Integrating HotViews and OpenGL

by

Michael C. Johnson

Thesis submitted to the Graduate Faculty of

Christopher Newport University in partial

fulfillment of the requirements

for the degree of

Master of Science in

Applied Physics

2000

Approved :

David Heddle, Chairman _____

David Doughty _____

John Hardie _____

Robert Hodson _____

ABSTRACT

Integrating HotViews and OpenGL

Michael C. Johnson

Master's Degree in Applied Physics. 2000

Dr. David Heddle, Associate Professor of Applied Physics

Scientific and engineering applications have a unique set of requirements. They must be able to draw complex machinery and allow the user to interact with it, present complex data in an intuitive manner, and provide for controls and user input. Software libraries are helpful in the creation of such applications because they provide code for accomplishing common tasks, such as graphing data or handling user input. They remove a level of complication from the application code and allow the programmer to concentrate on the technical core of the project.

HotViews is a C library for creating controls and data visualization programs. It provides the programmer with real-world coordinate tracking, many kinds of interface widgets, help windows, graphs, and much more. HotViews has been used to create a detector display for the CLAS detector at Jefferson Lab. It has also been used in commercial applications.

OpenGL is a 3D graphics library specification, with many implementations on several platforms. OpenGL provides an interface to basic 3D graphics tools, such as frame and auxiliary buffers, shading and light sources, and textures. OpenGL provides for the use of optimized 3D display hardware. Applications can benefit from using 3D graphics because they can more accurately simulate the real world.

This paper details the creation of a new version of HotViews which allows software writers to include 3D graphics in their GUIs, using OpenGL as a rendering engine.

Dedication

To my wife, Angela :

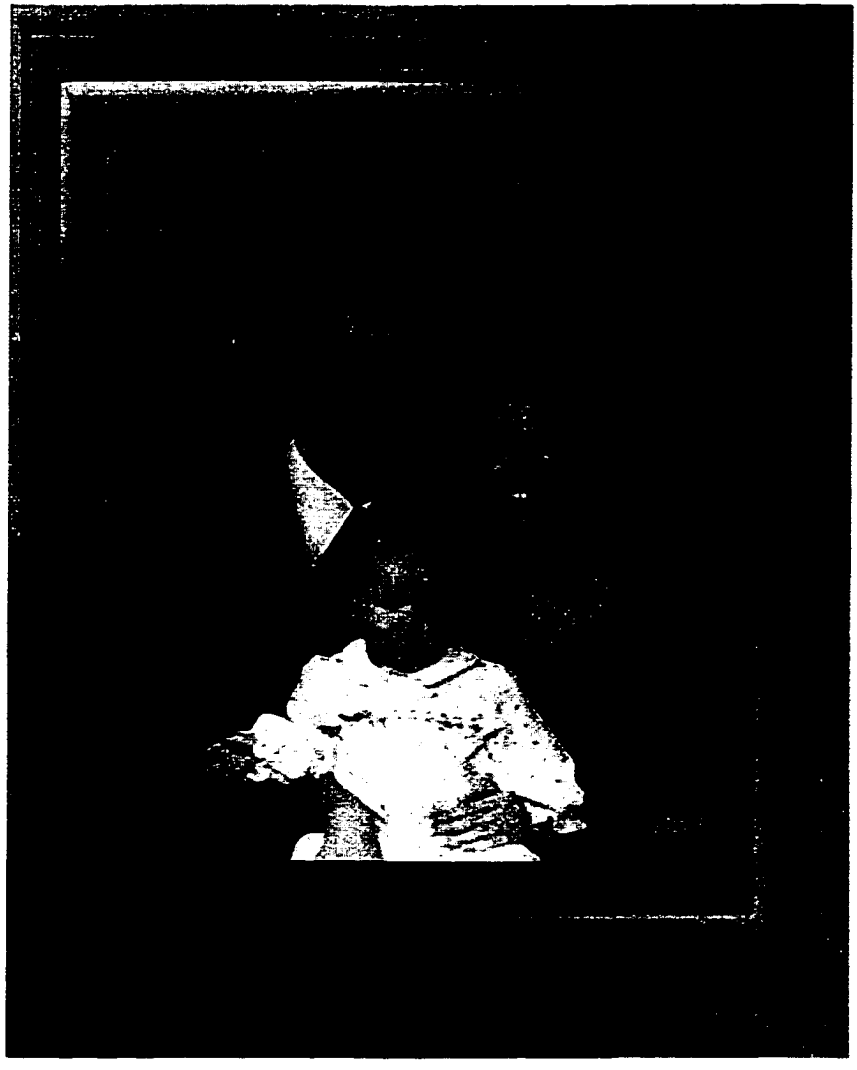
Believe in yourself

as you have believed in me

and you can do anything

And to my daughter, Ellissa :

It's never too early to start writing your thesis



Acknowledgements

Many thanks are due to Dr. David Heddle for patiently working with a student who juggles work, school, and a family. My appreciation also goes out to Jefferson Lab and to my boss, Dieter Cords, for supporting me in my pursuit of the Masters Degree.

Table of Contents

List of Tables

List of Figures

1. Introduction

This paper describes the creation of a graphics library and framework to assist programmers in writing scientific controls and data visualization applications. The most efficient way to do this is to integrate separate libraries, each of which contains some functionality that we need. HotViews provides tools for creating a technical application complete with multiple views of a device, plots, help messages, coordinate feedback, and controls all neatly tied together in a single system window. OpenGL is a 3D graphics library with support for many visual effects and drivers for specialized graphics hardware. Integrating these two graphics libraries will facilitate the creation of scientific programs which can display complex information in an intuitive manner.

The central problem in this process involves rendering conflicts between HotViews, OpenGL, X Windows, and Motif. X Windows provides a window manager and a universal interface to the underlying simple 2D graphics hardware. It controls which applications can render to what areas of the display screen, and specifies how they do so. HotViews makes all of its 2D drawing calls to X, and makes some calls to Motif for displaying menus and dialogues. OpenGL is typically used alone, with no other graphics libraries, for the display of 3D scenes and objects. OpenGL applications typically work in full screen mode, where the window manager is temporarily shut off, and render directly to the screen or to a single window exclusively. OpenGL works with X through GLX, an X Window extension. This extension allows OpenGL to render into a single window or pixmap. OpenGL operations, especially animation, tend to overwrite any other pixels within the same window. We resolve these conflicts and retain each library's individual functionality.

In order to do this, we carefully initialize all libraries to pick the same display resources and utilize the same colormaps. We take advantage of OpenGL's ability to render to a window or pixmap to allow it to create 3D graphics separate from the graphics put down by other drawing routines. We choose our windows and other settings carefully so that each library renders to the proper screen areas, even when parts of an item are occluded.

2. Motivations for Hv3D

Scientific and engineering applications provide a special challenge to programmers. They usually require the display of geometrically complex schematics along with numerical or graphed data. It is usually necessary for the application to allow a certain amount of interactivity with the rendered scenes. In many cases, the programmer is one of the end users of the application. Figure 1 provides an example of an interface to a scientific application.

Physics experiments provide several examples. The experimental physics community is continually coming up with packages to aid in the quick development of technical applications, for example EPICS and ROOT. In scientific environments, the manpower to develop software from the ground up in a low level graphical environment such as X Windows is missing. The time to develop software is also constrained by the experimental schedule. As detectors and systems are constructed and tested, control software is often needed early in the process. Data visualization software is used during experiments to verify and test the detector systems, even before experimental results are sought.

3D graphics libraries and APIs provide a more physically realistic display of any real object or scene than could be achieved using simple 2D drawing routines. Animated 3D graphics go a step further, giving the user the ability to interact with complex scenes by zooming and changing viewing perspectives. Rendering 3D graphics requires an application to track coordinates in a three dimensional Cartesian system and to map those coordinates to pixels on a two dimensional display. In order to render a realistic and visually pleasing scene, however, much more may be required. Modern 3D hardware and libraries provide for numerous visual effects, such as lighting and textures.

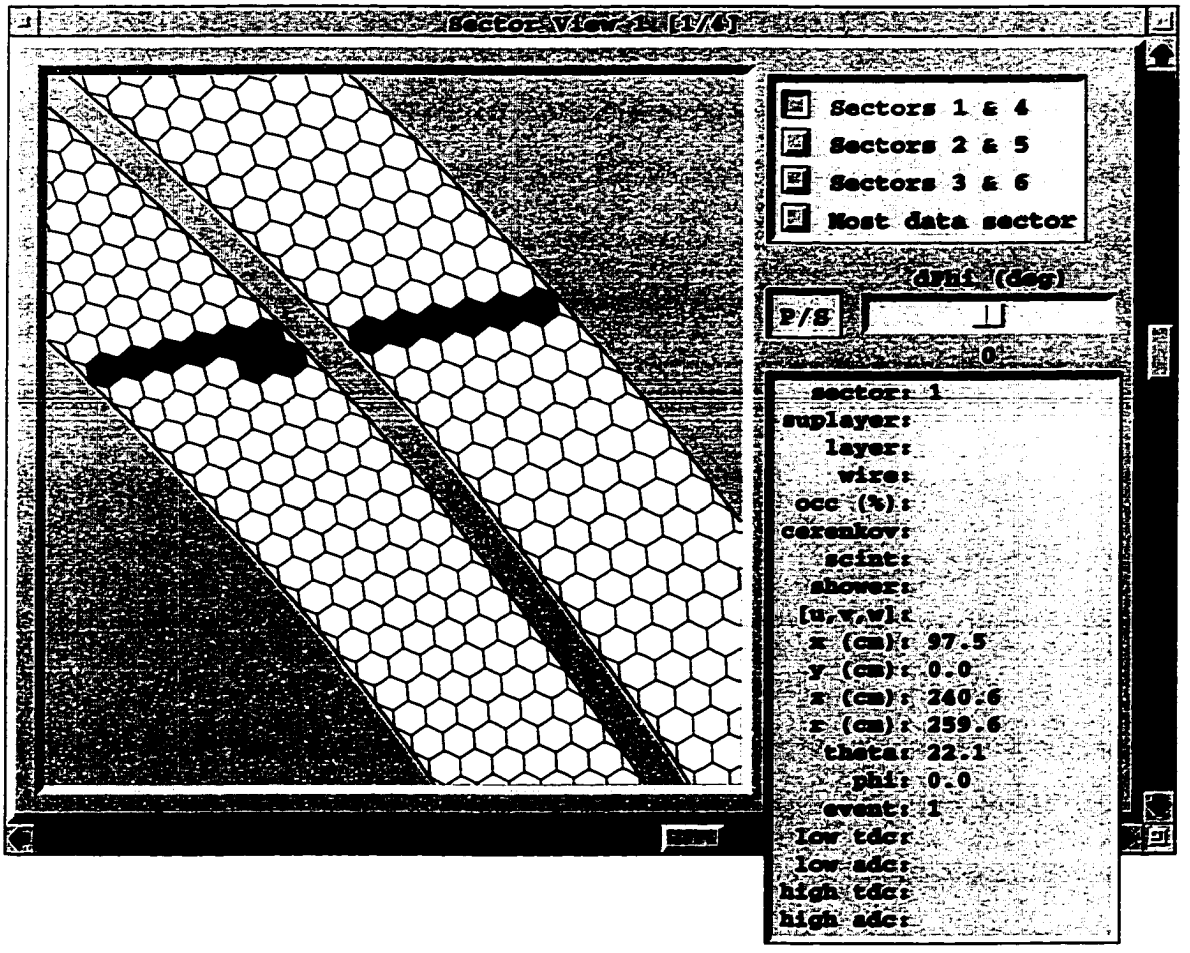


Figure 1 View of a drift chamber from a physics detector

2.1. Application Interface Development

A programmer has a wide variety of applications, libraries, and languages available to ease the development of application interfaces. The purpose and audience of the application can affect the choice of tool, as can the system and manpower available to code it. Once chosen, the tool or library should enable the programmer to concentrate on implementing the core purpose of the application. For scientific and engineering applications, this may be the coding of a complex data processing framework, or the description of a complex detector.

It is important, once data is obtained from a simulation or experiment, to display it in a useful manner. We may need to utilize charts, histograms, and other types of plotting tools to achieve

this, as shown in Figure 2. For complex detectors and other systems, it is also important for the user to be able to view and change the configuration of the system. This may be easier on the user if the program provides a picture of the physical system itself. The interface to a technical program may have schematics and other illustrations of the equipment, as well as graphical controls and fields to change their configuration.

