

A FORTRAN GRAPHICS AND ANIMATION LIBRARY FOR X WINDOWS

A FORTRAN GRAPHICS AND ANIMATION
LIBRARY FOR X WINDOWS

By
DUNCAN NAPIER, B.SC.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Science

McMaster University
©Copyright by Duncan Napier, December 1992

MASTER OF SCIENCE (1992)
(Computation)

McMASTER UNIVERSITY
Hamilton, Ontario

TITLE: A FORTRAN Graphics and Animation Library for X Windows

AUTHOR: Duncan Napier, B.Sc. (University of Waterloo)

SUPERVISORS: Professor Randall Dumont (Department of Chemistry)
 Professor Robin Griffin (Department of Computer Science and
 Systems)

NUMBER OF PAGES: vii, 194

Abstract

The requirements for a computer package for the graphical animation and visualization of scientific data are discussed. It is concluded from the analysis of these requirements that simplicity of implementation, interactive response and FORTRAN compatibility are features that are strongly desired by users. These features have a major impact on the design of the graphical library. The handling of graphics resources is further complicated by the need to remove the details of resource management from the FORTRAN programmer while at the same time maintaining some degree of flexibility and efficiency. A “black box” system of graphics library primitives has been constructed to conform the needs of the FORTRAN programmer. The library primitives are constructed from a lower-level commercial graphics library to simplify resource management and give the library a FORTRAN ‘flavor’. A modular style of programming that emphasizes event-driven passing of program control is developed to guide the FORTRAN programmer. A User’s Manual containing a series of instructional tutorials as well as a routine-by-routine description of the library is included along with a code listing.

Acknowledgements

I would like to acknowledge the support and guidance of my supervisors Dr. Randall Dumont and Dr. Robin Griffin. I would also like to thank Dr. Dumont and his group for providing me with the hardware and software for this project. This undertaking would also not have been possible without the valuable advice of Mr. Dan Trottier and Ms. Patricia Monger.

Table of Contents

Descriptive Note	ii
Abstract	iii
Acknowledgements	iv
List of Figures	vii
Chapter 1 - Introduction	1
Background And Objectives	1
Software Tools	3
XView	3
XGL	4
Languages	7
Levels of Software in GAL	8
Portability and Platform Issues	8
Chapter 2 - Planning and Conceptual Design	9
X Windows Tools and Utilities	10
Graphics Primitives	15
The View Model	16
Color and Animation	18
Chapter 3 - Software Architecture of GAL Applications ..	20
Static Program Structure of GAL Applications	23
Dynamic Structure of GAL Applications	25
Feature Interaction in GAL Applications	26

Chapter 4 - Epilogue	28
Enhancements to GAL	28
Extensions to GAL	31
Limitations of GAL	32
Conclusions	34
Appendix 1 - GAL User's Manual	36
Appendix 2 - GAL Code Listing	94
Appendix 2a - GAL Header Files	96
Appendix 2b - X Windows Utilities	100
Appendix 2c - Color and Animation	111
Appendix 2d - Views and Transformations	130
Appendix 2e - 2D Primitives	136
Appendix 2f - 3D Primitives	170
References	193

List of Figures

Figure 1. Software Layers.	9
Figure 2. Object Handling in GAL.	14
Figure 3. Double Buffering Schematic.	20
Figure 4. Conventional Interactive Loop Flow Diagram.	41
Figure 5. Notifier-based Loop Flow Diagram.	42
Figure 6. Flow Diagram of GAL Application.	61

Chapter 1 - Introduction

Background and Objectives

FORTRAN was developed as a programming language for the IBM 704 and quickly became the standard programming language within the scientific and engineering community [MACLEN86]. There have been numerous revisions and extensions since then and some 35 years later, FORTRAN still endures as a major language in the fields of scientific and engineering computation. FORTRAN's popularity has been attributed to the large amounts of code invested in libraries and routines over the years and even decades. Newer languages, such as C and Pascal, do not have this accumulated body of tried and tested code and appear unlikely to displace FORTRAN from its place within the sciences and engineering in the near future.

The emergence of increasingly powerful and sophisticated hardware at lower cost has also changed the role of computers in the sciences. The numerical processing ("number-crunching") that was long associated with scientific computation is now commonly integrated with data visualization and manipulation capabilities on scientific/engineering workstations using a Graphical User Interface (GUI). The challenge to the vendors of scientific/engineering workstations has been to increase machine and software performances to allow for the display and processing of large datasets at interactive rates [FOLEY90].

The above-mentioned factors have heavily influenced the requirements analysis in developing a graphics and animation library for members of the scientific community. The following user requirements were specified:

- The package had to support FORTRAN77.
- The only requirement placed on the user was a working knowledge of FORTRAN77.
- No knowledge of computer graphics on the part of the user was to be assumed. (An understanding of some basic principles of linear algebra *was* assumed).
- The package would have a simple kernel of commands and could be easily and quickly learned, with little or no training.

The requirements stated above reflected the need for a high-performance graphics package for scientists who are not professional or advanced programmers. The package envisaged would present the user with a reasonable choice of defaults and parameters as well as the ability to use the library in a ‘black-box’ manner.

- The package had to have animation capabilities.
- The graphics that were rendered could be manipulated interactively via the GUI.

These last two requirements are the key factors that initiated this undertaking. Fairly sophisticated and powerful FORTRAN-callable commercial graphics libraries do exist (for example PHIGS, SuperMongo, CA’s Disspla). No known commercially

available package meets all the guidelines stated above. The desire to incorporate these requirements into a working package initiated the development of a customized Graphics and Animation Library (GAL).

Software Tools

The complexity of the undertaking and the 5 to 6 month development time allocated for it required the use of some software tools that were specific to Sun Microsystems's SunOS operating system and OpenWindows (Sun's implementation of the Open Look GUI). This decision to implement the package on a specific platform, namely the Sun4 architecture running X Windows, was the starting point of the project. All graphical user interfaces were implemented through Sun's XView Toolkit for X Windows. The processing and rendering of the graphical environment was accomplished through Sun's XGL graphics package. Information on XGL can be found in [XGL91] and [XGL91A].

XView

XView (X Window-system-based Visual/Integrated Environment for Workstations) is a toolkit developed by Sun Microsystems Inc. that supports the development of graphics-based applications running under the XWindow System. The package has an object oriented style with a hierarchy of *widgets* - pre-built, user-interface objects such as canvases, control panels, buttons and sliders. XView is in turn built up from Xlib, the lowest programming level of the X Windows system. A comprehensive guide to the XView toolkit can be found in [HELLER92].

X Windows uses the client/server model in which the application (client) makes requests to the server (which runs on a workstation) to draw objects (text, windows and so forth). The application programmer links the application to X Windows through Xlib. Xlib commands are passed to the operating system which in turn passes an X Protocol Package to the server. For a brief discussion on X Windows refer to [POUNTA89]. An introduction to X Windows and Xlib is found in [JONES89]

The XView toolkit does support Sun FORTRAN but requires a knowledge of Sun's extensions to FORTRAN (which resemble C) as well as some degree of programming proficiency above and beyond those stated as requirements [SUN90]. It was decided to customize the XView widgets and objects to be used in GAL by writing bindings for the FORTRAN code in C and then interfacing the C subroutines with FORTRAN. (The term *binding* is used to refer to a subroutine call that interfaces the calling routine with one or more other subroutines.)

XGL

XGL is a software library of 2 dimensional (2-D) and 3 dimensional (3-D) graphics primitive functions designed to run on Sun hardware platforms. XGL requires a windowing system to manage drawing operations and cannot render graphics to a raw display screen [XGL91]. XGL is written in an object-oriented style. XGL is class-based, and the XGL programmer can only define instances of a class (i.e. an Object), but cannot create new classes. XGL objects correspond to graphics resources. The programmer renders graphics by manipulating these resources.

Manipulation is carried out through XGL's operators. In order to render an object the user defines an object and then proceeds to set its attributes using XGL's operators.

A simplified example of the use of objects to set options and generate display lists is as follows: the Raster Object is the display device resource. The Raster Object is associated with the XView Canvas Object. The hardware color scheme attribute of the Raster Object is set, depending upon whether the platform supports RGB color scheme or an Indexed color scheme (the latter uses hardware lookup tables). A Color Map Object is then initialized and a color table is built up as one of the Color Map attributes. The Color Map Object is *attached* to the Raster Object, becoming in effect an attribute of the Raster Object. Then, a Context Object is initialized and the attributes of the graphical context (the graphical image) are set. The Context Object is also attached to the Raster Object. A call to a graphics primitive results in the attributes being passed through the rendering pipeline (this process is transparent to the user) and rendered to the screen, a memory buffer (in the case of double-buffered animation) or a graphical output file. Essentially, using XGL involves initializing objects, attaching them to a resource and then setting their attributes using XGL's operators. Algorithm 1 shows the pseudocode of a code fragment of this sequence. (*This font is used for pseudocode statements, this font is used for comments*).

Algorithm 1. XGL pseudocode fragment to make a color map and graphical object.

This fragment makes a color map and then draws a red circle on a white background.

Declare the circle parameters

integer circle_color;

real circle_radius;

Initialize the object structures

initialize Raster_Object;

initialize Color_Map_Object;

initialize Graphical_Context_Object;

Attach the graphical context to the raster

attach Graphical_Context_Object to Raster_Object;

Attach the color map to the raster with attach operator

attach Color_Map_Object to Raster_Object;

Set Color_Scheme attribute of Color_Map_Object to Indexed_Color

Color_Map_Object.scheme := Indexed_Color;

Set up indices corresponding to various colors, 1 is red, 2 is purple, 3 is green, 4 is white

Color_Map_Object.color[1] := red;

Color_Map_Object.color[2] := purple;

Color_Map_Object.color[3] := green;

Color_Map_Object.color[4] := white;

Set the background color attribute of the graphical object to white (index 4)

Graphical_Context_Object.background_color := 4;

Set the color of the circle to red (index 1), radius 1.0

```
circle_color := 1;
```

```
circle_radius := 1.0;
```

```
Draw in the circle
```

```
set_circle_attributes(Graphical_Context_Object, circle_color, circle_radius);
```

The XGL libraries lie directly above the hardware and firmware of the display devices and graphic accelerators. This is done to minimize software overhead and maximize performance. XGL emphasizes transparent acceleration through graphical resources. Where the corresponding hardware does not exist, emulation facilities are often available. The purpose of the GAL package was to cohesively group the more intricate tasks required of XGL applications programming. These groupings would constitute a library of FORTRAN-callable modules that a non-specialist could easily learn and utilize.

Languages

XGL's object-oriented style and its use of sophisticated data structures make it unsuited to FORTRAN applications. The modules for the libraries were written in C and given a FORTRAN-compatible interface. This resulted in certain parameter-passing protocols and the *hiding* of data structures from the FORTRAN programmer. These features were considered necessary to give the library a FORTRAN flavor. The FORTRAN "flavoring" of the library had a large influence on the software design. The desired result was to obtain an environment in which the FORTRAN programmers could use GAL as they would use any other FORTRAN library.

The technical details of interfacing C and FORTRAN are discussed in manuals such as [SUN90] and [LOUK90]. Most of the details are minor, such as accommodating FORTRAN's parameter passing by reference (as opposed to value) and the concatenation of C subroutine names with an underscore (_). The real problems of interfacing C and FORTRAN arise in the software design stage when the designer has to decide what course of action to take in dealing with C structures that have no representation in FORTRAN77.

Levels of Software in GAL

A FORTRAN program that uses GAL (from now on referred to as the FORTRAN *application* or simply the *application*) may be thought of as consisting of four levels of software (refer to Figure 1). The highest level is FORTRAN code written by the user. GAL is intended to run on Sun platforms and is written to interface with Sun FORTRAN. The second level is the basis of this thesis - the Graphics and Animation Library. This level is made up of C code, supported by Sun programming tools, namely the XView toolkit and XGL. XView and XGL form the next layer and were used to construct software to render graphics in an event-driven X Windows environment. The final layer is the lowest level - the lower-level implementations of XView and XGL (although this does not suggest that their implementations are by any means similar).

Portability and Platform Issues

Due to the use of platform-specific software tools, the portability of GAL from platform-to-platform is limited. GAL requires an environment running Sun's OpenWindows System. As for the hardware requirements, an attempt was made to

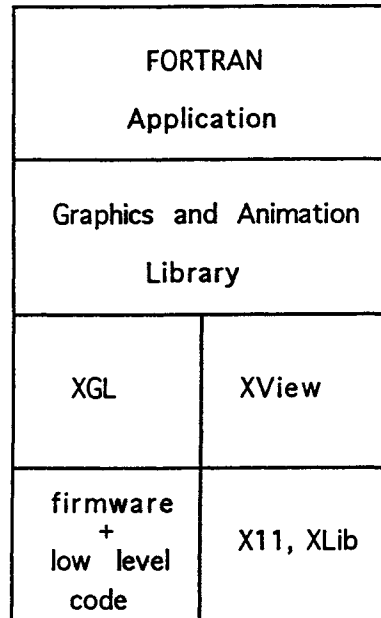


Figure 1. The layers of the Graphics and Animation Library Application. Each layer is built on another layer of software, down to the lowest level (firmware and X libraries).

accommodate as wide a variety of hardware options as possible. Graphics utilities such as z-buffering, double buffering and graphics acceleration are emulated in software by XGL where hardware facilities are lacking. The package supports the Indexed method of color representation. The RGB method is not supported, although allowance was made for future implementation of this system. (RGB systems for graphics applications are currently less common and cannot support double buffering, which is at the heart of this package).

Chapter 2 - Planning And Conceptual Design

Throughout the planning and design stage GAL was divided into 4 areas. These were:

1. X Window Tools and Utilities
2. Graphical Primitives
3. The View Model
4. Color and Animation.

These areas were explored simultaneously in the early stages of the project (4 to 6 weeks) to determine the feasibility of the requirements analysis. The partitioning of the project into 4 mini-projects corresponded to the 4 perceived areas of decision-making that would determine the implementation and appearance of a FORTRAN-callable graphics and animation package. These four areas were not mutually exclusive and were often found to overlap. A more detailed discussion of each area follows.

X Windows Tools and Utilities

The objective of this area of the project was to provide a simple interface with the XView toolkit for the FORTRAN programmer. Simplicity dictated that

the XView window and frame parameters should be set with as many defaults as possible. This was done for the purpose of minimizing the number of library calls and parameters that the user would need to use to set up the window and canvas (rendering surface) of the application. As a result, the only parameters that the user has control over in a window are the size of the window and the picture to be rendered to its canvas.

The types of XView widgets (the pre-written GUI icons of XView) were also kept to a minimum. Only 2 of the XView widgets were made available to the applications programmer - the panel button and the slider. The panel button initiates a callback response when pushed (an “all-or-none” action). The slider, on the other hand, allows for the input of one of a continuous series of values (a “choose-one-out-of-many” action).

The issue of how to handle objects had to be settled very early into the project. It was decided to “hide” structures from the FORTRAN programmer by making the structures global structures. Since the scoping rules of FORTRAN do not permit global variables (exceptions can be made via a COMMON block), the structures were deemed safe from corruption through indiscriminate access.

The creation of an object (a child) from its parent class and other object-oriented procedures were accomplished by invocation of the C libraries. The global objects had their attributes set or child classes created in a manner that was invisible to the FORTRAN programmer. The following example will be used to illustrate this. Suppose the FORTRAN programmer wished to create a window. The FORTRAN statement

```
CALL GAL_INIT_WINDOW(WIDTH, HEIGHT, LABEL)
```

causes the creation of the XView Frame Object. The Frame Object is of the Class Frame in XView, and has the attributes WIDTH (the width of the window), HEIGHT (its height) and LABEL (a character string labeling the window). Suppose one next wanted to paint a figure to the window. The user would then invoke the FORTRAN call

```
CALL GAL_PAINT_PROC(PAINT_PROCESS) .
```

PAINT_PROCESS is declared as an EXTERNAL variable in FORTRAN (meaning that it is the name of a procedure) and is an attribute of the Canvas Object (of Class Canvas). The Canvas Class is derived from the Frame Class via a C library call (i.e. the former is the child of the latter). PAINT_PROCESS is a GAL/FORTRAN procedure that carries out the rendering of graphical objects (graphical objects may be thought of as children of the Canvas Class). These library calls are “black boxes” to the FORTRAN applications programmer.

GAL Objects exist as static global structures in C. Referring back to Algorithm 1, an object such as Graphical_Context_Object was static and global throughout the duration of the program. Each time a new graphical primitive (e.g. squares) was added, the global object was modified and re-attached to the Raster_Object. The global structures were declared and compiled independently so that they are only accessible through access functions. Access functions are functions that return a value associated with a variable or object. Access functions were used to limit the accessibility to global structures, in accordance with good programming practice [HOLLUB87].

Access functions in the GAL source code can be identified by their names, which are all prefixed with the word ‘get_’. For example, the C function `get_frame()` returns the address of the frame object (i.e. its C type is a pointer to the frame - this is called an *object handle*). Figure 2 schematically illustrates the concept of global C structures within a framework that is ‘hidden’ from FORTRAN.

The next design challenge was the integration of XView’s notification-based tools into GAL. Notification-based programs are often used to run event-driven systems. In the conventional style of programming, interactive input is entered in a request loop (refer to Figure 4, Appendix) which is exited when the input is completed. Event-driven systems appear to present many sources of input to the user simultaneously. The inputs and their appropriate responses are handled by *callback procedures* (or simply *callbacks*) which are monitored for input in a notifier loop. Figure 5 (Appendix) illustrates how a notifier exists outside the main program. When an input is entered (by pressing a button on the screen with a mouse, for example) the notifier calls the callback procedure associated with that input.

In GAL, the statement

```
CALL GAL_PANEL_BUTTON(CALLBACK_PROC, X, Y)
```

would set up a button that called the user-defined subroutine `CALLBACK_PROC`. (The location of the button on the window and its label are also passed as parameters, `X` and `Y`. For a full description of the package and examples of applications, refer to the User’s Manual in the Appendix). Sliders are somewhat more elaborate

in their implementation, but the principles involved are unchanged. Algorithm 2 shows the button implementation in pseudocode.

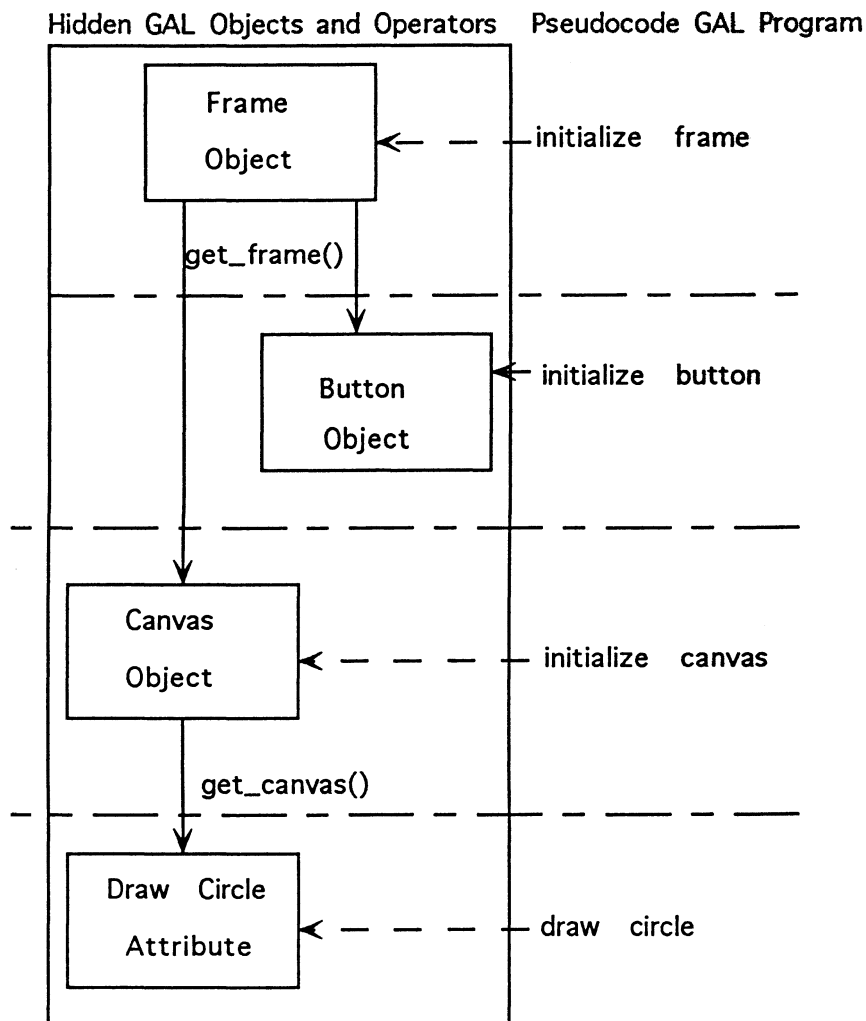


Figure 2. Schematic representation of the Object Hiding in GAL. The Objects are represented by small rectangular boxes. The solid arrows show class inheritance, which is carried out with the help of access functions. Objects in the large box are invisible to the outside world. The right hand side shows a sequence of GAL pseudocode calls. The broken arrows connect the GAL calls with their objects, via the library interface. All manipulations of Sun's object classes are handled in this manner.

Algorithm 2. Panel button implementation in pseudocode

```

subroutine GAL_PANEL_BUTTON(CALLBACK_PROC, X, Y);
integer : X, Y;
start
    draw panel button at (X, Y);
    register the callback procedure (the appropriate procedure in Figure 5) for this
button
    register CALLBACK_PROC() with notifier;
end

```

Graphics Primitives

Graphics primitives were implemented through XGL. The GAL library supplied FORTRAN bindings to XGL for the application. Parameters passed from the FORTRAN application were converted to object attributes through these bindings. XGL graphics are the attributes of an XGL Graphics Context Object, which could be either a 2-D or 3-D Context Object.

The Graphics Primitives stage of planning and design largely involved settling conflicts that arose between the speed and complexity of the graphics primitive library. The condition that the library be simple and easy to learn often ran counter to the implementation of the simplest, fastest code for the primitives. For example, it was thought that a single set of primitives for 2-D and 3-D graphics would be preferred (instead of having, say, one primitive to draw 2-D circles and another to draw 3-D circles). However, some overhead would be required to check for the dimensionality of the data, resulting in a slowdown. Since GAL was designed for

animation, and is therefore required to run as fast as possible, it was concluded that such overhead was undesirable. The result is a near-duplication of graphics primitives, one set each for 2-D and 3-D graphics. This type of dilemma occurred repeatedly during the design of graphics primitives, and the primitives of the GAL package are generally some compromise between efficiency and simplicity of form. The quantitative effect of this overhead was never established, since the vendor of XGL was either unwilling or unable to supply a profiling library for timing profiles. The remainder of the planning stage in this area involved obtaining a “wish list” of graphical functionalities that would be useful to scientists and engineers.

The View Model

GAL has a specific View Model associated with it. 2-D models suffer no distortion, since they are mapped 1:1 onto a 2-D view (the display screen). In order to map a 3-D model onto a 2-D screen, a 3-D view model must be established. In graphical systems, planar geometric projections are used to map a 3-D object onto a 2-D surface. The projection model used in GAL was a Parallel Orthographic Projection. The projection is parallel because the center of projection is at infinity and the projectors are parallel. The projection is also orthographic because the projectors are always orthogonal or perpendicular to the projection surface. Chapter 6 of [FOLEY90] contains an in-depth discussion of view projections and models.

Parallel orthographic projections are usually the least demanding projection from the computational standpoint because no scaling transformations are required. The only transformations needed in this type of projection are shear transformations.

Keeping the number of transformations to a minimum allowed the attainment of maximum speed.

The resulting 3-D images have a particular “motif” or characteristic “feel” to them. Orthographic parallel projections maintain the parallelism of parallel lines. This is often preferred from the standpoint of scientific and engineering graphics where straight and parallel lines are common. In orthographic parallel projections, angular dimensions are not conserved. The shear transforms give cause *foreshortening* which compensate for the lack of scaling perspective.

Orthographic parallel projections also distort the perception of depth in the projected image. Images at different distances from the user are not scaled (as in Perspective Projections) and ambiguities can arise in the interpretation of these images, especially in wireframe models. To overcome these difficulties, facilities to allow for View Transformations were implemented. View Transformations allow the user to transform the Model coordinate system, in effect moving the viewer around for a “better view”. It is important to distinguish between View Transformations and Model Transformations. View Transformations involve the transformation of the coordinate system. The allowed View Transformations in GAL were rotation of the principal (x,y,z) axes, translation in space and zooming (a scaling transformation). Model Transformations are specified by the user, and an example would be the effect of physical forces acting on an object. A sequence of Model Transformations that are drawn and redrawn on the screen is the basis of animation. View Transformations only change the viewer’s perspective of the object. View Transformations were implemented in GAL by XGL’s transformation operators. These operators set Transform Attribute of the XGL graphical context. The result was a

transformation of the model-to-screen projection. All transformations in GAL are with respect to the Origin.

Another design issue that had to be resolved was the coordinate system convention. XGL supports coordinate specifications in either Device Coordinates (i.e. pixels) or Model Coordinates (the physical coordinates/dimensions of the object). The latter was selected, since it is the more natural representation for scientists. The axis conventions are a right-handed axis coordinate system, with the positive y vector pointing vertically up the screen, the positive x vector pointing to the right, and the positive z vector pointing out at the user. These settings are fixed and cannot be altered through GAL. The mapping from model coordinates to device coordinates is done automatically in GAL and is based on two factors. The first factor is the size of the window, which is set in pixel units by the application. The second factor is the viewing volume, which is set in model coordinates in the application and passed to XGL via GAL bindings. The viewing volume is set by giving the dimensions of the “viewport” which are then mapped to the canvas. Note that rotation primitives are lacking for the 2-D systems. It was concluded from discussions that such a facility is of dubious value in 2-D, since the viewer does not gain any new visual information from a rotation in the plane of the object.

Color and Animation

This area of the project consumed the most time. Much of the early work was done using a simple color scheme of 2 colors - black and white. When the success of implementation was assured, the color table was enlarged to accommodate a wide variety of colors.

For reasons discussed previously, an Indexed Color Scheme was used. Indexed color involves associating an integer number with an r,g,b combination of color via a hardware lookup table. Indexed color allows for 8 bits of color (256 colors) based on a 24-bit color system (8 bits each for red, green and blue). Refer to [FOLEY90] and [SUN90] for more information. The system is set up with a default color scheme which may be altered (refer to the User's Manual). The color tables in GAL are initialized along with the XGL 2-D and 3-D Graphical Context Objects. All these initializations are carried out in the GAL library call `GAL_INIT_COLOR()`.

A simple, yet effective method of implementing animation was central to accomplishing the goals of GAL. The package emphasized double-buffered animation, although unbuffered animation is also possible. Double-buffering is a technique used to remove the flicker observed when the contents of a screen are erased and redrawn. Instead of erasing the screen prior to redrawing, the output to the screen is switched from one screen buffer to a second screen buffer. Screen buffers are storage devices (typically video RAM) that store a screen image. While one buffer is being displayed, the other (hidden) buffer is erased and redrawn, and when this is completed, the buffers are switched again. The result is a smooth transition from one "frame" to the next corresponding to a screen refresh raster scan. Some platforms possess hardware double buffering. This essentially consists of duplicate screen buffers (video chips that store a screen image). Hardware double buffering is at the present time an expensive accessory, and XGL has the ability to perform software emulation of the double buffer. Software emulation is carried out through color map double buffering and requires splitting the 8-bit color index into two parts, one for each buffer. This requires some program manipulation of the color index in forming a double buffered index. As a result of double buffering, an 8-bit index is reduced

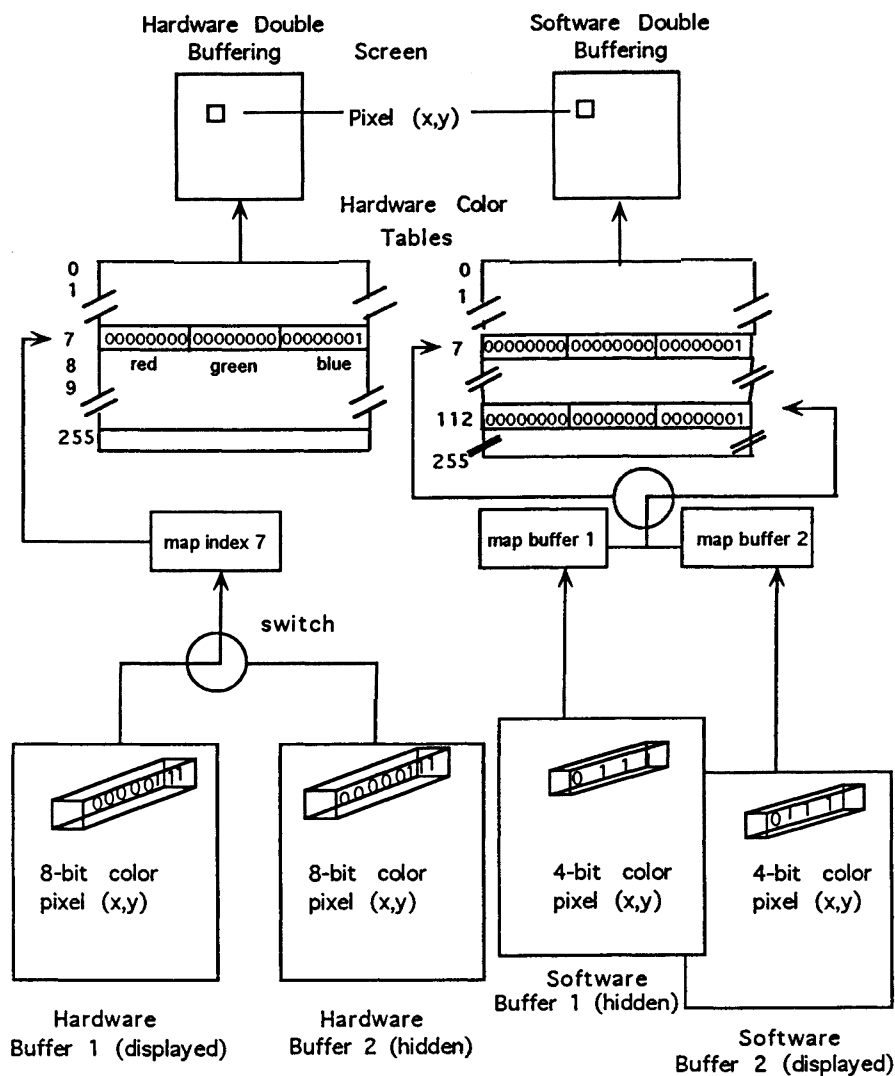


Figure 3. A schematic diagram to illustrate hardware double buffering (left) and software (color map) double buffering (right). The buffer contains the color index of each pixel (location (x,y)). In the hardware buffering, 8 bits or 256 colors are available. In software buffering, the color pixel is split into 2 “planes”, each 4 bits in size. In the hardware case, the pixel color is blue, corresponding to index 7 (00000111 in binary). In the software case, the color 7 is now 112, which corresponds to 7 in 2×4 -bit planes (01110111).

to 2×4 -bit indexes resulting in a maximum of only 16 colors in double-buffered images. Figure 3 shows the a schematic representation of double buffering concepts. Note that double buffering has use in applications other than animation. It was found that slow redrawing of large static datasets resulting from rotation, for example, could be made more pleasant by redrawing to a hidden buffer, and then switching the buffers.

A rigid, customized scheme was devised for animation for the sake of simplicity of programming on the part of GAL users. Animation is accomplished through a repetitive callback procedure. If animation is desired in a GAL application, the user invokes the animation button (supplying the name of the callback routine) and sets up the appropriate double buffer (2-D or 3-D). Pressing the animate button toggles the repetitive callback on, and the routine specified is called repeatedly. The algorithm for the animation loop is described in Algorithm 3.

Algorithm 3. The algorithm for the toggle on/off animate button.

while animate button not pressed

Carry out the animation sequence

get next frame data;

draw next frame to hidden buffer;

interchange displayed and hidden buffer;

Check for other inputs, carry them out, then return

check notifier for other inputs;

endwhile;

return to notifier;

By calling a routine to set up a 2-D or a 3-D buffer (only one is allowed at a time), and registering a Model Transformation/Drawing module as the animation callback, the application can be made to perform double buffered animation. (Refer to the User's Manual and source code in the Appendices for details).

Chapter 3 - Software Architecture of GAL Applications

Static Program Structure of GAL Applications

GAL applications are intended to be highly modular by design. (Refer to the Tutorials in the User's Manual). A typical GAL application consists of a Main routine and one or more subroutines. The Main routine typically serves 4 functions, which are carried out in the following order:

1. Initialization of values. Initialization refers to a procedure that only needs to be carried out once in the course of the application. An example is the assignment of starting coordinates of an animated object. The subroutine INITIALIZE in Tutorial 4 of the User's Manual is such an example.

2. Registration of callback procedures. Callback registration is essential to ensure that a specific event results in control being passed to a specific subroutine. All event-driven procedures in GAL as well as the procedures for painting the canvas and animation depend on the notifier. In even the most complex GAL applications, the callback procedures are divided into at most three categories.

(a) Rendering Callback Procedures. This is the rendering or drawing component of the application. It is made up of a subroutine or series of subroutines associated with the rendering of graphical primitives, and is fundamental to all graphical applications. These subroutines simply invoke the drawing of a primitive to the

screen while passing the relevant primitive attributes to the GAL. The callback for painting the XView Canvas belongs to this category. Typically, a Rendering Callback Procedure consists entirely of GAL 2-D and 3-D graphics primitives. Refer to FORTRAN subroutine DRAW_PIC in Tutorial 2 of User's Guide in the Appendix for an example of this type of callback.

(b) View Transform Callback Procedures. These callback procedures are associated with the transformation widgets (especially sliders). A View Transformation Procedures are written when the application requires an interactive view adjustment environment. Tutorial 3 in the Appendix User's Guide contains the subroutine ROTATE which enables interactive View Transformation of the graphical environment.

(c) Model Transform Callback Procedures. These callback procedures control the motion of objects on the screen during animation. The Model Transformation is defined by the applications programmer and animation is simply a series of successive redrawings of the transformed primitives. Tutorial 4 of the User's Manual uses the subroutine NEXT_FRAME for this purpose.

3. Setting of graphics environment. Following the registration of callback procedures, the graphics environment of the GAL application is set. No defaults are present, and the user must explicitly call GAL routines to set up color tables, z buffers, double buffers, viewport dimensions and any color table redefinitions.

4. Starting the notifier loop. The notifier loop is at the center of the event-driven application. The notifier loop is entered at the end of the Main program and the program goes into a wait-for-event state. When an event occurs (e.g. pushing a button) the procedure associated with this event has control passed to it. The as-

sociation between event and procedure was established during callback registration. The GAL library procedure `GAL_END` passes control to the notifier loop.

Dynamic Structure of GAL Applications

The dynamic or run-time structure of GAL applications is based on the notifier loop, as described previously. Figure 6 (Appendix) attempts to conceptually link the static and dynamic structures of a GAL application. A GAL program is conceptually a series of event-driven modules threaded together by the notifier. The flow of control through a callback subroutine and the subroutines invoked by a callback is analogous to a pipeline. The control sequence in Figure 6 is that for an animated graphical object that can be manipulated in 2-D or 3-D. The passing of control to the callback results in a sequence or chain of subroutine calls (refer to the tutorials in the index). The graphical objects are transformed according to the physical model, they are then piped through the view transformation settings (typically based upon a slider setting) before being rendered to the screen. After reaching the end of a “chain” of subroutines, control is again returned to the notifier. Tutorials 4 and 5 in the Appendix deal with three single stage callback modules (Model Transform, View Transform and Rendering), but as Figure 6 implies, any number of stages can be chained together. This thinking parallels the graphics or rendering pipelines that operate in computer graphics hardware. Examples of the application of several Model and View Transform stages would be the toggling transformations “on” and “off” by using flag settings to bypass designated modules. This is of use in the modeling of physical objects where forces or perturbations are altered, switched on or switched off interactively and in real time by the user. Similarly, the use of chained rendering subroutines gives the user the flexibility of adding or removing

transient graphics, for example, a grid, at the push of a button. This modularity has the advantage of flexibility and reusability in GAL applications code. The result of these threadings and chainings is the ability to build fairly sophisticated and highly interactive graphical environments.

The key to good design of GAL applications is to ensure that notifier threading and transfers of control are efficient and physically consistent with the system being modeled. There is a hierarchy or sequence associated with this threading or chaining, as Figure 6 shows. For instance, one chains the View Transformation modules to the Model Transformation modules and not vice versa. The reasons for this become obvious when one considers what would happen if the two were interchanged in Figure 6. The result would be a Model Transformation (corresponding to a frame advance in animation) on top of manipulation of the graphical object. This would be undesirable in the case where animation is suspended and the object is being manipulated in space by the user.

Feature Interaction in GAL Applications

There is some degree of interaction between independent primitives in GAL that needs to be clarified. The primitives that handle XWindow Tools and Utilities, and Color and Animation (refer to the divisions in Chapter 2) tend to be highly restricted in the manner and order in which they may be applied. For example, the primitive `GAL_START` must be invoked before any other GAL primitives and `GAL_END` must be the final GAL primitive in the program sequence. Referring back to the static structure, one observes a specific sequence of commands that are required to set up and initialize GAL's structures and objects. Once these structures

are initialized, the application is free to manipulate them in any chosen order. This second set of GAL primitives tend to belong to the graphics and transformation subset of GAL primitives. This is largely attributable to the fact that graphical objects can usually be processed in an order-independent manner [TORBO88].

Some interaction between GAL's primitives can be attributed to features of XGL. The mixing of 2-D and 3-D graphical primitives in the same application is an example. The reason for the partitioning of GAL's graphical primitives into 2-D and 3-D sets is due largely to the fact that XGL's operators for 2-D and 3-D graphical objects are only partially overloaded. In other words, some operators only operate on 2-D objects, some only on 3-D objects and others operate on both. This explains the reason for GAL having 2 primitives for setting the boundaries of the view space (one for 2-D graphics and one for 3-D graphics), but a single primitive for handling double buffering in both 2 and 3 dimensions. This arrangement is probably due to XGL having independent 2-D and 3-D rendering pipelines [XGL91]. This pipeline configuration speeds up the rendering of XGL's 2-D graphics. Initial tests on GAL have shown that mixing 2-D and 3-D primitives in the same application is possible, but there is no guarantee that unexpected events will not occur. The mixing of 2-D and 3-D primitives is not recommended from the implementation point of view besides the fact that, physically speaking, this practice is highly questionable. (A possible motive for mixing 2-D and 3-D primitives would be to gain performance in the case where 3-D objects exist in the plane of the screen, but once again, such a practice is not advised).

