Workstations, Networking, Distributed Graphics, and Parallel Processing

Michael John Muuss

Leader, Advanced Computer Systems Team
Ballistic Research Laboratory
Aberdeen Proving Ground
Maryland 21005-5066 USA

ABSTRACT

The process of design is iterative in nature; designs are formulated, analyzed, and improved until the goals are met. Modern graphics workstations provide a powerful platform for performing detailed design and limited analysis of solid model designs, with faster computers accessed via network links for full resolution analysis. A broad spectrum of analysis tools exist, and most express their output in graphical form. Their three main styles of graphics display are examined, along with a look at the underlying software mechanisms to support them.

Several enabling technologies are required for analysis tools to run on one computer, with graphical display on another computer, including the network-transparent framebuffer capability. The importance of portable external data representations is reviewed, and several specific external data representations are examined in significant detail. By carefully dividing application software into appropriate tools and connecting them with UNIX pipes, a measure of parallel processing can be achieved within one system. In a tool-oriented environment with machine-independent data formats, network-distributed computation can be accomplished.

The next step is to make a single tool execute faster using parallel processing. Many analysis codes are implemented using the ray-tracing paradigm, which is ideal for execution in parallel on tightly coupled shared-memory multiprocessors and loosely coupled ensembles of computers. Both are exploited using different mechanisms. The strategies used for operating on shared-memory multiprocessors such as the Denelcor HEP, Alliant FX/8, and Cray X-MP are presented along with measured performance data.

The strategies used for dividing the work among network-connected loosely coupled processors (each of which may itself be a parallel processor) are presented, including details of the dispatching algorithm and the design of the distribution protocol. The performance issues of this type of parallel processing are presented, including a set of measured speeds on a variety of hardware.

13 May 1995

Workstations, Networking, Distributed Graphics, and Parallel Processing

Michael John Muuss

Leader, Advanced Computer Systems Team
Ballistic Research Laboratory
Aberdeen Proving Ground
Maryland 21005-5066 USA

1. INTRODUCTION

The fundamental purpose of the Ballistic Research Laboratory CAD Package is to enable designers and engineers to create and analyze highly detailed three dimensional solid models. These models are considered "solid models" because the models are a computer description of closed, solid, three dimensional shapes represented by an analytical framework within which the three-dimensional material can be completely and unambiguously defined [DEIT83]. Completeness is assured because the representation contains a full description of a piece of solid matter; there is no view-specific information [MUUS87a].

The software in the BRL CAD Package can be grouped into three main categories. First and foremost, it includes a complete solid modeling system. Secondly, there is a set of libraries for producing graphical displays on a variety of display hardware. Finally, there is a large collection of software tools, each of which performs a single function quite well, and which can be easily combined in powerful ways using UNIX pipes [RITC78a].

The purpose of this paper is to describe the graphics tools, and graphics support environment necessary to enable visual displays to be the center of the design process, and to discuss how network capabilities and parallel processing are important in such an environment, both to support collaboration, and to improve performance. The first section will set the stage by reviewing the design process. A broad spectrum of design analysis tools exist, and most express their output in graphical form. In the second section, the three main styles of graphics display are examined, along with a look at the underlying software mechanisms to support them. In the third section, the extension of these graphics mechanisms to the network environment is examined in significant detail. In the fourth section, the power of parallel processing is exploited for multi-tool applications, both on multiprocessor hardware, and utilizing multiple uniprocessor machines on the network. In the fifth section, shared memory parallel processing techniques are employed to make *a single application tool* run much faster. Finally, an ensemble of network computers is employed, to make a single application tool run much faster.

1.1. The Modern Computing Environment

The cost of computer power continues to plummet. Therefore, computer systems will become more and more prevalent in the modern workplace. They will be purchased to improve productivity, lower costs, or improve the quality of the workplace. Due to system proliferation, users will be located on different machines, separated from each other. It is vitally important that they not become isolated from necessary information, or the proliferation of computer systems will bring a family of new problems.

To exploit the influx of new computer power, *automation systems and facilities, regardless of manufacturer, model, and location, must interoperate.* This interoperation requires the connection of these systems to data networks. A user should be able to move data between any pair of computers, without physical intervention, using one simple set of commands, regardless of the data's source or destination. Single vendor networks are impossible in the government arena, and difficult elsewhere, forcing computer planners into a systems integration role. Networking standards are young, or still emerging, so there are many issues still to be resolved. This paper will detail one powerful set of applications built for a network environment full of computers.

As one of the nation's foremost research and development laboratories, BRL has continued to help advance the state of the art in computer science and computer practice, starting in 1943 with what became the ENIAC project, which resulted in BRL possessing one of the worlds first electronic computers. The proliferation of hardware started immediately thereafter. The diversity of computers installed resulted in BRL scientists having to deal with issues of software portability in the 1950s. In the early 1960s, BRL developed solid modeling and ray-tracing technology which went into production use in 1965. BRL was part of the initial ARPANET in 1967, and by the advent of the TCP/IP protocols in 1982 had the largest campus area network in DoD. In 1988, the BRL facility includes two SuperComputers, a Cray X-MP/48 and a Cray-2, a fiber-optic based campus area network, dozens of super-minicomputer systems, over a dozen distributed computer facilities, and hundreds of workstations and personal computers (See Figure 1). More than 30 high performance high resolution color graphics workstations are located throughout the laboratory. The use of color graphics to combine *advanced modeling* with *scientific visualization* accounts for BRL's continuing ability to increase the pool of human knowledge in times of dwindling funding [MCCO87].

The BRL CAD Package is BRL's third generation of modeling software, and represents over 150,000 lines of **C** source code written since 1979. Parts of this system have roots in work done over two decades ago [MAGI67], most notably the solid modeling, and the ray tracing. It is highly portable, having run on five distinct generations of hardware running various versions of the UNIX operating system [MUUS87b]. As of this writing, this software is known to run on hardware from many different vendors, and is in use at over 300 sites worldwide. The non-military portions of this software are available; contact the author for details.

Hardware Supported	
Vendor	System
Alliant	FX/4, FX/8, FX/80
Convex	C1, C2
Cray	Cray-1, Cray X-MP, Cray-2
DEC	VAX-11/750, 780, etc
DEC	VaxStation-II GPX
Denelcor	HEP H-1000
Elxsi	6410, 6420
Gould	PN 6000, PN 9000
ISI	68020
Multiflow	Trace 7/200
Pyramid	90MX
Ridge	330
SGI	3-D (2400, 3030, etc)
SGI	4-D (4D/60, etc)
Sun	Sun-2, Sun-3, Sun-4
Tektronix	4132

1.2. The Iterative Nature of Design

Design is an iterative process. Starting with a set of goals, an initial concept is defined. The initial concept is refined until the design is believed to have reached a worthwhile plateau, and then the design is subjected to a variety of analyses techniques in an attempt to determine how well it meets the original goals. The result of the analysis determines whether the goals have been met. If the goals have not been met, then the outcome of the analysis drives another iteration of the refinement process.

In a full scale solid modeling system, there is no need for initial drawings; the designer expresses the initial structures directly into the modeling system's editor, just as a modern author creates his "rough draft'' directly into a word processor. At the completion of each version of the design, the model is subjected to a battery of analyses appropriate to the function of the object being designed. Strength, volume, weight, level of protection, and other similar evaluations can be reported, along with the production of a variety of images and/or drawings. These automated analyses help identify weaknesses or deficiencies in a design *early in the design process*. By detecting flaws early, the designer has the opportunity to correct his plans before having invested too much time in a bad design, or the designer can switch to an entirely different approach which may seem more promising than the original one. In this way, the solid modeling system allows the designer to concentrate on the important, creative aspects of the design process. Freeing the designer of routine analysis permits designs to be finished in less time than previously required, or allows much more rigorously optimized designs to be delivered in comparable timeframes and at the same cost as unoptimized designs created using older techniques [DEIT85].

The creative process that converts goals into designs has been the subject of some study, but the process is not understood well enough for there to be much opportunity for automation yet. However, the analysis of a given design against a fixed set of standards (the design goals) is amenable to a great deal of automation. Given a full description of the geometry of an object, and an indication of the material used to construct each part, it is possible to conduct a battery of analyses automaticly. These analyses might range from determining the center of mass, to creating an optical image of the object, to performing a detailed structural analysis of the object under various forms of loading. In light of the iterative nature of design, and the current lack of deep insight into the creative process, the BRL CAD Package has been focused on appropriately applying the power of computers to each part of the design loop. Each of these parts are supported by at least one main piece of software.

1) The designer is provided with powerful, interactive, and vision-oriented tools for specifying the geometry of the objects that they wish to build. The specification and modification of geometry is done using the solid model editor **mged** [WEAV80, MUUS83].

2) The application writer is provided a powerful platform upon which to build analysis tools, where the physics of a particular analysis is insulated from the complexities of the shapes under analysis. Analysis tools are built, free from the details of the underlying geometry using **librt**, a ray tracing library for model interrogation.

3) A broad family of automatic analysis tools are provided to the designer, to help assess the merits of each design change. Many analysis tools are built around a generic skeleton of code that provides some common housekeeping functions, and thus reduce development costs and help provide consistent interfaces.

4) The output of the analysis tools is provided to the designer in graphical form, to permit rapid comprehension of complex assessments. The model analysis tools produce graphical output using **libfb**, a generic framebuffer library with full network display capability, and **libplot3**, a library for the production of 2-D and 3-D UNIX-plot files.

5) The ability to store and manipulate images and signals is provided, to permit convenient handling of analysis output, and to permit easy comparison of simulation results with actual test data. The handling of all the images and data resulting from the analysis codes is managed via several groups of tools, including a collection of software tools for framebuffer manipulation, a collection of software tools for image processing and analysis, a collection of software tools for handling 2-D and 3-D UNIX-Plot wireframe plots, and a collection of software tools for signal generation and processing.

There is a substantial amount of technology and software required to achieve the environment described in the last paragraph. The technology of solid modeling is discussed in much greater detail elsewhere [MUUS87a].

1.3. Overview of Analysis Tools

Many phenomena that are ordinarily difficult to model can be handled simply and elegantly with ray-tracing. For example, an illumination model based on ray-tracing merely needs to fire a ray at each light source to determine the total light energy at each point. By dithering the location of the light sources, this gives a statistically good rendition of shadow penumbra effects. Unlike conventional rendering packages, ray-tracing also makes it easy to deal with objects that are partly or entirely reflective, and with transparent objects that have varying refractive indices. Furthermore, by applying the proper sorts of dither [COOK84], motion-blur, depth-of-field, translucency, and other effects are easily achieved.

Given that lighting model code exists for making static images, it is straightforward to develop code which can animate the position of the "eye" (camera) and light sources within the model, and only a modest additional effort to develop the capability of articulating the model geometry itself. Using animation can be very beneficial when trying to gain comprehension of complex geometry, especially when the object being modeled has never actually been physically built before. There is nothing that communicates complex structures more clearly than to observe them passing by.

The power of the lighting model code can further extended by making a provision to record the paths of all the rays followed when computing the light intensity for each pixel in an auxiliary file. This capability allows one to follow the path of the light rays passing through lenses reflecting from mirrors while performing image rendering, with no additional computation, which makes a fantastic debugging tool. Studying the paths of light rays as they are repeatedly bent by passing from air to glass and back again has traditionally been a painstaking manual procedure for lens designers. By modeling, it becomes possible to predict lens behavior, including making a determination of the exact focal length, finding the precise influence of spherical distortions and edge effects, determining the amount of image distortion due to internal reflection and scattering, and finding the level of reflections from the lens mounting hardware. Furthermore, experiments can be conducted to determine the effects of adding or removing baffles, irises, special lens coatings, etc.

In the design of vehicles, moments and products of inertia play a central role. Particularly when designing aircraft, weights, the center of gravity, and parameters related to inertia are vital aspects of creating a stable design with high performance. For ground vehicles and fixed structures these figures are quite significant, providing a good estimate of structural loading, and allowing transportation costs to be assessed. Moments of inertia are important in determining what conditions a vehicle may be overturned by maneuvering or impact, and can also help predict the vehicle's handling when maneuvering over rough terrain [DEIT83].

An important extension to the study of simple static and dynamic loading of structures is to consider dynamic stress caused by the impact of a high velocity object with the model. This could be as ordinary as a rock striking a car traveling at 80 kilometers per hour, hailstones falling from the sky, or a collision with another vehicle. On the other hand, if the object being designed is intended for use in a military setting, the range of threats which must be considered is greatly enlarged [WEAV82].

Synthetic aperture radar (SAR) is a technique with which a variety of image information about a distant object can be obtained by correlating multiple radar samples taken from various positions [TOOM82]. While standard radars only report target backscatter and range information, SAR techniques can resolve distinct scattering regions of a target [DEIT84b].

1.4. Tradeoff Analysis

The philosophy adopted at BRL has been to develop a broad set of analyses which are supported from the same geometry database [DEIT84b]. These analyses cover the spectrum from engineering decision aids, to design validators, to drafting and milling interfaces, to the generation of manufacturing drawings, to image generation for management comprehension and sales advantage. Key analysis capabilities have been developed to assess the strength, weight, protection, and performance levels offered by the structures represented by a solid model. Using this analysis information and additional domain-specific applications tools makes it possible to produce highly detailed designs constructed with a philosophy of *system optimization* right from the start [DEIT88]. This should allow the rapid development of systems with the desired levels of performance at the best attainable price.

1.5. Local Patterns of Use

At any stage of a design, the model can be used to make realistic images and engineering drawings. This capability is so powerful that it ordinarily justifies the expense of constructing the solid model. However, the real payoff from building a geometric model comes when it is time to analyze it. The model can be subjected to numerous engineering analyses, allowing the effects of varying many parameters to be studied in a controlled way.

From the perspective of a designer working in a highly interactive analysis environment, the majority of his time is spent using the graphics screen to interact with the model geometry, contemplating it, viewing it, and changing it. After a significant design change, one or more analyses may be conducted on the new design, with the output images being displayed on the graphics screen as they are computed. Then the interaction with the model continues.

2. LOCAL VIEWING OF LOCAL DATA

2.1. Main Styles of Graphics Display

There are several distinct types of graphics support that are required to implement the use of the software just described. First, interaction with the model geometry using **mged** is done in a highly interactive mode using wireframe displays. The **mged** program contains a library of *display manager* modules that support a wide variety of hardware types to produce these wireframe displays, and to deal with knobs, buttons, mice, joysticks, and other hardware-specific input devices.

Second, many of the applications programs produce full-color shaded images. To free the applications programs from having to deal with the specifics of each type of hardware needed to display these images, a hardware-independent framebuffer library called **libfb** has been built. By using this library, each application can be written to perform abstract operations on an idealized 24-bit deep RGB framebuffer, and leave the details of managing the particular hardware to the library. This library also allows applications to open an arbitrary number of different framebuffers, where each one may potentially be made by a different manufacturer.

Third, quite a few of the applications programs produce two dimensional or three dimensional plots for the designer to view. To provide this capability, the traditional UNIX-Plot capability has been extended to provide three dimensional plotting, plotting using floating point coordinates, and plotting in color. This extended UNIX-Plot support is provided to the applications in the form of the library **libplot3**, which is described in more detail below.

Finally, a number of applications presently under development will produce collections of polygons, suitable for display on hardware with intrinsic polygon drawing capabilities. A definite format for these polygons has not been adopted yet.

