SiliconGraphics

Pipeline

September/October 1992
Volume 3, Number 5
Bringing you the latest news and information from Silicon Graphics' Customer Support resources.

Table of Contents

1 Vizualizing Transformations and Projections - A Right Brain Approach
4 Handling Colormaps in Mixed Model Applications
8 Q&A
10 InfoAccess
INSERT: Customer Support Guide

Vizualizing Transformations and Projections - A Right Brain Approach

The matrix projections and transformations in the graphics library provide enough flexibility for most people, but some developers require non-standard matrix operations, and need to know more about the mathematics, which turns out to be (real) projective geometry.

The calculations are simple, but may appear mysterious or even somewhat magical the first time through. You might ask, "Why do we need the 'w' coordinate?", or "How was the formula for perspective derived and why does it work?" A complete de-mystification requires both a knowledge of the calculations (a left-brain task), and some good geometric mental images for your right brain. This article relates the calculations to a good mental model of projective geometry.

With a clear mental picture, it's not hard to answer questions such as:

Is there any way to discover the location of the viewing-position after transforming the scene with a matrix?

Why are my pictures so horrible when I set my near clipping plane to 0.0000001 and my far clipping plane to 1000000.0.?

How do I build a rotation matrix about an arbitrary axis?

How do I do a shearing transformation?

What's the formula for a transformation that does 2-point

(continued on next page)

Visualizing Transformations...

(continued from previous page)

perspective (the `GL perspective()` *command is 3-point perspective)?*

Homogeneous Coordinates and Projective Geometry

Projective geometry is not the same as Euclidean geometry, but it is closely related. This article considers 2 and 3 dimensional projective geometry. Since (as in Euclidean geometry) it's easier to visualize and draw things in 2 dimensions, we'll begin with 2D projective geometry.

> *Two postulates from 2D Euclidean geometry:*
>
> *-Every two points lie on a line.*
>
> *-Every two lines lie on a point, unless the lines are parallel, in which case, they don't.*
>
> *In 2D projective geometry, the postulates are replaced by:*
>
> *-Every two points lie on a line.*
>
> *-Every two lines lie on a point. (In other words, there are no parallel lines in projective geometry.)*

How can we visualize a 2D projective geometry model? The model must describe all the points and lines, what points are on what lines, etc. The easiest way is to take the points and lines from a standard 2D Euclidean plane and add elements until the projective postulates are satisfied.

The first problem is that the Euclidean parallel lines don't meet. Lines that are almost parallel meet way out in the direction of the lines, so for parallel lines, add a single point for each possible direction and add it to all the parallel lines going that way. You can think of these points as being points at infinity—at the "ends" of the lines. Note that each line includes a single point at infinity—the north-south line doesn't have both a north and south point at infinity. If you "go to infinity" to the north and keep going, you will find yourself looping around from the south. Projective lines form loops.

Now take all the new points at infinity and add a single line at infinity going through all of them. It, too, forms a loop that can be imagined to wrap around the whole original Euclidean plane. These points and lines make up the projective plane.

You might form a mental picture like the one shown in figure 1. The solid circle around the outside is the line at infinity, and it is beyond all the points on the Euclidean plane (shown with a fuzzy edge). The points labelled L_∞ are on the line L, and represent the same point.

Check the postulates. Two points in the Euclidean plane still determine a single projective line. One point in the plane and a point at infinity determine the projective line through the point and going in the given direction. Finally, the line at infinity passes through any two points at infinity.

How about lines? Two non-parallel lines in the Euclidean plane still meet in a point, and parallel lines have the same direction, so meet at the point at infinity in that direction. Every line on the original plane meets the line at infinity at the point at infinity corresponding to the line's direction.

Note: The projective postulates do not distinguish

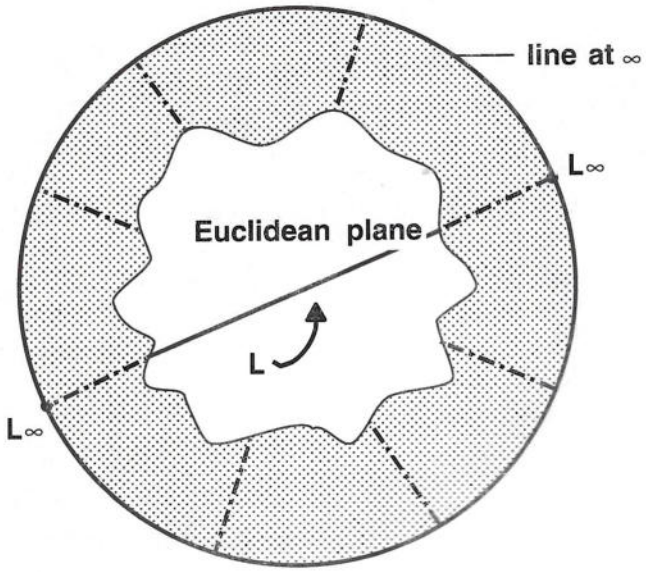


figure 1

between points and lines in the sense that if you saw them written in a foreign language:

Every two glorphs lie on a smynx,

Every two smynxes lie on a glorph,

there would be no way to figure out whether a smynx is a line and a glorph is a point or vice-versa. If you take any theorem in 2D projective geometry and replace "point" with "line" and "line" with "point", it makes a new theorem that is also true. This is called "duality"—see any text on projective geometry.

So we've got a nice mental picture—how do we assign coordinates and calculate with it? The answer is that every triple of real numbers [x, y, w] except [0.0, 0.0, 0.0] corresponds to a projective point. If "w" is non-zero, [x, y, w] corresponds to the point [x/w, y/w] in the original Euclidean plane; [x, y, 0.0] corresponds to the point at infinity corresponding to the direction of the line passing through [0.0, 0.0] and [x, y]. Generally, if α is any non-zero number, the homogeneous coordinates [x, y, w] and [αx, αy, αw] represent the same point.

It's a bit disturbing that the same projective point can be represented in many different ways. For example, [1,2,1], [2,4,2], and [-11,-22,-11] all refer to the Euclidean point [1,2]. But don't panic, you saw the same kind of thing in third grade — the fractions 1/2, 2/4, and 55/110 all represent the same number. Just as we usually use the fraction reduced to lowest terms, we usually use projective coordinates with w equal to 1.0 when that's possible.

Since projective points and lines are in some sense indistinguishable, it had better be possible to give line coordinates as sets of three numbers (with at least one non-zero). If the points are row vectors (as in the GL), the lines will be column vectors (written with a "T" exponent that represents "transpose"), so $[a, b, c]^T$ represents a line. The point P = [x, y, w] lies on the line $L = [a, b, c]^T$ if ax+by+cw = 0. In the Euclidean plane, the point [x, y] can be written in projective coordinates as [x, y, 1.0], so the condition becomes ax+by+c=0 — high-school algebra's equation for a line. The line passing through all the points at infinity has coordinates $[0.0, 0.0, 1.0]^T$. As with points, for any non-zero α, the linecoordinates $[a, b, c]^T$ and $[\alpha a, \alpha b, \alpha c]^T$ represent the same line. In matrix notation, the point P lies on the line L if and only if PL=0. Choosing to represent lines as column vectors and points as row vectors, would work as well. It has to work because points and lines are dual concepts.

Projective Transformations

Projective transformations transform (projective) points to points and (projective) lines to lines such that incidence is preserved. Thus, if T is a projective transformation and points P and Q lie on line L then T(P) and T(Q) lie on T(L). Similarly, if lines L and M meet at point P, then the lines T(L) and T(M) meet at the point T(P).

The reason projective transformations are so interesting is that if we use the model of the projective plane described above where we've simply added some elements to the Euclidean plane, the projective transformations restriced to the Euclidean plane include all rotations, translations, non-zero scales, and shearing operations. This would be powerful enough, but if we don't restrict the transformations to the Euclidean plane, the projective transformations also include the standard projections, including the very important perspective projection.

Rotation, translation, scaling, shearing (and all combinations thereof) map the line at infinity to itself, although the points on that line may be mapped to other points at infinity. For example, a rotation of 20 degrees maps each point at infinity corresponding to a direction to the point corresponding to the direction rotated 20 degrees (see figure 2). Pure translations preserve the directions, so a translation maps each point at infinity to itself.

The standard perspective transformation (with a 90 degree

(continued on page 6)

The following is a collection of do's, don'ts and tips for mixed model programming with respect to colormaps. A sample program is provided at the end of the article which uses a custom colormap for the normal buffer and a separate custom colormap for the overlay/popup buffer. The program utilizes the GL widget.

The Basics

1. When dealing with colormaps for mixed model applications remember that it is the X Window System policies and function calls which are pertinent. Any similar GL functionality is not legal. An example of this would be the use of mapcolor.