Chapter 4 - Epilogue

Enhancements to GAL

After the 4 to 6 week trial period, it was decided to go ahead with a full implementation of GAL. The foundations of the package had already been established, and the improvement and enhancement of the existing features was an ongoing process that continued for the next 4 months. Based on [KNUTH89], enhancements to the package could be classified as one of the following types.

1. Cleaning up. This refers to the improvement of consistency and clarity in the package. Naming conventions for the library were established -for example, all GAL library procedures start with the word 'GAL'. An attempt was made to ensure consistency in the ordering of parameter lists to aid the user. The library names are also highly mnemonic and descriptive for this purpose. These enhancements do very little for the software/hardware implementation but are concerned with the human factors in this package.

2. Efficiency. Efficiency in the implementation of the code was extremely important if the package was to fulfil its goal of high-speed rendering. The tradeoff between efficiency and convenience was always an issue (refer to the section *Limitations of GAL* below). Highly efficient, high-speed code was only vital for the graphics primitive rendering operations. The enforcement of efficiency was directed mainly at these graphics primitives. The processing of graphical information is typ-

ically of a repetitive nature (e.g. the processing of Cartesian coordinates) and the use of “tight” loops in the code was stressed.

3. Robustness. Attempts were made to ensure that the package was robust. The result of this can be seen in the XWindow utilities and color initialization primitives. The consolidation and cohesion of complex assignments and utilities removes some power from the user, but at the same time guarantees the robustness and stability of the package. If an attempt is made to use an XGL object before it is initialized, the program will crash. Fortunately, most computer graphics primitives can be processed in an order-independent manner and there are only a few problems that can result from specifying library commands in an incorrect order. These *order-dependent* primitives deal largely with XWindow, system and color settings.

4. Error Identification and Recovery. It has been said that roughly 25 percent of a program’s code should be devoted to error checking and handling [KNUTH89]. Fairly extensive error checking was carried out in the object initiation and object modification routines of GAL. The graphics primitives contained much less rigorous checking of errors due to the expected increase in overhead. All memory allocation and most object-handling operations were checked for validity. Few checks on user input error are carried out (especially for the high-speed graphics primitives). A possible solution to the problem of error-checking versus run-time performance could be the implementation of two libraries - the first, a “code-builder” library with extensive error checking, and the second, an identical optimized high-speed version to meet the demands of high-performance interactive graphics. Common errors that were encountered in the building of GAL applications could be attributed to one of the following causes.

(a) *Incorrect ordering of GAL calls (i.e. bad static structure).* These errors were checked for by ensuring that no object could be used unless it had first been initialized. Failure to obey this law results in an error message and the program being aborted. The error message identifies the offending library routine and suggests the most probable cause.

(b) *Mismatched subroutine parameters.* Due to the sparse error checking in the graphics primitives, mismatched parameters often resulted in undiagnosed crashes. Refer to the User's Manual.

(c) *Algorithmic errors.* These are not preventable, but can be reduced by clear, concise and logical structuring in the GAL package.

A discussion of GAL programming errors and their diagnoses is included in the User's Manual.

5. Generalization. The goal of generalization as described here was to minimize the number of primitives. The addition of high-level, specialized primitives (e.g. axes, contour, mesh plots) was carried out towards the end of the project. It was found that such primitives, while they were useful for certain specialized applications, were starting to restrict thinking on the scope and depth of the package. It was decided that complex primitives should be obtained from simple primitives if the functional integrity and general usefulness of the package was to be maintained. A policy adopted was that if a complex primitive was requested by a user, a demonstration program of such a primitive (or something similar) would be included along with the package. The partitioning of the primitive set into 2-D and 3-D primitives has already been discussed and is another aspect of this topic. A more general prim-

itive set could probably be attained if Sun's XGL allowed complete overloading (as opposed to the current partial overloading) of the 2-D and 3-D graphical context operators.

6. Portability. Hardware/Operating System portability was discussed in the Introduction and to summarize, it may be said that GAL's portability is restricted by its reliance on Sun's OpenWindows. XGL is also dependent on OpenWindows and hence the portability is fairly limited as things stand so far. Facilities for using hardware z-buffering, double buffering and graphics acceleration have been implemented in GAL and software emulation in XGL is used where they are not present. Care was taken to make GAL fully XGL 2.0 compatible. XGL 2.0 contains some features from older versions of XGL that will be discontinued in the next upgrade. These features were avoided to ensure that GAL will be compatible with the future XGL upgrade.

Extensions to GAL

GAL was written with some facilities for future extensions. The code was written to accommodate the RGB color scheme in the future. Some rewriting of the code for color implementation of the canvas background and the graphics primitives would be required should this be done. The graphics primitive library was written in a standard format with the intention of allowing for extension by persons with a knowledge of C and very little familiarity with XGL. The graphics primitive library is a standalone feature in this sense, and can be extended, edited, modified and changed with no changes to the GAL Utilities (i.e. widgets, color settings, buffers etc.). GAL's View Transformation primitives allow the transformation of a 3-D

object to any orientation in 3-D space and may be considered complete in that sense. 2-D rotations are somewhat restricted, but may be added on in the future (although as stated before, they appear to be of dubious benefit).

Hardcopy output would be another useful extension to GAL. Currently, screen dumps of the OpenWindows desktop are the extent of GAL's hardcopy ability. Screen dumps do not provide the full resolution capabilities that direct printer output can provide. Publication quality laser printers currently supply about 300 dots per square inch (dpi) resolution. Current screen resolutions are of the order of 110 dpi resolutions, and this results in some degradation of image quality of the resulting output. Direct output in the form of a Postscript file or similar would add a fair amount to GAL's utility. There are at present no plans to add such an extension, but it may be worth future consideration.

Limitations of GAL

There are limitations to GAL, and these are largely due to the belief that GAL's kernel should be as simple as possible. Flexibility has been lost because GAL is a higher abstraction of XGL library primitives. Donald Knuth [KNUTH74] describes a similar problem that was encountered in his design of assembly languages and states his personal belief that:

“...such languages should never be improved to the point where they are too easy or too pleasant to use; one must restrict their use to primitive facilities that are easy to implement efficiently.”

The objectives of GAL are, in a sense, the reverse of those of Knuth's assemblers.

One wants to make the package easy and pleasant to use, and the result is the suppression of lower level (i.e. primitive) facilities. One area in which this is obvious is the choice of colors in GAL. In principle, GAL should be capable of displaying 8 bits or 256 shades of color. Instead, the user is forced to use 4 bits or 16 colors. This is due to the color table initialization scheme, which is fairly complex and whose structure is subject to the use of double buffering. The desire to carry out all these tasks in one routine forced a design that yielded a standard color table, whether double buffering was implemented or not.

Solid modeling has been largely ignored. This is due to the fact that most systems currently in use are able only to render the most simple solid models at interactive rates. This is due in part to the computationally expensive processes of shading and z-buffering. The result is a primitive set dominated by wireframe primitives. Photorealism and other sophisticated graphical representations were not among the goals of this project, although they are within XGL's capabilities.

The temptation to add specialized and more complex primitives to the basic set of primitives was always present. There was often pressure from the users of the package to incorporate certain "pet features" into the package. In his development of the typesetting package \TeX , Knuth reports being bombarded with ideas for extensions [KNUTH89].

"By acting as an extremely conservative filter, and by believing that the system was always complete, I was perhaps able to save \TeX from the "creeping featurism" that destroys systems whose users are allowed to introduce a patchwork of loosely connected ideas."

Repeated extensions to GAL would only be useful to a point. Elaborate and extensive additions to GAL would eventually run counter to the goals stated in the Introduction. A user desiring more and more powerful graphics tools should abandon FORTRAN altogether for a more sophisticated language such as C.

Conclusions

The goals of FORTRAN support, animation and interactive manipulation were all attained in this project. The requirement of simplicity is more subjective, and whether this goal was successfully accomplished remains to be seen. The simplicity of the library kernel was one priority that was strongly adhered to during the course of this project. It was believed that a simple kernel would be the basis of a simple package. The tradeoffs for simplicity were some loss of flexibility and a program structure and philosophy that would be unfamiliar to most FORTRAN programmers. It was hoped that simplicity would outweigh the latter two factors. The notifier-based programming structure was deemed necessary if the package was to meet the needs of interactive and fairly sophisticated graphical applications.

Two factors contributed to the threaded/pipelined structure of GAL applications. The first, and more obvious, is that it is in fact a style of notifier programming. The programs have a very dynamical “flow” aspect to them resulting from the notifier-based transfers of control.

The other influence on the program structure is the use of an Object-Oriented tools. The “pipeline” sequence described above is actually a manifestation of the hidden manipulation of Sun’s Object Classes by GAL’s “black boxes”. Object-Oriented methods are largely unfamiliar to the many computer users in the general

population. The recent proliferation of Object-Oriented tools and systems has left much of this new power and sophistication out of reach of this population. Scientists, whose first priority is “doing science”, often do not have the inclination or interest to learn what have become increasingly sophisticated software tools. The purpose of GAL was to bridge the gap between XGL (new) and FORTRAN (old). The hiding of unfamiliar objects from the FORTRAN user seems somewhat counterproductive, but was deemed necessary in the requirements analysis. This raises the interesting point as to whether Object Oriented tools help or hinder the development of systems that are not Object Oriented. From the author’s standpoint, the interfacing of FORTRAN with Object Oriented C required some important decisions and careful thought (e.g. how to hide global objects). However, the result of this - a working interactive, animated graphics library for FORTRAN that was built in just over four months - can be credited in part to Object Oriented methodologies. Features such as reusable code, predefined classes and operator overloading allowed for the rapid implementation of a sophisticated package. Unix programming tools also speeded up prototyping, testing and development. Unix’s symbolic debugger, dbx, was indispensable, as were Makefiles, for rapid compilation. The use of multi-tasking, multiple windows in the X Windows environment was also extremely useful. The ability to debug a program in one window while its graphical output was running in another window saved many hours of time. The importance of such tools can be expected to grow in the future, as computers and programmers take on more complex and demanding tasks.

Appendix 1 - GAL User's Manual

A User's Manual for GAL :
A FORTRAN Graphics and Animation Library
for Scientific Computing

Duncan Napier

Department of Computer Science and Systems and

Department of Chemistry

McMaster University

Hamilton, Ontario, Canada L8S 4M1

email : napier@maccs.dcss.mcmaster.ca

Manual Table of Contents

Installation and Use	39
Programming with GAL : A Tutorial	41
Some Concepts	41
Introducing the GAL Canvas	44
View Transformations in GAL	48
Animation with GAL	51
Depth Perception and the GAL Visual Environment	57
The Structure of GAL Programs	60
Common Errors Encountered in GAL Programs	62
GAL Reference Manual	65
XView Tools	65
Color and Animation	68
View Transformation Primitives	72
GAL Graphics Primitives	74
GAL Bugs	91
Index of GAL Primitives	93

GAL is a program library that supports graphics and animation on Sun platforms running the OpenWindows windowing system. The package is built on top of Sun's XView Toolkit and XGL, a graphics package also developed by Sun. GAL is intended to be used by persons with a working knowledge of FORTRAN who wish to visualize data and animate it or view it interactively. No knowledge of X Windows programming or computer graphics is assumed.

This manual will attempt to familiarize the user with some GAL programming concepts, as well as provide a reference to GAL's graphics primitives along with a series of tutorials and an installation guide.

Installation and Use

In order to use GAL, your system must have Openwindows Version 2.0 or a later version and XGL Version 2.0 or later. The GAL package consists of the archived GAL library (libgal.a) which can usually be found in the subdirectory gal_lib and some FORTRAN programs that demonstrate some capabilities of GAL applications. The libraries that you need to link to run a GAL application are -lxview -lolgx -lX11 -lm (found in your openwindows directory) and -lxgl (found in your xgl home directory) and -lgal (found in gal_lib). Suppose your openwindows directory is in /usr/lib and your xgl direcorey is in /usr. You can compile your application (app.f) by

```
f77 -L/usr/lib/openwin/lib -L/usr/XGL-2.0/lib -Lgal_lib app.f -lxview -lolgx
-lX11 -lm -lxgl -lgal
```

This procedure is carried out in the demo makefile for you. Note that the directory gal_lib contains the archived library libgal.a (invoked as -lgal) and is supplied

to you with the package while the other libraries (-lxview, -lolgx, -lX11, -lm, -lxgl) are present on your system. The tutorials described in this manual are also included with the package.

The speed of rendering of GAL graphics and their appearance can be dependent on your hardware and OpenWindows defaults. Graphics accelerators (e.g. GX and GL accelerators on Sun SPARCstations) will greatly increase the speed of rendering. This increase becomes more noticeable with increasing complexity of the graphics. Some bugs exist and are discussed at the end of this manual.

Programming with GAL : A tutorial

Some Concepts

Windowing systems are event-driven as they present many sources of input to the user at any given time (e.g. menus, buttons, sliders and so on). GAL is built upon XView which has notification-based event handling. In GAL applications, these notifiers control the flow of event-driven code. In the conventional style of programming, interactive input is entered in a controlling loop (Figure 4).

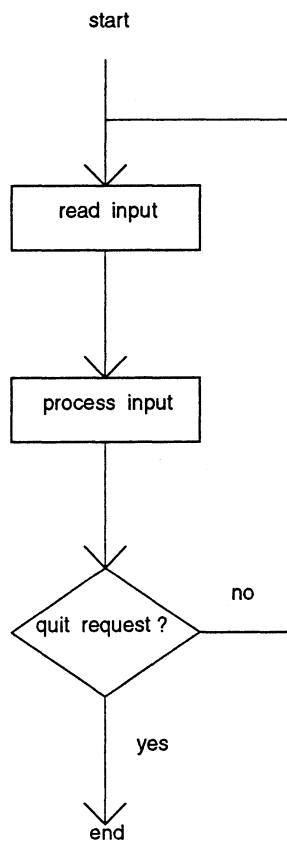


Figure 4. Flow diagram showing conventional style of interactive programming.

In notification-based programming, the programmer “registers” the interactive procedure with the notifier. The procedure associated with that event (i.e. pushing a button, selecting a menu) is called the notify procedure or callback procedure. The callback is called when its associated event is activated. The input controlling loop is located outside the main program in the notifier loop. Refer to Figure 5 for the flow diagram representation of notifier based programs.

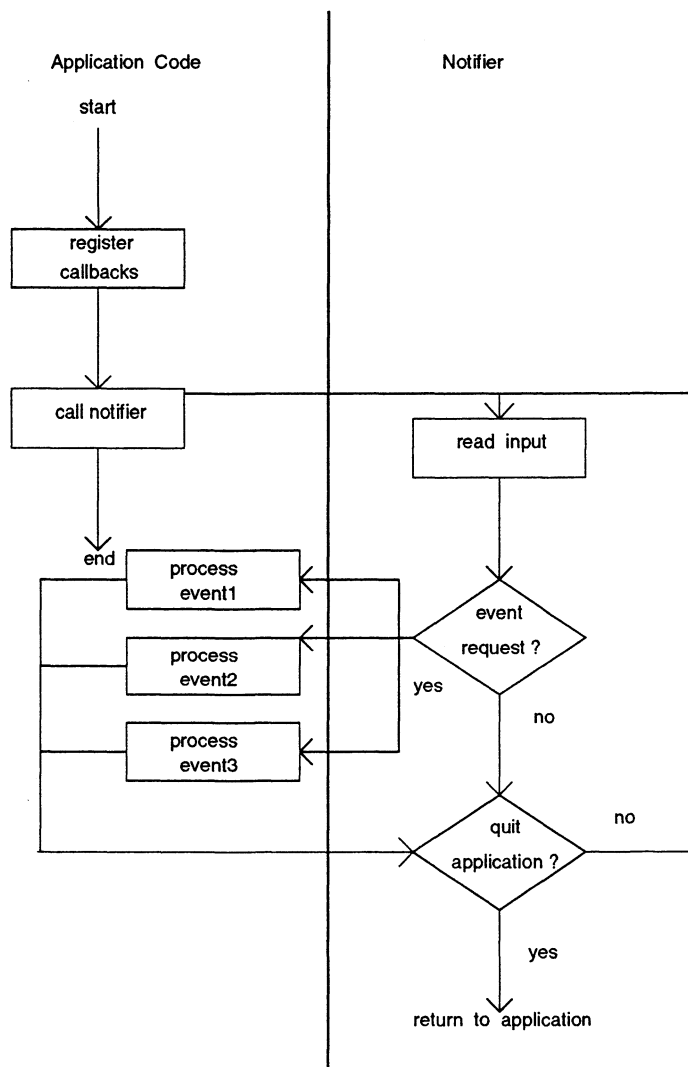


Figure 5. Flow diagram of notification-based program. The notifier exists as a loop outside of the program and waits for interactive input before transferring control to a callback procedure.

You may find all of this overwhelming, but all of these are carried out transparently, and one need not be an X Windows expert to use GAL, provided you follow a few ground rules. To convince you of this, we shall now proceed to write an event-driven GAL program.

Tutorial 1 : An Event Driven Panel.

Try writing and compiling the code in Listing 1. These are the commands required to put up a 500 pixel×500 pixel window with a button at approximately the middle of the window.

Listing 1 A simple example of an event-driven program.

C A FORTRAN application to put up an event-driven window.

PROGRAM LISTING1

C All callback procedures must be declared EXTERNAL

EXTERNAL QUIT

C Initialize the XView window and panel in Openwindows.

C Parameters passed are the x and y lengths in pixels and a

C title.

CALL GAL_START

CALL GAL_INIT_WINDOW(500, 500, 'My First GAL Application')

C Create a panel button, giving x, y coordinates, a label for

C the button and register the name of the procedure to invoke

C when pressed.

C x = 25 puts the button 250 pixels from the left edge of the

C window, 8 puts it about 240 pixels from the top of the

C window. The left upper corner is (0,0).

```
CALL GAL_PANEL_BUTTON(QUIT,25,8,'Press to Quit')
END.
```

C Subroutine QUIT-This subroutine invokes the destruction of
C the window

```
SUBROUTINE QUIT
CALL GAL_QUIT_PROC
RETURN
END
```

In this case, the procedure QUIT is the callback procedure and is invoked when the button to which it is registered is pressed. Notice that all GAL library calls are prefixed with the acronym GAL_.

Introducing the GAL Canvas.

The first tutorial has demonstrated most of the fundamental concepts of GAL. Now, we shall start doing more useful things with the package. Our goal is to draw a simple object using some GAL graphics primitives. This example will also show some of the other display device calls that need to be invoked.

The drawing surface of a GAL window is called the “canvas”. The example in Tutorial 1 did not have a canvas, and the surface observed was that of the control panel. The control panel’s color is a property of the OpenWindows Workspace and can be set from the desktop. The canvas is initialized by calling GAL_PAINT_PROC() and passing the name of the drawing subroutine as the callback procedure. The color tables of the canvas are initialized by calling the library routine GAL_INIT_COLOR().

The default color table consists of 8 colors : black, white, red, green, blue, yellow, cyan and magenta. Each color is associated with an integer number, its index. The color indices range from 0 through 7, respectively.

The background is defaulted to black. You can change the default color scheme by defining your own colors, but you are always limited to 8 colors. After the canvas and color palette have been initialized you must specify the boundaries of the viewing space in “Model Coordinates”. Model Coordinates are the physical space coordinates of the object being imaged, for example, dimensions in feet or centimeters. Model Coordinates are distinct from Device (or Screen, in this case) Coordinates, which are often expressed in pixels, as we saw in Tutorial 1. The viewing volume is defined by invoking `GAL_3D_ASPECT_SET()` in the case of Tutorial 2.

Tutorial 2: Modelling a 3-dimensional (3D) Object.

This tutorial involves drawing a “dumbell” composed of 2 wireframe spheres connected by a line. GAL models objects in Model Coordinates, so we’ll set some dimensions. The radii of the spheres will be 0.5 meters, and we will place their centers 3 meters apart. One sphere will be red, the other green, with a white line joining the two. The default coordinate system is a right-handed system in 3-dimensions, with the positive x coordinate pointing right, the positive y pointing vertically up and the positive z pointing out of the plane of the screen. The same defaults apply in 2D, but with no z-axis, of course. Listing 2 shows the GAL application that renders the model.

Listing 2 A GAL application for modeling a dumbell structure.

C A GAL application to model a 3D object.

PROGRAM LISTING2

C LINE_COORDS - this is the COMMON block that holds the coordinates of
C the line segment. It is defined here to avoid redeclaration in the
C callback procedure.

C All callback procedures must be declared EXTERNAL

COMMON/LINE_COORDS/X(2), Y(2), Z(2)

EXTERNAL DRAW_PIC

C Initialize the line segment start and finish coordinates

X(1) = -1.

X(2) = 1.

Y(1) = 0.

Y(2) = 0.

Z(1) = 0.

Z(2) = 0.

CALL GAL_START

CALL GAL_INIT_WINDOW(500, 500, '3Ddumbell')

C Register the callback routine for painting the canvas

CALL GAL_PAINT_PROC(DRAW_PIC)

C Set up default color palette

CALL GAL_INIT_COLOR

C Define the boundaries (xmin, xmax, ymin, ymax, z min,)

C z max and set window re-sizing protocol.

CALL GAL_3D_ASPECT_SET(-5., 5., -5., 5., -5., 5.)

CALL GAL_END

END.

C Subroutine DRAW_PIC - calls the graphics primitive routines

```

SUBROUTINE DRAW_PIC
COMMON/LINE_COORDS/X(2), Y(2), Z(2)
C Draw wireframe spheres, the parameters are color(red = 2
C , green = 3), radius followed by x, y and z coordinates
CALL GAL_3D_UNFILLED_SPHERE(2, 0.5, -1.5, 0., 0.)
CALL GAL_3D_UNFILLED_SPHERE(3, 0.5, 1.5, 0., 0.)
C Draw line, passing color (white = 1), line thickness, number
C of points in the polyline and 3 arrays, holding the x, y
C and z coordinates of the points, respectively.
CALL GAL_3D_SOLID_LINE(1, 3.0, 2, x, y, z)
RETURN
END

```

Try running the program. The perspective is through a 500×500 pixel viewport representing a space with 10 meters \times 10 meters \times 10 meters. The dumbbell should appear with its long axis lying along the x-axis. As with all GAL windows, you can exit by choosing the “Quit” option from the upper left-hand corner of the window.

Exercise 1 Try adjusting the colors and the dimensions of the object and viewport. Try resizing the window. If there are any extraneous or incorrect lines, select the “Refresh” option from the upper left hand corner of the window.

Exercise 2 Change `GAL_3D_ASPECT_SET()` to `GAL_3D_ASPECT_FREE()`. Resize the window, trying different window sizes and shapes.

View Transformations in GAL

Now that we have some idea of how to model objects in GAL, we will introduce the View Transformation. In GAL, the position or configuration of an object on the screen can be altered in 2 ways. The first is a Model Transformation in Model Coordinates, in which, for example, a modeled force acts on an object and moves it. This kind of Transformation is the basis of animation in the modeling of physical objects. Forces are modeled by the programmer and the object is moved around a fixed coordinate system. A View Transformation, implemented in Tutorial 3 through `GAL_3D_ROTATE`, involves the transformation of the viewing space. In a View Transformation, the viewer's perception of the coordinate system is altered. Tutorial 3 also introduces the slider which is used to present one of a continuous series of possible settings for the user (in this case, the options are rotation angles).

Tutorial 3: View Transformation of an Object in 3D.

This tutorial uses the same model as Tutorial 2, but we will now add some features that allow you to manipulate the object in 3D space by rotating about the principal axes (i.e. x and y axes). The code has some additions, namely the callback registrations for 2 sliders, `GAL_PANEL_SLIDER1()` and `GAL_PANEL_SLIDER2()` and a call to the `ROTATE` subroutine. In the FORTRAN subroutine `ROTATE`, `GAL_ROTATE()` is called prior to drawing to the canvas (`DRAW_PIC` remains unchanged and is now also called from `ROTATE`). Once the slider callbacks are registered, the user can access the slider readings from the values returned by the functions `GET_SLIDER1_VALUE` and `GET_SLIDER2_VALUE` respectively. A GAL application can use up to 7 sliders, numbered 1 through 7. Type in the modifications in

Listing 3 and observe the behavior of our dumbbell as it is manipulated on the screen.

Listing 3 Dumbell model with View Transformation Utilities

C A GAL application to model a 3D object.

```
PROGRAM LISTING3
```

```
COMMON/LINE_COORDS/X(2), Y(2), Z(2)
```

```
EXTERNAL DRAW_PIC, ROTATE
```

C Initialize the line segment start and finish coordinates

```
X(1) = -1.
```

```
X(2) = 1.
```

```
Y(1) = 0.
```

```
Y(2) = 0.
```

```
Z(1) = 0.
```

```
Z(2) = 0.
```

```
CALL GAL_START
```

```
CALL GAL_INIT_WINDOW(500, 500, '3DDumbell')
```

C Register the slider callback (draw_pic in this case), give

C slider a label, give the slider max and min values and

C slider position - an integer 0 through 5).

```
CALL GAL_PANEL_SLIDER1(ROTATE,'Y-Axis Rotate', 180, -180, 0)
```

```
CALL GAL_PANEL_SLIDER2(ROTATE,'Z-Axis Rotate', 180, -180, 0)
```

C Register the callback routine for painting the canvas

```
CALL GAL_PAINT_PROC(DRAW_PIC)
```

C Set up default color palette

```
CALL GAL_INIT_COLOR
```

C Define the boundaries

```

CALL GAL_3D_ASPECT_SET(-5., 5., -5., 5., -5., 5.)

CALL GAL_END

END.

```

C Subroutine ROTATE - rotates the dataset then calls DRAW_PIC

C called from LISTING3

```

SUBROUTINE ROTATE

```

```

    REAL Y_ANGLE, Z_ANGLE

```

C It is VERY IMPORTANT that the slider accessing functions get

C declared as integers

```

    INTEGER GET_SLIDER1_VALUE, GET_SLIDER2_VALUE

```

C Determine the rotation factors (in degrees) and convert them

C to reals, using the library FLOATJ() function

```

    Z_ANGLE = FLOATJ(GET_SLIDER1_VALUE())

```

```

    Y_ANGLE = FLOATJ(GET_SLIDER2_VALUE())

```

C Carry out the rotation, using the slider values (x is unchanging)

```

    CALL GAL_ROTATE(0., Y_ANGLE, Z_ANGLE)

```

C Draw the pictures. The above code could have been added to the

C DRAW_PIC routine, but separating the transformation routines

C from the drawing routines can increase the speed of rendering

C by avoiding unnecessary transformations

```

    CALL DRAW_PIC

```

```

    RETURN

```

```

    END

```

The addition of the second subroutine ROTATE is not essential, but emphasizes the “single-mindedness” approach to subroutine construction. A single module should

preferably perform a single task (for example draw, rotate, translate, change a force constant and so on). By doing so, you can ensure that the transformation is traversed only when its corresponding action is initiated, and not when just any action is initiated. This will become apparent if you wish to perform multiple transformations (whether they are View or Model Transformations) on a model.

Exercise 3 Try performing rotations on the object. Observe how the axes rotate along with the object.

Exercise 4 Add more sliders. Add an x-axis rotation, and add a slider that controls the separation of the spheres, or length of the line (hint: transfer the latter 2 transformations to DRAW_PIC through a COMMON block).

Animation with GAL

GAL is intended largely as an animation package and there is a set philosophy as to how animation is performed. All animated GAL applications are run through a repetitive callback. This means that you do not have to run routines by looping with a counter or conditional statement. The callback routine that is passed to GAL_ANIMATE_BUTTON is called repetitively when the corresponding button is pressed, and toggles off when it is pressed again. The callback routine for the button is typically the routine that performs the Model Transformation (which in turn calls the drawing routine).

Double buffering is usually used to remove the flicker that is observed when a new frame succeeds another during animation. Double buffering uses two regions to store an image. One buffer is visible and the information it holds is displayed on the

screen. The other buffer is hidden, and the next frame of the animation is written to it. When the buffers are switched, the input to the screen refresh cycle is switched, and the contents of the hidden buffer “wash in” smoothly. This process is repeated for an entire sequence of frames. This buffering is visually more appealing than the flicker observed in non-buffered animation due to the screen being erased and then redrawn.

Tutorial 4: Animation of the Modeled Object

Animation requires the addition of new features to the code. New invocations to GAL are required. Aside from the animation button discussed above, you must call `GAL_INIT_3D_BUFFER` to initialize the double buffer. The image is now drawn to a hidden canvas, and upon invocation of the `GAL_SWITCH_BUFFER` command, is drawn to the screen.

Conceptually, animation consists of drawing the image, displaying it, drawing the next sequence in the frame (in the hidden buffer) displaying it and so forth. The difference between 2 sequences of a frame is usually some kind of transformation. In the next example, this transformation is a rotation about the z-axis. Listing 4 contains this in the subroutine called `NEXT_FRAME`. The dumbbell is rotated in small increments of `THETA` (the angle of rotation). Holding true to concept of cohesiveness or “single-mindedness” note that initial coordinates of the primitives are now assigned in a separate routine, `INITIALIZE`. The spatial coordinates are accessed by `DRAW_PIC` via a shared `COMMON` block.

Listing 4 Animated GAL program of Rotating Dumbell

C A GAL application to model a 3D object.

PROGRAM LISTING4

EXTERNAL DRAW_PIC, ROTATE, NEXT_FRAME

C Initialize the line primitive

CALL INITIALIZE

CALL GAL_START

CALL GAL_INIT_WINDOW(500, 500, 'Animated Dumbell')

C Register the slider callback

CALL GAL_PANEL_SLIDER1(ROTATE,'Y-Axis Rotate', 180, -180, 0)

CALL GAL_PANEL_SLIDER2(ROTATE,'Z-Axis Rotate', 180, -180, 0)

C Set up the animation button

CALL GAL_ANIMATE_BUTTON(NEXT_FRAME,10, 0, 'Start/Stop')

C Register the callback routine for painting the canvas

CALL GAL_PAINT_PROC(DRAW_PIC)

C Set up default color palette)

CALL GAL_INIT_COLOR

C Set up the double buffer

CALL GAL_SET_3D_BUFFER

C Define the boundaries

CALL GAL_3D_ASPECT_SET(-5., 5., -5., 5., -5., 5.)

CALL GAL_END

END.

C Subroutine INITIALIZE- initializes coordinate and transform data

SUBROUTINE INITIALIZE

C LINE_COORDS - this COMMON block that holds the coordinates of

C of the line segment

COMMON/LINE_COORDS/X(2), Y(2), Z(2)

C SPHERE_COORDS - this COMMON block that holds the parameters of the
C two spheres along with the rotation angle of the Coordinate

C Transformation

COMMON/SPHERE_COORDS/RADIUS, SX(2), SY(2), SZ(2), THETA

C Initialize the line segment start and finish coordinates

X(1) = -1.

X(2) = 1.

Y(1) = 0.

Y(2) = 0.

Z(1) = 0.

Z(2) = 0.

C Initialize the sphere parameters

RADIUS = 0.5

SX(1) = -1.5

SX(2) = 1.5

SY(1) = 0.

SY(2) = 0.

SZ(1) = 0.

SZ(2) = 0.

THETA = 0.1

RETURN

END

C Subroutine ROTATE - rotates the dataset then calls DRAW_PIC

C called from LISTING3

SUBROUTINE ROTATE

REAL Y_ANGLE, Z_ANGLE

C It is VERY IMPORTANT that the slider accessing functions get
C declared as integers

```
INTEGER GET_SLIDER1_VALUE, GET_SLIDER2_VALUE
```

C Determine the rotation factors (in degrees) and convert them
C to reals, using the library FLOATJ() function

```
Z_ANGLE = FLOATJ(GET_SLIDER1_VALUE())
```

```
Y_ANGLE = FLOATJ(GET_SLIDER2_VALUE())
```

C Carry out the rotation

```
CALL GAL_ROTATE(0., Y_ANGLE, Z_ANGLE)
```

C Draw the pictures.

```
CALL DRAW_PIC
```

```
RETURN
```

```
END
```

C Subroutine DRAW_PIC - draws the primitives and switches buffer

```
SUBROUTINE DRAW_PIC
```

```
COMMON/LINE_COORDS/X(2), Y(2), Z(2)
```

```
COMMON/SPHERE_COORDS/RADIUS, SX(2), SY(2), SZ(2), THETA
```

C Draw wireframe spheres

```
CALL GAL_3D_UNFILLED_SPHERE(2, RADIUS, SX(1), SY(1), SZ(1))
```

```
CALL GAL_3D_UNFILLED_SPHERE(3, RADIUS, SX(2), SY(2), SZ(2))
```

C Draw line

```
CALL GAL_3D_SOLID_LINE(1, 3.0, 2, x, y, z)
```

C Switch the buffer now. This should be the last act in the

C drawing routine

```
CALL GAL_SWITCH_BUFFER
```

```
RETURN
```

END

C Subroutine NEXT_FRAME carries out the frame to frame transformation

SUBROUTINE NEXT_FRAME

COMMON/LINE_COORDS/X(2), Y(2), Z(2)

COMMON/SPHERE_COORDS/RADIUS, SX(2), SY(2), SZ(2), THETA

REAL SX_OLD(2), X_OLD(2)

C Carry out the transformations required to rotate about the z-axis

DO 100 I=1,2

X_OLD(I) = X(I)

SX_OLD(I) = SX(I)

X(I) = X(I)*COS(THETA)-Z(I)*SIN(THETA)

Z(I) = X_OLD(I)*SIN(THETA)-Z(I)*COS(THETA)

SX(I) = SX(I)*COS(THETA)-SZ(I)*SIN(THETA)

SZ(I) = SX_OLD(I)*SIN(THETA)-SZ(I)*COS(THETA)

100 CONTINUE

C Now that all the transformations are carried out, draw the picture

DRAW_PIC

RETURN

END

Exercise 5 Run the program and observe the behaviour of the animated object as you toggle animation on and off. Also note that you can change the view of the system by adjusting the sliders, either during animation or while the animation is paused. You should find that you can adjust the dumbbell to spin about any position in space.

Exercise 6 Change the drawing primitive for the sphere from `GAL_3D_UNFILLED_SPHERE` to `GAL_3D_FILLED_SPHERE`. Adjust your view of the dumbbell until you can see one sphere pass in front of the other and then behind it. What is wrong with the representation? Can you see that the program does not have any depth perception?

Depth Perception and the GAL Visual Environment.

If you have looked at the Tutorial and demonstration programs, you may have already noticed that the Visual Environment created by GAL applications has a specific “motif” or “feel” to them. Representing a 3D space on a 2D surface always results in a distortion resulting from the loss of visual information. You may imagine that the object that you see on the screen has been projected onto the screen. An analogy would be projection by a light source.

The projection model used is formally known as a parallel orthographic projection. This means that the Center of Projection (the light source in our analogy) is at infinity and the parallel Projectors (light rays) fall orthogonally or perpendicularly to the projection surface. The resulting projected image has some specific features.

A sense of perspective is provided by the shear transformation of “foreshortened” objects (objects that do not lie completely in the plane of the screen). The dimensions of objects on the screen are not dependent on their distance from the screen and are not appropriately scaled as in Perspective projections. Therefore, objects of the same physical dimensions at different distances from the screen will appear to have the same physical dimensions.

The shear transformation preserves the parallelism of parallel lines (but it does not preserve angles). All 2D objects in GAL exist in the plane of the screen and none of the features discussed apply.

The representation of solid objects in 3D usually requires some kind of hidden line or hidden surface removal. In GAL, this capability is activated by invoking the library call `GAL_SET_ZBUFFER`. The z-buffering is the technique used to determine the surface that is closest to the user and display the correct pixel value. The net result is that obscured parts of the objects are not drawn to the screen. The z-buffer has to be reset before every frame is drawn to the screen, and one should remember this when doing animation. The z-buffer is reset by invoking `GAL_RESET_ZBUFFER`. Z-Buffering is also required when drawing mesh surfaces with the the quadrilateral mesh primitive.

Listing 5 shows the implementation of z-buffering in our GAL application. The additions to Listing 4 are the implementation of z-buffer settings and the resetting calls, as well as the replacement of wireframe spheres with solid spheres (refer to Exercise 6). Note that z-buffering drastically slows down the speed of the application.

Tutorial 5: Z-Buffered Solid Dumbbells

Listing 5 Animated GAL program of rotating dumbell using solid spheres. The main program and `DRAW_PIC` are listed. The other subroutines remain unchanged.

C A GAL application to model a 3D object.

```
PROGRAM LISTING4
```

```
EXTERNAL DRAW_PIC, ROTATE, NEXT_FRAME
```

C Initialize the line primitive

CALL INITIALIZE

CALL GAL_START

CALL GAL_INIT_WINDOW(500, 500, 'Animated Dumbell')

C Register the slider callback

CALL GAL_PANEL_SLIDER1(ROTATE,'Y-Axis Rotate', 180, -180, 0)

CALL GAL_PANEL_SLIDER2(ROTATE,'Z-Axis Rotate', 180, -180, 0)

C Set up the animation button

CALL GAL_ANIMATE_BUTTON(NEXT_FRAME,10, 0, 'Start/Stop')

C Register the callback routine for painting the canvas

CALL GAL_PAINT_PROC(DRAW_PIC)

C Set up default color palette)

CALL GAL_INIT_COLOR

C Set up the z-buffer

CALL GAL_SET_ZBUFFER

C Set up the double buffer

CALL GAL_SET_3D_BUFFER

C Define the boundaries

CALL GAL_3D_ASPECT_SET(-5., 5., -5., 5., -5., 5.)

CALL GAL_END

END.

C Subroutine DRAW_PIC - draws the primitives and switches buffer

SUBROUTINE DRAW_PIC

COMMON/LINE_COORDS/X(2), Y(2), Z(2)

COMMON/SPHERE_COORDS/RADIUS, SX(2), SY(2), SZ(2), THETA

C Reset the z-buffer before drawing the next frame

```
CALL GAL_RESET_ZBUFFER

C Draw solid spheres

CALL GAL_3D_FILLED_SPHERE(2, RADIUS, SX(1), SY(1), SZ(1))

CALL GAL_3D_FILLED_SPHERE(3, RADIUS, SX(2), SY(2), SZ(2))

C Draw line

CALL GAL_3D_SOLID_LINE(1, 3.0, 2, x, y, z)

C Switch the buffer now. This should be the last act in the
C drawing routine

CALL GAL_SWITCH_BUFFER

RETURN

END
```

Exercise 7 Z-buffering slows down the speed of the application. Try using larger time (THETA) increments to speed up the animation. Also try resizing the window while animation is in progress. Note that z-buffering is canvas size dependent.

