Casual Viewing of 3-D Data from a Remote System

By Jeanne Sparlin and Andrew Taylor

The Virtual Frame Buffer and the associated OpenGL enhancements allow programmers to create environments for the low-cost viewing of 3-D data by a casual user.

Graphics users sometimes need to view a 3-D object and even look at the object from different angles and viewpoints. For example, suppose the engineering manager for a company needs to approve a design while traveling on a business trip. The manager simply needs to view the design from different angles and different perspectives, but does not need to edit or animate the object. In this scenario, the PC or laptop does not have either graphics software or hardware required to edit or animate the object, and the communication connection is very slow.

In these types of scenarios that require casual viewing of graphics, the data being viewed is often stored at a remote location from the person who is viewing the data. A super-graphics workstation is not necessary because the requirement is intermittent and occasional compared to a routine or daily use.

This article discusses the Virtual Frame Buffer and Open GL enhancement now available in AIX® to address the issue of casual viewing of remote 3-D graphics—which has previously required a workstation and graphics adapter. AIX now offers several alternatives for casual viewing of 3-D graphics.

Architectures for Remote Graphics

Figure 1 shows a traditional architecture for remote graphics. The application, X Server, and 3-D rendering software reside on the local workstation; the data resides remotely on the server. Each person who views the data uses a system that supports an X Server, 3-D rendering software, and a graphics adapter.

The application retrieves the model to be viewed from a remote system. In addition, all supported UNIX® vendors and Wintel[1] platforms need separate binaries for the application. This environment generally does not support network computers because they are smaller, lightweight systems and cannot support the system requirements needed to display and manipulate the graphics objects.

One advantage of this architecture is that it enables good performance for manipulating the viewed object. Because the model and rendering software reside on the same machine as the graphics hardware, the display is very fast.

Some disadvantages, however, include the higher cost of workstations and software because they require separate binaries for each UNIX and Wintel system that is supported. Applications also tend to be expensive because of the cost associated with developing and testing on various supported platforms.

Jeanne Sparlin

[1] Wintel refers to the combination of the Windows™ operating system running on Intel® processors.