2. All mixed model programs, even those using RGB mode for the normal buffer, need to call XSetWMColormapWindows. This function will identify windows that need to have colormaps installed by the window manager (through the use of the WM_COLORMAP_WINDOWS property). Calling this routine is necessary for RGB mode because some Silicon Graphics platforms implement the TrueColor visual through the use of colormaps. See the `install_colormaps()` function in the following cmapov.c program for a general routine to perform this operation for a single GL widget.

The xprop program (in */usr/bin/X11*) can be used to verify that the windows are contained in the appropriate property. Part of the output should look something like this:

```
WM_COLORMAP_WINDOWS(WINDOW): window id #
88080402, 0x5400011, 0x540000e
```

Here three windows have been added. The first window id is mentioned in decimal and the rest are in hexadecimal. The windows are given in priority order, with the first window receiving the highest priority to have its colormap installed into the hardware map.

An application is not limited to calling XSetWMColormapWindows just once. Whenever this function is called, the property will be updated. If the client has focus, the window manager will reexamine the new list and perform colormap installation. Knowing this, it would be possible for a client to contain multiple GL widgets, each with a different colormap. The correct one could be installed by monitoring EnterNotify and LeaveNotify events for the windows of the GL widgets. Some translations for these new events to an appropriate callback function, for example, glxInput, would be needed in this scheme.

There are also window manager functions to cycle through the entries on this list, causing each window's colormap installation priority to change. These functions are `f.next_cmap and f.prev_cmap`. Modifying the window manager initialization file (.4Dwmrc) will allow access to them. There is no standard way to add the functions; one possible way would be to add the following lines to the 4DwmKeyBindings:

```
# for colormap cycling
Ctrl<Key>F1        window        f.next_cmap
Ctrl<Key>F2        window        f.prev_cmap
```

Note that this removes the ability of any application to use Ctrl+F1 or Ctrl+F2 for any other purpose (which could be highly undesirable).

3. The default mixed model GL colormaps should not be modified. If custom colors are needed then new colormaps should be created, modified to the desired mapping, and finally used for the GL rendering area, be it an X window or a GL widget.

In order to create a colormap you need 3 basic pieces of information: the display, any window on the screen for which the colormap will be needed (the root window is fine) and the visual with which the colormap will be used. The display and root window are easy to obtain. The visual used for GL rendering in a mixed model application is obtained differently depending on the level at which you are working with respect to X/Motif.

At the Xlib level the visual can be obtained anytime after the call to `GLXgetconfig(3G)` is done. Visuals are associated with GLXconfig entries that have a mode of `GLX_VISUAL`. Typically some routines which linearly search through a GLXconfig array are useful for setting and getting values like this. Once the visual is found, a new colormap can be created using XCreateColormap. It should then be passed as an attribute to the window during the window's creation via the XSetWindowAttributes mechanism. (see *~4Dgifts/examples/GLX/gl-Xlib* for examples of this process).

At the Toolkit/Motif level the visual can be obtained anytime after the GL widget has been created. The visual for the normal buffer is stored in the standard resource XmNvisual. The visuals for the other buffers are stored in GL widget-specific resources (GlxNoverlayVisual, GlxNpopopVisual, GlxNunderlayVisual). Once the visual has been obtained, the colormap can be created using XCreateColormap. The new colormap should then be passed to the GL widget using XtSetValues, overriding its default colormap setting. Note that modifying the colormap of a GL widget after it is created is allowable due to its unique design; standard Motif widgets should not receive a new colormap after creation.

4. When a new colormap is going to be installed by the window manager only those cells in the new colormap which have been allocated and changed are installed. Thus colormap installation is not an "all-or-nothing" process. You might have a colormap for a 4096 color PseudoColor visual with only 3 colors allocated. In this case, for a real estate driven focus policy, when the mouse cursor travels into the client only 3 hardware colormap entries need to be changed — the rest are left to what they were prior to the cursor entering the client's window.

5. If custom colormaps are created an application should choose to reduce colormap flashing in general, and in particular for the normal buffer.

Colormap flashing occurs when a virtual colormap is installed into the hardware colormap causing colors to suddenly be interpreted to different RGB values. The most bothersome flashing typically occurs when values at the "low end" of the hardware colormap are changed. This is due to the fact that the colors at low indices are used more commonly for entities across the whole screen (such as window frames and WorkSpace colors). "Bothersome" is used very loosely here; it can happen at high indices as well.

Clients needing their own virtual colormap can create one using the XCreateColormap function. There are, then, two cases to examine with respect to colormap flashing: the AllocAll case and the AllocNone case (taken from the final parameter passed to XCreateColormap).

A. AllocAll:

In this case every color cell in the colormap has been allocated with read/write permission. This gives the application the flexibility to modify all of its colors at any time. The initial values of the cells in the virtual colormap are undefined. In this state they will not be loaded when the colormap is installed by the window manager.

The "lazy" method of reducing colormap flashing in this case is to simply place client specific colors at the far end of the colormap and allow all other colors to stay at whatever they were prior to having your colormap installed. Flashing will be minimized with fewer colors changing. If your client ever uses any of these "default" colors there could be problems though. The client may be displayed in false colors due to a transition from an application with a non-default colormap to your client.

The "minimalist" method then to avoid flashing and display correct colors in this instance would be to have the application copy the specific colors it uses from the default X colormap to its virtual colormap (and place its unique

(continued on page 9)

Visualizing Transformations...

(continued from page 3)

field of view, the eye at the origin, and looking down the y-axis) maps the origin to the point at infinity in the y-direction. The viewing trapezoid maps to a square (see figure 3).

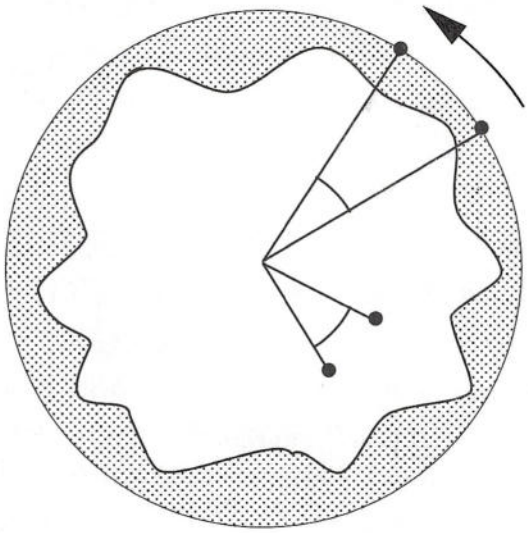


figure 2

Every non-singular 3x3 matrix (non-singular means that the matrix has an inverse) represents a projective transformation, and every projective transformation is represented by a non-singular 3x3 matrix. If M is such a transformation matrix and P is a projective PM is the transformed point. If L is a line, $M^{-1}L$ represents the transformed line. It's easy to see why this works: if P lies on L, PL= 0, so $PMM^{-1}L = 0$, so $(PM)(M^{-1}L) = 0$. The matrix representation is not unique — as with points and lines, any constant multiple of a matrix represents the same projective transformation.

Combinations of transformations are represented by products of matrices; a rotation represented by matrix R followed by a translation (matrix T) is represented by the matrix RT. A (2D) projective transformation is completely determined if you know the images of 4 independent points (or 4 independent lines). This is easy to see, as a 3x3 matrix has nine numbers in it, but since any constant multiple represents the same transformation, there are basically 8 degrees of freedom. Each point transformation that you lock down eliminates 2 degrees of freedom, so the images of 4 points completely determine the transformation.

Let's look at a simple example of how this can be used by deriving from scratch the rotation matrix for a 45 degree rotation about the origin. The origin maps to itself, the points at infinity along the x and y axes map to points at infinity rotated 45 degrees, and the point [1, 1] maps to $[0, \sqrt{2}]$. See figure 4.

