see eq. 23-4 in Newman and
Sproull), using S=d*tan(alpha), D=d, and F=1/one_over_f.  For a
straightforward viewing pyramid, without "hither" and "yon" clipping,
set d and one_over_f to 0.
*/
per_transform(alpha,d,one_over_f,t)
float alpha,d,one_over_f;
transform *t;
{
float tn;

tn = (float) tan((double) alpha);
make_identity(t);
t->m[2][2] = tn/(1.0-d*one_over_f);
t->m[2][3] = tn;
t->m[3][2] = -(tn*d)/(1.0-d*one_over_f);
t->m[3][3] = 0.0;
chk_transform(t);
}

/*
Sets 't' to a scaling and translation transformation that maps the
region xl<=x<=xr, yb<=y<=yt into the region -w<=x<=w,-w<=y<=w.  In
conjunction with clipping to the standard viewbox, this transformation
is pre-clipping part of the window-viewport transformation.
*/
```

```
/* gp.c */


wnd_transform(xl,yb,xr,yt,t)
float xl,yb,xr,yt;
transform *t;
{
float sx,sy;

sx = 2.0/(xr-xl); sy = 2.0/(yt-yb);
make_identity(t);
t->m[0][0] = sx; t->m[1][1] = sy;
t->m[3][0] = -sx*xl-1.0; t->m[3][1] = -sy*yb-1.0;
chk_transform(t);
}

/*
This procedure prints the transformation 't' on the standard output.
*/
prn_transform(t)
transform *t;
{
int i,j;

printf("Transform (%s)\n",(t->w_identity ? "w-identity":""));
for (i=0; i<=3; i++)
     {
     for (j=0; j<=3; j++)
     printf("%f ",t->m[i][j]);
     printf("\n");
     }
}

/*
This procedure copies the contents of matrix 'from' into matrix 'to'.
*/
mat_copy(from,to)
transform *from,*to;
{
int i,j;

for (i=0; i<=3; i++)
     for (j=0; j<=3; j++)
     to->m[i][j] = from->m[i][j];
chk_transform(to);
}

/*
This procedure sets c = a*b.  The transform 'c' may be the same
transform as 'a' or 'b'.
*/
mat_mul(a,b,c)
transform *a,*b,*c;
{
transform d;
int i,j,k;
float s;
```

```
for (i=0; i<=3; i++)
     for (j=0; j<=3; j++)
     {
     s = 0.0;
     for (k=0; k<=3; k++)
          s = s + a->m[i][k]*b->m[k][j];
     d.m[i][j] = s;
     }
mat_copy(&d,c);
chk_transform(c);
}

/*
This procedure sets [xo yo zo wo] to the result of post-multiplying
the vector [x y z w] by the transformation 't'.
*/
vec_mul(x,y,z,w,t,xo,yo,zo,wo)
float x,y,z,w,*xo,*yo,*zo,*wo;
transform *t;
{
float xtemp,ytemp,ztemp,wtemp;

xtemp = x*t->m[0][0] + y*t->m[1][0] + z*t->m[2][0] + w*t->m[3][0];
ytemp = x*t->m[0][1] + y*t->m[1][1] + z*t->m[2][1] + w*t->m[3][1];
ztemp = x*t->m[0][2] + y*t->m[1][2] + z*t->m[2][2] + w*t->m[3][2];
wtemp = (t->w_identity ? w : x*t->m[0][3] + y*t->m[1][3] + z*t->m[2][3] + w*t->m[3][3])
*xo = xtemp; *yo = ytemp; *zo = ztemp; *wo = wtemp;
}

/*
This procedure updates the 'm' matrix by concatenating 'q' and 'm'.
*/
static update_m(concat)
boolean concat;
{
if (concat)mat_mul(&q,&m,&m);
idn_transform(&q);
q_identity = TRUE;
}

/*
This procedure sets 'm' to the matrix 't'.
*/
set_m(t)
transform *t;
{
mat_copy(t,&m);
update_m(FALSE);
}

/*
This procedure returns 'm' in the matrix 't'.
*/
```

```
/* gp.c */


get_m(t)
transform *t;
{
if (!q_identity)update_m(TRUE);
mat_copy(&m,t);
}

/*
This procedure pushes the current value of 'm' onto the stack.
*/
push_m()
{
s_transform *p;

if (m_free != NULL)
     {
     p = m_free;
     m_free = m_free->next;
     }
else p = (s_transform *)malloc(sizeof(s_transform));
p->next = m_stack;
get_m(&(p->t));
m_stack = p;
}

/*
This procedure pops a transform off of the stack and puts it in 'm'.
*/
pop_m()
{
s_transform *p;

set_m(&(m_stack->t));
p = m_stack;
m_stack = m_stack->next;
p->next = m_free;
m_free = p;
}

/*
This procedure computes a scaling transformation 't', and then
sets q = q*t.
*/
scale(sx,sy,sz)
float sx,sy,sz;
{
transform t;

scl_transform(sx,sy,sz,&t);
mat_mul(&q,&t,&q);
q_identity = FALSE;
}

/*
```

```
/* gp.c */

This procedure computes a translation transform 't', and then
sets q = q*t.
*/
translate(tx,ty,tz)
float tx,ty,tz;
{
transform t;

trn_transform(tx,ty,tz,&t);
mat_mul(&q,&t,&q);
q_identity = FALSE;
}

/*
This procedure computes a rotation transform 't', and then sets
q = q*t.
*/
rotate(axis,theta)
axis_id axis;
float theta;
{
transform t;

rot_transform(axis,theta,&t);
mat_mul(&q,&t,&q);
q_identity = FALSE;
}

/*
This procedure sets q = q*t.
*/
concatenate(t)
transform *t;
{
mat_mul(&q,t,&q);
q_identity = FALSE;
}

/*
Sets the current viewport parameters of this package:
    xl: x coordinate of left edge of window
    yb: y coordinate of bottom edge of window
    xr: x coordinate of right edge of window
    yt: y coordinate of top edge of window
*/
set_viewport(xl,yb,xr,yt)
float xl,yb,xr,yt;
{
inquiry_response v;

vxl = xl; vyb = yb; vxr = xr; vyt = yt;
vcx = (vxl+vxr)/2.0; vcy = (vyb+vyt)/2.0;
vsx = (vxr-vxl)/2.0; vsy = (vyt-vyb)/2.0;
show_inquire(&v);
```

```
vsx = vsx*v.xadjust; vsy = vsy*v.yadjust;
}

/*
This procedure reads the current viewport parameter in this package.
*/
get_viewport(xl,yb,xr,yt)
float *xl,*yb,*xr,*yt;
{
*xl = vxl; *yb = vyb; *xr = vxr; *yt = vyt;
}

/*
This procedure sets the package for 'vector' mode. This means that
the matrix 'm' will contain the matrix for the complete transformation
to screen coordinates, rather than the intermediate transformation
to world coordinates that is needed to compute surface normals.
This previous matrix 'm' is saved on the stack.
*/
vec_mode()
{
push_m();
mat_mul(&m,&camera,&m);
}

/*
Sets the matrix 'm' to a transformation which transforms objects
to the camera's coordinate system and performs the perspective transformation.
The previous matrix 'm' is saved on the stack.
*/
cam_mode()
{
push_m();
set_m(&camera);
}

/*
Ends the current 'mode' by popping the previous transformation
off of the stack.
*/
end_mode()
{
pop_m();
}

/*
Sets 'm' to the matrix 'camera'.
*/
set_camera()
{
get_m(&camera);
}

save_preper()
```

```
{
mat_mul(&q,&m,&preper);
}

/*
Move the package's idea of the current location to the specified position.
*/
move_to(x,y,z)
float x,y,z;
{
cxo = x; cyo = y; czo = z;
if (!q_identity)update_m(TRUE);
vec_mul(x,y,z,1.0,&m,&cx,&cy,&cz,&cw);
}

/*
Draw a vector from the current position to the specified position and
update the current position.
*/
line_to(x,y,z)
float x,y,z;
{
float x1,y1,z1,w1;

cxo = x; cyo = y; czo = z;
x1 = cx; y1 = cy; z1 = cz; w1 = cw;
if (!q_identity)update_m(TRUE);
vec_mul(x,y,z,1.0,&m,&cx,&cy,&cz,&cw);
show_line(vsx*x1/w1+vcx,vsy*y1/w1+vcy,vsx*cx/cw+vcx,vsy*cy/cw+vcy);
}

/*
Sets [x1 y1 z1 w1] to the result of post-multiplying the vector
[x y z w] by the matrix 'm'.
*/
trans_point(x,y,z,w,x1,y1,z1,w1)
float x,y,z,w,*x1,*y1,*z1,*w1;
{
float x2,y2,z2,w2;

if (!q_identity)update_m(TRUE);
vec_mul(x,y,z,w,&m,&x2,&y2,&z2,&w2);
*x1 = x2/w2;
*y1 = y2/w2;
*z1 = z2/w2;
*w1 = 1.0;
}

/*
This procedure converts the given coordinate [x y z w] to screen coordinates.
*/
to_screen(x,y,z,w,xs,ys,zs)
float x,y,z,w,*xs,*ys,*zs;
{
```

```
/* gp.c */

float x1,y1,z1,w1;
vec_mul(x,y,z,w,&m,&x1,&y1,&z1,&w1);
*xs = (int) vsx*x1/w1+vcx; *ys = (int) vsy*y1/w1+vcy; *zs = z1;
if (z1<3.0)return(-1);
else return(0);
}

/*
This procedure returns this package's idea of the current position.
*/
get_pos(x,y,z)
float *x,*y,*z;
{
*x = cxo; *y = cyo; *z = czo;
}

/*
This procedure initializes the pipeline. It sets the viewport to the largest
square that fits on the screen, sets 'm' to the identity matrix, and empties
the 'm' stack. It must be called before any of the other procedures are used.
*/
pipe_init()
{
int siz;

m_stack = m_free = NULL;
idn_transform(&m);
update_m(FALSE);
show_inquire(&v);
siz = v.xmax - v.xmin;
if ((v.ymax-v.ymin)<siz)siz = v.ymax - v.ymin;
set_viewport((float) v.xmin,(float) v.ymin,(float) v.xmin+siz,(float) v.ymin+siz);
}

vec1(a, b, c, d, e, f, g, h)
float a, b, c, d;
float *e, *f, *g, *h;
{
register int i,j;

#ifdef DBGVEC
fprintf (stderr, "%f %f %f\n", a, b, c);
#endif

vec_mul(a,b,c,d,&preper,e,f,g,h);

#ifdef DBGVEC
for (i=0; i<4; i++) {
     for (j=0; j<4; j++)
               fprintf (stderr, "# %f ", preper.m[i][j]);
     fprintf (stderr, "\n");
     }
fprintf (stderr, "%f %f %f\n", *e, *f, *g);
#endif
```

```
/* gp.c */

}
```

```
/* gpak.c */

#ifdef GRINNELL
#include <grinnell.h>
#include <iusconfig.h>
#endif

#include <stdio.h>
/*
#include <gp.h>
*/
#include "gp.h"

#ifdef IKONAS
#include <ikdefs.h>
#endif

/* This file contains all of the device dependent graphics commands.
   Additional devices may be added here as they become available.  Note:
   some of these routines may not be compatible with other devices if
   the devices do not use the RGB color convention or have more than
   8 bits per primary color. */
/* Modified clear() so that it generates the appropriate background color */
/*  format for dynamic lighting .  -dh 8/22/82                                  */