The Structure of GAL Programs

By looking over the code in the tutorials as well as the demonstration programs, you may notice that GAL applications are written with a certain structure. The cohesiveness of the modules allows for reuseable and modifiable code. Typically, the program is divided into 4 components: initialization, view transformation, model transformation and drawing. Initialization is usually only called once from in the MAIN routine. View transformation subroutines are typically callback subroutines associated with the sliders or other manipulators. Model transformation is the step that causes the frame change observed with animation. Drawing is the final step in

the rendering process. Figure 6 illustrates a schematic of a typical animation program. The “chaining” of two or more routines emphasizes the order in the flow of command.

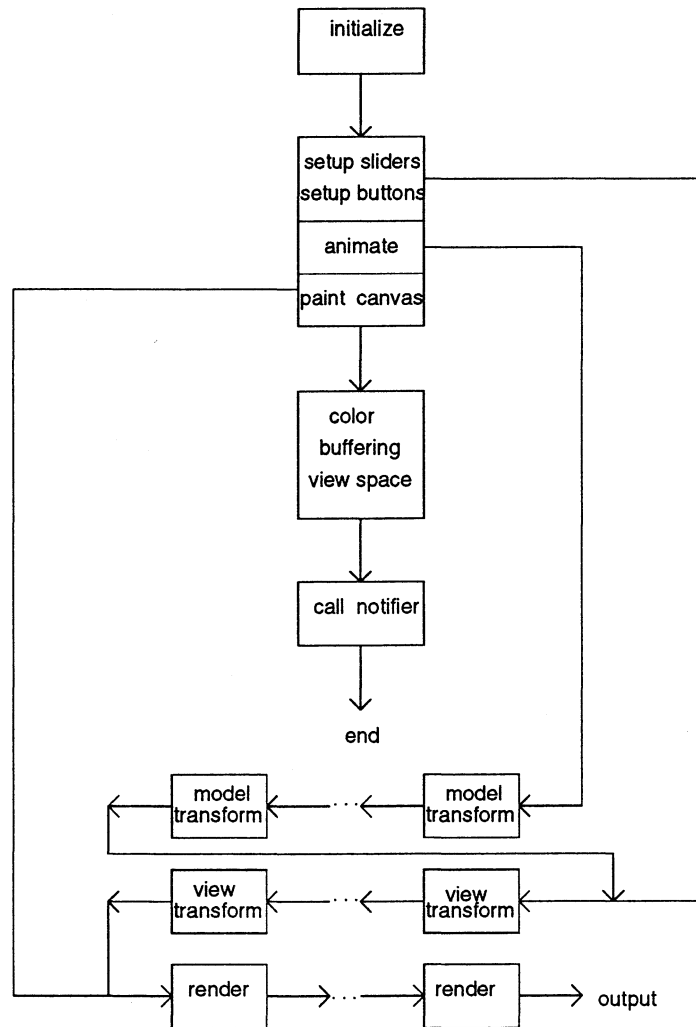


Figure 6. Flow chart showing the dynamic and static aspects of a typical GAL program. The upper portion of the diagram contains the Main routine. The notifier “threads” a series of callback routines that are chained together, forming a notifier-driven rendering/animation pipeline.

Common Errors Encountered in GAL Programs

Anyone who has ever programmed is no stranger to programming errors. Along with all the typical errors the FORTRAN programmer will encounter, there are some that are specific or occur frequently in GAL programs. Here are some of them. The types of errors made in writing GAL programs are typically of 3 types.

1. Errors in the ordering of GAL Primitives.

Symptom : The program aborts with an error message, e.g.

```
***** GAL ERROR *****
```

```
ERROR - UNABLE TO PAINT IN GAL_PAINT_PROC, CHECK THAT GAL_INIT_WINDOW
IS SET FIRST.
```

```
*****
```

Cure : Check the ordering of the GAL calls in your MAIN routine. This is usually due to a problem with the order of the GAL calls.

2. Mismatched\Erroneous subroutine parameters.

Symptom : Program crashes.

Some typical error messages:

```
*** Illegal = signal 4 code 2
```

- You have forgotten to declare a callback routine as external and the FORTRAN compiler has assumed that it is an integer/real.

```
*** Segmentation Violation = signal # code #
```

- A mismatch of parameters has occurred, e.g. an integer has been passed when a real should have been, etc.

Error number -5: -5

Operator: xgl_stroke_text

Operand: ***

- A font for the annotation primitives cannot be found. Any error messages of this form are from XGL and are typically due to improper initialization in the GAL MAIN routine or the creation of graphics that take up excessive amounts of memory.

memory allocation request in gal_2d_filled_polygon failed

- An extremely large number has been passed as a parameter for the number of points. Often symptomatic of an incorrect parameter list.

Symptom : The program runs, but a distorted window or graphic is observed.

These are often the hardest errors to track. A distorted frame is often the result of erroneous parameters being passed to the GAL_PANEL_BUTTON and GAL_PANEL_SLIDER library routines. If the panel is covered by the drawing surface(canvas), then it is usually because you have called GAL_PAINT_PROC before GAL_PANEL_BUTTON and GAL_PANEL_SLIDER. Circular buttons imply that the button labels have been incorrectly passed. The overlapping of sliders and buttons restricts their utility ...make sure that you keep them clear of each other.

Distorted graphics are often the result of incorrect parameter types being passed. Make sure that you pass an array when an array is required by the library. Beware of typos that become automatically defined variables!

3. Algorithm or design errors.

Symptoms : The program compiles and runs without crashing. However, the program does not function as intended. Graphical operations appear in the incorrect

sequences.

Cure : Check the logic of your design. By adhering to the recommended structure, as shown in the tutorials and demonstrations, there should be little confusion as to the order of execution of the various callbacks.

GAL Reference Manual.

XView Tools

gal_end

No parameters.

This call is made after all other GAL calls have been made. It sets up the event loop that checks for callback notifiers.

gal_init_window(width, height, label)

INTEGER width : window width in pixels

INTEGER height : window height in pixels

CHARACTER label: window title

The XView Window is created by this application. A control panel is also built. The panel size is automatically controlled by the positions of sliders and buttons.

gal_paint_proc(paint_proc)

EXTERNAL paint_proc

This library call creates the canvas and tells the application what to paint to the canvas. The canvas is painted by the procedure “paint_proc” that the user has specified. This applies when the canvas is being repainted, for example in animation, or when another window is placed in front of it and removed.

gal_panel_button(button_proc, x_pos, y_pos, label)

EXTERNAL button_proc : the subroutine invoked when the button is pressed.

INTEGER x_pos : x position on the panel (1 unit = 10 pixels)

INTEGER y_pos : y position on the panel (1 unit = 30 pixels)

CHARACTER label : button label

This creates a panel button. y position = 0 is closest to the top of the panel. The panel automatically resizes as buttons are added. The button width is also automatically sized to accommodate the width of the label. Button color, as with panel color, is set from the OpenWindows desktop. The ranges for x_pos and y_pos are set by the dimensions of the window (refer to gal_init_window).

gal_start

No parameters.

This is the first statement of a GAL application. It initializes some GAL variables that are used by the library. All statements of a GAL application are bracketed between gal_start and gal_end.

gal_panel_slider#(slider_proc, label, max, min, y_pos)

EXTERNAL slider_proc : the subroutine invoked when the slider is touched.

CHARACTER label : the label for the slider.

INTEGER max : the maximum value of the slider position.

INTEGER min : the minimum value of the slider position.

INTEGER y_pos : the y position of the slider on the panel. The panel is automatically sized to accommodate a slider.

A series of sliders, where # ranges from 1 to 6, can be built by invoking this call. A slider is useful for representing a continuous range of values. The process slider_proc would contain the function get_slider#_value from which the current value of the slider would be obtained. The default initial reading of the slider is 0, or the closest value to it in the range max - min.

get_slider#_value

No parameters. A FUNCTION returning type INTEGER.

This is a special GAL function that returns the current setting of the #th slider. Note that it is vital that the this function be declared as an integer if it is used, and that it always returns a value of type INTEGER. For use as a REAL or any other type, the slider value must be converted into a REAL. Note that the fineness of control can be “tuned” by multiplying the scale by a constant and then dividing to convert to the appropriate units.

gal_quit_proc

No parameters.

A call to this procedure destroys the window.

Color and Animation

gal_init_color

No parameters.

This call sets up the color tables and initializes the structures that will form the 2D and 3D objects being drawn. This call must be made before the dimensions of the view space, color index changes, the setting of double buffers and/or z buffers are invoked. The color system that is supported is Indexed color. Each color is associated with of “indexed” with a number. Color monitors use an additive color scheme with red (R), blue (B) and green (G) as the primary colors. For example, $R = 1.0$, $B = 1.0$, $G = 1.0$ gives white. $R = 0.0$, $B = 0.0$, $G = 0.0$ gives black. Between these two extremities lies the spectrum of usable colors. The user chooses a color by passing its index to the primitives. The background is defaulted to black (index 0).

The following are the defaults for the color indices:

index 0	black	$R = 0.0, G = 0.0, B = 0.0$
index 1	white	$R = 1.0, G = 1.0, B = 1.0$
index 2	red	$R = 1.0, G = 0.0, B = 0.0$
index 3	green	$R = 0.0, G = 1.0, B = 0.0$
index 4	blue	$R = 0.0, G = 0.0, B = 1.0$
index 5	yellow	$R = 1.0, G = 1.0, B = 0.0$
index 6	cyan	$R = 0.0, G = 1.0, B = 1.0$
index 7	magenta	$R = 1.0, G = 0.0, B = 1.0$

Refer to `gal_reset_color` for changing the defaults. This call must be made before the dimensions of the view space, color index changes, the setting of double buffers and/or z buffers are invoked.

gal_set_2d_buffer, gal_set_3d_buffer

No parameters.

These set up the double buffer for 2D and 3D objects, respectively. Only one of the two can be invoked per application. If an attempt is made to initialize both, an error message results. May only be set after gal_init_color.

gal_new_2d_frame, gal_new_3d_frame

No parameters.

These commands clear the GAL canvas, which reverts to the background color. Note that the command to clear a 2D canvas also results in any 3D objects on the screen being cleared as well, and vice versa.

gal_switch_buffer

No parameters.

Switching of the hidden and displayed buffers is accomplished by making a call to this routine. Once the buffers are set up, all drawing is done to the hidden buffer. No image will be seen until gal_switch_buffer is invoked. Conversely, anything drawn after the call is made will not be viewed until the call to switch the buffers is made again.

gal_set_zbuffer

No parameters.

This call sets up the z-buffer. It applies only to 3D objects.

gal_reset_zbuffer

No parameters.

Once the z buffer has been set up, it must be reset or cleared before the drawing of

each new frame. Failure to do so will result in all the previous images remaining on the screen.

gal_reset_color(index, red, green, blue)

INTEGER index

REAL red, green, blue

If the platform being used utilizes an Indexed Color Scheme (as is the most common), the default color table can be altered. The color indexed by x can be changed to the color $R = r$, $G = g$ and $B = b$ by simply invoking `gal_reset_color(x, r, g, b)`.

gal_animate_button(button_proc, x_pos, y_pos, label)

EXTERNAL button_proc : the subroutine is called repeatedly when the button is pressed and then stops when pressed again.

INTEGER x_pos : x position on the panel (1 unit = 10 pixels)

INTEGER y_pos : y position on the panel (0 through 6, 1 unit = 10 pixels)

CHARACTER label : button label

This is the animation button that works as a repeated call back, calling button_proc repeatedly. button_proc should be the routine that advances the frame of the animation. To stop the animation, press the button again.

gal_step_anim_button(button_proc, x_pos, y_pos, label)

EXTERNAL button_proc : the subroutine is called once when the button is pressed.

INTEGER x_pos : x position on the panel (1 unit = 10 pixels)

INTEGER y_pos : y position on the panel (0 through 6, 1 unit = 10 pixels)

CHARACTER label : button label

This button advances animation frame-by-frame. It is a customized button, but

could have been made by simply invoking the frame advancing routine by a generic button. The advantage of this customized button is that the user does not have to stop animation before advancing frame by frame.

View Transformation Primitives

gal_3d_pan_and_zoom(x_translate, y_translate, zoom)

gal_2d_pan_and_zoom(x_translate, y_translate, zoom)

REAL x_translate : the x-axis translation (in Model coordinates)

REAL y_translate : the y-axis translation (in Model coordinates)

REAL zoom : the scaling factor of the objects (> 0)

Panning and zooming are terms associated with camera or film work. Panning involves translating the object across the viewport, with no change in perspective. Zooming increases (zoom factor > 1) or decreases the size (zoom < 1) of the object on the screen. Zoom factors having a value < 0 are read as a value of 1. When zooming interactively (with a slider, for example) one tends to zoom in on the origin. Some degree of panning is often required to bring the desired region into the viewport.

gal_rotate(x_angle, y_angle, z_angle)

REAL x_angle, y_angle, z_angle : rotations about the x-axis, y-axis and z-axis, respectively, in degrees.

Rotations are only applicable to 3D objects. Some specific rules apply when using rotations:

1. All rotations are about one of the 3 principal axes (i.e. x, y or z)
2. The rotational scheme is such that any rotation, no matter what order the rotational data for the axes is input, is executed in a particular order. The unrotated object is first rotated by x_angle about the x-axis, followed by a y_angle rotation about the y axis and finally a z_angle rotation about the z-axis. This sequence is repeated each time this routine is called, and successive rotations about a given axis

are not cumulative. In other words, a succeeding x, y, z-rotation replaces the previous x, y, z rotation sequence.

GAL Graphics Primitives

2D Primitives

gal_2d_filled_circle(color_index, radius, x_coord, y_coord)

INTEGER color_index : ranges from 0 through 7 for 8 colors

REAL radius : circle radius in 2D Model Coordinates.

REAL x_coord : the x-axis coordinate of the circle centre.

REAL y_coord : the y-axis coordinate of the circle centre.

This draws a disk.

gal_2d_unfilled_circle(color_index, radius, x_coord, y_coord)

INTEGER color_index : ranges from 0 through 7 for 8 colors

REAL radius : circle radius in 2D Model Coordinates.

REAL x_coord : the x-axis coordinate of the circle centre.

REAL y_coord : the y-axis coordinate of the circle centre.

This draws a circle.

gal_2d_filled_rectangle(color_index, x_corner, y_corner)

INTEGER color_index : ranges from 0 through 7 for 8 colors

ARRAY REAL x_corner(2) : the maximum and minimum values of the x-coordinates.

ARRAY REAL y_corner(2) : the maximum and minimum values of the y-coordinates.

Draws a filled rectangle with corners at (x_corner(1), y_corner(1)) and (x_corner(2), y_corner(2)).

gal_2d_unfilled_rectangle(color_index, x_corner, y_corner)

INTEGER color_index : ranges from 0 through 7 for 8 colors

ARRAY REAL x_corner(2) : the maximum and minimum values of the x-coordinates.

ARRAY REAL y_corner(2) : the maximum and minimum values of the y-coordinates.

Draws an unfilled rectangle with corners at (x_corner(1), y_corner(1)) and (x_corner(2), y_corner(2)).

gal_2d_unfilled_polygon(color_index, num_pts, list_x, list_y)

INTEGER color_index : ranges from 0 through 7 for 8 colors

INTEGER num_pts : the number of vertices a 2D polygon.

REAL list_x : an array list_x(num_pts) of x-coordinates

REAL list_y : an array list_y(num_pts) of y-coordinates

A polygon with vertex list ({list_x(1), list_y(1)}, {list_x(2), list_y(2)}, ..., {list_x(num_pts), list_y(num_pts)}) is drawn to the screen. The vertex list represents a polygon where the n-1, n and n+1 pairs are joined by line segments. The first and last sets in the list are also joined by a line segment. (The set {x,y} corresponds to a the coordinates of the point (x,y)).

gal_2d_filled_polygon(color_index, num_pts, list_x, list_y)

Refer to gal_2d_unfilled_polygon.

gal_2d_????_line(color_index, thickness, num_pts, list_x, list_y)

????? = **solid** gives _____

????? = **dotted** gives

????? = **dashed** -----

????? = **dash_dotted** ..._..._

????? = **dash_dot** ._._._.

????? = **dash_dot_dot** _.._..._..

????? = **long_dash** 

INTEGER color_index : ranges from 0 through 7 for 8 colors

INTEGER num_pts : the number line segments in a polyline.

REAL thickness : line thickness

ARRAY REAL list_x(num_pts) : an array list_x(num_pts) of x-coordinates

ARRAY REAL list_y(num_pts) : an array list_y(num_pts) of y-coordinates

This call draws a polyline (a series of line segments joined end-to-end). The list contains the elements ($\{list_x(1), list_y(1)\}, \{list_x(2), list_y(2)\}, \dots, \{list_x(num_pts), list_y(num_pts)\}$) where sets $n-1$, n and $n+1$ are connected by line segments (where $n > 1$ and $n < num_pts$). Line thickness is an arbitrary thickness scale, where thickness = 1.0 is the thinnest line setting. (The set $\{x,y\}$ corresponds to a the coordinates of the point (x,y)).


gal_2d_?????_marker(color_index, num_pts, list_x, list_y)

????? = **asterisk** gives *

????? = **circle** gives \circ

????? = **cross** gives \times

????? = **plus** gives +

????? = **square** gives 

INTEGER color_index : ranges from 0 through 7 for 8 colors

INTEGER num_pts : the number of markers to be drawn

ARRAY REAL list_x(num_pts) : an array list_x(num_pts) of x-coordinates

ARRAY REAL list_y(num_pts) : an array list_y(num_pts) of y-coordinates

This call draws a series of markers drawn from a list. The list contains the elements ($\{list_x(1), list_y(1)\}, \{list_x(2), list_y(2)\}, \dots, \{list_x(num_pts), list_y(num_pts)\}$) where each set pair represents a marker coordinate. The user has a choice of 5

marker styles, as described above. The marker sizes are not adjustable, and are set to about 20 pixels across in size.

gal_2d_text_annotate(color_index, string, font_size, font, font_spacing, x_pos, y_pos, x_vector, y_vector)

INTEGER color_index : ranges from 0 through 7 for 8 colors

CHARACTER string : the string to be rendered (not more than 80 characters).

REAL font_size : a scaling factor for the characters

CHARACTER font : a selected XGL font

REAL font_spacing : spacing between characters

REAL x_pos : the x-axis coordinate, in Model Coordinates

REAL y_pos : the y-axis coordinate, in Model Coordinates

REAL x_vector : the x-component of the line vector

REAL y_vector : the y-component of the line vector

Annotation text is added by invoking this routine. Text should only be added as an overlay to a figure/model, and its implementation greatly slows down the animation speed. Text characters are transformed by the transformation calls and their use in animation is not advisable. The use of a toggle to turn text annotation on and off is probably the best approach.

The fonts available are : Cartographic, Cartographic_M, English_G, Greek, Greek_C, Greek_M, Headline, Italic_C, Italic_T, Miscellaneous, Miscellaneous_M, Roman, Roman_C, Roman_D, Roman_M, Roman_T, Script, Script_C. (Note the naming convention - lowercase with capitalized letter of first word - is essential for proper usage).

Text can be scaled, the separation between characters can be controlled, its position and alignment can also be altered. x_vector and y_vector control the alignment

of a line of text. For example, $x_vector = 1.0$ and $y_vector = 0.0$ give horizontal text. On the other hand, $x_vector = 0.0$ and $y_vector = 1.0$ gives vertical text.

gal_2d_real_annotate(color_index, number_string, field_width, decimal_places, font_size, font, font_spacing, x_pos, y_pos, x_vector, y_vector)

INTEGER color_index : ranges from 0 through 7 for 8 colors

REAL number_string : the numeric label

INTEGER field_width : the number of significant digits

INTEGER decimal_place : the number of decimal places

REAL font_size : a scaling factor for the characters

CHARACTER font : a selected XGL font

REAL font_spacing : spacing between characters

REAL x_pos : the x-axis coordinate, in Model Coordinates

REAL y_pos : the y-axis coordinate, in Model Coordinates

REAL x_vector : the x-component of the line vector

REAL y_vector : the y-component of the line vector

This library call is used to display numeric labels in GAL. Its purpose is to allow for the display of computed values. A real value is passed in the form of the parameter number_string. The number of significant figures is determined by field_width. The routine rounds off insignificant figures. If the number or decimal places is set equal to 0, an integer is displayed. Refer to gal_2d_text_annotate for details on the remaining parameter values.

gal_2d_axes(color_index, xmin, xmax, ymin, ymax, x_interval, y_interval)

INTEGER color_index : ranges from 0 through 7 for 8 colors

REAL xmin : the minimum of the x-axis

REAL xmax : the maximum of the x-axis

REAL ymin : the minimum of the y-axis

REAL ymax : the maximum of the y-axis

REAL x_interval : x-interval

REAL y_interval : y-interval

This routine draws a set of axes, an x-axis (length xmax-xmin, with tic marks separated by x_interval) and a y-axis (length ymax-ymin, with tic marks separated by x_interval). Currently, labels must be added through an text or real number annotation “overlay”.

gal_2d_filled_multicircle(color_index, num_circles, radius, x_coords, y_coords)

INTEGER color_index : ranges from 0 through 7 for 8 colors

INTEGER num_circles : the number of circles to be processed

ARRAY REAL radius(num_circles) : array of radii of the list of circles.

ARRAY REAL x_coord(num_circles) : the x-axis coordinate of the circle centre.

ARRAY REAL y_coord(num_circles) : the y-axis coordinate of the circle centre.

Draws a series of filled circles of the same color. A list of circle data is passed in the form of one array each for radii, x coordinates and y coordinates. All three arrays are of length num_circles. The ith circle has a radius radius(i) and center (x_coord(i), y_coord(i)).

gal_2d_unfilled_multicircle(color_index, num_circles, radius, x_coords, y_coords)

INTEGER color_index : ranges from 0 through 7 for 8 colors

INTEGER num_circles : the number of circles to be processed

ARRAY REAL radius(num_circles) : array of radii of the list of circles.

ARRAY REAL x_coord(num_circles) : the x-axis coordinate of the circle centre.

ARRAY REAL y_coord(num_circles) : the y-axis coordinate of the circle centre.

Draws a series of unfilled circles of the same color. A list of circle data is passed in the form of one array each for radii, x coordinates and y coordinates. All three arrays are of length num_circles. The ith circle has a radius radius(i) and center (x_coord(i), y_coord(i)).

gal_2d_filled_multirectangle(color_index, num_rectangles, x_corner, y_corner)

INTEGER color_index : ranges from 0 through 7 for 8 colors

INTEGER num_rectangles : the number of circles to be processed

ARRAY REAL x_corner(2×num_rectangles) : x axis maxima and minima for rectangles

ARRAY REAL y_corner(2×num_rectangles) : y axis maxima and minima for rectangles

Draws a series of filled rectangles. The maximum and minimum corners of the rectangles are specified by (x_corner(i), y_corner(i)) and (x_corner(i+1), y_corner(i+1)), where i is an odd-number.

gal_2d_unfilled_multirectangle(color_index, num_rectangles, x_corner, y_corner)

INTEGER color_index : ranges from 0 through 7 for 8 colors

INTEGER num_rectangles : the number of circles to be processed

ARRAY REAL x_corner(2×num_rectangles) : x axis maxima and minima for rectangles

ARRAY REAL y_corner(2×num_rectangles) : y axis maxima and minima for rectangles

Draws a series of unfilled rectangles. The maximum and minimum corners of the rectangles are specified by (x_corner(i), y_corner(i)) and (x_corner(i+1), y_corner(i+1)),

where i is an odd-number.

gal_solid_contour(color_index, thickness, row_size, column_size, level_interval, xmin, xmax, ymin, ymax, data_field)

INTEGER color_index : ranges from 0 through 7 for 8 colors

REAL thickness : thickness of the contour lines

REAL row_size : number of row elements in the data field.

REAL column_size : number of column elements in the data field.

REAL level_interval : the interval between contour levels

REAL xmin : the minimum x value

REAL xmax : the maximum x value

REAL ymin : the minimum y value

REAL ymax : the maximum y value

ARRAY REAL data_field(row_size×column_size) : array of z-values of data field, starting with point (xmin, ymin), ending with (xmax, ymax).

Contour plots of a uniformly-spaced data field can be generated when the z-axis values are entered as a vector array data_field. The array data_field has a dimension row_size×column_size and consists of ordered elements of z for all coordinates (x,y). Let the row_size be m and the column_size be n. If $z_{ij} = f(x_i, y_j)$, for the i th x-axis interval and the j th y-axis interval, then the array data_field would be

($f(x_1, y_1),$	$f(x_2, y_1),$	\dots	$f(x_m, y_1),$
	$f(x_1, y_2),$	$f(x_2, y_2),$	\dots	$f(x_m, y_2),$

	$f(x_1, y_n),$	$f(x_2, y_n),$	\dots	$f(x_m, y_n))$

The contour plotter assumes uniform intervals between the sampled points (starting at x and y minima and ending at their respective maxima). The algorithm relies upon interpolation between points of a uniform rectangular grid of z-axis data. The interval between z-levels of the contour is set with the variable `level_interval`. 8 choices of color are available (including background) and the line thickness is set to 1.0 or greater (anything less than 1.0 will be defaulted to 1.0).

gal_dotted_contour(data_field, row_size, column_size, level_interval, xmin, xmax, ymin, ymax, color_index, thickness)

ARRAY REAL data_field(row_size×column_size) : z-values of data field, starting with point (xmin, ymin), ending with (xmax, ymax).

REAL row_size : number of row elements in the data field.

REAL column_size : number of column elements in the data field.

REAL level_interval : the interval between contour levels

REAL xmin : the minimum x value

REAL xmax : the maximum x value

REAL ymin : the minimum y value

REAL ymax : the maximum y value

INTEGER color_index : ranges from 0 through 7 for 8 colors

REAL thickness : thickness of the contour lines

Identical to `gal_solid_contour`, except that a dotted line is drawn instead of a solid line.

GAL Graphics Primitives

3D Primitives

gal_3d_filled_sphere(color_index, radius, x_coord, y_coord, z_coord)

INTEGER color_index : ranges from 0 through 7 for 8 colors

REAL radius : sphere radius in 3D Model Coordinates.

REAL x_coord : the x-axis coordinate of the sphere centre.

REAL y_coord : the y-axis coordinate of the sphere centre.

REAL z_coord : the z-axis coordinate of the sphere centre.

This draws a solid sphere. Its actual implementation is through so-called *annotation* circles. Annotation circles are circles that always maintain the same orientation in relation to the plane of the screen, given a centre point and a radius.

gal_3d_unfilled_sphere(color_index, radius, x_coord, y_coord, z_coord)

INTEGER color_index : ranges from 0 through 7 for 8 colors

REAL radius : sphere radius in 2D Model Coordinates.

REAL x_coord : the x-axis coordinate of the sphere centre.

REAL y_coord : the y-axis coordinate of the sphere centre.

REAL z_coord : the z-axis coordinate of the sphere centre.

This draws a set of 3 orthoplanar circles that form a wireframe sphere.

gal_3d_unfilled_polygon(color_index, num_pts, list_x, list_y, list_z)

INTEGER color_index : ranges from 0 through 7 for 8 colors

INTEGER num_pts : the number of vertices a 3D polygon.

ARRAY REAL list_x(num_pts) : an array list_x(num_pts) of x-coordinates

ARRAY REAL list_y(num_pts) : an array list_y(num_pts) of y-coordinates

ARRAY REAL list_z(num_pts) : an array list_z(num_pts) of z-coordinates

A polygon with vertex list ($\{list_x(1), list_y(1), list_z(1)\}, \{list_x(2), list_y(2), list_z(2)\}, \dots, \{list_x(num_pts), list_y(num_pts), list_z(num_pts)\}$) is drawn to the screen.

The vertex list represents a polygon where the $n-1$, n and $n+1$ sets are joined by line segments. (The set $\{x,y,z\}$ corresponds to a the coordinates of the point (x,y,z)). The first and last sets in the list are also joined by a line segment.

gal_3d_filled_polygon(color_index, num_pts, list_x, list_y)

Refer to gal_3d_unfilled_polygon.

gal_3d_????_line(color_index, thickness, num_pts, list_x, list_y, list_z)

For ???? designations refer to **gal_2d_????_line**

INTEGER color_index : ranges from 0 through 7 for 8 colors

INTEGER num_pts : the number line segments in a polyline.

REAL thickness : line thickness

ARRAY REAL list_x(num_pts) : an array list_x(num_pts) of x-coordinates

ARRAY REAL list_y(num_pts) : an array list_y(num_pts) of y-coordinates

ARRAY REAL list_z(num_pts) : an array list_z(num_pts) of z-coordinates

This call draws a polyline (a series of line segments joined end-to-end). The polyline list contains the elements ($\{list_x(1), list_y(1), list_z(1)\}, \{list_x(2), list_y(2), list_z(2)\}, \dots, \{list_x(num_pts), list_y(num_pts), list_z(num_pts)\}$) where sets $n-1$, n and $n+1$ are connected by line segments (where $n > 1$ and $n < num_pts$). (The set $\{x,y,z\}$ corresponds to a the coordinates of the point (x,y,z)). Line thickness is an arbitrary thickness scale, where thickness = 1.0 is the thinnest line setting.

gal_3d_?????_marker(color_index, num_pts, list_x, list_y)

Refer to **gal_2d_????_marker** for the ????? designations.

INTEGER color_index : ranges from 0 through 7 for 8 colors

INTEGER num_pts : the number of markers to be drawn

ARRAY REAL list_x(num_pts) : an array list_x(num_pts) of x-coordinates

ARRAY REAL list_y(num_pts) : an array list_y(num_pts) of y-coordinates

ARRAY REAL list_z(num_pts) : an array list_z(num_pts) of z-coordinates

This call draws a series of markers drawn from a list. The list contains the elements $(\{list_x(1), list_y(1)\}, \{list_x(2), list_y(2)\}, \dots, \{list_x(num_pts), list_y(num_pts)\})$ where each set represents a marker coordinate. (The set $\{x,y,z\}$ corresponds to a the coordinates of the point (x,y,z)). The user has a choice of 5 marker styles, as described above. The marker sizes are not adjustable, and are set to about 20 pixels across in size.

gal_3d_text_annotate(color_index, string, font_size, font, font_spacing, x_pos, y_pos, z_pos, x_vector, y_vector)

INTEGER color_index : ranges from 0 through 7 for 8 colors

CHARACTER string : the string to be rendered (not more than 80 characters).

REAL font_size : a scaling factor for the characters

CHARACTER font : a selected XGL font

REAL font_spacing : spacing between characters

REAL x_pos : the x-axis coordinate, in Model Coordinates

REAL y_pos : the y-axis coordinate, in Model Coordinates

REAL z_pos : the z-axis coordinate, in Model Coordinates

REAL x_vector : the x-component of the line vector

REAL y_vector : the y-component of the line vector

Annotation text is added by invoking this routine. Text should only be added as an

overlay to a figure/model, and its implementation greatly slows down the animation speed. Text characters are transformed by the transformation calls and their use in animation is not advisable. The use of a toggle to turn text annotation on and off is probably the best approach.

The fonts available are : Cartographic, Cartographic_M, English_G, Greek, Greek_C, Greek_M, Headline, Italic_C, Italic_T, Miscellaneous, Miscellaneous_M, Roman, Roman_C, Roman_D, Roman_M, Roman_T, Script, Script_C. (Note the naming convention - lowercase with capitalized letter of first word - is essential for proper usage).

Text can be scaled, the separation between characters can be controlled, its position and alignment can also be altered. *x_vector* and *y_vector* control the alignment of a line of text. For example, *x_vector* = 1.0 and *y_vector* = 0.0 give horizontal text. On the other hand, *x_vector* = 0.0 and *y_vector* = 1.0 gives vertical text. In 3D, all text is defaulted to be in the plane of screen.

gal_3d_real_annotate(color_index, number_string, field_width, decimal_places, font_size, font, font_spacing, x_pos, y_pos, z_pos, x_vector, y_vector)

INTEGER color_index : ranges from 0 through 7 for 8 colors

REAL number_string : the numeric label

INTEGER field_width : the number of significant digits

INTEGER decimal_place : the number of decimal places

REAL font_size : a scaling factor for the characters

CHARACTER font : a selected XGL font

REAL font_spacing : spacing between characters

REAL x_pos : the x-axis coordinate, in Model Coordinates

REAL y_pos : the y-axis coordinate, in Model Coordinates

REAL z_pos : the z-axis coordinate, in Model Coordinates

REAL x_vector : the x-component of the line vector

REAL y_vector : the y-component of the line vector

This library call is used to display numeric labels in GAL. Its purpose is to allow for the display of computed values. A real value is passed in the form of the parameter number_string. The number of significant figures is determined by field_width. The routine rounds off insignificant figures. If the number or decimal places is set equal to 0, an integer is displayed. Refer to gal_3d_text_annotate for details on the remaining parameter values.

gal_3d_axes(color_index, xmin, xmax, ymin, ymax, zmin, zmax, x_interval, y_interval, z_interval)

INTEGER color_index : ranges from 0 through 7 for 8 colors

REAL xmin : the minimum of the x-axis

REAL xmax : the maximum of the x-axis

REAL ymin : the minimum of the y-axis

REAL ymax : the maximum of the y-axis

REAL zmin : the minimum of the z-axis

REAL zmax : the maximum of the z-axis

REAL x_interval : x-interval between tic marks

REAL y_interval : y-interval between tic marks

REAL z_interval : z-interval between tic marks

This routine draws a set of axes, an x-axis (length xmax-xmin, with tic marks separated by x_interval) y-axis (length ymax-ymin, with tic marks separated by x_interval) and a z-axis (length zmax-zmin, with tic marks separated by z_interval). Currently, labels must be added through an text or real number annotation “overlay”.

gal_3d_filled_multisphere(color_index, num_spheres, radius, x_coord, y_coord, z_coord)

INTEGER color_index : ranges from 0 through 7 for 8 colors.

INTEGER num_spheres : number of spheres.

ARRAY REAL radius(num_spheres) : array of sphere radii in 3D Model Coordinates.

ARRAY REAL x_coord(num_spheres) : array of x-axis coordinates of sphere centres.

ARRAY REAL y_coord(num_spheres) : array of y-axis coordinates of sphere centres.

ARRAY REAL z_coord(num_spheres) : array of z-axis coordinates of sphere centres.

This draws a list of identically colored solid spheres. Its actual implementation is through so-called *annotation* circles. Annotation circles are circles that always maintain the same orientation in relation to the plane of the screen, given a centre point and a radius.

gal_3d_unfilled_multisphere(color_index, num_spheres, radius, x_coord, y_coord, z_coord)

INTEGER color_index : ranges from 0 through 7 for 8 colors.

INTEGER num_spheres : number of spheres.

ARRAY REAL radius(num_spheres) : array of sphere radii in 3D Model Coordinates.

ARRAY REAL x_coord(num_spheres) : array of x-axis coordinates of sphere centres.

ARRAY REAL y_coord(num_spheres) : array of y-axis coordinates of sphere centres.

ARRAY REAL z_coord(num_spheres) : array of z-axis coordinates of sphere centres.

This procedure draws a list of identically colored wireframe spheres, given arrays of their radii and respective x,y and z model coordinates.

gal_quadrilateral_mesh(color_index, row, column, num_pts, mesh)

INTEGER color_index : ranges from 0 through 7 for 8 colors

INTEGER row : number of vertex rows in the mesh

INTEGER column : number of vertex columns in the mesh

INTEGER num_pts : number of points in the mesh array

ARRAY REAL mesh($3 \times \text{num_pts}$) : an array mesh($3 \times \text{num_pts}$) of vertex coordinates

The mesh list contains $i = \text{num_pts}$ sets : ($\{x_1, y_1, z_1\}, \{x_2, y_2, z_2\}, \dots, \{x_i, y_i, z_i\}$) entered as a 1 dimensional array (a vector). The sets represent (x,y,z) coordinate of each vertex, resulting in $3 \times \text{num_pts}$ elements. A quad mesh consists of a $m \times n$ matrix of vertices (column = m, row = n):

$$\begin{array}{cccc}
 \{x_1, y_1, z_1\} & \{x_2, y_2, z_2\} & \dots & \{x_m, y_m, z_m\} \\
 \{x_{m+1}, y_{m+1}, z_{m+1}\} & \{x_{m+2}, y_{m+2}, z_{m+2}\} & \dots & \{x_{2m}, y_{2m}, z_{2m}\} \\
 \cdot & \cdot & \cdot & \cdot \\
 \cdot & \cdot & \cdot & \cdot \\
 \cdot & \cdot & \cdot & \cdot \\
 \{x_{(n-1)m+1}, y_{(n-1)m+1}, z_{(n-1)m+1}\} & \{x_{(n-1)m+2}, y_{(n-1)m+2}, z_{(n-1)m+2}\} & \dots & \{x_{nm}, y_{nm}, z_{nm}\}
 \end{array}$$

The mesh above defines $(m-1) \times (n-1)$ quadrilaterals (4-sided polygons) formed from the adjacent vertices:

$$\begin{array}{cc}
 \{x_{k+1}, y_{k+1}, z_{k+1}\} & \{x_{k+2}, y_{k+2}, z_{k+2}\} \\
 \{x_{m+k+1}, y_{m+k+1}, z_{m+k+1}\} & \{x_{m+k+2}, y_{m+k+2}, z_{m+k+2}\}
 \end{array}$$

where $k \leq (n-1)m$ (m and n are the number of columns and rows, respectively).

gal_quadrilateral_surface(color_index, row, column, num_pts, mesh, color_array)

INTEGER color_index : ranges from 0 through 7 for 8 colors

INTEGER row : number of vertex rows in the mesh

INTEGER column : number of vertex columns in the mesh

INTEGER num_pts : number of points in the mesh array

ARRAY REAL mesh($3 \times \text{num_pts}$) : an array mesh($3 \times \text{num_pts}$) of vertex coordinates

ARRAY INTEGER color_array((column-1)×(row-1)) : an array color_array((column-1) × (row-1)) of integers assigning the colors of each facet of the surface.