If R is the unknown matrix:

$[0, 0, 1]R = k_1[0, 0, 1]$

$[1, 0, 0]R = k_2[\sqrt{2}, \sqrt{2}, 0]$

$[0, 1, 0]R = k_3[-\sqrt{2}, \sqrt{2}, 0]$

$[1, 1, 1]R = k_4[0, \sqrt{2}, 1]$

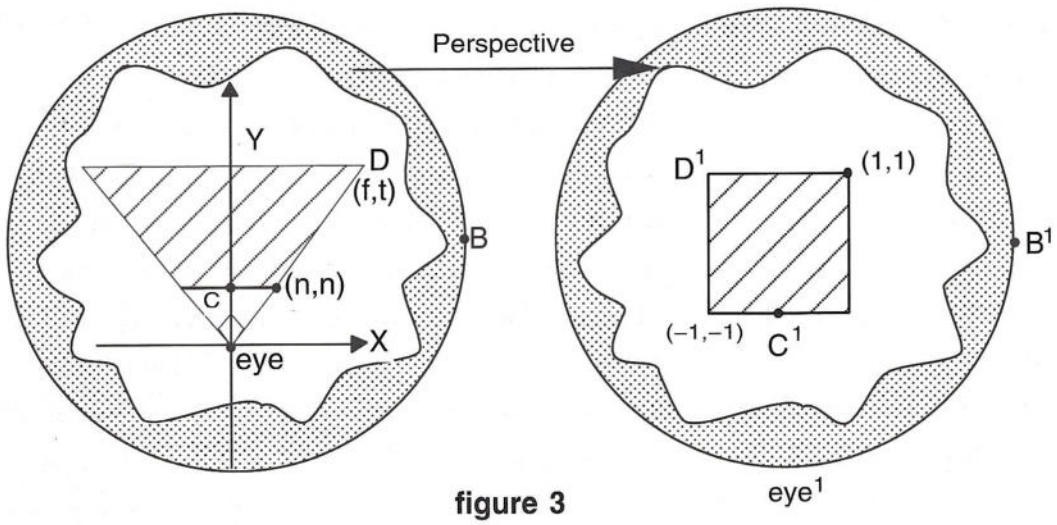


figure 3

The $k_1, \ldots, k_4$ can be any constants since any multiple of a projective point's coordinates represents the same projective point. M has basically 8 unknowns, so those 8 plus the 4 k_i's make 12. Each matrix equation represents 3 equations, so there is a

system of 12 equations and 12 unknowns that can be solved. The computations may be cumbersome, but it's a straight-forward brute-force solution that gives the rotation matrix as any multiple of:

$$R = \begin{bmatrix} \sqrt{2}/2 & \sqrt{2}/2 & 0 \\ -\sqrt{2}/2 & \sqrt{2}/2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

There's nothing special about rotation. Every projective transformation matrix can be determined in the same brute-force manner starting from the images of 4 independent points.

We illustrate with the determination of the (2D) perspective transformation (see figure 3). We want to map the shaded trapezoidal area into the square with corners [-1, -1] and [1, 1]. The unknown projection must satisfy:

$[0, 0, 1]P = k_1[0, 1, 0]$

$[0, n, 1]P = k_2[0, -1, 1]$

$[f, f, 1]P = k_3[-1, 1, 1]$

$[1, 0, 0]P = k_4[1, 0, 0]$

The same brute-force calculation gives the matrix P as (any multiple of):

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -(f+n)/(f-n) & -1 \\ 0 & 2fn/(f-n) & 0 \end{bmatrix}$$

Three Dimensional Projective Space

3D projective space has a similar model. Take 3D Euclidean space, add points at infinity in every 3-dimensional direction, and add a plane at infinity going through the points. In this case there will also be an infinite number of lines at infinity as well. In 3D, points and planes are dual objects.

Projective transformations in 3 dimensions are exactly analogous.

Points are represented by 4-tuple row vectors: [x, y, z, w], and planes by column vectors: $[a, b, c, d]^T$. Any multiple of a point's coordinates represents the same projective point. A point P lies on a plane M if PM = 0. All 3D projective transformations are represented by 4x4 non-singular matrices.

In 3 dimensions, the images of 5 points (or planes) completely determine a projective transformation. (A 4x4 matrix has 16 numbers, but 15 degrees of freedom because

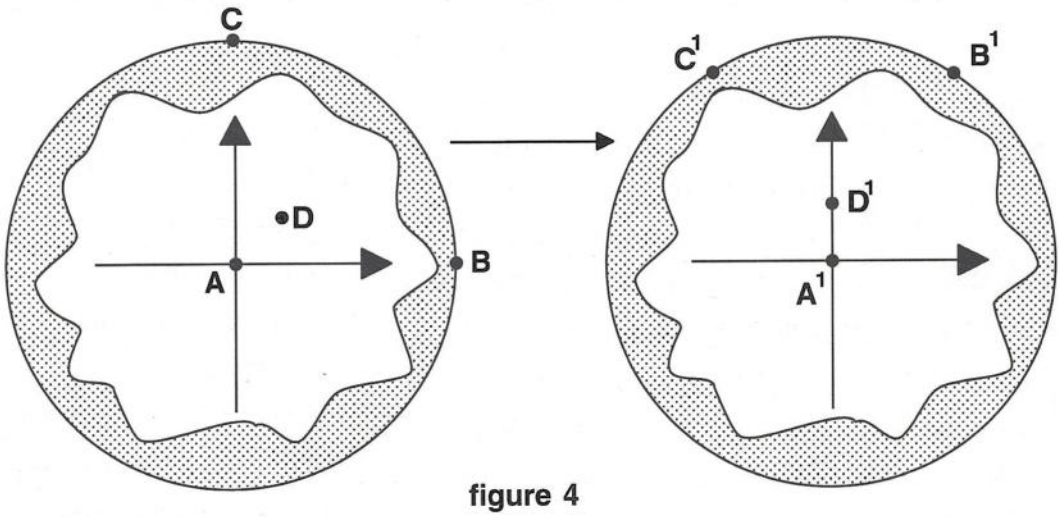


figure 4

any multiple represents the same transformation. Each point transformation that you nail down eliminates 3 degrees of freedom, so the images of 5 independent points completely determine the transformation.)

The brute-force solution has 20 equations and 20 unknowns (there will be 5 k_i's in addition to the 15 unknowns), and although the solution is time-consuming, it is straight-forward.

The calculation can be simplified. Suppose you want a transformation that takes P_1 to Q_1, ..., and P_5 to Q_5. Let

$I_1 = [1, 0, 0, 0]$

$I_2 = [0, 1, 0, 0]$

$I_3 = [0, 0, 1, 0]$

(continued on next page)

Visualizing Transformations. . .

(continued from previous page)

$I_4 = [0, 0, 0, 1]$

$I_5 = [1, 1, 1, 1]$

find the transformation P that takes P_i to I_i and the transformation Q that takes Q_i to I_i. Because of all the zeroes, these are much easier to work out. The transformation you want is PQ^{-1}.

Using the Mental Model

To review the questions from the first section — Perspective projections map the eye point to infinity, so if you know the projection matrix and want to find the eye point, find the point that maps via the projection to infinity in the z-direction.

If you're wondering about the bizarre effects of widely spaced near and far clipping planes, look at how much stretching occurs between the origin and .0000001.

Construction of the three projections is simply a matter of listing 5 independent points and their images and calculating the matrix by brute force.

The more ways you have to look at a mathematical concept, the better you will understand it. Perhaps the mental images here may provide additional insight.

Q: Is there a control sequence or a way to change the xwsh background after I've started up the window?

A: See the section under "DCS Pn .y Ps ST" in the ESCAPE SEQUENCES of the man page for xwsh(1G). You may want to use command `5.` DCS is `<ESC P>` or `\033P`. Pn is 5 ".y" is ".y". "Ps" is the color string (e.g. `gray50`). `ST` is `<ESC \>` or `\033\\`.

For example, the following command:

```
echo "\033P5.ygray50\033\\"
```

will set the background to `gray50`. And:

```
echo "\033P5.ymidnightblue\033\\"
```

will set the background to `midnightblue`.

Q: Is it true that SGI machines are smart enough not to send out packets across the network when two processes are communicating on the same workstation? Does IRIX 4.0.1 work this way?

A: Most systems do not send network data out over the media when the two end-points are on a single system. For example,

```
rcp localhost:/etc/hosts localhost:/tmp
```

involves about 6 TCP/IP virtual circuits, 3 copies of rcp, 2 of rsh, and 2 of rshd, but none of the data will ever touch the token ring, FDDI, or ethernet on an IRIS. This is because SGI machines are unable to transmit and receive at the same time on ethernet, and because it is too unreliable and slow to talk to yourself over FDDI and token ring. Sometimes the machine can receive its own FDDI or token ring transmissions and sometimes it cannot, depending on the size of the ring and other uncontrollable factors.

A consequence of this policy is often unnoticed. `ping`ing the broadcast address produces an answer even when the ethernet or other network hardware is broken. This is because broadcasts and multicasts are detected in the transmit code, and looped back in software. To see a tool that is used to tune systems and make them faster, try the following commands;

In one window, type:

```
ttcp -r -s -132768
```

In another window, enter;

```
ttcp -t -s =132768 localhost
```

Note: The name `localhost` can be replaced by any name

(continued on page 12)

Handling Colormaps...

(continued from page 5)

colors up high). The colors which are copied are replicated by position and RGB value. This is demonstrated in the 4Dgifts example: *~4Dgifts/examples/GLX/glxwidget/demos/mscrn_rotate_btn.c* in unison with `copycmap.[ch]` (in the same directory).

A "peace-of-mind" method would be to copy an entire block of lower colors from the default map to the virtual map (whether or not they are used). The cmapov.c code provided below demonstrates this method. It attempts to replicate the first 256 default X colors (if there are that many) in its colormap, and begins placing the custom colors starting at the high end of the new colormap and working backwards towards lower indices. Obviously if there are 256 (or less) colors available then this allocation process begins overwriting the default colors with custom colors.