#ifdef GRINNELL
static int frm;                        /* Grinnell frame number */
#endif

#ifdef IKONAS
typedef struct
     {
     char red;
     char green;
     char blue;
     char fill;
     } pixel;
#endif

static inquiry_response v;  /* Environment information */

/* Initialize graphics display */
int initgraphics()
{
#ifdef GRINNELL
int dum1,dum2,dum3;
if (g_init(G_VISION_IFVC)==G_GMRERROR)exit(-1);
/* The following is a hack to make sure that the bad memory channel
   is not allocated.  This can be removed if the bad chip is ever replaced. */
dum1 = g_allocfrm(G_MAPPED,8); dum2 = g_allocfrm(G_MAPPED,8);
dum3 = g_allocfrm(G_MAPPED,8);
g_freefrm(dum1); g_freefrm(dum2);
frm = g_allocfrm(G_RGB,8);  /* Allocate RGB with 8 bits per primary */
```

```
g_freefrm(dum3);
g_chgdisp(frm);                        /* Make Grinnell display this frame */
#endif
#ifdef IKONAS
setorig(BORIG);
#endif
#ifdef DEBUG
fprintf(stderr,"Initializing graphics.\n");
#endif
show_inquire(&v);                   /* Get environment information */
#ifdef GRINNELL
return(frm);                           /* Return frame if needed by user of package */
#endif
#ifdef RAMTEK
openpl();
#endif
}

/* Set pixel (x,y) to color (r g b) */
writepixel(x,y,r,g,b)
int x,y,r,g,b;
{
#ifdef GRINNELL
g_setcpixel(frm,511-y,x,r,g,b);
#endif
#ifdef DEBUG
fprintf(stderr,"Writepixel: (%d,%d) = (%d %d %d)\n",x,y,r,g,b);
#endif
}

/* Read pixel (x,y) and return color (r g b) */
readpixel(x,y,r,g,b)
int x,y,*r,*g,*b;
{
#ifdef GRINNELL
g_getcpixel(frm,511-y,x,r,g,b);
#endif
#ifdef DEBUG
fprintf(stderr,"Readpixel: (%d,%d)\n",x,y);
#endif
}

/* Return device characteristics */
show_inquire(v)
inquiry_response *v;
{
#ifdef DEBUG
v->xmin = 0;
v->ymin = 0;
v->xmax = 511;
v->ymax = 511;
v->imax = 255;
v->xadjust = 1.0; v->yadjust = 1.0;
#endif
```

```
/* gpak.c */

#ifdef GRINNELL
v->xmin = 0;
v->ymin = 0;
v->xmax = 511;
v->ymax = 511;
v->imax = 255;
v->xadjust = 1.0; v->yadjust = 1.0;
#endif
#ifdef IKONAS
v->xmin = 0;
v->ymin = 0;
v->xmax = 511;
v->ymax = 511;
v->imax = 255;
v->xadjust = 1.0;
v->yadjust = 1.0;
#endif
#ifdef RAMTEK
v->xmin = 0;
v->ymin = 0;
v->xmax = 1023;
v->ymax = 1023;
v->imax = 255;
v->xadjust = 1.0;
v->yadjust = 1.0;
#endif
}

/* Draw a line from (x1,y1) to (x2,y2) */
show_line(x1,y1,x2,y2)
float x1,y1,x2,y2;
{
#ifdef GRINNELL
int ax,ay,bx,by;
ax = (int) x1;
ay = (int) (511-y1);
bx = (int) x2;
by = (int) (511-y2);
g_linedraw(frm,ay,ax,by,bx);
#endif
#ifdef DEBUG
fprintf(stderr,"Line from (%d,%d) to (%d,%d)\n",(int)x1,(int)y1,(int)x2,(int)y2);
#endif
}

/* Erase the screen */
show_erase()
{
#ifdef GRINNELL
g_ersfrm(frm);
#endif
#ifdef DEBUG
fprintf(stderr,"Erasing screen.\n");
#endif
```

```
/* gpak.c */

#ifdef RAMTEK
erase();
#endif
}

/* Set portion of scan line y from x1 to x2 to color (r g b) */
hline(y,x1,x2,r,g,b)
int y,x1,x2,r,g,b;
{
#ifdef IKONAS
pixel pbuf[512];
int i;

for (i=x1; i<=x2; i++)
    {
    pbuf[i].red = r;
    pbuf[i].green = g;
    pbuf[i].blue = b;
    }
ikpwrite(&pbuf[x1],4*sizeof(char)*(x2-x1+1),x1,y);
#endif
#ifdef GRINNELL
g_setcblkpix(frm,511-y,511-y,x1,x2,r,g,b);
#endif
#ifdef DEBUG
fprintf(stderr,"hline: Line %d from %d to %d in (%d %d %d)\n",y,x1,x2,r,g,b);
#endif
#ifdef RAMTEK
ram_color(r,g,b);
box(x1,y,x2,y);
#endif
}

/* Set scan line y using supplied arrays */
write_raster(y,rarray,garray,barray)
int y;
char *rarray,*garray,*barray;
{
#ifdef IKONAS
pixel pbuf[512];
int i;

for (i=0; i<512; i++)
    {
    pbuf[i].red = rarray[i];
    pbuf[i].green = garray[i];
    pbuf[i].blue = barray[i];
    }
ikpwrite(pbuf,sizeof(pixel)*512,0,y);
#endif
#ifdef GRINNELL
g_blkcset(frm,511-y,511-y,0,511,rarray,garray,barray);
#endif
#ifdef DEBUG
```

```
fprintf(stderr,"setting raster %d\n",y);
#endif
}

/* Set portion of scan line y from x1 to x2 using supplied arrays */
segment(y,x1,x2,rarray,garray,barray)
int y,x1,x2;
char *rarray,*garray,*barray;
{
#ifdef IKONAS
pixel pbuf[512];
int i;
#endif
#ifdef GRINNELL
g_blkcset(frm,511-y,511-y,x1,x2,&rarray[x1],&garray[x1],&barray[x1]);
#endif
#ifdef IKONAS
for (i=x1; i<=x2; i++)
     {
     pbuf[i].red = rarray[i];
     pbuf[i].green = garray[i];
     pbuf[i].blue = barray[i];
     }
ikpwrite(&pbuf[x1],4*sizeof(char)*(x2-x1+1),x1,y);
#endif
#ifdef DEBUG
fprintf(stderr,"Setting raster %d from (%d to %d)\n",y,x1,x2);
#endif
}

/* Read scan line y into supplied arrays */
read_raster(y,rarray,garray,barray)
int y;
char *rarray,*garray,*barray;
{
#ifdef GRINNELL
g_blkcget(frm,511-y,511-y,0,511,rarray,garray,barray);
#endif
#ifdef DEBUG
fprintf(stderr,"Reading raster %d\n",y);
#endif
}

/* Clear screen using specified color */
clear(r,g,b)
int r,g,b;
{
int colorformat;      /* used to re-format color for dynamic lighting - dh */
#define CUTOFF1 42   /* cutoff of desired intensities at 1/6 of max. */
#define CUTOFF2 128 /* cutoff of desired intensities at 3/6 of max. */
#define CUTOFF3 213 /* cutoff of desired intensities at 5/6 of max. */
#ifdef IKONAS
face *f;
/*
```

```
/* gpak.c */

pixel pbuf[512];
int i;

for (i=0; i<512; i++)
     {
     pbuf[i].red = r;
     pbuf[i].green = g;
     pbuf[i].blue = b;
     }
for (i=0; i<512; i++)
     ikpwrite(pbuf,sizeof(pbuf),0,i);
*/
/* fast polygon fill is faster than method above */
f = (face *)malloc(sizeof(face)+4*sizeof(point));
f->vlist[0].sx = 0.0;
f->vlist[0].sy = 0.0;
f->vlist[1].sx = 0.0;
f->vlist[1].sy = 511.0;
f->vlist[2].sx = 511.0;
f->vlist[2].sy = 511.0;
f->vlist[3].sx = 511.0;
f->vlist[3].sy = 0.0;
f->num = 4;

/* Change r, g, and b to appropriate format for dynamic lighting - dh 8/22/82 */
/* Re-format the color and store it in the green byte */
if (r < CUTOFF1) colorformat = 0;
else if (r < CUTOFF2) colorformat = 1;
else if (r < CUTOFF3) colorformat = 2;
else colorformat = 3;
if (g < CUTOFF1);
else if (g < CUTOFF2) colorformat = colorformat + 4;
else if (g < CUTOFF3) colorformat = colorformat + 8;
else colorformat = colorformat + 12;
if (b < CUTOFF1);
else if (b < CUTOFF2) colorformat = colorformat + 16;
else if (b < CUTOFF3) colorformat = colorformat + 32;
else colorformat = colorformat + 48;
r = 0;       /* normal is stored in red byte - loc. 0 represents background */
g = colorformat;     /* reformatted color info. is stored in green byte */
b = 0;       /* blue byte is unused - set it to zeros */

po_write(r,g,b,f);
free(f);
#endif
#ifdef GRINNELL
g_setcblkpix(frm,511,0,0,511,r,g,b);
#endif
#ifdef DEBUG
fprintf(stderr,"Clearing screen\n");
#endif
#ifdef RAMTEK
ram_color(r,g,b);
box(0,0,1023,1023);
```

/* gpak.c */

```
#endif
}

/* Set a rectangular area on screen to (r g b) */
set_block(xl,yb,xr,yt,r,g,b)
int xl,yb,xr,yt,r,g,b;
{
#ifdef GRINNELL
g_setcblkpix(frm,511-yt,511-yb,xl,xr,r,g,b);
#endif
#ifdef DEBUG
fprintf(stderr,"Setting block: xl=%d yb=%d xr=%d yt=%d to (%d %d %d)\n",xl,yb,xr,yt,r,g,
#endif
}
```

```c
/* init.c */

/*
#include <gp.h>
*/
#include "gp.h"
#include <math.h>
#include <signal.h>

/* This file contains the initialization procedure for graphics package. */
/*   Modified to include call to read in normal table - dh  8/21/82        */

/* Main initialization call */
int init()
{
int frm,handler();
inquiry_response v;
char st[132];

init_shade();
frm = initgraphics();                          /* Initialize Grinnell */
show_inquire(&v);                         /* Get display characteristics */
clipwindow((float)v.xmin,(float)v.xmax,(float)v.ymin,(float)v.ymax);
set_smooth(FALSE);                          /* Default to no smooth shading */
pipe_init();                                    /* Initialize matrix package */
set_color(0,255,0);                       /* Make green default color */
/* signal(SIGQUIT,handler); */                   /* Catch quits so that when making
                                                a movie we have some way to
                                                temporarily halt the program if
                                                someone turns on the lights! */

normlinit();          /* read in table of 254 "ideal" normals */
return(frm);
}

/* Interrupt handler */
handler()
{
char c;

signal(SIGQUIT,handler);
printf("Do you really wish to quit? ");
scanf("%c",&c);
if (c=='y')exit(-1);
}
```

```
/* clip.c */

/*
#include <gp.h>
*/
#include "gp.h"