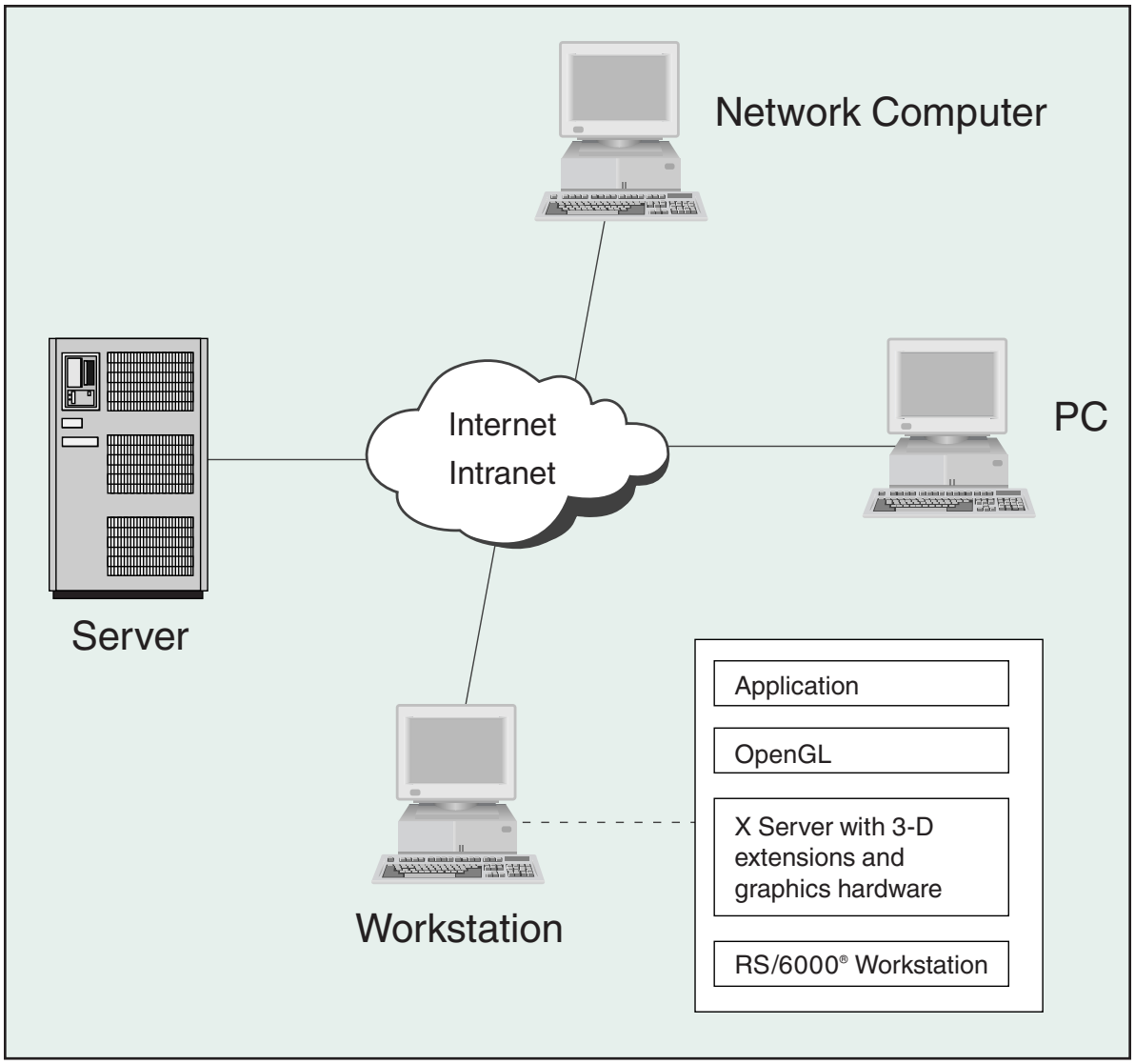

Figure 1. Traditional workstation environment

Figure 2 shows another architecture for remote graphics: remote application/local rendering software. The client application and data reside on the server system and the rendering software and graphics hardware reside on the client system.

The application sends the protocol to the X Server or 3-D extension to the X Server, for example, OpenGL™. The local workstation requires an X Server, the appropriate 3-D rendering software, and possibly graphics hardware. Most major UNIX vendors provide X Servers and appropriate 3-D extensions for their systems. X Server products for the PC are available for Wintel systems.

The X Server and 3-D rendering software are large processes, so they use valuable system resources. Network computers do not have the resources or software to operate in this environment.

One advantage of the remote application/local rendering architecture is that it requires only one application binary that executes on the server system. The development cost is lowered because the application is tested on only one system or hardware architecture.

Disadvantages of this architecture include lower performance because of the amount of data sent across the network each time the model is rerendered. In some cases, the