Note that these methods do not eliminate colormap flashing; they only attempt to reduce it. Also, do not confuse this copying of colors with color sharing. Sharing takes place when multiple applications attempt to agree where colors will be in a given colormap which they will use. The methods above are for a custom colormap which is in effect "emulating" another (default) colormap.

B. AllocNone

In this case no color cells are allocated in the colormap. The client may choose to allocate either read or read/write color cells at this point. As well, the client may choose to leave some cells unallocated. Initially the cells have no definition.

In terms of needing a custom colormap and reducing colormap flashing this (the AllocNone case) is not the most logical method to use. The allocation routines available will not allow an application to place its colors at specific indices which inhibits emulating the color values of the default colormap. Color placement can be controlled if the client attempts to allocate all the cells in the colormap after creating it, but this basically reduces this method to the AllocAll case.

The only extra ability gained then is that you can obtain a custom colormap whose cells are all allocated read-only. This doesn't serve any real purpose though, unless this custom colormap will be shared by multiple applications.

On systems with only 8 bitplanes, 1 hardware colormap, and using double buffering for the normal buffer, it will be difficult to avoid flashing if custom colors are desired. This is because only 16 colors will be available for the two 4-bit colormapped buffers which are created. Thus when a program attempts to place its custom colors "at the high end of the colormap", they are really modifying colors with "very low indices" (13, 14, 15, for example). The 8 bit Personal Iris is most prone to this. The starter Indigo (which is also an 8 bit machine) can usually avoid this problem by relying on its multiple hardware colormaps.

6. The showmap and xshowcmap commands can be useful in debugging colormap setups. They are less useful on machines with more than one hardware colormap, and not useful at all for debugging colormaps for buffers other than the normal buffer.

Use of the Overlay Buffer

1. Some machines do not have overlay planes. In this case you can attempt to use the popup planes. The getgdesc GL function allows a program to check for this situation. Also, the two buffers are not completely inter-changeable because you will have at most 2 popup planes, but you may have more than 2 overlay planes.

2. Overlays always run in colormap mode for Silicon Graphic's current systems.

3. Although VGX (and VGX alone) supports double buffered overlays within pure GL applications, it does not

(continued on next page)

Handling Colormaps...

(continued from previous page)

work for mixed model clients.

4. Remember to set GlxNuseOverlay to True before creating the GL widget (or GlxNusePopup for the popup bitplanes).

5. Remember to add a GlxNoverlayExposeCallback (or GlxNpopupExposeCallback for the popup planes).

6. Note that it is not possible to allocate all of the colormap entries for an overlay colormap because color index 0 is reserved for transparency. Thus using AllocAll when invoking the XCreateColormap function will fail. Instead AllocNone should be used, and then a program can allocate the colors it needs using other X functions (e.g. XAllocColor, XAllocColorCells).

7. If a mixed model client is using both the normal and overlay buffer the viewport needs to be set for both of them—not just the normal buffer. (see the `cb_gl_resize` function in cmapov.c below).

Other Mixed Model Hints

1. There is no need to do a redraw for the GL widget during the resize callback because the next event will cause an expose callback. A number of the 4Dgifts' GLX examples do this but it is redundant.

2. The ginit callback of the GL widget is called after the widget is realized (thus its X window(s) exist at this point). In fact it is the very first callback invoked for the gl widget in all mixed model programs. If you need to do initialization which relies on the existance of the actual X windows for any of the buffers, this is a good place to do it.

3. Underlays are not supported in mixed model

The following is a list of faxables that was created to help you save time in solving your computer problems. To have one faxed to you, call the Silicon Graphics' Hotline at 1(800)800-4SGI and request it by name.

COMMUNICATIONS

Beta Instructions for using IVR.

Setting up `sendmail.cf` notes.(3.3.X)
Hints on how to configure the `sendmail.cf` *file for IRIX*

Setting up `sendmail.cf` notes.(4.0.1)
Hints on how to configure the `sendmail.cf` *file for IRIX*

LPR setup instructions.
Installing, configuring and troubleshooting tips for the Berkeley spooling system.

Changing `tcp_sendspace` instructions.
Modification of `sendspace` *for negotiating tcp window sizes. These instructions might be needed for 4.0.X and PC connectivity*

Modem setup instructions.
Configuring a dial-in/dial-out modem, and helpful hints
Serial pinouts.
Cable configuration information for connecting a terminal, printer or modem.

UNIX

Partitioning a disk drive notes (Pipeline Article).

How to change the size of swap space on a disk.

Logging in via Pandora on 4.0.1 sends user back to Pandora.
Correcting login problems under IRIX 4.0.1

Unix Q & A for 4.0.1

Configuring floppy drives on SGI systems

Reading DOS formatted floppies on SGI systems

GRAPHICS

Graphics corrected Explorer 1.0 `README` file

Graphics Q & A for 4.0.X

Example Code

The primary purpose of the following code is to demonstrate how to use custom overlay colors without calling mapcolor (which is illegal in mixed model applications). The last Pipeline's gloverlay.c program was in error due to this fact.

The key functions are:

`normal_cmap_init` — create custom normal-buffer colormap for gl widget

`normal_cmap_set` — map a color for the normal-buffer at the "high end"

`overlay_cmap_init` — create custom overlay-buffer colormap for gl widget

`overlay_cmap_set` — map a color for the overlay-buffer at the "high end"

The routines have been broken up this way so that the custom-overlay-buffer operations can be removed and the custom-normal-buffer functionality remains intact, and vice versa. Also, the routines which set colors in the respective maps may wish to implement different policies for locating "a good place to put the color" (thus they are implemented separately even though much of their code is the same).

```
/** header: cmapov.c ************************************************************/

/*
 * Mixed Model program demonstrating
 *      ...using custom colormaps for the normal and overlay buffers.
 *      ...moving things with the mouse.
 *
 * compiling:
 *      cc -float -prototypes -O cmapov.c -o cmapov \
 *          -s -lXirisw -lXm_s -lXt_s -lgl_s -lX11_s -lm -lc_s -lPW
 *
 * operating:
 *      Use the left mouse button to move the red, green, and blue blocks.
 *      Verify that they "pass under" the yellow, magenta, and cyan blocks
 *      which are in the overlay planes.
 *
 * programming:
 *      Brett Bainter, 92.08.20
 *
 */

/** notes ***********************************************************************/

/*
 * bugs:
 *      There is a known bug with 8 bit PI's that will cause this program to
 *      run in false colors.  It will initially come up correctly but when
 *      the cursor moves to a gl window (for example, showmap), the colormap
 *      will not be reloaded when the cursor returns to the cmapov window.
 */

/** includes ********************************************************************/

#include <stdio.h>                      /* standard */
```

(continued on page 14)

Q&A

(continued from page 8)

for the local system.

This will give you a very rough estimate of the speed of the "loopback driver". You'll find it's faster than any physical driver on the IRIS, and in some cases, far faster than any network media such as FDDI, ethernet or token ring.

Q: I have a routine which uses an array 'x' (in a FORTRAN common block), and a scalar variable named 'x_'. Using dbx, 'x' is recognized as a scalar rather than an array. dbx does not recognize 'x_' at all.

The compiler does distinguish between the two cases as illustrated in this example:

```
integer x(2), x_
x(1) = 10
x(2) = 10
x_ = 20
print *, x, x_
end
```

which outputs:

```
     10          10          20
```

What is wrong?

A: dbx knows one is debugging a FORTRAN program because of the ".f" extension on the source file name. Since this is a FORTRAN program dbx will append an "_" to all names.

For example:

```
%dbx a.out
dbx version 2.10 11/15/91 2:02
Type 'help' for help.
Reading symbolic information of `a.out' . . .
MAIN:2    2  x(1) = 10
(dbx) w
     1          integer x(2), x_
     2          x(1) = 10
     3          x(2) = 10
     4          x_ = 20
     5          print *, x, x_
     6          end
(dbx) stop at 5
Process     0: [2] stop at "/tmp/sue.f":5
(dbx) run
Process  5682 (a.out) started
[2] Process  5682 (a.out) stopped at [MAIN:5
,0x400274]
   5  print *, x, x_
(dbx) print x
20
```

dbx appends an underscore making 'x' now become 'x_' and then looks for the symbol in the symbol table. If found the second variable 'x_' which was set to 20.

```
(dbx) print x_
"x_" is not defined.
```

Likewise, dbx appended an underscore to 'x_' and it could not find 'x__' in the symbol table.

This is a dbx problem.

Q: What is the capacity of the DAT tape drive on an indigo? I can't get over 100mb of data on a 60 meter DAT tape (using bru).

A: The 60 meter DAT tapes hold about 1.2GB. You tend to significantly lose capacity if you can't give it data fast enough. You also lose some capacity with different formats. For example, bru uses more tape than tar for overhead and error checking.