/*  Modified version of Bob Hon's routines */

/*  package to clip a polygon against a window
 */


static point
lfirst,rfirst,tfirst,bfirst,                /* first pts */
lsave,rsave,bsave,tsave;                  /* prev pts */

static float left,right,bottom,top; /* window to clip against */

static char lflag,rflag,tflag,bflag;

static unsigned int vcount;

static face *ans;

clipwindow(l,r,b,t)
float l,r,b,t;
{
  left = l;
  right = r;
  bottom = b;
  top = t;
  return(0);
}

face *clippoly(f)
face *f;
{
  unsigned int i;

  lflag = rflag = bflag = tflag = 1;
  if (f->num < 3) return(0);
  if ((ans = (face *) malloc(2*f->num*sizeof(point)+sizeof(face)))==0)return(0);

  vcount = 0;
  for (i = 0; i < f->num; i++) clipleft(f->vlist[i]);
  finish();
  ans->num = vcount;
  ans->smooth = f->smooth;
  ans->red = f->red;
  ans->green = f->green;
  ans->blue = f->blue;
  ans->bred = f->bred;
  ans->bgreen = f->bgreen;
```

/* clip.c */

```
  ans->bblue = f->bblue;
  return(ans);
}

static clipleft(p)
point p;
{
  if (lflag)
     {lsave = p; lfirst = p; lflag = 0;}
  else
     {if ((lsave.sx < left && p.sx > left) ||
      (lsave.sx > left && p.sx < left))
         {point temp;
     float alpha;
     alpha = ((float) (lsave.sx - left))/(float) (lsave.sx - p.sx);
     temp.sx = left; temp.sy = (lsave.sy + alpha*(p.sy-lsave.sy));
     temp.sz = lsave.sz + alpha*(p.sz-lsave.sz);
     cliptop(temp);
         }
      lsave = p;
     }
  if (lsave.sx >= left) cliptop(lsave);
}

static cliptop(p)
point p;
{
  if (tflag)
     {tsave = p; tfirst = p; tflag = 0;}
  else
     {if ((tsave.sy < top && p.sy > top) ||
      (tsave.sy > top && p.sy < top))
         {point temp;
     float alpha;
     alpha = ((float) (tsave.sy - top))/(float) (tsave.sy - p.sy);
     temp.sx = (tsave.sx + alpha*(p.sx-tsave.sx)); temp.sy = top;
     temp.sz = tsave.sz + alpha*(p.sz-tsave.sz);
     clipright(temp);
         }
      tsave = p;
     }
  if (tsave.sy <= top) clipright(tsave);
}

static clipright(p)
point p;
{
  if (rflag)
     {rsave = p; rfirst = p; rflag = 0;}
  else
     {if ((rsave.sx < right && p.sx > right) ||
      (rsave.sx > right && p.sx < right))
         {point temp;
     float alpha;
```

```
/* clip.c */

      alpha = ((float) (rsave.sx - right))/(float) (rsave.sx - p.sx);
      temp.sx = right; temp.sy = (rsave.sy + alpha*(p.sy-rsave.sy));
      temp.sz = rsave.sz + alpha*(p.sz-rsave.sz);
      clipbot(temp);
          }
       rsave = p;
      }
   if (rsave.sx <= right) clipbot(rsave);
}

static clipbot(p)
point p;
{
   if (bflag)
      {bsave = p; bfirst = p; bflag = 0;}
   else
      {if ((bsave.sy < bottom && p.sy > bottom) ||
       (bsave.sy > bottom && p.sy < bottom))
          {point temp;
      float alpha;
      alpha = ((float) (bsave.sy - bottom))/(float) (bsave.sy - p.sy);
      temp.sx = (bsave.sx + alpha*(p.sx-bsave.sx)); temp.sy = bottom;
      temp.sz = bsave.sz + alpha*(p.sz-bsave.sz);
      send(temp);
          }
       bsave = p;
      }
   if (bsave.sy >= bottom) send(bsave);
}

static finish()
{
   float alpha;
   point temp;

   if (!tflag && ((lsave.sx < left) && (lfirst.sx > left) ||
       (lsave.sx > left) && (lfirst.sx < left)))
          {alpha = ((float) (lsave.sx - left))/(float) (lsave.sx - lfirst.sx);
      temp.sx = left; temp.sy = (lsave.sy + alpha*(lfirst.sy-lsave.sy));
      temp.sz = lsave.sz + alpha*(lfirst.sz-lsave.sz);
      cliptop(temp);
          }
   if ( !rflag && ((tsave.sy < top) && (tfirst.sy > top) ||
       (tsave.sy > top) && (tfirst.sy < top)))
          {alpha = ((float) (tsave.sy - top))/(float) (tsave.sy - tfirst.sy);
      temp.sx = (tsave.sx + alpha*(tfirst.sx-tsave.sx)); temp.sy = top;
      temp.sz = tsave.sz + alpha*(tfirst.sz-tsave.sz);
      clipright(temp);
          }
   if ( !bflag && ((rsave.sx < right) && (rfirst.sx > right) ||
       (rsave.sx > right) && (rfirst.sx < right)))
          {alpha = ((float) (rsave.sx - right))/(float) (rsave.sx - rfirst.sx);
      temp.sx = right; temp.sy = (rsave.sy + alpha*(rfirst.sy-rsave.sy));
      temp.sz = rsave.sz + alpha*(rfirst.sz-rsave.sz);
```

```
      clipbot(temp);
          }
   if ( vcount != 0 && ((bsave.sy < bottom) && (bfirst.sy > bottom) ||
       (bsave.sy > bottom) && (bfirst.sy < bottom)))
          {alpha = ((float) (bsave.sy - bottom))/(float) (bsave.sy - bfirst.sy);
      temp.sx = (bsave.sx + alpha*(bfirst.sx-bsave.sx)); temp.sy = bottom;
      temp.sz = bsave.sz + alpha*(bfirst.sz-bsave.sz);
      send(temp);
          }
}

static send(p)
point p;
{
   if (vcount != 0)
      if ((ans->vlist[vcount-1].sx == p.sx) && (ans->vlist[vcount-1].sy == p.sy)) return;
   ans->vlist[vcount++] = p;
}
```

```
/* hidden.c */

/*
#include <gp.h>
*/
#include "gp.h"
#include <math.h>
#include <stdio.h>
#define VERSION 2.0

/* This file contains all of the routines associated with the hidden
   surface algorithm.  This package uses an algorithm presented by
   Fuchs, et al. at SIGGRAPH '80.  See the conference proceedings for
   more info. */

#define EPSILON .05

/* List element record */
typedef struct listel_struct
     {
     face *f;
     struct listel_struct *next;
     } listel;

/* Head of list record */
typedef struct
     {
     int count;
     listel *pointer;
     } head;

static head main;
static node *root = NULL;     /* Pointer to root of binary tree */
static boolean treebuilt = FALSE;
static int input =0;                   /* Polygon counts */
static int total =0;
static FILE *fd;
static int file;
static int testnum;

static float ex,ey,ez;                 /* Viewpoint (eye position) */

/* Test whether or not "clipf" will split source */
boolean willsplit(clipf,source)
face *clipf,*source;
{
boolean intersect,positive,negative;
float *val;
point *v;
int i;

intersect = positive = negative = FALSE;
/* Allocate an array of floats - one for each vertex */
val = (float *)malloc(source->num*sizeof(float));
```

```
/* hidden.c */

for (i=0; i<source->num; i++)
    {
    v = &source->vlist[i];
    /* Compute distance (positive or negative) from clipping plane */
    val[i] = v->x*clipf->ni + v->y*clipf->nj + v->z*clipf->nk + v->w*clipf->d;
    /* Determine whether this vertex lies on positive side, negative side,
       or within splitting plane */
    if (fabs(val[i])<EPSILON)val[i] = 0.0;
    else if (val[i]>0.0)positive = TRUE;
    else negative = TRUE;
    if (positive && negative)
    {
    intersect = TRUE;
    break;
    }
    }
free(val);
return(intersect);
}

/* This routine uses polygon 'clipf' to potentially split polygon 'source'
   into two polygons, 'pos' and 'neg'. If the source polygon lies
   entirely on one side of the clipping polygon, then either 'pos' or
   'neg' will be set to NULL. */
split(clipf,source,pos,neg)
face *clipf,*source,**pos,**neg;
{
int i,siz,negcount,poscount;
face *fpos,*fneg;
boolean intersect,positive,negative;
point new,*prev,*curr,*v;
float alpha,one_minus_alpha,*val,val2;

intersect = positive = negative = FALSE;
/* Allocate an array of floats - one for each vertex */
val = (float *)malloc(source->num*sizeof(float));
for (i=0; i<source->num; i++)
    {
    v = &source->vlist[i];
    /* Compute distance (positive or negative) from clipping plane */
    val[i] = v->x*clipf->ni + v->y*clipf->nj + v->z*clipf->nk + v->w*clipf->d;
    /* Determine whether this vertex lies on positive side, negative side,
       or within splitting plane */
    if (fabs(val[i])<EPSILON)val[i] = 0.0;
    else if (val[i]>0.0)positive = TRUE;
    else negative = TRUE;
    /* Source plane must intersect splitting plane (vertices lie on opposite sides.) */
    if (positive && negative)intersect = TRUE;
    }
if (!intersect)                    /* No intersection - this makes things easy */
    {
    if (positive)
    {
    /* Source lies entirely on positive side of clipping plane */
```

```
        *pos = source;
        *neg = NULL;
        }
        else
        {
        /* Source lies entirely on negative side of clipping plane */
        *neg = source;
        *pos = NULL;
        }
        free(val);                    /* Release space allocated for val array */
        return;            /* All done! */
        }
siz = sizeof(face) + ((3*source->num)/2)*sizeof(point);
/* Allocate new face structures for the two new polygons */
fpos = (face *)malloc(siz);
fneg = (face *)malloc(siz);
*pos = fpos;
*neg = fneg;
poscount = negcount = 0;
prev = &source->vlist[source->num-1];
val2 = val[source->num-1];
for (i=0; i<source->num; i++)
        {
        curr = &source->vlist[i];
        /* If this vertex is in splitting plane, copy to both polygons */
        if (val[i]==0.0)
        {
        fneg->vlist[negcount++] = *curr;
        fpos->vlist[poscount++] = *curr;
        }
        else if (val[i]<0.0)
        {
        /* If this vertex is on negative side and previous was also
           on negative side, copy to negative polygon. */
        if (val2 <= 0.0)fneg->vlist[negcount++] = *curr;
        /* If this vertex is on negative side and previous was on
          positive side, we must compute intersection and copy
           to both polygons.  Include this vertex in negative polygon. */
        else
            {
            alpha = val[i]/(val[i]-val2);
            one_minus_alpha = 1.0-alpha;
            new.x = one_minus_alpha*curr->x + alpha*prev->x;
            new.y = one_minus_alpha*curr->y + alpha*prev->y;
            new.z = one_minus_alpha*curr->z + alpha*prev->z;
            new.w = one_minus_alpha*curr->w + alpha*prev->w;
#ifdef SMOOTH
            if (source->smooth)
                {
                new.ni = one_minus_alpha*curr->ni + alpha*prev->ni;
                new.nj = one_minus_alpha*curr->nj + alpha*prev->nj;
                new.nk = one_minus_alpha*curr->nk + alpha*prev->nk;
                }
#endif
```