The color_array array contains the colors (0 through 7) for the (m-1) × (n-1) facets for a m × n grid. (Refer to gal_quadrilateral_mesh() for details on the quadrilateral configuration. The integer color_array(k) maps the color of the quadrilateral specified by the coordinates

$$\begin{array}{cc} \{x_k, y_k, z_k\} & \{x_{k+1}, y_{k+1}, z_{k+1}\} \\ \{x_{m+k}, y_{m+k}, z_{m+k}\} & \{x_{m+k+1}, y_{m+k+1}, z_{m+k+1}\} \end{array}$$

where k is some integer and $1 \leq k < (m-1) \times (n-1)$ (m is the column length). The color assigned to color_array(k) describes the color of the ith facet in row j, where $i = (k \bmod (j \times m)) + 1$. Note that gal_quadrilateral_surface() renders a solid object and should be used in conjunction with the z-buffer.

GAL Bugs

Color choices - GAL should have a 16 color (4-bit) palette. Instead, only 8 colors (3 bits) are available. Utilizing the entire 8 bits for double-buffering results in mysterious color mapping problems.

“multi” drawing primitives - Primitives that draw arrays of geometric objects (e.g. `gal_3d_multisphere()`, `gal_2d_multicircle()`) appear to have upper limits for the number of objects that can be drawn. For example, the author’s experience on the author’s machine (a SPARCstation 1+ with GX accelerator) suggests that `gal_3d_multisphere()` behaves erratically when set up to draw arrays of sphere coordinates larger than 64.

Mixing of z-buffered solid and wireframe objects - Caution is advised here - for some yet-to-be determined reason, wireframe objects (e.g. polygons, spheres) are not to the screen when z-buffering is carried out. It is advised that if you are drawing solid spheres and filled polygons, and require unfilled polygons, draw the unfilled polygons as a line (using `gal_3d_solid_line()` as opposed to `gal_3d_filled_polygon()`). Another option would be to do a depth sort your primitives (draw the most distant objects first).

This list is by no means comprehensive or complete. If you find any more bugs or have any inquiries or suggestions, for that matter, contact me by email at napier@maccs.dcss.mcmaster.ca.

Index of GAL Primitives

The Listing of GAL primitives in the manual follows a logical (in this user's opinion, at least!) ordering that was based on similarities in functionality. The following is an alphabetical listing of the primitives.

gal_2d_asterisk_marker	76	gal_2d_unfilled_polygon	75
gal_2d_axes	78	gal_2d_unfilled_rectangle	74
gal_2d_circle_marker	76	gal_3d_asterisk_marker	85
gal_2d_cross_marker	76	gal_3d_axes	87
gal_2d_dash_dot_dot_line	75	gal_3d_circle_marker	85
gal_2d_dash_dot_line	75	gal_3d_cross_marker	85
gal_2d_dash_dotted_line	75	gal_3d_dash_dot_dot_line	84
gal_2d_dotted_line	75	gal_3d_dash_dot_line	84
gal_2d_filled_circle	64	gal_3d_dash_dotted_line	84
gal_2d_filled_multicircle	79	gal_3d_filled_multisphere	88
gal_2d_filled_polygon	75	gal_3d_filled_polygon	83
gal_2d_filled_rectangle	74	gal_3d_filled_sphere	83
gal_2d_long_line	75	gal_3d_long_line	84
gal_2d_plus_marker	76	gal_3d_pan_and_zoom	72
gal_2d_real_annotate	78	gal_3d_plus_marker	85
gal_2d_solid_line	75	gal_3d_real_annotate	86
gal_2d_square_marker	76	gal_3d_solid_line	84
gal_2d_text_annotate	77	gal_3d_square_marker	85
gal_2d_unfilled_circle	74	gal_3d_text_annotate	85
gal_2d_unfilled_multicircle	79	gal_3d_unfilled_multisphere	88

gal_3d_unfilled_polygon	83
gal_3d_unfilled_sphere	83
gal_animate_button	70
gal_dotted_contour	82
gal_init_color	68
gal_init_window	65
gal_new_2d_frame	69
gal_new_3d_frame	69
gal_paint_proc	65
gal_panel_button	65
gal_panel_slider#	66
gal_quadrilateral_mesh	88
gal_solid_mesh	89
gal_quit_proc	67
gal_reset_color	70
gal_reset_zbuffer	69
gal_rotate	72
gal_set_2d_buffer	69
gal_set_3d_buffer	69
gal_set_zbuffer	69
gal_solid_contour	71
gal_start	66
gal_step_anim_button	71
gal_switch_buffer	69
get_slider#_value	67

Appendix 2 - GAL Code Listing

Appendix 2a - GAL Header Files

```

/*****
 *
 * xv.h - contains the headings for structures used in GAL and
 * definition of its subroutines
 *
 *****/

#include <stdio.h>
#include <string.h>

#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/canvas.h>
#include <xview/panel.h>
#include <xview/xv_xrect.h>
#include <xview/server.h>
#include <X11/Xlib.h>
#include <X11/Xatom.h>
#include <X11/Xutil.h>

/* The XGL header file */
#include </home/epheusus/duncan/xgl-2.0/include/xgl/xgl.h>

/* error banner */
#define GAL_ERROR "***** GAL Error *****"
#define TRAIL "*****\n"
#define MAX_COLORS 8

/* GAL primitive and associated library functions */
void gal_init_window();
void gal_end();
void gal_start();
void gal_panel_button();
void gal_init_color();
void gal_animate_button();
void gal_step_anim_button();
void step_proc1();
void gal_quit_proc();
void start_stop_proc();
void get_anim_proc();
void gal_2d_unfilled_circle();
void gal_2d_filled_circle();
void gal_2d_unfilled_multicircle();
void gal_2d_filled_multicircle();
void gal_2d_unfilled_rectangle();
void gal_2d_filled_rectangle();
void gal_2d_unfilled_multirectangle();
void gal_2d_filled_multirectangle();
void gal_3d_unfilled_sphere();
void gal_3d_filled_sphere();
void gal_2d_filled_polygon();
void gal_3d_filled_polygon();
void gal_2d_unfilled_polygon();
void gal_3d_unfilled_polygon();
void gal_quadrilateral_mesh();
void gal_quadrilateral_surface();
void gal_2d_solid_line();
void gal_2d_dotted_line();
void gal_2d_dashed_line();
void gal_2d_dash_dotted_line();
void gal_2d_dash_dot_line();
void gal_2d_dash_dot_dot_line();
void gal_2d_long_dash_line();
void gal_3d_solid_line();
void gal_3d_dotted_line();

```



```

void    gal_3d_dashed_line_();
void    gal_3d_dash_dotted_line_();
void    gal_3d_dash_dot_line_();
void    gal_3d_dash_dot_dot_line_();
void    gal_3d_long_dash_line_();
void    gal_2d_cross_marker_();
void    gal_2d_plus_marker_();
void    gal_2d_asterisk_marker_();
void    gal_2d_square_marker_();
void    gal_2d_circle_marker_();
void    gal_3d_cross_marker_();
void    gal_3d_plus_marker_();
void    gal_3d_asterisk_marker_();
void    gal_3d_square_marker_();
void    gal_3d_circle_marker_();
void    gal_2d_axes_();
void    gal_3d_axes_();
void    gal_new_2d_frame_();
void    gal_new_3d_frame_();
void    gal_switch_buffer_();
void    gal_set_3d_buffer_();
void    gal_set_2d_buffer_();
int     gal_dbuf_color_map();
void    gal_3d_aspect_set_();
void    gal_2d_aspect_set_();
void    gal_3d_aspect_free_();
void    gal_2d_aspect_free_();
void    gal_reset_color();
void    gal_2d_text_annotate_();
void    gal_3d_text_annotate_();
void    gal_2d_real_annotate_();
void    gal_3d_real_annotate_();
void    gal_rotate_();
static void    xv_wm_install();
char    *nullcpy();
int     xlib_checkvisual();
Xgl_win_ras    xglut_create_window_raster_from_xv_canvas();
void    gal_panel_slider1_();
void    gal_panel_slider2_();
void    gal_panel_slider3_();
void    gal_panel_slider4_();
void    gal_panel_slider5_();
void    gal_panel_slider6_();
void    slider1_notify_proc();
void    slider2_notify_proc();
void    slider3_notify_proc();
void    slider4_notify_proc();
void    slider5_notify_proc();
void    slider6_notify_proc();
int     get_slider1_value_();
int     get_slider2_value_();
int     get_slider3_value_();
int     get_slider4_value_();
int     get_slider5_value_();
int     get_slider6_value_();
void    (*get_slider1_proc())();
void    (*get_slider2_proc())();
void    (*get_slider3_proc())();
void    (*get_slider4_proc())();
void    (*get_slider5_proc())();
void    (*get_slider6_proc())();
void    gal_3d_pan_and_zoom();
void    gal_2d_pan_and_zoom();

```

/* Global object and flag access functions */

```

Frame *get_frame();           /* access functions for GAL's objects */
Panel *get_panel();
Canvas *get_canvas();
Display *get_display();
Xgl_win_ras *get_ras();       /* raster object */
Xgl_2d_ctx *get_2d_ctx();     /* 3d context object */
Xgl_3d_ctx *get_3d_ctx();     /* 2d context object */
Xgl_gcach *get_gcach();       /* text context object */
Xgl_sys_st *get_sys_state();  /* system state used by xgl_object_create */
Xgl_color *get_color_table(); /* retrieves the color table */
Xgl_color_list *get_cmap_info(); /* retrieves color map information
structure */
Xgl_color_rgb *get_rgb();
Xgl_usgn32 *get_dbuf_on();    /* retrieves the value of dbuffer flag */
int *get_dbuf_alloc();       /* returns size of dbuffer */
short *get_gal_start();      /* returns flag to check gal_start */
short *get_aspect_flag();    /* returns flag to check the view volume */

/* Double buffering structures */

typedef struct {
    /* USER SUPPLIED FIELDS */
    Xgl_ctx ctx;
    Xgl_sgn32 number_of_colors_per_buffer;
    Xgl_color *color_table;
    /* COMPUTED FIELDS */
    Xgl_boolean cmap_dbuffering;
    Xgl_boolean xgl_dbuffering;
    Xgl_boolean current_buffer_is_buffer_0;
    Xgl_sgn32 bits_per_buffer;
    Xgl_sgn32 buf0_pm, buf1_pm;
    Xgl_cmap cmap0, cmap1;
    Xgl_sgn32 buffers_requested;
    Xgl_sgn32 buffers_allocated;
    Xgl_sgn32 buf_draw;
    Xgl_sgn32 buf_display;
    Xgl_sgn32 buf_min_delay;
} Xglut_dbuf_info;

Xglut_dbuf_info *get_dbuf_info();

/* XGL hardware inquiry variables. These tell the library what hardware
facilities exist */
Xgl_boolean xglut_hw_zbuffer;
Xgl_boolean xglut_hw_shading;
Xgl_color_type xglut_hw_color_type;

/* A macro to build a color map for the cmap_info structure */
#define XGLUT_SETCMAP_INFO(arg, arg_start, arg_length, arg_info) do { \
    (arg).start_index = (arg_start); \
    (arg).length = (arg_length); \
    (arg).colors = (arg_info); \
} while (0)

/* some axis defaults for the axis - drawing primitives */
static float axis_thickness = 1.0;
static int axis_num_pts = 2;
static float zero = 0.0;

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

```

```

*
* color.h - contains values used in coloring primitives of GAL
*
*****/

/* color indices */
#define BACKGROUND_INDEX      0
#define WHITE_INDEX           1
#define RED_INDEX              2
#define GREEN_INDEX           3
#define BLUE_INDEX            4
#define YELLOW_INDEX          5
#define CYAN_INDEX            6
#define MAGENTA_INDEX         7
/* #define OLIVE_INDEX        8
#define PURPLE_INDEX          9
#define AQUA_INDEX           10
#define PINK_INDEX            11
#define LIME_INDEX            12
#define SKY_INDEX             13
#define DARK_GRAY_INDEX       14
#define LIGHT_GRAY_INDEX      15

*/

#define COLOR_SIZE 8

Xgl_color      black_color, white_color, red_color, green_color, blue_color;
Xgl_color      yellow_color, cyan_color, magenta_color, olive_color;
Xgl_color      purple_color, aqua_color, pink_color, lime_color, sky_color;
Xgl_color      dark_gray_color, light_gray_color;

/* double buffering stuff */
static Xgl_usgn32 toggle = 0;
static Xgl_usgn32 set_dbuf = 0;

/* Color */
static Xgl_color      color_table[COLOR_SIZE];

```

Appendix 2b - X Windows Utilities

```

/*****
 * GAL source code by Duncan Napier, Computer Science and Systems,
 * McMaster University, Hamilton, Ontario, Canada, October 1992.
 * The horizontal asterisks (/*****/) demarcate source files
 *****/

#include "../include/xv.h"

static short    set_start; /* A flag to ensure routine has been set */

/*****
 * gal_start_() - a routine to initialize all the objects of the gal
 * object set for error detection. This is the first statement of any
 * GAL routine. This routine is a syntactic device for the GAL
 * programmer. Initially, it was intended to be an initialization
 * routine for error detection. However, more efficient methods for
 * error detection were developed. After this, point, it was decided
 * that a bracketing statement would clarify GAL applications - and
 * this is now the only function of this procedure.
 *****/

void gal_start_() {
    extern short    set_start;

    set_start = 1;
}

/*****
 * get_gal_start - access function for the gal_start flag.
 * called by - gal_end()
 *****/

short get_gal_start() {
    extern short    set_start;

    return set_start;
}

/*****

#include "../include/xv.h"

static Frame frame;      /* XView's frame object */
static Panel panel;      /* XView's panel object */

/*****
 * gal_init_window() - Initializes the top-level Xwindow via Xview
 * toolkit. The set of possible child widgets is initialized. It also
 * creates a panel on the window. parameters passed - Xwindow width,
 * height, label, n - number of characters (automatic from FORTRAN).
 * functions called - xv_init, xv_create.
 *****/

void gal_init_window_(width, height, label, n)
int    *width, *height;
char    *label;
int    n;
{
    char    *frame_label;
    extern Frame frame;
    extern Panel panel;

    frame_label = malloc(n);

```

```

    if (!frame_label) {
        printf("\n\n\n%s\n",GAL_ERROR);
        printf("BAD PARAMETER IN
                GAL_INIT_WINDOW\n");
        printf("\n\n\n%s\n",TRAIL);
        exit(1);
    }

    (void) memcpy(argv[0], label, n);
    (void) memcpy(frame_label, label,n);

    /* pass the label to the XView frame, along with window dimensions */

    frame = (Frame) xv_create (NULL, FRAME,
        FRAME_LABEL, frame_label, XV_WIDTH,
        *width, XV_HEIGHT, *height, NULL);
    free(frame_label);

    /* create a panel */
    if (!panel) {
        panel = (Panel) xv_create (frame, PANEL, NULL);
    } else
    {
        printf("\n\n\n%s\n",GAL_ERROR);
        printf("CANNOT CREATE SECOND PANEL, ONE ALREADY EXISTS !
        \n\n");
        printf("\n\n\n%s\n",TRAIL);
        exit (1);
    }
}

/*****
 * get_frame() - An access function that passes the frame widget's
 * address to external routines. '
 *****/

Frame *get_frame() {

    return &frame;

}

/*****
 * get_panel() - function to access the XView panel widget
 *****/

Panel *get_panel() {

    return &panel;

}

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

#include "../include/xv.h"

static Display *display; /* XGL display object */
static Canvas canvas; /* XView's canvas object */

/*****
 * gal_paint_proc() - this procedure produces the XView canvas for XGL
 * rendering.
 * subroutines called - get_panel, get_frame, xv_get,

```

```

* xlib_checkvisual, window_fit_height xv_wm_install.
*****/

void gal_paint_proc(draw_proc)
void (*draw_proc)();
{
    int            win_visual_class;
    Xgl_sgn32      ex_win_depth;
    Frame          *global_frame;
    Panel          *panel;

    panel = get_panel();

    global_frame = get_frame();

    if (!*global_frame) {
        printf("\n \n \n %s \n", GAL_ERROR) ;
        printf("ERROR - UNABLE TO PAINT IN GAL_PAINT_PROC, CHECK THAT
                GAL_INIT_WINDOW IS SET FIRST. \n");
        printf("\n \n \n %s
                \n", TRAIL) ;
        exit (1);
    }

    display = (Display *) xv_get (*global_frame, XV_DISPLAY);

    /* determine color class of the display (rgb or indexed) */

    if (xlib_checkvisual(display, DefaultScreen(display), 24)) {
        ex_win_depth = 24;
        win_visual_class = TrueColor;
    } else {
        ex_win_depth = 16;
        win_visual_class = PseudoColor;
    }

    /* check for the existence of a pane then create the XView canvass */

    if (!*panel) {
        canvas = (Canvas) xv_create (*global_frame,
            CANVAS,
            XV_X,                0,
            CANVAS_AUTO_CLEAR,    FALSE,
            CANVAS_RETAINED,      FALSE,
            CANVAS_FIXED_IMAGE,   FALSE,
            CANVAS_REPAINT_PROC,  draw_proc,
            WIN_DEPTH,            ex_win_depth,
            XV_VISUAL_CLASS,
            win_visual_class, NULL);
    } else {
        window_fit_height (*panel);

        canvas = (Canvas) xv_create (*global_frame, CANVAS,
            XV_X,                0,
            WIN_BELOW,           *panel,
            CANVAS_AUTO_CLEAR,    FALSE,
            CANVAS_RETAINED,      FALSE,
            CANVAS_FIXED_IMAGE,   FALSE,
            CANVAS_REPAINT_PROC,  draw_proc,
            WIN_DEPTH,            ex_win_depth,
            XV_VISUAL_CLASS,
            win_visual_class, NULL);
    }
}

```

```

        /* install cursor tracking across the canvas */
        xv_wm_install();
    }

    /*****
     * get_canvas() - an access routine for the canvas object
     *****/

    Canvas *get_canvas() {
        return &canvas;
    } /* get_canvas */

    /*****
     * xv_wm_install() given a xview frame and canvas; tell the
     * server/window manager to track the cursor and color map events.
     * procedures called - xv_get, canvas_paint_window, XInternAtom,
     * XChangeProperty.
     *****/
    static void xv_wm_install() {
        Atom    catom;
        Window  frame_window, canvas_window;
        Display *display;
        Frame   *global_frame;
        Canvas  *global_canvas;

        global_frame = get_frame();
        global_canvas = get_canvas();
        display       = (Display *)xv_get(*global_frame, XV_DISPLAY);
        canvas_window = (Window)xv_get(
            (Xv_Window)canvas_paint_window(*global_canvas),
            XV_XID);

        frame_window = (Window)xv_get(*global_frame, XV_XID);
        catom        =
            XInternAtom(display, "WM_COLORMAP_WINDOWS", False);
        XChangeProperty(display, frame_window, catom, XA_WINDOW,
            32, PropModeAppend, &canvas_window, 1);

        return;
    } /* xv_wm_install */

    /*****
     * xlib_checkvisual() - get the visual type which matches the depth
     * argument routines called - XGetVisualInfo.
     *****/
    int
    xlib_checkvisual(display, screen, depth)
    Display *display;
    int     screen;
    int     depth;
    {
        XVisualInfo template;
        XVisualInfo *visuals, *v;
        int nvisuals, i;

        template.screen = screen;
        template.depth = depth;

        visuals = XGetVisualInfo(display, VisualScreenMask |

```



```

        VisualDepthMask, &template, &nvisuals);

    for (v = visuals, i = 0; i < nvisuals; v++, i++)
        if (v->depth == depth) {
            return(1);
            break;
        }

    return(0);
} /* xlib_checkvisual */

/*****
 * get_display() - access routines for the display object.
 *****/

Display *get_display() {
    extern Display *display;

    return display;
} /* get_display */

/*****/

#include "../include/xv.h"

/*****
 * gal_end() - enters the event loop for XWindows.
 * procedures called - get_frame, xv_main_loop
 * This is the last statement of a graphical command
 *****/

void gal_end() {
    Frame *global_frame;

    global_frame = get_frame();

    /* if there is no gal_start, warn */
    if (!get_gal_start())
    {
        printf("\n\n\n %s \n", GAL_ERROR) ;
        printf("WARNING -\n\n\n");
        printf("\n\n\n %s \n", TRAIL) ;
    }

    /* if the viewport dimensions are not set, abort */
    if (!get_aspect_flag())
    {
        if (*get_canvas())
        {
            printf("\n\n\n %s \n", GAL_ERROR) ;
            printf("ERROR - THE DIMENSIONS OF THE VIEWSPACE\n\n\n");
            printf("\n\n\n %s\n", TRAIL) ;
            exit (1);
        }
    }

    /* register callbacks if a frame exists */
    if (!*global_frame)
    {
        printf("\n\n\n %s \n", GAL_ERROR) ;
        printf("CANNOT CREATE PROGRAM WITHOUT WINDOW! \n\n\n");
        printf("\n\n\n %s \n", TRAIL);
    }
}

```

```

        exit (1);
    }
    /* start the event-driven program */
    else xv_main_loop(*global_frame);

    exit(0);
}

/*****

#include "../include/xv.h"

/*****
 * gal_panel_button_() - a puts a button on the panel and registers the
 * callback routines called - get_panel, xv_create
 *****/

void gal_panel_button_(process,x_pos, y_pos, label,n)
int      *x_pos,*y_pos;    /* the panel coordinates */

int      n;
char     *label;
void     (*process)();    /* the callaback procedure */

{
    char     *string = "";
    Panel    *panel;

    panel = get_panel();

    if (!(*panel)) {
        printf("\n \n \n %s \n",GAL_ERROR) ;
        printf("ATTEMPT TO CREATE BUTTON BEFORE WINDOW! \n \n");
        printf("\n \n \n %s \n",TRAIL) ;
        exit (1);
    }
    else {
        /* null-terminate the string */
        (void) strcpy(string, label, n);
        (void) xv_create (*panel, PANEL_BUTTON,
            PANEL_ITEM_X, xv_col (*panel, *x_pos),
            PANEL_ITEM_Y, xv_row (*panel, *y_pos),
            PANEL_LABEL_STRING, string, PANEL_NOTIFY_PROC,
            process, NULL);
    }
}

/*****

#include "../include/xv.h"
#include "../include/color.h"

static int  slider1_value;    /* global slider value */
static void (*slider1_proc)(); /* global pointer to slider function */

void (*get_slider1_proc)(); /* access function for slider callback */

/*****
 * This file contains routines required to implement XView sliders via
 * gal calls. The slider implementation consists of 4 parts: '
 *
 * 1) gal_panel_slider#() - sets up the panel slider and generates the

```

```

* corresponding global routine (passed from a FORTRAN call)
* 2)slider#_notify_proc() - the corresponding notifier procedure
*   This procedure obtains the value generated at the slider it also
*   calls the subroutine (user_supplied) passed to its parent.
* 3)get_slider#_value_() - Access function for the slider value.
* 4)(*get_slider1_proc())() - Access function that returns the
*   procedure passed to gal_panel_slider#().
*
* These calls collectively set up the slider, access the slider value,
* and then calls a function of the user's choice
*
* Note there are 6 separate slider procedures, only one is shown here for
* the sake of brevity ...
*****/

void gal_panel_slider1(slider_proc_dummy, slider_label, max_value,
min_value, slider_pos, str_len)
void (*slider_proc_dummy)(); /*a dummy variable pointer to FORTRAN
function */
char *slider_label;          /* the label for the slider */
int *max_value, *min_value, *slider_pos, str_len; /* slider limits
*/

{
    extern void (*slider1_proc)();
    Panel *panel;
    char *slider_label_copy;

    panel = get_panel();

    if (!(*panel))
    {
        printf("\n\n\n %s\n", GAL_ERROR) ;
        printf("ATTEMPT TO CREATE SLIDER1 BEFORE PANEL.
        CHECK GAL_INIT_WINDOW. \n\n");
        printf("\n\n\n %s\n", TRAIL) ;
        exit (1);
    }
    else
    {
        slider1_proc = slider_proc_dummy;

        slider_label_copy = malloc(80);

        if (!slider_label_copy) {
            printf("memory allocation request in gal_panel_slider1
            failed\n");
            exit(1);
        }

        (void) memcpy(slider_label_copy, slider_label, str_len);

        panel_create_item(*panel, PANEL_SLIDER, PANEL_ITEM_Y,
ATTR_ROW(*slider_pos), PANEL_ITEM_X, ATTR_COL(0),
PANEL_NOTIFY_PROC, slider1_notify_proc,
PANEL_LABEL_STRING, slider_label_copy ,
PANEL_VALUE, 0, PANEL_MIN_VALUE, *min_value,
PANEL_MAX_VALUE, *max_value, PANEL_SLIDER_WIDTH,
256, 0);

        free(slider_label_copy);
    }
}

```

```

/*****
 * slider1_notify_proc - XView slider notifier process
 *****/

void slider1_notify_proc(item, value, event)
Panel_item item;
int value;          /* slider value */

Event    event;
{
    extern int slider1_value;
    Xgl_3d_ctx *ctx3d;

    slider1_value = value;
    ctx3d = get_3d_ctx();
    xgl_context_new_frame(*ctx3d);

    /* call FORTRAN program corresponding to slider1 */

    (*get_slider1_proc())();
}

/*****
 * get_slider1_value_() - access function for slider value
 *****/

int get_slider1_value_(){
    extern int slider1_value;

    return(slider1_value);
}

/*****
 * get_slider1_proc() - access function for slider function
 *****/

void (*get_slider1_proc())() {
    extern void (*slider1_proc)();

    if (slider1_proc == NULL) {
        printf("Incorrect/inaccessible slider1 procedure!!!\n");
        exit(0);
    } else return(slider1_proc);
}

/*****/

#include "../include/xv.h"

/*****
 *
 * gal_new_(2d/3d)_frame_() - these commands clear the graphical
 * contexts
 *
 * procedures called - xgl_object_set, xgl_context_new_frame
 *
 *****/
void gal_new_2d_frame_() {
    Xgl_2d_ctx *ctx;

    ctx = get_2d_ctx();

    xgl_object_set(*ctx, XGL_CTX_NEW_FRAME_ACTION,
        XGL_CTX_NEW_FRAME_VRETRACE |

```

```

        XGL_CTX_NEW_FRAME_CLEAR, NULL);

    xgl_context_new_frame(*ctx);
}

/*****
 * gal_new_(2d/3d)_frame_() - these commands clear the graphical
 * contexts
 *
 * procedures called - xgl_object_set, xgl_context_new_frame
 *****/
void gal_new_3d_frame_() {
    Xgl_3d_ctx *ctx;

    ctx = get_3d_ctx();

    xgl_object_set(*ctx, XGL_CTX_NEW_FRAME_ACTION,
        XGL_CTX_NEW_FRAME_VRETRACE |
        XGL_CTX_NEW_FRAME_CLEAR, NULL);

    xgl_context_new_frame(*ctx);
}

/*****

#include "../include/xv.h"

/*****
 * gal_set_zbuffer_() - sets up hidden line/hidden surface removal in
 * XGL
 *****/
gal_set_zbuffer_() {
    Xgl_3d_ctx *ctx_3d;

    ctx_3d = get_3d_ctx();

    if (!*ctx_3d) {
        printf("\n\n\n %s\n", GAL_ERROR) ;
        printf("ERROR - UNABLE TO SET ZBUFFER IN GAL_SET_ZBUFFER,
            CHECK THAT GAL_INIT_WINDOW IS SET\n");
        printf("\n\n\n %s\n", TRAIL) ;
        exit (1);
    }

    xgl_object_set(*ctx_3d, XGL_3D_CTX_HLHSR_MODE,
        XGL_HLHSR_ZBUFFER, NULL);
}

/*****

#include "../include/xv.h"

/*****
 * gal_set_zbuffer_() - sets up hidden line/hidden surface removal in
 * XGL
 *****/
gal_set_zbuffer_() {
    Xgl_3d_ctx *ctx_3d;

    ctx_3d = get_3d_ctx();

```

```

        if (!*ctx_3d) {
            printf("\n \n \n %s \n",GAL_ERROR) ;
            printf("ERROR - UNABLE TO SET ZBUFFER IN GAL_SET_ZBUFFER,
                CHECK THAT GAL_INIT_WINDOW IS SET \n");
            printf("\n \n \n %s\n",TRAIL) ;
            exit (1);
        }

        xgl_object_set(*ctx_3d, XGL_3D_CTX_HLHSR_MODE,XGL_HLHSR_ZBUFFER, NULL);
    }

/*****

#include "../include/xv.h"

/*****
 * nullcpy() - a function that returns a null-terminated 'd'
 * (destination) copy of the source 's'. Note that both strings
 * must be initialized * before being passed. Supplied by Patricia
 * Monger.
 * procedures called - strncpy
 *****/

char *nullcpy(d,s,icnt)
char *d, /* null terminated string (destination) */
*s; /* non-null terminated string (source) */
{
    if(icnt > 0)
    {
        while(s[icnt - 1] == ' ') icnt--;
        (void)strncpy(d,s,icnt);
    }
    d[icnt] = '\0';
    return(d);
}

```

Appendix 2c - Color and Animation

```

/*****
 * These routines control the color settings and double_buffering in GAL.
 * Duncan Napier, McMaster University, Hamilton, Ontario, Canada.
 * October 1992.
 *****/

#include "../include/xv.h"
#include "../include/color.h"

static Xgl_win_ras ras; /* XGL raster object */
static Xgl_2d_ctx ctx_2d; /* XGL 2d context object */

static Xgl_3d_ctx ctx_3d; /* XGL 3d context object */

static Xgl_gcach gcach; /* XGL's gcach object for annotation
primitives */

static Xgl_sys_st sys_st; /* system state used by xgl_object_create */
static Xglut_dbuf_info dbuf_info; /* XGL double buffering information */
static Xgl_color_list cmap_info; /* XGL color map structure */

Xgl_color black_color, white_color, red_color, green_color,
blue_color, yellow_color, cyan_color, magenta_color,
olive_color, purple_color, aqua_color, pink_color,
lime_color, sky_color, dark_gray_color, light_gray_color;
static Xgl_color_rgb default_rgb[COLOR_SIZE]={
    { 0.2, 0.2, 0.2 }, { 1.0, 1.0, 1.0 },
    { 1.0, 0.0, 0.0 }, { 0.0, 1.0, 0.0 },
    { 0.0, 0.0, 1.0 }, { 1.0, 1.0, 0.0 },
    { 0.0, 1.0, 1.0 }, { 1.0, 0.0, 1.0 },
    { 0.5, 0.5, 0.0 }, { 0.5, 0.0, 0.5 },
    { 0.0, 0.5, 0.5 }, { 1.0, 0.5, 0.5 },
    { 0.5, 1.0, 0.5 }, { 0.5, 0.5, 1.0 },
    { 0.5, 0.5, .5 }, { 0.7, 0.7, 0.7 } };

/*****
 *
 * gal_init_color_() - Initializes the raster object and color tables for
 * XGL rendering as well as the graphical contexts. This is taken mainly
 * the demos that accompanied XGL.
 *
 * procedures called - canvas_paint_window, xv_get, DefaultScreen,
 * xgl_inquire, xgl_object_create
 *****/

void gal_init_color_() {
    Display *display;
    Window frame_window;
    Window canvas_window;
    Xv_window pw;
    Xgl_X_window xgl_x_win;
    Xgl_color_type ex_color_type;
    Xgl_cmap cmap;
    Xgl_obj_desc obj_desc;
    Xgl_inquire *inq_info;
    Frame *global_frame; /* the global frame object */
    Canvas *global_canvas; /* the global canvas object */
    extern Xgl_color color_table[COLOR_SIZE]; /* the global color table */
    Xgl_color_list *cmap_info; /* a global structure that contains
    colormap information */
    Xglut_dbuf_info dbuf_info; /* global d buffer info structure */

    /* the folowing acces functions return the addresses of their
    respective structures */
    global_frame = get_frame();
    global_canvas = get_canvas();

```



```

display = get_display();
cmap_info = get_cmap_info();

/* Check to make sure the frame has already been created */
if (!*global_frame) {
    printf("\n\n\n%s\n", GAL_ERROR);
    printf("ERROR IN GAL_INIT_COLOR. MAKE SURE GAL_INIT_WINDOW
HAS BEEN CALLED FIRST. \n\n");
    printf("\n\n\n%s\n", TRAIL);
    exit (1);
}

/* Check to make sure the canvas has already been created */
if (!*global_canvas) {
    printf("\n\n\n%s\n", GAL_ERROR);
    printf("ERROR IN GAL_INIT_COLOR. MAKE SURE GAL_PAINT_PROC
HAS BEEN CALLED FIRST. \n\n");
    printf("\n\n\n%s\n", TRAIL);
    exit (1);
}

display = (Display *) xv_get (*global_frame, XV_DISPLAY);

pw = (Xv_Window) canvas_paint_window (*global_canvas);
canvas_window = (Window) xv_get (pw, XV_XID);

frame_window = (Window) xv_get (*global_frame, XV_XID);

xgl_x_win.X_display = (void *) XV_DISPLAY_FROM_WINDOW (pw);
xgl_x_win.X_window = (Xgl_usgn32) canvas_window;
xgl_x_win.X_screen = (int) DefaultScreen (display);

/* create the system state */
sys_st = xgl_open (NULL);

obj_desc.win_ras.type = XGL_WIN_X;
obj_desc.win_ras.desc = &xgl_x_win;
if (!(inq_info = xgl_inquire(&obj_desc))) {
    printf(" \n\n\n%s\n", GAL_ERROR);
    printf("ERROR IN INQUIRY IN GAL_INIT_COLOR, CANNOT
DETERMINE COLOR TYPE OF DEVICE\n");
    printf(" \n\n\n%s\n", TRAIL);
    exit(1);
}

/* check the color type of the device */
if (inq_info->color_type.index)
    ex_color_type = XGL_COLOR_INDEX;
else if (inq_info->color_type.rgb)
    ex_color_type = XGL_COLOR_RGB;
else {
    printf(" \n\n\n%s\n", GAL_ERROR);
    printf("UNKNOWN COLOR TYPE\n");
    printf(" \n\n\n%s\n", TRAIL);
    exit(1);
}
free(inq_info);

/* if accelerated color type is indexed then create ex color map
* color maps are attached to the sys_st */

if (ex_color_type == XGL_COLOR_INDEX) {

```

```

/* initialize the color indices - the color indices are
   declared
       as static r,g,b values, and can be changed if desired
       by a routine gal_reset_color() */

color_table[BACKGROUND_INDEX].rgb = default_rgb[0];
color_table[WHITE_INDEX].rgb = default_rgb[1];
color_table[RED_INDEX].rgb = default_rgb[2];
color_table[GREEN_INDEX].rgb = default_rgb[3];
color_table[BLUE_INDEX].rgb = default_rgb[4];
color_table[YELLOW_INDEX].rgb = default_rgb[5];
color_table[CYAN_INDEX].rgb = default_rgb[6];
color_table[MAGENTA_INDEX].rgb = default_rgb[7];
color_table[OLIVE_INDEX].rgb = default_rgb[8];
color_table[PURPLE_INDEX].rgb = default_rgb[9];
color_table[AQUA_INDEX].rgb = default_rgb[10];
color_table[PINK_INDEX].rgb = default_rgb[11];
color_table[LIME_INDEX].rgb = default_rgb[12];
color_table[SKY_INDEX].rgb = default_rgb[13];
color_table[DARK_GRAY_INDEX].rgb = default_rgb[14];
color_table[LIGHT_GRAY_INDEX].rgb = default_rgb[15];

/* initialize the color table */
cmap_info->start_index = 0;
cmap_info->length = COLOR_SIZE;
cmap_info->colors = color_table;
cmap = xgl_object_create
    (sys_st, XGL_CMAP, 0,
     XGL_CMAP_COLOR_TABLE_SIZE,
     COLOR_SIZE,
     XGL_CMAP_COLOR_TABLE,
     cmap_info, NULL);
}

/* this beast was picked from a demo, it appears to be equivalent
   to the above one (and works too), but does some hardware
   checks first */

ras = xglut_create_window_raster_from_xv_canvas(sys_st,
*global_canvas, ex_color_type);

if (ex_color_type == XGL_COLOR_INDEX) {
    /* setup index color values in global color structures */
    xgl_object_set(ras, XGL_RAS_COLOR_MAP, cmap, NULL);
    black_color.index = BACKGROUND_INDEX;
    white_color.index = WHITE_INDEX;
    red_color.index = RED_INDEX;
    green_color.index = GREEN_INDEX;
    blue_color.index = BLUE_INDEX;
    yellow_color.index = YELLOW_INDEX;
    cyan_color.index = CYAN_INDEX;
    magenta_color.index = MAGENTA_INDEX;
    olive_color.index = OLIVE_INDEX;
    purple_color.index = PURPLE_INDEX;
    aqua_color.index = AQUA_INDEX;
    pink_color.index = PINK_INDEX;
    lime_color.index = LIME_INDEX;
    sky_color.index = SKY_INDEX;
    dark_gray_color.index = DARK_GRAY_INDEX;
    light_gray_color.index = LIGHT_GRAY_INDEX;
} else {
    /* setup rgb color values in global color structures */
    black_color.rgb = default_rgb[0];

```

```

        white_color.rgb = default_rgb[1];
        red_color.rgb = default_rgb[2];
        green_color.rgb = default_rgb[3];
        blue_color.rgb = default_rgb[4];
        yellow_color.rgb = default_rgb[5];
        cyan_color.rgb = default_rgb[6];
        magenta_color.rgb = default_rgb[7];
/* olive_color.rgb = default_rgb[8];
        purple_color.rgb = default_rgb[9];
        aqua_color.rgb = default_rgb[10];
        pink_color.rgb = default_rgb[11];
        lime_color.rgb = default_rgb[12];
        sky_color.rgb = default_rgb[13];
        dark_gray_color.rgb = default_rgb[14];
        light_gray_color.rgb = default_rgb[15];
*/
    }

/* create the 2d context object */

    ctx_2d = xgl_object_create(sys_st, XGL_2D_CTX, 0,
                              XGL_CTX_DEVICE, ras,
                              XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
                              XGL_CTX_BACKGROUND_COLOR, &black_color,
                              XGL_CTX_VDC_ORIENTATION,
                              XGL_Y_UP_Z_TOWARD, NULL);

    gcache = xgl_object_create(sys_st, XGL_GCACHE, NULL, NULL);

/* the 3d ctx has several default values set : the coordinate system
* is set with + y pointing up and z toward the user.
* The z-buffer is set to obscure hidden surfaces and lines.
* The z-buffering is used to control the appearance of the
* annotated solid circles. HLHSR_MODE requires a manual reset
* for transformations. */

    ctx_3d = xgl_object_create(sys_st, XGL_3D_CTX, 0,
                              XGL_CTX_DEVICE, ras,
                              XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
                              XGL_CTX_BACKGROUND_COLOR, &black_color,
                              XGL_CTX_VDC_ORIENTATION,
                              XGL_Y_UP_Z_TOWARD, NULL);
}

/*****
*
* gal_set_3d_buffer_ - this routine sets up 3d double buffering. the
* the color table field is set in dbuf_info here because of problems
* with the passing of pointer fields (i.e. *color_table) with access
* functions.
*
* functions called - get_3d_ctx, get_dbuf_info, get_cmap_info, get_color_table
* xglut_dbuf_init, xglut_dbuf_on.
*****/

void gal_set_3d_buffer_() {
    Xglut_dbuf_info *dbuf_info;
    Xgl_3d_ctx *ctx_3d;
    Xgl_color_list
        cmap_info;
    Xgl_win_ras wras;
    extern Xgl_usgn32 set_dbuf;

    /* Check to see the buffer is not already set */