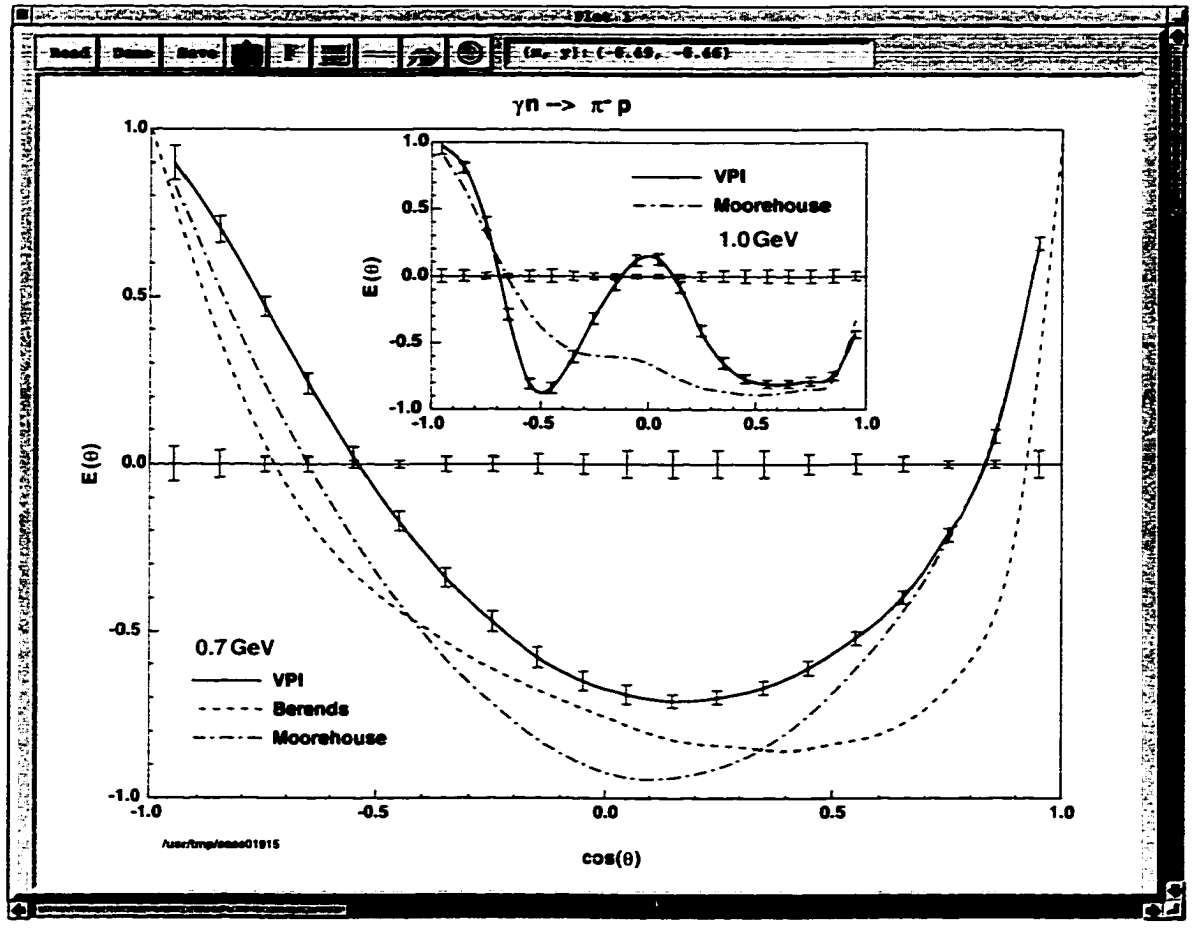


Figure 2 hvplot, a data plotting application created using Hv

2.1.1. Interface Building Tools

RAD tools allow one to quickly assemble a user interface, often using a drag-and-drop interface, with customizable control widgets such as scrollbars, dials, buttons, text fields, and so on. Figure 3 shows a simple GUI. Often there are a few fairly sophisticated but specialized complex widgets available, such as graphs. But most often these widgets are not engineered for a data

intensive application. For example, a graph widget in such an application may become bogged down after only a few hundred data points have been added. These packages may add flexibility to the application by producing actual text source code, which can be altered by the programmer directly, allowing one to create a specialized application. This code tends to be large, however, and hard to maintain. Drawing a complex object, such as a physics detector, can be a foreboding task using such tools, as every piece must be drawn manually. [1]

More specialized environments exist for the development of controls and data acquisition applications. Labview, for example, provides a graphical programming environment and device drivers. One can link various modules together to create automated systems in a short amount of time. The display for monitoring such a system tends to be very simplistic. Although this is appropriate for a cryogenic target, for example, it fails to render very complex scientific equipment. The graphical language produced is very inflexible and makes heavy use of system memory and resources. One cannot expand on the application produced by calling outside libraries. [2]

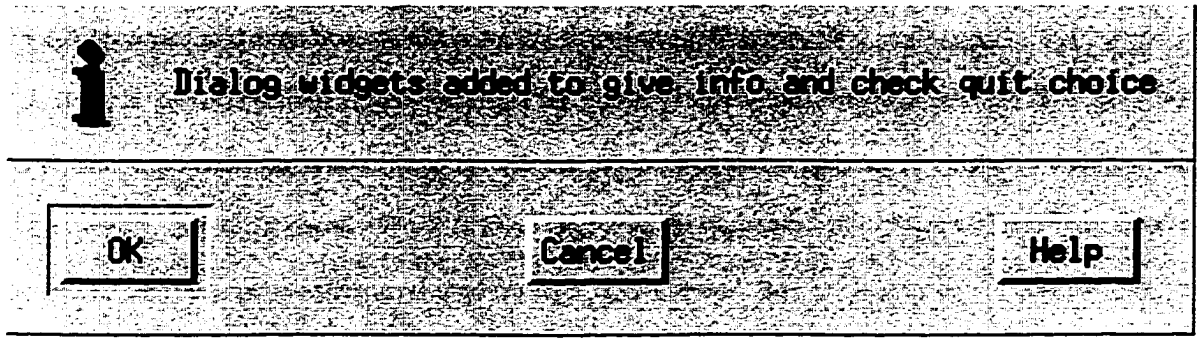


Figure 3 Sample GUI with buttons and text

2.1.2. Interface Building Libraries

Programming at a lower level provides us with more flexibility but may require more time for development. This can be greatly mitigated by using a function library. A general purpose language, such as C, has several advantages :

- Portability : Our program may be ported between different operating systems if we carefully choose our API and utility libraries. Such libraries must exist on more than one platform for this to work.

- Performance : An editor and compiler (or SDK) gives us the greatest control over the response time and execution time of our program. We can find and eliminate bottlenecks, create custom data structures, and streamline code as needed.
- Flexibility : Physics experiments and other complex systems require interfaces that are completely unique. We can choose to make calls to low level graphics routines, or higher level widgets wherever appropriate.

But if we don't wish to be bogged down by details outside the purpose of the application, we need to pick a graphics library to assist us. Our chosen libraries should be able to build GUIs, or graphical user interfaces, and draw arbitrary items that we describe.

ROOT is a C++ class library developed at CERN for large scale data analysis programs. These classes provide tools for statistical analysis and other processes used in a scientific environment. The ROOT framework also comes with some useful development tools, such as a C++ interpreter and an object oriented database. For GUI building, it provides arbitrary wrappers to the Motif widgets and graphing functions. For detector description, it has an interface to OpenGL for 3D graphics and functions containing 2D drawing primitives. [3]

Java is an interpreted object oriented language created and standardized by Sun Microsystems, and supported on a number of platforms. Java provides a number of class libraries containing graphics and GUI tools. Work is even ongoing to support a 3D graphics interface through OpenGL. Java was originally developed for web applications, but has expanded to general purpose programming. [4]

These environments provide us with many tools for data processing and communication, but only supply the basics for building interfaces. In order to create an interactive display of a system, we would have to call basic 2D drawing routines and write our event handlers from scratch. Only basic interactive widgets would be provided for us, such as buttons and text inputs. Multiple windows would be handled by the system, not the library, which may not be very efficient if the application uses many windows. ROOT provides copious graphing abilities, while Java does not.

While developing for Unix and the X Window System, we may choose to use the X (Xlib), X

Toolkit (Xt), and Motif (Xm) libraries. X provides basic drawing and windowing functions. Xt provides the ability to create graphical widgets and callback functions to handle user input events. Xm provides a generous set of widgets that the programmer or user may customize. One could build an application making calls to the Xlib library for drawing graphics, and to Xt and Motif for building a GUI. This would give us a fine level of control over the style and performance of our graphics, but we would have to build our graphical objects from scratch. In addition to drawing the objects, we would need to code the program responses for interacting with these objects.

2.2. 3D Graphics

In order to render 3D shapes in a pleasing and useful manner, we must be able to define primitive shapes in a three dimensional coordinate system. These shapes may be polygons, polyhedrons, or something more sophisticated. In order to easily create or animate complicated shapes, we must also be able to scale, translate, and rotate our objects once they are defined. When using a low level graphics library, we build solid shapes out of connected polygons by defining their vertices. These shapes may be gathered together into composite objects for easier handling in our code. A high level library might allow us to build shapes out of a menu of solid objects, and might also provide routines for handling the shapes and accepting user input. This type of solid modeling makes it easier to create scenes quickly, but the basic computations themselves are almost always done on polygons, so a low level library like OpenGL passes information in terms of vertices of polygons used as the boundaries of solid objects. Regardless, it is important that our library provide routines for performing the matrix algebra required to manipulate objects as described above, and to map three dimensional coordinates to a two dimensional screen.

Some special effects must also be supported in order to create a realistic scene. In addition to rendering basic shapes in various colors, a good graphics library will add shading, light source effects, fog, alpha blending, antialiasing, and textures. Shading and light sources make subtle changes to the color of an element depending on its location from and angle to the viewer. Fog and alpha blending allow the mixing of pixels on the screen, which allows one to draw transparent and blurry shapes. Antialiasing is a method used to correct the jagged edges which may appear

when converting an analog line (defined by real number coordinates) into discrete pixels. This can be accomplished by selectively blurring adjacent pixels. Textures are flat pictures that can be transposed onto the sides of various shapes. Each of these adds realism to a scene and can help to make it visually pleasing. This list is not exhaustive, but these are the most commonly provided and often used special effects.

The graphics pipeline, shown in Table 1, describes the steps involved in processing 3D graphics information. These stages are mostly independent and each stage may execute in parallel. This allows for the creation of dedicated hardware which takes advantage of this parallelism. While all stages may be executed through software on a general purpose CPU, the trend is for more and more stages to be processed by special purpose hardware on specialized video cards. Such cards are available on high end workstations such as those made by SGI, but the technology also has a tendency to trickle down into lower cost PC cards. [5]

Stage	Output	Description
Pre-Pipeline	Vertices in 3D coordinates	Application defines objects
Modeling Transformations	Vertices in 3D coordinates	Objects transformed from independent systems to world coordinates
Trivial Rejection	Vertices in 3D coordinates	Certain shapes, such as back facing polygons, are dropped at this stage.
Illumination	Vertices in 3D coordinates and colors	Object colors are created using material properties and light sources
Viewing Transformation	Vertices in 3D coordinates and colors	Objects are transformed from world to eye coordinates
Clipping	Vertices in 3D coordinates and colors	Objects not within the defined viewing frustum (box) are dropped
Projection	Vertices in 2D coordinates and colors	Objects are transformed from 3D to 2D coordinates
Rasterization	Pixels	Objects are interpolated into pixels
Display	Light	Pixels are sent to a display device

Table 1 The Graphics Pipeline

The pipeline can be described starting with the 3D object information, typically defined in terms of vertices, polygons, and colors, and moving through stages until this information is changed into pixels for display. Most of these stages involve changing coordinate systems through matrix multiplications called transformations. The modeling transform takes care of reorienting

objects from their own coordinate system to the world coordinate system. Some objects may be immediately discarded at this point since they are out of view. Lighting calculations are done to illuminate objects. A viewer coordinate and orientation is used to transform all objects into the coordinate system of the viewer. More objects may be clipped at this point. The objects are then transformed once again into the 2D coordinate system of the screen itself. Finally, the graphical data is converted into discrete pixels and sent to the display area or frame buffer. The most current graphics hardware can perform operations for all of these stages.

Besides having dedicated processors, 3D graphics cards may also have special buffers providing special effects such as those shown in Figure 4. These operations are often referred to as pixel operations, in contrast to the vertex operations described above. These can include a double buffer, for rendering to an invisible buffer which is flushed to the screen when complete. This prevents the viewer from seeing partially completed pictures. A depth buffer keeps track of the Z coordinate of each pixel, and ensures the correct display of overlayed objects. Stencil and accumulator buffers can be used to paint special effects on some areas of the screen while leaving other areas alone. In addition, 3D graphics cards may have dedicated memory for textures. [6]

More work is being removed from the central processor all the time. Unless the programmer knows exactly what system the software will run on, it is impossible to know which stages should be executed by software and which will run on dedicated hardware. A 3D graphics API like OpenGL takes care of this for us. It contains routines to emulate all stages in software, but special device drivers will move the responsibility to 3D hardware if it is supplied. [5]

2.2.1. 3D Graphics Libraries

For an application to render 3D graphics well, it often relies on the presence of specialized video hardware. This creates two problems : it requires specialized code to take advantage of a graphics device's special abilities, and maintaining portability across platforms and graphics devices requires completely different code or device drivers in each case.