2.2. Graphics Standards -vs- New Hardware

There are a great many graphics standards available today; so many as to make a mockery of the word "standard". Support for the three main styles of graphic display in the BRL CAD Package could perhaps be provided using one or more standards in existence today, with some effort. However, most of these capabilities were created *circa* 1980, and were oriented towards providing high performance over a well defined and moderately limited domain of operations. As a result, the capabilities provided by the BRL CAD Package overlap with some of the capabilities that exist in current day standard graphics packages. There are a number of compelling reasons for continuing to support the existing BRL styles of operation, even in a climate rich with standards. First, there is no known standard that can provide operations with semantics that match the existing operations and capabilities fairly closely. Second, there is no standard that is implemented across as wide a range of hardware as the present BRL library implementations. Third, there is no known implementation of a standard graphics package that can deliver performance roughly comparable to the existing implementations. Finally, implementing support for each new type of display hardware typically requires from four to forty man-hours of effort, which is significantly less effort than would be required to implement a full standard package on the same hardware, the assumption here being that new hardware does not support the necessary standard packages. New hardware rarely supports the graphics standards of interest.

One of the strongest arguments in favor of standard graphics libraries is based on the notion that if new hardware is always delivered with support for the existing standards, and if all important user code is written to use the existing standards, then moving the existing user code to new hardware should just require recompilation, and everything will work. Unfortunately, in practice there have been several problems encountered when attempting to realize this ideal. Graphics hardware has been evolving at a fantastic pace. Not only have speeds increased, but completely new features never seen before keep appearing. Standards activities, on the other hand, tend to succeed best in fields of endeavor that have a more moderate growth rate, because establishing standards is a very time consuming task. Standards are therefore forced to lag the "state of the art" by half a decade or so.

The strategy adopted early in the development of the BRL CAD Package was based on the premise that a graphics interface standard with suitable power, flexibility, and performance would probably not emerge during the lifetime of the software. Therefore, it was necessary to be positioned such that new hardware could be supported with great agility. All current BRL applications can be satisfied using some combination of the three basic families of graphics operations. In each case, a low level interface has been constructed that provides the absolute minimum set of features possible while retaining the ability to actually accomplish something, following the "small is beautiful" philosophy [KERN84].

These remarks should not be taken as a global condemnation of standards. Because of the highly focused domain of operations, the lack of committee participation in the design, and the orientation towards performance in favor of generality, the support for graphic displays in the BRL CAD Package has been easy to implement, cleanly designed, and highly effective. For uses that fall within the supported domain of operations, these libraries are likely to prove highly satisfying. For other uses, some other solution is required. However, it is important to note that all of the display generation and handling in the BRL CAD Package has been built on these abstractions without significant strain.

The **mged** Display Managers and the **libfb** interface routines both provide capabilities to interactively manipulate display hardware, so high performance is a significant issue. To retain high performance while also achieving good modular software design, an object-oriented interface has been used in both cases. While it is possible to use **libplot3** in an interactive setting, the portable machine-independent plot file provides the standard interface. Here, only a single hardware-specific plotfile display program is needed for each type of hardware.

As a result of the graphics interface strategy selected, it is necessary to create three new software modules each time new hardware is to be supported. However, because of the very careful selection of the interfaces, it typically requires less than one man-week of programming effort to support each new type of hardware. Because only several pieces of exciting new hardware appears each year, this strategy has proven to have been most effective.

2.3. Handling Full-Color Images

2.3.1. The Frame Buffer Abstraction

The framebuffer that **libfb** provides is very simple, and has only a few assumptions. The display surface is rectangular, and is covered with individually addressable pixels. The display surface is *npixels* wide, and *nlines* high. Pixels are addressed as (x,y) using first quadrant Cartesian coordinates so that (0,0) is the lower left corner of the display, with 0 <= x < *npixels*, and 0 <= y < *nlines*.

The rectangular display surface requested must fit entirely on the screen space available for the selected piece of hardware. Therefore, the library does not have to provide any services to clip the display surface to the hardware screen. Multi-pixel writes start at a given x,y coordinate, and proceed to the right (the +X direction) until x = *npixels*−1. If more pixels remain to be written, the y coordinate is increased by one and the x coordinate is reset to zero, so that pixel writing proceeds on the next line *up* on the screen.

2.3.2. Pixels

Each pixel is 24 bits deep, with eight bits assigned to each of Red, Green, and Blue. Pixels are dealt with by **libfb** as atomic units of type **RGBpixel**. The **C** language declaration for this is:

```
typedef unsigned char RGBpixel[3];
```

Where element [0] is Red, [1] is Green, and [2] is Blue. As a result of this, pixels are stored tightly packed in memory, using three bytes per pixel, in a large **unsigned char** array. The sequence of colors in the array is thus Red Green Blue, Red Green Blue....

One seemingly attractive alternative to this format might have been to define the pixel differently, as:

```
typedef struct {
        unsigned char red;
        unsigned char green;
        unsigned char blue;
} RGBpixel;
```

However, such a choice would invite the compiler to provide structure padding, to ensure word alignment of the start of the structure. This could waste a significant amount of memory. The worst case is machines with 64-bit word lengths like the Crays; such a pixel definition would expand each pixel from three to eight bytes!

2.3.3. The Color Map

The 8-bit Red value of a pixel is mapped by a Red colormap (color lookup table) with 256 (2**8) entries to determine the actual intensity displayed on the screen. Green and Blue are handled similarly. Each of the 256 colormap entries is expected to be 16 bits wide, with the values represented as binary fixed point notation, with the radix point to the left of bit zero, so that intensities in the color map can be thought of as varying from 0.0 to 1.0 (or, an **unsigned int** varying from 0x0000 to 0xFFFF, in hexadecimal). The colormap transformation will generally be handled by hardware. When the hardware color map is less than 16 bits wide, the low order bits are discarded. Where the hardware colormap capability is missing, it is simulated by hardware-specific **libfb** routines. Such simulation implies that the hardware-specific routines will need to retain a memory copy of the original un-mapped pixels, to permit the original pixel values to be read back. The colormap is important for certain image processing operations, as well as to permit approximate corrections for the nonlinearity of the display device (gamma correction). For details on better methods for achieving the gamma correction, consult Hall [HALL83].

2.3.4. The “libfb” Interface

The interface to the **libfb** routines has been loosely modeled after the interface to the UNIX **stdio** I/O library. In particular, the routine *fb_open()* returns a pointer to a object of type FBIO. While the contents of the FBIO object are opaque to the application, all other **libfb** routines require that this pointer be provided as the first parameter. All the internal state data for the library routines and the underlying hardware is stored in the FBIO object. With this formalism, it now becomes easy to write applications that utilize an arbitrary number of display devices simultaneously, even if each device is potentially a different type of hardware. The routines that comprise **libfb** are listed in the table below [MUUS87c].

libfb routines	
fb_open(device,width,height)	open the device
fb_close(fbp)	close the device
fb_read(fbp,x,y,buf,count)	read count pixels at x,y
fb_write(fbp,x,y,buf,count)	write count pixels at x,y
fb_clear(fbp,color)	clear to an optional color
fb_rmap(fbp,colormap)	read a colormap
fb_wmap(fbp,colormap)	write a colormap
fb_window(fbp,x,y)	place x,y at center
fb_zoom(fbp,xzoom,yzoom)	pixel replicate zoom
fb_getwidth(fbp)	actual device width in pixels
fb_getheight(fbp)	actual device height
fb_cursor(fbp,mode,x,y)	cursor in image coords
fb_scursor(fbp,mode,x,y)	cursor in screen coords
fb_log(format,arg,...)	user replaceable error logger

A set of buffered I/O routines is also provided, in which a *band* of scanlines is kept in memory, with the library transparently providing band exchanges via bulk framebuffer reads and writes. While using these routines can speed up programs that make “well behaved” single pixel reads and writes, it does not make the drawing of vertical lines significantly faster, since such a line could run through quite a few bands, causing many band exchanges. In practice, very few programs use buffered I/O, but because it can be provided by the library independent of the specifics of the underlying display hardware, it seemed appropriate to implement it as a general capability.

libfb buffered I/O	
fb_ioinit(fbp)	set up a memory buffer
fb_seek(fbp,x,y)	move to an x,y location
fb_tell(fbp,xp,yp)	gives the current location
fb_rpixel(fbp,pixelp)	read a pixel and bump location
fb_wpixel(fbp,pixelp)	write and bump current location
fb_flush(fbp)	bring display up to date

2.3.5. Specifying a Particular Framebuffer

The display to be used is selected by the string parameter *device* in the call to **fb_open()**. If that parameter is a null string, the UNIX runtime environment variable FB_FILE is used. If FB_FILE is undefined, the default display for that system type is selected (a library compile-time option). The format of the *device* string is

[host:]/dev/*device_name*[#]

to designate a hardware device (where the square brackets ‘‘[]’’ denote an optional parameter), or simply

disk_filename

to specify the name of an ordinary file to act as a virtual framebuffer (with the image conveniently being stored in **pix**(5) format into that file). The string prefix ‘‘/dev/’’ is used to identify that this request is for a hardware display device, as opposed to naming an ordinary file. In such cases, the *device_name* part generally will *not* correspond to actual system hardware entries in the directory /dev. If the /dev prefix is not given, then a pathname to an ordinary disk file is assumed.

If a hostname is given, a network connection is opened to the framebuffer library daemon (**rfbd**) on that machine. The remaining part of the string is passed to the remote daemon to complete the open (this generalizes the open to allow multiple "hops" in order to reach a desired host). More about this later.

The *debug* interface prints a message each time a library routine is called, as an aid to debugging. The disk file interface is used to directly manipulate images in a **pix**(5) format file, as if that image was in a framebuffer. Support for specific display hardware is optional, and is determined by compile-time configuration options. The current list of supported devices is given in the table.

libfb supported interfaces	
	STANDARD
file	Manipulate disk image
debug	Print library calls
	OPTIONAL
remote	Network access to remote display
adage	Adage RDS-3000 ("Ikonas")
sgi	Silicon Graphics Iris display (3D)
sgi	Silicon Graphics Iris display (4D)
sun	Sun Microsystems SunView (bw & color)
X	MIT X-windows interface, for X11
ptty	AT&T 5620 w/special Moss layer
rat	Raster Technology One/80
ug	Ultra Graphics

Because the model of the display is that of a traditional framebuffer, providing support for much simpler display devices like Sun workstations requires a lot of software simulation, because of the many features that are not implemented in the hardware. The first device supported (in 1981) was an Ikonas (now Adage RDS-3000), which operates as either a 512x512 or 1024x1024 display with 24 bit pixels. The Ikonas hardware does not allow the current display resolution to be sensed from software, so any program opening the display must ‘‘know’’ whether the framebuffer should be opened in low or high resolution mode. As a result, every program that deals with a framebuffer, even those which have little to do with display size (such as those which read or write colormaps), includes a **–h** ‘‘high resolution’’ flag (for 1024x1024 operation) and a default resolution of 512x512, to make it convenient to open the display with the proper resolution.

2.4. Wireframe Display support in MGED

The **mged** editor is used to specify and modify the geometry in the solid model database. The editor presents a representation of the geometry using wireframe displays.

Support for a variety of display hardware is achieved through the use of an object-oriented programming interface between the main **mged** program and the *Display Manager* routines. When objects are first called up for view, vector representations of the wireframes are created in an internal form, and made available to the currently attached Display Manager.

Display Managers for two main families of display systems are built with a single common formalized interface. First, display systems with displaylist capability are supported. When objects are first called up for view, the internal vector lists are converted to device-specific formats and placed into displaylist memory. All subsequent operations are performed by transmitting displaylist updates, and replacing viewing matrices. Second, display systems with no displaylist memory are supported. For these displays, each time the main **mged** program signals that the screen needs to be updated, the current viewing matrices are combined with the internal vector list, the vectors are clipped, and output to the display. At present, all versions of **mged** have support for these types of display devices by default:

plot	any UNIX-plot filter
tek	Tektronix 4014 and family
tek4109	Tektronix 4109

These optional display devices are also supported, when specifically configured:

sgi	Silicon Graphics Iris 3D
sgi	Silicon Graphics Iris 4D
sun	Sun Microsystems SunView
X	MIT X-Windows X11R3
X10	MIT X-Windows X10R4
vg	Vector General 3300
mg	Megatek 7250
rat	Raster Tech One/180,380
rat80	Raster Tech One/80
ps	Evans and Sutherland PS-300
mer	Megatek Merlin 9200
sab	Saber Technology

2.5. libplot3: Plotting Support

UNIX systems have traditionally supported device-independent plotting on a variety of displays, using the **libplot**(3) library routines to produce **plot**(5) format plot files. These plot files can be displayed on a variety of displays using the **plot**(1) program (renamed **tplot**(1) in SystemV). Standard device support on SystemV includes the DASI 300, DASI 450, and Versatec D1200A. Standard device support on Berkeley UNIX includes the Tektronix 4013, 4014, DASI 450, DASI 300, AED 512, BBN Bitgraph, Imagen laserprinter, HP 2648, Versatec D1200A, and the Benson Varian plotter. Crude plots can also be printed on any terminal with cursor-addressing capability. The BRL SystemV extensions [GWYN85] provide support for the Selanar HiREZ-100, HP 2648, HP 7750A plotter, HP 2686A LaserJet, Megatek 7255, Tektronix 4014, Tektronix 4105, Trilog ColorPlot II, Teletype 5620, and Digital Engineering's Retrographics VT100.

The standard plotting interface permits plotting in two dimensions, using 16-bit signed integer coordinates. The intervals to be used in X and Y must be specified in advance. There is support for text strings, circles, arcs, points, and lines, with an option for selecting line types. Plot files created using **libplot** are binary, and portable to other machines. Unfortunately, due to the origins of this software on the PDP-11 and the VAX, the integers are written in VAX (''Little-Endian'') byte order.

BRL's **libplot3**(3) library provides a great many additional features over the standard UNIX **libplot**(3) library. Colors may be specified as 24-bit RGB quantities. Additional routines exist to plot using three coordinates. The two dimensional and three dimensional integer-valued routines have been supplemented with routines that take floating-point values. These floating point values are written into the plot file in a transportable, machine-independent binary format using **htond**(3). The 3-D floating point routines have two interface styles, one where each coordinate is a formal parameter to the subroutine, and one where location in space is passed as a pointer to a vector. The second form is especially convenient when using the vector math routines.

The **libplot3**(3) library provides a powerful and portable mechanism for creating 2-D and 3-D plot files that are upwards (but not downwards) compatible with the original UNIX **libplot**(3) library. In addition to the low level routines mentioned so far, additional library routines exist for drawing coordinate axes, scaling data, etc, so that this capability can be used for traditional data plotting [MUUS78], as well as for representing three dimensional wireframes.

3. VIEWING OF REMOTE DATA

3.1. The Motivation for Complicating Graphics by adding Networking

Collaboration is a very important aspect of creative endeavors, both scientific and artistic. Collaboration implies having more than one person involved, which implies that there is probably more than one workstation or computer involved, which implies that some form of machine communication is required.

In ''the old days'', graphics displays were either terminals attached to a mainframe with an ordinary serial line, such as the Tektronix 4014, or graphics displays were high-speed peripherals connected directly to an I/O channel, such as the Evans and Sutherland PS-1. Networking involved little more than connecting all the terminals to *the* computer. In either case, sharing of data between users was automatic, because all users used the same computer.

Such simplicity no longer exists. In the modern computing environment, computers are everywhere. To retain the benefits of synergistic interaction, the barriers between machines must be removed.†

3.1.1. The Modern Computing Environment

The state of most computer facilities in the late 1980s can be best summarized by the word ''diversity'' [QUAR86]. A single computer is no longer able to serve the needs of an entire facility; in many cases a single computer may be allocated to serve the needs of a small group of people, or even an individual. Fortunately, the cost per unit of processing power has fallen steadily and rapidly for the last fifteen years. As a result of the increase in demand and decrease in cost, computer systems have proliferated wildly, appearing in all aspects of business, education, and government.

Proliferation of computer systems within organizations has served to highlight the importance of networking. Some organizations have opted to depend on a single vendor for all their hardware needs; such organizations have typically been able to enjoy a reasonable degree of interoperability between their systems, but often at a sharp expense of having to forgo more powerful or less expensive offerings from other vendors. Educational and government institutions, always forced to be very short-term cost conscious, have often been forced to diversify. Having a collection of dissimilar machines has complicated the task of achieving communication between them.