```

```

if (!(set_dbuf)) {
    ctx_3d = get_3d_ctx();

    if (!*ctx_3d) {
        printf("\n \n \n %s \n", GAL_ERROR) ;
        printf("ERROR - UNABLE TO SET BUFFER IN
        GAL_SET_3D_BUFFER, CHECK THAT GAL_INIT_COLOR IS
        SET \n");
        printf("\n \n \n %s \n", TRAIL) ;
        exit
            (1);
    }

    dbuf_info = get_dbuf_info();
    cmap_info = *get_cmap_info();
    dbuf_info->number_of_colors_per_buffer = cmap_info.length;
    dbuf_info->color_table = get_color_table();
    dbuf_info->ctx = *ctx_3d;

    /* initialize the buffers for animation */
    xglut_dbuf_init(dbuf_info);
    xglut_dbuf_on(dbuf_info);

    /* set the color map flag if no hardware double buffering */
    if (get_dbuf_alloc) set_dbuf = 1;

} else {
    printf(" \n \n \n %s \n", GAL_ERROR);
    printf("DOUBLE BUFFER INITIALIZED TWICE CHECK FOR EXTRA
        GAL_SET_3D_BUFFER\n");
    printf(" \n \n \n %s \n", TRAIL);
    exit(1);
}

}

/*****
 * gal_set_2d_buffer_ - this routine sets up 2d double buffering. the
 * the color table field is set in dbuf_info here because of problems
 * with the passing of pointer fields (i.e. *color_table) with access
 * functions.
 *
 * functions called - get_2d_ctx, get_dbuf_info, get_cmap_info,
 * get_color_table, xglut_dbuf_init, xglut_dbuf_on.
 *****/

void gal_set_2d_buffer_() {
    Xglut_dbuf_info *dbuf_info;
    Xgl_2d_ctx *ctx_2d;
    Xgl_color_list
        cmap_info;
    extern Xgl_usgn32 set_dbuf;

    /* Check to see the buffer is not already set */
    if (!(set_dbuf)) {
        ctx_2d = get_2d_ctx();

        if (!*ctx_2d) {
            printf("\n \n \n %s \n", GAL_ERROR) ;
            printf("ERROR - UNABLE TO SET BUFFER IN
            GAL_SET_2D_BUFFER, CHECK THAT GAL_INIT_COLOR IS
            SET \n");
            printf("\n \n \n %s \n", TRAIL) ;
            exit
                (1);
        }
    }
}

```

```

        dbuf_info = get_dbuf_info();
        cmap_info = *get_cmap_info();
        dbuf_info->number_of_colors_per_buffer = cmap_info.length;
        dbuf_info->color_table = get_color_table();
        dbuf_info->ctx = *ctx_2d;

        /* initialize the buffers for animation */
        xglut_dbuf_init(dbuf_info);
        xglut_dbuf_on(dbuf_info);

        /* set the color map flag if no hardware double buffering */
        if (get_dbuf_alloc) set_dbuf = 1;

    } else {
        printf(" \n \n \n %s \n", GAL_ERROR);
        printf("DOUBLE BUFFER INITIALIZED TWICE CHECK FOR EXTRA
            GAL_SET_2D_BUFFER\n");
        printf(" \n \n \n %s \n", TRAIL);
        exit(1);
    }
}

/*****
 *
 * gal_switch_buffer_ - This procedure is the drives the switch_buffer
 * routine by passing it dbuf_inf
 *
 * functions called - get_dbuf_info,get_color_table
 *
 *****/

void gal_switch_buffer_() {
    Xglut_dbuf_info *dbuf_info;

    dbuf_info = get_dbuf_info();
    dbuf_info->color_table = get_color_table();
    switch_buffer(dbuf_info);
}

/*****
 * get_ras() - access function for the raster object
 *****/

Xgl_win_ras *get_ras() {
    extern Xgl_win_ras ras;

    return &ras;
}

/*****
 * get_2d_ctx() - access function for a 2d context
 *****/

Xgl_2d_ctx *get_2d_ctx() {
    extern Xgl_2d_ctx ctx_2d;

    return &ctx_2d;
}

```

```

/*****
 * get_3d_ctx() - access function for a 3d context
 *****/

Xgl_3d_ctx *get_3d_ctx() {
    extern Xgl_3d_ctx ctx_3d;

    return &ctx_3d;
}

/*****
 * get_sys_state() - access function for the system state object
 *****/

Xgl_sys_st *get_sys_state() {
    extern Xgl_sys_st sys_st;

    return &sys_st;
}

/*****
 * acces function for the dbuf_info record
 *****/

Xglut_dbuf_info *get_dbuf_info() {
    extern Xglut_dbuf_info dbuf_info;

    return(&dbuf_info);
}

/*****
 *
 * xglut_create_window_raster_from_xv_canvas - this code was lifted from a demo
 * do not know if it has any advantages, but it seems to carry out some
 * hardware checks
 *
 *****/

Xgl_win_ras xglut_create_window_raster_from_xv_canvas(sys_st,
xv_canvas, color_type)
Xgl_sys_st sys_st;
Canvas xv_canvas;
Xgl_color_type color_type;
{
    Xgl_X_window xgl_x;
    Xv_Window canvas_pw;
    Display *display;
    int screen;
    Window window;
    Xgl_win_ras ras;
    Xgl_obj_desc obj_desc;
    Xgl_inquire *inq_info;
    extern
    Xgl_color_type xglut_hw_color_type;
    extern Xgl_boolean
    xglut_hw_zbuffer;
    extern Xgl_boolean xglut_hw_shading;

```

```

    canvas_pw = (Xv_Window)canvas_paint_window(xv_canvas);

    display = (Display *)XV_DISPLAY_FROM_WINDOW(canvas_pw);
    screen = DefaultScreen(display);
    window = (Window)xv_get(canvas_pw, XV_XID);

    xgl_x.X_display = (void *)display;
    xgl_x.X_screen = screen;
    xgl_x.X_window = window;

    obj_desc.win_ras.type = XGL_WIN_X | XGL_WIN_X_PROTO_DEFAULT;
    obj_desc.win_ras.desc = &xgl_x;

    if (!(inq_info = xgl_inquire(&obj_desc))) {
        printf("error getting inquiry\n");
        exit(1);
    }

    if (inq_info->color_type.index)
        xglut_hw_color_type = XGL_COLOR_INDEX;
    else
        xglut_hw_color_type = XGL_COLOR_RGB;

    if (inq_info->hlhsr_mode == XGL_HLSR_ZBUFFER)
        xglut_hw_zbuffer = TRUE;
    else
        xglut_hw_zbuffer = FALSE;

    if (inq_info->shading == XGL_INQ_SOFTWARE)
        xglut_hw_shading = FALSE;
    else
        xglut_hw_shading = TRUE;

    free (inq_info);

    ras = xgl_object_create(sys_st, XGL_WIN_RAS, &obj_desc,
        XGL_DEV_COLOR_TYPE, color_type, 0);

    return(ras);
}

/*****
 * get_color_table - access function for the color_table object
 *****/

Xgl_color *get_color_table() {
    extern Xgl_color color_table[COLOR_SIZE];
    Xgl_color *p_color_table;

    p_color_table = color_table;
    return(p_color_table);
}

/*****
 * get_cmap_info - access function for the cmap_info object
 *****/

Xgl_color_list *get_cmap_info() {
    extern Xgl_color_list cmap_info;

    return &cmap_info;
}

/*****

```

```

*
* gal_reset_color_ - changes the color table defaults
*
*****/

void gal_reset_color_(index, r, g, b)
int      *index;
float     *r, *g, *b;

{
    extern Xgl_color_rgb  default_rgb[COLOR_SIZE];
    Xgl_sys_st      *sys_st;

    sys_st = get_sys_state();

    if (*sys_st) {
        printf(" \n \n \n %s \n", GAL_ERROR);
        printf("WARNING - NO
                COLOR RESETTING IN GAL_RESET_COLOR. YOU MUST CALL THIS
                PROCEDURE BEFORE GAL_INIT_COLOR. \n");
        printf(" \n \n \n %s
                \n", TRAIL);
    }

    if ((*r < 0. | *r > 1.0) | (*g < 0. | *g > 1.0) | (*b < 0. | *b >
        1.0))
        printf("\n \n Unable to reset color ... check all r,g,b
                values are less than or equal to 1.0 !!! \n");
    else
    {
        default_rgb[*index].r = *r;
        default_rgb[*index].g = *g;
        default_rgb[*index].b = *b;
    }
}

/*****
*
* get_dbuf_on - access function returns 1 if double buffers are set,
* 0 if not
*
* called from - all primitives which use color index.
*****/
Xgl_usgn32 get_dbuf_on() {
    extern Xgl_usgn32  set_dbuf;

    return(set_dbuf);
}

/*****
*
* get_gcache - access function for the cmap_info object
*
*****/

Xgl_gcache *get_gcache() {
    extern Xgl_gcache gcache;

    return &gcache;
}

/*****
#include "../include/xv.h"

```



```

#include "../include/color.h"

#define XGLUT_ITIMER_NULL      ((struct itimerval *)0)

static void    (*anim_proc)();
void    (*get_anim_proc())();
static
void    (*rev_proc)();
void    (*get_rev_proc())();
static void
start_stop_proc();
static void step_proc1();
static void step_proc2();
void    gal_quit_proc();
void    gal_animate_button_();
void
gal_step_anim_button_();
void    gal_step_rev_button_();

/*****
 *
 * gal_animate_button_ - this button is a customized feature that repeatedly
 * calls the procedure passed to it until the button is toggled off
 *
 * processes called - get_panel,nullcpy,xv_create
 *
 *****/

void gal_animate_button_(process, x_pos, y_pos, label, n)
int    *x_pos, *y_pos, n;
char    *label;
void    (*process)();
{
    extern void    (*anim_proc)(); /* this is a global procedure
                                    that is assigned a value passed by
                                    the user */

    char    *string = "";
    Panel    *panel;

    panel = get_panel();
    anim_proc = process;

    if (!(*panel)) {
        printf("\n\n\n %s\n",GAL_ERROR) ;
        printf("ATTEMPT TO CREATE GAL_ANIMATE_BUTTON BEFORE PANEL!\n\n\n");
        printf("\n\n\n %s\n",TRAIL) ;
        exit (1);
    } else {

        (void) nullcpy(string, label, n);

        (void) xv_create (*panel, PANEL_BUTTON,
            PANEL_ITEM_X, xv_col (*panel, *x_pos),
            PANEL_ITEM_Y, xv_row (*panel, *y_pos),
            PANEL_LABEL_STRING, string, PANEL_NOTIFY_PROC,
            start_stop_proc, NULL);
    }
}

/*****
 *
 * gal_step_anim_button_ - this button is a customized feature that toggles the
 * the animation off and then advances by 1 frame when pressed again
 *
 *****/

```

```

* processes called - get_panel,nullcpy,xv_create
*
*****/

void gal_step_anim_button_(process, x_pos, y_pos, label, n)
int      *x_pos, *y_pos, n;
char      *label;
void      (*process)();
{
    extern void      (*anim_proc)(); /* a global process passed by
    the user*/
    char      *string = "";
    Panel      *panel;

    panel = get_panel();
    anim_proc = process;

    if (!(*panel)) {
        printf("\n \n \n %s \n",GAL_ERROR) ;
        printf("ATTEMPT TO CREATE GAL_STEP_ANIM_BUTTON BEFORE PANEL! \n
        \n");
        printf("\n \n \n %s \n",TRAIL) ;
        exit (1);
    } else {

        (void) nullcpy(string, label, n);

        (void) xv_create (*panel, PANEL_BUTTON,
            PANEL_ITEM_X, xv_col (*panel, *x_pos),
            PANEL_ITEM_Y, xv_row (*panel, *y_pos),
            PANEL_LABEL_STRING, string, PANEL_NOTIFY_PROC,
            step_procl, NULL);
    }
}

/*****
* this procedure steps through the animation, frame-by frame, using
* an XView timer function
*
* procedures called - notify_set_itimer_func()
*
* called from gal_step_anim_button_()
*
*****/

static void step_procl() {
    void (*anim_proc)();

    Frame      *frame;

    frame = get_frame();

    anim_proc = get_anim_proc();

    if(toggle) {
        toggle = 0;
        (void)notify_set_itimer_func(*frame, *anim_proc,
            ITIMER_REAL,
            XGLUT_ITIMER_NULL,
            XGLUT_ITIMER_NULL);
    }
    anim_proc();
} /* step_proc */

```

```

/*****
 * the quit button ends the session
 *
 * procedure called - xv_destroy_safe()
 *
 *****/

void gal_quit_proc() {
    Frame    *frame;

    frame = get_frame();

    if (xv_destroy_safe(*frame) == XV_OK)
        exit(0);
}

/*****
 * start/stop for the animation uses XView timer functions attributes
 * to toggle the repetitive callback on and off.
 *
 * called from - gal_animate_button_()
 *
 * procedure called - notify_set_itimer_func ()
 *****/

static void start_stop_proc() {
    void (*anim_proc)();

    Frame    *frame;

    frame = get_frame();

    anim_proc = get_anim_proc();

    if (toggle ^= 1) {
        (void)notify_set_itimer_func (*frame, *anim_proc, ITIMER_REAL,
                                     &NOTIFY_POLLING_ITIMER,
                                     XGLUT_ITIMER_NULL);
    } else {
        (void)notify_set_itimer_func (*frame, *anim_proc, ITIMER_REAL,
                                     XGLUT_ITIMER_NULL,
                                     XGLUT_ITIMER_NULL);
    }
} /* start_stop_proc */

/*****
 * access function for the animation procedure
 *****/

void (*get_anim_proc())() {
    extern void (*anim_proc)();

    return (anim_proc);
}

/*****

#include "../include/xv.h"

```

```

static int dbuf_alloc; /* flag to indicate status of buffering */

/*****
 * xglut_dbuf_init() - A buffer color initialization program. This was lifted
 * from Sun's demo and accomodates buffers of different sizes.
 *
 * called from - gal_set_3d_buffer_(),gal_set_2d_buffer_()
 *
 * subroutines called - xgl_object_set, xgl_object_get, xgl_context_new_frame
 *****/

void xglut_dbuf_init(dbuf_information)
Xglut_dbuf_info *dbuf_information;
{
    Xgl_win_ras      wras;
    Xgl_color         color_table0[256];
    Xgl_color         color_table1[256];
    Xgl_color_list    cmap_info0;
    Xgl_color_list    cmap_info1;
    Xgl_sgn32         i, ncolors, nbits;
    Xgl_sys_state     *sys_st;
    extern int        dbuf_alloc;

    sys_st = get_sys_state(); /* check USER supplied context */
    if
    (!(dbuf_information->ctx)) {
        printf("\n \n \n %s \n",GAL_ERROR) ;
        printf("DOUBLE BUFFER ERROR CHECK TO SEE GAL_INIT_COLOR
                INIALIZED BEFORE GAL_SET_#D_BUFFER \n");
        printf("\n \n \n %s \n",TRAIL) ;
        return;
    }

    /* get window raster */ xgl_object_get(dbuf_information->ctx,
        XGL_CTX_DEVICE, &wras);
    if (!wras) {
        printf("\n \n \n %s \n",GAL_ERROR) ;
        printf("DOUBLE BUFFER ERROR CHECK TO SEE GAL_INIT_COLOR
                INIALIZED BEFORE GAL_SET_#D_BUFFER \n");
        printf("\n \n \n %s \n",TRAIL) ;
        return;
    }

    /* request double buffering from window raster */
    xgl_object_set(wras, XGL_WIN_RAS_BUFFERS_REQUESTED, 2, 0);
    dbuf_information->buffers_requested = 2;

    /* get number of buffers available in hardware underlying window
       raster */ xgl_object_get(wras,
        XGL_WIN_RAS_BUFFERS_ALLOCATED,
        &(dbuf_information->buffers_allocated));

    /* set dbuffering flags accordingly */ if
    (dbuf_information->buffers_allocated == 2) {
        /* great!!! HW double buffering support */ dbuf_alloc =
            dbuf_information->buffers_allocated;
        dbuf_information->cmap_dbuffering = FALSE;
        dbuf_information->xgl_dbuffering = TRUE;
    } else {
        /* we must do color map double buffering */
        dbuf_information->cmap_dbuffering = TRUE;
        dbuf_information->xgl_dbuffering = FALSE;
    }

    /* set dbuf_information accordingly */ if

```

```

(dbuf_information->xgl_dbuffering) {
    xgl_object_set(wras,
        XGL_WIN_RAS_BUF_DISPLAY, 0, XGL_WIN_RAS_BUF_DRAW, 0, 0);
    xgl_context_new_frame (dbuf_information->ctx);
    xgl_object_set(wras,
        XGL_WIN_RAS_BUF_DISPLAY, 1, XGL_WIN_RAS_BUF_DRAW, 1, 0);
    xgl_context_new_frame (dbuf_information->ctx);
    xgl_object_set(wras,
        XGL_WIN_RAS_BUF_DISPLAY, 0, XGL_WIN_RAS_BUF_DRAW, 0, 0);
    xgl_object_get(wras,
        XGL_WIN_RAS_BUF_DISPLAY, &(dbuf_information->buf_display));
    xgl_object_get(wras,
        XGL_WIN_RAS_BUF_DRAW, &(dbuf_information->buf_draw));
    xgl_object_get(wras,
        XGL_WIN_RAS_BUF_MIN_DELAY,
        &(dbuf_information->buf_min_delay));
    dbuf_information->current_buffer_is_buffer_0 = TRUE;
    dbuf_information->buf0_pm = -1;
    dbuf_information->buf1_pm = -1;
    if (xglut_hw_color_type == XGL_COLOR_INDEX) {
        Xgl_color_list cmap_info;

        cmap_info.start_index = 0;
        cmap_info.length =
            dbuf_information->number_of_colors_per_buffer;
        cmap_info.colors = dbuf_information->color_table;
        dbuf_information->cmap0 = xgl_object_create(*sys_st,
            XGL_CMAP, 0, XGL_CMAP_COLOR_TABLE_SIZE,
            cmap_info.length, XGL_CMAP_COLOR_TABLE, &cmap_info, 0);
        dbuf_information->cmap1 = dbuf_information->cmap0;
        xgl_object_set(wras,
            XGL_RAS_COLOR_MAP, dbuf_information->cmap0, 0);
    }
}
else if (dbuf_information->cmap_dbuffering) {
    /* check args in data structure for color map double buffering
    */ if (dbuf_information->number_of_colors_per_buffer <= 0) {
        printf("\n \n \n %s \n", GAL_ERROR) ;
        printf("ERROR -
            INVALID NUMBER OF DOUBLE BUFFER COLORS\n");
        printf("\n \n
            \n %s \n", TRAIL) ;
        dbuf_information->cmap_dbuffering =
            FALSE;
        return;
    }

    if (!(dbuf_information->color_table)) {
        printf("\n \n \n %s \n", GAL_ERROR) ;
        printf("ERROR - MISSING COLOR TABLES\n");
        printf("\n \n \n %s \n", TRAIL) ;
        dbuf_information->cmap_dbuffering = FALSE;
        return;
    }

    if (xglut_hw_color_type == XGL_COLOR_RGB) {
        printf("\n \n \n %s \n", GAL_ERROR) ;
        printf("ERROR - CANNOT DO COLOR MAP DOUBLE BUFFERING,
            RGB DEVICE\n");
        printf("\n \n \n %s \n", TRAIL) ;
        return;
    }

    /* setup plane mask for number of buffers and */ if
        (dbuf_information->number_of_colors_per_buffer <= 2) {
        dbuf_information->buf0_pm = 0x01; /* buffer 0 is bit 0 */
    }
}

```

```

        dbuf_information->buf1_pm = 0x02; /* buffer 1 is bit 1 */
        ncolors = 4;
        nbits = 1;
    } else if
(dbuf_information->number_of_colors_per_buffer <= 4) {
    dbuf_information->buf0_pm = 0x03; /* buffer 0 is bits 0,1
        */
    dbuf_information->buf1_pm = 0x0c; /* buffer 1 is bits
        2,3 */
    ncolors = 16;
    nbits = 2;
} else if
(dbuf_information->number_of_colors_per_buffer <= 8) {
    dbuf_information->buf0_pm = 0x07; /* buffer 0 is bits 0,1,2
        */
    dbuf_information->buf1_pm = 0x38; /* buffer 1 is bits
        3,4,5 */
    ncolors = 64;
    nbits = 3;
} else if
(dbuf_information->number_of_colors_per_buffer <= 16) {
    dbuf_information->buf0_pm = 0x0f; /* buffer 0 is bits
        0,1,2,3 */
    dbuf_information->buf1_pm = 0xf0; /* buffer 1 is
        bits 4,5,6,7 */
    ncolors = 256;
    nbits = 4;
} else {
    printf("\n \n \n %s \n", GAL_ERROR) ;
    printf("ERROR - MAXIMUM OF 16 COLORS SUPPORTED\n ");
    printf("\n \n \n %s \n", TRAIL) ;

    return;
}

#define DBICT dbuf_information->color_table

for (i = 0; i < ncolors; i++) {
    int i0 = i & dbuf_information->buf0_pm;
    int i1 = (i >>
        nbits) & dbuf_information->buf0_pm;

    color_table0[i].rgb = DBICT[i0].rgb;
    color_table1[i].rgb = DBICT[i1].rgb;
}

cmap_info0.start_index = 0;
cmap_info0.length = ncolors;
cmap_info0.colors = color_table0;
dbuf_information->cmap0 = xgl_object_create(*sys_st,
    XGL_CMAP, 0, XGL_CMAP_COLOR_TABLE_SIZE, ncolors,
    XGL_CMAP_COLOR_TABLE, &cmap_info0, 0);

cmap_info1.start_index = 0;
cmap_info1.length = ncolors;
cmap_info1.colors = color_table1;
dbuf_information->cmap1 = xgl_object_create(*sys_st,
    XGL_CMAP, 0, XGL_CMAP_COLOR_TABLE_SIZE, ncolors,
    XGL_CMAP_COLOR_TABLE, &cmap_info1, 0);

dbuf_information->bits_per_buffer = nbits;

dbuf_information->current_buffer_is_buffer_0 = TRUE;
dbuf_information->buf_display = 0;
dbuf_information->buf_draw =
1;

```

```

dbuf_information->buf_min_delay = 0;

/* set USER context and raster to first color map (or buffer) */
xgl_object_set(dbuf_information->ctx,
XGL_CTX_PLANE_MASK, dbuf_information->buf0_pm, 0);
xgl_object_set(wras,
XGL_RAS_COLOR_MAP, dbuf_information->cmapl, 0);
} else {
    printf("\n\n\n %s\n",GAL_ERROR) ;
    printf("ERROR - UNABLE TO
            CREATE 2 BUFFERS FOR DOUBLE BUFFERING\n");
    printf("\n\n\n %s
            \n",TRAIL) ;
    return;
}
}

/*****
 * xglut_dbuf_on() - creates the double buffers. The program searches for
 * hardware buffers. If they are absent color map double buffering is
 * implemented. A slightly modified version of Sun's demo.
 *
 * called from - gal_set_3d_buffer,gal_set_2d_buffer
 *
 * subroutines called - xgl_object_set,xgl_object_get
 *****/

void xglut_dbuf_on(dbuf_information)
Xglut_dbuf_info *dbuf_information;
{
    Xgl_win_ras      wras;
    Xgl_ctx          ctx;
    Xgl_ctx_new_frame_action    nf_save;

    /* check USER supplied context */ if (!(ctx =
    dbuf_information->ctx)) {
        printf("\n\n\n %s\n",GAL_ERROR) ;
        printf("ERROR - NO GRAPHICAL CONTEXT. CHECK THAT
                GAL_INIT_COLOR HAS BEEN SET
                BEFORE GAL_SET_2D_BUFFER/GAL_SET_3D_BUFFER. \n");
        printf("\n\n\n %s\n",TRAIL) ;

        return;
    } /* get window raster */
    xgl_object_get(dbuf_information->ctx, XGL_CTX_DEVICE, &wras);
    if
    (!wras) {
        printf("ERROR - xglut_dbuf_on: no window raster in context.
                Check that gal_init_color has been invoked prior to
                gal_set_3d_buffer/gal_set_2d_buffer\n");
        return;
    }

    if (dbuf_information->xgl_dbuffering) {
        xgl_object_get(ctx, XGL_CTX_NEW_FRAME_ACTION, &nf_save);

        xgl_object_set(ctx,
            XGL_CTX_NEW_FRAME_ACTION,
            nf_save|XGL_CTX_NEW_FRAME_SWITCH_BUFFER, 0);

        xgl_object_set (wras, XGL_WIN_RAS_BUFFERS_REQUESTED, 2, 0);

        xgl_object_set(wras,
            XGL_WIN_RAS_BUF_DISPLAY, 0, XGL_WIN_RAS_BUF_DRAW, 1, 0);
    }
}

```

```

    dbuf_information->buf_display = 0;
    dbuf_information->buf_draw = 1;

    } else if (dbuf_information->cmap_dbuffering) {
        xgl_object_get(ctx, XGL_CTX_NEW_FRAME_ACTION, &nf_save);

        xgl_object_set(ctx,
            XGL_CTX_NEW_FRAME_ACTION,
            nf_save|XGL_CTX_NEW_FRAME_VRETRACE, 0);

        if (dbuf_information->current_buffer_is_buffer_0) {
            xgl_object_set(wras,
                XGL_RAS_COLOR_MAP, dbuf_information->cmap0, 0);
            xgl_object_set(ctx,
                XGL_CTX_PLANE_MASK, dbuf_information->buf1_pm, 0);
            dbuf_information->buf_display = 0;
            dbuf_information->buf_draw = 1;
        } else {
            xgl_object_set(wras,
                XGL_RAS_COLOR_MAP, dbuf_information->cmap1, 0);
            xgl_object_set(ctx,
                XGL_CTX_PLANE_MASK, dbuf_information->buf0_pm, 0);
            dbuf_information->buf_display = 1;
            dbuf_information->buf_draw = 0;
        }
        dbuf_information->current_buffer_is_buffer_0 ^= TRUE;
    }
}

/*****
 * switch_buffer() - This routine switches buffers by toggling between a
 * hidden buffer (to which the object is drawn, after the previous object
 * is cleared) and a displayed buffer.
 *
 * subroutines called - xgl_object_get, xgl_object_set
 *****/

void switch_buffer(dbuf_information) Xglut_dbuf_info *dbuf_information;
{
    Xgl_win_ras wras;

    /* check USER supplied context */
    if (!(dbuf_information->ctx)) {
        printf("\n\n\n%s\n",GAL_ERROR) ;
        printf("ERROR - UNABLE TO SWITCH BUFFER, NO
            GRAPHICAL CONTEXT. CHECK IF GAL_INIT_COLOR
            IMPLEMENTED\n");
        printf("\n\n\n%s\n",TRAIL) ;
        return;
    } /*
        get window raster */
    xgl_object_get(dbuf_information->ctx,
        XGL_CTX_DEVICE, &wras);
    if (!wras) {
        printf("\n\n\n%s\n",GAL_ERROR) ;
        printf("ERROR - UNABLE TO SWITCH BUFFER, NO
            GRAPHICAL CONTEXT. CHECK IF GAL_INIT_COLOR
            IMPLEMENTED\n");
        printf("\n\n\n%s\n",TRAIL) ;
        return;
    }

    /* if doing HW dbuffering then set bit in context new frame and
        let xgl do the switch; otherwise switch the cmap buffers */
    if (dbuf_information->xgl_dbuffering) {

```



```

Xgl_ctx          ctx;

ctx = dbuf_information->ctx;

/* just do a new frame because xglut_dbuf_on should turn
 * ON the SWITCH attribute */
xgl_context_new_frame(ctx);

/* switch buffers */
xgl_object_get(wras,
               XGL_WIN_RAS_BUF_DISPLAY, &(dbuf_information->buf_display));
xgl_object_get(wras,
               XGL_WIN_RAS_BUF_DRAW, &(dbuf_information->buf_draw));
xgl_object_get(wras,
               XGL_WIN_RAS_BUF_MIN_DELAY,
               &(dbuf_information->buf_min_delay));
dbuf_information->current_buffer_is_buffer_0 =
    (dbuf_information->buf_display == 0);
} else if
(dbuf_information->cmap_dbuffering) {
    Xgl_ctx          ctx;

    ctx = dbuf_information->ctx;

    if (dbuf_information->current_buffer_is_buffer_0) {
        xgl_object_set(wras,
                       XGL_RAS_COLOR_MAP, dbuf_information->cmap0, 0);
        xgl_object_set(ctx,
                       XGL_CTX_PLANE_MASK, dbuf_information->buf1_pm, 0);
        dbuf_information->buf_display = 0;
        dbuf_information->buf_draw = 1;
        dbuf_information->buf_min_delay = 0;
    } else {
        xgl_object_set(wras,
                       XGL_RAS_COLOR_MAP, dbuf_information->cmap1, 0);
        xgl_object_set(ctx,
                       XGL_CTX_PLANE_MASK, dbuf_information->buf0_pm, 0);
        dbuf_information->buf_display = 1;
        dbuf_information->buf_draw = 0;
        dbuf_information->buf_min_delay = 0;
    }
    dbuf_information->current_buffer_is_buffer_0 ^= TRUE;
    xgl_context_new_frame(ctx); /* clear back buffer planes */
}
else /* unbuffered switch */ {
    Xgl_ctx          ctx;

    ctx = dbuf_information->ctx;

    xgl_context_new_frame(ctx); /* clear back buffer planes */
    return;
}
}

/*****
 * access function for the number of buffers allocated
 *****/

int get_dbuf_alloc() {
    extern int dbuf_alloc;

    return(dbuf_alloc);
}

```

Appendix 2d - Views and Transformations

```

/*****
 *
 * These subroutines deal with the the view model and the view space.
 *
 * Duncan Napier, McMaster University, Hamilton, Ontario, Canada
 * October, 1992
 *
 *****/
#include "../include/xv.h"
#include "../include/color.h"

/*****
 * gal_pan_x() - This procedure performs a translation in the x-y
 * plane and follows with a zoom (enlargement). Note that no
 * interference with the rotation transforms results because the
 * REPLACE attribute affects only the global model transform, and
 * not the view transform (i.e. the rotate transform)
 *
 * routines called - xgl_object_get, xgl_transform_scale,
 * xgl_transform_translate.
 *****/

void gal_3d_pan_and_zoom(x_shift,y_shift, factor)
float *x_shift,*y_shift, *factor;
{
    Xgl_pt      tmp_pt,zoom_pt;
    Xgl_trans    xshift_trans,zoom_trans;
    Xgl_pt_f3d    shift;
    Xgl_pt_f3d    zoom;
    Xgl_3d_ctx    *ctx3d;

    ctx3d = get_3d_ctx();

    xgl_object_get(*ctx3d, XGL_CTX_GLOBAL_MODEL_TRANS,
&xshift_trans);
    xgl_object_get(*ctx3d,
XGL_CTX_GLOBAL_MODEL_TRANS, &zoom_trans);

    /* zoom factors <= 0 are forbidden */

    if (*factor <= 0.) *factor = 1.;

    /* set up the zoom transformation */

    zoom.x = *factor; zoom.y = *factor; zoom.z = *factor;
    zoom_pt.pt_type = XGL_PT_F3D; zoom_pt.pt.f3d = &zoom;

    /* set up the translation transformation */

    /* don't forget to scale the translation */ shift.x =
*x_shift*(*factor);
    shift.y = *y_shift*(*factor);
    shift.z = 0.;
    tmp_pt.pt_type = XGL_PT_F3D;
    tmp_pt.pt.f3d = &shift;

    /* carry out the transforms on the object */

    xgl_transform_scale(zoom_trans, &zoom_pt, XGL_TRANS_REPLACE);
    xgl_transform_translate(xshift_trans, &tmp_pt,
XGL_TRANS_POSTCONCAT);
}

void gal_2d_pan_and_zoom(x_shift,y_shift, factor)

```

```

float    *x_shift,*y_shift,*factor;
{
    Xgl_pt      tmp_pt, zoom_pt;
    Xgl_trans    xshift_trans, zoom_trans;
    Xgl_pt_f2d    shift;
    Xgl_pt_f2d    zoom;
    Xgl_2d_ctx    *ctx2d;

    ctx2d = get_2d_ctx();

    xgl_object_get(*ctx2d, XGL_CTX_GLOBAL_MODEL_TRANS,
&xshift_trans); xgl_object_get(*ctx2d,
XGL_CTX_GLOBAL_MODEL_TRANS, &zoom_trans);

    /* zoom factors <= 0 are forbidden */

    if (*factor <= 0.) *factor = 1.;

    /* set up the zoom transformation */

    /* don't forget to scale the translation */
    zoom.x = *factor;
    zoom.y = *factor;
    zoom_pt.pt_type = XGL_PT_F2D;
    zoom_pt.pt.f2d = &zoom;

    /* set up the transformation */

    shift.x = *x_shift*(*factor);
    shift.y = *y_shift*(*factor);
    tmp_pt.pt_type = XGL_PT_F2D;
    tmp_pt.pt.f2d = &shift;

    /* carry out the transforms on the object */

    xgl_transform_scale(zoom_trans, &zoom_pt, XGL_TRANS_REPLACE);
    xgl_transform_translate(xshift_trans, &tmp_pt,
XGL_TRANS_POSTCONCAT);
}

/*****/

#include "../include/xv.h"
#include "../include/color.h"

/*****/
* This procedure carries out rotational transformations of a 3d
* context object. The slider values determine the rotations about the
* x, y and z axes.
*
* procedures called xgl_object_get, xgl_transform_rotate
*****/

void gal_rotate_(x_angle, y_angle, z_angle)
float    *x_angle, *y_angle, *z_angle;
/* Slider values passed from FORTRAN */
{

    Xgl_3d_ctx    *ctx3d;
    Xgl_trans    view_trans;

    ctx3d = get_3d_ctx();

```

```

/* create the view transform attribute */
xgl_object_get(*ctx3d, XGL_CTX_VIEW_TRANS, &view_trans);

/* Carry out the required transformations given the
   slider positions. Note the specific order of the
   transformations */
xgl_transform_rotate(view_trans,*x_angle*3.14159/180.,
                     XGL_AXIS_X, XGL_TRANS_REPLACE);
xgl_transform_rotate(view_trans,*y_angle*3.14159/180.,
                     XGL_AXIS_Y, XGL_TRANS_POSTCONCAT);
xgl_transform_rotate(view_trans,*z_angle*3.14159/180.,
                     XGL_AXIS_Z, XGL_TRANS_POSTCONCAT);
}

/*****/

#include "../include/xv.h"

static short   aspect_flag;    /* a flag to ensure that the viewport
is set */

/*****
 * gal_(3d/2d)_aspect_set_() - procedures to set the boundaries of the
 * 3D/2D view space. This procedure maintains the aspect ratio
 * throughout widow resizing.
 *
 * procedures called - xgl_object_set()
 *****/

void gal_3d_aspect_set_(xmin, xmax, ymin, ymax, zmin, zmax)
float   *xmin, *xmax, *ymin, *ymax, *zmin, *zmax;
{
    Xgl_3d_ctx      *ctx;
    Xgl_bounds_f3d vdc_3d_window;
    extern
    short aspect_flag;

    aspect_flag = 1;
    /* pass the model space dimensions and convert them to device coords */
    vdc_3d_window.xmin = *xmin;
    vdc_3d_window.xmax = *xmax;
    vdc_3d_window.ymin = *ymin;
    vdc_3d_window.ymax = *ymax;
    vdc_3d_window.zmin = *zmin;
    vdc_3d_window.zmax = *zmax;

    ctx = get_3d_ctx();

    if (!*ctx) {
        printf("\n\n\n %s\n",GAL_ERROR) ;
        printf("ERROR - UNABLE TO SET GAL_3D_ASPECT_SET,
                CHECK THAT GAL_INIT_COLOR IS SET\n");
        printf("\n\n\n %s\n",TRAIL) ;
        exit (1);
    }

    xgl_object_set(*ctx, XGL_CTX_VDC_WINDOW, &vdc_3d_window, NULL);

    xgl_object_set(*ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_ASPECT,
                  NULL);
}

void gal_2d_aspect_set_(xmin, xmax, ymin, ymax)
float   *xmin, *xmax,*ymin, *ymax;

```

```

{
    Xgl_2d_ctx      *ctx;
    Xgl_bounds_f2d vdc_2d_window;
    extern
    short aspect_flag;

    aspect_flag = 1;
    vdc_2d_window.xmin = *xmin;
    vdc_2d_window.xmax = *xmax;
    vdc_2d_window.ymin = *ymin;
    vdc_2d_window.ymax = *ymax;

    ctx = get_2d_ctx();

    if (!*ctx) {
        printf("\n\n\n %s\n", GAL_ERROR) ;
        printf("ERROR - UNABLE TO SET GAL_2D_ASPECT_SET
            , CHECK THAT GAL_INIT_COLOR IS SET\n\n ");
        printf("\n\n\n %s\n", TRAIL) ;
        exit (1);
    }

    xgl_object_set(*ctx, XGL_CTX_VDC_WINDOW, &vdc_2d_window, NULL);

    xgl_object_set(*ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_ASPECT, NULL);
}

/*****
 * gal_(3d/2d)_aspect_free_() - procedures to set the boundaries of the
 * 3D/2D view space. This procedure adjusts the aspect ratio to ensure
 * as much of the canvas is covered during resizing.
 *
 * procedures called - xgl_object_set()
 *
 *****/

void gal_3d_aspect_free_(xmin, xmax, ymin, ymax, zmin, zmax)
float *xmin, *xmax, *ymin, *ymax, *zmin, *zmax;
{
    Xgl_3d_ctx      *ctx;
    Xgl_bounds_f3d vdc_3d_window;
    extern
    short aspect_flag;

    aspect_flag = 1;
    vdc_3d_window.xmin = *xmin;
    vdc_3d_window.xmax = *xmax;
    vdc_3d_window.ymin = *ymin;
    vdc_3d_window.ymax = *ymax;
    vdc_3d_window.zmin = *zmin;
    vdc_3d_window.zmax = *zmax;

    ctx = get_3d_ctx();

    if (!*ctx) {
        printf("\n\n\n %s\n", GAL_ERROR) ;
        printf("ERROR - UNABLE TO SET GAL_3D_ASPECT_FREE
            , CHECK THAT GAL_INIT_COLOR IS SET\n\n ");
        printf("\n\n\n %s\n", TRAIL) ;
        exit (1);
    }

    xgl_object_set(*ctx, XGL_CTX_VDC_WINDOW, &vdc_3d_window, NULL);

    xgl_object_set(*ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_ALL, NULL);
}