We need a 3D graphics library which is supported on a number of platforms, and which contains device drivers for many 3D video cards. In order to take advantage of a card's abilities,

however, we need to give up as much of the coding for the 3D graphics as possible to the library. In other words, we want to push graphics data into the top of the aforementioned 3D graphics pipeline and let the library handle all steps up to the display. This makes our coding even simpler. We simply describe what we want in the terse terms of colors and polygon vertices, and let the library handle the rest. Our program is left to concentrate on scientific coding. [5][7]

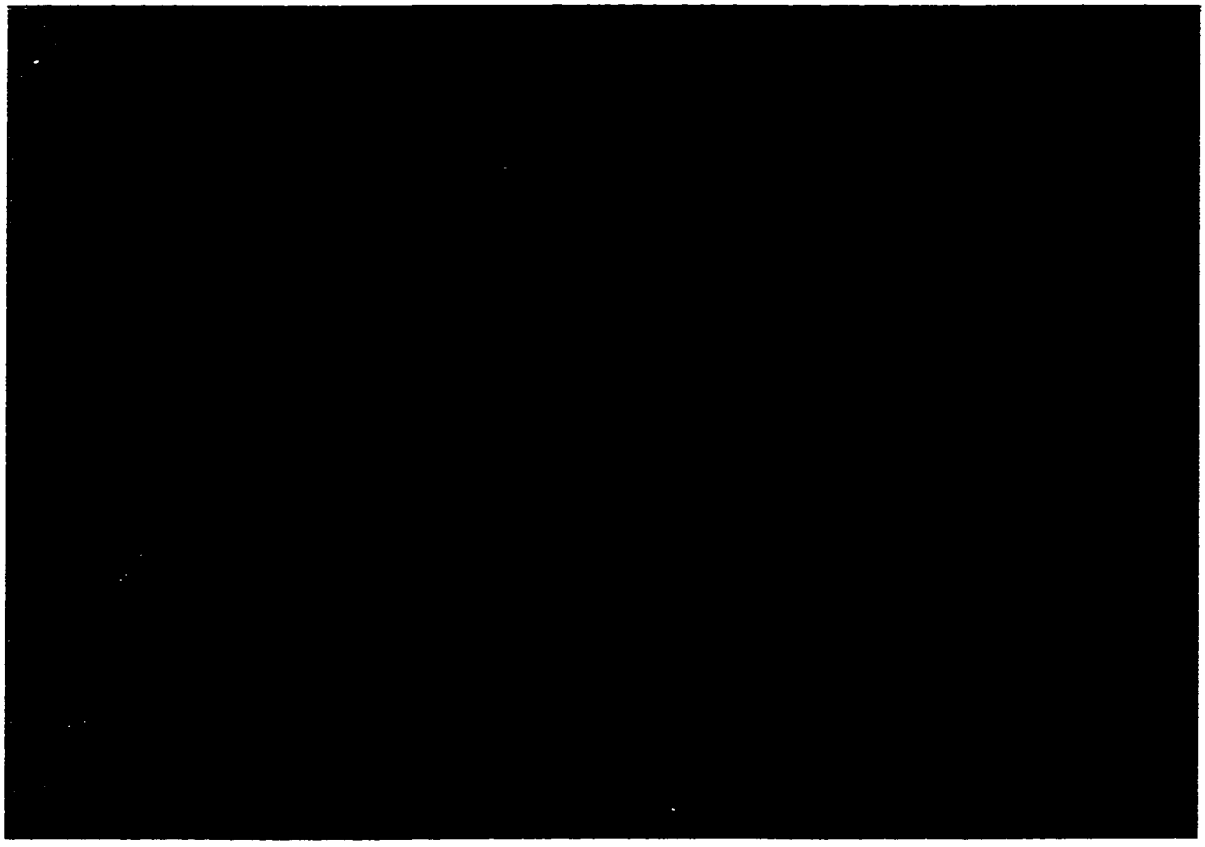


Figure 4 Sample 3D scene with reflection effect achieved through alpha blending

2.2.2. OpenGL

OpenGL is a 3D graphics library. The OpenGL specification describes an API to a function library and state machine, which together describe an interface to an underlying 3D graphics architecture. The OpenGL API has been referred to as an architecture, highlighting the fact that different implementations of the API may result in different levels of performance, cost, and capabilities. It is a widely accepted standard and is used by many applications. Documentation is available detailing every part of the library. [6]

Implementations of OpenGL are available on almost all Unix platforms, as well as Microsoft Windows and other systems. OpenGL achieves this level of interoperability by carefully defining the API to be system independent. An API typically describes a software interface to some underlying system, such as a specific kind of hardware or operating system. The interface is typically described by a group of functions in a library. Glide, for example, is a library of functions for interfacing with 3D graphics cards created by 3Dfx Interactive, Inc. The POSIX 1003.1 API gives a list of C functions for performing many standard system tasks within an operating system. The OpenGL standard also specifies a library of functions and calling standards, but does so in a platform and hardware independent manner. There is no standard implementation to borrow code from, and function inputs, outputs, and states are carefully and completely specified. The operational implementations of the functions are left unspecified. [8][9]

OpenGL was introduced by SGI Inc. in 1992. The standard is maintained by an independent consortium with members from many companies in the computer graphics industry. A conformance test suite is available to ensure that implementations are truly compatible with the standard.

Many of the alternative 3D libraries are connected closely to one particular platform. PEX, a library based on the PHIGS library, is an extension to the X Window System. Code compiled using this library necessarily only runs in the X Window System, typically a Unix system. And not all releases of X come with this library. If we wanted to develop for Microsoft Windows we could use DirectX / Direct3D. But if we want our code to be portable across many platforms and

to support many kinds of graphics cards, OpenGL is practically the only choice. Figure 5 provides a sample of OpenGL's rendering abilities.

Figure 5 A 3D object created with OpenGL

2.3. HotViews

Scientific applications often require data visualization tools, such as rainbow charts and plots, and GUI building widgets, such as buttons and sliders. HotViews provides a large selection of tools for these purposes. Built in graphical items provided include buttons, selection boxes, text, LEDs, option boxes, rainbow scales, sliders, and wheels. HotViews, or Hv, also supports graphing and visualization through coordinate tracking and copious drawing routines. [10][11]

But Hv also provides a unique look and feature set specifically useful to scientific and engineering applications. Some features include : [10][11]

- Items may have drag and drop ability and may be moved and resized through simple cursor actions.
- Basic 2D drawing tools are available from the library and may be made available to the final application itself.
- A scientific plotting tool is provided which supports spline smoothing and curve fitting.
- Support for animations and simulations is provided through timed callbacks.
- Balloon and menu based help may be easily integrated into the final application.
- There are many fonts and colors to select from.
- Function keys may be mapped to user functions
- A virtual desktop allows an application to take up a much larger working area than is provided on the display device.

Hv provides a managing window with some features of a Macintosh desktop, such as a single menubar. This main window contains Views, which are subwindows which may be dragged, resized, and hidden. Views provide functionality for pointer tracking (feedback), scrolling, zooming, zoom-to-fill, control widgets, and more. Figure 6 displays a single Hv View. Within the main drawing area, called a HotRect, an application can provide the user with the ability to drag and drop items, or to find an object's position in real world, not pixel, coordinates. These and other features help the programmer easily provide the user with the ability to interact with complex

objects and data sets. [10][11]

Hv does not make special provisions for 3D graphic rendering within Views or HotRects.

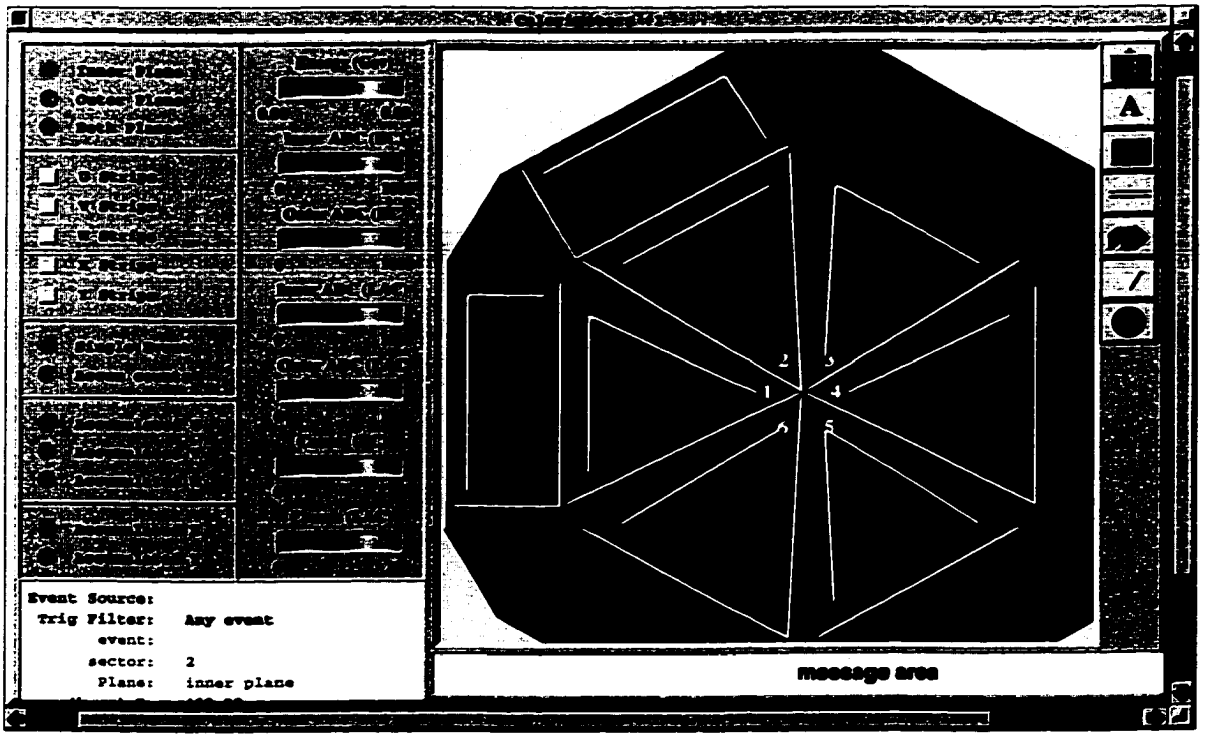


Figure 6 View of a detector with controls and rainbow bars

2.3.1. Example HotViews Applications

HotViews was developed for scientific and engineering applications which produce graphical output and allow the user to interact with it. This may be achieved directly by clicking on representations of real world objects, or indirectly through buttons and controls. Several applications which use Hv help to illustrate the use of its unique features. [11][10]

- **CED** was developed for Jefferson Lab for the display of the CLAS detector. CED, pictured in Figure 7, provides for several different views of the various components of this detector. One unique feature of the application is the ability to display detector data over the diagram of the detector itself, providing the user with a visual representation of the data. This data can come from raw or processed data files, or even live from the detector itself through Hv's simulation callbacks.

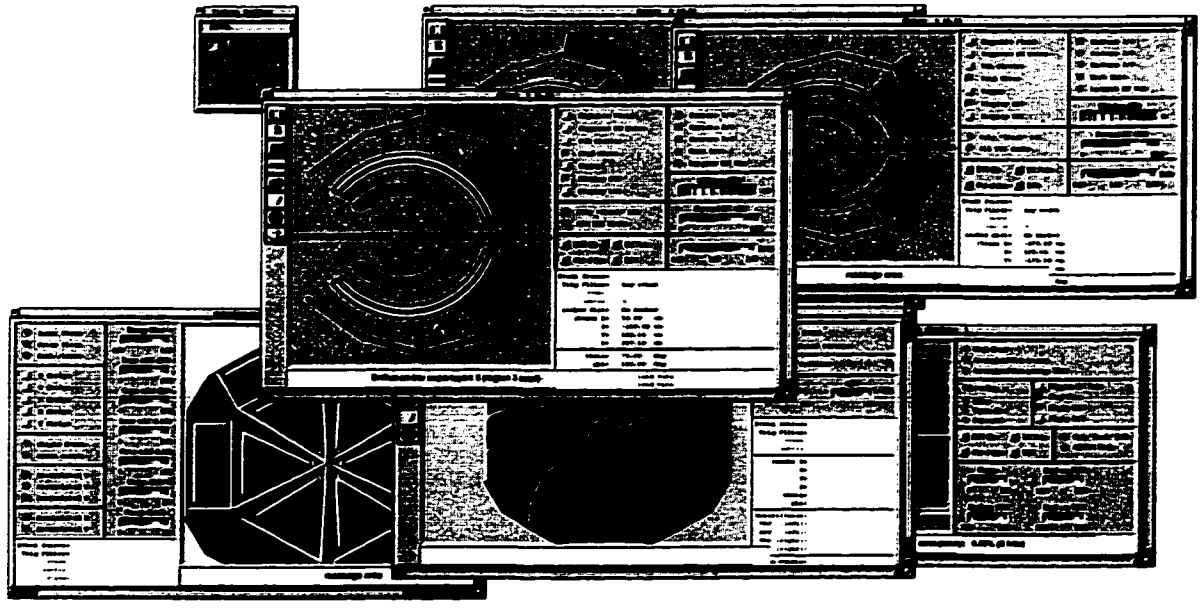


Figure 7 CED : Clas Event Display

•**TIGRIS** was also developed for the CLAS detector, but for the purpose of specifying trigger parameters for the detector's readout electronics. A user can actually click on different detector components, specifying their significance to the current experiment. The application turns these graphical representations of the experiment into data for the trigger electronics. TIGRIS is pictured in Figure 8.