If you can only store 100MB of data, however, something else is wrong.

Your first diagnostic step is to measure the capacity of a scratch tape of the same type. Use a generic system tool 'dd' to do this:

The command:

```
dd if=/dev/zero of=/dev/tape bs=1024k
```

will copy blocks of zeros to the tape. When done, 'dd' will report the number of megabyte chunks (1024 * k where k= 1024 or (2^10) residing on the tape.

An answer such as:

```
100+1 in
100+1 out
```

translates to 100 megabytes: And 1024 megabytes is a gigabyte (1073741824). A gigabye takes about two hours to transfer. An example of the output is:

```
$ timex dd bs=1024k if=/dev/zero of=/dev/tape
dd: write error: No space left on device
1254+0 records in
1254+0 records out
real  2:00:22.10
user        0.05
sys        25.63
```

This translates to a little over 1.2 gigabytes.

If you can only store 100 MB on a tape, the problem is most likely a side effect of */etc/brutab*. 100MB sounds like bru thinks that this is a QIC-150 or QIC-120 tape, as opposed to a DAT tape.

The reason for */etc/brutab* is that for reliable tape archives, a backup tool must close the tape prior its physical end. Recovery from an "end of media error" is fragile. To make reliable backups bru plans on the end of media based on the contents of */etc/brutab* Most cartridge tape drives cannot backspace and write an end of file mark on a previous block. For this reason it is important for the application (bru) to close the tape prior to the physical end of the media.

The text file */etc/brutab* describes for bru with a regular expression (`regexp`) the various sizes of tape drives.

Note: This will be an approximation of the outcome, as different runs of the backup program will yield different results.

```
#  fragment of /etc/brutab
#
/dev/r*mt/ts0d[0-7]nr* \
        size=44032K seek=0 \
        prerr=EIO pwerr=EIO zrerr=ENOSPC
zwerr=ENOSPC frerr=ENOSPC fwerr=0 \
        wperr=EROFS norewind reopen tape advance
```

Since media length is often un-reported by tape drives (un-sensible), the operator must, in general, intervene and specify the length on the command line or by site specific bru command line options or brutab changes.

Caution: Any site-specific changes should be recorded on paper (system notebook) for easy access to backups.

It is very important to make backups in ways that are familiar to you and can easily be verified.

Q: Showcase documents that contain images can take a long time to print. What can I do to speed things up?

A: Although images inherently have a lot of data associated with them, there is one trick that can decrease the data size and therefore the overall size of the files. That is to import images, shrink them down to the size you want, and then, if you know you will not use the full-sized images again, replace the images with smaller, identical ones. This trick only works if your current method is to import large files and shrink them down once inside the Showcase file. The reason you want to swap out image files is because Showcase stores the original data. If you import an image that is 500 by 500 pixels, and then shrink it down to 100 by 100 pixels for use in your document, Showcase will keep the data for the 500 by 500 pixel image. The data is saved incase the images need to be stretched back to their full size later on. If you do not plan on stretching the images, use smaller images. The Showcase files will be smaller, and printing will take less time.

The quality of the printed image will suffer a little. The degradation of the image will depend on the resolution on your printer.

To replace images within Showcase do the following:

-Click on Gizmos

-Open the Image Gizmo

-Use the Image Gizmo to take a screen snapshot of the image in your document

-Delete the original image in your document

-Replace it with the image you just took a snapshot of

Q: Running the `osview` command, in the 'CPU Usage' display, when the 'gfxf' is greater than 0, does the waiting consume cpu time, or is it descheduled, freeing the cpu to context switch to handle non-graphics process?

(continued on page 27)

(continued from page 11)

```
#include <Xm/Xm.h>                  /* for motif */
#include <Xm/Form.h>                /* motif widget */
#include <Xm/Frame.h>               /* motif widget */
#include <Xm/PushB.h>               /* motif widget */
#include <Xm/RowColumn.h>           /* motif widget */
#include <Xm/Separator.h>           /* motif widget */
#include <X11/Xirisw/GlxMDraw.h>    /* gl widget */

/** defines ***************************************************************/
/** typedefs **************************************************************/
/** prototypes ************************************************************/

extern void main(int argc, char *argv[], char *envp[]);

/* setup */
static void  install_colormaps(Widget top_level, Widget glw);
static void  normal_cmap_init(Widget glw);
static Pixel normal_cmap_set(Widget glw, int index, short r, short g, short b);
static void  overlay_cmap_init(Widget glw);
static Pixel overlay_cmap_set(Widget glw, int index, short r, short g, short b);

/* mixed model support */
static void cb_gl_expose(Widget w, XtPointer client_data, XtPointer call_data);
static void cb_gl_resize(Widget w, XtPointer client_data, XtPointer call_data);
static void cb_gl_ginit(Widget w, XtPointer client_data, XtPointer call_data);
static void cb_gl_input(Widget w, XtPointer client_data, XtPointer call_data);
static void cb_gl_overlay_expose(
    Widget w, XtPointer client_data, XtPointer call_data
);

/* callbacks (misc) */
static void cb_quit(Widget w, XtPointer client_data, XtPointer call_data);

/* drawing */
static void draw_normal_frame(void);
static void draw_overlay_frame(void);
static void draw_boxes(int c1, int c2, int c3);

/** variables *************************************************************/

/* mixed-model configuration */
static GLXconfig glx_config[] = {
    {GLX_NORMAL, GLX_DOUBLE, TRUE},
    {GLX_OVERLAY, GLX_BUFSIZE, 2},
    { 0, 0, 0 },
};

/* information which allows us to use the overlay or popup buffer */
static struct {
    char *use;
    char *expose_cb;
    char *window;
```

```
    char *visual;
    char *colormap;
} *over_res, over_res_map[2] = {
    /* describe needed overlay resources */
    {   GlxNuseOverlay, GlxNoverlayExposeCallback, GlxNoverlayWindow,
        GlxNoverlayVisual, GlxNoverlayColormap
    },
    /* describe analogous popup resources for when overlays aren't there */
    {   GlxNusePopup, GlxNpopupExposeCallback, GlxNpopupWindow,
        GlxNpopupVisual, GlxNpopupColormap
    },
};

/* normal buffer colors */
static Pixel n_grey, n_red, n_green, n_blue;

/* overlay buffer colors */
static Pixel o_trans, o_yellow, o_magenta, o_cyan;

/* gl window info */
static struct {
    Dimension width;          /* in pixels */
    Dimension height;         /* in pixels */
    float pt[3];              /* world position of moving object */
} glwin = {400, 400, {15.0, 20.0, 0.0}};

/** functions ****************************************************************/

/*
 * main - program entry point.
 */
void main(int argc, char *argv[], char *envp[])
{
    XtAppContext app_context;       /* application context */
    Widget app_shell;               /* first widget */
    Widget form;                    /* surrounds others */
    Widget rowcol;                  /* manages input buttons */
    Widget button;                  /* quit button */
    Widget separator;               /* between input and output */
    Widget frame;                   /* to surround gl widget */
    Widget glw;                     /* the gl widget inside window */
    Arg args[15];                   /* for name/value pairs */
    int n;                          /* for reusable indices */

    /* perform capabilities check */
    /* use popup planes if there is not enough overlay planes */
    over_res = &over_res_map[0];
    if (getgdesc(GD_BITS_OVER_SNG_CMODE) < 2) {
        glx_config[1].buffer = GLX_POPUP;
        over_res = &over_res_map[1];
    }
    printf("\nUsing the %s planes\n",
        over_res==over_res_map? "OVERLAY" : "POPUP"
```

(continued on next page)