```
            fneg->vlist[negcount++] = fpos->vlist[poscount++] = new;
            fneg->vlist[negcount++] = *curr;
            }
        }
        else
        {
        /* If this vertex is on the positive side and previous was also
           on positive side, copy to positive polygon. */
        if (val2 >= 0.0)fpos->vlist[poscount++] = *curr;
        /* If this vertex is on positive side and previous was on
           negative side, we must compute intersection and copy
           to both polygons. Include this vertex in positive polygon. */
        else
            {
            alpha = val[i]/(val[i]-val2);
            one_minus_alpha = 1.0-alpha;
            new.x = one_minus_alpha*curr->x + alpha*prev->x;
            new.y = one_minus_alpha*curr->y + alpha*prev->y;
            new.z = one_minus_alpha*curr->z + alpha*prev->z;
            new.w = one_minus_alpha*curr->w + alpha*prev->w;
#ifdef SMOOTH
            if (source->smooth)
                {
                new.ni = one_minus_alpha*curr->ni + alpha*prev->ni;
                new.nj = one_minus_alpha*curr->nj + alpha*prev->nj;
                new.nk = one_minus_alpha*curr->nk + alpha*prev->nk;
                }
#endif
            fneg->vlist[negcount++] = fpos->vlist[poscount++] = new;
            fpos->vlist[poscount++] = *curr;
            }
        }
        prev = curr;
        val2 = val[i];
        }
if (negcount <= 2)            /* Degenerate source polygon */
        {
        free(fneg);
        *neg = NULL;
        }
else
        {
        fneg->num = negcount;
        /* Information from source polygon must be copied to new polygon. */
        fneg->red = source->red;
        fneg->green = source->green;
        fneg->blue = source->blue;
        fneg->bred = source->bred;
        fneg->bgreen = source->bgreen;
        fneg->bblue = source->bblue;
        fneg->smooth = source->smooth;
        fneg->ni = source->ni;
        fneg->nj = source->nj;
        fneg->nk = source->nk;
```

```
/* hidden.c */

     fneg->d = source->d;
/*
     printf("Negative side:\n");
     if (negcount>2)for (i=0; i<negcount; i++)
     printf("neg[%d]=(%f,%f,%f,%f)\n",i,fneg->vlist[i].x,fneg->vlist[i].y,fneg-v);vlist[i].z,fneg->vlist
*/
     }
if (poscount <= 2)               /* Degenerate source polygon */
     {
     free(fpos);
     *pos = NULL;
     }
else
     {
     fpos->num = poscount;
     /* Information from source polygon must be copied to new polygon. */
     fpos->red = source->red;
     fpos->green = source->green;
     fpos->blue = source->blue;
     fpos->bred = source->bred;
     fpos->bgreen = source->bgreen;
     fpos->bblue = source->bblue;
     fpos->smooth = source->smooth;
     fpos->ni = source->ni;
     fpos->nj = source->nj;
     fpos->nk = source->nk;
     fpos->d = source->d;
/*
     printf("Positive side:\n");
     if (poscount>2)for (i=0; i<poscount; i++)
     printf("pos[%d]=(%f,%f,%f,%f)\n",i,fpos->vlist[i].x,fpos->vlist[i].y,fpos-);vlist[i].z,fpos->vlist[i
*/
     }
/* If this package is used for animation, program size will grow
   considerably if we don't free source polygon here.  The problem
   is, there is no way to know if the user of this package will
   still need this polygon. */
free(source);
free(val);
}

/* Insert the given face into the specified polygon list */
insert(f,listh)
face *f;
head *listh;
{
listel *l;

if (f)
     {
     listh->count++;
     l = (listel *)malloc(sizeof(listel));
     l->next = listh->pointer;
     listh->pointer = l;
```

```
        l->f = f;
        }
}

/* Save the polygon for later tree building */
tree_enter(f)
face *f;
{
insert(f,&main);
input++;
}

/* Choose a good polygon for this node in tree */
/* The method used below is to check several polygons at random
   do see which splits the fewest polygons.  This method is not
   optimal but seems to work very well. */
listel *choose(h)
head *h;
{
listel *p,*next,*choice;
int *splitcount;
boolean *checked;
int scount[500];
boolean check[500];
int i,j,count,min,num,random;

if (h->count>500)
        {
        splitcount = (int *)malloc(h->count*sizeof(int));
        checked = (boolean *)malloc(h->count*sizeof(boolean));
        }
else
        {
        splitcount = scount;
        checked = check;
        }
for (i=0; i<h->count; i++)checked[i] = FALSE;
min = 999999;
choice = h->pointer;
num = testnum;
if (num==0 || num>h->count)num = h->count;
for (i=0; i<num; i++)
        {
        if (num!=h->count) do random = rand()%h->count; while (checked[random]);
        else random = i;
        checked[random] = TRUE;
        p = h->pointer;
        next = h->pointer;
        for (j=0; j<random; j++)next = next->next;
        count = 0;
        while (p!=NULL)
        {
        if (next!=p)if (willsplit(next->f,p->f))count++;
        p = p->next;
```

```
/* hidden.c */

      }
      splitcount[i] = count;
      if (count<min)
      {
      min = count;
      choice = next;
      }
      if (count==0)break;
      }
/*
printf("Entries=%d min=%d\n",h->count,min);
*/
if (h->count>500)
      {
      free(splitcount);
      free(checked);
      }
return(choice);
}

/* Build a BSP tree for stored polygon list */
build()
{
char s[30];
int seed;

treebuilt = TRUE;
testnum = 5;
/* Number of polygons to check at each node */
fprintf(stderr,"Enter number of polygons to check [%d]: ",testnum);
if (gets(s)&&s[0])
     sscanf(s,"%d",&testnum);
seed = 0;
/* Seed for random number generator */
fprintf(stderr,"Enter seed [%d]: ",seed);
if (gets(s)&&s[0])
     sscanf(s,"%d",&seed);
srand(seed);
fprintf(stderr,"Building tree...");
buildtree(&root,&main);
fprintf(stderr,"done\n");
fprintf(stderr,"Seed: %d Test: %d In: %d Out: %d Ratio: %.2f\n",seed,testnum,input,total/(float)input);(floa
}

/* Build a tree returning pointer in tree using polygons pointed to by h */
buildtree(tree,h)
node **tree;
head *h;
{
listel *choice,*p,*temp;
head *poshead,*neghead;
face *pos,*neg;

*tree = (node *)malloc(sizeof(node));
```

```
/* hidden.c */

(*tree)->pos = NULL;
(*tree)->neg = NULL;
poshead = (head *)malloc(sizeof(head));
neghead = (head *)malloc(sizeof(head));
poshead->count = neghead->count = 0;
poshead->pointer = neghead->pointer = NULL;
choice = (listel *)choose(h);
(*tree)->f = choice->f;
total++;
p = h->pointer;
while (p!=NULL)
     {
     if (p!=choice)
     {
     split(choice->f,p->f,&pos,&neg);
     insert(pos,poshead);
     insert(neg,neghead);
     }
     temp = p;
     p = p->next;
     if (temp!=choice)free(temp);
     }
free(h); free(choice);
if (poshead->count)buildtree(&((*tree)->pos),poshead);
if (neghead->count)buildtree(&((*tree)->neg),neghead);
}

/* Traverse this tree in back to front order based on eye position. */
back_to_front(tree)
node *tree;
{
face *f;

if (tree)
     {
     f = tree->f;
     /* If eye is on positive side of this face, then recursively traverse
        negative subtree, draw this face, and recursively traverse
        positive subtree. */
     if ((f->ni*ex + f->nj*ey + f->nk*ez + f->d)>0.0)
     {
     back_to_front(tree->neg);
     draw(f);
     back_to_front(tree->pos);
     }
     /* If eye is on negative side of this face, then recursively traverse
        positive subtree, draw this face, and recursively traverse
        negative subtree. */
     else
     {
     back_to_front(tree->pos);
     draw(f);
     back_to_front(tree->neg);
     }
```

```
    }
}

/* Get eye position then traverse entire binary tree. */
traverse()
{
if (!treebuilt && input)build();
if (input)fprintf(stderr,"In: %d Out: %d Ratio: %.2f\n",input,total,(float)total/(float)input);
if (!input)fprintf(stderr,"%d polygons.\n",total);
get_eye(&ex,&ey,&ez);
back_to_front(root);
}

/* Free up space used by entire binary tree */
free_all()
{
free_tree(root);
root = NULL;
input = 0; total = 0;
}

/* Free up space used by given binary tree */
static free_tree(t)
node *t;
{
if (t)
    {
    free_tree(t->pos);
    free_tree(t->neg);
    free(t->f);
    free(t);
    }
}

/* Dump tree information in ASCII */
adump_data()
{
long t;
char s[132];

fprintf(stderr,"Enter ascii tree file name: ");
scanf("%s",s);
getchar();
fd = fopen(s,"w");
if (fd==NULL)
    {
    fprintf(stderr,"Cannot write file\n");
    return;
    }
t = time(0);
fprintf(fd,"; Created by AGP Version %.1fe on %s",VERSION,ctime(&t));
fprintf(fd,"; This file contains %d polygons, created from %d polygons.\n",total,input);
do
    {
```

```
/* hidden.c */


      fprintf(stderr,"Enter comment: ");
      if (gets(s)&&s[0])fprintf(fd,"; %s\n",s);
      }
while (s[0]);
aout_tree(root);
fclose(fd);
}

/* Dump tree information to file */
static aout_tree(tree)
node *tree;
{
int i;
float w;

if (tree)
      {
      fprintf(fd,"%d %d\n",(tree->pos ? 1:0),(tree->neg ? 1:0));
      fprintf(fd,"%d %d %d %d\n",tree->f->num,tree->f->red,tree->f->[illegible]
      fprintf(fd,"%f %f %f %f\n",tree->f->ni,tree->f->nj,tree->f->nk,tree->f->d);
      for (i=0; i<tree->f->num; i++)
      {
      w = tree->f->vlist[i].w;
      fprintf(fd,"%f %f %f\n",tree->f->vlist[i].x/w,tree->f->vlist[i].y/w,
                                    tree->f->vlist[i].z/w);
      }
      aout_tree(tree->neg); aout_tree(tree->pos);
      }
}

/* Read in a tree stored in ASCII */
aread_data()
{
char s[70];

fprintf(stderr,"Enter ascii tree file name: ");
scanf("%s",s);
getchar();
fd = fopen(s,"r");
if (fd==NULL)
      {
      fprintf(stderr,"Cannot open file\n");
      return;
      }
free_all();
ain_tree(&root);
close(fd);
}

/* Read in ASCII tree information */
static ain_tree(t)
node **t;
{
int pos,neg;
```

```
/* hidden.c */


int red,green,blue,bred,bgreen,bblue,nverts,i;
face *f;
char s[132];

reads(s);
sscanf(s,"%d %d",&pos,&neg);
reads(s);
sscanf(s,"%d %d %d %d",&nverts,&red,&green,&blue,&bred,&bgreen,&bblue);
f = (face *)malloc(sizeof(face)+nverts*sizeof(point));
f->red = red; f->green = green; f->blue = blue;
f->bred = bred; f->bgreen = bgreen; f->bblue = bblue;
total++;
reads(s);
sscanf(s,"%f %f %f %f",&f->ni,&f->nj,&f->nk,&f->d);
f->num = nverts;
for (i=0; i<nverts; i++)
    {
    reads(s);
    sscanf(s,"%f %f %f",&f->vlist[i].x,&f->vlist[i].y,&f->vlist[i].z);
    f->vlist[i].w = 1.0;
    }
*t = (node *)malloc(sizeof(node));
(*t)->f = f;
(*t)->pos = NULL;
(*t)->neg = NULL;
if (neg)ain_tree(&(*t)->neg);
if (pos)ain_tree(&(*t)->pos);
}