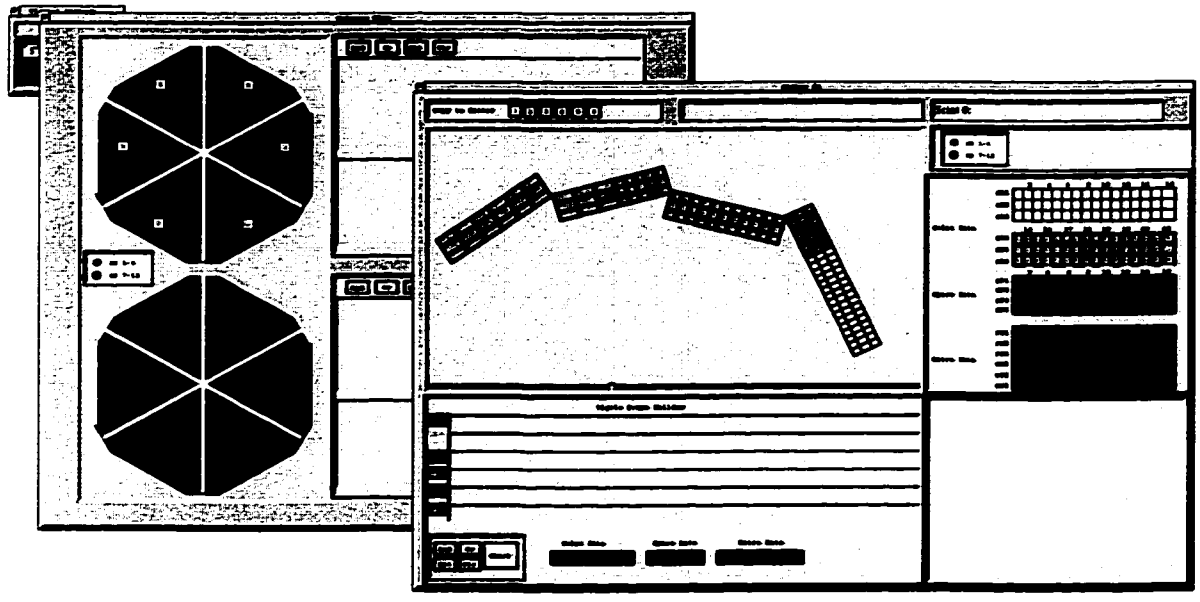


Figure 8 Tigris : Trigger Interactive Graphics

•**CAPS**, pictured in Figure 9, is a commercial application using the Hv library. SPARTA, Inc. of Mclean VA has developed a theater missile defense simulation by combining a FORTRAN engine with an Hv graphical interface. Objects such as radars and interceptors are placed on maps and can then be dragged and rotated. Once the Hv interface has established a scenario, it sends the information to the FORTRAN simulation and, upon completion of the simulation, graphically displays the results. The user can then interact with the results through the Hv feedback mechanism.

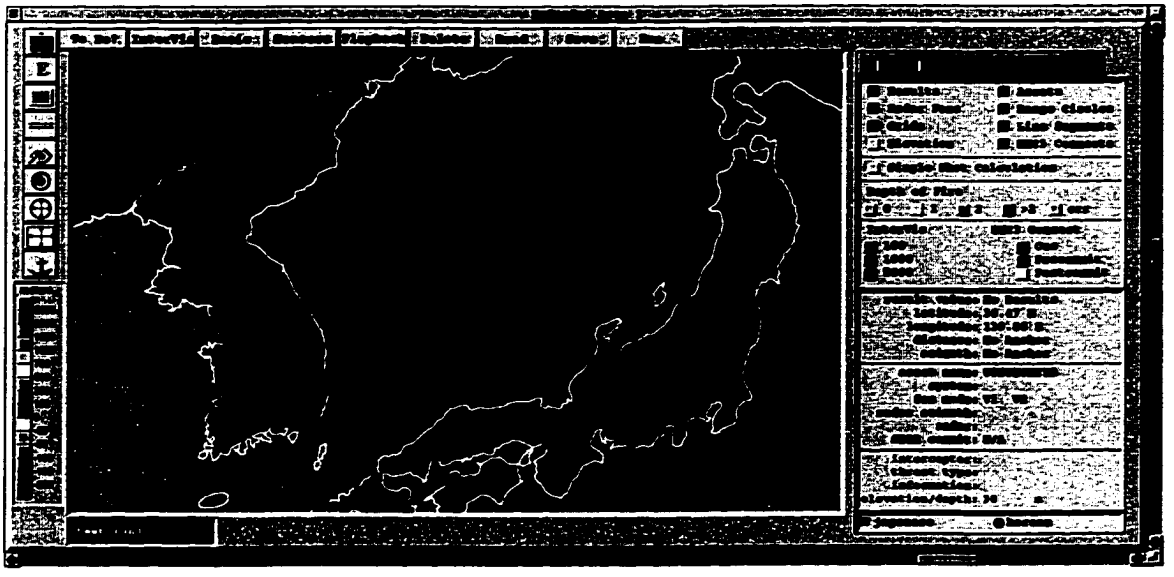


Figure 9 CAPS

2.4. Advantages of Integrated 2D/3D Graphics in HotViews

Once we enable HotViews to draw 3D shapes within a View, and incorporate controls and feedback into an application, we will have a useful display for complex physics detectors and other systems. Many applications require a complex set of choices to be presented to the user at once. It does not necessarily suffice to assume that the user knows of all possibilities and can dig through hierarchical windows or menus to get to the appropriate option. A particular example is the control room environment of a physics detector, where an operator might need to be presented with a physical view of a device, scales, graphs, and numerical feedback. Any one of these items going to a certain state or out of a particular range could give the user an indication of abnormal behavior

in the system. The integrated nature of Hv allows the application to display large amounts of graphical information at once.

To this we would add a three-dimensional picture of the system. A solid object based 3D picture of the system can be recognized more quickly. CAD applications, for example, have to draw items in 3D. The ability to rotate and scale 3D scenes gives us another unique feature. If we change the orientation and perspective of the viewer, we can give the user a feel of travelling through a 3D environment. This can make it easy for the user to find a particular component or location that he is looking for. 3D graphics simply give a more intuitive representation of any real object. Figures 10 and 11 show the difference between a real detector system and a typical 2D schematic of that system.

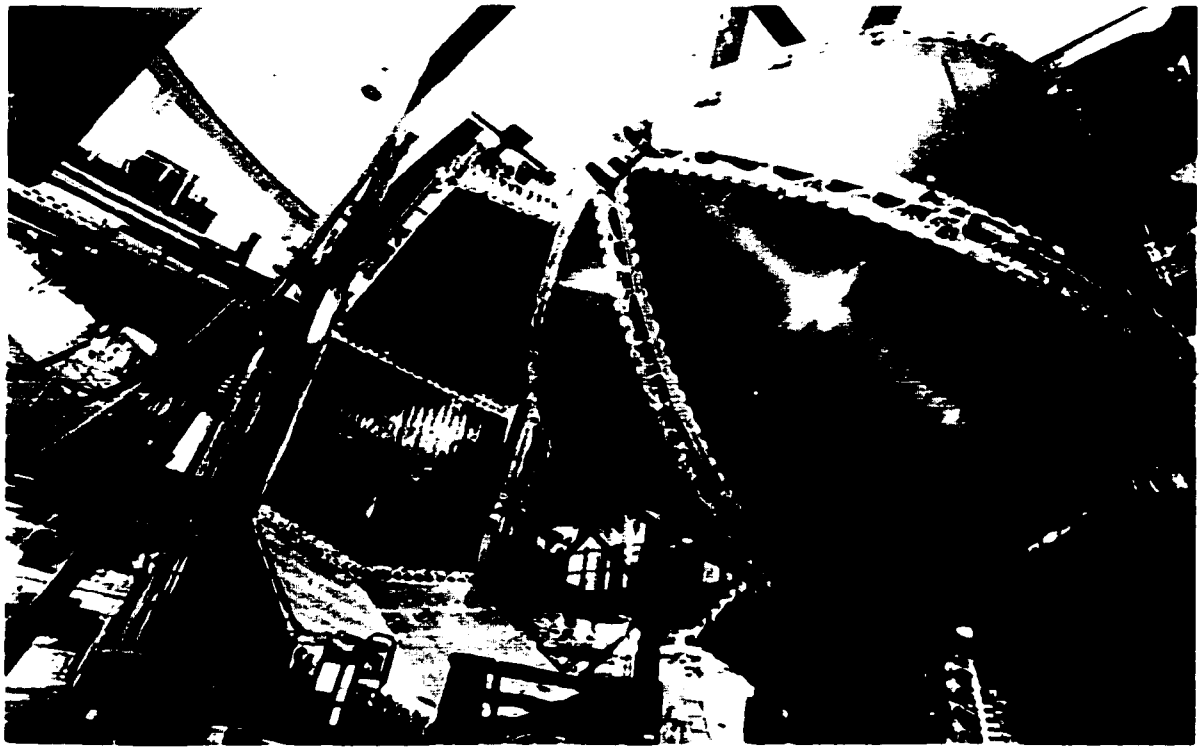


Figure 10 Photo of the Hall B drift chambers

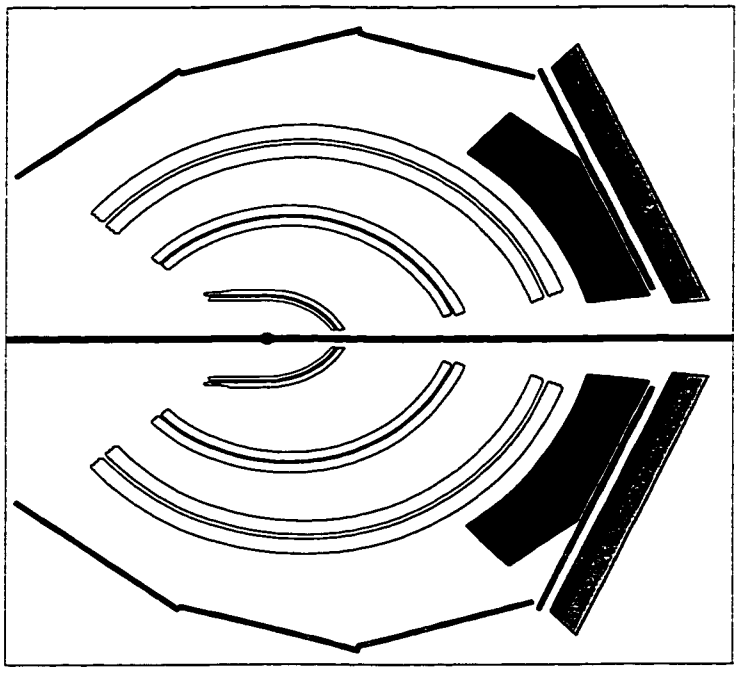


Figure 11 2D Schematic of the Hall B drift chambers

3. A 2D/3D Graphics Framework for Scientific Environments

Complex scientific and engineering applications require specific graphics libraries. They require functionality for building GUIs, displaying complex data, and interacting with complex renditions of equipment or environments. Basic GUI and graphics libraries are not adequate to the task (see 2.1.1 & 2.1.2). HotViews is useful in these environments, providing a framework and features needed by many applications in these environments (see 2.3 & 2.3.1). But a 3D display of the desired components or environment will add depth and information that the user can benefit from (see 2.4). Using a new version of HotViews integrated with OpenGL will give applications portability, since each library is available on many platforms. And we will retain most of the benefits of using either library on its own. (see 2.2, 2.2.1, 2.2.2, & 2.3).

3.1. Solving Rendering Conflicts from Multiple Sources

If the specific problems of 3D graphics rendering, widget behavior, and 3D graphics hardware support are being handled by our chosen graphics libraries, our problem becomes getting these independent programs to cooperatively render to the same graphics device. There are three separate rendering engines to be concerned with here :

- X Windows : This includes our windowing system and window manager. All requests to render in an X system are done through the X Windows protocol, either through Xlib or a library extension. X therefore controls what rendering is allowed, taking care that occluded or iconized windows do not overwrite any other part of the screen. Full screen rendering is not typically allowed.
- OpenGL : The OpenGL API intentionally leaves any integration with the target system's windowing system unspecified. This keeps the API platform independent. An OpenGL implementation then has to provide for communication with the underlying windowing system in addition to implementing the API's basic abilities. For X Windows, this is done through an extension to X : the GLX extension. In its most trivial implementation, GLX may simply convert all OpenGL rendering requests into pixels and then make drawing requests to Xlib.

A better implementation, however, will actually extend the X protocol to include OpenGL's own 3D structures.

- HotViews is our 2D rendering engine, and will make calls directly to X to render controls, views, items, and so on. It also uses Motif for the display of dialogues and menus.

Many of OpenGL's special effects involve operations which require changing the brightness of a particular color. This means that the library must have access to a color scheme which allows one to numerically adjust the brightness of a given color to an arbitrary degree, such as an RGB colormap. If the color scheme relies on a lookup index, where the pixel value is an arbitrary number, then math operations on those pixels are not useful. When using an RGB or even a grayscale colormap, however, increasing or decreasing a certain number changes its brightness. In another example, colors can be mixed by averaging the values of two pixels. For 3D graphics it is very desirable to have an RGB color scheme.