(continued from previous page)

```
);

/* initialize toolkit, creating application shell */
n = 0;
XtSetArg(args[n], XmNtitle, "CMode Overlay"); n++;
app_shell = XtAppInitialize(
    &app_context, "Cmapov", NULL, 0, &argc, argv, NULL, args, n
);

/* create container for app */
n = 0;
form = XmCreateForm(app_shell, "form", args, n);
XtManageChild(form);

/* create the command area */
n = 0;
XtSetArg(args[n], XmNleftAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNorientation, XmVERTICAL); n++;
rowcol = XmCreateRowColumn(form, "rowcol", args, n);
XtManageChild(rowcol);

/* create the command area buttons */
n = 0;
button = XmCreatePushButton(rowcol, "Quit", args, n);
XtAddCallback(button, XmNactivateCallback, cb_quit, NULL);
XtManageChild(button);

/* create separator between command area and output area */
n = 0;
XtSetArg(args[n], XmNleftAttachment, XmATTACH_WIDGET); n++;
XtSetArg(args[n], XmNleftWidget, rowcol); n++;
XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNorientation, XmVERTICAL); n++;
separator = XmCreateSeparator(form, "separator", args, n);
XtManageChild(separator);

/* create the output area */
/* create the frame */
n = 0;
XtSetArg(args[n], XmNleftAttachment, XmATTACH_WIDGET); n++;
XtSetArg(args[n], XmNleftWidget, separator); n++;
XtSetArg(args[n], XmNleftOffset, 5); n++;
XtSetArg(args[n], XmNrightAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNrightOffset, 5); n++;
XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNbottomOffset, 5); n++;
XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNtopOffset, 5); n++;
XtSetArg(args[n], XmNshadowThickness, 6); n++;
```

```
    frame = XmCreateFrame(form, "frame", args, n);
    XtManageChild(frame);

    /* create the gl widget */
    n = 0;
    XtSetArg(args[n], GlxNglxConfig, glx_config); n++;
    XtSetArg(args[n], over_res->use, True); n++;
    XtSetArg(args[n], XmNborderWidth, 0); n++;
    XtSetArg(args[n], XmNwidth, glwin.width); n++;
    XtSetArg(args[n], XmNheight, glwin.height); n++;
    glw = GlxCreateMDraw(frame, "glw", args, n);
    XtManageChild(glw);
    XtAddCallback(glw, GlxNexposeCallback, cb_gl_expose, 0);
    XtAddCallback(glw, GlxNresizeCallback, cb_gl_resize, 0);
    XtAddCallback(glw, GlxNginitCallback, cb_gl_ginit, 0);
    XtAddCallback(glw, GlxNinputCallback, cb_gl_input, 0);
    XtAddCallback(glw, over_res->expose_cb, cb_gl_overlay_expose, 0);

    /* setup custom normal colormap */
    normal_cmap_init(glw);
    n_grey  = normal_cmap_set(glw, 0, 125, 125, 125);
    n_red   = normal_cmap_set(glw, 1, 255,   0,   0);
    n_green = normal_cmap_set(glw, 2,   0, 255,   0);
    n_blue  = normal_cmap_set(glw, 3,   0,   0, 255);

    /* setup custom overlay colormap */
    overlay_cmap_init(glw);
    o_trans   = 0;    /* transparent is always zero */
    o_yellow  = overlay_cmap_set(glw, 0, 255, 255,   0);
    o_magenta = overlay_cmap_set(glw, 1, 255,   0, 255);
    o_cyan    = overlay_cmap_set(glw, 2,   0, 255, 255);

    /* realize the app, creating the actual x windows */
    XtRealizeWidget(app_shell);

    /* setup for colormap installation */
    install_colormaps(app_shell, glw);

    /* enter the event loop */
    XtAppMainLoop(app_context);
}

/*- support: setup ------------------------------------------------------*/
/*
 * install_colormaps - let the window manager know about our colormaps.
 *
 * This has been generalized to handle any windows a gl widget might have.
 * It may not necessarily being using any of them.
 */
static void install_colormaps(Widget top_level, Widget glw)
{
    Window overlay_win, popup_win, underlay_win;
```

(continued on next page)

(continued from previous page)

```
    Window window[5];
    int i;

    XtVaGetValues(
        glw,
        GlxNoverlayWindow, &overlay_win,
        GlxNpopupWindow, &popup_win,
        GlxNunderlayWindow, &underlay_win,
        NULL
    );
    i = 0;
    if (overlay_win)
        window[i++] = overlay_win;
    if (popup_win)
        window[i++] = popup_win;
    if (underlay_win)
        window[i++] = underlay_win;
    window[i++] = XtWindow(glw);
    window[i++] = XtWindow(top_level);
    XSetWMColormapWindows(XtDisplay(top_level), XtWindow(top_level), window, i);
}


/*- support: custom normal colormap ----------------------------------------*/
/*
 * normal_cmap_init - create a new normal colormap for the gl widget.
 *
 * The gl widget must already be created prior to calling this function,
 * however the gl widget does not need to be realized for it to work.  This
 * is because the window it uses in creating the colormap is the root window
 * on the same screen.
 */
static void normal_cmap_init(Widget glw)
{
    Display *display;
    Window window;
    XVisualInfo *visinfo;
    Colormap pmap;
    Colormap cmap;
    XColor *color;
    int ncolors;
    int i;

    /* get display; any window on the same screen; and the visual */
    display = XtDisplay(glw);
    window = RootWindowOfScreen(XtScreen(glw));
    XtVaGetValues(glw, XmNvisual, &visinfo, NULL);

    /* create new normal colormap, allocating all entries */
    cmap = XCreateColormap(display, window, visinfo->visual, AllocAll);

    /* set new normal colormap for the gl widget */
```

```
    XtVaSetValues(glw, XmNcolormap, cmap, NULL);

    /*
     * duplicate the parent's default colors for the lower colormap entries
     * (max 256) to avoid colormap flashing on machines with only one h/w
     * colormap.
     */
    XtVaGetValues(XtParent(glw), XmNcolormap, &pmap, NULL);
    ncolors = visinfo->colormap_size;
    printf("\nnormal colors = %d\n", ncolors);
    if (ncolors > 256)
        ncolors = 256;
    color = (XColor *) XtMalloc(ncolors*sizeof(XColor));
    for (i=0; i<ncolors; i++)
        color[i].pixel = i;
    XQueryColors(display, pmap, color, ncolors);
    XStoreColors(display, cmap, color, ncolors);
    XtFree(color);
}


/*
 * normal_cmap_set - map a color for the normal buffer.
 *
 * This uses a simple scheme of mapping the colors backwards from the highest
 * colormap index.
 */
static Pixel normal_cmap_set(Widget glw, int index, short r, short g, short b)
{
    XVisualInfo *visinfo;
    Colormap cmap;
    XColor color;
    int n_last;

    XtVaGetValues(glw, XmNvisual, &visinfo, XmNcolormap, &cmap, NULL);
    n_last = visinfo->colormap_size-1;
    color.pixel = n_last - index; /* work backwards from the last position */
    color.flags = DoRed | DoGreen | DoBlue;
    color.red   = r << 8;
    color.green = g << 8;
    color.blue  = b << 8;
    XStoreColor(XtDisplay(glw), cmap, &color);
    return (color.pixel);
}


/*- support: custom overlay colormap ----------------------------------*/
/*
 * overlay_cmap_init - create a new overlay colormap for the gl widget.
 *
 * The gl widget must already be created prior to calling this function,
 * however the gl widget does not need to be realized for it to work.  This
```

(continued on next page)

(continued from previous page)

```
 * is because the window it uses in creating the colormap is the root window
 * on the same screen.
 */
static void overlay_cmap_init(Widget glw)
{
    Display *display;
    Window window;
    XVisualInfo *visinfo;
    Colormap cmap;
    XColor color;
    int ncolors;
    Pixel *pixel;
    unsigned long plane_mask[1];
    int result;

    /* get display; any window on the same screen; and the visual */
    display = XtDisplay(glw);
    window = RootWindowOfScreen(XtScreen(glw));
    XtVaGetValues(glw, over_res->visual, &visinfo, NULL);

    /*
     * create new overlay colormap, allocating no entries.
     * (AllocAll would fail here because index 0 is reserved for transparency)
     */
    cmap = XCreateColormap(display, window, visinfo->visual, AllocNone);

    /* set new overlay colormap for the gl widget */
    XtVaSetValues(glw, over_res->colormap, cmap, NULL);

    /* allocate every color except transparency as read/write */
    ncolors = visinfo->colormap_size;       /* including transparent color */
    printf("\noverlay colors = %d\n", ncolors);
    pixel = (Pixel *) XtMalloc(ncolors*sizeof(Pixel));    /* stub array */
    result = XAllocColorCells(
        display, cmap, True, plane_mask, 0,
        &pixel[1], ncolors-1 /* one less due to transparency */
    );
    XtFree((char *) pixel);

    /* check for booboo */
    if (result == 0)
        fprintf(stderr, "XAllocColorCells failed for overlay buffer.\n");
}


/*
 * overlay_cmap_set - map a color for the overlay buffer.
 *
 * This uses a simple scheme of mapping the colors backwards from the highest
 * colormap index.
 */
static Pixel overlay_cmap_set(Widget glw, int index, short r, short g, short b)
```

```
{
    XVisualInfo *visinfo;
    Colormap cmap;
    XColor color;
    int n_last;

    XtVaGetValues(
        glw, over_res->visual, &visinfo, over_res->colormap, &cmap, NULL
    );
    n_last = visinfo->colormap_size-1;
    color.pixel = n_last - index; /* work backwards from the last position */
    color.flags = DoRed | DoGreen | DoBlue;
    color.red   = r << 8;
    color.green = g << 8;
    color.blue  = b << 8;
    XStoreColor(XtDisplay(glw), cmap, &color);
    return (color.pixel);
}


/*- support: callbacks (gl widget) ---------------------------------------*/
/*
 * cb_gl_expose - handle expose events for the gl widget.
 */
static void cb_gl_expose(Widget w, XtPointer client_data, XtPointer call_data)
{
    GlxDrawCallbackStruct *glx = (GlxDrawCallbackStruct *) call_data;

    GLXwinset(XtDisplay(w), XtWindow(w));
    draw_normal_frame();
}


/*
 * cb_gl_resize - handle resize events for the gl widget.
 */
static void cb_gl_resize(Widget w, XtPointer client_data, XtPointer call_data)
{
    GlxDrawCallbackStruct *glx = (GlxDrawCallbackStruct *) call_data;
    Window overlay_window;

    /* squirrel away size */
    glwin.width = glx->width;
    glwin.height = glx->height;

    /* setup normal buffer viewport */
    GLXwinset(XtDisplay(w), XtWindow(w));
    viewport(0, glx->width-1, 0, glx->height-1);

    /* setup overlay buffer viewport */
    XtVaGetValues(w, over_res->window, &overlay_window, NULL);
    GLXwinset(XtDisplay(w), overlay_window);
```