/* Dump tree information in binary to a file */
dump_data()
{
char s[132];

if (!treebuilt)build();
fprintf(stderr,"Enter binary tree file name: ");
scanf("%s",s);
getchar();
file = creat(s,0644);
if (file==-1)
    {
    fprintf(stderr,"Cannot write file\n");
    dump_data();
    return;
    }
out_tree(root);
close(file);
}

/* Output binary tree information to file */
static out_tree(tree)
node *tree;
{
int i;
```

```
if (tree)
      {
      write(file,&tree->pos,sizeof(tree->pos));
      write(file,&tree->neg,sizeof(tree->neg));
      i = sizeof(face)+tree->f->num*sizeof(point);
      write(file,&i,sizeof(i));
      write(file,tree->f,i);
      out_tree(tree->pos); out_tree(tree->neg);
      }
}

/* Read binary tree information from file */
read_data()
{
char s[70];

fprintf(stderr,"Enter binary tree file name: ");
scanf("%s",s);
getchar();
file = open(s,0);
if (file==-1)
      {
      fprintf(stderr,"Cannot open file\n");
      read_data();
      return;
      }
free_all();
in_tree(&root);
close(file);
}

/* Read in tree information */
static in_tree(t)
node **t;
{
node *pos,*neg;
int i;
face *f;
char s[132];

read(file,&pos,sizeof(pos));
read(file,&neg,sizeof(neg));
read(file,&i,sizeof(i));
f = (face *)malloc(i);
read(file,f,i);
total++;
*t = (node *)malloc(sizeof(node));
(*t)->f = f;
(*t)->pos = NULL;
(*t)->neg = NULL;
if (pos)in_tree(&(*t)->pos);
if (neg)in_tree(&(*t)->neg);
}
```

```
/* hidden.c */


static reads(s)
char *s;
{
do fgets(s,132,fd); while (s[0]==';');
}
```

```
/* movie.c */

/*
#include <gp.h>
*/
#include "gp.h"
#include <stdio.h>
#include <math.h>

/* This file contains the routines for interactively specifying a
   view of the current scene.  Once the necessary information is
   specified it calls the hidden surface module to traverse the BSP tree. */

#define EPSILON .000001

/* Creates matrix which converts to eye's coordinate system and performs
   the perspective transformation.  Note:  this routine always orients
   the object so that positive Z is up.  This may cause problems if one
   attempts to use it for animation. */
view(ex,ey,ez,cx,cy,cz,scl)
float ex,ey,ez,cx,cy,cz,scl;
{
transform t;
float phi,theta,xdiff,ydiff,zdiff,dist;

xdiff = ex-cx; ydiff = ey-cy; zdiff = ez-cz;
idn_transform(&t);
set_m(&t);
translate(-ex,-ey,-ez);
if ((fabs(xdiff)>EPSILON)||(fabs(ydiff)>EPSILON))
     {
     rotate(XAXIS,PI/2.0);
     dist = sqrt(xdiff*xdiff + ydiff*ydiff);
     if (ydiff < 0)theta = asin(-xdiff/dist);
     else theta = asin(xdiff/dist) + PI;
     rotate(YAXIS,-theta);
     phi = (float) atan((double) (zdiff/dist));
     rotate(XAXIS,-phi);
     }
else rotate(XAXIS,PI);

/* save pre-perspective transformation */
save_preper();

scale(scl,scl,-1.0);
per_transform((float) atan((double)0.25),0.0,0.0,&t);
concatenate(&t);
set_camera();
idn_transform(&t); set_m(&t);
}

/* Ask user for view information */
movie()
{
```

```
/* movie.c */

float ex,ey,ez,cx,cy,cz,lx,ly,lz,scl;
float a,d,sp;
char s[128];
int r,g,b;

scl = 1.0; cx = cy = cz = 0.0;
ex = 100.0; ey = 150; ez = 200;
r = -1; g = b = 255;
a = 20.0; sp = 0.0; d = 80.0;
for (;;)
    {
    fprintf(stderr,”Viewpoint [%.2f %.2f %.2f]: ”,ex,ey,ez);
    if (gets(s)&&s[0])
    sscanf(s,”%f %f %f”,&ex,&ey,&ez);
    fprintf(stderr,”Center point [%.2f %.2f %.2f]: ”,cx,cy,cz);
    if (gets(s)&&s[0])
    sscanf(s,”%f %f %f”,&cx,&cy,&cz);
    if ((ex==cx)&&(ey==cy)&&(ez==cz))return;
    fprintf(stderr,”Scale factor [%.2f]: ”,scl);
    if (gets(s)&&s[0])
    sscanf(s,”%f”,&scl);
    lx = ex; ly = ey; lz = ez;
#ifdef ERICSTUFF
    fprintf(stderr,”Light position [%.2f %.2f %.2f]: ”,lx,ly,lz);
    if (gets(s)&&s[0])
    sscanf(s,”%f %f %f”,&lx,&ly,&lz);
    fprintf(stderr,”Ambient, diffuse, specular [%d %d %d]: ”,(int)a,(int)d,(int)sp);
    if (gets(s)&&s[0])
    sscanf(s,”%f %f %f”,&a,&d,&sp);
#endif
    set_ambient(a); set_diffuse(d); set_specular(sp);
    fprintf(stderr,”Enter background color [”);
    if (r>=0)fprintf(stderr,”%d %d %d”,r,g,b); else fprintf(stderr,”Don't erase”);
    fprintf(stderr,”]: ”);
    if (gets(s)&&s[0])
    sscanf(s,”%d %d %d”,&r,&g,&b);
    if (r>=0)clear(r,g,b);
    set_eyepoint(ex,ey,ez);
    light_vector(lx-cx,ly-cy,lz-cz);
    view(ex,ey,ez,cx,cy,cz,scl);
    traverse();
#ifdef IKONAS
    ikflush();
#endif
#ifdef RAMTEK
    ramflush();
#endif
    }
}
```

```
/* normltbl.c */

/* maintain prototype normal table */
#include <stdio.h>
#include "/unc/dh/surf2/prodesc.h"
#include "normal.h"

double normalp (i, j)
     register int i,j;
{
if ((i>=1) && (i<=TOTALPTS+1)) return (np[i][j]);
else return (0.0);  /* ???? */
}

normlinit () {
FILE *nfile, *fopen();
int tblsize;
int ptnum;
double      x,
     y,
     z;
register int i;

if (NULL == (nfile = fopen (NORMLTBL, "r")))
              fprintf (stderr, "normal table open error.\n");
fscanf (nfile, "%d", &tblsize);
if (tblsize != TOTALPTS+1) {
     fprintf (stderr, "normal table size error.\n");
     return;
     }
for (i=1; i<=TOTALPTS+1; i++) {
     fscanf (nfile, "%d %F %F %F", &ptnum, &x, &y, &z);
     if (i != ptnum) {
              fprintf (stderr, "normal table read error at ptnum = %d.\n",
                             ptnum);
              return;
              }
     np[ptnum][0] = x;
     np[ptnum][1] = y;
     np[ptnum][2] = z;

#ifdef DEBUG
     fprintf (stderr, "%f %f %f\n", normalp(ptnum,0), normalp(ptnum,1),
              normalp(ptnum,2));
#endif
     }
}
```

```
/* shade.c */

/*
#include <gp.h>
*/
#include "gp.h"
#include <math.h>
#include <ikdefs.h>
#include <stdio.h>

#define NEWPOLY

/* Routines associated with shading and scan conversion */
/*   Modified to store color and normal info. into frame buffer (instead */
/*   of intensity data) in appropriate format for dynamic lighting      */
/*   (Only compute_color is actually modified.)        - dh   8/21/82              */

#define EPS 0.001
extern face *clippoly();

typedef struct
     {
     float ni,nj,nk,dni,dnj,dnk;
     int x,ystart,endy,next;
     float fx,dx;
     float fx1,fy1;
     int slope;
     } edge;

#define UP 1
#define DOWN -1

static float red,green,blue;              /* The current object color */
static float bred,bgreen,bblue;           /* Back side of current object */
static float li,lj,lk;                    /* Vector to light source */
static float ei,ej,ek;                    /* Vector to eye point */
static float hi,hj,hk;                    /* Direction of max highlight */
static float ex,ey,ez;                    /* Eye position */

/* Note: ambient plus diffuse should equal 100.  specular should be 0-100. */
static float ambient,                        /* Ambient light in scene */
          diffuse,                      /* Diffuse reflection coefficient */
          specular;                     /* Specular reflection coefficient */

static boolean smooth;                       /* Whether or not to smooth shade */
static boolean backfacing = FALSE;  /* Draw backfacing polygons? */

init_shade()
{
#ifdef NEWPOLY
ikup();
#endif
}
```

```
/* shade.c */


/* Normalize a vector to unit length */
normalize(x,y,z)
float *x,*y,*z;
{
float size;
size = (float) sqrt((double) (*x * (*x) + *y * (*y) + *z * (*z)));
if (fabs(size)<.00001)size = 1.0;
*x = *x/size; *y = *y/size; *z = *z/size;
}

/* Turn on/off smooth shading */
set_smooth(val)
boolean val;
{
smooth = val;
}

/* Turn on/off display of backfacing polygons */
set_backfacing(val)
boolean val;
{
backfacing = val;
}

/* Set face normal and plane equation */
static set_normal(f)
face *f;
{
int i,sub;

f->ni = f->nj = f->nk = 0.0;
for (i=0; i<f->num; i++)
     {
     sub = (i==(f->num-1) ? 0 : i+1);
     f->ni = f->ni + (f->vlist[sub].y-f->vlist[i].y)*(f->vlist[sub].z+f->vlist[i].z);
     f->nj = f->nj + (f->vlist[sub].z-f->vlist[i].z)*(f->vlist[sub].x+f->vlist[i].x);
     f->nk = f->nk + (f->vlist[sub].x-f->vlist[i].x)*(f->vlist[sub].y+f->vlist[i].y);
     }
normalize(&f->ni,&f->nj,&f->nk);
f->d = -(f->ni*f->vlist[0].x + f->nj*f->vlist[0].y + f->nk*f->vlist[0].z);
}

/* Set coefficient for amount of ambient light in scene */
set_ambient(percent)
float percent;
{
ambient = percent;
}

/* Set coefficient of diffuse reflection */
set_diffuse(percent)
float percent;
{
diffuse = percent;
```

```
/* shade.c */

}

/* Set coefficient of specular reflection */
set_specular(percent)
float percent;
{
specular = percent;
}

/* Set the vector to light source */
light_vector(x,y,z)
float x,y,z;
{
li = x; lj = y; lk = z;
normalize(&li,&lj,&lk);
}

/* Set the source color of subsequent objects */
set_color(r,g,b)
int r,g,b;
{
red = (float) r;
green = (float) g;
blue = (float) b;
}