3.2. Proof of Principle

A sample application is needed to demonstrate the ability to render 3D graphics from within a HotViews application. Since all OpenGL applications take the same basic steps for initialization, and since creating objects for display is done independently of the interface to the windowing system, it should be sufficient to create an Hv application with one or more views containing 3D objects. There are several well known simple 3D objects which demonstrate several aspects of OpenGL programming. These objects are often included in demonstration programs released with implementations of the OpenGL library. If we can embed some of these applications into an Hv framework, we will have demonstrated a method for integrating OpenGL applications with Hv. We chose to implement the objects shown in Figures 12, 13, 14, and 15. To do this, we did not create controls and feedback, although we could easily add them to a serious application and retain our 3D effects.

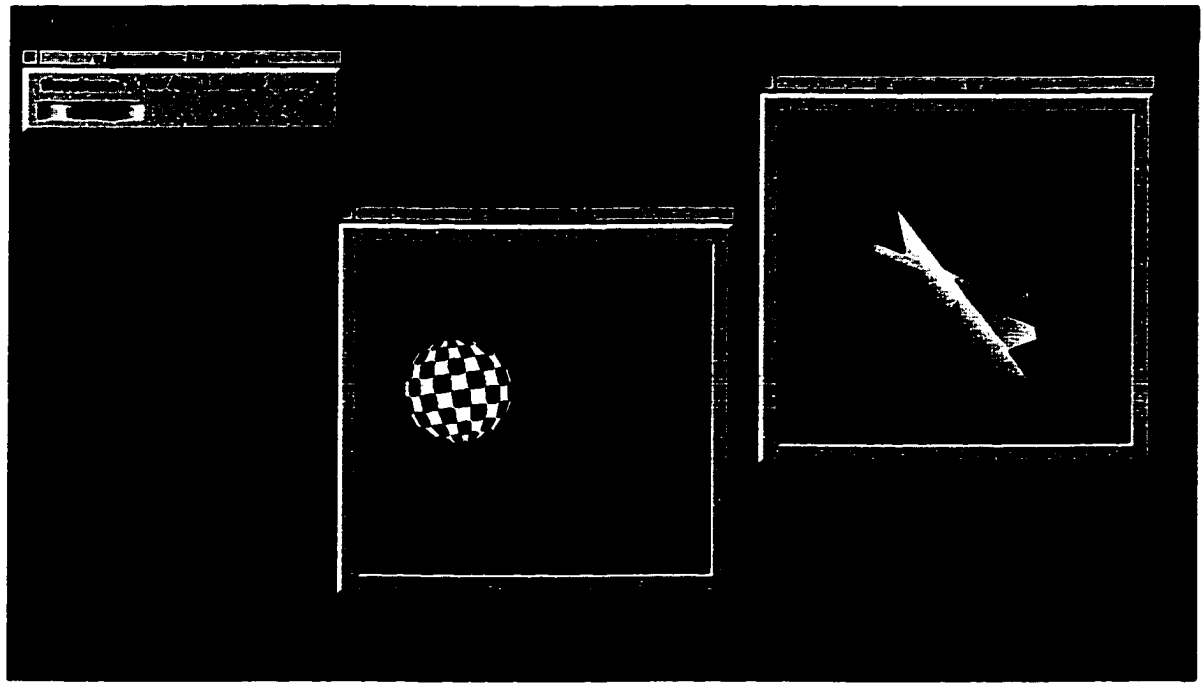


Figure 12 Hv3D animated example 1, part 1

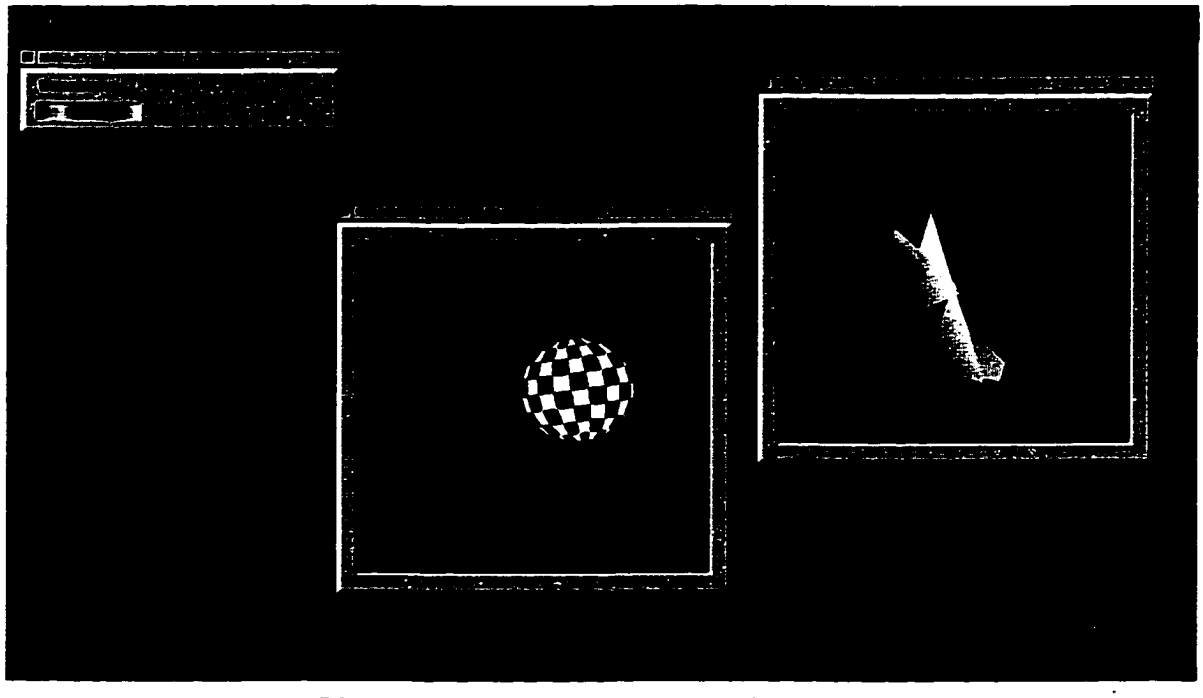


Figure 13 Hv3D animated example 1, part 2

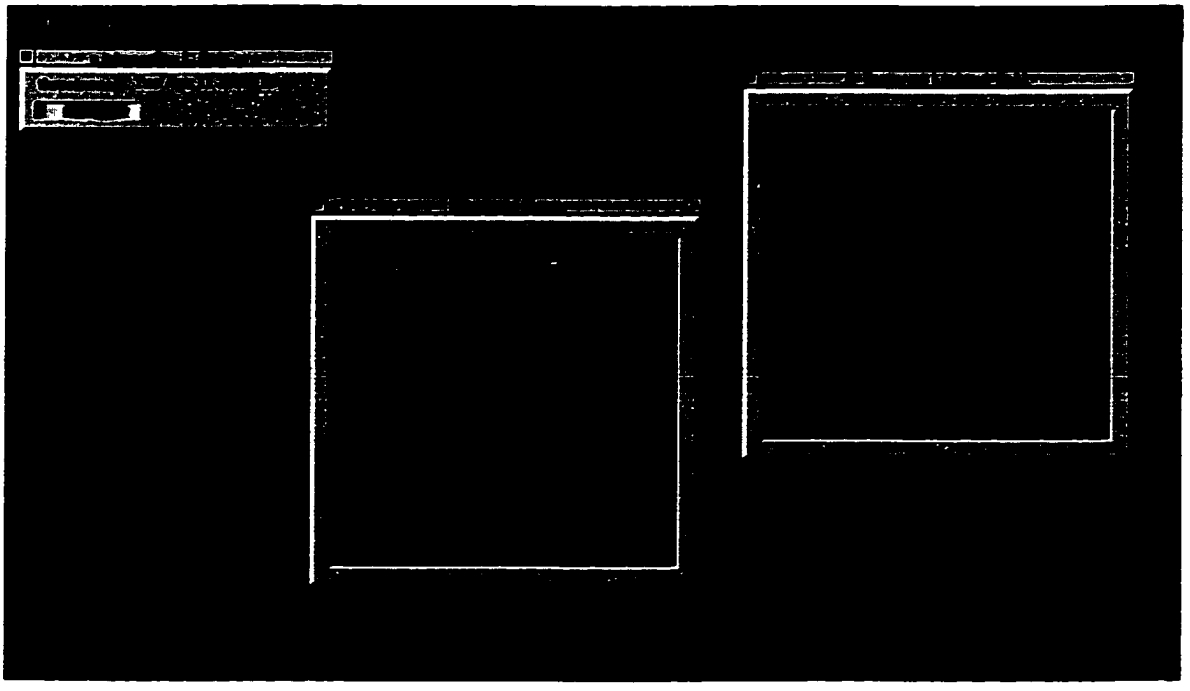


Figure 14 Hv3D animated example 2, part 1

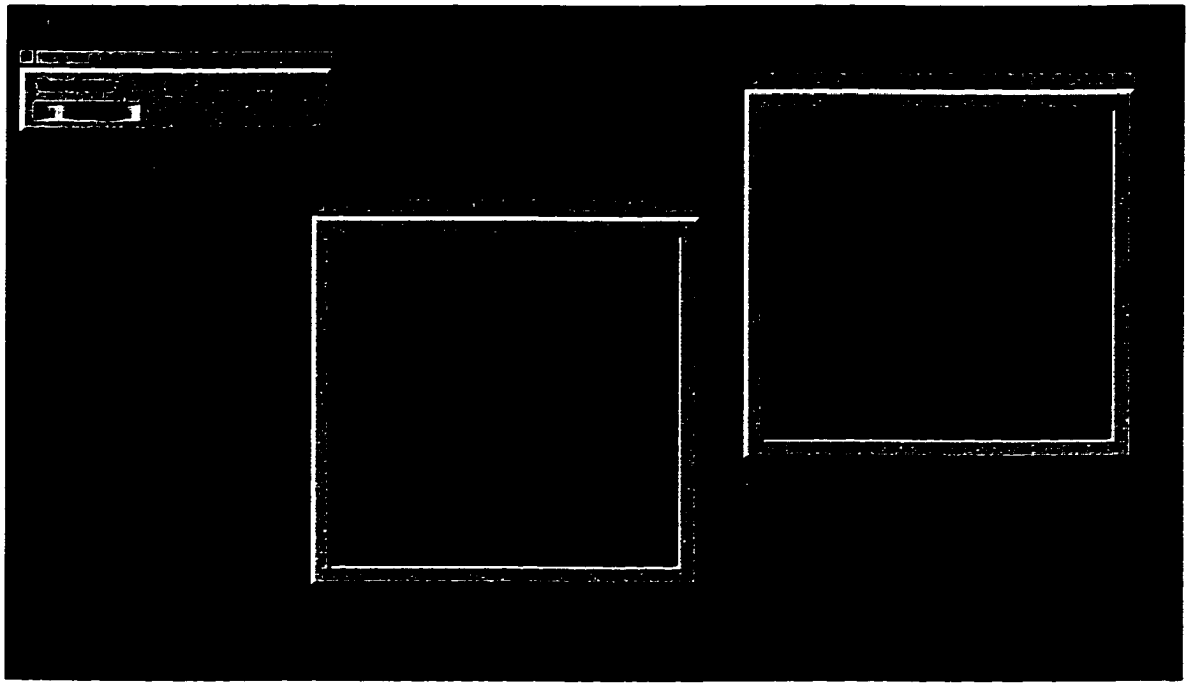


Figure 15 Hv3D animated example 2, part 2

4. Implementation Methods

To achieve these goals, we need to alter the HotViews library and implement a sample application. Both the application and the library will call functions from the OpenGL library. Two source trees were developed simultaneously during this project. One contains modified Hv code, with changes designed to allow OpenGL to render within the Hv window. The other contains a sample Hv application, the bulk of which is OpenGL code to render 3D shapes. These calls obey the OpenGL API and make no assumptions about underlying graphics hardware. The goal was to implement the OpenGL demo programs with only the minimum overhead needed to embed them into Hv. In fact, most of our application code is dedicated to creating the 3D objects in OpenGL.

4.1. Test Platform

HotViews runs on a dozen or more different Unix operating systems from different vendors. It requires the X Window System and the Motif graphics libraries, both of which are present or available on most systems. Since OpenGL is platform independent, we can incorporate it into HotViews without loosing portability.

The main test platform consisted of a Linux PC with no special graphics hardware. XFree86 provided an X implementation, along with a Redhat release of Motif. Mesa provided an OpenGL API. Mesa is a GNU open source implementation of the OpenGL library API. Although it is not a licensed implementation, it does pass most of the conformance tests. Commercial versions of OpenGL are typically licensed by SGI.

The X device configuration was set up in two different ways to test the application behavior under changing conditions. One configuration provided a 24 bit TrueColor visual only, while the other allowed an 8 bit visual of any visual type. This helped to test our portability to systems with different kinds of display hardware. One subsidiary goal of the software was to dynamically detect different types of resources and use them appropriately to facilitate the display of 3D and 2D graphics simultaneously.