```

```

}

void gal_2d_aspect_free_(xmin, xmax, ymin, ymax)
float  *xmin, *xmax, *ymin, *ymax;
{
    Xgl_2d_ctx      *ctx;
    Xgl_bounds_f2d vdc_2d_window;
    extern
    short aspect_flag;

    aspect_flag = 1;
    vdc_2d_window.xmin = *xmin;
    vdc_2d_window.xmax = *xmax;
    vdc_2d_window.ymin = *ymin;
    vdc_2d_window.ymax = *ymax;

    ctx = get_2d_ctx();

    if (!*ctx) {
        printf("\n \n \n %s \n", GAL_ERROR) ;
        printf("ERROR- UNABLE TO SET GAL_2D_ASPECT_FREE
                , CHECK THAT GAL_INIT_COLOR IS SET \n");
        printf("\n \n \n %s \n", TRAIL) ;
        exit (1);
    }

    xgl_object_set(*ctx, XGL_CTX_VDC_WINDOW, &vdc_2d_window, NULL);

    xgl_object_set(*ctx, XGL_CTX_VDC_MAP, XGL_VDC_MAP_ALL, NULL);
}

/*****
 * get_aspect_flag - acces function for the viewport dimension flag
 *
 * called by - gal_end()
 *****/
short get_aspect_flag() {
    extern short    aspect_flag;

    return aspect_flag;
}

```

Appendix 2e - 2D Primitives


```

#include "../include/xv.h"
#include "../include/color.h"

/*****
 * gal_2d_unfilled_circle_() - a primitive drawing routine to draw a
 * circle routines called - get_2d_ctx, xgl_object_set,xgl_multi_circle
 *****/

void gal_2d_unfilled_circle_(color_index, rad, x, y)
int      *color_index;
float    *x, *y, *rad; /* x and y coordinates and radius */
{
    Xgl_2d_ctx      *ctx;
    Xgl_color        color;
    Xgl_circle_list  circle_list;
    Xgl_pt_f2d        center;
    Xgl_circle_f2d    circs;

    /*
     * get a smooth circle */

    ctx = get_2d_ctx();

    /* define the closeness of the circle points */
    xgl_object_set(*ctx, XGL_CTX_CURVE_APPROX, XGL_CURVE_METRIC_VDC,
                   XGL_CTX_CURVE_APPROX_VALUE, 1.0, NULL);

    /* draw the circle */
    circle_list.type = XGL_MULTICIRCLE_F2D;
    circle_list.num_circles = 1;
    circle_list.bbox = 0;
    circs.center.flag = 0;
    circs.center.x = *x;
    circs.center.y = *y;
    circs.radius = *rad;
    circle_list.circles.f2d = &circs;

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
        *color_index; } else color.index = *color_index;

    /* set fill to hollow */ xgl_object_set (*ctx,
        XGL_CTX_SURF_FRONT_FILL_STYLE,XGL_SURF_FILL_HOLLOW,
        XGL_CTX_SURF_FRONT_COLOR, &color, NULL);

    xgl_multicircle(*ctx, &circle_list);

}

/*****
 * gal_2d_unfilled_rectangle_() - a primitive drawing routine to draw a
 * rectangle routines called - get_2d_ctx, xgl_object_set,xgl_multi_rectangle
 *****/

void gal_2d_unfilled_rectangle_(color_index, x, y)
int      *color_index;
float    x[], y[]; /* the max/min x values and max/min y values */
{
    Xgl_2d_ctx      *ctx;
    Xgl_color        color;
    Xgl_rect_list    rect_list;
    Xgl_pt_f2d        center;

```

```

Xgl_rect_f2d      rects;

    ctx = get_2d_ctx();

    /* draw the rectangles */
    rect_list.rect_type = XGL_MULTIRECT_F2D;
    rect_list.num_rects = 1;
    rect_list.bbox = 0;
    rects.corner_min.x = x[0];
    rects.corner_min.y = y[0];
    rects.corner_min.flag = 0;
    rects.corner_max.x = x[1];
    rects.corner_max.y = y[1];
    rect_list.rects.f2d = &rects;

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
        *color_index; } else color.index = *color_index;

    xgl_object_set (*ctx,
        XGL_CTX_SURF_FRONT_FILL_STYLE,XGL_SURF_FILL_HOLLOW,XGL_CTX_SURF_FRONT_COLOR,&color
, NULL);

    xgl_multirectangle(*ctx, &rect_list);

}

/*****
 * gal_2d_filled_circle_() - a primitive drawing routine to draw a
 * filled circle routines called - get_2d_ctx, xgl_object_set,
 * xgl_multi_circle
 *****/

void gal_2d_filled_circle_(color_index, rad,  x, y)
int      *color_index;
float     *x, *y, *rad;
{
    Xgl_2d_ctx      *ctx;
    Xgl_color        color;
    Xgl_circle_list  circle_list;
    Xgl_pt_f2d        center;
    Xgl_circle_f2d    circs;

    ctx = get_2d_ctx();

    xgl_object_set(*ctx, XGL_CTX_CURVE_APPROX, XGL_CURVE_METRIC_VDC,
        XGL_CTX_CURVE_APPROX_VALUE, 1.0, NULL);

    /* draw the circles */
    circle_list.type = XGL_MULTICIRCLE_F2D;
    circle_list.num_circles = 1;
    circle_list.bbox = 0;
    circs.center.flag = 0;
    circs.center.x = *x;
    circs.center.y = *y;
    circs.radius = *rad;
    circle_list.circles.f2d = &circs;

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
        *color_index; } else color.index = *color_index;

    xgl_object_set (*ctx, XGL_CTX_SURF_FRONT_COLOR, &color, NULL);

```

```

        xgl_multicircle(*ctx, &circle_list);

}

/*****
 * gal_2d_unfilled_rectangle_() - a primitive drawing routine to draw a
 * rectangle routines called - get_2d_ctx, xgl_object_set,xgl_multi_rectangle
 *****/

void gal_2d_filled_rectangle_(color_index, x, y)
int      *color_index;
float     x[], y[];
{
    Xgl_2d_ctx      *ctx;
    Xgl_color        color;
    Xgl_rect_list    rect_list;
    Xgl_pt_f2d        center;
    Xgl_rect_f2d      rects;

    ctx = get_2d_ctx();

    /* draw the rectangles*/
    rect_list.rect_type = XGL_MULTIRECT_F2D;
    rect_list.num_rects = 1;
    rect_list.bbox = 0;
    rects.corner_min.x = x[0];
    rects.corner_min.y = y[0];
    rects.corner_min.flag = 0;
    rects.corner_max.x = x[1];
    rects.corner_max.y = y[1];
    rect_list.rects.f2d = &rects;

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
        *color_index; } else color.index = *color_index;

    xgl_object_set (*ctx, XGL_CTX_SURF_FRONT_COLOR, &color, NULL);

    xgl_multirectangle(*ctx, &rect_list);
}

/*****
 * gal_2d_filled_polygon_() - draws a filled polygon given an ordered
 * list of points. The polygon is filled with color color_index. The
 * user specifies the number of points and then gives the x and
 * y-coordinates of each point.
 *
 * routines called - xgl_object_set,xgl_polygon.
 *****/

void gal_2d_filled_polygon_(color_index, num_pts, list_pts_x,
list_pts_y)
float list_pts_x[], list_pts_y[];
int    *color_index, *num_pts;
{
    Xgl_2d_ctx      *ctx;
    Xgl_color        color;
    Xgl_pt_list      poly_pts;
    int              i;

```

```

/* allocate memory for the coordinates to be processed */

if (!(poly_pts.pts.f2d = (Xgl_pt_f2d
*)malloc(*num_pts*sizeof(Xgl_pt_f2d)))) {
    printf("memory allocation request in
    gal_2d_filled_polygon failed\n"); exit(1); }

poly_pts.pt_type = XGL_PT_F2D; poly_pts.bbox = NULL;
poly_pts.num_pts = *num_pts;

/* assign the vertices to their individual points */

for ( i=0; i<*num_pts; i++) {

    poly_pts.pts.f2d[i].x = list_pts_x[i];
    poly_pts.pts.f2d[i].y = list_pts_y[i];

}

if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
*color_index; } else color.index = *color_index;

ctx = get_2d_ctx();

xgl_object_set (*ctx, XGL_CTX_SURF_FRONT_COLOR, &color, NULL);

xgl_polygon (*ctx, XGL_FACET_NONE, NULL, NULL, 1, poly_pts);

free(poly_pts.pts.f2d); }

/*****
* gal_2d_unfilled_polygon()-draws an unfilled polygon given an
* ordered list of points. The polygon is filled with color
* color_index. The user specifies the number of points and then gives
* the x and y-coordinates of each point.
*
* routines called - xgl_object_set, xgl_polygon.
*****/

void gal_2d_unfilled_polygon(color_index, num_pts, list_pts_x,
list_pts_y)
float list_pts_x[], list_pts_y[];
int *color_index, *num_pts;
{
    Xgl_2d_ctx      *ctx;
    Xgl_color       color;
    Xgl_pt_list     poly_pts;
    int             i;

    /* allocate memory for the coordinates to be processed */

    if (!(poly_pts.pts.f2d = (Xgl_pt_f2d
*)malloc(*num_pts*sizeof(Xgl_pt_f2d)))) {
        printf("memory allocation request in
        gal_2d_filled_polygon failed\n"); exit(1); }

    poly_pts.pt_type = XGL_PT_F2D;
    poly_pts.bbox = NULL;
    poly_pts.num_pts = *num_pts;

    /* assign the vertices to their individual points */

```

```

    for ( i=0; i<*num_pts; i++) {

        poly_pts.pts.f2d[i].x = list_pts_x[i];
        poly_pts.pts.f2d[i].y = list_pts_y[i];

    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
        *color_index; } else color.index = *color_index;

    ctx = get_2d_ctx();

    xgl_object_set(*ctx,XGL_CTX_SURF_FRONT_FILL_STYLE,XGL_SURF_FILL_HOLLOW,
XGL_CTX_SURF_FRONT_COLOR, &color, NULL);

    xgl_polygon (*ctx, XGL_FACET_NONE, NULL, NULL, 1, poly_pts);

    free(poly_pts.pts.f2d); }

/*****
 * gal_2d_solid_line_() - draws a solid line given the color,
 * thickness, number of points and coordinates.
 *
 * routines called - malloc, xgl_object_set, xgl_multipolyline.
 *****/

void gal_2d_solid_line_(color_index, thickness, num_pts, list_pts_x,
list_pts_y)
float list_pts_x[], list_pts_y[], *thickness;
int *color_index, *num_pts;
{
    Xgl_2d_ctx      *ctx;
    Xgl_ccolor      color;
    Xgl_pt_list     line_pts;
    int             i;

    /* allocate memory for the coordinates to be processed */

    if (!(line_pts.pts.f2d = (Xgl_pt_f2d
*)malloc(*num_pts*sizeof(Xgl_pt_f2d)))) {
        printf("memory allocation request in gal_2d_solid_line
failed\n"); exit(1); }

    line_pts.pt_type = XGL_PT_F2D;
    line_pts.num_pts = *num_pts;
    line_pts.bbox = 0;

    for ( i=0; i<*num_pts; i++) {

        line_pts.pts.f2d[i].x = list_pts_x[i];
        line_pts.pts.f2d[i].y = list_pts_y[i];

    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
        *color_index; } else color.index = *color_index;

    ctx = get_2d_ctx();

    xgl_object_set(*ctx, XGL_CTX_LINE_WIDTH_SCALE_FACTOR,
        *thickness, XGL_CTX_LINE_COLOR, &color, NULL);

```

```

        xgl_multipolyline(*ctx, NULL, 1, line_pts);

        free(line_pts.pts.f2d); }

/*****
 * gal_2d_dotted_line_() - draws a dotted line given the color,
 * thickness, number of points and coordinates.
 *
 * routines called - malloc, xgl_object_set, xgl_multipolyline.
 *****/

void gal_2d_dotted_line_(color_index, thickness, num_pts, list_pts_x,
list_pts_y)
float list_pts_x[], list_pts_y[], *thickness;
int *color_index, *num_pts;
{
    Xgl_2d_ctx      *ctx;
    Xgl_color        color;
    Xgl_pt_list      line_pts;
    int              i;

    /* allocate memory for the coordinates to be processed */

    if (!(line_pts.pts.f2d = (Xgl_pt_f2d
*)malloc(*num_pts*sizeof(Xgl_pt_f2d)))) {
        printf("memory allocation request in gal_2d_dotted_line
        failed\n"); exit(1); }

    line_pts.pt_type = XGL_PT_F2D;
    line_pts.num_pts = *num_pts;
    line_pts.bbox = 0;

    for ( i=0; i<*num_pts; i++) {

        line_pts.pts.f2d[i].x = list_pts_x[i];
        line_pts.pts.f2d[i].y = list_pts_y[i];

    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
    *color_index; } else color.index = *color_index;

    ctx = get_2d_ctx();

    xgl_object_set(*ctx,
        XGL_CTX_LINE_WIDTH_SCALE_FACTOR, *thickness,
        XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNEED,
        XGL_CTX_LINE_PATTERN, xgl_lpat_dotted,
        XGL_CTX_LINE_COLOR, &color, NULL);

    xgl_multipolyline(*ctx, NULL, 1, line_pts);

    free(line_pts.pts.f2d); }

/*****
 * gal_2d_dashed_line_() - draws a dashed line given the color,
 * thickness, number of points and coordinates.
 *
 * routines called - malloc, xgl_object_set, xgl_multipolyline.
 *****/

```

```

void gal_2d_dashed_line_(color_index, thickness, num_pts, list_pts_x,
list_pts_y)
float list_pts_x[], list_pts_y[], *thickness;
int *color_index, *num_pts;
{
    Xgl_2d_ctx *ctx;
    Xgl_color color;
    Xgl_pt_list line_pts;
    int i;

    /* allocate memory for the coordinates to be processed */

    if (!(line_pts.pts.f2d = (Xgl_pt_f2d
*)malloc(*num_pts*sizeof(Xgl_pt_f2d)))) {
        printf("memory allocation request in gal_2d_dashed_line
failed\n"); exit(1); }

    line_pts.pt_type = XGL_PT_F2D;
    line_pts.num_pts = *num_pts;
    line_pts.bbox = 0;

    for ( i=0; i<*num_pts; i++) {

        line_pts.pts.f2d[i].x = list_pts_x[i];
        line_pts.pts.f2d[i].y = list_pts_y[i];

    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
*color_index; } else color.index = *color_index;

    ctx = get_2d_ctx();

    xgl_object_set(*ctx,
        XGL_CTX_LINE_WIDTH_SCALE_FACTOR, *thickness,
        XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNEDED,
        XGL_CTX_LINE_PATTERN, xgl_lpat_dashed,
        XGL_CTX_LINE_COLOR, &color, NULL);

    xgl_multipolyline(*ctx, NULL, 1, line_pts);

    free(line_pts.pts.f2d); }

/*****
 * gal_2d_dash_dotted_line_() - draws a dashed-dotted line given the
 * color, thickness, number of points and coordinates.
 *
 * routines called - malloc, xgl_object_set, xgl_multipolyline.
 *****/

void gal_2d_dash_dotted_line_(color_index, thickness, num_pts,
list_pts_x, list_pts_y)
float list_pts_x[], list_pts_y[], *thickness;
int *color_index, *num_pts;
{
    Xgl_2d_ctx *ctx;
    Xgl_color color;
    Xgl_pt_list line_pts;
    int i;

    /* allocate memory for the coordinates to be processed */

```

```

if (!(line_pts.pts.f2d = (Xgl_pt_f2d
*)malloc(*num_pts*sizeof(Xgl_pt_f2d)))) {
    printf("memory allocation request in
    gal_2d_dash_dotted_line failed\n"); exit(1); }

line_pts.pt_type = XGL_PT_F2D;
line_pts.num_pts = *num_pts;
line_pts.bbox = 0;

for ( i=0; i<*num_pts; i++) {

    line_pts.pts.f2d[i].x = list_pts_x[i];
    line_pts.pts.f2d[i].y = list_pts_y[i];

}

if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
*color_index; } else color.index = *color_index;

ctx = get_2d_ctx();

xgl_object_set(*ctx,
                XGL_CTX_LINE_WIDTH_SCALE_FACTOR, *thickness,
                XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNEED,
                XGL_CTX_LINE_PATTERN, xgl_lpat_dashed_dotted,
                XGL_CTX_LINE_COLOR, &color, NULL);

xgl_multipolyline(*ctx, NULL, 1, line_pts);

free(line_pts.pts.f2d); }

/*****
* gal_2d_dash_dot_line_() - draws a dashed-dotted line given the
* color, thickness, number of points and coordinates.
*
* routines called - malloc, xgl_object_set, xgl_multipolyline.
*****/

void gal_2d_dash_dot_line_(color_index, thickness, num_pts, list_pts_x,
list_pts_y)
float list_pts_x[], list_pts_y[], *thickness;
int *color_index, *num_pts;
{
    Xgl_2d_ctx *ctx;
    Xgl_color color;
    Xgl_pt_list line_pts;
    int i;

    /* allocate memory for the coordinates to be processed */

    if (!(line_pts.pts.f2d = (Xgl_pt_f2d
*)malloc(*num_pts*sizeof(Xgl_pt_f2d)))) {
        {
            printf("\n \n memory allocation request in
            gal_2d_dash_dot line failed\n"); exit(1); }

        line_pts.pt_type = XGL_PT_F2D;
        line_pts.num_pts = *num_pts;
        line_pts.bbox = 0;

```



```

    for ( i=0; i<*num_pts; i++) {

        line_pts.pts.f2d[i].x = list_pts_x[i];
        line_pts.pts.f2d[i].y = list_pts_y[i];

    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
        *color_index; } else color.index = *color_index;

    ctx = get_2d_ctx();

    xgl_object_set(*ctx,
        XGL_CTX_LINE_WIDTH_SCALE_FACTOR, *thickness,
        XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNEED,
        XGL_CTX_LINE_PATTERN, xgl_lpat_dash_dot,
        XGL_CTX_LINE_COLOR, &color, NULL);

    xgl_multipolyline(*ctx, NULL, 1, line_pts);

    free(line_pts.pts.f2d); }

/*****
 * gal_2d_dash_dot_dot_line_() - draws a dashed-dotted line given the
 * color, thickness, number of points and coordinates.
 *
 * routines called - malloc, xgl_object_set, xgl_multipolyline.
 *****/

void gal_2d_dash_dot_dot_line_(color_index, thickness, num_pts,
list_pts_x, list_pts_y)
float list_pts_x[], list_pts_y[], *thickness;
int *color_index, *num_pts;
{
    Xgl_2d_ctx      *ctx;
    Xgl_color        color;
    Xgl_pt_list      line_pts;
    int              i;

    /* allocate memory for the coordinates to be processed */

    if (!(line_pts.pts.f2d = (Xgl_pt_f2d
        *)malloc(*num_pts*sizeof(Xgl_pt_f2d)))) {
        printf("memory allocation request in
            gal_2d_dash_dot_dot line failed\n"); exit(1); }

    line_pts.pt_type = XGL_PT_F2D;
    line_pts.num_pts = *num_pts;
    line_pts.bbox = 0;

    for ( i=0; i<*num_pts; i++) {

        line_pts.pts.f2d[i].x = list_pts_x[i];
        line_pts.pts.f2d[i].y = list_pts_y[i]; }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
        *color_index; } else color.index = *color_index;

```

```

ctx = get_2d_ctx();

xgl_object_set(*ctx,
               XGL_CTX_LINE_WIDTH_SCALE_FACTOR, *thickness,
               XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNEDED,
               XGL_CTX_LINE_PATTERN, xgl_lpat_dash_dot_dotted,
               XGL_CTX_LINE_COLOR, &color, NULL);

xgl_multipolyline(*ctx, NULL, 1, line_pts);

free(line_pts.pts.f2d); }

/*****
 * gal_2d_long_dash_line_() - draws a dashed-dotted line given the
 * color, thickness, number of points and coordinates.
 *
 * routines called - malloc, xgl_object_set, xgl_multipolyline.
 *****/

void gal_2d_long_dashed_line_(color_index, thickness, num_pts,
list_pts_x, list_pts_y)
float list_pts_x[], list_pts_y[], *thickness;
int *color_index, *num_pts;
{
    Xgl_2d_ctx ctx;
    Xgl_color color;
    Xgl_pt_list line_pts;
    int i;

    /* allocate memory for the coordinates to be processed */

    if (!(line_pts.pts.f2d = (Xgl_pt_f2d
*)malloc(*num_pts*sizeof(Xgl_pt_f2d)))) {
        printf("memory allocation request in gal_2d_long_dashed
line failed\n"); exit(1); }

    line_pts.pt_type = XGL_PT_F2D;
    line_pts.num_pts = *num_pts;
    line_pts.bbox = 0;

    for ( i=0; i<*num_pts; i++) {
        line_pts.pts.f2d[i].x = list_pts_x[i];
        line_pts.pts.f2d[i].y = list_pts_y[i]; }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
*color_index; } else color.index = *color_index;

    ctx = get_2d_ctx();

    xgl_object_set(*ctx,
                   XGL_CTX_LINE_WIDTH_SCALE_FACTOR, *thickness,
                   XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNEDED,
                   XGL_CTX_LINE_PATTERN, xgl_lpat_long_dashed,
                   XGL_CTX_LINE_COLOR, &color, NULL);

    xgl_multipolyline(*ctx, NULL, 1, line_pts);

    free(line_pts.pts.f2d); }

```

```

/*****
 * gal_2d_cross_marker_() - a routine to draw a given marker type. The
 * user supplies the color, number of markers and their locations
 *
 * routines called - malloc, xgl_object_set, xgl_multimarker
 *****/

void gal_2d_cross_marker_(color_index, num_pts, list_pts_x, list_pts_y)
float list_pts_x[], list_pts_y[];
int *color_index, *num_pts;
{
    Xgl_2d_ctx      *ctx;
    Xgl_color        color;
    Xgl_pt_list      marker_pts;
    int              i;

    /* allocate memory for the coordinates to be processed */

    if (!(marker_pts.pts.f2d = (Xgl_pt_f2d
        *)malloc(*num_pts*sizeof(Xgl_pt_f2d)))) {
        printf("memory allocation request in
            gal_2d_cross_marker failed\n"); exit(1); }

    marker_pts.pt_type = XGL_PT_F2D;
    marker_pts.num_pts = *num_pts;
    marker_pts.bbox = 0;

    for ( i=0; i<*num_pts; i++) {

        marker_pts.pts.f2d[i].x = list_pts_x[i];
        marker_pts.pts.f2d[i].y = list_pts_y[i];

    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
        *color_index; } else color.index = *color_index;

    ctx = get_2d_ctx();

    /* the marker size is defaulted to 10 pixels in size */
    xgl_object_set(*ctx,
        XGL_CTX_MARKER_COLOR, &color,
        XGL_CTX_MARKER_SCALE_FACTOR, 10.0,
        XGL_CTX_MARKER_DESCRIPTION, xgl_marker_cross,
        NULL);

    xgl_multimarker(*ctx, marker_pts);

    free(marker_pts.pts.f2d); }

/*****
 * gal_2d_plus_marker_() - a routine to draw a given marker type. The
 * user supplies the color, number of markers and their locations
 *
 * routines called - malloc, xgl_object_set, xgl_multimarker
 *****/

void gal_2d_plus_marker_(color_index, num_pts, list_pts_x, list_pts_y)
float list_pts_x[], list_pts_y[];
int *color_index, *num_pts;
{
    Xgl_2d_ctx      *ctx;
    Xgl_color        color;

```

```

Xgl_pt_list      marker_pts;
int              i;

/* allocate memory for the coordinates to be processed */

if (!(marker_pts.pts.f2d = (Xgl_pt_f2d
*)malloc(*num_pts*sizeof(Xgl_pt_f2d)))) {
    printf("memory allocation request in gal_2d_plus_marker
    failed\n"); exit(1); }

marker_pts.pt_type = XGL_PT_F2D;
marker_pts.num_pts = *num_pts;
marker_pts.bbox = 0;

for ( i=0; i<*num_pts; i++) {

    marker_pts.pts.f2d[i].x = list_pts_x[i];
    marker_pts.pts.f2d[i].y = list_pts_y[i];

}

if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
*color_index; } else color.index = *color_index;

ctx = get_2d_ctx();

/* the marker size is defaulted to 10 pixels in size */
xgl_object_set(*ctx,
               XGL_CTX_MARKER_COLOR, &color,
               XGL_CTX_MARKER_SCALE_FACTOR, 10.0,
               XGL_CTX_MARKER_DESCRIPTION, xgl_marker_plus,
               NULL);

xgl_multimarker(*ctx, marker_pts);

free(marker_pts.pts.f2d); }

/*****
 * gal_2d_asterisk_marker_() - a routine to draw a given marker type.
 * The user supplies the color, number of markers and their locations
 *
 * routines called - malloc, xgl_object_set, xgl_multimarker
 *****/

void gal_2d_asterisk_marker_(color_index, num_pts, list_pts_x,
list_pts_y)
float list_pts_x[], list_pts_y[];
int    *color_index, *num_pts;
{
    Xgl_2d_ctx      *ctx;
    Xgl_color       color;
    Xgl_pt_list     marker_pts;
    int             i;

    /* allocate memory for the coordinates to be processed */

    if (!( marker_pts.pts.f2d = (Xgl_pt_f2d
*)malloc(*num_pts*sizeof(Xgl_pt_f2d)))) {
        printf("memory allocation request in
        gal_2d_asterisk_marker failed\n"); exit(1); }

    marker_pts.pt_type = XGL_PT_F2D;
    marker_pts.num_pts = *num_pts;
    marker_pts.bbox = 0;

```

```

    for ( i=0; i<*num_pts; i++) {

        marker_pts.pts.f2d[i].x = list_pts_x[i];
        marker_pts.pts.f2d[i].y = list_pts_y[i];

    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
    *color_index; } else color.index = *color_index;

    ctx = get_2d_ctx();

    /* the marker size is defaulted to 10 pixels in size */
    xgl_object_set(*ctx,
        XGL_CTX_MARKER_COLOR, &color,
        XGL_CTX_MARKER_SCALE_FACTOR, 10.0,
        XGL_CTX_MARKER_DESCRIPTION,
        xgl_marker_asterisk, NULL);

    xgl_multimarker(*ctx, marker_pts);

    free(marker_pts.pts.f2d); }

/*****
 * gal_2d_square_marker_() - a routine to draw a given marker type. The
 * user supplies the color, number of markers and their locations
 *
 * routines called - malloc, xgl_object_set, xgl_multimarker
 *****/

void gal_2d_square_marker_(color_index, num_pts, list_pts_x,
list_pts_y)
float list_pts_x[], list_pts_y[];
int *color_index, *num_pts;
{
    Xgl_2d_ctx      *ctx;
    Xgl_color        color;
    Xgl_pt_list      marker_pts;
    int              i;

    /* allocate memory for the coordinates to be processed */

    if (!( marker_pts.pts.f2d = (Xgl_pt_f2d
    *)malloc(*num_pts*sizeof(Xgl_pt_f2d)))) {
        printf("memory allocation request in
        gal_2d_square_marker failed\n"); exit(1); }

    marker_pts.pt_type = XGL_PT_F2D;
    marker_pts.num_pts = *num_pts;
    marker_pts.bbox = 0;

    for ( i=0; i<*num_pts; i++) {

        marker_pts.pts.f2d[i].x = list_pts_x[i];
        marker_pts.pts.f2d[i].y = list_pts_y[i];

    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
    *color_index; } else color.index = *color_index;

    ctx = get_2d_ctx();

    /* the marker size is defaulted to 10 pixels in size */

```

```

    xgl_object_set(*ctx,
                  XGL_CTX_MARKER_COLOR, &color,
                  XGL_CTX_MARKER_SCALE_FACTOR, 10.0,
                  XGL_CTX_MARKER_DESCRIPTION, xgl_marker_square,
                  NULL);

    xgl_multimarker(*ctx, marker_pts);

    free(marker_pts.pts.f2d); }

/*****
 * gal_2d_circle_marker_() - a routine to draw a given marker type. The
 * user supplies the color, number of markers and their locations
 *
 * routines called - malloc, xgl_object_set, xgl_multimarker
 *****/

void gal_2d_circle_marker_(color_index, num_pts, list_pts_x,
list_pts_y)
    float list_pts_x[], list_pts_y[];
    int *color_index, *num_pts;
{
    Xgl_2d_ctx *ctx; Xgl_color color; Xgl_pt_list
    marker_pts; int i;

    /* allocate memory for the coordinates to be processed */

    if (!(marker_pts.pts.f2d = (Xgl_pt_f2d
*)malloc(*num_pts*sizeof(Xgl_pt_f2d)))) {
        printf("memory allocation request in
gal_2d_circle_marker failed\n"); exit(1); }

    marker_pts.pt_type = XGL_PT_F2D; marker_pts.num_pts = *num_pts;
    marker_pts.bbox = 0;

    for ( i=0; i<*num_pts; i++) {

        marker_pts.pts.f2d[i].x = list_pts_x[i];
        marker_pts.pts.f2d[i].y = list_pts_y[i];

    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
*color_index; } else color.index = *color_index;

    ctx = get_2d_ctx();

    /* the marker size is defaulted to 10 pixels in size */
    xgl_object_set(*ctx,
                  XGL_CTX_MARKER_COLOR, &color,
                  XGL_CTX_MARKER_SCALE_FACTOR, 10.0,
                  XGL_CTX_MARKER_DESCRIPTION, xgl_marker_circle,
                  NULL);

    xgl_multimarker(*ctx, marker_pts);

    free(marker_pts.pts.f2d); }

/*****
 * gal_2d_axes_() - draws x and y axes given maxima, minima and increments
 * procedures called - gal_2d_solid_line_()
 *****/
void gal_2d_axes_(color_index, xmin, xmax, ymin,
ymax, x_interval, y_interval)
float *xmin, *xmax, *ymin, *ymax, *x_interval, *y_interval;

```

```

int      *color_index;
{
extern float      axis_thickness, zero;
extern int axis_num_pts;
float list_pts_x[2], list_pts_y[2], *xmin_new, *ymin_new;
float tic_x, tic_y, tic_start, tic_fin;
int      ntics, i;

    /* If the extremities of the axes cross the origin, make the
axis lie on the origin */
    if ((*ymax > 0.) && (*ymin < 0.)) ymin_new = &zero
        else ymin_new = ymin;
    if ((*xmax > 0.) && (*xmin < 0.)) xmin_new = &zero;
        else xmin_new = xmin;

    /* Draw the x-axis */
    list_pts_x[0] = *xmin;
    list_pts_x[1] = *xmax;
    list_pts_y[0] = *ymin_new;
    list_pts_y[1] = *ymin_new;
    gal_2d_solid_line_(color_index, &axis_thickness, &axis_num_pts,
list_pts_x, list_pts_y);

    /* Draw the y-axis */
    list_pts_x[0] = *xmin_new;
    list_pts_x[1] = *xmin_new;
    list_pts_y[0] = *ymin;
    list_pts_y[1] = *ymax;
    gal_2d_solid_line_(color_index, &axis_thickness, &axis_num_pts,
list_pts_x, list_pts_y);

    /* calculate the size of the tics (1/30 the axis length) */
    tic_x = (*ymax - *ymin)/30; tic_y = (*xmax - *xmin)/30;

    /* calculate the number of tics */ ntics = (int)
(((*xmax-*xmin)/ *x_interval);

    /* calculate the y coordinates of the tics */ tic_start =
*ymin_new - tic_x; tic_fin = *ymin_new + tic_x;

    /* draw the n+1 tics */
    for (i = 0; i <= (ntics+1); i++) {
        list_pts_x[1] = list_pts_x[0] = *xmin +
i*( *x_interval);
        list_pts_y[0] = tic_start;
        list_pts_y[1] = tic_fin;
        gal_2d_solid_line_(color_index, &axis_thickness,
&axis_num_pts, list_pts_x, list_pts_y); }

    /* calculate the number of tics */
    ntics = (int) ((*ymax-*ymin)/ *y_interval);

    /* calculate the y coordinates of the tics */
    tic_start = *xmin_new - tic_y;
    tic_fin = *xmin_new + tic_y;

    /* draw the n+1 tics */ for (i = 0; i <= (ntics+1); i++) {
        list_pts_y[1] = list_pts_y[0] = *ymin + i*( *y_interval);
        list_pts_x[0] = tic_start;
        list_pts_x[1] = tic_fin;
        gal_2d_solid_line_(color_index, &axis_thickness,
&axis_num_pts, list_pts_x, list_pts_y); }
}

/*****
* gal_2d_text_annotate() - draws a string of text to the screen. The

```

```

* user supplies the string, font size, font (i.e. the name of the XGL
* font file), the spacing between letters, the orientation and the
* color routines called - malloc, xgl_object_set, xgl_stroke_text;
*****/

void gal_2d_text_annotate(color_index, string, font_size, font,
font_spacing, x_pos, y_pos, x_vector, y_vector, str_len, font_len)
float *font_size, *font_spacing, *x_pos, *y_pos, *x_vector, *y_vector;
int *color_index, str_len, font_len;
char *string, *font ;
{
    Xgl_sfnt sfnt;
    Xgl_obj_desc obj_desc;
    Xgl_pt_f2d text_pos;
    Xgl_pt_f2d up_vector;
    Xgl_color sf_color;
    Xgl_2d_ctx *ctx;
    char *font_copy;
    Xgl_gcache *gcache;

    ctx = get_2d_ctx();
    gcache = get_gcach();

    if (!(font_copy = malloc(80))) {
        printf("memory allocation request in gal_2d_text_annotate
        failed\n");
        exit(1); }

    (void) strcpy(font_copy, font, font_len);

    /* append the '.phont' font file extension to the font
    * name. */

    (void) strcat(font_copy, ".phont");

    if (!(obj_desc.sfnt_name = (char *)malloc(80))) {
        printf("memory allocation request in gal_2d_text_annotate
        failed\n");
        exit(1); }

    strcpy(obj_desc.sfnt_name, font_copy) ;

    free(font_copy);

    sfnt = xgl_object_create (*ctx, XGL_SFNT, &obj_desc, NULL);

    free(obj_desc.sfnt_name);

    up_vector.x = *x_vector; up_vector.y = *y_vector;

    if (get_dbuf_on()) { sf_color.index = *color_index*COLOR_SIZE +
    *color_index; } else sf_color.index = *color_index;

    xgl_object_set(*ctx,
        XGL_CTX_SFNT, sfnt,
        XGL_CTX_SFNT_CHAR_HEIGHT, *font_size,
        XGL_CTX_SFNT_CHAR_UP_VECTOR, up_vector,
        XGL_CTX_SFNT_CHAR_SPACING, *font_spacing,
        XGL_CTX_SFNT_TEXT_COLOR, &sf_color,
        XGL_CTX_LINE_COLOR, &sf_color, NULL);

    /* invoke the display of the gcache */
    text_pos.x = *x_pos, text_pos.y = *y_pos;
    xgl_gcach_stroke_text (*gcache, *ctx, string, &text_pos,
    NULL);

```



```

    xgl_context_display_gcache(*ctx, *gcache, FALSE, TRUE);
    xgl_context_post(*ctx, TRUE);
}

/*****
 * gal_2d_real_annotate() - draws a real number to the screen. The user
 * supplies the string, the number of significant figures, the number
 * of decimal places, font size, font (i.e. the name of the XGL font
 * file), the spacing between letters, the orientation and the color
 *
 * routines called - malloc, xgl_object_set, xgl_stroke_text;
 *****/

void gal_2d_real_annotate(color_index, real, field_width,
mantissa, font_size, font, font_spacing, x_pos, y_pos, x_vector,
y_vector, font_len)
float *font_size, *font_spacing, *x_pos, *y_pos, *x_vector, *y_vector, *real;
int *color_index, font_len, *field_width, *mantissa;
char *font ;
{
    char string[20], format[20];
    Xgl_sfont sfont;
    Xgl_obj_desc obj_desc;
    Xgl_pt_f2d text_pos;
    Xgl_pt_f2d up_vector;
    Xgl_color sf_color;
    Xgl_2d_ctx *ctx;
    char *font_copy;
    Xgl_gcache *gcache;

    ctx = get_2d_ctx(); gcache = get_gcache();

    if (!(font_copy = malloc(80))) {
        printf("memory allocation request in gal_2d_real_annotate
failed\n");
        exit(1); }

    (void) strcpy(font_copy, font, font_len);

    /* append the '.phont' font file extension to the font
    * name. */

    (void) strcat(font_copy, ".phont");

    if (!(obj_desc.sfont_name = (char *)malloc(80))) {
        printf("memory allocation request in gal_2d_real_annotate
failed\n");
        exit(1); }

    strcpy(obj_desc.sfont_name, font_copy) ;

    free(font_copy);

    sfont = xgl_object_create (*ctx, XGL_SFONTS, &obj_desc, NULL);

    free(obj_desc.sfont_name);

    sprintf(format, "%%.d.%df", *field_width, *mantissa);
    sprintf(string, format, *real);

    up_vector.x = *x_vector; up_vector.y = *y_vector;

    if (get_dbuf_on()) { sf_color.index = *color_index*COLOR_SIZE +
    *color_index; } else sf_color.index = *color_index;

    xgl_object_set(*ctx,