† For authorized users. A sensible security posture must still prevail, but that is the topic for some other paper.

Accomplishing communication between two different vendor specific communication systems has always been possible, although often expensive. The development of the vendor independent TCP/IP protocol family [FEIN85], the use of TCP/IP on the ARPANET, and the development of fast, inexpensive local area network (LAN) technologies like Ethernet [XERO80] has resulted in a common communication standard that is in widespread use. The TCP/IP protocol family took the promise of Open Systems Interconnection (OSI) from an ideal to a production capability in 1982, and TCP/IP has been in widespread use ever since. Through the use of the TCP/IP protocols, computers from any manufacturer can communicate reliably, effectively, and efficiently with all other computers on the network.

With processing power getting less expensive, computers becoming physically smaller, and the invention of Ethernet making local area networking inexpensive enough to be practical in any facility, the emergence of the workstation was inevitable. Personal computers (PCs) and workstations share the common trait of having a processor, main memory, and video display dedicated to working on the problems of a single user. A workstation is distinguished from a personal computer mainly by the ability to multiprogram, ie, the workstation has the ability to interleave the running of more than one program without interference. The ability to multiprogram is necessary to make a computer system into a good network citizen. The network environment is populated by many ‘‘daemon’’ programs that provide services to, and demand services of other machines on the network; these programs use a small amount of processor time on an ongoing basis to conduct those services.

Workstations are a commonplace item in all current computer installations. Recent advances in VLSI have made it clear that the workstation is the heir apparent to the throne once held by the ordinary terminal. For example, the Sun-3/50 is a 1-1.5 million instruction per second (MIPS) Motorola 68020 based machine with a Motorola 68881 floating point unit, 4 Mbytes of main memory, a 10 million bit per second (Mbps) Ethernet interface, and an 1152x896 pixel display -- yet as of this writing the selling price in the USA is only four thousand dollars, a fraction of the cost of a graphics terminal only a decade earlier. Furthermore, high performance workstations offer compute performance that rivals that of many common minicomputers, often at a small fraction of the price. What is happening in the computer industry is that while the super-minicomputers grow upwards to compete with the SuperComputers, the workstations grow upwards to compete with the conventional minicomputer.

3.1.2. Computer Categories

A properly balanced computer network has a mix of different resources. For a laboratory working on science and engineering problems, the resources typically fall into three categories: SuperComputers, super-minicomputers, and workstations. It can sometimes be difficult to distinguish between different types of computers, so a small diversion to define terms is necessary.

<table>
<tr><th colspan="3">Processor Categorization</th></tr>
<tr><th>Category</th><th>MFLOPS</th><th>I/O</th></tr>
<tr><td>SuperComputer</td><td>>500</td><td rowspan="3">High performance I/O channels with dedicated I/O processors. 3-15 Mbytes/sec per channel.</td></tr>
<tr><td>Mini-SuperComputer</td><td>>50</td></tr>
<tr><td>Mainframe</td><td>3 - 50</td></tr>
<tr><td>Super-Minicomputer</td><td>7 - 100</td><td rowspan="2">Medium performance I/O controlled by same CPU(s) that run user code. 1-3 Mbytes/sec per channel. Hardware error recovery.</td></tr>
<tr><td>Minicomputer</td><td>0.5 - 7</td></tr>
<tr><td>Workstation</td><td>0.1 - 3</td><td rowspan="2">Low cost. Low speed I/O controlled by CPU. Software error correction. <1 Mbytes/sec per channel.</td></tr>
<tr><td>Microcomputer</td><td>0.0 - 0.5</td></tr>
</table>

A SuperComputer
represents the fastest processing and I/O capabilities available. The instruction set is optimized for scientific processing. The SuperComputer processor is best utilized primarily for computationally intensive processing loads, because of the long time needed to quench the processor pipelines and save the state of the processor. Due to the complexity of the processor, some machine-specific software development is often necessary to obtain maximum performance.

A Mini-SuperComputer
is processor which executes the exact same instruction set and binary software as a SuperComputer, but at a much slower speed and at a much lower cost. These systems are typically purchased to offload software development from SuperComputers, or in circumstances where a full SuperComputer is not required, or prohibitively expensive. Having the exact same hardware complexity as a full SuperComputer, similar machine-specific software development is necessary for maximum performance.

A Mainframe computer
typically has an instruction set balanced for general-purpose programming and business applications, and usually runs a business-oriented operating system.

A Super-Minicomputer
is a very large mini-computer architecture device, utilizing lower performance I/O devices and a collection of mini-computer processing elements to make a system which has a substantial aggregate processing power, but retains the low-overhead interactive orientation of the minicomputer architecture. Suitable for highly interactive tasks, especially graphics display, as well as medium-scale numeric processing. I/O devices are typically controlled by the same processors that runs users programs, with hardware to perform I/O error correction.

A Minicomputer
is intended for interactive computing, to be shared between a small to medium community of users. The hardware is optimized for interactive processing, and generally low cost, with modest I/O bandwidths. I/O devices are typically controlled by the same processor that runs users programs, with hardware to perform I/O error correction.

A Workstation
is intended for a single user at at time, with high resolution monochrome or medium resolution color displays, and high interactivity for a single application. Only small amounts of processing can be performed local to the workstation.

A Microcomputer
is cheap. Typically they lack multi-programming, and lack TCP/IP network support, making the effective utilization of multiple microcomputers difficult. However, they are often extremely cost-effective sources of computing power for clerical and administrative applications.

3.1.3. Network Concepts, Standard Protocols

The International Standards Organization (ISO) has adopted a reference model for Open Systems Interconnection (OSI) that divides network service into seven layers of abstraction, to permit the different aspects of a complete network implementation to be more easily considered [TANE81].

7	Application	(User)
6	Presentation	Data transformation
5	Session	Create connections
4	Transport	Segmentation & Reliability
3	Network	Routing
2	Link	Build bit streams
1	Physical	Electrical transitions

Except for the discussion about network bandwidth, which is primarily determined by the characteristics of the hardware that is providing a particular Physical Layer, this paper will simply assume that a reliable, full duplex, eight-bit binary path is available between pairs of computers, on demand, and that the Session

Layer and lower layers will handle all necessary details. Instead, the focus is on the activities in the Presentation and Application Layers, to take the raw network capability, and use it to provide meaningful graphics across network links.

There are quite a few network protocol families in existence that can be viewed using the ISO model (with varying degrees of success). Most are vendor specific, and two are standards devised by non-vendors.

Vendor	Protocol
Xerox	NS
Digital	DECNET
IBM	SNA
US Govt	TCP/IP
ISO	TP4

3.1.4. The Need for Bandwidth

People who are tasked with planning for network bandwidth requirements generally are not prepared for the volume of data that graphical applications generate and consume. At BRL, there are two functional requirements for network bandwidth, which define the minimum acceptable performance of a network, and they are based on user impatience factors [IBM82]. Any user should be able to achieve file transfer rates at the rate of one megabyte per minute, or 140 Kbits per second. A CAD user should be able to retrieve a 512x512 24-bit deep image in 30 seconds, or 210 Kbits per second. These values are bare minimums. More reasonable performance would be to transfer data at the rate of one megabyte per ten seconds, or 838 Kbits per second, with the image transfer happening in three seconds, or 2 megabits per second. While the minimum values are no problem, the more desirable speeds begin to approach the limits of conventional technology (considering that the network supports hundreds of users).

Various LAN Speeds (Mbps)	
0.056	Point-to-Point
1.544	Point-to-Point
10	Ethernet
10	ProNet-10 Ring
16	DEC PCL-11
50	Hyperchannel
80	ProNet-80 Ring
100	FDDI
1200	UltraNet

Fortunately, for graphics problems there is an upper bound on the amount of network bandwidth that is likely to be required per user. In this case, the limiting factor is the bandwidth of the human eye-brain system, which has been estimated at approximately 10 Gbps (10,000 Mbps). For most purposes, the eye can be deceived with images of significantly lower resolution. The best example of this is the 4.2 Mhz bandwidth of an NTSC television signal [CONR80], which provides low but very usable image quality. For computer graphics applications, more spatial resolution than that provided by NTSC is often desirable, while 30 frames per second animation rates are only needed for the most demanding work. The table below provides data rates for several types of animation.

Resolution	FPS	Data rate
512x512x24	15	95 Mbps
512x512x24	30	190 Mbps
1024x1024x24	15	375 Mbps
1024x1024x24	30	755 Mbps
1024x1024x24	60	1510 Mbps
2048x2048x24	24	2415 Mbps

There are not many computers that can compute animation sequences in real time, although there are quite a few that have I/O systems capable of reading, postprocessing, and displaying precomputed data at these data rates. The first line in the table describes a data rate comparable to the maximum throughput of the new generation of LAN technology, the proposed FDDI (fiber-optic data distribution interconnect), 100 Mbps. To date, only the Ultra device is capable of operating in this range of data rates, with a peak speed of 1200 Mbps, but a variety of other devices of comparable bandwidth are presently in development [MCCO87].

3.2. The BRL Remote Frame Buffer Capability

The machine at BRL which had the nicest display hardware on it had one of the slowest processors (a VAX 11/780). To allow computation and display to occur on separate machines, the **libfb** frame buffer library was given support for an extra type of display hardware: the remote frame buffer. The goal of this capability was to be able to direct graphics operations to any display screen attached to the network, merely by specifying in the *device* string the name of the host computer and the name of the display device. (Refer back to the section on **libfb** for a discussion of the three methods for providing the *device* string, and the syntax). The details of establishing the network connection to the appropriate computer, as well as the hardware-specific display support, should all be entirely transparent to the user, except perhaps for some performance variations.

Detection that the given *device* string indicates the use of a network display is handled by the **libfb** routines that implement the fb_open() function; rather than vectoring to a module to support a local display, the fb_open() function forces all operations to vector through the if_remote.c module. When the if_remote.c module is called to open the framebuffer, it assumes the role of a ‘‘network client’’, and causes a network connection to be established to the ‘‘network server’’ on the indicated remote machine. The client then passes the hardware device specification to the server. The server process calls fb_open() on it’s copy of **libfb**, and returns the status code (success or failure) to the client, which returns the status code to the calling application.

The routines in the if_remote module convert all the parameters to the library calls into machine independent messages. These messages are sent from the client to the server using the ‘‘package’’ (PKG) protocol [MUUS87c]. When the server receives a message, it determines which operation is being requested, converts the machine independent values in the message back into the native format of the server machine, and then invokes the intended library routine.

The definition of the protocol exchanged by the client and server machines is a level seven (Application) protocol that depends on level six services (Presentation) for the external data representation conversions. The machine independent messages generated by this protocol are transmitted via **libpkg**, which provides level six (Presentation) and level five (Session) services. **libpkg** provides capabilities for the transmission of both synchronous messages and asynchronous messages. Synchronous messages are used by the if_remote protocol when return codes from the server are significant. In this case, the client library routine does not return until a reply message has been received from the server. For significantly improved performance, the fb_write() routine uses asynchronous messages, so that the client library routine returns a ‘‘success’’ indication immediately, while the network and server continue to work on displaying the data. This allows the production of data and the display of the data to proceed at the maximum possible rate. With this design, the library depends on reliable delivery services from the level four (Transport) services of the particular communication protocol.

One of the delightful features of this design is that the remote framebuffer capability can be stacked to an arbitrary depth. That is to say, one could specify the *device* as

host1:host2:/dev/*device_name*[#]

In this case, the original client would contact *host1* as a server. *host1*, in processing the remainder of the *device* string, would vector to if_remote.c, thus acting as a client, and would contact *host2*, which would perform as the actual server. This sort of an arrangement was a good test of the symmetry of the protocol, and ensured that no implementation details had been overlooked. Stacking servers in this manner is rarely useful, but the capability comes in handy just often enough so as to be worth keeping around. One use is to bridge across a change underlying protocols. For example, assume three hosts, A, B, and C, where A and B communicate via the TCP/IP protocols, and B and C communicate via the DECNET protocols. There is no way for an image to be transmitted directly from A to C. However, in this case, machine B can act as an application-level ‘‘relay’’, and bridge the gap. Another use for stacking servers is to bypass network software or traffic routing problems, by forcing the data to flow through a suitable intermediary.

By careful implementation, the use of remote framebuffers across reasonably high performance network connections has performance which is only slightly slower than local display performance. The network framebuffer concept is a very powerful one; once accustomed to it, users refuse to deal with more restrictive environments. This capability has been used to conduct data analysis at NASA facilities, with live graphics transmitted from BRL computers, miles away.

Will new standards like the MIT X Window System will make the BRL framebuffer library unnecessary? In the short term, no. X Version 11 does not support 24-bit color images, nor does it provide a powerful enough interface for controlling many important framebuffer operations such as the colormaps, pan and zoom. However, to provide compatibility with X-based systems, an if_X module implementing the framebuffer library functions is being developed, so that all systems that run X will have **libfb** support.

3.3. The Remote MGED Display Manager

In a manner similar to that just described for **libfb**, there is a special display manager for **mged** called dm-net.c that communicates with an mged server process on a remote machine, and facilitates network execution of **mged**. This software is still considered experimental, but it has been demonstrated to work correctly. However, the proper level of interactive response depends on having a network path between the client and the server with low latency and low packet loss, so that display motions remain smooth.

The underlying mechanism is very similar to that just described for **libfb**, with specially formatted messages carried via **libpkg** between the client and server processes, and it will not be further described here.

3.4. The PKG Protocol

The ‘‘package’’ (PKG) protocol is a critical ‘‘enabling technology’’ upon which a great many BRL network applications are built. The **libpkg** services bridge levels six (presentation) and five (session) in the OSI model, and is layered on top of a virtual circuit provided by the level five and four (transport) capabilities of the native operating system. One of the main purposes of **libpkg** is to insulate the applications programmer from the details of establishing and managing a network connection.

libpkg frees the programmer from concerns about storage and buffer management, and other details relating to the network communications internals, by allowing exchange of messages of any size (up to 2^{32}-1 bytes), with automatic allocation of sufficient dynamic memory for the receive buffer handled by the library. Thus, messages exchanged by users of **libpkg** may be of arbitrary size. There is no requirement for different messages of the same type to have the same size. It is perfectly reasonable to transmit messages with no data, for example, to signal that a particular event has happened. The **libpkg** interface supports a dynamic mix of synchronous and asynchronous message transmission paradigms, allowing the implementation of traditional blocking remote procedure call operations, as well as permitting non-blocking operations (such as used by the fb_write() operation described above), and ‘‘surprise’’ asynchronous operations (such as used by the fb_log() operation).

Each message is transmitted across the virtual circuit with an eight byte header, to permit the receiving **libpkg** routines to properly receive and dispatch the message.

```
 0                   1                   2                   3 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Magic Number          |         Message Type          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Count of actual data bytes                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
:                                                               :
:                             [Data]                            :
```

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

All three header fields are transmitted in Big-Endian byte order; the format of the data field is determined by the calling application and is opaque to the **libpkg** routines. The *magic number* field is set to the value 0x41FE hexadecimal, and is used by the receiver routines to defend against various system failures, such as an application program writing data to the wrong file descriptor (the network connection, in this case). The *type* field is provided by the caller of pkg_send(), and is used on the receiving end to appropriately dispatch the message: either a blocking read is satisfied by the arrival of the appropriate message type, or an asynchronous event is vectored through the pkg_switch table and the message is dispatched to an application specific handler.