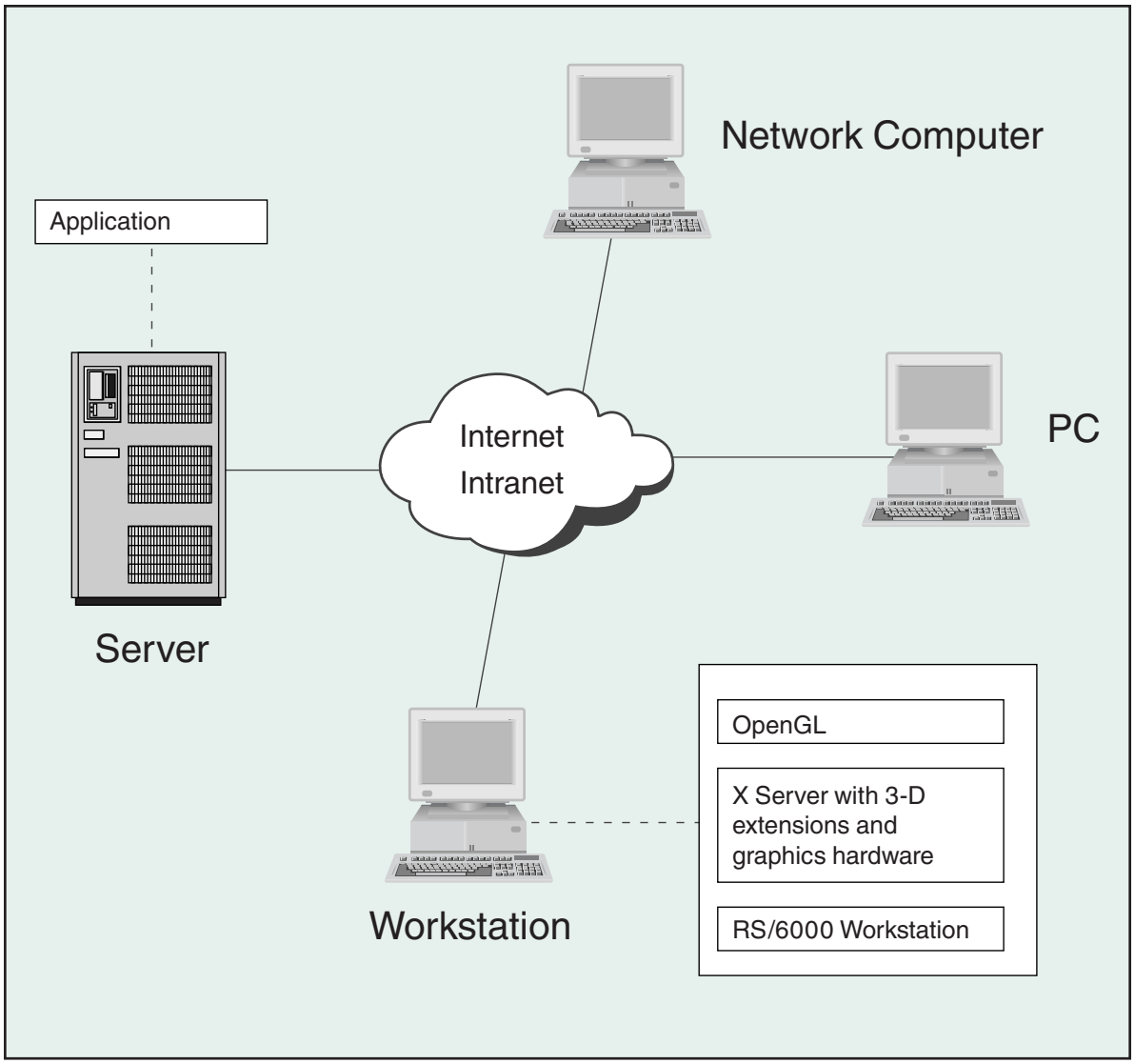

Figure 2. Remote application using local rendering software

inability to direct hardware access can also lower performance. All client systems need an X Server and 3-D graphics software.

Figure 3 demonstrates the Virtual Frame Buffer (VFB) environment. The application, X Server, and 3-D rendering software all reside on the server system. The application renders images on the server, then distributes images to viewing stations on a network, saves images to a database, or uses them in some other way.

Within this client/server or rendering server environment, the end user does not directly drive the application. Instead, the application receives commands from software that is driven by an end user on a low-end system. Although the server system (host machine) may have a graphics adapter, it will more likely contain a Virtual Frame Buffer (VFB) or a non-visible frame buffer.

The VFB and associated OpenGL enhancements are new IBM rendering technologies designed specifically to make 3-D rendering of server applications more efficient and scalable.[2] These two new rendering technologies are discussed in the next sections.

[2] OpenGL is currently supported with Virtual Frame Buffer, but graPHIGS™, PEX, and OpenGL 3.2 are not.