4.2. Visuals

X provides the user with two kinds of objects for graphics output, called drawables. The

window is used when graphics output is directed to the screen, although such output may be dropped or ignored if a window is iconified, occluded, or otherwise not visible. The pixmap is used when graphics output will not immediately be displayed, perhaps because it is destined for a file or for later use.

A drawable has many attributes describing its behavior. The depth of a drawable is a simple integer indicating how many bits are allocated for each individual pixel on the screen. A drawable's visual determines how those bits are translated into a color. The number and type of visuals allowed depend on the graphics hardware being used. A colormap gives an exact mapping between each possible combination of bits in a pixel and the resulting color displayed. Colormaps and visuals are conceptually similar : a drawable's visual describes the general structure of its colormap. Table 2 shows the six available visual types provided by X Windows.

A monochrome colormap provides only black and white or grayscale information. Usually, this type of colormap would only be used if the graphics hardware available could not produce full color.

An single indexed RGB colormap describes the color of each pixel as proportional mixtures of red, green, and blue elements in an indirect manner. Each pixel value is considered to be an index into a table of RGB values, which can be inserted into the table in any order. Graphics hardware with few bits available for each pixel, for example 8bpp (bits per pixel), often use single indexed colormaps because more bits may be used in the table entries (24 bits, for example) to describe a finer gradient of a particular color. For example, one could take half of the entries in a table for an 8bpp index and create 128 shades of yellow. When rendering a particular scene, this allows one to trade off a palette of some colors which aren't used for more shades of other colors which are used.

A decomposed index colormap simply divides each pixel by groups of bits into separate indices for the color components of red, green, and blue, much like the table does for the single indexed colormaps. So, for a 24 bit pixel, 8 bits may be used apiece for red, green, and blue. The resulting set of three tables uses much less memory than a single monolithic table. One may consider the

colormap a trivial mapping using zero for the least intense color, and the highest possible unsigned integer for the most intense color. But the progression of color intensities in these tables can be linear or nonlinear, and may be set up by the system or the application.

Visuals also determine whether a colormap table is read-only or may be altered.

Visual Type	Colormap Read / Write	Colormap Read-Only
Monochrome	Grayscale	StaticGray
Single Index RGB	PseudoColor	StaticColor
Decomposed Index RGB	DirectColor	TrueColor

Table 2 Visuals and Colormaps

4.3. Colormaps

For 3D graphics, we prefer to use an RGB colormap. This can be achieved straightforwardly using a decomposed index colormap, such as TrueColor, or by using a single index colormap and carefully allocating all colors in the color table to correspond to a decomposition of the bits in the index. Mesa, for example, will utilize a single index colormap in this manner. Since our target hardware may have multiple visuals available, we must carefully set up our defaults at window initialization time.

Each window in X is allowed to have its own colormap. Applications usually share colormaps unless they need to numerically specify several unique colors. Colormaps may not have their tables completely filled at creation time. Hv used to pick the standard shared colormap of the system and assign new colors to it where allowed. If it ran out of room, it would create a new colormap. Now, Hv picks and names its colors by selecting a nearest match from those already present in the RGB colormap. The X library does this work for us; the calls to assign named colors simply pick the closest value if the colormap has already been filled, or it will attach new color values to the colormap if the colormap is not full.

4.4. Pixmap Rendering

OpenGL incorporates the idea of multiple program threads rendering to the same screen at once. In OpenGL terms, each thread is bound to a context which contains information about the

OpenGL state machine, enabled options, and buffers. GLX allows a thread's context to be bound to a window or pixmap. Rendering to the Hv window directly creates conflicts with the X based rendering that Hv already uses. In graphics terms, OpenGL and X write to the same framebuffer.

To complicate things further, OpenGL and X are both asynchronous protocols, and the programmer is not guaranteed that objects will be rendered in the same order as they were called in the program. This can be partially alleviated by using routines to flush either pipeline to synchronize the two protocols. But, since OpenGL routines may overwrite an entire framebuffer, the entire X window would have to be redrawn each time an OpenGL View was updated. This would not result in good performance, especially if the 3D scene was animated.

There are some ways to assure that OpenGL only draws to a portion of a window, such as using clipping planes, stencil buffers, or a viewport. However, when OpenGL changes a scene, such as in an animation, it begins by clearing the framebuffer. This erases the entire window.

One could also change Hv dramatically by making each View operate in a separate system window, creating them as children of the primary Hv window. This would have a negative impact on performance, because Hv applications typically use many Views. But if we selected only OpenGL views to operate in separate windows, we would run into problems when the window was partially occluded by a non-OpenGL view.

X requires that all drawing or rendering be done within windows. The workspace background is called the root window. All icons and menu bars are windows, as are application windows. In X terms, a child window is a window which is displayed on top of another, larger window called the parent. If a child window is displayed at all, or realized, in X terms, then it covers a rectangular area of its parent window. Sibling windows are children of the same parent. Siblings may partially occlude one another. X keeps track of which window is on top by defining a stacking order for sibling windows and changing the order as the windows are manipulated. X applications usually create a child of the root window. They may optionally create children for the application window to render buttons or pop up menus. The rules of the windowing system would not allow a child window to be clipped partially by the parent Hv main window. Windows may only be partially

occluded by sibling windows. By these rules, we could not render our OpenGL to a child window unless we were sure that no other views were partially occluding the OpenGL view.

We chose to bind our GLX context to a pixmap created at initialization time. Our OpenGL rendering can be done to this pixmap without fearing that it will erase any part of the Hv window. Once we have completed drawing a scene, we can verify that the final pixels have been rendered to the pixmap by flushing all OpenGL commands using `glFlush`. The completed pixmap can then be copied to the Hv window using `XCopyArea`. This command, like most X commands, takes as an argument a drawing context. This context contains general drawing information, including a list of arbitrary regions that the command is not allowed to draw to. We make sure that the regions include other views and items that should not be covered. Figure 16 demonstrates successful occluding.

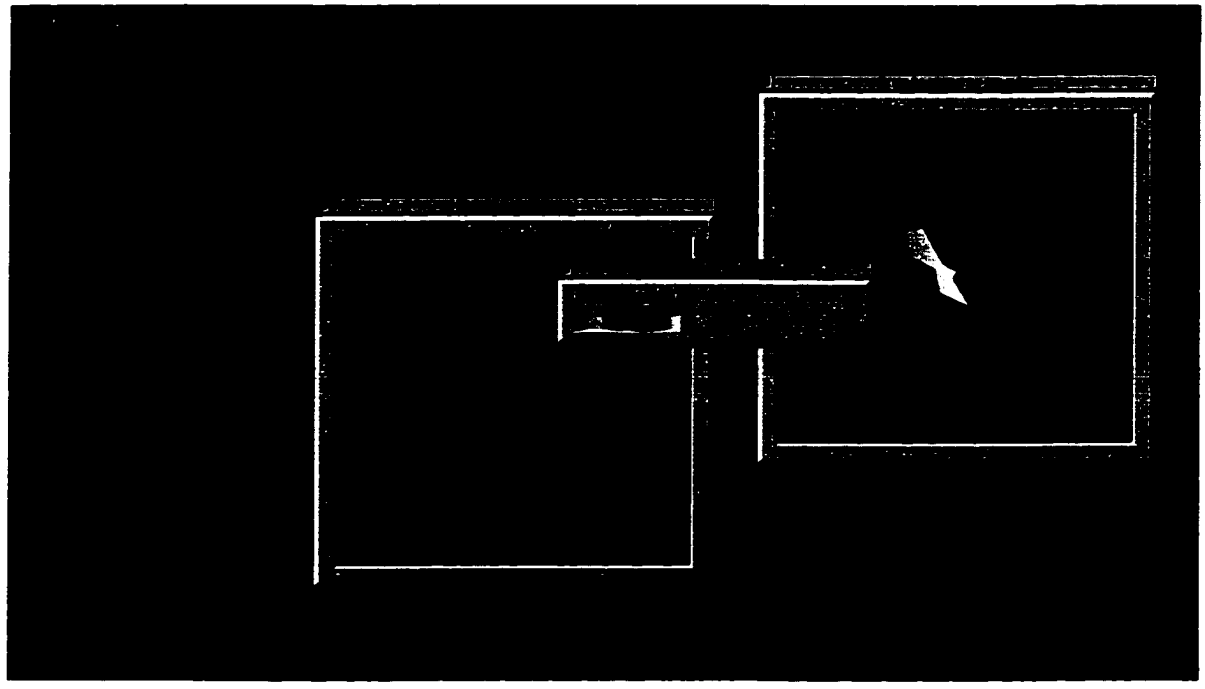


Figure 16 Occluding example 1

4.5. Motif

The use of Motif presents an additional problem. Motif creates widgets, which are windows with their own visuals, depths, and colormaps. These characteristics are determined by the values of resource variables that each widget contains. Widget behavior and resource variables are specified in an object oriented manner, with subclasses inheriting these characteristics from superclasses. Table 3 illustrates some of the superclass – subclass relationships of widgets. Note that this only determines which resource variables a widget can have, not what values they hold. Widgets also have a hierarchical relationship due to their implementation as windows. Some of these relationships, called parent – child relationships, are shown in Table 4. For example a shell widget, which interacts with the window manager, may contain a form widget. A form widget, which places widgets in a geometric layout, may contain several buttons.

Widget	Description	Superclass
Core	Superclass of almost all widgets	None
Composite	Superclass of widgets with children	Core
Constraint	Controls geometry of children	Composite
Manager	Provides visual effects, eg shadow & highlighting	Constraint
DrawingArea	Blank canvas for interactive drawing	Manager
BulletinBoard	Allows children to be placed at x,y coodinates	Manager
Form	Constrains children by defining their attachments	BulletinBoard
Shell	Handles interaction with the window manager	Core

Table 3 Widget Superclass / Subclass Examples

There is no problem with separate widgets or windows using different visual resources, even if they use different resources from their parent windows. However, Hv, like most Motif applications, creates a top level widget for the entire application. It then creates a form on which to place the menu bar and main graphics area, and then a drawing area for the main graphics. This drawing area is where our OpenGL graphics will be rendered, and therefore it has to have the correct visual resources.

Widgets get their resource values from various sources. The user can set them directly by passing argument lists to Motif functions. Some resource values can be set this way only at widget creation time, while others can be set any time. Some resource values are inherited from parent widgets, and still others are set to default values.

This combination of sources for resource values can be hazardous. If the visual, depth, and colormap of a widget or window are not compatible with each other, a fatal X BadMatch error will result. If a widget does not have a specfic resource variable for these, then they will be set automatically by the library to match their parents. When only depth is a resource for a widget, we must be careful to set a legal depth for the inherited visual and colormap.

Hv Name	Widget Type	Visual	Depth	Colormap	Parent
Hv_toplevel	Shell	Yes	Yes	Yes	None
Hv_form	Form	No	Yes	No	Hv_toplevel
Hv_canvas	DrawingArea	Yes	Yes	Yes	Hv_form

Table 4 Widget Parent / Child Examples

This can be a very difficult bug to track down. X Windows provides an asynchronous interface to the graphics device, and errors can be reported much later then they were actually caused. Even explicitly synchronizing the library with special debugging functions does not guarantee that we will see error reports shortly after we cause an error. To compound the problem, Motif is asynchronous as well. Widgets may not be instantiated when we expect, and not even in the order we would expect.

To bypass this complication, we decided to explicitly specify all visual resources for all widgets. To do this, the Hv library now wraps all `XmCreate` calls into trivial functions which attach a new argument list containing the visual resources. An additional benefit of homogenizing the visual resources in this manner is that all Hv widgets and subwindows will use the same colormap, and moving the mouse from widget to widget will not cause colors to change.

4.6. Top Window Optimization

Rendering to a pixmap does have some disadvantages. The GLX extension decodes OpenGL commands coming in over the X protocol and sends them to the graphics hardware. A 3D video card can have its own optimized command set, and GLX has drivers built in for many different cards. GLX can then take advantage of each card's abilities to provide faster rendering, better graphical detail, and special effects. This performance improvement can be so drastic that the X message protocol itself actually becomes the bottleneck. GLX provides another optimization for this case and can bypass the X protocol all together, using its own optimized message protocol. This is known as direct rendering. But we only get these benefits when we render directly to the graphics card's own framebuffer. This only happens when we bind our context to a window, not to a pixmap.

We cannot draw some of the Views in separate child windows and others in the HotView main window because of the occluding rules of X Windows. When a child widow is displayed on top of the HotViews window, it would draw over everything within its rectangular area. No 2D drawing done manually by Hv to the main window would be visible in this area.

In one case, however, this would not matter. When an OpenGL View is the top View,

there will be nothing occluding it and a child window would not interfere. Since the user will be interacting with the top View, it is desirable for 3D animation in this View to be most efficient.

Our OpenGL rendering routines detect the presence of a top View and treat it as a special case. The pixels in this case are sent directly to a child window rather than to a pixmap as in all other cases. The child window is created, switched from context to context, and deleted as needed. When this optimization is enabled, the program cannot use Hv 2D items or drawing routines in the same HotRect as the 3D graphics. In this case, it would be better to continue using the Pixmap method which allows for occluded regions.