```

```

        XGL_CTX_SFONTS, sfont,
        XGL_CTX_SFONTS_CHAR_HEIGHT, *font_size,
        XGL_CTX_SFONTS_CHAR_UP_VECTOR, up_vector,
        XGL_CTX_SFONTS_CHAR_SPACING, *font_spacing,
        XGL_CTX_SFONTS_TEXT_COLOR, &sf_color,
        XGL_CTX_LINE_COLOR, &sf_color, NULL);

    text_pos.x = *x_pos, text_pos.y = *y_pos;

    xgl_gcachestroke_text (*gcachest, *ctx, string, &text_pos,
        NULL);
    xgl_context_display_gcachest(*ctx, *gcachest, FALSE, TRUE);
    xgl_context_post(*ctx, TRUE);
}

/*****
 * gal_2d_unfilled_multicircle() - a primitive drawing routine to draw
 * a series of identically colored circles.
 *
 * routines called - get_2d_ctx, xgl_object_set, xgl_multi_circle
 *****/

void gal_2d_unfilled_multicircle(color_index, num_circles, rad, x, y)
int      *color_index, *num_circles;
float    x[], y[], rad[];
{
    Xgl_2d_ctx      *ctx;
    Xgl_color        color;
    Xgl_circle_list  circle_list;
    Xgl_pt_f2d        center;
    int              i;

    /* allocate memory for the coordinates to be processed */

    if (!(circle_list.circles.f2d = (Xgl_circle_f2d
        *)malloc(*num_circles*sizeof(Xgl_circle_f2d)))) {
        printf("memory allocation request in
            gal_2d_unfilled_multicircle failed\n"); exit(1); }

    /*
     * get a smooth circle */

    ctx = get_2d_ctx();

    xgl_object_set(*ctx, XGL_CTX_CURVE_APPROX, XGL_CURVE_METRIC_VDC,
        XGL_CTX_CURVE_APPROX_VALUE, 1.0, NULL);

    /* draw the circles */
    circle_list.type = XGL_MULTICIRCLE_F2D;
    circle_list.num_circles = *num_circles;
    circle_list.bbox = 0;

    for (i=0; i<*num_circles; i++) {
        circle_list.circles.f2d[i].center.flag = 0;
        circle_list.circles.f2d[i].center.x = x[i];
        circle_list.circles.f2d[i].center.y = y[i];
        circle_list.circles.f2d[i].radius = rad[i]; }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
        *color_index; } else color.index = *color_index;

    xgl_object_set (*ctx,
        XGL_CTX_SURF_FRONT_FILL_STYLE, XGL_SURF_FILL_HOLLOW, XGL_CTX_SURF_FRONT_COLOR, &color

```

```

r, NULL);

    xgl_multicircle(*ctx, &circle_list);
    free(circle_list.circles.f2d);
}

/*****
 * gal_2d_unfilled_multicircle_() - a primitive drawing routine to draw
 * a series of identically colored circles.
 *
 * routines called - get_2d_ctx, xgl_object_set, xgl_multi_circle
 *****/

void gal_2d_filled_multicircle_(color_index,num_circs, rad,  x, y)
int      *color_index, *num_circs;
float    x[], y[],rad[];
{
    Xgl_2d_ctx      *ctx;
    Xgl_color       color;
    Xgl_circle_list circle_list;
    Xgl_pt_f2d      center;
    int             i;

    /* allocate memory for the coordinates to be processed */

    if (!(circle_list.circles.f2d = (Xgl_circle_f2d
    *)malloc(*num_circs*sizeof(Xgl_circle_f2d))) {
        printf("memory allocation request in
        gal_2d_filled_polygon failed\n"); exit(1); }

    /*
     * get a smooth circle */

    ctx = get_2d_ctx();

    xgl_object_set(*ctx, XGL_CTX_CURVE_APPROX, XGL_CURVE_METRIC_VDC,
        XGL_CTX_CURVE_APPROX_VALUE, 1.0, NULL);

    /* draw the circles */
    circle_list.type = XGL_MULTICIRCLE_F2D;
    circle_list.num_circles = *num_circs;
    circle_list.bbox = 0;

    for ( i=0; i<*num_circs; i++) {
        circle_list.circles.f2d[i].center.flag = 0;
        circle_list.circles.f2d[i].center.x = x[i];
        circle_list.circles.f2d[i].center.y = y[i];
        circle_list.circles.f2d[i].radius = rad[i]; }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
        *color_index; } else color.index = *color_index;

    xgl_object_set (*ctx, XGL_CTX_SURF_FRONT_COLOR, &color, NULL);

    xgl_multicircle(*ctx, &circle_list);
    free(circle_list.circles.f2d);
}

/*****
 * gal_2d_unfilled_multirectangle_() - a primitive drawing routine to
 * draw a series of identically colored unfilled rectangles
 *

```

```

* routines called - get_2d_ctx, xgl_object_set, xgl_multirectangle
*****/

void gal_2d_unfilled_multirectangle_(color_index, num_rects, x, y)
int      *color_index, *num_rects;
float    x[], y[];
{
    Xgl_2d_ctx      *ctx;
    Xgl_color        color;
    Xgl_rect_list    rect_list;
    Xgl_pt_f2d        center;
    Xgl_rect_f2d      rects;
    int              rect_pts, i, rect_inc;

    /* allocate memory for the coordinates to be processed */

    if (!(rect_list.rects.f2d = (Xgl_rect_f2d
*)malloc(*num_rects*sizeof(Xgl_rect_f2d)))) {
        printf("memory allocation request in
        gal_2d_unfilled_multi_rectangle failed\n"); exit(1); }

    /*
    * get a smooth circle */

    ctx = get_2d_ctx();

    /* draw the rectangles*/
    rect_list.rect_type = XGL_MULTIRECT_F2D;
    rect_list.num_rects = *num_rects;
    rect_list.bbox = 0; rect_pts = 2*(*num_rects);

    for ( i=0; i<rect_pts; i++) { rect_inc = 2*i;
    rect_list.rects.f2d[i].corner_min.flag = 0;
    rect_list.rects.f2d[i].corner_min.x = x[rect_inc];
    rect_list.rects.f2d[i].corner_min.y = y[rect_inc];
    rect_list.rects.f2d[i].corner_max.x = x[++rect_inc];
    rect_list.rects.f2d[i].corner_max.y = y[rect_inc]; }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
    *color_index; } else color.index = *color_index;

    xgl_object_set (*ctx,
    XGL_CTX_SURF_FRONT_FILL_STYLE,XGL_SURF_FILL_HOLLOW,XGL_CTX_SURF_FRONT_COLOR, &color,
    NULL);

    xgl_multirectangle(*ctx, &rect_list);
    free(rect_list.rects.f2d);
}

/*****
* gal_2d_filled_multirectangle_() - a primitive drawing routine to
* draw a series of identically colored filled rectangles
*
* routines called - get_2d_ctx, xgl_object_set, xgl_multirectangle
*****/

void gal_2d_filled_multirectangle_(color_index, num_rects, x, y)
int      *color_index, *num_rects;
float    x[], y[];
{

```

```

Xgl_2d_ctx      *ctx;
Xgl_color       color;
Xgl_rect_list   rect_list;
Xgl_pt_f2d      center;
Xgl_rect_f2d    rects;
int             rect_pts, i, rect_inc;

    /* allocate memory for the coordinates to be processed */

    if (!(rect_list.rects.f2d = (Xgl_rect_f2d
*)malloc(*num_rects*sizeof(Xgl_rect_f2d)))) {
        printf("memory allocation request in
gal_2d_unfilled_multi_rectangle failed\n"); exit(1); }

/*
 * get a smooth circle */

    ctx = get_2d_ctx();

/* draw the rectangles*/
    rect_list.rect_type = XGL_MULTIRECT_F2D;
    rect_list.num_rects = *num_rects;
    rect_list.bbox = 0; rect_pts = 2*(*num_rects);

    for ( i=0; i<rect_pts; i++) {
        rect_inc = 2*i;
        rect_list.rects.f2d[i].corner_min.flag = 0;
        rect_list.rects.f2d[i].corner_min.x = x[rect_inc];
        rect_list.rects.f2d[i].corner_min.y = y[rect_inc];
        rect_list.rects.f2d[i].corner_max.x = x[++rect_inc];
        rect_list.rects.f2d[i].corner_max.y = y[rect_inc]; }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
        *color_index; } else color.index = *color_index;

    xgl_object_set (*ctx, XGL_CTX_SURF_FRONT_COLOR, &color, NULL);

    xgl_multirectangle(*ctx, &rect_list);
    free(rect_list.rects.f2d);
}

```

```

/*****
 * This contour generator is based on Bruce Giles' which appeared in the
 * June 1992 issue of Dr. Dobbs Journal, #189, Volume 17, Issue 6, pp. 44-46.
 *****/

#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <values.h>

#if defined (NEVER)
#include <ieeefp.h>
#else
#define NaN      0xFFFFFFFF
#define isnanf(x) ((x) == NaN)
#endif

typedef unsigned short ushort;
typedef unsigned char uchar;

#define DEFAULT_LEVELS 16

/*Mnemonics for contour line drawings*/
#define EAST      0
#define NORTH     1
#define WEST      2
#define SOUTH     3

/* Mnemonics for relative data point positions */
#define SAME      0
#define NEXT      1
#define OPPOSITE  2
#define ADJACENT  3

/* Bit-mapped information in 'map' field */
#define EW_MAP    0x01
#define NS_MAP    0x02

void Polyline();
#define MXY_to_L(g,x,y) ((ushort) (y) * (g)->dim_x + (ushort) (x) + 1)
#define XY_to_L(g,x,y)  ((ushort) (y) * (g)->dim_x + (ushort) (x) )

typedef struct
{
    float x,y;
} LIST;

typedef struct
{
    short   dim_x; /*dimensions of grid array */
    short   dim_y;
    float   max_value;
    float   min_value;
    float   mean;
    float   std;
    short   contour_mode; /* control variable */
    float   first_level; /* first (and subsequent) contour level*/
    float   step;
    char    format[2]; /* format of contour levels */
    float   *data; /* pointer to grid data */
    char    *map; /* pointer to 'in-use' map */
    LIST    *list; /* used by 'Polyline()' */
    ushort  count;
} GRID;

```

```

typedef struct
{
    short    x,y;
    uchar    bearing;
} POINT;

void Contour();

int scaleData();
void startLine();

void startEdge();
void startInterior();
void drawLine();

void markInUse();
uchar faceInUse();
float getDataPoint();

void gal_solid_contour_();
void initPoint();
void lastPoint();
uchar savePoint();

/*****

/*****
* Ooops! I know the use of global variables is a no-no, but I decided that this
* was the quickest way of implementing a choice of line types. The XGL drawing
* procedures were patched onto the contour generator by Bruce Giles
*****/

#include "contour.h"

static float *data_set;
static float xmin_plot, ymin_plot, xmax_plot, ymax_plot;

static int contour_color;
static float thick;
static short line_switch;

/*****
* gal_solid_contour_() - draws a plot of solid contours, given the datafields,
* dimensions of the uniformly spaced data field, the extremas of the dimensions,
* the color and the thickness of the lines.
*
*****/
void
gal_solid_contour_(color, thickness, x_data, y_data, interval, xmin, xmax, ymin, ymax, data)
float data[], *xmin, *ymin, *xmax, *ymax, *thickness;
int    *x_data, *y_data, *color;
float  *interval;
{
    extern float *data_set, ymax_plot, ymin_plot, xmax_plot, ymax_plot;
    double d_interval;

    /* this flag determines whether the lines are to be solid or dotted */
    extern short line_switch;

    xmin_plot = *xmin;
    ymin_plot = *ymin;
    xmax_plot = *xmax;
    ymax_plot = *ymax;

```

```

        contour_color = *color;
        thick = *thickness;
        line_switch = 0;

        data_set = data;
        d_interval = (double) *interval;

        Contour(data, *x_data, *y_data, d_interval);
    }

/*****
 * gal_dotted_contour_() - draws dotted-line contours, given the datafields,
 * dimensions of the uniformly spaced data field, the extremas of the dimensions
 * the color and the thickness of the lines.
 *
 *****/

void
gal_dotted_contour_(data, x_data, y_data, interval, xmin, xmax, ymin, ymax,
color, thickness)
float data[], *xmin, *ymin, *xmax, *ymax, *thickness;
int      *x_data, *y_data, *color;
float    *interval;
{
    extern float *data_set, ymax_plot, ymin_plot, xmax_plot, ymax_plot;
    double d_interval;
    extern short line_switch;

    xmin_plot = *xmin;
    ymin_plot = *ymin;
    xmax_plot = *xmax;
    ymax_plot = *ymax;

    contour_color = *color;
    thick = *thickness;
    line_switch = 1;

    data_set = data;
    d_interval = (double) *interval;

    Contour(data, *x_data, *y_data, d_interval);
}

/* getDataPoint -- Return the value of the data point in the specified corner
 * of the specified cell (the 'point' parameter contains the address of the
 * top-left corner of the cell)*/

float
getDataPoint (grid, point, corner)
GRID      *grid;
POINT     *point;
uchar     corner;
{
    ushort dx, dy;
    ushort offset;
    float datapoint;

    switch ((point->bearing + corner) % 4)
    {
        case SAME : dx = 0; dy = 0; break;
        case NEXT : dx = 0; dy = 1; break;
        case OPPOSITE : dx = 1; dy = 1; break;
        case ADJACENT : dx = 1; dy = 0; break;
    }
}

```



```

offset = XY_to_L (grid, point->x + dx, point->y + dy);
if ((short)(point->x + dx) >= grid->dim_x ||
    (short)(point->y + dy) >= grid->dim_y ||
    (short)(point->x + dx) < 0 ||
    (short)(point->y + dy) < 0)
{
    return NaN;
}
else
{
    /* why doesn't grid->data[offset work ??? */
    return data_set[offset];
}
}

/* Polyline - draws the lines by calling GAL routines gal_2d_solid_line_
 * and gal_2d_dotted_line_()
 */

void
Polyline(n, list)
int n;
LIST *list;
{
    float *x, *y, thickness;
    int color, num_pts;
    extern int contour_color;
    extern float thick;

    x = (float *)malloc(n*sizeof(float));
    y = (float *)malloc(n*sizeof(float));

    if (n < 2)
        return;

    num_pts = n;
    for (n = 0; n < num_pts; n++)
    {
        x[n] = xmax_plot*(list -> x) + xmin_plot;
        y[n] = ymax_plot*(1 - (list -> y)) + ymin_plot;
        list++;
    }

    if (!line_switch)
        gal_2d_solid_line_(&contour_color, &thick, &num_pts,
x, y);
    else
        gal_2d_dotted_line_(&contour_color, &thick, &num_pts,
x, y);

    free(x), free(y);
}

/*****

#include "contour.h"

/*****
 * This contour generator is by Bruce Giles and appeared in the June 1992 issue
 * of Dr. Dobbs Journal, #189, Volume 17, Issue 6, pp. 44-46.
 *****/
void

```

```

Contour (data, dim_x, dim_y, inc)
float  *data;
int    dim_x;
int    dim_y;
double inc;
{
    GRID  grid;

    grid.data = data;
    grid.dim_x = dim_x;
    grid.dim_y = dim_y;

    /* Allocate buffers used to contain contour information */
    if ((grid.map = malloc ((dim_x + 1) * dim_y)) == NULL)
    {
        printf("contour generator unable to allocate memory\n");
        free ((char *) grid.map);
        exit(1);
    }
    grid.list = (LIST *) malloc (2*dim_x * dim_y*sizeof(LIST));
    if (grid.list == (LIST *) NULL)
    {
        printf("contour generator unable to allocate memory\n");
        free ((char *) grid.map);
        exit(1);
    }
    /* Check for uniformity, then generate field */
    if (scaleData (&grid, inc, data))

        startLine(&grid);

    /* Release data structures */
    free( (char *) grid.map);
    free( (char *) grid.list);
}

int
scaleData(grid, inc, data)
GRID *grid;
double inc;
float *data;
{
    ushort i;
    float  step, level;
    float  sum, sum2, count;
    float  p, *v, r;
    char   *s;
    short  n1, n2;
    int    first, n;
    long   x;

    sum = sum2 = count = 0.0;

    first = 1;
    s = grid->map;
    v = data + grid->dim_x * grid->dim_y;

    /* determine the max, min grid values and other statistical stuff */
    for (i = 0; i < grid->dim_x * grid->dim_y; i++, data++, v++, s++)
    {
        r = *data;
        sum += r;
        sum2 += r*r;
        count += 1.0;

        if (first)

```

```

        {
            grid->max_value = grid->min_value = r;
            first = 0;
        }
        else if (grid->max_value < r)
            grid->max_value = r;
        else if (grid->min_value > r)
            grid->min_value = r;
    }

    grid->mean = sum/count;

    /* check to ensure no uniformity */
    if (grid->min_value == grid->max_value)
        return 0;

    grid->std = sqrt ((sum2 - sum*sum/count)/(count - 1.0));
    if (inc > 0.0)
    {
        /* use specified increment */
        step = inc;
        n = (int) (grid->max_value-grid->min_value)/step + 1;

        while (n > 40)
        {
            step *= 2.0;
            n = (int) (grid->max_value - grid->min_value)/ step + 1;
        }
    }
    else
    {
        /* choose a reasonable number of levels */
        n = (inc == 0.0) ? DEFAULT_LEVELS : (short) fabs(inc);

        step = 4.0 * grid->std/(float) n;
        p = pow(10.0, floor(log10 ((double) step)));
        step = p*floor ((step + p /2.0) /p);
    }

    n1 = (int) floor (log10 (fabs (grid->max_value)));
    n2 = -((int) floor (log10 (step)));

    if (grid->max_value*grid->min_value < 0.0)
        level = step*floor(grid->mean/step);
    else
        level = step*floor(grid->min_value/step);
    level -= step*floor((float) (n-1)/ 2);

    /* Back up to include additional levels, if necessary */
    while (level - step > grid->min_value)
        level -= step;

    grid->first_level = level;
    grid->step = step;
    return 1;
}

/* startLine() - locate first point of contour lines by checking edges of
   gridded data set, then interior points, for each contour level. */
static void
startLine(grid)
GRID *grid;
{
    ushort idx, i, edge;
    double level;

```

```

for (idx = 0, level = grid->first_level; level < grid->max_value;
    level += grid->step, idx++)
{
    /* clear flags */
    grid->contour_mode = (level >= grid->mean);
    memset (grid->map, 0, grid->dim_x * grid->dim_y);

    /* Check edges */
    for (edge = 0; edge < 4; edge++)
        startEdge(grid, level, edge);
    /* Check for interior points */
    startInterior(grid, level);
}

}

/* startEdge -- For a specified contour level and edge of gridded data set,
   check for (properly directed) contour line */
static void
startEdge(grid, level, bearing)
GRID      *grid;
float     level;
uchar     bearing;
{
    POINT point1, point2;
    float last, next;
    short i, ds;

    switch (point1.bearing = bearing)
    {
        case EAST:
            point1.x = 0;
            point1.y = 0;
            ds = 1;
            break;
        case NORTH:
            point1.x = 0;
            point1.y = grid->dim_x - 2;
            ds = 1;
            break;
        case WEST:
            point1.x = grid->dim_x - 2;
            point1.y = grid->dim_y - 2;
            ds = -1;
            break;
        case SOUTH:
            point1.x = grid->dim_x - 2;
            point1.y = 0;
            ds = -1;
            break;
    }
    switch(point1.bearing)
    {
        /* Find first point with valid data */
        case EAST:
        case WEST:
            next = getDataPoint (grid, &point1, SAME);
            memcpy ((char *) &point2, (char *) &point1,
                    sizeof (POINT));
            point2.x = -ds;

            for (i = 1; i < grid->dim_y; i++,
                point1.y = point2.y += ds)
            {
                last = next;

```

```

        next = getDataPoint (grid, &point1, NEXT);
    if (last >= level && level > next)
    {
        drawLine (grid, &point1, level);
        memcpy ((char *) &point1, (char *) &point2,
            sizeof (POINT));
        point1.x = point2.x + ds;
    }
}
break;
/* Find the first point with valid data */
case SOUTH:
case NORTH:
    next = getDataPoint (grid, &point1, SAME);
    memcpy ((char *) &point2, (char *) &point1,
        sizeof(POINT));
    point2.y += ds;

    for (i = 1; i < grid->dim_x; i++,
        point1.x = point2.x +=ds)
    {
        last = next;
        next = getDataPoint (grid, &point1, NEXT);
        if (last >= level && level > next)
        {
            drawLine (grid, &point1, level);
            memcpy ((char *) &point1, (char *) &point2,
                sizeof (POINT));
            point1.y = point2.y - ds;
        }
    }
break;
}
}

```

```

static void
startInterior(grid, level)
GRID    *grid;
float   level;
{
    POINT point;
    ushort x,y;
    float next, last;
    for (x = 1; x < grid->dim_x - 1; x++)
    {
        point.x = x;
        point.y = 0;
        point.bearing = EAST;
        next = getDataPoint (grid, &point, SAME);
        for (y = point.y; y < grid->dim_y; y++, point.y++)
        {
            last = next;
            next = getDataPoint (grid, &point, NEXT);
            if (last >= level && level > next)
            {
                if (!faceInUse (grid, &point, WEST))
                {
                    drawLine (grid, &point, level);
                    point.x = x;
                    point.y = y;
                    point.bearing = EAST;
                }
            }
        }
    }
}

```

```

    }
}

/* drawLine -- Given an initial contour point by either 'startEdge' or
 * 'startInterior', follow the contour until it encounters an edge or previously
 * contoured cell */

static void
drawLine(grid, point, level)
GRID    *grid;
POINT   *point;
float    level;
{
    uchar exit_bearing;
    uchar adj, opp;
    float fadj, fopp;

    initPoint (grid);

    for (;;)
    {
        /* add current point to vector list. If either of the points
         * is missing, return immediately (open contour) */
        if (!savePoint (grid, point, level))
        {
            lastPoint (grid);
            return;
        }

        /* Has the face of this cell been marked for use? If
         so, then this is a closed contour */
        if (faceInUse (grid, point, WEST))
        {
            lastPoint (grid);
            return;
        }

        /* Examine adjacent and opposite corners of the cell;
         * determine appropriate action */
        markInUse (grid, point, WEST);

        fadj = getDataPoint (grid, point, ADJACENT);
        fopp = getDataPoint (grid, point, OPPOSITE);

        /* If either point is missing, return immediately (open contour) */
        if (isnanf (fadj) || isnanf (fopp))
        {
            lastPoint (grid);
            return;
        }

        adj = (fadj <= level) ? 2 : 0;
        opp = (fopp >= level) ? 1 : 0;
        switch (adj + opp)
        {
            /* Exit EAST face */
            case 0:
                markInUse (grid, point, NORTH);
                markInUse (grid, point, SOUTH);
                exit_bearing = EAST;
                break;
            /* Exit SOUTH face */
            case 1:
                markInUse (grid, point, NORTH);
                markInUse (grid, point, EAST);
                exit_bearing = SOUTH;
                break;

```

```

        /* Exit EAST face */
        case 2:
            markInUse (grid, point, EAST);
            markInUse (grid, point, SOUTH);
            exit_bearing = NORTH;
            break;
    /* Exit NORTH or SOUTH face, depending upon contour level */
    case 3:
        exit_bearing = (grid->contour_mode) ? NORTH : SOUTH;
        break;

    }

    /* update face number, coordinate of defining corner */
    point->bearing = (point->bearing + exit_bearing) % 4;
    switch (point->bearing)
    {
        case EAST : point->x++; break;
        case NORTH : point->y--; break;
        case WEST : point->x--; break;
        case SOUTH : point->y++; break;
    }
}

/* initPoint -- Initialize the contour point list.
 * see also savePoint, lastPoint */

static void
initPoint(grid)
GRID *grid;
{
    grid->count = 0;
}

/* faceInUse -- Determine if the specified cell face has been marked as
 * contoured. This necessary to prevent infinite processing of closed lines.
 * see also : see also markInUse() */

static uchar
faceInUse(grid, point, face)
GRID *grid;
POINT *point;
uchar face;
{
    uchar r;
    face = (point->bearing + face) % 4;
    switch (face)
    {
        case NORTH:
        case SOUTH:
            r = grid->map[MXV_to_L(grid, point->x, point->y
+ (face == SOUTH ? 1 : 0))] & NS_MAP;
            break;
        case EAST:
        case WEST:
            r = grid->map[MXV_to_L(grid, point->x + (face ==
EAST ? 1 : 0), point->y)] & EW_MAP;
            break;
    }
    return r;
}

```

```

/* lastPoint -- Generate the actual contour line from the contour point
 * list. see also savePoint, lasrPoint */
static void
lastPoint(grid)
GRID *grid;
{
    if (grid->count)
        Polyline(grid->count, grid->list);
}

static uchar
savePoint(grid, point, level)
GRID *grid;
POINT *point;
float level;
{
    float last, next;
    float x, y, ds;
    char s[80];

    static int cnt = 0;

    last = getDataPoint (grid, point, SAME);
    next = getDataPoint (grid, point, NEXT);

    /* Are the points the same value ? */
    if (last == next)
    {
        printf ( " x, y, bearing = %2d, %2d, %d", point->x, point->y,
point->bearing);
        printf("%8g, %8g ", last, next);
        printf("potential divide by zero ! \n");
        return 0;
    }

    x = (float) point->x;
    y = (float) point->y;

    ds = (float) ((last - level)/(last-next));

    switch (point->bearing)
    {
        case EAST:          y += ds;          break;
        case NORTH: x += ds;   y += 1.0;      break;
        case WEST : x += 1.0;   y += 1.0 - ds; break;
        case SOUTH: x += 1.0 -ds;             break;
    }

    /* Update to contour point list */
    grid->list[grid->count].x = x/(float) (grid->dim_x -1);
    grid->list[grid->count].y = y/(float) (grid->dim_y -1);

    grid->count++;

    return 1;
}

static void
markInUse(grid, point, face)
GRID *grid;
POINT *point;
uchar face;
{

```



```

    face = (point->bearing + face) % 4;
    switch (face)
    {
        case NORTH:
        case SOUTH:
            grid->map[MXY_to_L(grid,
                point->x, point->y + (face == SOUTH ? 1 : 0))]
|= NS_MAP;
            break;
        case EAST:
        case WEST:
            grid->map[MXY_to_L (grid,
                point->x + (face == EAST ? 1 : 0), point -> y)]
|= EW_MAP;
            break;
    }
}

```

Appendix 2f - 3D Primitives

```

/*****
 *
 * The following are the 3d Primitives for GAL
 * Duncan Napier, McMaster University, Hamilton, Ontario, Canada
 * October 1992
 *
 *****/

#include "../include/xv.h"
#include "../include/color.h"

/*****
 * gal_3d_unfilled_sphere_() - this procedure draws a wireframe
 * "sphere" from a set of three normal circles sharing a common
 * centre.
 *****/

void gal_3d_unfilled_sphere_(color_index, rad, x, y, z)
int *color_index;
float *rad, *x, *y, *z;
{
    Xgl_3d_ctx      *ctx;
    Xgl_color        color;
    Xgl_circle_list  circle_list;
    Xgl_circle_f3d   circs;
    int              j,i;

/* define the circle planes */
    static Xgl_pt_f3d vector[5][3] = {{{1.,0.,0.}, {0.,1.,0.},
    {0.,0.,0.}}, {{1.,0.,0.}, {0.,0.,1.}, {0.,0.,0.}}, {{0.,1.,0.},
    {0.,0.,1.}, {0.,0.,0.}}, {{0.70711,1.,0.70711}, {-0.70711,1.,-0.70711},
    {-0.70711,0.70711,0.}}, {{-0.70711,1.,0.70711}, {0.70711,1.,-0.70711},
    {0.70711,0.70711,0.}}};

    /* allocate memory for the coordinates to be processed */

    if (!(circle_list.circles.f3d = (Xgl_circle_f3d
    *)malloc(5*sizeof(Xgl_circle_f3d))) {
        printf("\n \n memory allocation request in
        gal_unfilled_sphere failed\n \n"); exit(1);
    }

    ctx = get_3d_ctx();

    xgl_object_set(*ctx, XGL_CTX_CURVE_APPROX, XGL_CURVE_METRIC_VDC,
        XGL_CTX_CURVE_APPROX_VALUE, 0.1,
        XGL_CTX_MIN_TESSELLATION, 25,
        XGL_CTX_MAX_TESSELLATION, 30, NULL);

/* draw the circles */
    circle_list.type = XGL_MULTICIRCLE_F3D; circle_list.num_circles
    = 5; circle_list.bbox = 0;

    for (j = 0; j < 5; j++) {
        circle_list.circles.f3d[j].dir_normalized = FALSE;
        circle_list.circles.f3d[j].dir_normal = TRUE;
        circle_list.circles.f3d[j].center.flag = 0;

        for (i = 0; i < 3; i++) {
            circle_list.circles.f3d[j].dir[i] = vector[j][i];
        }
    }
}

```

```

        circle_list.circles.f3d[j].center.x = *x;
        circle_list.circles.f3d[j].center.y = *y;
        circle_list.circles.f3d[j].center.z = *z;
        circle_list.circles.f3d[j].radius = *rad;
    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
        *color_index; } else color.index = *color_index;

    xgl_object_set(*ctx, XGL_CTX_SURF_FRONT_FILL_STYLE, XGL_SURF_FILL_HOLLOW,
XGL_CTX_SURF_FRONT_COLOR, &color, NULL);

    xgl_multicircle(*ctx, &circle_list);
    free(circle_list.circles.f3d);
}

/*****
 *
 * gal_3d_filled_sphere() - this draws an XGL annote circle, one that
 * remains in the plane of the screen and is used to represent a
 * parallel
 * projected square.
 *
 *****/

void gal_3d_filled_sphere_(color_index, rad, x, y, z)
int    *color_index;
float  *rad, *x, *y, *z;
{
    Xgl_3d_ctx      *ctx;
    Xgl_color        color;
    Xgl_circle_list  circle_list;
    Xgl_circle_af3d  circs;

    /*
     * get a smooth circle */

    ctx = get_3d_ctx();

    xgl_object_set(*ctx, XGL_CTX_CURVE_APPROX, XGL_CURVE_METRIC_VDC,
        XGL_CTX_CURVE_APPROX_VALUE, 0.1,
        XGL_CTX_MIN_TESSELLATION, 25,
        XGL_CTX_MAX_TESSELLATION, 30, NULL);

    /* draw the circles */
    circle_list.type = XGL_MULTICIRCLE_AF3D;
    circle_list.num_circles = 1;
    circle_list.bbox = 0;
    circs.center.flag = 1;
    circs.center.x = *x; circs.center.y = *y;
    circs.center.z = *z; circs.radius = *rad;

    circle_list.circles.af3d = &circs; if (get_dbuf_on()) {
    color.index = *color_index*COLOR_SIZE + *color_index; } else
    color.index = *color_index;

    /* set the edges on the soild circle */

    xgl_object_set (*ctx,XGL_CTX_SURF_FRONT_COLOR, &color,
XGL_CTX_SURF_EDGE_FLAG, TRUE, XGL_CTX_EDGE_COLOR, &black_color,
    NULL);

    xgl_multicircle(*ctx, &circle_list);

```

```

}

/*****
 * gal_3d_solid_line_() - draws a solid line given the color, thickness,
 * number of points and coordinates.
 *
 * routines called - malloc, xgl_object_set, xgl_multipolyline.
 *****/

void gal_3d_solid_line_(color_index, thickness, num_pts, list_pts_x,
list_pts_y, list_pts_z)
float list_pts_x[], list_pts_y[], list_pts_z[], *thickness;
int *color_index, *num_pts;
{
    Xgl_3d_ctx      *ctx;
    Xgl_color        color;
    Xgl_pt_list      line_pts;
    int              i;

    /* allocate memory for the coordinates to be processed */

    if (!(line_pts.pts.f3d = (Xgl_pt_f3d
*)malloc(*num_pts*sizeof(Xgl_pt_f3d)))) {
        printf("memory allocation request in gal_solid_line
        failed\n"); exit(1); }

    line_pts.pt_type = XGL_PT_F3D;
    line_pts.num_pts = *num_pts;
    line_pts.bbox = 0;

    for ( i=0; i<*num_pts; i++) {

        line_pts.pts.f3d[i].x = list_pts_x[i];
        line_pts.pts.f3d[i].y = list_pts_y[i];
        line_pts.pts.f3d[i].z = list_pts_z[i];

    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
    *color_index; } else color.index = *color_index;

    ctx = get_3d_ctx();

    xgl_object_set(*ctx, XGL_CTX_LINE_WIDTH_SCALE_FACTOR,
    *thickness, XGL_CTX_LINE_COLOR, &color, NULL);

    xgl_multipolyline(*ctx, NULL, 1, line_pts);

    free(line_pts.pts.f3d); }

/*****
 * gal_3d_dotted_line_() - draws a dotted line given the color, thickness,
 * number of points and coordinates.
 *
 * routines called - malloc, xgl_object_set, xgl_multipolyline.
 *****/

void gal_3d_dotted_line_(color_index, thickness, num_pts, list_pts_x,
list_pts_y, list_pts_z)
float list_pts_x[], list_pts_y[], list_pts_z[], *thickness;
int *color_index, *num_pts;

```

```

{

    Xgl_3d_ctx      *ctx;
    Xgl_color       color;
    Xgl_pt_list     line_pts;
    int             i;

    /* allocate memory for the coordinates to be processed */

    if (!(line_pts.pts.f3d = (Xgl_pt_f3d
*)malloc(*num_pts*sizeof(Xgl_pt_f3d)))) {
        printf("memory allocation request in gal_3d_dotted_line
        failed\n"); exit(1); }

    line_pts.pt_type = XGL_PT_F3D; line_pts.num_pts = *num_pts;
    line_pts.bbox = 0;

    for ( i=0; i<*num_pts; i++) {

        line_pts.pts.f3d[i].x = list_pts_x[i];
        line_pts.pts.f3d[i].y = list_pts_y[i];
        line_pts.pts.f3d[i].z = list_pts_z[i];

    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
    *color_index; } else color.index = *color_index;

    ctx = get_3d_ctx();

    xgl_object_set(*ctx,
        XGL_CTX_LINE_WIDTH_SCALE_FACTOR, *thickness,
        XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNEED,
        XGL_CTX_LINE_PATTERN, xgl_lpat_dotted,
        XGL_CTX_LINE_COLOR, &color, NULL);

    xgl_multipolyline(*ctx, NULL, 1, line_pts);

    free(line_pts.pts.f3d); }

/*****
 * gal_3d_dashed_line() - draws a dashed line given the color, thickness,
 * number of points and coordinates.
 *
 * routines called - malloc, xgl_object_set, xgl_multipolyline.
 *****/

void gal_3d_dashed_line(color_index, thickness, num_pts, list_pts_x,
list_pts_y, list_pts_z)
float list_pts_x[], list_pts_y[], list_pts_z[], *thickness;
int *color_index, *num_pts;
{

    Xgl_3d_ctx      *ctx;
    Xgl_color       color;
    Xgl_pt_list     line_pts;
    int             i;

    /* allocate memory for the coordinates to be processed */

```

```

if (!(line_pts.pts.f3d = (Xgl_pt_f3d
*)malloc(*num_pts*sizeof(Xgl_pt_f3d)))) {
    printf("memory allocation request in gal_3d_dashed_line
    failed\n"); exit(1); }

line_pts.pt_type = XGL_PT_F3D; line_pts.num_pts = *num_pts;
line_pts.bbox = 0;

for ( i=0; i<*num_pts; i++) {

    line_pts.pts.f3d[i].x = list_pts_x[i];
    line_pts.pts.f3d[i].y = list_pts_y[i];
    line_pts.pts.f3d[i].z = list_pts_z[i];

}

if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
*color_index; } else color.index = *color_index;

ctx = get_3d_ctx();

xgl_object_set(*ctx,
    XGL_CTX_LINE_WIDTH_SCALE_FACTOR, *thickness,
    XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNEDED,
    XGL_CTX_LINE_PATTERN, xgl_lpat_dashed,
    XGL_CTX_LINE_COLOR, &color, NULL);

xgl_multipolyline(*ctx, NULL, 1, line_pts);

free(line_pts.pts.f3d); }

/*****
* gal_3d_dash_dotted_line_() - draws a dashed-dotted line given the color,
* thickness, number of points and coordinates.
*
* routines called - malloc, xgl_object_set, xgl_multipolyline.
*****/

void gal_3d_dash_dotted_line_(color_index, thickness, num_pts,
list_pts_x, list_pts_y, list_pts_z)
float list_pts_x[], list_pts_y[], list_pts_z[], *thickness;
int *color_index, *num_pts;
{
    Xgl_3d_ctx      *ctx;
    Xgl_color       color;
    Xgl_pt_list     line_pts;
    int             i;

    /* allocate memory for the coordinates to be processed */

    if (!(line_pts.pts.f3d = (Xgl_pt_f3d
*)malloc(*num_pts*sizeof(Xgl_pt_f3d)))) {
        printf("\n \n memory allocation request in
        gal_3d_dash_dotted_line failed\n \n"); exit(1); }

    line_pts.pt_type = XGL_PT_F3D; line_pts.num_pts = *num_pts;
    line_pts.bbox = 0;

```

```

for ( i=0; i<*num_pts; i++) {

    line_pts.pts.f3d[i].x = list_pts_x[i];
    line_pts.pts.f3d[i].y = list_pts_y[i];
    line_pts.pts.f3d[i].z = list_pts_z[i];

}

if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
*color_index; } else color.index = *color_index;

ctx = get_3d_ctx();

xgl_object_set(*ctx,
               XGL_CTX_LINE_WIDTH_SCALE_FACTOR, *thickness,
               XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNEED,
               XGL_CTX_LINE_PATTERN, xgl_lpat_dashed_dotted,
               XGL_CTX_LINE_COLOR, &color, NULL);

xgl_multipolyline(*ctx, NULL, 1, line_pts);

free(line_pts.pts.f3d); }

/*****
 * gal_3d_dash_dot_line_() - draws a dashed-dotted line given the color,
 * thickness, number of points and coordinates.
 *
 * routines called - malloc, xgl_object_set, xgl_multipolyline.
 *****/

void gal_3d_dash_dot_line_(color_index, thickness, num_pts, list_pts_x,
list_pts_y, list_pts_z)
float list_pts_x[], list_pts_y[], list_pts_z[], *thickness;
int *color_index, *num_pts;
{

    Xgl_3d_ctx      *ctx;
    Xgl_color        color;
    Xgl_pt_list      line_pts;
    int              i;

    /* allocate memory for the coordinates to be processed */

    if (!(line_pts.pts.f3d = (Xgl_pt_f3d
*)malloc(*num_pts*sizeof(Xgl_pt_f3d)))) {
        printf("memory allocation request in gal_3d_dash_dot
line failed\n"); exit(1); }

    line_pts.pt_type = XGL_PT_F3D; line_pts.num_pts = *num_pts;
    line_pts.bbox = 0;

    for ( i=0; i<*num_pts; i++) {

        line_pts.pts.f3d[i].x = list_pts_x[i];
        line_pts.pts.f3d[i].y = list_pts_y[i];
        line_pts.pts.f3d[i].z = list_pts_z[i];

    }