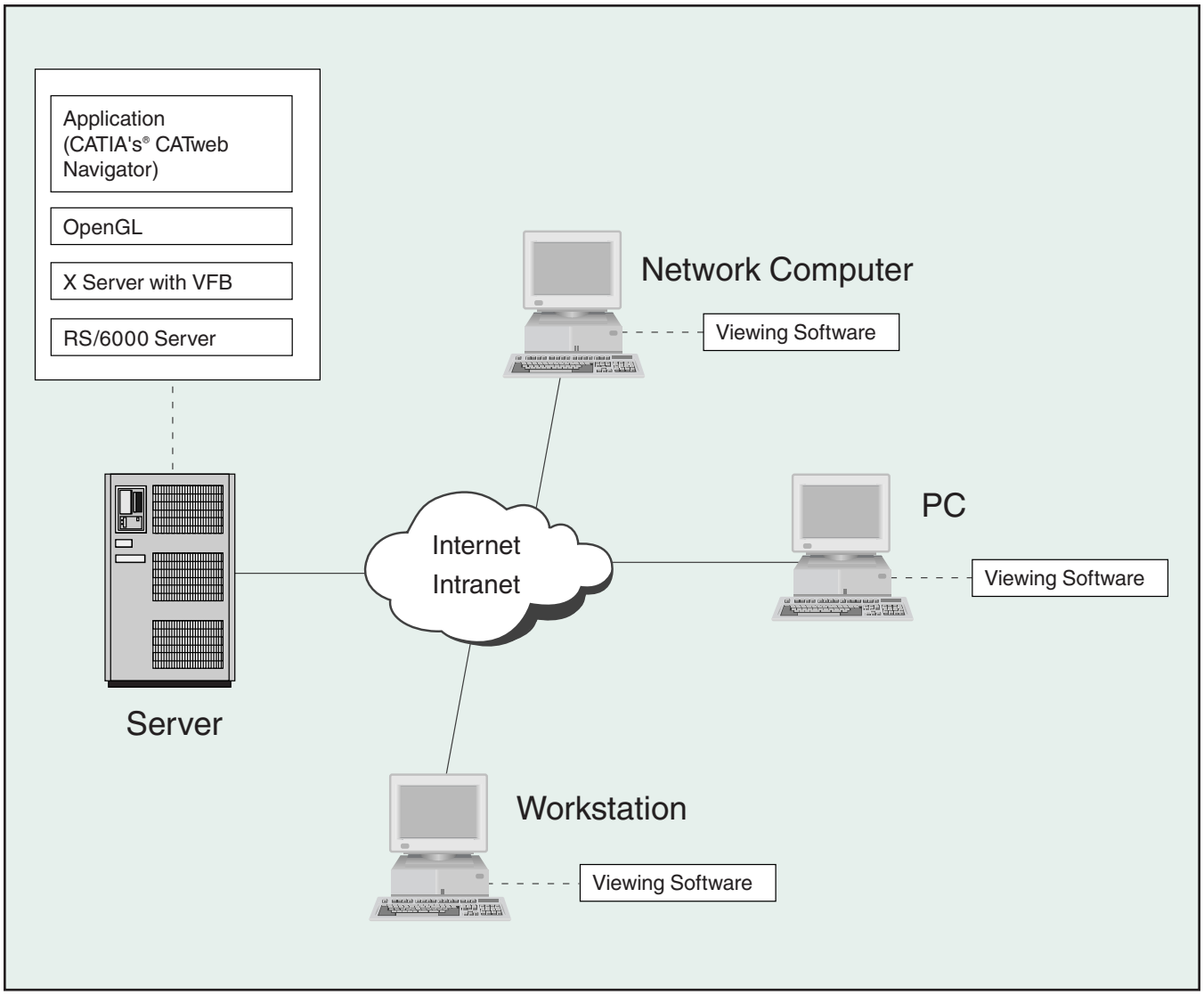

Figure 3. Rendering server environment

Rendering Server Environment

The Virtual Frame Buffer allows the X Server to initialize and run without any physical graphics device present. In the past, the X Server has required one or more graphics devices in order to run, and would exit if no graphics devices were present.

Figure 4 shows the traditional X Server architecture. The shaded section contains rendering software tuned for each graphics adapter. In the VFB solution, this software is replaced by X device-dependent software (ddx) that drives a software graphics adapter. That is, the frame buffer is stored in system memory and all graphics rendering, such as lines or polygons, is handled completely within the software using the system CPU.

VFB is intended for use in a rendering server environment. This means that when the X client application renders an image, the application will then query the image back to the application (for example, `XGetImage`), save it to a file, distribute it to a network, or save it to a database. In this mode, an end user does not directly use the X application interactively. Instead, the application is being driven remotely as a rendering server.

Because there is no physical graphics device, no special hardware, such as RAMDAC,