The performance gained from this optimization comes from offloading some of the processing from the main CPU to the 3D graphics hardware, from bypassing the unsuited X protocol when the program is running on the same station as the display screen, and from utilizing the specialized ASICs, display memory, and texture memory that the 3D graphics hardware supplies. The benefit of this optimization will vary greatly across different systems with different display hardware.

Figure 17 shows a top view rendered in an X window occluding a lower view rendered in a pixmap. Note that the user sees no difference between a view rendered to a window or a pixmap, except that the window rendering may be faster and use fewer CPU cycles.

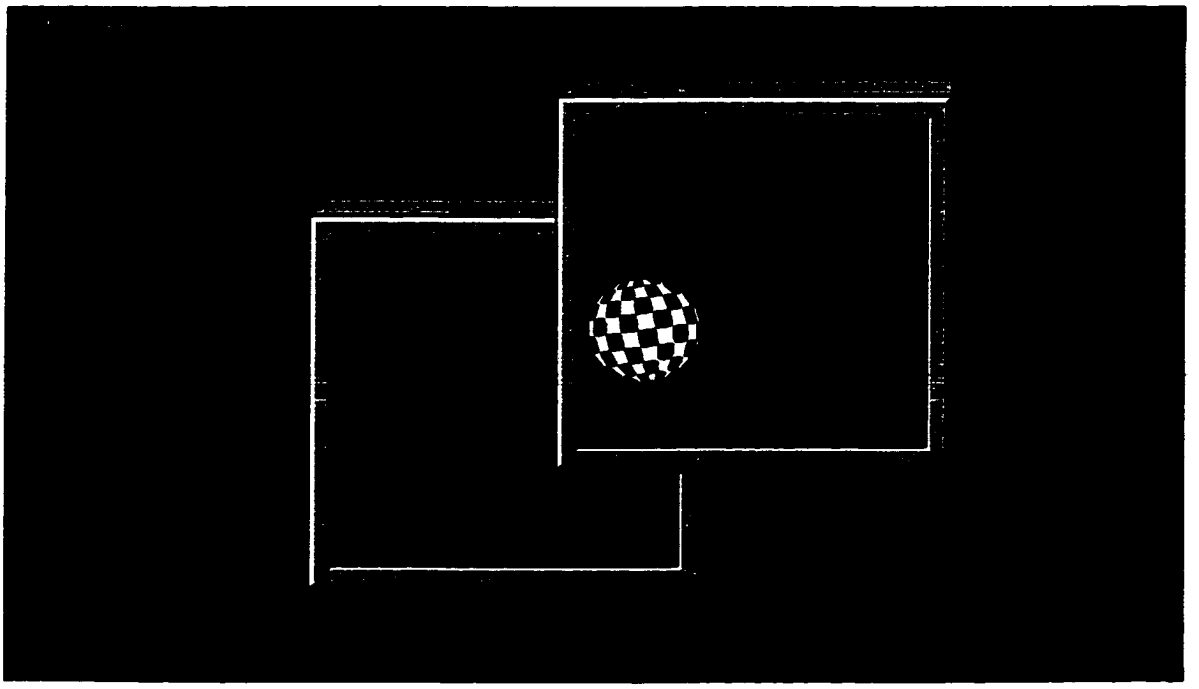


Figure 17 Occluding example 2

5. Conclusions

We have found a solution to the multiple rendering problem. The various libraries render cooperatively to the same X Window. They do so in an efficient manner, adding to program performance when drawing complex scenes.

The new version of HotViews requires an implementation of OpenGL. This must be supplied separately. The 3D features supported will depend on which OpenGL implementation is used. Once Hv is linked to the OpenGL implementation, all of the available 3D features are supported. HotViews imposes no additional limitations on 3D rendering.

Hv is a large library, utilizing over 70000 lines of code. OpenGL implementations are even larger. Mesa, for instance, uses over 200000 lines of code. OpenGL has many implementations, supported systems, and graphics hardware device support. Rewriting either library would be an enormous effort. By integrating both libraries, we allow the programmer to take advantage of both libraries' functionality at once.

Future work in HotViews could include enabling real world coordinate feedback for 3D views. Pbuffers, which are being introduced in newer releases of OpenGL, could be utilized instead of or in addition to pixmaps. They would allow the same kind of off screen rendering to take place, but could utilize 3D graphics hardware to increase performance. The solutions that we found here could be tied into the HotViews library directly via new options and functions, to allow applications easier and transparent access to the methods we've invented.

Appendix A : Annotated HotViews Library Code Selections

Changes made directly to the HotViews library are described here. Note that this is only a partial listing.

A global variable had to be added to `Hv.h` to track X visual selection, along with the variables that already track depth and colormap information. A drawing control bit had to be added to views to indicate when they are going inactive. The custom drawing routine is called once more while the View is going inactive so that it can switch contexts from the top window to a pixmap. It will also unmap the window, which may later be mapped by a different view.

```
#define Hv_GOINGIA                  02000
/* Bit 10 -> View is going to be deactivated */
extern XVisualInfo    *Hv_visinfo;
```

Prototypes for the Hv Motif wrapper functions are global because Motif widgets may be created in any module.

```
/*--------- Hv_motif.c ---------*/

extern void Hv_MotifInit (void);
extern Hv_Widget HvXmCreateBulletinBoardDialog (Hv_Widget, char
*, Arg *, int);
extern Hv_Widget HvXmCreateCascadeButtonGadget (Hv_Widget, char
*, Arg *, int);
extern Hv_Widget HvXmCreateDrawingArea (Hv_Widget, char *, Arg *
, int);
extern Hv_Widget HvXmCreateFileSelectionDialog (Hv_Widget, char
*, Arg *, int);
```

`Hv_Matchvisual` was deleted from the module `Hv_color.c`. It was used to discover the visual class of the default visual of the display. We no longer concern ourselves with the default visual, depth, or colormap of the display. Instead, we use GLX to hunt for a more desirable configuration.

In order to allow fall through code to select named colors from the colormap, we go ahead and fill in the global variables for default colormap, visual, and depth. We use the display default colormap only if GLX happened to pick a visual which matched the display default visual.

```
  if (DefaultVisual(dpy,scr) == defvis) {
#ifdef DEBUG
    printf ("Using Default Root Window Colormap.\n");
#endif
    defcmap=DefaultColormap(dpy,scr);
  } else {
#ifdef DEBUG
    printf ("Using New Colormap.\n");
#endif
    /* Hopefully, this will allow us to allocate colors later in
 code, so
       that the fall-through will work
       Don't worry about the RootWindow parameter, it is only us
ed by
       X to determine the proper screen.  I use the screen to fi
nd a
       window.  Isn't that sick?
    */
    defcmap=XCreateColormap(dpy, RootWindow(dpy, Hv_visinfo->scr
een),
                            Hv_visinfo->visual, AllocNone);
  }

  Hv_visClass = Hv_visinfo->class;
```

All calls to **XmCreateWIDGET** were replaced with calls to **HvXmCreateWIDGET**, as in this example from **Hv_dlogs.c**. These wrappers are defined in the module **Hv_motif.c**. This allows us to set default resources for all widgets without excessively interfering with legacy code. We use these wrappers in several modules.

```
  submenu = HvXmCreatePulldownMenu(rc, "optionsubmenu", NULL, 0)
;
  popup = HvXmCreateOptionMenu(rc, "option_menu", arg, n);
```

These arrays in **Hv_init.c** set the features we ask GLX to look for in an overloaded visual.

```
  /* To get a GLX overloaded visual, we
     will try these in order of preference  */

  /* Should get rid of the X_VISUAL_EXT later */
  int dblBufRGB[] = {GLX_X_VISUAL_TYPE_EXT, GLX_TRUE_COLOR_EXT,
GLX_RGBA,
                     GLX_DOUBLEBUFFER, GLX_DEPTH_SIZE, 8, None};
  int snglBufRGB[] = {GLX_X_VISUAL_TYPE_EXT, GLX_TRUE_COLOR_EXT,
 GLX_RGBA,
                      GLX_DEPTH_SIZE, 8, None};
  int dblBuf[] = {GLX_DOUBLEBUFFER, GLX_DEPTH_SIZE, 8, None};
  int snglBuf[] = {GLX_DEPTH_SIZE, 8, None};

  XVisualInfo  *visinfo;
  int screen;
```

Motif and Xt initialization was simpler in the older Hv, consisting of a single call to **XtVaAppInitialize**. We must break down this call into several steps in order to better customize the visual resources used by the main parent widgets.

```
  /* MOTIF INITIALIZATION : toolkit, appcontext, and display */

  XtToolkitInitialize();
  Hv_appContext = XtCreateApplicationContext();
  if (fbr) XtAppSetFallbackResources(Hv_appContext, (String *)fb
r);
  Hv_display = XtOpenDisplay(Hv_appContext, NULL,
                             (String)Hv_appName, NULL,
                             (XrmOptionDescList)NULL, (Cardinal)
0,

#if XtSpecificationRelease > 4
                             (int *)&argc,
#else
                             (Cardinal *)&argc,
#endif
                             (String *)argv);
  screen = DefaultScreen(Hv_display);
```

Next, we incrementally attempt to get a desirable visual setup through GLX from the X Server.

```
  /* GLX INITIALIZATION : visual */
  printf ("Trying for a GLX Double RGB Visual.\n");
  visinfo = glXChooseVisual(Hv_display, screen, dblBufRGB);
  if (!visinfo) {
    printf ("Trying for a GLX Single RGB Visual.\n");
    visinfo = glXChooseVisual(Hv_display, screen, snglBufRGB);
  }
  if (!visinfo) {
    printf ("Trying for a GLX Double Indexed Visual.\n");
    visinfo = glXChooseVisual(Hv_display, screen, dblBuf);
  }
  if (!visinfo) {
    printf ("Trying for a GLX Single Indexed Visual.\n");
    visinfo = glXChooseVisual(Hv_display, screen, snglBuf);
  }
  if (!visinfo) {
    fprintf (stderr, "Couldn't find a GL capable visual.\n");
    exit (-1);
  }

  Hv_visinfo = visinfo;
  printf ("Hv_visinfo Visual ID : %x\n",XVisualIDFromVisual(Hv_v
isinfo));
```

The top level widget, the application shell, is created manually.

```
  /* MOTIF INITIALIZATION : shell */

  /* Hmm... Are we stuck with Motif resources here? IE XmNvisual
? */
  Hv_toplevel = XtVaAppCreateShell ((String)Hv_appName, NULL,
                                    applicationShellWidgetClass,
 Hv_display,
                                    XtNvisual, Hv_visinfo->visua
l,
                                    XtNdepth, Hv_visinfo->depth,
                                    NULL);
```

The initialization of the new Motif module, the creation of the application form widget, and the assignment of a colormap to the top level widget must be done in a careful order in order to assure that subwidgets inherit the right values.

```
  /* set the colormap attribute for the Hv_toplevel widget.
   This may be inherited by children. */
  XtVaSetValues(Hv_toplevel, XmNcolormap, Hv_colorMap, NULL);
  Hv_MotifInit();                /* Set up default arguments for
widgets */
  Hv_InitControls();             /* default values for control va
rs */
  Hv_CreateForm();               /* Set up main form */
  Hv_InitBalloons();             /* initialize baloon help */
```

In `Hv_views.c`, the View drawing routine needed to be modified to call the user drawing routine one last time with the draw control bit, `Hv_GOINGIA` set. This allows our custom drawing routine to delete the optimizing top window, if needed.

```
Hv_SetBit(&(View->drawcontrol), Hv_GOINGIA);
printf ("One more time!.\n");
Hv_DrawViewHotRect(View);
Hv_ClearBit(&(View->drawcontrol), Hv_ACTIVE);
Hv_ClearBit(&(View->drawcontrol), Hv_GOINGIA);
```

The `Hv_motif.c` module was added to the Hv library to provide wrappers to all widget creation function calls. We do this in order to alter the argument lists passed to all widgets.

```
#undef XMDEBUG

#include "Hv.h"
#include <Xm/MenuShell.h>
#include <Xm/FileSB.h>

Arg dargv[10];
int dargc;
```

The initialization of this module, called in `Hv_init.c`, sets up a global argument list containing the chosen visual, depth, and colormap of the Hv application.

```
void Hv_MotifInit () {
  /* Set up default argument lists for Motif widgets.  If a list
     already exists, we will have to add these items to it. */
  XtSetArg (dargv[dargc],XmNdepth,Hv_visinfo->depth);
  XtSetArg (dargv[dargc],XmNcolormap,Hv_colorMap);
  XtSetArg (dargv[dargc],XmNvisual,Hv_visinfo->visual);

  printf ("HV Default Visual ID : %x\n",XVisualIDFromVisual(Hv_v
isinfo->visual));
  printf ("HV Default Depth : %d\n",Hv_visinfo->depth);
}
```

A typical wrapper to an `XmCreateWIDGET` call looks like this. The calling argument list is merged with the standard list created above, and the underlying function is called from the Xm library.

```
Hv_Widget HvXmCreateBulletinBoardDialog (Hv_Widget parent, char
* name,
                                         Arg * argv, int argc) {
  Arg * newargs;
  Hv_Widget rwidget;

  newargs = XtMergeArgLists (argv, argc, dargv, 2);
  rwidget = XmCreateBulletinBoardDialog (parent, name, newargs,
                                         argc+2);
  XtFree ((char *) newargs);

  return (rwidget);
}
```

More trivial wrappers are created for gadgets. Since they do not actually create windows, visuals are not important.

```
Hv_Widget HvXmCreateCascadeButtonGadget (Hv_Widget parent, char
* name,
                                         Arg * argv, int argc) {
  Hv_Widget rwidget;

  rwidget = XmCreateCascadeButtonGadget (parent, name, argv, arg
c);

  return (rwidget);
}
```

Appendix B : Annotated 3D Application Code

This sample application illustrates the ability to render OpenGL graphics from within a HotViews application. It utilizes OpenGL's ability to contain more than one copy of the 3D state machine to maintain different 3D objects in separate views.