Typically, PKG is layered on top of a TCP connection, although PKG has also been run over DECNET and X.25. Multiple PKG connections per process are supported, which is used by **remrt**. When using TCP, the TCP option SO_KEEPALIVE is enabled so that all communications failures and remote system failures are noticed by the TCP layer after an appropriate time interval, avoiding the need for application-level timeouts. **libpkg** handles the incremental aggregation of received data into full messages, freeing the calling application from having to handle the details of short reads. The Berkeley UNIX *select*(3) system call provides the ability to easily handle asynchronous communications traffic on multiple connections.

libpkg Routines	
pkg_open	Open net conn to host
pkg_permserver	Be permanent server, and listen
pkg_transerver	Be transient server, and listen
pkg_getclient	Server: accept new connection
pkg_close	Close net connection
pkg_send	Send message
pkg_waitfor	Get specific msg, do others
pkg_bwaitfor	Get specific msg, user buffer
pkg_get	Read bytes, assembling msg
pkg_block	Wait for full msg to be read

This protocol had originally been developed to make the remote **mged** display possible, but later found uses in command and control experiments [CHAM86], distributed processing programs [MUUS87d], and other network applications. The functionality provided by **libpkg** is not significantly greater than the functionality provided by the virtual circuit support of the basic operating system, and amounts to to only about 1300 lines of source code, yet it provides a convenient and worthwhile abstraction of the underlying network mechanism that makes the implementation of distributed applications very much simpler.

3.5. External Data Representation

When transmitting information on a network, careful attention must be given to the format and order that data is transmitted in. In a heterogeneous network the internal processor architectures of the communication endpoints are likely to be different, yet effective and accurate transmission of data is still desired.

Moving binary data between disparate computer architectures is not a new idea, and has been done for at least three decades. So far, the communication has typically involved specific pairs of computers, typically a mainframe or supercomputer sending results to some kind of "front-end". In such cases, one machine would be given the task of always converting the data to and from the format used by the other machine. When constructing distributed applications that run on a wide variety of hardware that can all communicate with each other, it is no longer possible to predict in advance how the machines may be coupled together.

Two strategies exist for coping with such heterogeneity. Either (1) all machines can agree to adopt a single standard for communication, or (2) all machines must have the software to enable them to convert internal data to the internal formats of all other machine types. The first strategy is weakest when two computers of similar type wish to communicate, where their internal format is not the standard format. Transmission of a block of data would therefore require converting it to the standard format, sending it, and converting it back to the internal format, when using the internal format would have been acceptable *in this case*. The second strategy always performs well when similar machines converse, because no format conversion is required. However, this strategy creates a significant software coding and testing burden on the implementors when the number of different formats **N** becomes significant, ie, **N** greater than 3, because of the need to write **N****2** conversion routines. Also, the task of adding support for a new type of machine becomes formidable: **N** new conversion routines must be written, and every existing machine needs to be updated to support the new format.

The overall strategy favored by the DARPA Network Research community has been to establish a single standard representation for each type of communication, and then to have all machines communicate using that single standard. The BRL CAD Package has continued in this fine tradition. The external data representation described here fits into level six (Presentation) of the OSI reference model, and serves generally the same purpose as the ISO X.409 Abstract Syntax Notation (ASN.1). Note that both the data representation described here and the Sun XDR standard [SUN87] use implicit typing, ie, no indication of the data format becomes part of the external data representation, while the ASN.1 standard includes many details of the of the data format and data structuring in the external representation of the data.

3.5.1. Ordering and Numbering Bits in a Byte

In the discussion that follows, it is assumed that the underlying hardware handles the disassembly and reassembly of serial bit streams into 8-bit bytes in a transparent manner. In this document, the leftmost bit in a diagram is the high order or most significant bit (MSB), and this bit is labeled bit zero. For example, the following diagram represents the value 170 decimal, 0xCC hex, 0252 octal:

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|1 0 1 0 1 0 1 0|
+-+-+-+-+-+-+-+-+
```

3.5.2. Transmitting sequences of bytes

The organization and order of transmission of data described in this document is specified at the byte level. The order of transmission of a group of bytes is the normal order in which they are read in English. In the following example, the bytes are transmitted in ascending numerical order (ie, 1, 2, 3, ...).

```
 0                   1                   2                   3 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       1       |       2       |       3       |       4       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       5       |       6       |       7       |       8       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       9       |      10       |      11       |      12       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This convention is often referred to as the "Big-Endian" order [COHE80] because the big (most significant) end of the binary number is transmitted first. Big-Endian notation is the standard order in the TCP/IP protocols of the DARPA InterNet [POST81], now formalized as MIL-STD-1777 [DCA83]. The Big-Endian order is also the internal format commonly used by IBM, Gould, Sun, SGI, Alliant, Apple, and many other computers. For these reasons, the Big-Endian order is used for the BRL CAD Package external data representation.

3.5.3. Transmitting Integers

All integers are transmitted in Big-Endian, or "network order", regardless of the internal representation of the particular machines. To send a signed 16-bit quantity, the number is represented in twos-compliment form, and the byte sequence is:

```
 0                   1           1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       1       |       2       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S|MSB                       LSB|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where "S" is the sign bit, MSB is the most significant bit, and LSB is the least significant bit. Byte "1" is transmitted first, and byte "2" is transmitted second.

To send a 32-bit signed integer, the number is represented in twos-compliment form, and the byte sequence is:

```
 0                   1                   2                   3 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       1       |       2       |       3       |       4       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S|MSB                                                       LSB|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Unsigned integers are encoded similarly, occupying either 16 or 32 bit positions.

3.5.4. Transmitting Strings

Character strings are represented using the ASCII character set. They are placed into the external message without regard for alignment, and are terminated by a zero byte (all bits off). There is no limit to the length of a character string.

```
+---------+---------+---//---+---------+---------+-+-+-+-+-+-+-+-+
|  Byte 1 |  Byte 2 |   ...  |Byte N-1 | Byte N  |0 0 0 0 0 0 0 0|
+---------+---------+---//---+---------+---------+-+-+-+-+-+-+-+-+
```

3.5.5. Floating Point Representations

The most portable representation of a floating point number would be to encode it as a printable ASCII string. However, using a printable ASCII representation would require 16 decimal digits to contain the mantissa, two digits for the signs of the exponent and mantissa, three decimal digits for the exponent, and at least two additional characters for the mantissa decimal (radix) point and the exponent marker. Such an encoding would look like "-1.234567890123456e−103", and requires at least 23 characters (bytes).

Exchanging floating point numbers in binary form is a very compact representation, using a small and constant amount of bandwidth: eight bytes. Therefore, using an eight byte binary format requires about one third the network bandwidth as the printable ASCII representation. In addition to the significant bandwidth savings, using a binary format is also likely to require significantly less computer time for format conversion to machine-specific formats, because of the strong similarities in internal binary formats and the simplicity of using bit operations to implement the conversions.

3.5.6. Binary Floating Point Representations

Virtually all modern computers utilize twos compliment binary format as the internal representation for integers. Thus, the only issue with transmitting integers was to define a byte order; the underlying representation was already common to all the computers. In the case of floating point numbers, the situation is more complex. Not only does the byte ordering need to be defined, but the network representation of the floating point number itself needs to be defined.

Just as integers come in various sizes, floating point numbers also come in different sizes, the most common of which are called ‘‘single precision’’ (typically 32 bits), and ‘‘double precision’’ (typically 64 bits). This discussion will focus entirely on the double precision format.

Between the various vendors, there have been quite a few different internal representations for floating point numbers. Typically, each format has several advantages over the others. The main alternatives are:

1) IEEE Standard 754 double precision floating point is most popularly used in new microprocessor designs, and is found in a wide variety of new computer systems, especially new workstations. As stated by Kahan [KAHA86] "This standard is on its way to becoming more widely adopted than any other design for computer arithmetic. VLSI chips that conform to some version of that standard have been produced by a host of manufacturers, among them Intel i8087, i80287, Motorola 68881, National Semiconductor 32081, Weitek WTL-1032 and WTL-1165, Western Electric (AT&T) WE32106, Zilog Z8070" and MIPS FPS2000. The radix is 11 bits wide, in base 2, excess-1023 format. The fraction has 52 significant bits, a ‘‘hidden’’ leading one to the left of the high-order bit, and a radix point to the *right* of the hidden bit, giving roughly 16 decimal digits of significance. Overflow threshold is 2.0**1024, or 1.8e+308. Underflow threshold is 0.5**1022, or 2.2e−308. There are explicit representations for *infinity*, and not-a-number (NaN), both Signaling Nan and Quite NaN.

```
 0                   1                   2                   3    6 6
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1  2 3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+//-+-+
|S|   Exponent            | Fraction                                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+//-+-+
```

When the exponent is 2047 (all bits set), a special condition exists. If all fraction bits are zero, the value is infinity times the sign. If the fraction is non-zero, and the MSB (bit 12) of the fraction is zero, then this represents a signaling NaN, otherwise this represents a quiet NaN.

2) IBM System/360 double precision (eight-byte, or ‘‘long’’) floating point [IBM70], is also used by other vendors such as Gould. The radix is 7 bits wide, in base 16, excess-64 format. The fraction has 56 significant bits, having a radix point to the left of the high-order bit, with no hidden bits, giving roughly 17 decimal digits of significance. Overflow threshold is 16.0**63, or 7.2e+75. Underflow threshold is 16.0**-65, or 5.4e−79. There is no representation of *infinity*. There is no representation of not-a-number (NaN).

```
 0                   1                   2                   3    6 6
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1  2 3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+//-+-+
|S|   Exponent    | Fraction                                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+//-+-+
```

3) Digital Equipment Corporation’s VAX-11 double-precision ‘‘D’’ format, also used on the earlier PDP-11 machines. The radix is 8 bits wide, in base 2, excess-128 format. The fraction has 56 significant bits, with a ‘‘hidden’’ leading bit, giving roughly 17 decimal digits of significance. Overflow threshold is 2.0**127, or 1.7e+38. Underflow threshold is 0.5**128, or 2.9e−39. This range of representable numbers is comparatively narrow. There is no representation of *infinity*. There are reserved values to represent signaling NaN [DEC76].

```
 0 0    1 1 1 1    3 3 3 3    4 4 4 4 5 5 5 5 5 5 5 5 5 5 6 6 6 6
 0 1    4 5 6 7    0 1 2 3    6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
+-+-+//-+-+-+-+//-+-+-+-+//-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Fract_D | Fract_C | Fract_B |S|    Exponent     |    Fract_A   |
+-+-+//-+-+-+-+//-+-+-+-+//-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Note that the actual fraction is made by combining (left to right) Fract_A, Fract_B, Fract_C, and Fract_D. Also note that this ordering is neither pure "Big-Endian" or "Little-Endian", but a peculiar mixture of the two, being Big-Endian within each 16-bit unit, but Little-Endian when combining the 16-bit sections.

4) Cray Research Incorporated full precision floating point. The radix is 15 bits wide, in base 2, excess-16384 format. The fraction has 48 significant bits, having a radix point to the left of the high-order bit, with no hidden bits, giving at least 14 decimal digits of significance. Overflow threshold is 2.0**8191, or 1.0e+2466. Underflow threshold is 0.5**8192, or 1.0e−2466. There is no representation of *infinity*. There are reserved values to represent NaN. Whether the NaN is signaling or quiet depends on the setting of the hardware Floating-point Mode flag [CRAY82].

```
 0                   1                   2                   3     6 6
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1   2 3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+//-+-+
|S|          Exponent           |              Fraction              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+//-+-+
```

5) Digital Equipment Corporation's VAX-11 double-precision "G" format. This format is considered optional, and is not supported on all VAX machines, with hardware support available only on some models. Thus it is much less frequently used than the DEC "D" format, above, even though it is very close to the IEEE format. The radix is 11 bits wide, in base 2, excess-1024 format. The fraction has 52 significant bits, having a "hidden" leading one to the left of the high-order bit, and a radix point to the *left* of the hidden bit, giving roughly 15 decimal digits of significance. Overflow threshold is 2.0**1023, or 8.9e+307. Underflow threshold is 0.5**1024, or 5.5e−309. There is no representation of *infinity*. There are reserved values to represent signaling NaN [DEC82].

```
 0 0    1 1 1 1    3 3 3 3    4 4 4 4 5 5 5 5 5 5 5 5 5 5 6 6 6 6
 0 1    4 5 6 7    0 1 2 3    6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
+-+-+//-+-+-+-+//-+-+-+-+//-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Fract_D | Fract_C | Fract_B |S|       Exponent       |Fract_A|
+-+-+//-+-+-+-+//-+-+-+-+//-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

6) The native double-precision format of the Convex machines (some of which also have IEEE capability), is a Big-Endian version of Digital Equipment Corporation's VAX-11 double-precision "G" format described above, and has the same properties [CONV88].

```
 0                   1                   2                   3     6 6
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1   2 3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+//-+-+
|S|   Exponent          | Fraction                                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+//-+-+
```

The salient features of these different formats are summarized in this table:

Format	Radix bits	Fraction bits	Decimal digits	Overflow threshold	Underflow threshold
IEEE	11	52+1	16	1.8e+308	2.2e−308
IBM 360	7	56,53	17	7.2e+75	5.4e−79
VAX "D"	8	56+1	17	1.7e+38	2.9e−39
Cray	15	48	14	1.0e+2466	1.0e−2466
VAX "G"	11	52+1	15	8.9e+307	5.5e−309
Convex	11	52+1	15	8.9e+307	5.5e−309

3.5.7. A Standard for Network Floating Point Numbers

From all of the possibilities just mentioned, the ANSI/IEEE Standard 754 for Binary Floating-Point Arithmetic [IEEE85] was chosen. This choice was made on the basis of the IEEE format's overall technical merits, including extended dynamic range, it's status as an accepted and respected standard, and on the basis of it's widespread implementation in common graphics workstations. This choice was made even though this format has a few less bits in the fraction than other formats, most notably that of the VAX, and thus delivers slightly less accuracy. The differences in accuracy were considered to be small enough to not matter; depending on more than 15 decimal digits of accuracy is an impediment to portability in any case, and can not be improved upon in the external data representation.

Therefore, when communicating binary floating point data across a network connection, all floating point values must be converted from the host-specific representation to the 64-bit IEEE representation before transmitting it, and upon reception, all floating point values must be converted from the 64-bit IEEE representation to the host-specific representation. When the IEEE representation is transmitted, it is to be transmitted in Big-Endian byte order. That is, bits 0-7 are transmitted in the first byte, bits 8-15 are transmitted in the second byte, and so forth.

3.5.8. Floating Point Converter Interface

A pair of library routines have been written (in the C [RITC78b] programming language) to convert between a local host computer's C **double** data type (typically 64-bit double precision) and the *network floating point format*, which is the 64-bit IEEE double precision representation, in "network" (ie, Big-Endian) order. The subroutines are defined in C as:

```
htond( netptr, hostptr, count );
unsigned char *netptr;
unsigned char *hostptr;
int count;

ntohd( hostptr, netptr, count );
unsigned char *hostptr;
unsigned char *netptr;
int count;
```

The names are acronyms for Host TO Network Double (**htond**) and Network TO Host Double (**ntohd**), in the style of the existing Berkeley network data representation subroutines, such as **htons** (host to network short), **htonl** (host to network long), etc. However, unlike their Berkeley counterparts, these routines have two significant differences. First, neither the input or output buffers need to be word aligned, to permit the greatest flexibility in converting the data, even though this may impose a speed penalty on some architectures. This property can be especially useful when creating tightly packed network messages containing mixed data types, freeing the programmer from the necessity to add "pad" bytes to ensure alignment. The concept of alignment is hard to make both portable and storage efficient, and is thus properly avoided as a subroutine constraint. Second, these subroutines operate on a sequential block of numbers, rather than on just a single value. This allows entire arrays to be conveniently converted with a single subroutine call, thus improving the clarity of the code, saving on subroutine linkage execution costs, and allowing the hope for effective vectorization of the subroutines on those machines with vector hardware capability.