(continued on next page)

(continued from previous page)

```
    viewport(0, glx->width-1, 0, glx->height-1);
}


/*
 * cb_gl_ginit - perform any necessary graphics initialization.
 */
static void cb_gl_ginit(Widget w, XtPointer client_data, XtPointer call_data)
{
    GlxDrawCallbackStruct *glx = (GlxDrawCallbackStruct *) call_data;

    GLXwinset(XtDisplay(w), XtWindow(w));
    mmode(MVIEWING);
    ortho2(-0.5, 100.5, -0.5, 100.5);
    gflush();
}

/*
 * cb_gl_input - handle input for the gl window.
 */
static void cb_gl_input(Widget w, XtPointer client_data, XtPointer call_data)
{
    static Boolean active = False;  /* currently moving? */
    static float dx, dy;            /* offset from current position */
    /**/
    GlxDrawCallbackStruct *glx = (GlxDrawCallbackStruct *) call_data;
    XEvent *event = glx->event;     /* what occured */
    int msx, msy;                   /* gl window mouse position */
    float mwx, mwy;                 /* gl world  mouse position */

    GLXwinset(XtDisplay(w), XtWindow(w));

    /* map to gl window coords */
    msx = event->xbutton.x;                         /* same x */
    msy = (glwin.height-1) - event->xbutton.y;      /* flip y */

    /* map to gl world coords */
    mwx = 0.0 + ((msx - 0) / (float)glwin.width ) * 100.0;
    mwy = 0.0 + ((msy - 0) / (float)glwin.height) * 100.0;

    /* process event */
    switch (event->type) {
    case ButtonPress:
        if (event->xbutton.button == Button1) {
            /* compute delta from current position */
            dx = mwx - glwin.pt[0];
            dy = mwy - glwin.pt[1];
            active = True;
        }
        break;
    case MotionNotify:
        if (active) {
```

```
            /* compute new position and draw */
            glwin.pt[0] = mwx - dx;
            glwin.pt[1] = mwy - dy;
            draw_normal_frame();
        }
        break;
    case ButtonRelease:
        if (event->xbutton.button == Button1) {
            /* we're done */
            active = False;
        }
    break;
    }
}


/*
 * cb_gl_overlay_expose - handle overlay expose events for the gl widget.
 */
static void cb_gl_overlay_expose(
    Widget w, XtPointer client_data, XtPointer call_data
)
{
    GlxDrawCallbackStruct *glx = (GlxDrawCallbackStruct *) call_data;

    GLXwinset(XtDisplay(w), glx->window);
    draw_overlay_frame();
}


/*- support: callbacks (misc) ---------------------------------------------*/
/*
 * cb_quit - exit application.
 */
static void cb_quit(Widget w, XtPointer client_data, XtPointer call_data)
{
    exit(0);
}


/*- support: drawing ------------------------------------------------------*/
/*
 * draw_normal_frame - render objects in the normal buffer and swap.
 */
static void draw_normal_frame(void)
{
    color(n_grey);
    clear();
    pushmatrix();
        translate(glwin.pt[0], glwin.pt[1], glwin.pt[2]);
        draw_boxes(n_red, n_green, n_blue);
    popmatrix();
```

(continued on next page)

(continued from previous page)

```
    swapbuffers();
    gflush();
}


/*
 * draw_overlay_frame - render objects in the overlay buffer.
 */
static void draw_overlay_frame(void)
{
    color(o_trans);
    clear();
    pushmatrix();
        translate(15.0, 60.0, 0.0);
        draw_boxes(o_yellow, o_cyan, o_magenta);
    popmatrix();
    gflush();
}


/*
 * draw_boxes - draw three boxes in three different colors.
 */
static void draw_boxes(int c1, int c2, int c3)
{
    static float vert[][2] = {      /* a box */
        { 0.0,  0.0},
        {20.0,  0.0},
        {20.0, 20.0},
        { 0.0, 20.0},
    };

    pushmatrix();
    color(c1);
    bgnpolygon();
        v2f(vert[0]); v2f(vert[1]); v2f(vert[2]); v2f(vert[3]);
    endpolygon();
    translate(25.0, 0.0, 0.0);
    color(c2);
    bgnpolygon();
        v2f(vert[0]); v2f(vert[1]); v2f(vert[2]); v2f(vert[3]);
    endpolygon();
    translate(25.0, 0.0, 0.0);
    color(c3);
    bgnpolygon();
        v2f(vert[0]); v2f(vert[1]); v2f(vert[2]); v2f(vert[3]);
    endpolygon();
    popmatrix();
}

/** eof ***************************************************************/
```

SiliconGraphics

Customer Support Guide

Customer Center: (800) 800-4SGI

For the following support services:

- Telephone Support
- On-Site Support
- Parts Support
- Customer Education

***IRIS Universe*: (415) 390-1278**

FAX Numbers

Contract Administration: **(415) 967-8544**
Customer Education: **(415) 965-2309**
Logistics: **(415) 960-2883** (in the U.S.)
(416) 674-5911 (in Canada)
Technical Assistance Center: **(415) 961-6502**

Contract Administration (areas):

Eastern Area: **(415) 390-5210**
Central Area: **(415) 390-1622**
Western Area: **(415) 390-3577**
Canada: **(416) 674-5300**

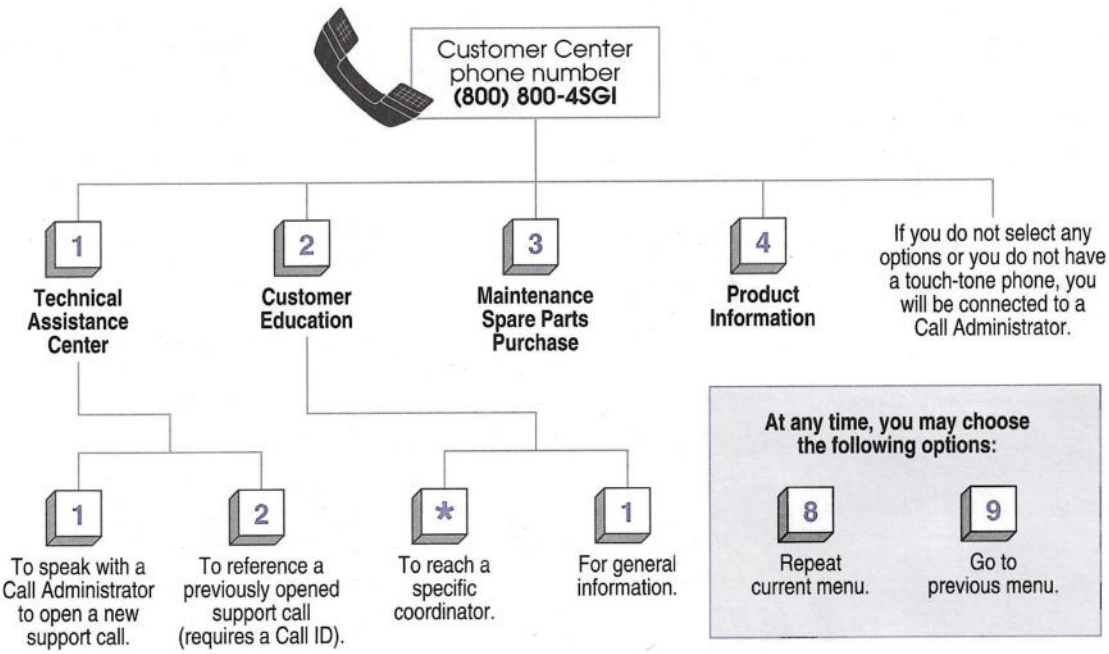
Area Offices:

(Area offices handle support contract renewals.)
Eastern: **(508) 562-4800**
Central: **(313) 478-5446**
Western: **(415) 960-1940**
Canada: **(416) 674-5300**

Corporate Offices: (415) 960-1980

If you have a request that is not addressed by any of your other options.

Reference Guide to Silicon Graphics' Automated Call Distribution system



Stuck for a solution? Turn to Silicon Graphics.