/* Set color of back side of subsequent objects */
set_bcolor(r,g,b)
int r,g,b;
{
bred = (float) r;
bgreen = (float) g;
bblue = (float) b;
}

/* Set the eyepoint (a.k.a. viewpoint, camera) */
set_eyepoint(x,y,z)
float x,y,z;
{
ex = x; ey = y; ez = z;
}

/* Return eye position */
get_eye(x,y,z)
float *x,*y,*z;
{
*x = ex; *y = ey; *z = ez;
}

/* Set light vector given light position and center point */
set_light_source(lx,ly,lz,cx,cy,cz)
float lx,ly,lz,cx,cy,cz;
{
light_vector(lx-cx,ly-cy,lz-cz);
```

```
}

/* Return true if face is front-facing based on current eye position */
static boolean visible(f)
face *f;
{
if ((f->ni*(ex-f->vlist[0].x) + f->nj*(ey-f->vlist[0].y) +
      f->nk*(ez-f->vlist[0].z))<0)return(FALSE);
else return(TRUE);
}

/* Set the vector from given point to eye position */
eye_vector(x,y,z)
float x,y,z;
{
ei = ex-x; ej = ey-y; ek = ez-z;
normalize(&ei,&ej,&ek);
hi = ei+li; hj = ej+lj; hk = ek+lk;
normalize(&hi,&hj,&hk);
}

/* Return val to the nth power */
static float power(val,n)
float val;
float n;
{
float result;

if (val<=0)return(0.0);
result = (float) exp((double) (n*log((double)val)));
return(result);
}


/* Compute the highlight coefficient based on given surface normal */
static float compute_highlight(ni,nj,nk)
float ni,nj,nk;
{
float dot;

/* Take dot product of vector of maximum highlight and surface normal */
dot = hi*ni + hj*nj + hk*nk;
/* Return coefficient raised to 30th power to assure rapid drop off */
return(power(dot,30.0));
}

/* Compute the appropriate format (in the frame buffer) for the color */
/* and the supplied surface normal in order to do dynamic lighting    */
/*                                                  - dh  8/21/82     */
static compute_color(ni,nj,nk,r,g,b)
float ni,nj,nk;      /* supplied surface normal */
int *r,*g,*b;        /* color specified */
{
#include "/unc/dh/surf2/protoargs.h"        /* where TOTALPTS is defined */
```

```
/* shade.c */

#define CUTOFF1 42   /* cutoff of desired intensities at 1/6 of max. */
#define CUTOFF2 128  /* cutoff of desired intensities at 3/6 of max. */
#define CUTOFF3 213  /* cutoff of desired intensities at 5/6 of max. */

int ptnum,   /* loop index - coded number for a normal */
    closest;        /* coded number for "ideal" normal that is closest */
                    /* to given normal                                  */
double      dotprod,
    maxdotprod;     /* current max. calculated dot product */
double      normalp();      /* returns 1 component of 1 "ideal" normal */

float tnx, tny, tnz, tnw;
float tox, toy, toz, tow;

/* Transform normals using all transformations but scaling to screen */
/*  coordinates & perspective                                         */
vec1 (ni, nj, nk, 1.0, &tnx, &tny, &tnz, &tnw);      /* transform normal endpt. */
vec1 (0.0, 0.0, 0.0, 1.0, &tox, &toy, &toz, &tow);  /* transform origin */
tnx = tnx-tox;
tny = tny-toy;
tnz = tnz-toz;

/* Calculate the "ideal" normal that is closest to the given surface normal */
/* and store its number in the red byte                                      */
closest = 1;
maxdotprod = tnx*normalp(1,0) + tny*normalp(1,1) + tnz*normalp(1,2);
#ifdef DBGDOTP
fprintf (stdout, "%f %f %f, %f %f %f, %f\n", tnx, tny, tnz, normalp(1,0),
            normalp(1,1), normalp(1,2), maxdotprod);
#endif
for (ptnum=2; ptnum<=TOTALPTS+1; ptnum++) {
    dotprod = tnx*normalp(ptnum,0) + tny*normalp(ptnum,1)
                    + tnz*normalp(ptnum,2);
#ifdef DBGDOTP
fprintf (stdout, "%f %f %f, %f %f %f, %f\n", tnx, tny, tnz, normalp(ptnum,0),
            normalp(ptnum,1), normalp(ptnum,2), dotprod);
#endif
    if (dotprod > maxdotprod) {
            maxdotprod = dotprod;
            closest = ptnum;
            }
    }
#ifdef DBGDOTP2
fprintf (stdout, "%f %f %f, %f %f %f, %f\n", tnx, tny, tnz, normalp(closest,0),
            normalp(closest,1), normalp(closest,2), maxdotprod);
#endif
*r = closest;

/* Re-format the color and store it in the green byte */
if (red < CUTOFF1) *g = 0;
else if (red < CUTOFF2) *g = 1;
else if (red < CUTOFF3) *g = 2;
else *g = 3;
if (green < CUTOFF1);
```

```
else if (green < CUTOFF2) *g = *g + 4;
else if (green < CUTOFF3) *g = *g + 8;
else *g = *g + 12;
if (blue < CUTOFF1);
else if (blue < CUTOFF2) *g = *g + 16;
else if (blue < CUTOFF3) *g = *g + 32;
else *g = *g + 48;

/* The blue byte is unused - set it to zeros */
*b = 0;

#ifdef ERICSTUFF
float hfactor,factor,cosine;
inquiry_response v;

show_inquire(&v);
/* Get highlight factor */
hfactor = compute_highlight(ni,nj,nk)*specular*((float)v.imax)/100.0;
/* Cosine of angle between surface normal and vector to light source */
cosine = ni*li + nj*lj + nk*lk;
if (cosine<0)cosine = 0;
/* Compute diffuse reflection factor */
factor = (ambient+cosine*diffuse)/100.0;
/* Compute colors */
*r = (int) (red*factor + hfactor);
*g = (int) (green*factor + hfactor);
*b = (int) (blue*factor + hfactor);
if (*r > v.imax)*r = v.imax;
if (*g > v.imax)*g = v.imax;
if (*b > v.imax)*b = v.imax;
#endif
}

/* Set face information and enter face into BSP tree */
poly_write(f)
face *f;
{
if (!smooth)f->smooth = FALSE;
set_normal(f);
f->red = red; f->green = green; f->blue = blue;
f->bred = bred; f->bgreen = bgreen; f->bblue = bblue;
tree_enter(f);
return(0);
}

/* Same as poly_write, but draws face instead of entering into BSP tree */
display(f)
face *f;
{
if (!smooth)f->smooth = FALSE;
set_normal(f);
f->red = red; f->green = green; f->blue = blue;
f->bred = bred; f->bgreen = bgreen; f->bblue = bblue;
draw(f);
```

```
}

/* Draw face on display */
draw(f)
face *f;
{
int r,g,b,i;
face *temp;

/* Set vector from this polygon to eye point */
eye_vector(f->vlist[0].x,f->vlist[0].y,f->vlist[0].z);

if (!visible(f))
      {
      if (!backfacing)return;
      set_color(f->bred,f->bgreen,f->bblue);
      compute_color(-f->ni,-f->nj,-f->nk,&r,&g,&b);
      }
else
      {
      set_color(f->red,f->green,f->blue);
      compute_color(f->ni,f->nj,f->nk,&r,&g,&b);
      }

/* Convert polygon to screen coordinates */
cam_mode();
for (i=0; i<f->num; i++)
{
      if (to_screen(f->vlist[i].x,f->vlist[i].y,f->vlist[i].z,f->vlist[i].w,&f->vlist[i].sx,&f->vlist[i].sy,&f->
      {
      end_mode();
      return;
      }
}
end_mode();

/* Clip to display window */
temp = clippoly(f);
/* Scan convert to display */
po_write(r,g,b,temp);
if (temp)free(temp);
return(0);
}

int round(x)
float x;
{ int ans = x<0 ? (-x)+0.5:x+0.5;
  return(x<0 ? -ans:ans);
}

#ifdef OLDPOLY
#ifdef SMOOTH
/* Smooth shade a scan_line based on edge information supplied */
static scan_line(y,e1,e2)
```

```
/* shade.c */

int y;
edge e1,e2;
{
int i,r,g,b;
float l_dot_v1,l_dot_v2,v1_dot_v2,h_dot_v1,h_dot_v2,dx;
char rbuf[512],gbuf[512],bbuf[512];
float fact,fact1,fact2,fact3,factor,hfactor,arg,root;
float a,da,one_minus_a;
inquiry_response v;

normalize(&e1.ni,&e1.nj,&e1.nk); normalize(&e2.ni,&e2.nj,&e2.nk);
l_dot_v1 = li*e1.ni + lj*e1.nj + lk*e1.nk;
l_dot_v2 = li*e2.ni + lj*e2.nj + lk*e2.nk;
v1_dot_v2 = e1.ni*e2.ni + e1.nj*e2.nj + e1.nk*e2.nk;
h_dot_v1 = hi*e1.ni + hj*e1.nj + hk*e1.nk;
h_dot_v2 = hi*e2.ni + hj*e2.nj + hk*e2.nk;
dx = (float) (e2.x - e1.x);
a = 1.0;
da = (dx==0.0 ? 0.0 : 1.0/dx);
fact = 2.0*(1.0-v1_dot_v2);
show_inquire(&v);
for (i=e1.x; i<=e2.x; i++)
     {
     one_minus_a = 1.0-a;
     arg = fact*a*a - fact*a + 1.0;
     root = (float)sqrt((double)arg);
     fact1 = (a*l_dot_v1 + one_minus_a*l_dot_v2)/root;
     if (fact1<0.0)fact1 = 0.0;
     fact2 = (a*h_dot_v1 + one_minus_a*h_dot_v2)/root;
     fact3 = power(fact2,30.0);
     factor = (ambient+fact1*diffuse)/100.0;
     hfactor = fact3*specular*((float)v.imax)/100.0;
     r = (int) (red*factor + hfactor);
     g = (int) (green*factor + hfactor);
     b = (int) (blue*factor + hfactor);
     if (r>v.imax)r=v.imax;
     if (g>v.imax)g=v.imax;
     if (b>v.imax)b=v.imax;
     rbuf[i] = r; gbuf[i] = g; bbuf[i] = b;
     a = a - da;
     }
segment(y,e1.x,e2.x,rbuf,gbuf,bbuf);
}
#endif

/* Scan conversion routine. This routine is based on a scan conversion
   routine written by Bob Hon. Instead of rounding vertices to the nearest
   integer, however, it carries through floating point values to assure
   "exact" scan conversion. */
po_write(r,g,b,f)
int r,g,b;
face *f;
{
  int i,numedge,nvertex;
```

```
/* shade.c */

   point prev;
   edge *edgearray;
   int curr,active,fol,save;
   int ycur;
   int wind;
   edge start;

   nvertex = f->num;
   if ((edgearray = (edge *) calloc(nvertex,sizeof(edge))) == 0) return(-1);