The code fragments which follow do not contain the complete program.

The program begins by including headers needed for OpenGL and Hv. Each library should also be linked to the application at compile time.

```
#include <Hv.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <GL/gl.h>
#include <GL/glx.h>
```

The X Window data structure is for the top window optimization. We only need one global window structure because there will only be one optimized View at any one time. Each 3D View type will have an instance of the **drawdata** structure type. It contains information the drawing routine will need, such as a drawable to render to, GLX and X drawing contexts, visual information, and more.

```
Window subwin;
int subwinowner=-1;

struct drawdata {
  int routine;
  Display * dpy;
  XVisualInfo    *vi;
  Pixmap pm;
  Window w;
  GC xgc;
  XGCValues xgcv;
  GLXPixmap gpm;
  GLXContext      cx;
};
```

Global variables are used to keep track of the state of the four separate 3D views. In this example, the bouncing ball, the velocity and rotation of the ball is significant, as well as the **BALL** integer which is used as a handler for the display list.

```
/* Bounce */
static GLuint make_ball(void);
static void ballbounce(Hv_View View);
GLuint BALL;
GLfloat Zrot = 0.0, Zstep = 6.0;
GLfloat Xpos = 0.0, Ypos = 1.0;
GLfloat Xvel = 0.2, Yvel = 0.0;
GLfloat Xmin = -4.0, Xmax = 4.0;
GLfloat Ymin = -3.8, Ymax = 4.0;
GLfloat G = -0.1;
int moving = 0;
```

The main function in an Hv application is usually simple. `Hv_Go` calls the library's non exiting event handler. Most of our work is done in `PrivateInitialization`.

```
int main (int argc, char ** argv) {
  int error;

  Hv_VaInitialize (argc, argv,
                   Hv_USEWELCOMEVIEW, True,
                   NULL);

  if ((error=PrivateInitialization()) != 0) {
    printf ("Error : %d\n",error);
    printf ("Something went wrong in initialization.\n");
    exit (error);
  }

  Hv_Go();

}
```

A separate view is created for each 3D object. The initialization function, `ShowglSetupView`, will take care of some OpenGL initialization. In Hv, animation is set up through callbacks using Xt. The animation function is set here. One drawback of this method for testing purposes is that system performance is not exploited since frames are drawn at a specific frequency. In order to figure out the maximum frame rate, we would need to draw frames continuously in a separate thread.

```
/* Create Initial Views */

Hv_VaCreateView (
                  &view1,
                  Hv_TAG,                 TRIANGLE,
                  Hv_INITIALIZE,          ShowglSetupView,
                  Hv_CUSTOMIZE,           ShowglCustomize,
                  Hv_FEEDBACK,            ShowglFBFill,
                  Hv_HOTRECTWIDTH,        300,
                  Hv_HOTRECTHEIGHT,       300,
                  Hv_TITLE,               "A spinning triangle",
                  Hv_DRAWCONTROL,         Hv_NOHOTRECTFILL,
                  Hv_SIMPROC,             trianglespin,
                  Hv_SIMINTERVAL,         50,
                  NULL
                  );
```

The optimized window is created here. It will be mapped and unmapped later in the code. Configuration of this window is simple since it is merely a child of the application window and does not interact with the window manager.

```
  // Will use this subwindow for optimized rendering when View i
s on top
  swa.colormap = Hv_colorMap;
  subwin = XCreateWindow(Hv_display, Hv_mainWindow, 0, 0, 300, 3
00, 0,
                         Hv_colorDepth, CopyFromParent, Hv_v
isinfo->visual,
                         CWColormap, &swa);
```

`ShowglSetupView` is called once for each of the 3D views. The `drawdata` structure is initialized using values chosen in the Hv library initialization routines and saved in library global variables.

```
  // Set up drawing area
  struct drawdata * dd = malloc(sizeof(struct drawdata));

  dd->dpy = Hv_display;
  dd->w = Hv_mainWindow;
  /* Kind of brain dead : set up the appropriate drawing routine
 by
     incrementing a variable. */
  droutine++;
  dd->routine = droutine;

  printf ("current mainWindow widget : %x\n",XtWindowToWidget(dd
->dpy, dd->w));

  dd->vi = Hv_visinfo;
```

A separate GLX context is created for each 3D view. Each context has its own copy of the OpenGL state machine. Effects or transforms in one context have no effect on the other.

```
 /* create an OpenGL rendering context */
 dd->cx = glXCreateContext(dd->dpy, dd->vi,
                           /* no display list sharing */ None,
                           /* favor direct */ GL_TRUE);
 if (dd->cx == NULL) {printf ("Could not create a context.\n");
exit(-1);}
```

A pixmap must be created, first in X, then in GLX. GLX adds information and resources to the pixmap, such as auxilliary buffers. GLX can then handle the pixmap exactly as it would handle rendering to a window. This is referred to as overloading the visual.

```
  if (!(dd->pm = XCreatePixmap(dd->dpy, RootWindow(dd->dpy,dd->v
i->screen),
                               HRMAXWIDTH, HRMAXHEIGHT, dd->vi->de
pth))) {
    printf ("Couldn't create a pixmap in X.\n");
    printf ("GL Error : %d\n",glGetError());
    exit(-1);
  }
  if (!(dd->gpm = glXCreateGLXPixmap (dd->dpy, dd->vi, dd->pm)))
 {
    printf ("Couldn't create a pixmap in GLX.\n");
    printf ("GL Error : %d\n",glGetError());
    exit(-1);
  }
  glXMakeCurrent(dd->dpy, dd->gpm, dd->cx);
```

We need to set up an X drawing context even though we aren't making calls directly to many X drawing routines. Copying the pixmap to a window requires this as a parameter.

```
// Haven't set xgcv
dd->xgc = XCreateGC(dd->dpy, dd->w, 0, &(dd->xgcv));
```

Commands like **glEnable** and **glDepthFunc** turn on features in the OpenGL library. **glClearDepth** and **glClearColor** perform operations on the frame buffer and auxillliary buffers. **glLoadIdentity**, **gluPerspective**, and **glTranslatef** all operate on the matrices used for co-ordinate transforms. It is important to note that the operations will only affect the state of the current GLX context, in this case the triangle view.

```
  switch (dd->routine) {

  case TRIANGLE :
    /* setup OpenGL state */
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glClearDepth(1.0);
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glLoadIdentity();
    gluPerspective(40.0, 1.0, 10.0, 200.0);
    glTranslatef(0.0, 0.0, -50.0); glRotatef(-58.0, 0.0, 1.0, 0.
0);
    break;
```

We create a custom item to reserve drawing space for our OpenGL rendering. We simply inherit a built in Hv item, **Hv_WORLDRECTITEM**, and customize the drawing routine by calling our own function. We pass the **drawdata** structure to this function.

```
  GlCanvas = Hv_VaCreateItem (View,
                              Hv_TYPE,          Hv_WORLDRECTITEM
,
                              Hv_USERDRAW,      draw3d,
                              Hv_USERPTR,       (void *)dd,
                              NULL);
```

The **draw3d** function is where the most interesting work is done, aside from the visual configuration done in the Hv library. The **drawdata** structure is set up for use.

```
void draw3d (Hv_Item Item,
             Hv_Region region) {

  int testclip;
  int testresult;
  GLint i;

  struct drawdata * dd = malloc(sizeof(struct drawdata));
  dd = (struct drawdata *)(Item->userptr);
```

Our drawing routine checks to see if the View is on top of all others. If it is, and the View does not already own the subwindow, the subwindow is mapped to the screen. OpenGL rendering is then bound to that window, and the program is bound to the 3D object's OpenGL state.

If the View is not on top, the OpenGL rendering is bound to a pixmap.

```
  if ((Hv_views.last == Item->view) &&
      !(Hv_CheckBit(Item->view->drawcontrol,Hv_GOINGIA))) {
    printf ("BEGIN : %d is on top.\n",dd->routine);
    if (subwinowner != dd->routine) {
      subwinowner=dd->routine;
      XMapWindow(dd->dpy, subwin);
    }
    XMoveWindow(dd->dpy, subwin, Item->area->left, Item->area->t
op);
    glXMakeCurrent(dd->dpy, subwin, dd->cx);
    glViewport (0,0,300,300);
  } else {
    if (subwinowner == dd->routine) {
      XUnmapWindow(dd->dpy, subwin);
      subwinowner=-1;
    }
    printf ("BEGIN : %d is not on top.\n",dd->routine);
    glXMakeCurrent(dd->dpy, dd->gpm, dd->cx);
    glViewport (0,0,Item->area->width,Item->area->height);
  }

  // Construct the GL 3D commands
  // The display list depends on the shape being drawn
  switch (dd->routine) {
```

A different drawing routine is chosen depending on the View.

```
  case SPECTEX :
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glLightfv(GL_LIGHT0, GL_POSITION, LightPos);
    glPushMatrix();
    glRotatef(90.0, 1.0, 0.0, 0.0);
    /* Typical method: diffuse + specular + texture */
    glEnable(GL_TEXTURE_2D);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, White);  /* enable diffuse
*/
    glLightfv(GL_LIGHT0, GL_SPECULAR, White);  /* enable specula
r */
#ifdef GL_VERSION_1_2
    glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SINGLE_COLOR)
;
#endif
    glCallList(Sphere);
    glPopMatrix();
    break;
```

At the end of our drawing routine, we must again distinguish between rendering to a window or rendering to a pixmap. Some extra work must be done after flushing the OpenGL queue when we are drawing to a pixmap. The pixmap must be copied over to the window. This operation interprets Hv's occluded region information so that overlapping Views are not overwritten.

```
/* SwapBuffers + Flush will work in either sb or db mode */
if (Hv_views.last != Item->view) {
  printf ("END : %d is not on top.\n",dd->routine);
  // Next, copy pixmap to main window
  glXSwapBuffers(dd->dpy,dd->pm);
  glFlush();
  XSetRegion (dd->dpy, dd->xgc, region);
  XCopyArea(dd->dpy, dd->pm, dd->w, dd->xgc,
            0, 0, Item->area->width, Item->area->height,
            Item->area->left, Item->area->top);
} else {
  printf ("END : %d is on top.\n",dd->routine);
  glXSwapBuffers(dd->dpy,subwin);
  glFlush();
}
```

Several functions are left to animate the shapes through callbacks or create display lists. Callbacks for animation usually just adjust some coordinate parameter which the drawing routine passes on to OpenGL through a simple routine such as **glRotatef**. A small part of one display list is show here as an example.

```
  glShadeModel(GL_FLAT);

  glNormal3f(0.0, 0.0, 1.0);

  /* draw front face */
  glBegin(GL_QUAD_STRIP);
  for (i = 0; i <= teeth; i++) {
    angle = i * 2.0 * M_PI / teeth;
    glVertex3f(r0 * cos(angle), r0 * sin(angle), width * 0.5);
    glVertex3f(r1 * cos(angle), r1 * sin(angle), width * 0.5);
    glVertex3f(r0 * cos(angle), r0 * sin(angle), width * 0.5);
    glVertex3f(r1 * cos(angle + 3 * da), r1 * sin(angle + 3 * da
), width * 0.5);
  }
  glEnd();
```

References

[1]"Borland C++ Builder 5", 2000, Inprise Corporation

[2]"Labview Demonstration Guide", National Instruments Corporation, March 1996

[3]Rene Brun, Fons Rademakers, "ROOT — An Object Oriented Data Analysis Framework", CERN, NIKHEF, 1997

[4]Cay S. Horstmann, Gary Cornell, "Core Java 2 Volume 1", 1999

[5]Ron Fosner, "All Aboard Hardware T and L", Game Developer, April 2000

[6]Mark J. Kilgard, "Realizing OpenGL : Two Implementations of One Architecture", Silicon Graphics Inc., 1997 ACM SIGGRAPH Eurographics Workshop

[7]Mark Segal, Kurt Akeley, "The Design of the OpenGL Graphics Interface ", Silicon Graphics Computer Systems, 1994

[8]"Glide Programming Guide", 3Dfx Interactive Inc, June 1997

[9]Mark Segal, Kurt Akeley, "The OpenGL Graphics System : A Specification", Silicon Graphics Inc, March 1998

[10]David P. Heddle, "Hv Programming Manual", Christopher Newport University and Jefferson Lab, May 31 1996

[11]David P. Heddle, "Hv : A Graphical User Interface Library for Scientific and Engineering Applications"

[12]Mark J. Kilgard, David Blythe, Deanna Hohn, "System Support for OpenGL Direct Rendering", Silicon Graphics Inc.