3.5.9. Significance

The significance of having established a standard for the exchange of floating point numbers should not be underestimated. Most existing network applications restrict themselves to the transmission of integers, either for reasons of efficiency, or to simplify the conversion problem. For the purposes of scientific computing, converting data to integers is rarely acceptable, because of the requirement for having a large dynamic range available while also maintaining many digits of significance.

Having the capability of conveniently and efficiently transmitting floating point numbers over the the network has given rise to a whole variety of important new capabilities. One example which has already had significant impact, was the result of including the 3-D floating point plotting capabilities of **libplot3**, which provides a machine independent ‘‘meta-file’’ capability for plots of full resolution, allowing arbitrary amounts of zooming to examine extraordinarily fine details in the data.

The **htond** and **ntohd** routines have been implemented and tested on a wide variety of machine families, including the Cray X-MP, Cray-2, Alliant, Sun-3, Silicon Graphics 3-D and 4-D, DEC VAX, and Gould PowerNode. When combined with the Berkeley routines **htons**, **ntohs**, **htonl**, and **ntohl**, or with the **libpkg** versions (**pkg_gshort**, **pkg_pshort**, **pkg_glong**, and **pkg_plong**), it therefore becomes possible to easily read and write messages in a portable, machine independent format. This capability is the cornerstone of the successful growth of the BRL CAD Package software into networked environments.

3.5.10. Comparison with Sun XDR

This document has described a standard for the encoding of data, without making any attempt to provide a mechanism for describing the data, either in the program source code, or in the transmitted external representation. The format described in this document specifically assumes that the external representation does not have any alignment restrictions, so that strings and binary data may be mixed freely.

The Sun Microsystems XDR standard [SUN87] is very similar to the encoding proposed here, with a few significant differences. It is worthwhile noting that the BRL encoding was selected in 1984, and developed independently of the work at Sun. It is therefore interesting that, with the exception of alignment issues and the representation of strings, the same external format was selected. This is almost certainly due to the exhortations of Cohen [COHE80]. Therefore, with some careful coding it is possible for XDR software and BRL CAD Package software to interoperate.

XDR assumes that all quantities are aligned on four byte boundaries, for the greatest efficiency of processing on 32-bit word-addressable computers. XDR also prefixes all variable length fields with a 32-bit length indicator, and then pads the variable field. These choices consume some additional network bandwidth, but in most cases the difference in bandwidth requirements is small (for cases seen to date, XDR generates between 0% and 20% more bytes in the external representation).

Sun XDR goes a step further than just defining a standard representation for data exchange; XDR also provides a syntax for describing the external data structure. This description is preprocessed to produce compilable code that automaticly converts data between the internal and external formats, using the description of the data to be exchanged. This is a very nice feature indeed, and will certainly make the exchange of complex data structures much more convenient than the explicit conversion strategy. The XDR preprocessing step presumes a much closer coupling to the compiler environment than the BRL CAD approach, which may result in some additional (one-time) portability efforts to support each new type of system. In summary, it seems that either interface can be readily used, with the BRL CAD interface being easier to use for simpler data structures, with a few less restrictions, while the Sun XDR is far better at making the handling of complex data structures easy.

4. PIPELINE ORIENTED DISTRIBUTED COMPUTATION

Carefully dividing application software into appropriate tools provides significant rewards for software developers and maintainers, in terms of decreased program complexity, decreased incidence of bugs, and increased maintainability [KERN76]. For the user, having the ability to combine a set of software tools in arbitrary ways provides the potential for performing functions never imagined by the original software designers.

Traditional operating systems require that each command in a multi-command sequence be run sequentially, with the output from each command stored in a temporary file, for input into later steps. In the UNIX system, this style of operation can be improved upon, if the data passing between pairs of commands was originally stored in a single temporary file. In this case, pairs of commands can be connected together using UNIX pipes. In so doing, not only has the requirement for managing the temporary files (and their attendant storage) been eliminated, but a degree of parallelism has been introduced as well. As an example, this three command sequence:

```
step1  < input  > tmp1
step2  < tmp1  > tmp2
step3  < tmp2  > output
rm tmp1 tmp2
```

can be reduced to this pipeline:

```
step1 < input | step2 | step3 > output
```

The standard UNIX shell notation is used here, with **<** meaning “read from” (redirect standard input), **>** meaning “write to” (redirect standard output), and **|** meaning “pipe standard output from command on the left into standard input of command on the right”.

Systems with multiple processors have existed since the 1960s. Experimental UNIX systems with multiple processors existed in the late 1970s [MUUS79] The Dual-VAX work at Purdue University [GOBL81] popularized the use of multiple processors with the UNIX system, and paved the way for widespread commercial support. Multiprocessor UNIX systems are now common, with most major vendors supporting at least one model with multiple processors. Vendors like Alliant and Sequent have based their entire product lines on parallel processing. Utilizing the UNIX pipeline constructs permits the convenient and transparent exploitation of local multiprocessor resources with no reprogramming.

4.1. Tool-Oriented Image Processing

A large number of simple tools for manipulating images and framebuffers are provided in the CAD package. They have been written in the traditional UNIX Software Tools fashion: each performs a single basic function, and is intended to be connected together with other tools to achieve an overall goal. A fair amount of effort has gone into making a standard interface to the tools. All tools provide a usage message if executed with no arguments, and a common collection of flags is defined for all of the tools. If a tool expects to see binary data, and discovers that the binary data is routed to a user’s terminal, the tool aborts, rather than splattering a binary stream on an unsuspecting terminal.

The use of software tools for computer graphics is not new. Recent systems advocating this tools based approach include those of Duff [DUFF85] and Peterson [PETE86]. The BRL CAD Package has proven to be extremely flexible as as result of this approach. Generally, a new tool is added whenever the existing ones are found to be inadequate. Success can be claimed if users can easily achieve day-to-day tasks without having to write any new programs.

4.2. File Formats

The effective construction of software tools depends on having agreed upon standards for the format of the data that is passed between the various tools. It is important to note that UNIX itself does not impose any form on data files; there are no end of record markers or other operating system generated information. Therefore, the interpretation of the bytes within a file is implicit, and left entirely up to the programs which are used to read the file. The tools that are provided with the UNIX system are primarily intended for dealing with data stored in ASCII text files. These tools are quite powerful in their problem domain, but representing graphics constructs in a printable ASCII form typically would represent an unacceptable performance penalty. Therefore, it is necessary to define a variety of file formats that are more appropriate to the graphical tasks at hand. For efficiency reasons, binary formats are generally to be preferred, both in terms of being a compact representation, and also requiring the least processing time to handle by virtue of needing little or no conversion before use.

Significantly more stringent requirements are placed on the specification of file formats if files containing binary data are to be portable between different types of machines without need for explicit data conversion.

4.2.1. Storing Images: pix(5) and bw(5) files

By far the most common image formats are either eight bit per pixel black and white **bw**(5) and 24-bit per pixel color **pix**(5) formats. The files have the simplest possible format, in order to enable the rapid development of new tools. As a result of pixel arrays being stored tightly packed in memory in the well defined order of RGB, the RGBpixel definition is a useful format for storing pixels in files. RGBpixels are tightly packed arrays with three **unsigned char** elements. The values in each byte are viewed as intensities from 0 (off), through 255 (full on). Therefore, **pix**(5) format picture files are defined as being a sequence of RGBpixels. The ordering of the pixels in the file is first quadrant Cartesian, and corresponds to the pixel writing order of **libfb**. The first pixel has coordinates (0,0) and is the lower left corner of the image. The second pixel has coordinates (1,0), and subsequent pixels are stored left-to-right. The first pixel after the end of the first scanline has coordinates (0,1). Subsequent scanlines are stored bottom-to-top. This can be more simply thought of as a byte stream composed of repeated RGB triples, eg, RGBRGBRGB. Because **pix** files are defined as a byte stream, they are inherently portable to all machines with eight bit bytes.

A close relative of the **pix**(5) format file is the **bw**(5) format file. This is used for storing eight bit deep monochrome (‘‘black and white’’) images. **bw** files are stored in the same first quadrant layout as **pix** files, with the exception that each pixel is only one byte wide instead of being three bytes wide.

The **pix**(5) and **bw**(5) file formats have no header. The use of a simple headerless image format is the only fundamental design choice that has continued to be debated. The primary advantage of the headerless format is the ease of connecting tools together with pipes, and the ease of creating new tools without having to use I/O routines specific to the reading and writing of images. Each program is simply handed data, and the interpretation of the data is implicit in the definition of the tool used and the active options. This also allows the whole wealth of UNIX tools to be applied to images as well, as there is no special-purpose header that has to be skipped. More importantly, tools don’t have to know how to do the ‘‘right thing’’ with the header information, in the presence of seemingly contradictory information from either the definition of the tool or from the command line options. For example, processing a **pix** image with a **bw** tool like **bwmod** is often done. If files had headers, tools would almost certainly balk at being feed input with the wrong pixel depth, which in this example would be very frustrating, yet having tools infer how to do the ‘‘right thing’’ could also be extremely complicated if the header contains very much information, such as with the NASA FITS image format.

Having ‘‘raw’’ headerless data has its price however. It is difficult to tell whether a given image is color or not, what its dimensions are, etc. Consistent file naming conventions (using ‘‘.bw’’ or ‘‘.pix’’ as filename suffixes) address the first issue; doing most work in ‘‘standard sizes’’ of 512x512 or 1024x1024 pixels helps alleviate the second issue. In general, only the scanline length needs to be known, as the number of scanlines can then be found by dividing the file size by the scanline length. The algorithms of many tools simply run until all of the data is gone, and some don’t even care about scanline lengths at all, so the inconvenience of having headerless files tends to be minor. Using a good Shell with history and command-line screen editing features, or writing sets of simple goal-specific Shell scripts tends to minimize the aggravation of having to repetitively specify unusual image sizes on every command.

4.2.2. libplot3 Files

The original UNIX **plot**(5) format file was machine independent, although with the unfortunate choice of Little-Endian (VAX) byte ordering for the 16-bit integers. The extensions to the file format to provide all of the additional features (3-D, floating point coordinates, color) described above were all done in such a way to preserve the machine independent property of UNIX-plot files. In this way, not only is it possible to view plot files on displays attached to local computers, but it is also very convenient to produce images on remote displays.

4.2.3. Image Compression: The Utah RLE Format

Another format for storing images is the University of Utah Run Length Encoded **rle** images, which typically have a file suffix of ‘‘.rle’’. This is the standard image format of the Utah Raster Toolkit [PETE86]. Because this format typically requires less disk storage than the equivalent **pix** or **bw** file, and the compression operation takes some processor time, this format is typically used for long-term image storage, and image exchange with other institutions that use the Utah Raster Toolkit. Institutions that have neither the Utah nor BRL image processing tools often have an easier time importing images in **pix** format, because it is so simple.

4.2.4. Model Databases

Model databases are normally stored in a machine-specific binary form with a typical filename extension of ‘‘.g’’, with portable ASCII versions of those databases having a filename extension of ‘‘.asc’’. It is beyond the scope of this paper to describe the format in detail, other than to note that **mged** and **librt** know how to read the model databases, and that programs that wish to create geometry using procedural methods may do so with the services of the **libwdb** library for writing databases.

4.3. Framebuffer Tools

Most image manipulation and processing is performed either on data streams, or on disk files. This was done in order to separate the operations for handling a display device from the generic operations of image handling. A common beginning to a processing pipeline is a **fb-pix** command to get an image from a framebuffer, just as a common end of processing pipeline is a **pix-fb** command to display the final result. Several useful operations can be performed with direct interaction with a framebuffer, so some device independent tools are provided, including tools to allow changing colormaps, panning and zooming through an image, moving a pointer, adding labeling, etc. Where tools require the user to move a cursor or the image and cursor support is not available, both **emacs**-style and **vi**-style keyboard motion commands are accepted by all programs.

Selected Framebuffer Tools	
fb-pix	framebuffer to color image
fb-bw	framebuffer to black and white
fb-cmap	read a framebuffer colormap
fbcmap	can load several "standard" colormaps
fbclear	clear to an optional RGB color
fbgamma	load or apply gamma correcting colormaps
fbzoom	general zoom and pan routine
fbpoint	select pixel coordinates
fblabel	put a label on an image
fbcolor	a color selecting tool
fbscanplot	scanline RGB intensity plotter
fbanim	a "postage-stamp" animator
fbcmrot	a colormap rotator
fbed	a framebuffer image editor

4.4. Image Manipulation

A substantial collection of tools for image manipulation are provided as part of the package. These can generate statistics, histograms, extract parts of an image, rotate, scale, and filter them, etc. Space does not permit a discussion of all of the type of manipulation supported, but some of the more interesting tools are listed in the table.

Selected Image Tools	
pixstat	statistics - min, max, mean, etc.
pixhist	histogram
pixhist3d	RGB color space cube histogram
pixfilter	apply selected 3x3 filters
pixrect	extract a rectangle
pixrot	rotate, reverse, or invert
pixscale	scale up or down
pixdiff	compare two images
pixmerge	merge two/three images
pixtile	mosaic images together
gencolor	source a byte pattern
bwmod	apply expressions to each byte

4.5. Format Conversion

The **N**2** problem of format conversion between all the different ‘‘external’’ file formats is simplified by providing tools to convert all external file formats into the simple **pix** and **bw** formats. A selection of these conversion tools is listed in the table below. In all of the tables, the tool for the reverse conversion is omitted, e.g. in addition to the **rle-pix** tool, there is also a **pix-rle** for converting color images into **rle** format. Also, only the color (**pix**) version of a tool has been listed, even though most have black and white (**bw**) equivalents.

Most of the tools listed have a wide variety of options consistent with their basic function. For example, the tool to convert a color image to a black and white image (**pix-bw**), allows a selection of color blending values to be selected: equal weighting, NTSC weighting, or ‘‘typical CRT’’ weighting. It also allows arbitrary weights to be given for selecting or mixing of the color planes in any way desired.

Selected Format Conversion Tools	
g2asc	model database to portable ascii form
bw-pix	black and white to color image
bw3-pix	three black and whites to color RGB
rle-pix	Utah’s RLE format to color image
ap-pix	Applicon Ink-Jet to color image
sun-pix	Sun bitmap to color or black and white
mac-pix	MacIntosh MacPaint bitmaps to color

4.6. User Interface

Using software tools effectively comes with experience. The BRL CAD Package has tried to ease the difficulty of learning a new set of tools by using a common set of flags and common tool naming conventions throughout the package. The ‘‘user interface’’ is ultimately the Unix shell, and its conventions for establishing pipes, passing arguments to programs, etc. A shell with history recall and screen-oriented command editing, such as the **tcsh**, is a major convenience when constructing complicated command pipelines.

Constructing very complex interconnections between processing tools from the command line is sometimes difficult. One limitation is the single input, single output notion of a Unix pipe. Image manipulation often calls for three or more channels of data, as with **pix-bw3**. The most common solution to this problem is the use of intermediate files. Other approaches include extensions to the **tee** program, or a special tool such as **chan** [MOOR85] which demultiplexes a stream, feeds each channel to a different program, and remultiplexes the results.

4.7. Pipeline Processing

Systems which facilitate the coupling of dataflow oriented tools allow complex custom applications to be put together without writing any code. Consider this simple image processing example:

```
pixinterp2x -s512  < image.pix | \
pixfilter -s1024 -lo | \
pixmerge -n 63/0/127 -- - background.pix | \
pixrot -r -i 1024 1024 | \
pix-fb -h
```

which roughly says: take the file 512x512 color picture in *image.pix* and perform bi-linear interpolation to increase the image size to 1024x1024, then low-pass filter the large image with a 3x3 filter kernel, composite the filtered image with an existing color picture in *background.pix* with background replacement done on all pixels with RGB value 63/0/127, then rotate the image 180 degrees and display it on the current frame buffer.