   /* go through vertex list, deal with horizontal edges, figure slopes */
   prev = f->vlist[nvertex-1];
   numedge = 0;
   for (i = 0; i < nvertex; i++)
     {
      int ldy; float fldx,fldy;
      ldy = round(f->vlist[i].sy)-round(prev.sy);
      if (ldy==0)
        {prev = f->vlist[i]; continue;}        /* ignore horizontal lines */
      fldx = f->vlist[i].sx - prev.sx;
      fldy = f->vlist[i].sy - prev.sy;
#ifdef SMOOTH
      if (f->smooth)
     {
     edgearray[numedge].dni = (f->vlist[i].ni-prev.ni)/fldy;
     edgearray[numedge].dnj = (f->vlist[i].nj-prev.nj)/fldy;
     edgearray[numedge].dnk = (f->vlist[i].nk-prev.nk)/fldy;
     }
#endif
      edgearray[numedge].slope = ldy < 0 ? DOWN:UP;
      edgearray[numedge].dx = fldx/fldy;
      if (fldy < 0.0)
        {/* draw from current to prev */
     edgearray[numedge].ystart = round(f->vlist[i].sy);
     edgearray[numedge].endy = round(prev.sy)-1;
     edgearray[numedge].fx1 = prev.sx;
     edgearray[numedge].fx1 = f->vlist[i].sx;
     edgearray[numedge].fy1 = f->vlist[i].sy;
     edgearray[numedge].fx = ((float)edgearray[numedge].ystart+0.5-edgearray[numedge].fy1)*edge
     edgearray[numedge].x = round(edgearray[numedge].fx);
#ifdef SMOOTH
     if (f->smooth)
          {
          edgearray[numedge].ni = f->vlist[i].ni;
          edgearray[numedge].nj = f->vlist[i].nj;
          edgearray[numedge].nk = f->vlist[i].nk;
          }
#endif
        }
      else
        {/* draw from prev to current */
     edgearray[numedge].ystart = round(prev.sy);
     edgearray[numedge].endy = round(f->vlist[i].sy)-1;
     edgearray[numedge].fx1 = prev.sx;
```

/* shade.c */

```
      edgearray[numedge].fy1 = prev.sy;
      edgearray[numedge].fx = ((float)edgearray[numedge].ystart+0.5-edgearray[numedge].fy1)*edge
      edgearray[numedge].x = round(edgearray[numedge].fx);
#ifdef SMOOTH
      if (f->smooth)
            {
            edgearray[numedge].ni = prev.ni;
            edgearray[numedge].nj = prev.nj;
            edgearray[numedge].nk = prev.nk;
            }
#endif
          }
       prev = f->vlist[i];
       numedge++;
      }

   if (numedge<2)
      {
      free(edgearray);
      return(-1);
      }
   /* sort list by y */
   sort(numedge,edgearray);

   curr = 0;
   ycur = edgearray[0].ystart;
   active = -1;
   while (1)
      {
       while (curr < numedge && ycur == edgearray[curr].ystart)
          {active = insert(curr,active,edgearray);       /* insert into active list */
     curr++;
          }
       /* draw lines between intersections */
       fol = active;
       while (fol != -1)
          {
     start = edgearray[fol];
     wind = edgearray[fol].slope == UP ? 1:-1;
     while (wind != 0)
        {
         fol = edgearray[fol].next;
         if (edgearray[fol].slope == UP) wind++; else wind--;
        }
      if (fabs(start.fx-edgearray[fol].fx)>EPS)
          {
          if (!f->smooth)hline(ycur,start.x,edgearray[fol].x,r,g,b);
#ifdef SMOOTH
          else scan_line(ycur,start,edgearray[fol]);
#endif
          }
     fol = edgearray[fol].next;
         }
```

```
/* shade.c */

        fol = active;
        ycur++;                     /* update y location */
        while (fol != -1)        /* do dda */
            {
      edgearray[fol].fx = ((float)ycur+0.5-edgearray[fol].fy1)*edgearray[fol].dx+edgearray[fol].fx1;
      edgearray[fol].x = round(edgearray[fol].fx);
#ifdef SMOOTH
      if (f->smooth)
          {
          edgearray[fol].ni += edgearray[fol].dni;
          edgearray[fol].nj += edgearray[fol].dnj;
          edgearray[fol].nk += edgearray[fol].dnk;
          }
#endif
      fol = edgearray[fol].next;
            }
        /* drop dead edges and resort */
        fol = active; active = -1;
        while (fol != -1)
          {save = edgearray[fol].next;
      if (edgearray[fol].endy < ycur) {fol = save; continue;}
      active = insert(fol,active,edgearray);
      fol = save;
          }
        if (active == -1 && curr >= numedge)
      {
      free(edgearray);
      break;
      }
      }
return(1);
}

static sort(n,array)
int n;
edge array[];
{
  int i,j,nsorted;
  edge current;

  nsorted = 1;

  while (nsorted < n)
    {
     current = array[nsorted];
     for (i = 0; i < nsorted; i++)
        {
    if (array[i].ystart <= current.ystart) continue;
    /* found where it goes */
    for (j = nsorted; j >i; j--) array[j] = array[j-1];
    array[i] = current;
    break;
        }
     nsorted++;
```

```
    }
}

static insert(index,head,array)
int index,head;
edge array[];
{
  int prev,curr;
  if (head == -1)
    {array[index].next = -1;
     return(index); /* new head of list */
    }
  prev = -1;  curr = head;
  while (curr != -1 && array[index].x > array[curr].x)
    {prev = curr; curr = array[curr].next;}
  array[index].next = curr;
  if (prev == -1) return(index);
  else array[prev].next = index;
  return(head);
}
#endif


#ifdef NEWPOLY


/* this file contains three routines used in painting */
/* polygons on the ikonas using the bsp tree algorithm */

#define SP(a)(0202<<16|(a))
#define RECIP SP(50)
#define MCSI SP(0)
#define LIST SP(600)
#define PFILL 10
#define BUFFSIZE 3300
#define WAIT 3500
#define SIGNAL 3600
#define FLAG SP(599)

int poly[BUFFSIZE], next=0, flag;

/* program to load bmp and initialize ikonas */
ikup()
{
     int i, recip[512];
     static int mcsi[]={WAIT,FLAG,0,PFILL,LIST,SIGNAL,FLAG,-1,0};
     extern int flag;
     int zero;

     /* reset ikonas, load default color map */
     if (127==system("/usr/ikonas/ikreset")) abort();

     /* leave ikonas at 30hz */
     if (127==system("/usr/ikonas/ikset 30hz")) abort();
```

```
/* shade.c */


    /* load mcsi and polygon fill routine into bmp */
    if (127==system("/usr/ikonas/ril /unc/eric/lib/mcsi.obj /unc/eric/lib/apfsilgobj"))na/berit()lib/v

    /* initialize reciprocal table */
    recip[0] = 0;
    recip[1] = 0x7fff;
    for(i=2; i<512; i++)
            if (i <= 128) recip[i] = ((float)(1<<15))/i + 0.5;
            else if (i <= 256) recip[i] = ((float)(1<<16))/i + 0.5;
            else recip[i] = ((float)(1<<23))/i + 0.5;

    ikwrite(recip, sizeof(recip), RECIP);

    ikwrite(mcsi, sizeof(mcsi), MCSI);

    flag= -1;
    ikwrite(&flag, sizeof(flag), FLAG);
    setcmask(IKRESET|RUNPROC);
    ikwrite(&zero,0,0);
    setcmask(RUNPROC);
    ikwrite(&zero,0,0);

    }        /* end of ikup */

/* routine to accept a polygon and buffer it. */
/* on buffer overflow, write buffer to ikonas */

po_write(r,g,b,f)
int r,g,b;
face *f;
{
    extern int poly[], next;
    int j,jind;

    /* if this polygon would overflow buffer, then write buffer */

#ifdef DEBUG2
    fprintf(stderr, "in ikfastfill.  pgon.numpoints is: %d\n",
            pgon.numpoints);
#endif

    if (f->num + next > BUFFSIZE/2) {
            if (f->num > BUFFSIZE/2) abort();
#ifdef DEBUG2
    fprintf(stderr, "before calling ikflush from ikfastfill\n");
#endif
            ikflush();
            }

    for(j=0; j<f->num; j++) {
            jind = f->num - j - 1;
            poly[next+2*jind+1]=0;
            poly[next+2*jind] = (511-round(f->vlist[j].sy))<<16|round(f->vlist[j].sx);
```

```
                if (jind==0) poly[next+2*jind+1] = r<<16;
                if (jind==1) poly[next+2*jind+1] = g<<16;
                if (jind==2) poly[next+2*jind+1] = b<<16;
#ifdef DEBUG2
    fprintf(stderr, "x is %d, y is %d, and next is %d\n",
                pgon.points[j].x,pgon.points[j].y,next);
#endif
                }

    next += f->num * 2;
    poly[next-1] |= 1<<30;

    }           /* end of po_write */

/* routine to flush buffer to ikonas */
ikflush()
{
    extern int poly[], next;
    int zero=0;
    extern int flag;

#ifdef DEBUG2
    fprintf(stderr, "just entered ikflush\n");
    fprintf(stderr, "next = %d\n",next);
#endif
    if (next != 0) {
                poly[next-1] |= 1<<31;  /* set next poly flag */
#ifdef DEBUG2
    fprintf(stderr, "before ikwrite\n");
#endif
                do ikread(&flag, sizeof(flag), FLAG);
                        while (flag == 0);      /*  wait for BMP to be done */
                ikwrite(poly, next*sizeof(int), LIST);
                ikwrite(&zero, sizeof(zero), FLAG);
                next=0;
                }
    }
#endif
```

```
/* shapes.c */

/*
#include <gp.h>
*/
#include "gp.h"
#include <math.h>

/* This file contains assorted primitive shapes */

extern char *malloc();

/* A plate with four specified vertices */
plate(x1,y1,z1,x2,y2,z2,x3,y3,z3,x4,y4,z4)
float x1,y1,z1,x2,y2,z2,x3,y3,z3,x4,y4,z4;
{
face *f;

f = (face *)malloc(sizeof(face)+4*sizeof(point));
f->smooth = FALSE;
f->num = 4;
trans_point(x1,y1,z1,1.0,&f->vlist[0].x,&f->vlist[0].y,&f->vlist[0].z,&f->vlist[0].w);
trans_point(x2,y2,z2,1.0,&f->vlist[1].x,&f->vlist[1].y,&f->vlist[1].z,&f->vlist[1].w);
trans_point(x3,y3,z3,1.0,&f->vlist[2].x,&f->vlist[2].y,&f->vlist[2].z,&f->vlist[2].w);
trans_point(x4,y4,z4,1.0,&f->vlist[3].x,&f->vlist[3].y,&f->vlist[3].z,&f->vlist[3].w);
poly_write(f);
}