```



```

        if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
        *color_index; } else color.index = *color_index;

        ctx = get_3d_ctx();

        xgl_object_set(*ctx,
                        XGL_CTX_LINE_WIDTH_SCALE_FACTOR, *thickness,
                        XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNEED,
                        XGL_CTX_LINE_PATTERN, xgl_lpat_dash_dot,
                        XGL_CTX_LINE_COLOR, &color, NULL);

        xgl_multipolyline(*ctx, NULL, 1, line_pts);

        free(line_pts.pts.f3d); }

/*****
 * gal_3d_dash_dot_dot_line_() - draws a dashed-dotted line given the color,
 * thickness, number of points and coordinates.
 *
 * routines called - malloc, xgl_object_set, xgl_multipolyline.
 *****/

void gal_3d_dash_dot_dot_line_(color_index, thickness, num_pts,
list_pts_x, list_pts_y, list_pts_z)
float list_pts_x[], list_pts_y[], list_pts_z[], *thickness;
int *color_index, *num_pts;
{
    Xgl_3d_ctx      *ctx;
    Xgl_ccolor      color;
    Xgl_pt_list     line_pts;
    int             i;

    /* allocate memory for the coordinates to be processed */

    if (!(line_pts.pts.f3d = (Xgl_pt_f3d
*)malloc(*num_pts*sizeof(Xgl_pt_f3d)))) {
        printf("memory allocation request in
        gal_3d_dash_dot_dot line failed\n"); exit(1); }

    line_pts.pt_type = XGL_PT_F3D; line_pts.num_pts = *num_pts;
    line_pts.bbox = 0;

    for ( i=0; i<*num_pts; i++) {

        line_pts.pts.f3d[i].x = list_pts_x[i];
        line_pts.pts.f3d[i].y = list_pts_y[i];
        line_pts.pts.f3d[i].z = list_pts_z[i]; }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
    *color_index; } else color.index = *color_index;

    ctx = get_3d_ctx();

    xgl_object_set(*ctx,
                    XGL_CTX_LINE_WIDTH_SCALE_FACTOR, *thickness,
                    XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNEED,
                    XGL_CTX_LINE_PATTERN, xgl_lpat_dash_dot_dotted,
                    XGL_CTX_LINE_COLOR, &color, NULL);

```

```

    xgl_multipolyline(*ctx, NULL, 1, line_pts);

    free(line_pts.pts.f3d); }

/*****
 * gal_3d_long_dash_line_() - draws a dashed-dotted line given the color,
 * thickness, number of points and coordinates.
 *
 * routines called - malloc, xgl_object_set, xgl_multipolyline.
 *****/

void gal_3d_long_dashed_line_(color_index, thickness, num_pts,
list_pts_x, list_pts_y, list_pts_z)
float list_pts_x[], list_pts_y[], list_pts_z[], *thickness;
int *color_index, *num_pts;
{
    Xgl_3d_ctx      *ctx;
    Xgl_color       color;
    Xgl_pt_list     line_pts;
    int             i;

    /* allocate memory for the coordinates to be processed */

    if (!(line_pts.pts.f3d = (Xgl_pt_f3d
*)malloc(*num_pts*sizeof(Xgl_pt_f3d)))) {
        printf("memory allocation request in gal_3d_long_dashed
line failed\n"); exit(1); }

    line_pts.pt_type = XGL_PT_F3D; line_pts.num_pts = *num_pts;
    line_pts.bbox = 0;

    for ( i=0; i<*num_pts; i++) {

        line_pts.pts.f3d[i].x = list_pts_x[i];
        line_pts.pts.f3d[i].y = list_pts_y[i];
        line_pts.pts.f3d[i].z = list_pts_z[i]; }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
*color_index; } else color.index = *color_index;

    ctx = get_3d_ctx();

    xgl_object_set(*ctx,
        XGL_CTX_LINE_WIDTH_SCALE_FACTOR, *thickness,
        XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNEED,
        XGL_CTX_LINE_PATTERN, xgl_lpat_long_dashed,
        XGL_CTX_LINE_COLOR, &color, NULL);

    xgl_multipolyline(*ctx, NULL, 1, line_pts);

    free(line_pts.pts.f3d); }

/*****
 * gal_3d_cross_marker_() - a routine to draw a given marker type. The
 * user supplies the color, number of markers and their locations
 *
 * routines called - malloc, xgl_object_set, xgl_multimarker
 *****/

```

```

*****/

void gal_3d_cross_marker_(color_index, num_pts, list_pts_x, list_pts_y,
list_pts_z)
float list_pts_x[], list_pts_y[], list_pts_z[];
int *color_index, *num_pts;
{
    Xgl_3d_ctx      *ctx;
    Xgl_color       color;
    Xgl_pt_list     marker_pts;
    int             i;

    /* allocate memory for the coordinates to be processed */

    if (!(marker_pts.pts.f3d = (Xgl_pt_f3d
*)malloc(*num_pts*sizeof(Xgl_pt_f3d)))) {
        printf("\n \n memory allocation request in
gal_3d_cross_marker failed\n \n"); exit(1); }

    marker_pts.pt_type = XGL_PT_F3D; marker_pts.num_pts = *num_pts;
    marker_pts.bbox = 0;

    for ( i=0; i<*num_pts; i++) {

        marker_pts.pts.f3d[i].x = list_pts_x[i];
        marker_pts.pts.f3d[i].y = list_pts_y[i];
        marker_pts.pts.f3d[i].z = list_pts_z[i];

    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
*color_index; } else color.index = *color_index;

    ctx = get_3d_ctx();

    /* the marker size is defaulted to 10 pixels in size */
    xgl_object_set(*ctx,
        XGL_CTX_MARKER_COLOR, &color,
        XGL_CTX_MARKER_SCALE_FACTOR, 10.0,
        XGL_CTX_MARKER_DESCRIPTION, xgl_marker_cross,
        NULL);

    xgl_multimarker(*ctx, marker_pts);

    free(marker_pts.pts.f3d); }

/*****
 * gal_3d_plus_marker_() - a routine to draw a given marker type. The
 * user supplies the color, number of markers and their locations
 *
 * routines called - malloc, xgl_object_set, xgl_multimarker
 *****/

void gal_3d_plus_marker_(color_index, num_pts, list_pts_x, list_pts_y,
list_pts_z)
float list_pts_x[], list_pts_y[], list_pts_z[];
int *color_index, *num_pts;
{

```

```

Xgl_3d_ctx      *ctx;
Xgl_color       color;
Xgl_pt_list     marker_pts;
int             i;

/* allocate memory for the coordinates to be processed */

if (!(marker_pts.pts.f3d = (Xgl_pt_f3d
*)malloc(*num_pts*sizeof(Xgl_pt_f3d)))) {
    printf("\n \n memory allocation request in
    gal_3d_plus_marker failed\n \n"); exit(1); }

marker_pts.pt_type = XGL_PT_F3D; marker_pts.num_pts = *num_pts;
marker_pts.bbox = 0;

for ( i=0; i<*num_pts; i++) {

    marker_pts.pts.f3d[i].x = list_pts_x[i];
    marker_pts.pts.f3d[i].y = list_pts_y[i];
    marker_pts.pts.f3d[i].z = list_pts_z[i];

}

if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
*color_index; } else color.index = *color_index;

ctx = get_3d_ctx();

/* the marker size is defaulted to 10 pixels in size */
xgl_object_set(*ctx,
    XGL_CTX_MARKER_COLOR, &color,
    XGL_CTX_MARKER_SCALE_FACTOR, 10.0,
    XGL_CTX_MARKER_DESCRIPTION, xgl_marker_plus,
    NULL);

xgl_multimarker(*ctx, marker_pts);

free(marker_pts.pts.f3d); }

/*****
* gal_3d_asterisk_marker_() - a routine to draw a given marker type.
* The user supplies the color, number of markers and their locations
*
* routines called - malloc, xgl_object_set, xgl_multimarker
*****/

void gal_3d_asterisk_marker_(color_index, num_pts, list_pts_x,
list_pts_y, list_pts_z)
float list_pts_x[], list_pts_y[], list_pts_z[];
int *color_index, *num_pts;
{

    Xgl_3d_ctx      *ctx;
    Xgl_color       color;
    Xgl_pt_list     marker_pts;
    int             i;

/* allocate memory for the coordinates to be processed */

if (!(marker_pts.pts.f3d = (Xgl_pt_f3d
*)malloc(*num_pts*sizeof(Xgl_pt_f3d)))) {
    printf("\n \n memory allocation request in

```

```

        gal_3d_asterisk_marker failed\n \n"); exit(1); }

marker_pts.pt_type = XGL_PT_F3D; marker_pts.num_pts = *num_pts;
marker_pts.bbox = 0;

for ( i=0; i<*num_pts; i++) {

    marker_pts.pts.f3d[i].x = list_pts_x[i];
    marker_pts.pts.f3d[i].y = list_pts_y[i];
    marker_pts.pts.f3d[i].z = list_pts_z[i];

}

if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
*color_index; } else color.index = *color_index;

ctx = get_3d_ctx();

/* the marker size is defaulted to 10 pixels in size */
xgl_object_set(*ctx,
               XGL_CTX_MARKER_COLOR, &color,
               XGL_CTX_MARKER_SCALE_FACTOR, 10.0,
               XGL_CTX_MARKER_DESCRIPTION,
               xgl_marker_asterisk, NULL);

xgl_multimarker(*ctx, marker_pts);

free(marker_pts.pts.f3d); }

/*****
 * gal_3d_square_marker_() - a routine to draw a given marker type. The
 * user supplies the color, number of markers and their locations
 *
 * routines called - malloc, xgl_object_set, xgl_multimarker
 *****/

void gal_3d_square_marker_(color_index, num_pts, list_pts_x,
list_pts_y, list_pts_z)
float list_pts_x[], list_pts_y[], list_pts_z[];
int *color_index, *num_pts;
{
    Xgl_3d_ctx      *ctx;
    Xgl_color       color;
    Xgl_pt_list     marker_pts;
    int             i;

    /* allocate memory for the coordinates to be processed */

    if (!(marker_pts.pts.f3d = (Xgl_pt_f3d
*)malloc(*num_pts*sizeof(Xgl_pt_f3d)))) {
        printf("\n \n memory allocation request in
        gal_3d_square_marker failed\n \n"); exit(1); }

    marker_pts.pt_type = XGL_PT_F3D; marker_pts.num_pts = *num_pts;
    marker_pts.bbox = 0;

    for ( i=0; i<*num_pts; i++) {

        marker_pts.pts.f3d[i].x = list_pts_x[i];
        marker_pts.pts.f3d[i].y = list_pts_y[i];
        marker_pts.pts.f3d[i].z = list_pts_z[i];

```

```

    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
    *color_index; } else color.index = *color_index;

    ctx = get_3d_ctx();

    /* the marker size is defaulted to 10 pixels in size */
    xgl_object_set(*ctx,
                  XGL_CTX_MARKER_COLOR, &color,
                  XGL_CTX_MARKER_SCALE_FACTOR, 10.0,
                  XGL_CTX_MARKER_DESCRIPTION, xgl_marker_square,
                  NULL);

    xgl_multimarker(*ctx, marker_pts);

    free(marker_pts.pts.f3d); }

/*****
 * gal_3d_circle_marker() - a routine to draw a given marker type. The
 * user supplies the color, number of markers and their locations
 *
 * routines called - malloc, xgl_object_set, xgl_multimarker
 *****/

void gal_3d_circle_marker(color_index, num_pts, list_pts_x,
list_pts_y, list_pts_z)
float list_pts_x[], list_pts_y[], list_pts_z[];
int *color_index, *num_pts;
{
    Xgl_3d_ctx      *ctx;
    Xgl_color       color;
    Xgl_pt_list     marker_pts;
    int             i;

    /* allocate memory for the coordinates to be processed */

    if (!(marker_pts.pts.f3d = (Xgl_pt_f3d
    *)malloc(*num_pts*sizeof(Xgl_pt_f3d))) {
        printf("\n \n memory allocation request in
        gal_3d_circle_marker failed\n \n"); exit(1); }

    marker_pts.pt_type = XGL_PT_F3D; marker_pts.num_pts = *num_pts;
    marker_pts.bbox = 0;

    for ( i=0; i<*num_pts; i++) {

        marker_pts.pts.f3d[i].x = list_pts_x[i];
        marker_pts.pts.f3d[i].y = list_pts_y[i];
        marker_pts.pts.f3d[i].z = list_pts_z[i];

    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
    *color_index; } else color.index = *color_index;

    ctx = get_3d_ctx();

    /* the marker size is defaulted to 10 pixels in size */
    xgl_object_set(*ctx,
                  XGL_CTX_MARKER_COLOR, &color,
                  XGL_CTX_MARKER_SCALE_FACTOR, 10.0,
                  XGL_CTX_MARKER_DESCRIPTION, xgl_marker_circle,

```

```

        NULL);

    xgl_multimarker(*ctx, marker_pts);

    free(marker_pts.pts.f3d); }

/*****
 * gal_3d_text_annotate() - draws a string of text to the screen. The user
 * supplies the string, font size, font (i.e. the name of the XGL font file),
 * the spacing between letters, the orientation and the color
 *
 * routines called - xgl_object_set, xgl_annotate_text;
 *****/

void gal_3d_text_annotate(color_index, string, font_size, font,
font_spacing, x_pos, y_pos, z_pos, x_vector, y_vector, str_len,
font_len)
float *font_size, *font_spacing, *x_pos, *y_pos, *z_pos, *x_vector, *y_vector;
int *color_index, str_len, font_len;
char *string, *font;
{
    Xgl_sfont sfont;
    Xgl_obj_desc obj_desc;
    Xgl_pt_f3d text_pos;
    Xgl_pt_f2d up_vector;
    Xgl_color sf_color;
    Xgl_3d_ctx *ctx;
    char *font_copy;
    Xgl_gcache *gcache;
    static Xgl_pt_f3d dir[2] = { {1., 0., 0.}, {0., 1., 0.}};

    ctx = get_3d_ctx();
    gcache = get_gcache();

    if (!(font_copy = malloc(80))) {
        printf("\n \n memory allocation request in
        gal_3d_text_annotate failed\n \n");
        exit(1); }

    (void) strcpy(font_copy, font, font_len);

    /* append the '.phont' font file extension to the font
     * name. */

    (void) strcat(font_copy, ".phont");

    if (!(obj_desc.sfont_name = (char *)malloc(80))) {
        printf("memory allocation request in gal_3d_text_annotate
        failed\n");
        exit(1); }

    strcpy(obj_desc.sfont_name, font_copy);

    free(font_copy);

    sfont = xgl_object_create (*ctx, XGL_SFONTS, &obj_desc, NULL);

    free(obj_desc.sfont_name);

    up_vector.x = *x_vector; up_vector.y = *y_vector;

    if (get_dbuf_on()) { sf_color.index = *color_index*COLOR_SIZE +
    *color_index; } else sf_color.index = *color_index;

```

```

xgl_object_set(*ctx,
               XGL_CTX_SFONTS, sfont,
               XGL_CTX_SFONTS_CHAR_HEIGHT, *font_size,
               XGL_CTX_SFONTS_CHAR_SPACING, *font_spacing,
               XGL_CTX_SFONTS_CHAR_UP_VECTOR, up_vector,
               XGL_CTX_SFONTS_TEXT_COLOR, &sf_color,
               XGL_CTX_LINE_COLOR, &sf_color, NULL);

text_pos.x = *x_pos, text_pos.y = *y_pos, text_pos.z = *z_pos;

xgl_gcache_stroke_text (*gcache, *ctx, string, &text_pos,
                        dir);
xgl_context_display_gcache(*ctx, *gcache, FALSE, TRUE);
xgl_context_post(*ctx, TRUE);
}

/*****
 * gal_3d_real_annotate() - draws a string of text to the screen. The
 * user supplies the string, font size, font (i.e. the name of the XGL
 * font file), the spacing between letters, the orientation and the color
 *
 * routines called - xgl_object_set, xgl_annotate_text;
 *****/

void gal_3d_real_annotate(color_index, real, log, mantissa, font_size,
font, font_spacing, x_pos, y_pos, z_pos, x_vector, y_vector, font_len)
float *font_size, *font_spacing, *x_pos, *y_pos, *z_pos, *x_vector, *y_vector, *real;
int *color_index, font_len, *mantissa, *log;
char *font;
{
    char string[20];
    Xgl_sfont sfont;
    Xgl_obj_desc obj_desc;
    Xgl_pt_f3d text_pos;
    Xgl_pt_f2d up_vector;
    Xgl_color sf_color;
    Xgl_3d_ctx *ctx;
    char *font_copy, format[20];
    Xgl_gcache *gcache;
    static Xgl_pt_f3d dir[2] = { {1., 0., 0.}, {0., 1., 0.}};

    ctx = get_3d_ctx(); gcache = get_gcache();

    if (!(font_copy = malloc(80))) {
        printf("memory allocation request in gal_3d_real_annotate\n");
        exit(1); }

    (void) strcpy(font_copy, font, font_len);

    /* append the '.phont' font file extension to the font
     * name. */

    (void) strcat(font_copy, ".phont");

    if (!(obj_desc.sfont_name = (char *)malloc(80))) {
        printf("\n\n memory allocation request in\n\n");
        gal_3d_text_annotate failed\n\n");
        exit(1); }

    strcpy(obj_desc.sfont_name, font_copy);

    free(font_copy);

```



```

    sfont = xgl_object_create (*ctx, XGL_SFONTE, &obj_desc, NULL);

    free(obj_desc.sfont_name);

    sprintf(format, "%%d.%%f", *log, *mantissa); sprintf(string,
    format, *real);
    up_vector.x = *x_vector;
    up_vector.y = *y_vector;

    if (get_dbuf_on()) {
        sf_color.index = *color_index*COLOR_SIZE +
        *color_index; } else sf_color.index = *color_index;

    xgl_object_set(*ctx,
        XGL_CTX_SFONTE, sfont,
        XGL_CTX_SFONTE_CHAR_HEIGHT, *font_size,
        XGL_CTX_SFONTE_CHAR_SPACING, *font_spacing,
        XGL_CTX_SFONTE_CHAR_UP_VECTOR, up_vector,
        XGL_CTX_SFONTE_TEXT_COLOR, &sf_color,
        XGL_CTX_LINE_COLOR, &sf_color, NULL);

    text_pos.x = *x_pos, text_pos.y = *y_pos, text_pos.z = *z_pos;

    xgl_gcache_stroke_text (*gcache, *ctx, string, &text_pos,
    dir);
    xgl_context_display_gcache(*ctx, *gcache, FALSE, TRUE);
    xgl_context_post(*ctx, TRUE);
}

*****
* gal_3d_axes() - draws x, y and axes given maxima, minima and increments
*
* procedures called - gal_3d_solid_line_()
*****/

void gal_3d_axes(color_index, xmin, xmax, ymin, ymax, zmin, zmax,
x_interval, y_interval, z_interval)
float *xmin, *xmax, *ymin, *ymax, *zmin, *zmax;
float *x_interval, *y_interval, *z_interval;
int *color_index;
{
    extern float axis_thickness, zero;
    extern int axis_num_pts;
    float list_pts_x[2], list_pts_y[2], list_pts_z[2];
    float *xmin_new, *ymin_new, *zmin_new;
    float tic_x, tic_y, tic_z, tic_start, tic_fin;
    int ntics, i;

    /* If the extremities of the axes cross the origin, make the
    axis lie on the origin */

    if ((*ymax > 0.) && (*ymin < 0.)) ymin_new = &zero;
        else ymin_new = ymin;
    if ((*xmax > 0.) && (*xmin < 0.)) xmin_new = &zero;
        else xmin_new = xmin;
    if ((*zmax > 0.) && (*zmin < 0.)) zmin_new = &zero;
        else zmin_new = zmin;

    /* Draw the x-axis */
    list_pts_x[0] = *xmin;
    list_pts_x[1] = *xmax;
    list_pts_y[0] = *ymin_new;

```

```

list_pts_y[1] = *ymin_new;
list_pts_z[0] = *zmin_new;
list_pts_z[1] = *zmin_new;

gal_3d_solid_line_(color_index, &axis_thickness, &axis_num_pts,
list_pts_x, list_pts_y, list_pts_z);

/* Draw the y-axis */
list_pts_x[0] = *xmin_new;
list_pts_x[1] = *xmin_new;
list_pts_y[0] = *ymin;
list_pts_y[1] = *ymax;

gal_3d_solid_line_(color_index, &axis_thickness, &axis_num_pts,
list_pts_x, list_pts_y, list_pts_z);

/* Draw the z-axis */
list_pts_y[0] = *ymin_new;
list_pts_y[1] = *ymin_new;
list_pts_z[0] = *zmin;
list_pts_z[1] = *zmax;

gal_3d_solid_line_(color_index, &axis_thickness, &axis_num_pts,
list_pts_x, list_pts_y, list_pts_z);

/* calculate the size of the tics (1/50 the axis length) */
tic_x = (*ymax - *ymin)/50;
tic_y = (*xmax - *xmin)/50;
tic_z = (*ymax - *ymin)/50;

/* calculate the number of x-axis tics */
ntics = (int) ((*xmax-*xmin)/ *x_interval);

/* calculate the y coordinates of the tics */
tic_start = *ymin_new - tic_x;
tic_fin = *ymin_new + tic_x;

/* draw the n+1 tics */
for (i = 0; i <= (ntics+1); i++) {
    list_pts_x[1] = list_pts_x[0] = *xmin + i*(*x_interval);
    list_pts_z[1] = list_pts_z[0] = *zmin_new;
    list_pts_y[0] = tic_start; list_pts_y[1] = tic_fin;
    gal_3d_solid_line_(color_index, &axis_thickness, &axis_num_pts, list_pts_x
, list_pts_y, list_pts_z);
}

/* calculate the number of y-axis tics */ ntics = (int)
((*ymax-*ymin)/ *y_interval);

/* calculate the y coordinates of the tics */ tic_start =
*xmin_new - tic_y; tic_fin = *xmin_new + tic_y;

/* draw the n+1 tics */
for (i = 0; i <= (ntics+1); i++) {
    list_pts_y[1] = list_pts_y[0] = *ymin + i*(*y_interval);
    list_pts_z[1] = list_pts_z[0] = *zmin_new;
    list_pts_x[0] = tic_start; list_pts_x[1] = tic_fin;
    gal_3d_solid_line_(color_index, &axis_thickness, &axis_num_pts,
list_pts_x, list_pts_y, list_pts_z);
}

/* calculate the number of z-axis tics */ ntics = (int)
((*zmax-*zmin)/ *z_interval);

```

```

/* calculate the y coordinates of the tics */ tic_start =
*ymin_new - tic_z; tic_fin = *ymin_new + tic_z;

/* draw the n+1 tics */
for (i = 0; i <= (ntics+1); i++) {
    list_pts_z[1] = list_pts_z[0] = *zmin + i*(z_interval);
    list_pts_x[1] = list_pts_x[0] = *xmin_new;
    list_pts_y[0] = tic_start;
    list_pts_y[1] = tic_fin;
    gal_3d_solid_line_(color_index, &axis_thickness,
    &axis_num_pts, list_pts_x, list_pts_y, list_pts_z);
}

}

/*****
* gal_quadrilateral_mesh_() - draws a quadrilateral mesh given the color
* the number of rows, number of columns of data and a vector containing
* ordered triplets of x,y,z coordinates.
*
* procedures called - malloc, get_3d_ctx, xgl_object_set,
* xgl_quadrilateral_mesh
*****/
void gal_quadrilateral_mesh_(color_index, row_index, column_index,
num_pts, mesh)
int    *column_index, *row_index, *num_pts, *color_index;
float  mesh[];
{
    Xgl_3d_ctx      *ctx3d;
    Xgl_pt_list     pl;
    Xgl_color        color;
    int              i, j, k;

    /* allocate memory for the coordinates to be processed */

    if (!(pl.pts.f3d = (Xgl_pt_f3d
    *)malloc(*column_index*(*row_index)*sizeof(Xgl_pt_f3d)))) {
        printf("memory allocation request in gal_quadrilateral_mesh
        failed\n"); exit(1); }

    /* set the type and pointers to the coordinates */ pl.pt_type =
    XGL_PT_F3D; pl.bbox = NULL; pl.num_pts = *num_pts;

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
    *color_index; } else color.index = *color_index;

    /* allocate every third vector element to a new pl.pts.f3d */
    j = 0;
    for (i=0; i< (*column_index)*(*row_index) ; i++) {

        pl.pts.f3d[i].x = mesh[j++]; pl.pts.f3d[i].y =
        mesh[j++]; pl.pts.f3d[i].z = mesh[j++];

    }

    ctx3d = get_3d_ctx();

    xgl_object_set(*ctx3d,
        XGL_CTX_SURF_FRONT_FILL_STYLE,
        XGL_SURF_FILL_HOLLOW, XGL_CTX_SURF_FRONT_COLOR,
        &color, NULL);

    xgl_quadrilateral_mesh(*ctx3d,*row_index,*column_index,NULL,pl);

```

```

        free(pl.pts.f3d); }

/*****
 * gal_quadrilateral_surface_() - draws a quadrilateral surface given the color
 * the number of rows, number of columns of data ,a vector containing
 * ordered triplets of x,y,z coordinates and a listing of every color of every
 * quad on the surface.
 *
 * procedures called - malloc, get_3d_ctx, xgl_object_set,
 * xgl_quadrilateral_mesh
 *****/
void gal_quadrilateral_mesh_(color_index, row_index, column_index,
num_pts, mesh, color_list)
int      *column_index, *row_index, *num_pts, *color_index;
float    mesh[];
int      color_list[];
{
    Xgl_3d_ctx      *ctx3d;
    Xgl_facet_list  facet_list;
    Xgl_pt_list     pl;
    Xgl_color       color;
    int             i, j, k;

    /* allocate memory for the coordinates to be processed */

    if (!( pl.pts.f3d = (Xgl_pt_f3d
*)malloc(*column_index*(*row_index)*sizeof(Xgl_pt_f3d))) ) {
        printf("memory allocation request in gal_quadrilateral_mesh
failed\n"); exit(1); }

    if (!( facet_list.facets.color_facets = (Xgl_color_facet
*)malloc((*column_index-1)*(*row_index-1)*sizeof
(Xgl_color_facet))) ) {printf("memory allocation request in
gal_quadrilateral_surface failed\n"); exit(1); }

    /* double buffer the facet list if the double buffer is set */
    if (get_dbuf_on()) { for
(i = 0; i < facet_list.num_facets; i++) {
        facet_list.facets.color_facets[i].color.index =
        color_list[i]*COLOR_SIZE + color_list[i];
    }
    }
    else { for (i = 0; i < facet_list.num_facets; i++) {
        facet_list.facets.color_facets[i].color.index = color_list[i];
    }
    }

    /* set the type and pointers to the coordinates */ pl.pt_type =
XGL_PT_F3D; pl.bbox = NULL; pl.num_pts = *num_pts;

    /* set the facet_list members */
    facet_list.facet_type = XGL_FACET_COLOR;
    facet_list.num_facets = (*row_index-1)*(*column_index-1);

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
*color_index; } else color.index = *color_index;

    /* allocate every third vector element to a new pl.pts.f3d */
    j = 0;
    for (i=0; i< (*column_index)*(*row_index) ; i++) {

        pl.pts.f3d[i].x = mesh[j++]; pl.pts.f3d[i].y =

```

```

        mesh[j++]; pl.pts.f3d[i].z = mesh[j++];

    }

    ctx3d = get_3d_ctx();
    xgl_object_set(*ctx3d,
        XGL_CTX_SURF_EDGE_FLAG, TRUE,
        XGL_CTX_EDGE_COLOR, &color,
        NULL);

    xgl_quadrilateral_mesh(*ctx3d,*row_index,*column_index,facet_list,pl);

    free(pl.pts.f3d);

    free(facet_list.facets.color_facets);}

/*****
 * gal_3d_unfilled_polygon()- draws a polygon from a list of 3d
 * points.
 *
 * procedures called - malloc, xgl_object_set, xgl_polygon
 *****/

void gal_3d_unfilled_polygon_(color_index, num_pts, list_pts_x,
list_pts_y, list_pts_z)
float list_pts_x[], list_pts_y[], list_pts_z[];
int *color_index,*num_pts;
{
    Xgl_3d_ctx      *ctx;
    Xgl_color        color;
    Xgl_pt_list      poly_pts;
    int              i;

    /* allocate memory for the coordinates to be processed */
    if (!(poly_pts.pts.f3d = (Xgl_pt_f3d
*)malloc(*num_pts*sizeof(Xgl_pt_f3d))) {
        printf("\n \n memory allocation request in
        gal_filled_polygon failed\n \n"); exit(1); }

    poly_pts.pt_type = XGL_PT_F3D; poly_pts.bbox = NULL;
    poly_pts.num_pts = *num_pts;

    /* assign the vertices to their individual points */

    for ( i=0; i<*num_pts; i++) {

        poly_pts.pts.f3d[i].x = list_pts_x[i];
        poly_pts.pts.f3d[i].y = list_pts_y[i];
        poly_pts.pts.f3d[i].z = list_pts_z[i];

    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
        *color_index; } else color.index = *color_index;

    ctx = get_3d_ctx();

    xgl_object_set
    (*ctx,XGL_CTX_SURF_FRONT_FILL_STYLE,XGL_SURF_FILL_HOLLOW,XGL_CTX_SURF_FRONT_COLOR,
    &color, NULL);

    xgl_polygon (*ctx, XGL_FACET_NONE, NULL, NULL, 1, poly_pts);

    free(poly_pts.pts.f3d); }

```

```

/*****
 * gal_3d_filled_polygon()- draws a polygon from a list of 3d
 * points.
 *
 * procedures called - malloc, xgl_object_set, xgl_polygon
 *****/

void gal_3d_filled_polygon_(color_index, num_pts, list_pts_x,
list_pts_y, list_pts_z)
float list_pts_x[], list_pts_y[], list_pts_z[];
int *color_index, *num_pts;
{
    Xgl_3d_ctx *ctx;
    Xgl_color color;
    Xgl_pt_list poly_pts;
    int i;

    /* allocate memory for the coordinates to be processed */
    if (!(poly_pts.pts.f3d = (Xgl_pt_f3d
*)malloc(*num_pts*sizeof(Xgl_pt_f3d)))) {
        printf("\n \n memory allocation request in
        gal_filled_polygon failed\n \n"); exit(1); }

    poly_pts.pt_type = XGL_PT_F3D; poly_pts.bbox = NULL;
    poly_pts.num_pts = *num_pts;

    /* assign the vertices to their individual points */

    for ( i=0; i<*num_pts; i++) {

        poly_pts.pts.f3d[i].x = list_pts_x[i];
        poly_pts.pts.f3d[i].y = list_pts_y[i];
        poly_pts.pts.f3d[i].z = list_pts_z[i];

    }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
    *color_index; } else color.index = *color_index;

    ctx = get_3d_ctx();

    xgl_object_set
    (*ctx,XGL_CTX_SURF_FRONT_COLOR,&color, NULL);

    xgl_polygon (*ctx, XGL_FACET_NONE, NULL, NULL, 1, poly_pts);

    free(poly_pts.pts.f3d); }

/*****
 * gal_3d_filled_multisphere() - a primitive drawing routine to draw a
 * series of identically colored spheres.
 *
 * routines called - get_3d_ctx, xgl_object_set, xgl_multi_circle
 *****/

void gal_3d_filled_multisphere_(color_index,num_circs, rad, x, y, z)
int *color_index, *num_circs;
float x[], y[], z[], rad[];
{
    Xgl_3d_ctx *ctx;
    Xgl_color color;
    Xgl_circle_list circle_list;

```

```

int                i;

    /* allocate memory for the coordinates to be processed */

    if (!(circle_list.circles.af3d = (Xgl_circle_af3d
    *)malloc(*num_circs*sizeof(Xgl_circle_af3d))) {
        printf("\n \n memory allocation request in
        gal_3d_filled_multisphere failed\n \n"); exit(1); }

    /*
    * get a smooth circle */

    ctx = get_3d_ctx();

    xgl_object_set(*ctx, XGL_CTX_CURVE_APPROX, XGL_CURVE_METRIC_VDC,
                    XGL_CTX_CURVE_APPROX_VALUE, 0.1,
                    XGL_CTX_MIN_TESSELLATION, 25,
                    XGL_CTX_MAX_TESSELLATION, 30, NULL);

    /* draw the circles */
    circle_list.type = XGL_MULTICIRCLE_AF3D;
    circle_list.num_circles = *num_circs; circle_list.bbox = 0;

    for ( i=0; i<*num_circs; i++) {
        circle_list.circles.af3d[i].center.flag = 1;
        circle_list.circles.af3d[i].center.x = x[i];
        circle_list.circles.af3d[i].center.y = y[i];
        circle_list.circles.af3d[i].center.z = z[i];
        circle_list.circles.af3d[i].radius = rad[i]; }

    if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
    *color_index; } else color.index = *color_index;

    xgl_object_set (*ctx, XGL_CTX_SURF_FRONT_COLOR, &color,
    XGL_CTX_SURF_EDGE_FLAG, TRUE, XGL_CTX_EDGE_COLOR, &black_color,
    NULL);

    xgl_multicircle(*ctx, &circle_list);

    free(circle_list.circles.af3d);
}

/*****
*
* gal_3d_unfilled_multisphere_() - this procedure draws a wireframe
* "sphere" from a set of three normal circles sharing a common centre.
*
*****/

void gal_3d_unfilled_multisphere_(color_index, num_spheres, rad, x, y,
z)
int      *color_index, *num_spheres;
float    rad[], x[], y[], z[];
{
    Xgl_3d_ctx      *ctx;
    Xgl_color        color;
    Xgl_circle_list  circle_list;
    Xgl_circle_f3d    circs;
    int              j,i,k, count;

    /* define the circle planes */
    static Xgl_pt_f3d vector[5][3] = {{(1.,0.,0.), {0.,1.,0.),
{0.,0.,0.}}, {(1.,0.,0.), {0.,0.,1.}, {0.,0.,0.}}, {(0.,1.,0.),
{0.,0.,1.}, {0.,0.,0.}}, {(0.70711,1.,0.70711), {-0.70711,1.,-0.70711},
{-0.70711,0.70711,0.}}, {(0.70711,1.,0.70711), {0.70711,1.,-0.70711},

```

```

{0.70711,0.70711,0.}}};

/* allocate memory for the coordinates to be processed */

if (!(circle_list.circles.f3d = (Xgl_circle_f3d
*)malloc(5*(*num_spheres)*sizeof(Xgl_circle_f3d))) {
    printf("memory allocation request in
    gal_unfilled_sphere failed\n"); exit(1);
}

ctx = get_3d_ctx();

xgl_object_set(*ctx, XGL_CTX_CURVE_APPROX, XGL_CURVE_METRIC_VDC,
    XGL_CTX_CURVE_APPROX_VALUE, 0.1,
    XGL_CTX_MIN_TESSELLATION, 25,
    XGL_CTX_MAX_TESSELLATION, 30, NULL);

/* draw the circles */
circle_list.type = XGL_MULTICIRCLE_F3D;
circle_list.num_circles = 5*(*num_spheres);
circle_list.bbox = 0;

k = 0;
for (j = 0; j < circle_list.num_circles; j) {
    for (count = 0; count < 5; count++) {
        circle_list.circles.f3d[j].dir_normalized =
        FALSE; circle_list.circles.f3d[j].dir_normal =
        TRUE; circle_list.circles.f3d[j].center.flag =
        0;

        for (i = 0; i < 3; i++) {
            circle_list.circles.f3d[j].dir[i] =
            vector[count][i];
        }

        circle_list.circles.f3d[j].center.x = x[k];
        circle_list.circles.f3d[j].center.y = y[k];
        circle_list.circles.f3d[j].center.z = z[k];
        circle_list.circles.f3d[j].radius = rad[k];
        j++;
    } k++; }

if (get_dbuf_on()) { color.index = *color_index*COLOR_SIZE +
*color_index; } else color.index = *color_index;

xgl_object_set (*ctx,
XGL_CTX_SURF_FRONT_FILL_STYLE,XGL_SURF_FILL_HOLLOW,XGL_CTX_SURF_FRONT_COLOR, &color,
NULL);

xgl_multicircle(*ctx, &circle_list);

free(circle_list.circles.af3d);
}

```


References

- MACLEN86 MacIennan, B.J., Principles of Programming Languages, 2nd Ed., Holt, Reinhart and Winston Inc., Fort Worth, Texas, 1987.
- FOLEY90 Foley, J.D., A. van Dam, S.K. Feiner, J.F. Hughes, Computer Graphics : Principles and Practice, Addison-Wesley, Reading, Massachusetts, 1990.
- HELLER92 Heller, D., O'Reilly and Associates Inc. Volume Seven: Xview Programming Manual, O'Reilly and Associates , Sebastopol, California, 1990.
- HOLLUB87 Hollub, A.E., The C Companion, Prentice-Hall Software Series, Englewood Cliffs, New Jersey, 1987.
- KNUTH74 Knuth, D.K., "Structured Programming with GOTO Statements", Computing Surveys 6, December 1974, pp.261 - 301.
- KNUTH89 Knuth, D.K., "The Errors of T_EX", Software - Practice and Experience 19, John Wiley and Sons, New York, New York, 1989, pp. 607-685.
- LOUK90 Loukides, M., Unix for FORTRAN Programmers, Nutshell Handbooks, O'Reilly and Associates , Sebastopol, California, 1990.
- POUNTA89 Pountain, D., "The X Window System", Byte, January, 1989, pp. 353.
- JONES89 Jones, O., Introduction to the X Window System, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.
- SUN90 Sun FORTRAN User's Guide, Part No: 800-3417-10, Revision A, 16 March, 1990, Sun Microsystems Inc., Mountain View, California, 1990.
- TORBO88 Torborg, J.G., "A Parallel Processor Architecture for Graphics Arithmetic Operations", Computer Graphics, Volume 21, 4, 1987, pp. 197-204.

- XGL91 XGL 2.0 Reference Guide, Part No: 800-5732-10, Revision A, 4 October, 1991, Sun Microsystems Inc., Mountain View, California, 1991.
- XGL91A XGL 2.0 Software Installation Guide, Part No: 800-5733-10, Revision A, 4 October, 1991, Sun Microsystems Inc., Mountain View, California, 1991.