- Bundled Value
- Convenience
- Partnership
- Support Services
- Support Programs
- Training

Tap into the Silicon Graphics support system — services and programs that complement your Silicon Graphics hardware and software. Note the services your support agreement provides. *Pipeline subscriptions are bundled with support programs as indicated. The newsletter is not available separately.*

Support_Services

Silicon Graphics provides support services to assist you with installing, maintaining and using your workstation. The services described below are available on a time and materials basis or bundled in the support programs shown on this page.

Telephone Support
A Silicon Graphics engineer provides technical assistance via the telephone. The Telephone Support staff consists of experts in UNIX, hardware, communications, graphics and programming languages.

Parts Exchange
Parts are provided to repair a hardware failure in your system. The faulty part is returned to Silicon Graphics. This process is provided transparently with On-Site Support coverage.

On-Site Support
A Silicon Graphics engineer is dispatched to your site if your system fails. The engineer troubleshoots the problem and repairs the system, replacing any necessary parts.

Software Updates
New releases of operating system software are provided as they occur. The updates are available on storage media or as right-to-use licenses.

Pipeline Newsletter
The bi-monthly newsletter contains technical articles and answers to common computing questions. A subscription is provided for the length of the support agreement. Subscriptions are only available bundled with support programs.

Full_Support

Telephone Support · On-Site Support · Parts Exchange · Software Updates · *Pipeline* Newsletter

Also includes:

Set-Up/Installation
Set-up for systems along with installation of upgrades and add-ons.

Operational Instruction
Up to two hours of operational instruction provided at system set-up.

Basic_Support

Telephone Support · Parts Exchange · Software Updates · *Pipeline* Newsletter

Notifier

!

To obtain more information about the support services or programs listed here, call your local sales office or call SGI Express at

(800) 800-SGI1

Silicon Graphics Customer Support Division

S/W_Maintenance_Support

Telephone Support · Software Updates · *Pipeline* Newsletter

Telephone Support provided for software-related questions only.

S/W_Subscription_Service

Software Updates · *Pipeline* Newsletter

SGI_Logo

Indigo_Parts_Care

Parts Exchange

Q&A

(continued from page 13)

A: When the gfx FIFO is full, the CPU is spinning, waiting for it to drain to a less than "full" state. Initially this happens at high spl (processor interrupt level), then it drops to allow most other interrupts to enter. On RealityEngine, instead of going low spl, we deschedule until a low-water interrupt occurs.

The reason gfx seems so abusive is that the FIFO is full for such short periods of time that in nearly all cases instantly descheduling would kill graphics performance.

Q: I have recently begun receiving the following error on our 320 and Indigo (running IRIX 4.0.1):

```
% rsh ibm
ibm: ibm: cannot open
% rsh gumby
gumby: gumby: cannot open
```

Why is this happening and how do I avoid it in the future?

A: There are two programs named `rsh`. One is the remote shell located in */usr/bsd* (this is the one you want). The other is a restricted shell located in */bin* (this is the one you're running). The PATH environment variable controls which one is selected. It is a list of directories to search for commands. To see your path, type `echo $PATH`. To see which rsh would be selected type `which` rsh. To fix your problem, you can either explictly run the "correct" rsh by typing */usr/bsd/rsh* instead of `rsh`, or you can change your PATH variable, putting */usr/bsd* before */bin*. in your `login file`.

SiliconGraphics

Pipeline

Editor: Susan Berning

Contributors: Dave Anderson, Rick Avila, Ivan Bach, Brett Bainter, Nina Cabello, Tom Davis, Al Fohrman, Dave Frederick, Ivan Hajadi, Brian McClendon, Martin McDonald, Dave Olson, Joyce Richards, Calvin Vu, Joe Yetter

Published bi-monthly by the Customer Support Division of Silicon Graphics, Inc. for customers with support agreements. Copyright © 1992, Silicon Graphics, Inc.

The *Pipeline* welcomes your comments. Please contact us with any questions or suggestions for the newsletter. Call a telephone number at right for subscription inquiries.* Send article and Q&A topics along with other correspondence to:

Silicon Graphics, Inc.
P.O. Box 7311, M/S 12-180
Mountain View, CA 94039-7311
Attn: Susan Berning

e-mail address:
pipeline@sgi.com

***Please have the serial number of your system available when making inquiries.**

In the U.S. and Canada, to subscribe to *Pipeline*, or change your current mailing address, for technical assistance and other support services, and, to reach Customer Education, call the Silicon Graphics Customer Center at: **(800) 800-4SGI**

For customer support overseas, call the Silicon Graphics office nearest you:

Australia — Brisbane
Phone: (61) 7 - 257 .11.94
Australia — Melbourne
Phones: 008 - 335 430
(61) 3 - 882.82.11 (in Melbourne)
Australia — Sydney
Phones: 008 - 251 073
(61) 2 - 879.95.00 (in Sydney)
Belgium — Brussels
Phone: (32) 2 - 675.21.10
Denmark — Copenhagen
Phone: (45) 31 - 95.00.88
Finland — Helsinki
Phone: (358) 0 - 4354.2071
France — Paris
Phone: (33) 1 - 34.65.96.85
Germany — Berlin
Phone: (49) 30 - 823.10.42
Germany — Karlsruhe
Phone: (49) 721 - 96.20.70
Germany — Köln-Niehl
Phone: (49) 221 - 71.52.40
Germany — Munich
Phone: (49) 89 - 461.08.0
Holland — De Meern
Phone: (31) 34.06 - 21.711
Hong Kong
Phone: (852) 5 - 25.72.37
Italy — Milan
Phone: (39) 2 - 57.51.01.08
Israel — Tel Aviv
Phone: (972) 3 - 49.21.91
Japan — Kawasaki
Phone: (81) 44 - 812.60.60
Latin America — Pompano Beach
Phone: (1) 305 - 785.28.62
FAX: (1) 305 - 781.95.79
Norway — Oslo
Phone: (47) 2 - 73.20.15
Singapore
Phone: (65) 776.0970
Spain — Madrid
Phone: (34) 1 - 572.03.60
Sweden — Stockholm
Phone: (46) 8 - 33.07.05
Switzerland — Lausanne/Vidy
Phone: (41) 21 - 25.94.12
Switzerland — Zurich
Phone: (41) 1 - 731.10.70
Taiwan — Taipei
Phone: (886) 2 - 393.41.88
United Kingdom — Reading
Phones: 0734 - 30.63.63 (s/w ?)
0734 - 30.60.40 (other ?)

SiliconGraphics
Computer Systems
P.O. Box 7311, M/S 12-134
Mountain View, CA 94039-7311

BULK RATE
U.S. POSTAGE
PAID
PERMIT NO. 913
SAN JOSE, CA

Address correction requested

If you have received this newsletter, but do not have a Silicon Graphics workstation, please pass it along to a member of your organization that does. Send inquiries (in the U.S. and Canada) to the address listed on the previous page. ***Send address corrections to the address above or log a call (see previous page).***

This newsletter is printed on recycled paper using soy ink.

Customer Education Training Schedules

Starting Dates

		Length	October				November					December			
			5	12	19	26	2	9	16	23	30	7	14	21	28
4D Western Education Center (Mountain View, CA)	Graphics Library Programming I	4.5 days	●								●				
	Graphics Library Programming II & PowerVision	4.5 days		●								●			
	Motif Programming	4 days							●						
	Realtime Programming	4.5 days									●				
	Parallel Programming	4.5 days						●							
	Mastering IRIX	4.5 days			●			●			●				
	System Administration	4.5 days				●			●			●			
	Advanced System Administration	4.5 days		●			●								
	Network Administration	4.5 days											●		
	IRIS Inventor	4.5 days			●										
	IRIS Explorer	4.5 days				●							●		
	System Maintenance (Power Series)	10 days				●						●			
	End User Fundamentals	2 days													
4D Eastern Education Center (Bethesda, MD)	Graphics Library Programming I	4.5 days			●										
	Graphics Library Programming II & PowerVision	4.5 days				●									
	Motif Programming	4 days											●		
	Realtime Programming	4.5 days		●											
	Parallel Programming	4.5 days													
	Mastering IRIX	4.5 days					●								
	System Administration	4.5 days									●				
	Advanced System Administration	4.5 days										●			
	Network Administration	4.5 days	●												
	System Maintenance (Power Series)	10 days						●							
4D Southern Education Center (Dallas, TX)	Graphics Library Programming I	4.5 days						●							
	Graphics Library Programming II	3 days													
	Mastering IRIX	4.5 days		●							●				
	System Administration	4.5 days			●							●			
	Network Administration	4.5 days													
	IRIS Inventor	4.5 days							●						

To register for one of these courses, or get additional information on training programs and policies, please call Customer Education at (800) 800-4SGI (U.S. and Canada) or an overseas Silicon Graphics office. *(Schedule subject to change.)*