/* A curved plate given four specified vertices and vertex normals */
cplate(x1,y1,z1,x2,y2,z2,x3,y3,z3,x4,y4,z4,
       ni1,nj1,nk1,ni2,nj2,nk2,ni3,nj3,nk3,ni4,nj4,nk4)
float x1,y1,z1,x2,y2,z2,x3,y3,z3,x4,y4,z4;
float ni1,nj1,nk1,ni2,nj2,nk2,ni3,nj3,nk3,ni4,nj4,nk4;
{
face *f;
float x,y,z,w;
float xp,yp,zp,wp;

f = (face *)malloc(sizeof(face)+4*sizeof(point));
f->smooth = TRUE;
f->num = 4;
trans_point(x1,y1,z1,1.0,&f->vlist[0].x,&f->vlist[0].y,&f->vlist[0].z,&f->vlist[0].w);
#ifdef SMOOTH
trans_point(ni1,nj1,nk1,1.0,&x,&y,&z,&w);
trans_point(0.0,0.0,0.0,1.0,&xp,&yp,&zp,&wp);
f->vlist[0].ni = x/w - xp/wp;
f->vlist[0].nj = y/w - yp/wp;
f->vlist[0].nk = z/w - zp/wp;
normalize(&f->vlist[0].ni,&f->vlist[0].nj,&f->vlist[0].nk);
#endif
trans_point(x2,y2,z2,1.0,&f->vlist[1].x,&f->vlist[1].y,&f->vlist[1].z,&f->vlist[1].w);
#ifdef SMOOTH
trans_point(ni2,nj2,nk2,1.0,&x,&y,&z,&w);
trans_point(0.0,0.0,0.0,1.0,&xp,&yp,&zp,&wp);
```

```
f->vlist[1].ni = x/w - xp/wp;
f->vlist[1].nj = y/w - yp/wp;
f->vlist[1].nk = z/w - zp/wp;
normalize(&f->vlist[1].ni,&f->vlist[1].nj,&f->vlist[1].nk);
#endif
trans_point(x3,y3,z3,1.0,&f->vlist[2].x,&f->vlist[2].y,&f->vlist[2].z,&f->vlist[2].w);
#ifdef SMOOTH
trans_point(ni3,nj3,nk3,1.0,&x,&y,&z,&w);
trans_point(0.0,0.0,0.0,1.0,&xp,&yp,&zp,&wp);
f->vlist[2].ni = x/w - xp/wp;
f->vlist[2].nj = y/w - yp/wp;
f->vlist[2].nk = z/w - zp/wp;
normalize(&f->vlist[2].ni,&f->vlist[2].nj,&f->vlist[2].nk);
#endif
trans_point(x4,y4,z4,1.0,&f->vlist[3].x,&f->vlist[3].y,&f->vlist[3].z,&f->vlist[3].w);
#ifdef SMOOTH
trans_point(ni4,nj4,nk4,1.0,&x,&y,&z,&w);
trans_point(0.0,0.0,0.0,1.0,&xp,&yp,&zp,&wp);
f->vlist[3].ni = x/w - xp/wp;
f->vlist[3].nj = y/w - yp/wp;
f->vlist[3].nk = z/w - zp/wp;
normalize(&f->vlist[3].ni,&f->vlist[3].nj,&f->vlist[3].nk);
#endif
poly_write(f);
}

/* An approximation to a circle with specified radius and number of sides */
/* Circle is drawn in XY plane with center at (0.0,0.0) */
circle(radius,nsides)
float radius;
int nsides;
{
float theta,incr;
face *circ;
int i;

circ = (face *)malloc(sizeof(face) + (nsides+1)*sizeof(point));
circ->smooth = FALSE;
circ->num = nsides+1;
incr = 2.0*PI/(float)nsides;
theta = 0.0;
for (i=0; i<=nsides; i++)
    {
    trans_point(radius*(float)cos(theta),radius*(float)sin(theta),0.0,1.0,
            &circ->vlist[i].x,&circ->vlist[i].y,&circ->vlist[i].z,&circ->vlist[i].w);
    theta = theta-incr;
    }
poly_write(circ);
}

/* An approximation to a cylinder with specified radius, height, and # of sides */
/* Cylinder is drawn centered on ZAXIS with base at Z=0 and top at Z=height */
cylinder(radius,height,nsides)
float radius,height;
```

```
/* shapes.c */


int nsides;
{
double theta,theta2;
float incr,x1,y1,x2,y2;
int i;

incr = 2.0*PI/(float)nsides;
theta = 0.0;
for (i=0; i<nsides; i++)
     {
     theta2 = theta+incr;
     x1 = radius*(float)cos(theta);
     y1 = radius*(float)sin(theta);
     x2 = radius*(float)cos(theta2);
     y2 = radius*(float)sin(theta2);
     theta = theta2;
     cplate(x1,y1,0.0,x1,y1,height,x2,y2,height,x2,y2,0.0,
               x1,y1,0.0,x1,y1,0.0,x2,y2,0.0,x2,y2,0.0);
     }
push_m(); translate(0.0,0.0,height); circle(radius,nsides); pop_m();
push_m(); rotate(XAXIS,PI); circle(radius,nsides); pop_m();
}

/* An approximation to a sphere with specified radius and # of sides per strip */
/* Sphere is drawn with center at the origin */
sphere(radius,n)
float radius;
int n;
{
double theta,theta2,phi,phi2;
float incr,x1,y1,x2,y2,x3,y3,x4,y4,z1,z2;
int i,j;

if (n%2 == 1)n++;
incr = 2.0*PI/(float)n;
phi = 0.0;
for (i=0; i<n/2; i++)
     {
     phi2 = phi+incr;
     z1 = (float)cos(phi)*radius;
     z2 = (float)cos(phi2)*radius;
     theta = 0.0;
     for (j=0; j<n; j++)
     {
     theta2 = theta+incr;
     x1 = (float) (cos(theta)*sin(phi)*radius);
     y1 = (float) (sin(theta)*sin(phi)*radius);
     x2 = (float) (cos(theta)*sin(phi2)*radius);
     y2 = (float) (sin(theta)*sin(phi2)*radius);
     x3 = (float) (cos(theta2)*sin(phi)*radius);
     y3 = (float) (sin(theta2)*sin(phi)*radius);
     x4 = (float) (cos(theta2)*sin(phi2)*radius);
     y4 = (float) (sin(theta2)*sin(phi2)*radius);
     theta = theta2;
```

```
/* shapes.c */

      cplate(x3,y3,z1,x4,y4,z2,x2,y2,z2,x1,y1,z1,
             x3,y3,z1,x4,y4,z2,x2,y2,z2,x1,y1,z1);
      }
      phi = phi2;
      }
}

/* Draws an outline of a rectilinear solid */
outline(x,y,z)
float x,y,z;
{
vec_mode();
move_to(0.0,0.0,0.0);
line_to(x,0.0,0.0); line_to(x,0.0,z); line_to(0.0,0.0,z); line_to(0.0,0.0,0.0);
line_to(0.0,y,0.0); line_to(0.0,y,z); line_to(x,y,z); line_to(x,y,0.0);
line_to(0.0,y,0.0);
move_to(0.0,0.0,z); line_to(0.0,y,z);
move_to(x,0.0,0.0); line_to(x,y,0.0);
move_to(x,0.0,z); line_to(x,y,z);
end_mode();
}

/* A rectilinear solid drawn in positive-x, positive-y, positive-z */
rect(x,y,z)
float x,y,z;
{
/* outline(x,y,z); */
plate(0.0,0.0,0.0,x,0.0,0.0,x,y,0.0,0.0,y,0.0);
plate(0.0,0.0,0.0,0.0,0.0,z,x,0.0,z,x,0.0,0.0);
plate(x,0.0,0.0,x,0.0,z,x,y,z,x,y,0.0);
plate(x,y,0.0,x,y,z,0.0,y,z,0.0,y,0.0);
plate(0.0,y,0.0,0.0,y,z,0.0,0.0,z,0.0,0.0,0.0);
plate(0.0,0.0,z,0.0,y,z,x,y,z,x,0.0,z);
}

/* An approximation to a cone with specified radius, height, and # of sides */
/* Cone is draw centered on ZAXIS with base at Z=0 and top at Z=height */
cone(radius,height,n)
float radius,height;
int n;
{
float theta,theta2,x1,y1,x2,y2,incr;
int i;

incr = 2.0*PI/(float) n;
theta = 0.0;
for (i=0; i<n; i++)
     {
     theta2 = theta - incr;
     x1 = radius*(float)cos(theta);
     y1 = radius*(float)sin(theta);
     x2 = radius*(float)cos(theta2);
     y2 = radius*(float)sin(theta2);
     plate(x1,y1,0.0,0.0,0.0,height,0.0,0.0,height,x2,y2,0.0);
```

```
/* shapes.c */

    theta = theta2;
    }
}
```

```
/* gp.h */

#define IKONAS

#define XAXIS 1
#define YAXIS 2
#define ZAXIS 3

#define TRUE -1
#define FALSE 0

#define NULL 0

#define PI 3.141592654

#define SMOOTH

typedef short int boolean;
typedef short int axis_id;

typedef struct trans_struct /* Structure for transformation matrix */
    {
    float m[4][4];              /* The 4x4 matrix */
    boolean w_identity;                    /* Whether or not 'w' will be changed by this matrix
    } transform;

typedef struct s_t_struct    /* Structure for matrix node in stack */
    {
    struct s_t_struct *next;           /* Pointer to next node in stack */
    transform t;                /* The matrix itself */
    } s_transform;

typedef struct                           /* Info about current environment */
    {
    int xmin,xmax,ymin,ymax;             /* Boundaries of display device */
    int imax;                             /* Maximum pixel intensity */
    float xadjust,yadjust;  /* To adjust for non-square aspect ratios */
    } inquiry_response;

typedef struct
    {
#ifdef SMOOTH
    float ni,nj,nk;             /* Surface normal at this point */
#endif
    float x,y,z,w;              /* The point itself */
    float sx,sy,sz;             /* The screen coordinates of the point */
    } point;


typedef struct
    {
    int red,green,blue;                 /* The color of the face */
    int bred,bgreen,bblue;  /* The color of the backside of face */
    float ni,nj,nk;             /* Normal to face */
```

/* gp.h */

```
    float d;                       /* d component of plane equation */
    boolean smooth;            /* Whether or not to smooth shade this face */
    int num;                          /* The number of vertices making up the face */
    point vlist[1];          /* The vertices themselves */
    } face;

typedef struct node_struct   /* Binary tree node */
    {
    face *f;                          /* Pointer to face structure */
    struct node_struct *pos;          /* Pointer to positive sub-tree */
    struct node_struct *neg;          /* Pointer to negative sub-tree */
    } node;
```

D BIBLIOGRAPHY

1. Bass, Daniel H., Using the Video Lookup Table for Reflectivity Calculations: Specific Techniques and Graphic Results, Department of Computer Science, University of Rochester, 1979, [unpublished paper].

2. Bui-Tuong, Phong, "Illumination for Computer Generated Pictures", Communications of the Association for Computing Machinery, 18(6), June 1975, pp. 311-317.

3. Fuchs, Henry, Abram, Gregory D., and Grant, Eric D., "Near Real-Time Shaded Display of Rigid Objects", Computer Graphics, 17(3), July 1983, pp. 65-72.

4. Lipscomb, James S., Three-Dimensional Cues for a Molecular Computer Graphics System, Department of Computer Science, University of North Carolina at Chapel Hill, 1979.

5. Newman, W. and Sproull, R., Principles of Interactive Computer Graphics, New York, McGraw-Hill, 1979, pp. 389-390.

6. Sekuler, R. and Levinson, E., "The Perception of Moving Targets", Scientific American, 236(1), January 1977, pp. 60-73.

7. Tucker, Jonathan B., "Computer Graphics Achieves New Realism", High Technology, 4(6), June 1984, pp. 40-53.

8. Wallach, H. and O'Connell, D. N., "The Kinetic Depth Effect", Journal of Experimental Psychology, 45(4), April 1953, pp. 205-217.