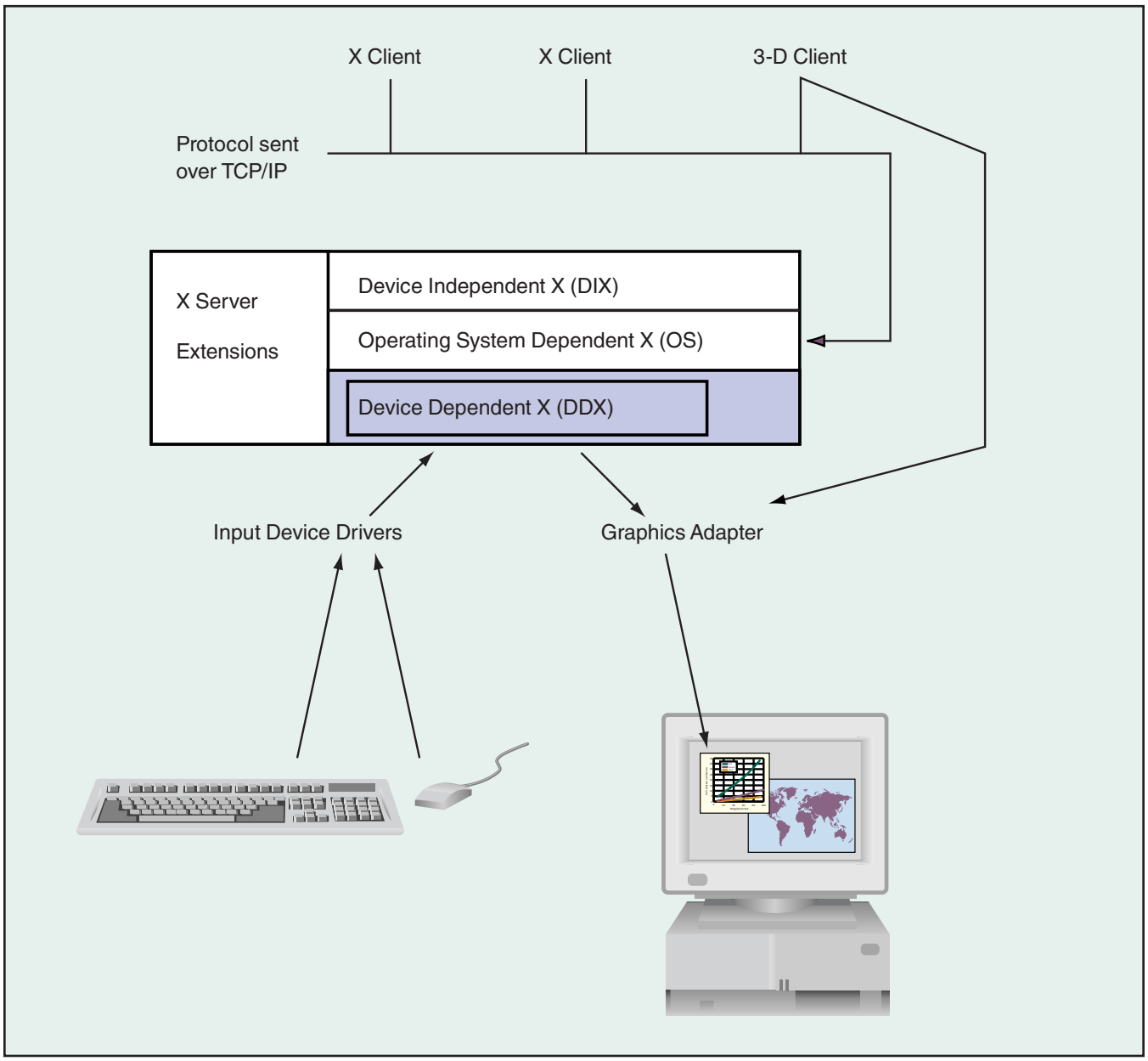

Figure 4. Traditional X environment

is necessary to generate signals used to display the images on the monitor. Therefore, it is impossible to connect a monitor to the system and view the contents of the X Server Virtual Frame Buffer. Although this takes some time to get used to, it is a good solution for rendering server environments, which do not require a high level of interactivity. It also avoids the added expense of a graphics adapter.

One disadvantage, however, is that debugging applications can be more difficult because you cannot see the contents of the frame buffer directly. For this reason, it is helpful to develop applications with a physical graphics adapter, then port the applications to the VFB environment.

X client applications like `xwininfo`, `xwd`, and `xwud` (which are shipped with the AIXwindows® environment) can be used to help verify that your application is running correctly. The `xwininfo` client application provides information about windows, such as the window ID,

geometry, color class (such as PseudoColor, TrueColor), and depth. The xwd application extracts the image and colormap information for a specified window and stores the data in xwd format. Then, xwud displays data that was extracted using the xwd command.

Note: If you are running OpenGL and want to use xwd, set the _OGL_MIXED_MODE environment variable. This environment variable turned on in production mode will slow performance, because drawing occurs in both the vfb frame buffer and the special OpenGL private frame buffer. It is best to use this environment variable in debug mode because its use may slow performance.

The VFB environment does not require input devices. You cannot see the frame buffer, so moving the mouse to do traditional mouse input is impossible. New methods of performing application input are discussed in the next section.

Enhancements to OpenGL

The soft graphics versions of the OpenGL application programming interface (API) contain enhancements designed to work specifically with the Virtual Frame Buffer. These enhancements provide a full software implementation of OpenGL (geometry pipeline and rasterizer), which supports a direct OpenGL context.