Note how no intermediate images are stored in disk files throughout the whole procedure. For this example this may not be significant because the image at most stages required only 3 Mbytes of storage, but this becomes more important when manipulating 400Mbyte image data, which is processed in exactly the same way.

It is also worth noting that for larger images, this type of image processing can take a significant amount of processing time. While operations like this could be performed in a batch mode, there is a significant advantage to being able to observe the progress of the computation. As the results arrive on the display, the opportunity exists to abort the whole process if something is going wrong. In addition to the savings of significant amounts of computer time, there is also the potential to save ‘‘people time’’ as well, by increasing the number of cases that can be attempted within a single day, and thus increasing the speed of the project (assuming that the computer processing requirements are part of the critical path of the project).

This type of processing interactive computing works most effectively in an environment where each graphics user has their own multi-window workstation (not necessarily a graphics workstation) so that the user can switch to another window and continue working while continuing to monitor the progress of the graphics output. For example, a group of ten people, each with a dedicated inexpensive monochrome workstation, might share the services of two or three color workstations, using the network capabilities described below.

4.8. General Network Computing

Using UNIX interactively on a Cray is pretty heady stuff. Simply being able to open a "Cray X-MP" window or a "Cray-2" window on a workstation, and having the same environment (Shells, screen editors, compilers, source code tools, TCP networking, etc.), as on all the rest of the the network machines (Suns, SGIs, VAXen, Goulds, Alliants, etc) can be worth a lot. However, being able to **rsh** an image processing command over to a Cray without having to make special arrangements to put the files over on the Cray first, or having to submit a batch job, allows the power of SuperComputing to be harnessed without having to pay a stiff premium in inconvenience. For people acquainted with the Berkeley UNIX networking capabilities, the capabilities described in this section may seem trivial, and almost not worthy of mention. People not acquainted with the ease and power of the Berkeley **rsh** remote shell command may be surprised at the hidden synergistic power that appears when this capability is combined with a good collection of tools.

While logged in on the console of an SGI workstation, to dynamically produce some plot data, and then locally view the resulting **plot** file, this pipeline would be needed:

```
cruncher | pl-sgi
```

In this case, ‘‘cruncher’’ is assumed to produce a **plot** file on it’s standard output (*stdout*). The **pl-sgi** program reads a **plot** file on standard input (*stdin*) and produces a color wireframe display in a new window. However, if the computation speed of the workstation was inadequate, a slight variation could be used:

```
rsh Cray.arpa cruncher | pl-sgi
```

Simply by directing remote execution (via the **rsh** command) and naming the remote machine to perform the computation (in this case ‘‘Cray.arpa’’), the power of another machine can be brought to bear on the task. The **rsh** command [LEFF83] provides a number of significant features: (a) it connects to the specified machine, and validates access permission, (b) it passes the specified command to a Shell on the specified machine for parsing and execution, (c) it copies its standard input to the remote command, (d) the standard output of the remote command is copied to **rsh**s standard output, (e) the standard error output of the remote command is returned separately, and is copied to **rsh**s standard error output, (f) local interrupt, quit, and terminate signals are propagated to the remote command, (g) **rsh** exits when the remote command does, returning the remote exit status. Therefore, performing an operation on a remote machine with **rsh** produces an effect that is indistinguishable from having performed the same operation locally.

Another variation of this example might be useful if, instead of being logged in on the graphics workstation, you were logged in directly on the Cray, and after having finished some code development you wished to see an image. In this case, the command would be:

```
cruncher | rsh Vax.arpa pl-fb
```

This would send the **plot** file to the machine ‘‘Vax.arpa’’ and cause it to be displayed on a framebuffer. To generate a videotape to display the effect of varying a parameter in a simulation running on the Cray, with the display and videotape capability on the VAX, consider the following modest Shell script (which may be typed at the keyboard or run from a file):

```
for parm in ‘loop 1 100 2‘
do
    cruncher $parm | rsh Vax.arpa "pl-fb; vas4 record 1"
done
```

This simple script runs the **loop** tool to generate all the
integers between 1 and 100 inclusive, with an increment of 2,
which are assigned to the shell variable *parm*.
For each value of *parm*, the **cruncher** program is
run with *parm* as a formal argument.
The **plot** format output is sent to the Vax,
displayed on the framebuffer, and then the **vas4**
directs the Lyon-Lamb video animation controller to record
the image onto one frame of videotape.

Now, consider this variation on the earlier image processing example:

```
pixinterp2x -s512  < image.pix | \
rsh Cray.arpa "pixfilter -s1024 -lo" | \
rsh Alliant.arpa "pixmerge -n 63/0/127 -- - background.pix | \
rsh Vax.arpa "pixrot -r -i 1024 1024 | pix-fb -h"
```

which roughly says: grab an image on my local machine, perform bi-linear interpolation locally, then send it to the Cray for low-pass filtering, then send it to the Alliant for compositing with a background image, then send it to a trusty VAX to (a) rotate the image 180 degrees and (b) display it on the frame buffer.

Having learned about **rsh**, the possibilities of such combinations of different tools running on different machines are staggering. With the proper infrastructure of computers, operating systems, display hardware, network software, and image processing tools all connected together in compatible ways, the tremendous potential of distributed computing can be easily harnessed, without users having to write any programs!

5. SHARED-MEMORY PARALLEL PROCESSING

In the preceding section, we have seen how different processors can be harnessed to achieve a single goal. The discussion so far has focused on using multiple processors (a) within a single, multi-CPU system, through the use of UNIX pipes, and (b) by distributing different tools in a pipeline to different machines on the network. This section extends the investigation into parallel processing one level further, to harness the power of multiple processor machines to make a single application run faster. For the purposes of this discussion, the application to be parallelized is a ray-tracing program, but the techniques developed here are quite general.

5.1. The Need for Speed

Images created using ray-tracing have a reputation for consuming large quantities of computer time. For complex models, 10 to 20 hours of processor time to render a single frame on a DEC VAX-11/780 class machine is not uncommon. Using the ray-tracing paradigm for engineering analysis [MUUS87a] often requires many times more processing than rendering a view of the model. Examples of such engineering analyses include the predictive calculation of radar cross-sections, heat flow, and bi-static laser reflectivity. For models of real-world geometry, running these analyses approaches the limits of practical execution times, even with modern SuperComputers. There are three main strategies that are being employed to attempt to decrease the amount of elapsed time it takes to ray-trace a particular scene:

1) Advances in algorithms for ray-tracing. Newer techniques in partitioning space [KAPL85] and in taking advantage of ray-to-ray coherence [ARVO87] promise to continue to yield algorithms that do fewer and fewer ray/object intersections which do not contribute to the final results. Significant work remains to be done in this area, and an order of magnitude performance gain remains to be realized. However, there is a limit to the gains that can be made in this area.

2) Acquiring faster processors. A trivial method for decreasing the elapsed time to run a program is to purchase a faster computer. However, even the fastest general-purpose computers such as the Cray X-MP and Cray-2 do not execute fast enough to permit practical analysis of all real-world models in appropriate detail. Furthermore, the speed of light provides an upper bound on the fastest computer that can be built out of modern integrated circuits; this is already a significant factor in the Cray X-MP and Cray-2 processors, which operate with 8.5 ns and 4.3 ns clock periods respectively.

3) Using multiple processors to solve a single problem. By engaging the resources of multiple processors to work on a single problem, the speed-of-light limit can be circumvented. However, the price is that explicit attention must be paid to the distribution of data to the various processors, synchronization of the computations, and collection of the results.

Parallel processing is still a relatively young art, and presently there is only limited support available for the automatic parallelization of existing code, with newer vendors like Alliant leading the crowd. For now, there are few general techniques for taking programs intended for serial operation on a single processor, and automatically adapting them for operation on multiple processors [OHR86]. The **Worm** program developed at Xerox PARC [SHOC82] is one of the earliest known network image-rendering applications. More recently at Xerox PARC, Frank Crow has attempted to distribute the rendering of a single image across multiple processors [CROW86], but discovered that communication overhead and synchronization problems limited parallelism to about 30% of the available processing power. A good summary of work to date has been collected by Peterson [PETE87].

Ray-tracing analysis of a model has the very nice property that the computations for each ray/model intersection are entirely independent of other ray/model intersection calculations. Therefore, it is easy to see how the calculations for each ray can be performed by separate, independent processors. The underlying assumption is that each processor has read-only access to the entire model database. While it would be possible to partition the ray-tracing algorithm in such a way as to require only a portion of the model database being resident in each processor, this would significantly increase the complexity of the implementation as well as the amount of synchronization and control traffic needed. Such a partitioning has therefore not yet been seriously attempted.

It is the purpose of the research reported in the rest of this paper to explore the performance limits of parallel operation of ray-tracing algorithms where available processor memory is not a limitation. While it is not expected that this research will result in a general purpose technique for distributing arbitrary programs across multiple processors, the issues of the control and distribution of work and providing reliable results in a potentially unreliable system are quite general. The techniques used here are likely to be applicable to a large set of other applications.

5.2. Raytracing Background

The origins of modern ray-tracing come from work at MAGI under contract to BRL, initiated in the early 1960s. The initial results were reported by MAGI [MAGI67] in 1967. Extensions to the early developments were undertaken by a DoD Joint Technical Coordinating Group effort, resulting in publications in 1970 [JTCG70] and 1971 [JTCG71]. A detailed presentation of the fundamental analysis and implementation of the ray-tracing algorithm can be found in these two documents. Also see [APPE68]

More recently, interest in ray-tracing developed in the academic community, with Kay's [KAY79] thesis in 1979 being a notable early work. One of the central papers in the ray-tracing literature is the work of Whitted [WHIT80]. Model sampling techniques can be improved to provide substantially more realistic images by using the "Distributed Ray Tracing" strategy [COOK84]. For an excellent, concise discussion of ray-tracing, consult pages 363-381 of Rogers [ROGE85].

There are several implementation strategies for interrogating the model by computing ray/geometry intersections. The traditional approach has been batch-oriented, with the user defining a set of "viewing angles", turning loose a big batch job to compute all the ray intersections, and then post-processing all the ray data into some meaningful form. However, the major drawback of this approach is that the application has no dynamic control over ray paths, making another batch run necessary for each level of reflection, etc.

In order to be successful, applications need: (1) dynamic control of ray paths, to naturally implement reflection, refraction, and fragmentation into multiple subsidiary rays, and (2) the ability to fire rays in arbitrary directions from arbitrary points. Nearly all non-batch ray-tracing implementations have a specific closely coupled application (typically a model of illumination), which allows efficient and effective control of the ray paths. However, the most flexible approach is to implement the ray-tracing capability as a general-purpose library, to make the functionality available to any application as needed, and this is the approach taken in the BRL CAD Package [MUUS87c]. The ray-tracing library is called **librt**, while the ray-tracing application of interest here (an optical spectrum lighting model) is called **rt**.

5.3. The Structure of librt

In order to give all applications dynamic control over the ray paths, and to allow the rays to be fired in arbitrary directions from arbitrary points, BRL has implemented its third generation ray-tracing capability as a set of library routines. **librt** exists to allow application programs to intersect rays with model geometry. There are four parts to the interface: three preparation routines and the actual ray-tracing routine. The first routine which must be called is rt_dirbuild(), which opens the database file, and builds the in-core database table of contents. The second routine to be called is rt_gettree(), which adds a database sub-tree to the active model space. rt_gettree() can be called multiple times to load different parts of the database into the active model space. The third routine is rt_prep(), which computes the space partitioning data structures and does other initialization chores. Calling this routine is optional, as it is called by rt_shootray() if needed. rt_prep() is provided as a separate routine to allow independent timing of the preparation and ray-tracing phases of applications.

To compute the intersection of a ray with the geometry in the active model space, the application must call rt_shootray() once for each ray. Ray-path selection for perspective, reflection, refraction, etc, is entirely determined by the application program. The only parameter to the rt_shootray() is a **librt** "application" structure, which contains five major elements: the vector a_ray.r_pt which is the starting point of the ray to be fired, the vector a_ray.r_dir which is the unit-length direction vector of the ray, the pointer *a_hit() which is the address of an application-provided routine to call when the ray intersects the model geometry, the pointer *a_miss() which is the address of an application-provided routine to call when the ray does not hit any geometry, the flag a_onehit which is set non-zero to stop ray-tracing as soon as the ray has intersected at least one piece of geometry (useful for lighting models), plus various locations for each application to store state (recursion level, colors, etc). Note that the integer returned from the application-provided a_hit()/a_miss() routine is the formal return of the function rt_shootray(). The rt_shootray() function is prepared for full recursion so that the a_hit()/a_miss() routines can themselves fire additional rays by calling rt_shootray() recursively before deciding their own return value.

In addition, the function rt_shootray() is serially and concurrently reentrant, using only registers, local variables allocated on the stack, and dynamic memory allocated with rt_malloc(). The rt_malloc()

function serializes calls to **malloc**(3). By having the ray-tracing library fully prepared to run in parallel with other instances of itself in the same address space, applications may take full advantage of parallel hardware capabilities, where such capabilities exist.

5.4. A Sample Ray-Tracing Program

A simple application program that fires one ray at a model and prints the result is included below, to demonstrate the simplicity of the interface to **librt**.

```
#include <brlcad/raytrace.h>
struct application ap;
main() {
    rt_dirbuild("model.g");
    rt_gettree("car");
    rt_prep();
    ap.a_point = [ 100, 0, 0 ];
    ap.a_dir = [ -1, 0, 0 ];
    ap.a_hit = &hit_geom;
    ap.a_miss = &miss_geom;
    ap.a_onehit = 1;
    rt_shootray( &ap );
}
hit_geom(app, part)
struct application *app;
struct partition *part;
{
    printf("Hit %s", part->pt_forw->pt_regionp->reg_name);
}
miss_geom(){
    printf("Missed");
}
```

5.5. Normal Operation: Serial Execution

When running the **rt** program on a serial processor, the code of interest is the top of the subroutine hierarchy. The function main() first calls get_args() to parse any command line options, then calls rt_dirbuild() to acquaint **librt** with the model database, and view_init() to initialize the application (in this case a lighting model, which may call mlib_init() to initialize the material-property library). Finally, rt_gettree() is called repeatedly to load the model treetops. For each frame to be produced, the viewing parameters are processed, and do_frame() is called.

Within do_frame(), per-frame initialization is handled by calling rt_prep(), mlib_setup(), grid_setup(), and view_2init(). Then, do_run() is called with the linear pixel indices of the start and end locations in the image; typically these values are zero and width*length-1, except for the ensemble computer case. In the non-parallel cases, the do_run() routine initializes the global variables cur_pixel and last_pixel, and calls worker(). At the end of the frame, view_end() is called to handle any final output, and print some statistics.

The worker() routine obtains the index of the next pixel that needs to be computed by incrementing cur_pixel, and calls rt_shootray() to interrogate the model geometry. view_pixel() is called to output the results for that pixel. worker() loops, computing one pixel at a time, until *cur_pixel > last_pixel*, after which it returns.

When rt_shootray() hits some geometry, it calls the a_hit() routine listed in the application structure to determine the final color of the pixel. In this case, colorview() is called. colorview() uses view_shade() to do the actual computation. Depending on the properties of the material hit and the stack of shaders that are being used, various material-specific renderers may be called, followed by a call to rr_render() if reflection or refraction is needed. Any of these routines may spawn multiple rays, and/or recurse on colorview().

5.6. Parallel Operation on Shared-Memory Machines

By capitalizing on the serial and concurrent reentrancy of the **librt** routines, it is very easy to take advantage of shared memory machines where it is possible to initiate multiple "streams of execution" or "threads" within the address space of a single process. In order to be able to ensure that global variables are only manipulated by one instruction stream at a time, all such shared modifications are enclosed in critical sections. For each type of processor, it is necessary to implement the routines RES_ACQUIRE() and RES_RELEASE() to provide system-wide semaphore operations. When a processor acquires a resource, and any other processors need that same resource, they will wait until it is released, at which time exactly one of the waiting processors will then acquire the resource.