OpenGL uses these enhancements to create a private software frame buffer and Z buffer (and possibly alpha, stencil, and accumulation buffers) in system memory for each OpenGL drawable. Consequently, when OpenGL renders graphics, all rendering is drawn in this private software frame buffer. Since each drawable has its own custom-sized software frame buffer to fit the drawable size, rendering into one drawable will not overlap with rendering into another drawable. This is true even if the corresponding X windows for each drawable overlap. Because OpenGl is rendering into its private software frame buffer, there is no data transfer to the OpenGL extension of the X Server. The result is improved performance, since all of the OpenGL graphics commands are rendered directly to the frame buffer.

When images are read back from the drawable (for example, glReadPixels), the read is from the private software frame buffer. As a result, multiple OpenGL rendering server applications can all render concurrently. Since each drawable is rendering to its private software frame buffer, the results from glReadPixels will be correct and complete, because no other application's windows will be occluding parts of the target application's windows.

Enabling multiple applications to render concurrently is a very important feature and causes symmetric multiprocessing (SMP) machines to be fully exploited. For example, running four or more OpenGL applications concurrently on a four-processor SMP machine will cause all four processors to be used.

The VFB and associated OpenGL enhancements are new IBM rendering technologies designed specifically to make 3-D rendering of server applications more efficient and scalable.

To further increase performance, the default OpenGL behavior does not transfer (blit) the contents of the private frame buffer to the VFB frame buffer. As a result, you cannot mix X and OpenGL rendering to a single window. This behavior is fine for most applications, since the image will be queried from the private OpenGL frame buffer using glReadPixels. If required, the transfer behavior can be changed by setting an environment variable (_OGL_MIXED_MODE) that will cause the private frame buffer to be "blitted" to VFB. This environment variable is useful for developing and debugging rendering server applications, but it is not recommended for runtime execution.

```
Start the X Server using XVFB
            /usr/lpp/X11/bin/X -force -vfb -x GLX -x abx -x dbe

      Run the xclient client application
            xclock -display :0.0 &

Use the Xwd and xwud to extract the image of the entire root window on the vfb
X server and display the image on another system
            xwd -display :0.0 -root | xwud -display somedisplay:0.0
```

Figure 5. Rendering of an XVFB server displaying on a second server

Exploiting the VFB Server Environment

Applications should be enhanced in order to operate in a VFB environment. The following represent several enhancements that are needed:

- **Method to extract rendered image from rendering server**. The image to be displayed is extracted or retrieved from the rendering server. Typically, an application would do an `XGetImage` for X applications or a `glReadPixels` for OpenGL applications.

  The programmer must decide for the application the frequency and type of actions that will extract the image from the rendering server. Typically, an application draws an image to the back buffer and "swaps" the back buffer for the front buffer when the image is complete. At this swap time the image could be extracted since the programmer knows that the image is completely drawn. For example, one criteria might be each time the buffer is swapped using the `XdbeSwapBuffers` or `glXSwapBuffer` subroutine.

- **Method to send commands to remote application on server platform**. The application needs input to perform actions, such as opening files or transforming objects. There should be a method to send input to the application from the viewing station. This method could be a command language, a socket connection, interaction with an HTTP server, or another type of communication service. The `XRecord` and `XTest` extensions could be used to send events to the X Server and communicate with the application through traditional X events.

- **Method to display image on a display station**. The VFB environment should contain a method to display the rendered image on the viewing station (for example, the network computer, PC, or workstation). The programmer can determine the display method for the extracted image.

If the viewing station is an AIX workstation executing an AIXwindows X Server, you can use the `xwud` command to display the extracted image. The `xwud` takes an image in the `xwd` format, creates a window, massages the colormap, and does an `XPutImage` to display the extracted image. Figure 5 shows a simple example of rendering on a `XVFB` server and displaying on another server.

More sophisticated methods can include image compression and Java™ display applets. The advantage of a Java display applet is that the image can be displayed on a variety of platform types.

Configuration of the VFB Environment

The `X11.vfb` fileset must be installed before you can use the VFB environment. If it is not available, follow the usual instructions for installing filesets.