In order to minimize contention between processors over the critical sections of code, all critical sections are kept as short as possible: typically only a few lines of code. Furthermore, there are different semaphores for each type of resource accessed in critical sections. *res_syscall* is used to interlock all UNIX system calls and some library routines, such as **write**(2), **malloc**(3), **printf**(3), etc. *res_worker* is used by the function worker() to serialize access to the variable cur_pixel, which contains the index of the next pixel to be computed. *res_results* is used by the function view_pixel to serialize access to the result buffer. This is necessary because few processors have hardware multi-processor interlocking on byte operations within the same word. *res_model* is used by the **libspl** spline library routines to serialize operations which cause the model to be further refined during the raytracing process, so that data structures remain consistent.

Application of the usual client-server model of computing would suggest that one stream of execution would be dedicated to dispatching the next task, while the rest of the streams of execution would be used for ray-tracing computations. However, in this case, the dispatching operation is trivial and a "self-dispatching" algorithm is used, with a critical section used to protect the shared variable cur_pixel. The real purpose of the function do_run() is to perform whatever machine-specific operation is required to initiate *npsw* streams of execution within the address space of the **rt** program, and then to have each stream call the function worker(), each with appropriate local stack space.

Each worker() function will loop until no more pixels remain, taking the next available pixel index. For each pass through the loop, RES_ACQUIRE(res_worker) is used to acquire the semaphore, after which the index of the next pixel to be computed, *cur_pixel*, is acquired and incremented, and before the semaphore is released, ie,

```
worker() {
	while(1)  {
		RES_ACQUIRE( &rt_g.res_worker );
		my_index = cur_pixel++;
		RES_RELEASE( &rt_g.res_worker );
		if( my_index > last_pixel )
			break;
		a.a_x = my_index%width;
		a.a_y = my_index/width;
		...compute ray parameters...
		rt_shootray( &a );
	}
}
```

On the Denelcor HEP H-1000 each word of memory has a full/empty tag bit in addition to 64 data bits. RES_ACQUIRE is implemented using the Daread() primitive, which uses the hardware capability to wait until the semaphore word is full, then read it, and mark it as empty. RES_RELEASE is implemented using the Daset() primitive, which marks the word as full. do_run() starts additional streams of execution using the Dcreate(worker) primitive, which creates another stream which immediately calls the worker() function.

On the Alliant FX/8, RES_ACQUIRE is implemented using the hardware instruction test-and-set (TAS) which tests a location for being zero. As an atomic operation, if the location is zero, it sets it non-zero and sets the condition codes appropriately. RES_ACQUIRE embeds this test-and-set instruction in a polling loop to wait for acquisition of the resource. RES_RELEASE just zeros the semaphore word.

Parallel execution is achieved by using the hardware capability to spread a loop across multiple processors, so a simple loop from 0 to 7 which calls worker() is executed in hardware concurrent mode. Each concurrent instance of worker() is given a separate stack area in the ‘‘cactus stack’’.

On the Cray X-MP and Cray-2, the Cray multi-tasking library is used. RES_ACQUIRE maps into LOCKON, and RES_RELEASE maps into LOCKOFF, while do_run() just calls TSKSTART(worker) to obtain extra workers.

5.7. Performance Measurements

An important part of the BRL CAD Package is a set of five benchmark model databases and associated viewing parameters, which permit the relative performance of different computers and configurations to be made using a significant production program as the basis of comparison. For the purposes of this paper, just the "Moss" database is used for comparison. Since this benchmark generates pixels the fastest, it will place the greatest demands on any parallel processing scheme. The benchmark image is computed at 512x512 resolution.

The relative performance figures for running **rt** in the parallel mode with Release 1.20 of the BRL CAD Package are presented below. The Alliant FX/8 machine was brl-vector.arpa, configured with 8 Computational Elements (CEs), 6 68012 Interactive Processors (IPs), 32 Mbytes of main memory, and was running Concentrix 2.0, a port of 4.2 BSD UNIX. The Cray X-MP/48 machine was brl-patton.arpa, serial number 213, with 4 processors, 8 Mwords of main memory, with a clock period of 8.5 ns, and UNICOS 2.0, a port of System V UNIX. Unfortunately, no comprehensive results are available for the Denelcor HEP, the only other parallel computer known to have run this code.

Parallel **rt** Speedup -vs- # of Processors				
# Processors	FX/8	(eff)	X-MP/48	(eff)
1	1.00	100%	1.00	100%
2	1.84	92.0%	1.99	99.5%
3	2.79	93.0%	2.96	98.7%
4	3.68	92.0%	3.86	96.5%
5	4.80	96.0%		
6	5.70	95.0%		
7	6.50	92.9%		
8	7.46	93.3%		

The multiple-processor performance of **rt** increases nearly linearly for shared memory machines with small collections of processors. The slight speedup of the Alliant when the fifth processor is added comes from the fact that the first four processors share one cache memory, while the second four share a second cache memory. To date, **rt** holds the record for the best achieved speedup for parallel processing on both the Cray X-MP/48 and the Alliant. Measurements on the HEP, before it was dismantled, indicated that near-linear improvements continued through 128 streams of execution. This performance is due to the fact that the critical sections are very small, typically just a few lines of code, and that they account for an insignificant portion of the computation time. When **rt** is run in parallel and the number of processors is increased, the limit to overall performance is determined by the total bandwidth of the shared memory, and by memory conflicts over popular regions of code and data.

6. DISTRIBUTED COMPUTATION ON LOOSELY-COUPLED ENSEMBLES OF PROCESSORS

The basic assumption of this design is that network bandwidth is modest, so that the number of bytes and packets of overhead should not exceed the number of bytes and packets of results. The natural implementation would be to provide a remote procedure call (RPC) interface to rt_shootray(), so that when additional subsidiary rays are needed, more processors could potentially be utilized. However, measurements of this approach on VAX, Gould, and Alliant computers indicates that the system-call and communications overhead is comparable to the processing time for one ray/model intersection calculation. This much overhead rules out the RPC-per-ray interface for practical implementations. On some tightly coupled ensemble computers, there might be little penalty for such an approach, but in general, some larger unit of work must be exchanged.

It was not the intention of the author to develop another protocol for remote file access, so the issue of distributing the model database to the **rtsrv** server machines is handled outside of the context of the **remrt** and **rtsrv** software. In decreasing order of preference, the methods for model database distribution that are currently used are Sun NFS, Berkeley **rdist**, Berkeley **rcp**, and ordinary DARPA **ftp**. Note that the binary databases need to be converted to a portable format before they are transmitted across the network, because **rtsrv** runs on a wide variety of processor types. Except for the model databases and the executable code of the **rtsrv** server process itself, no file storage is used on any of the server machines.

6.1. Distribution of Work

The approach used in **remrt** involves a single dispatcher process, which communicates with an arbitrary number of server processes. Work is assigned in groups of scanlines. As each server finishes a scanline, the results are sent back to the dispatcher, where they are stored. Completed scanlines are removed from the list of scanlines to be done and from the list of scanlines currently assigned to that server. Different servers may be working on entirely different frames. Before a server is assigned scanlines from a new frame, it is sent a new set of options and viewpoint information.

The underlying communications layer used is the package (PKG) protocol, provided by the **libpkg** library, so that all communications are known to be reliable, and communication disruptions are noticed. Whenever the dispatcher is notified by the **libpkg** routines that contact with a server has been lost, all unfinished scanlines assigned to that server will be requeued at the head of the ‘‘work to do’’ queue, so that it will be assigned to the very next available server, allowing tardy scanlines to be finished quickly.

6.2. Distribution Protocol

When a server process **rtsrv** is started, the host name of the machine running the dispatcher process is given as a command line argument. The server process can be started from a command in the dispatcher **remrt**, which uses **system**(3) to run the **rsh** program, or directly via some other mechanism. This avoids the need to register the **rtsrv** program as a system network daemon and transfers issues of access control, permissions, and accounting onto other, more appropriate tools. Initially, the **rtsrv** server initiates a PKG connection to the dispatcher process and then enters a loop reading commands from the dispatcher. Some commands generate no response at all, some generate one response message, and some generate multiple response messages. However, note that the server does not expect to receive any additional messages from the dispatcher until after it has finished processing a request, so that requests do not have to be buffered in the server. While this simplifies the code, it has some performance implications, which are discussed later.

In the first stage, the message received must be of type MSG_START, with string parameters specifying the pathname of the model database and the names of the desired treetops. If all goes well, the server responds with a MSG_START message, otherwise diagnostics are returned as string parameters to a MSG_PRINT message and the server exits.

In the second stage, the message received must be of type MSG_OPTIONS or MSG_MATRIX. MSG_OPTIONS specifies the image size and shape, hypersampling, stereo viewing, perspective -vs- ortho view, and control of randomization effects (the ‘‘benchmark’’ flag), using the familiar UNIX command line option format. MSG_MATRIX contains the 16 ASCII floating point numbers for the 4x4 homogeneous transformation matrix which represents the desired view.

In the third stage, the server waits for messages of type MSG_LINES, which specify the starting and ending scanline to be processed. As each scanline is completed, it is immediately sent back to the dispatcher process to minimize the amount of computation that could be lost in case of server failure or communications outage. Each scanline is returned in a message of type MSG_PIXELS. The first two bytes of that message contain the scanline number in network order 16-bit binary. Following that is the 3*width bytes of RGB data that represents the scanline. When all the scanlines specified in the MSG_LINES command are processed, the server again waits for another message, either another MSG_LINES command or a MSG_OPTIONS/MSG_MATRIX command to specify a new view.

At any time, a MSG_RESTART message can be received by the server, which indicates that it should close all it’s files and immediately re-**exec**(2) itself, either to prepare for processing an entirely new model, or as an error recovery aid. A MSG_LOGLVL message can be received at any time, to enable and disable the issuing of MSG_PRINT output. A MSG_END message suggests that the server should commit suicide, courteously.

6.3. Dispatching Algorithm

The dispatching (scheduling) algorithm revolves around two main lists, the first being a list of currently connected servers and the second being a list of frames still to be done. For each unfinished frame, a list of scanlines remaining to be done is maintained. For each server, a list of the currently assigned scanlines is kept. Whenever a server returns a scanline, it is removed from the list of scanlines assigned to that server, stored in the output image, and also in the optional attached framebuffer. (It can be quite entertaining to watch the scanlines racing up the screen, especially when using processors of significantly different speeds). If the arrival of this scanline completes a frame, then the frame is written to disk on the dispatcher machine, timing data is computed, and that frame is removed from the list of work to be done.

When a server finishes the last scanline of its assignment and more work remains to be done, the list of unfinished frames is searched and the next available increment of work is assigned. Work is assigned in blocks of consecutive scanlines, up to a per-server maximum assignment size. The block of scanlines is recorded as the server's new assignment and is removed from the list of work to be done.

6.4. Reliability Issues

If the **libpkg** communications layer looses contact with a server machine, or if **remrt** is commanded to drop a server, then the scanlines remaining in the assignment are requeued at the head of the list of scanlines remaining for that frame. They are placed at the head of the list so that the first available server will finish the tardy work, even if it had gone ahead to work on a subsequent frame.

Presently, adding and dropping server machines is a manual (or script driven) operation. It would be desirable to develop a separate machine-independent network mechanism that **remrt** could use to inquire about the current loading and availability of server machines, but this has not been done. This would permit periodic status requests to be made and automatic reacquisition of eligible server machines could be attempted. Peterson's Distrib [PETE87] System incorporates this as a built-in part of the distributed computing framework, but it seems that using an independent transaction-based facility such as Pistritto's Host Monitoring Protocol (HMP) facility [NATA84] would be a more general solution.

If the dispatcher fails, all frames that have not been completed are lost; on restart, execution resumes at the beginning of the first uncompleted frame. By carefully choosing a machine that has excellent reliability to run the dispatcher on, the issue of dispatcher failure can be largely avoided. However, typically no more than two frames will be lost, minimizing the impact. For frames that take extremely long times to compute, it would be reasonable extend the dispatcher to snapshot the work queues and partially assembled frames in a disk file, to permit operation to resume from the last "checkpoint".

6.5. Distributed remrt Performance

Ten identical Sun-3/50 systems were used to test the performance of **remrt**. All had 68881 floating point units and 4 Mbytes of memory, and all were in normal timesharing mode, unused except for running the tests and the slight overhead imposed by /etc/update, **rwhod**, etc. To provide a baseline performance figure for comparison, the benchmark image was computed in the normal way using **rt**, to avoid any overhead which might be introduced by **remrt**. The elapsed time to execute the ray-tracing portion of the benchmark was 2639 seconds; the preparation phase was not included, but amounted to only a few seconds.

remrt Speedup -vs- # of Processors						
	Ratios		Elapsed Seconds			
# CPUs	Theory	Sun-3/50	Theory	Sun-3/50	Total Speedup	Efficiency
1	1.0000	1.0072	2639.0	2658	0.993	99.3%
2	0.5000	0.5119	1319.5	1351	1.953	97.7%
3	0.3333	0.3357	879.6	886	2.979	99.3%
4	0.2500	0.2524	659.7	666	3.949	98.7%
5	0.2000	0.2027	527.8	535	4.916	98.3%
6	0.1666	0.1686	429.8	445	5.910	98.5%
7	0.1429	0.1470	377.0	388	6.778	96.8%
8	0.1250	0.1266	329.9	334	7.874	98.4%
9	0.1111	0.1133	293.2	299	8.796	97.7%
10	0.1000	0.1019	263.9	269	9.777	97.8%

The ‘‘speedup’’ figure of 0.993 for 1 CPU shows the loss of performance of 0.7% introduced by the overhead of the **remrt** to **rtsrv** communications, versus the non-distributed **rt** performance figure. The primary result of note is that the speedup of the **remrt** network distributed application is very close to the theoretical maximum speedup, with a total efficiency of 97.8% for the ten Sun case! The very slight loss of performance noticed (2.23%) is due mostly to ‘‘new assignment latency’’, discussed further below. Even so, it is worth noting that the speedup achieved by adding processors with **remrt** was even better than the performance achieved by adding processors in parallel mode with **rt**. This effect is due mostly to the lack of memory and semaphore contention between the **remrt** machines.

Unfortunately, time did not permit configuring and testing multiple Alliants running **rtsrv** in full parallel mode, although such operation is supported by **rtsrv**.

When **remrt** is actually being used for producing images, many different types of processors can be used together. The aggregate performance of all the available machines on a campus network is truly awesome, especially when a Cray or two is included! Even in this case, the network bandwidth required does not exceed the capacity of an Ethernet (yet). The bandwidth requirements are sufficiently small that it is practical to run many **rtsrv** processes distributed over the ARPANET/MILNET. On one such occasion in early 1986, 13 Gould PN9080 machines were used all over the east coast to finish some images for a publication deadline.

6.6. Performance Issues

The policy of making work assignments in terms of multiple adjacent scanlines reduces the processing requirements of the dispatcher and also improves the efficiency of the servers. As a server finishes a scanline, it can give the scanline to the local operating system to send to the dispatcher machine, while the server continues with the computation, allowing the transmission to be overlapped with more computation. When gateways and wide-area networks are involved (with their accompanying increase in latency and packet loss), this is an important consideration. In the current implementation, assignments are always blocks of three scanlines because there is no general way for the **rtsrv** process to know what kind of machine it is running on and how fast it is likely to go. Clearly, it would be worthwhile to assign larger blocks of scanlines to the faster processors so as to minimize idle time and control traffic overhead. Seemingly the best way to determine this would be to measure the rate of scanline completion and dynamically adjust the allocation size. This is not currently implemented.