Once the `X11.vfb` fileset is installed, start the X Server using the `-vfb` option, shown in Figure 6. In this example, the `-force` option allows you to run the X Server (that is, `X` command) from any shell. If you do not specify the `-force` option, you should invoke the X Server from the low-function terminal (LFT).

```
/usr/bin/X11/X -force -vfb -x GLX -x abx -x dbe
```

Figure 6. The -vfb option

```
/usr/bin/X11/xinit — -force -vfb -x GLX -x abx -x dbe
```

Figure 7. The .xinitrc file

```
xvfb:2:respawn:/usr/bin/X11/X -force -vfb -x GLX -x abx -x dbe >/dev/null
```

Figure 8. Entry to start the X Server automatically


```
/usr/bin/X11/xinit — -force -vfb -x GLX -x abx -x dbe
```

Figure 7. The .xinitrc file


```
mkitab "xvfb:2:respawn:/usr/bin/X11/X -force -vfb -x GLX -x abx -x dbe >/
dev/null"

rmitab "xvfb"
```

Figure 9. Commands to remove entries for automatically starting X Server

The `-vfb` option instructs the X Server to ignore any physical graphics devices that may be present on the system and use the Virtual Frame Buffer instead. The remaining options (`-x GLX`, `-x abx`, and `-x dbe`) instruct the X Server to load extensions that are necessary to render through OpenGL.

By invoking the X Server directly, not by calling `xinit` or invoking it through the desktop, no client applications—including the window manager—are automatically started. Executing without a window manager in a rendering server environment is generally fine because windows will not be manipulated directly with the mouse or keyboard.

To invoke a window manager, you can invoke the X Server directly, as shown in the example above, and then invoke `mwm`, `dtwm`, or some other window manager. In the example shown in Figure 7, the `xinit` command both starts the X Server and several client applications including `mwm`. The client applications that are started automatically are configured in the `.xinitrc` file.

When the X Server is used in a rendering server role, it is often desirable to have the X Server start automatically. This can be done by adding an entry to the `/etc/inittab` file. The entry shown in Figure 8 will cause the X Server to start automatically when the system boots, and also to restart automatically if the X Server should die.

Figure 9 shows two commands that can be used to add or remove the entry shown in Figure 4 from the `/etc/inittab` file.

Some applications may need to execute differently when running in the Virtual

```
/usr/lpp/X11/Xamples/bin/xprop -root | grep XVFB
```

Figure 10. Command to check for XVFB use

Frame Buffer environment. To facilitate applications and tune their behavior for XVFB, the X Server automatically sets the XVFB_SCREEN property to TRUE for any screen that is using XVFB.[3] This property is stored on the root window of any VFB screen.

Users can check the XVFB property by using the xprop command, a utility to display X Server properties associated with windows and fonts. To check for XVFB use the command shown in Figure 10.

To check for the XVFB property, the application would use the XInternAtom

```
      Atom   atom;
Atom  actual_type;
Display      *display;
Screen       screen;
unsigned long       nitems;
unsigned long       bytes_after;
unsigned char       *prop;
int                vfb_screen = False;
.
.
.
/* SET UP SCREEN VARIABLE*/
.
.
.
.
.
/* OBTAIN THE ATOM THAT CORRESPONDS TO THE "XVFB_SCREEN" STRING*/
atom{1} = XInternAtom (display, "XVFB_SCREEN", True);

if (atom != none) {

      /* ATOM EXISTS SO GET THE PROPERTY*/
      if ((status = XGetWindowProperty (display, RoowWindow(display, screen),
             atom, 0, 100, False, atom, &actual_type,&actual_format,
             &nitems, &bytes_after, &prop)) == True ){

             /* BE SURE THAT THE PROPERTY IS SET TO TRUE*/
             if (strcmp((char *)prop,"TRUE") == 0) {
                   vfb_screen = True;
              }
      }
}
```

Figure 11. Checking for the XVFB property

[3] A property is an X Window System® concept. Each property consists of a name, type, and data. Each property is associated with some X Server resource such as a window, font, pixmap, or colormap. X Window System subroutines are available to obtain property information.

```
screen #0:
 dimensions:    1280x1024 pixels (325x260 millimeters)
 resolution:    100x100 dots per inch
 depths (2):    1, 24
 root window id:    0x26
 depth of root window:    24 planes
 number of colormaps:    minimum 1, maximum 1
 default colormap:    0x24
 default number of colormap cells:    256
 predicated pixels:    black 0, white 16777215
 options:    backing-store NO, save-under NO
 current input event mask:    0x0
 number of visuals:    3
 default visual id: 0x21
   visual:
   visual id:    0x21
    class:    TrueColor
    depth:    24 planes
    size of colormap:    256 entries
    red, green, blue masks:    0xff0000, 0xff00, 0xff
    significant bits in color specification:    8 bits
 visual:
    visual id:    0x22
    class:    TrueColor
    depth:    24 planes
    size of colormap:    256 entries
    red, green, blue masks:    0xff0000, 0xff00, 0xff
    significant bits in color specification:    8 bits
 visual:
    visual id:    0x23
    class:    TrueColor
    depth:    24 planes
    size of colormap:    256 entries
    red, green, blue masks:    0xff0000, 0xff00, 0xff
    significant bits in color specification:    8 bits
```

Figure 12. Sample output for VFB screen

and `XGetWindowProperty` subroutines. Figure 11 shows one way that the code could be written.

The current `XVFB` screen only supports 24 bit visuals.[4] The `xdpyinfo` command will return X Server and visual information about screen #0. Figure 12 shows sample output for a VFB screen.

The maximum number of concurrent client applications for an IBM X Server is 128. Client number 129 and those beyond will receive the error message shown in Figure 13.

```
Xlib: Connection to “displayname” refused by server
Xlib: Maximum number of client reached.
```

Figure 13. Error message for clients above 128

[4] A visual is an X Window System concept that gives hints about the configuration of the rendering environment such as depth, pixel type, organization, number, and size of colormaps.

```
/use/bin/X11/X  -vfb -x GLX -x dbe -x abx -auth $HOME/.Xauthority

xinit — -vfb -x GLX -x abx  -auth $HOME/ .Xauthority
```

Figure 14. Invoking an X Server with .Xauthority

```
Display     *display
GLXContext  context
int   OGLEnhancement = False:
.
.
.
.
if ((glXIsDirect(display,context) == True) &&
    (strcmp(glGetString(GL_RENDERER), "SoftRaster") == 0){}
      OGLEnhancement = True;
}
```

Figure 15. Checking for the VFB environment

Security Configuration in X Window System

This section describes several hints about the security configuration in the X Window System environment.

The display access control of the XVFB X Server is the same as the access control for any other X Server: it can be host-based or it can use the `MIT_MAGIC_COOKIE`. When using host-based access, any client on a host in the host access control list can access the X Server. The list of allowed hosts is stored in the X Server and can be changed with the `xhost` command. `xhost+` will allow all remote X client applications to connect to the X Server.

Another method is to create an `/etc/X0.hosts` file. Any host name added to this file can connect to the X Server. The `-ac` flag is specified when the X Server is invoked. The result is the same as using `xhost+`.

The IBM X Server also supports the `MIT_MAGIC_COOKIE_1` mechanism for access control. When using these mechanisms, the client sends a cookie with the connection setup information. Cookies are machine-readable, randomly generated access values that are stored in a `~/.Xauthority` file. When the X Server starts a session, it reads a cookie from the `~/.Xauthority` (or other file specified by the user in the `-auth` argument to the X Server). Subsequently, only clients that know the cookie can connect to the X Server. When it is used this way, `xauth` can limit access to specific users on specific systems.

The `~/.Xauthority` file can be generated each time the X Server is invoked. It then is copied to a remote system when a user wants to connect. Figure 14 shows how to invoke the X Server by specifying an `.Xauthority` file.

The `.Xauthority` file can be generated by many different methods. A random number generator usually creates the key, and the `xauth` command is called to add the key into the `.Xauthority` file.[5] Scripts can be written to copy the `.Xauthority` file to client workstations when specific users log on to the host system.

OpenGL Configuration

To ensure that the OpenGL enhancement for VFB is available, your application should

[5] The `xauth` command, `xdm` command, and other access information are documented as part of the AIX publications.

verify that a direct context and a soft rasterizer are available. In direct context, OpenGL is rendering directly to the frame buffer (for example, hardware or VFB). This value can be determined by using the `glXIsDirect` subroutine. A soft rasterizer means that OpenGL is rendering by using software and not hardware. Both a direct context and soft rasterizer must be true to use the OpenGL enhancement. Figure 15 shows sample code that you can add to your application to check for the VFB environment.

VFB Server-supported Operating Environments

Virtual Frame Buffer Server-supported operating environments include AIX 4.1.5, AIX 4.2.1, AIX 4.3.1, or later versions.

Sample code for modifying applications to exploit the VFB environment can be found at *http://www.rs6000.ibm.com/solution/interactive/rendserv.html*.

Jeanne Sparlin, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Ms. Sparlin is currently working in the Business Development Department of IBM Visual Systems. She has previously worked in the Data Management, SNA, Dialog Design Aid, and the X Window System development departments on the RS/6000 and RT/PC. She has a degree from Truman State University.

Andrew Taylor, IBM Corporation, 11400 Burnet Road, Austin, TX 78758.