By increasing the scanline block assignment size for the faster processors, the amount of time the server spends waiting for a new assignment (termed ‘‘new assignment latency’’) is diminished, but not eliminated. Because the current design assumes that the server will not receive another request until the previous request has been fully processed, no easy solution exists. Extending the server implementation to buffer at least one additional request would permit this limitation to be overcome, and the dispatcher would then have the option of sending a second assignment before the first one had completed, to always keep the server ‘‘pipeline’’ full. For the case of very large numbers of servers, this pipelining is important to keep delays in the dispatcher from affecting performance. In the case of very fast servers, pipelining is important in achieving maximum server utilization, by overcoming network and dispatcher delays.

To obtain an advantage from the pipeline effect of the multiple scanline work assignments, it is important that the network implementations in both the servers and the dispatcher have adequate buffering to hold an entire scanline (typically 3K bytes). For the dispatcher, it is a good idea to increase the default TCP receive space (and thus the receive window size) from 4K bytes to 16K bytes. For the server machines, it is a good idea to increase the default TCP transmit space from 4K bytes to 16K bytes. This can be accomplished by modifying the file /sys/netinet/tcp_usrreq.c to read:

```
int   tcp_sendspace = 1024*16;
int   tcp_recvspace = 1024*16;
```

or to make suitable modifications to the binary image of the kernel using *adb*(1):

```
adb -w -k /vmunix
tcp_sendspace?W 0x4000
tcp_recvspace?W 0x4000
```

The dispatcher process must maintain an active network connection to each of the server machines. In all systems there is some limit to the number of open files that a single process may use (symbol NOFILE); in 4.3 BSD UNIX, the limit is 64 open files. For the current implementation, this places an upper bound on the number of servers that can be used. As many campus networks have more than 64 machines available at night, it would be nice if this limit could be eased. One approach is to increase the limit on the dispatcher machine. Another approach is to implement a special ''relay server'' to act as a fan-in/fan-out mechanism, although the additional latency could get to be an issue. A third approach is to partition the problem at a higher level. For example, having the east campus do the beginning of a movie, and the west campus do the end would reduce the open file problem. Additionally, if gateways are involved, partitioning the problem may be kinder to the campus network.

6.7. Conclusions

Parallel computing is good.

This paper has shown how it is possible to implement good graphics interfaces within the confines of a single uniprocessor machine. With the adoption of a ''software tools'' approach when providing data handling capabilities, it was shown how to transparently take advantage of multi-processor machines, and thus realize a speed advantage. Furthermore, the careful selection of file formats permitted realizing a further speed advantage in the accomplishment of a single task by utilizing multiple systems located across a network.

Carying the theme of increased speed further, multi-processor sytems were examined as a vehicle for making single image generation tools operate faster. An important result was to note that when operation in a shared memory parallel environment was an initial design goal, the implementation of concurrently reentrant code did not significantly increase the complexity of the software. Having code with such properties allows direct utilization of nearly any shared memory multiprocessor with a minimum of system-specific support, namely the RES_ACQUIRE and RES_RELEASE semaphore operations, and some mechanism for starting multiple streams of execution within the same address space.

Finally, collections of processors connected only by conventional speed network links are considered as the final environment for making a single tool operate faster using multiprocessing. It was shown that network distributed computing need not be inefficient or difficult. The protocol and dispatching mechanism described in the preceding sections has been shown to be very effective at taking the computationally intensive task of generating ray-traced images and distributing it across multiple processors connected only by a communications network. There are a significant number of other application programs that could directly utilize the techniques and control software implemented in **remrt** to achieve network distributed operation. However, the development and operation of this type of program is still a research effort; the technology is not properly packaged for widespread, everyday use. Furthermore, it is clear that the techniques used in **remrt** are not sufficiently general to be applied to all scientific problems. In particular, problems where each ''cell'' has dependencies on some or all of the neighboring cells will require different techniques.

Massive proliferation of computers is a trend that is likely to continue through the 1980s into the 1990s and beyond. Developing software to utilize significant numbers of network connected processors is the coming challenge. This paper has presented a strategy that meets this challenge, and provides a simple, powerful, and efficient method for distributing a significant family of scientific analysis codes across multiple computers.

Acknowledgements

The author would like to thank Dr. Paul Deitz for providing his unflagging support and encouragement, Phil Dykstra, Paul Stay, Gary Moss, Chuck Kennedy, and Bob Reschly for the long hours as we designed and built this software, and Dr. Dave Rogers for once again persuading me to write it all down.

The following strings which have been included in this paper are known to enjoy protection as trademarks; the trademark ownership is acknowledged in the table below.

Trademark	Trademark Owner
Cray	Cray Research, Inc
Ethernet	Xerox Corporation
FX/8	Alliant Computer Systems Corporation
IBM 370	International Business Machines Corporation
Macintosh	Apple Computer, Inc
MacPaint	Apple Computer, Inc
NFS	Sun Microsystems, Inc
PowerNode	Gould, Inc
ProNet	Proteon, Inc
Sun Workstation	Sun Microsystems, Inc
SunView	Sun Microsystems, Inc
UNIX	AT&T Bell Laboratories
UNICOS	Cray Research, Inc
VAX	Digital Equipment Corporation
VaxStation	Digital Equipment Corporation
X Window System	Massachusetts Institute of Technology

References

APPE68.
A. Appel, "Some Techniques for Shading Machine Renderings of Solids," *AFIPS 1968 Spring Joint Computer Conference,* 32, pp. 37-45 (1968).

ARVO87.
J. Arvo and D. Kirk, "Fast Ray Tracing by Ray Classification," *Computer Graphics (Proceedings of Siggraph '87),* 21, 4 (July 1987).

CHAM86.
S. Chamberlain, M. Muuss, and J. Pistritto, *The BRL Fire Support Application for ADDCOMPE,* BRL Special Publication BRL-SP-53 (April 1986).

COHE80.
D. Cohen, *On Holy Wars and a Plea for Peace, IEN-137 (big-endian little-endian),* DDN Network Information Center, Menlo Park CA (April 1980).

CONR80.
Conrac, *Raster Graphics Handbook,* Van Nostrand Reinhold, New York (1980).

CONV88.
Convex, *Convex Vector C Compiler User's Guide,* Convex Computer Corporation, Doc 720-630-200, Richardson, TX (January 1988).

COOK84.
R. Cook and Porter, L. Carpenter, "Distributed Ray Tracing," *Computer Graphics (Proceedings of Siggraph '84),* 18, 3, pp. 137-145 (July 1984).

CRAY82.
Cray, *Cray-1 Computer Systems Mainframe Reference Manual,* Cray Research Inc, HR-0029, Mendota Heights, MN (November 1982).

CROW86.
F. C. Crow, *Experiences in Distributed Execution: A Report on Work in Progress,* Siggraph '86 Tutorial "Advanced Image Systhesis", Dallas, TX (August 18-22, 1986).

DCA83.
DCA, *MIL-STD-1777: Internet Protocol,* Defense Communications Agency, U. S. Government Printing Office (12 August 1983).

DEC76.
DEC, *Digital PDP-11/70 Processor Handbook,* Digital Equipment Corporation, Maynard, MA (1976).

DEC82.
DEC, *VAX Technical Summary,* Digital Equipment Corporation, Maynard, MA (1982).

DEIT88.
P. Deitz, W. Mermagen Jr, and P. Stay, "An Integrated Environment for Army, Navy, and Air Force Target Description Support," *Proceedings of the Tenth Annual Symposium on Survivability/Vulnerability* (April 1988).

DEIT83.
P. H. Deitz, *Solid Geometric Modeling - The Key to Improved Materiel Acquisition from Concept to Deployment,* Defense Computer Graphics 83, Washington DC (10-14 October 1983).

DEIT84b.
P. H. Deitz, *Predictive Signature Modeling via Solid Geometry at the BRL,* Sixth KRC Symposium on Ground Vehicle Signatures, Houghton MI (21-22 August 1984).

DEIT85.
P. H. Deitz, *The Future of Army Item-Level Modeling,* Army Operations Research Symposium XXIV, Ft. Lee VA (8-10 October 1985).

DUFF85.
Tom Duff, "Compositing 3-D Rendered Images," *Computer Graphics,* 19, 2, Proceedings of SIGGRAPH 85 (July, 1985).

FEIN85.
E. J. Feinler, O. J. Jacobsen, M. K. Stahl, and C. A. Ward, *DDN Protocol Handbook, NIC 50004,* DDN Network Information Center, Menlo Park CA (December 1985).

GOBL81.
G. H. Goble and M. H. Marsh, *A Dual Processor VAX 11/780,* Purdue University Technical Report TR-EE 81-31 (September 1981).

GWYN85.
D. A. Gwyn, *VLD/VMB UNIX Supplementary Manual for System V,* BRL Internal Publication (October 1985).

HALL83.
R. Hall, *A Methodology for Realistic Image Synthesis,* MS dissertation, Cornell University (August 1983).

IBM70.
IBM, *IBM System/370 Principles of Operation,* International Business Machines Corporation, GA22-7000-4, Poughkeepsie, NY (1970).

IBM82.
IBM, *The Economic Value of Rapid Response Time,* International Business Machines Corporation (1982).

IEEE85.
IEEE, *ANSI/IEEE Standard 754 for Binary Floating-Point Arithmetic,* Institute of Electrical and

Electronic Egnineers, Piscataway NJ (August 1985).

JTCG70.
JTCG, *MAGIC Computer Simulation, Vol. 1, User Manual,* Joint Technical Coordinating Group for Munitions Effectiveness, 61JTCG/ME-71-7-1 (July 1970).

JTCG71.
JTCG, *MAGIC Computer Simulation, Vol. 2, Analyst Manual,* Joint Technical Coordinating Group for Munitions Effectiveness, 61JTCG/ME-71-7-2-1 (May 1971).

KAHA86.
W. Kahan, Z-S. A. Liu, S. McDonald, K-C. Ng, and P. Tang, *Introduction to Mathematical Library Functions,* The 4.3 Berkeley Software Distribution, Berkeley CA (May 1986).

KAPL85.
M. R. Kaplan, *Space-Tracing, a Constant Time Ray-Tracer,* Siggraph '85 Tutorial ''State of the Art in Image Synthesis'', San Francisco CA (July 22-26, 1985).

KAY79.
D. S. Kay, *Transparency, Refraction, and Ray Tracing for Computer Synthesized Images,* Cornell Univ (Jan 1979).

KERN76.
B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, Reading, Massachusetts (1976).

KERN84.
Brian W. Kernighan and Rob Pike, *The UNIX Programming Environment,* Prentice-Hall, Englewood Cliffs, New Jersey (1984).

LEFF83.
S. J. Leffler, *Bug fixes and changes in 4.2BSD,* The 4.2 Berkeley Software Distribution, Berkeley CA (July 1983).

MAGI67.
MAGI, *A Geometric Description Technique Suitable for Computer Analysis of Both Nuclear and Conventional Vulnerability of Armored Military Vehicles,* MAGI Report 6701, AD847576 (August 1967).

MCCO87.
B. H. McCormick, T. A. DeFanti, and M. D. Brown, "Visualization in Scientific Computing," *Computer Graphics,* 21, 6, pp. c7-c8 (November 1987).

MOOR85.
R. F. Moore, *CARL Startup Kit,* Computer Audio Research Laboratory, University of California at San Diego (1985).

MUUS78.
M. Muuss, *The Terminal Independent Graphics Package: System Design and Operation,* Army Research Office Report (September 1978).

MUUS79.
M. Muuss, *The Implementation of a Multi-Processor UNIX System: Preliminary Design,* Army Research Office Report No. 1127 (October 1979).

MUUS87b.
M. Muuss, "Why Buy UNIX?," *New Zealand Computer Scene and Office Automation,* 7, pp. 74-78 (1987).

MUUS83.
M. J. Muuss, K. A. Applin, R. J. Suckling, G. S. Moss, E. P. Weaver, and C. Stanley, *GED: An Interactive Solid Modeling System for Vulnerability Analysis,* BRL Technical Report ARBRL-TR-02480, NTIS AD No. A-126-657 (March 1983).

MUUS87a.
M. J. Muuss, "Understanding the Preparation and Analysis of Solid Models" in *Techniques for Computer Graphics,* ed. D. F. Rogers, R. A. Earnshaw, Springer-Verlag (1987).

MUUS87c.
M. J. Muuss and P. Dykstra, K. Applin, G. Moss, E. Davisson, P. Stay, C. Kennedy, *Ballistic Research Laboratory CAD Package, Release 1.21,* BRL Internal Publication (June 1987).

MUUS87d.
M. J. Muuss, "RT and REMRT - Shared Memory Parallel and Network Distributed Ray-Tracing Programs," *USENIX: Proceeding of the Fourth Computer Graphics Workshop* (Oct 1987).

NATA84.
R. B. Natalie, M. J. Muuss, D. P. Kingston, C. M. Kennedy, and D. A. Gwyn, *The First BRL VAX UNIX Manual,* BRL Internal Publication (Fall 1984).

OHR86.
S. Ohr, "Minisupercomputers Mix Vector Speed, Scalar Flexibility," *Electronic Design,* 34, 5, pp. 107-114 (March 1986).

PETE86.
J. W. Peterson, R. G. Bogart, and S. W. Thomas, "The Utah Raster Toolkit," *USENIX: Proceedings of the Third Computer Graphics Workshop* (1986).

PETE87.
J. W. Peterson, *Distributed Computation for Computer Animation,* University of Utah Technical Report UUCS 87-014 (June 1987).

POST81.
J. Postel, "Internet Protocol Specification, RFC-791" in *DDN Protocol Handbook, Vol 2, NIC 50004,* DDN Network Information Center, Menlo Park CA (September 1981).

QUAR86.
J. S. Quarterman and J. C. Hoskins, "Notable Computer Networks," *Communications of the ACM,* 29, 10, pp. 932-971 (October 1986).

RICA87.
G. Ricart, "Research Univsersity Bandwidth Estimates" in *A Report to the Congress on Computer Networks to Support Research in the United States: A Study of Critical Problems and Future Options, Volume III, A Compendium of Supporting Technical Data,* Federal Coordinating Council on Science, Engineering, and Technology (June 1987).

RITC78a.
D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell System Technical Journal,* 57, 6, pp. 1905-1929 (1978).

RITC78b.
D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "The C Programming Language," *Bell System Technical Journal,* 57, 6, pp. 1991-2019 (1978).

ROGE85.
David F. Rogers, *Procedural Elements for Computer Graphics,* McGraw-Hill, New York (1985).

SHOC82.
J. F. Shoch and J. A. Hupp, "The Worm Programs -- Early Experience with a Distributed Computation," *Communications of the ACM,* 25, 3, p. 172 (March 1982).

SUN87.
Sun, *XDR: External Data Representation Standard, RFC-1014,* DDN Network Information Center, Menlo Park CA (June 1987).

TANE81.
A. S. Tanenbaum, *Computer Networks,* Prentice-Hall (1981).

TOOM82.
J. C. Toomay, *Radar Principles for the Non-Specialist,* Lifetime Learning Publications, London (1982).

WEAV80.
E. P. Weaver and M. J. Muuss, *Interactive Construction of COM-GEOM Targets,* BRL Spring

Technical Conference (1980).

WEAV82.
E. P. Weaver and P. H. Deitz, “Solid Modeling in Survivability/Vulnerability,” *Proceedings of the Second Joint Technical Coordinating Group for Aircraft Survivability Workshop on Survivability and Computer-Aided Design,* 1, USAF Museum, Wright-Patterson AFB, OH (18-20 May 1982).

WHIT80.
J. T. Whitted, “An Improved Illumination Model for Shaded Display,” *Communications of the ACM,* 23, 6, pp. 343-349 (June 1980).

XERO80.
Xerox, *The Ethernet, a Local Area Network: Data Link Layer and Physical Layer Specification,* X3T51/80-50 Xerox, Stamford CT (October 1980).