IRIS IM Programming Guide

Document Number 007-1472-020

CONTRIBUTORS

Written by Ken Jones
Production by Lorrie Williams
Engineering contributions by Bob Blean, Susan Dahlberg, Todd Newman, Kevin Smith, and Joel Tesler

IRIS IM Programming Guide
Document Number 007-1472-020

Contents

Introduction

This guide is for all programmers interested in developing applications using IRIS IM™, Silicon Graphics' port of the industry-standard OSF/Motif™ for use on Silicon Graphics IRIS workstations.

This guide also contains advice for pure X and X toolkit programmers about programming in the Silicon Graphics® X environment, including how to work with nondefault visuals. The Silicon Graphics IRIX™ operating system contains a fully compliant native X Window System™ that provides standard X toolkits, font support, and improved X performance.

X server enhancements include support for multiple visuals, overlay windows, the Display PostScript™ Extension, the Shape Extension, the X Input Extension, the Shared Memory Extension, and simultaneous display on multiple graphics monitors. X applications can use high-performance GL rendering for 2D and 3D graphics.

Suggestions for Further Reading

For comprehensive information on the X Window System, Xlib, Xt, and X protocol, see the Digital Press X series:

- *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD*, Third Edition, X Version 11, Release 5, Scheifler, Robert W. and James Gettys, et al., Digital Press—Digital Equipment Corporation, Burlington MA, 1992. ISBN 1-55558-088-2
- *X Window System Toolkit: The Complete Programmer's Guide and Specification*, Asente, Paul J. and Ralph R. Swick, Digital Press—Digital Equipment Corporation, Burlington MA, 1992. ISBN 1-55558-051-3

Or refer to the O'Reilly X Window System Series, Volumes 1,2, 4, and 5:

- Volume One: *Xlib Programming Manual*, by Adrian Nye, published by O'Reilly & Associates, Inc., Sebastopol, California.
- Volume Two: *Xlib Reference Manual*, published by O'Reilly & Associates, Inc., Sebastopol, California.
- Volume Four: *X Toolkit Intrinsics Programming Manual*, by Adrian Nye and Tim O'Reilly, published by O'Reilly & Associates, Inc., Sebastopol, California.
- Volume Five: *X Toolkit Intrinsics Reference Manual*, published by O'Reilly & Associates, Inc., Sebastopol, California.

For detailed information about the X server and X version 11, Release 5:

- *The X Window System Server: X Version 11, Release 5*, Elias Israel and Erik Fortune, Digital Press - Digital Equipment Corporation, Bedford MA, 1992. ISBN 1-55558-096-3
- *R5 Update—Programmer's Supplement for Release 5 of the X Window System, Version 11*, David Flanagan, O'Reilly & Associates, Inc., Sebastopol CA, 1991. ISBN 0-937175-86-2

For information on OSF/Motif:

- *OSF/Motif Programmer's Guide*, *Revision 1.2*, Open Software Foundation, PTR Prentice-Hall, Inc., Englewood Cliffs NJ, 1993.
- *OSF/Motif Programmer's Reference*, *Revision 1.2*, Open Software Foundation, PTR Prentice-Hall, Inc., Englewood Cliffs NJ, 1993.
- *OSF/Motif User's Guide*, *Revision 1.2*, Open Software Foundation, PTR Prentice-Hall, Inc., Englewood Cliffs NJ, 1993. ISBN 0-13-643131-3
- Volume Six: *Motif Programming Manual*, by Dan Heller, published by O'Reilly & Associates, Inc., Sebastopol, California.

These and other books in the X Window System series and OSF/Motif series are available from bookstores. They are referred to throughout this document.

For more information on programming in OpenGL, refer to these manuals:

- *OpenGL Reference Manual*, from the OpenGL Architecture Review Board, published by Addison-Wesley Publishing Company, Reading, Massachusetts, 1992. ISBN 0-201-63276-4.
- *OpenGL Programming Guide*, written by Jackie Neider, Tom Davis, and Mason Woo, published by Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-63274-8

For more information on programming with IRIS GL, refer to these Silicon Graphics manuals:

- *Graphics Library Programming Guide*
- *Graphics Library Programming Tools and Techniques*

The Silicon Graphics book *OpenGL Porting Guide* contains information on porting IRIS GL programs to OpenGL, including information on converting GL/IRIS IM mixed-model programs.

Typographical Conventions

This guide uses the following typographical conventions:

functions()	appear in boldface font with parentheses.
arguments	appear in italic font.
file names	appear in italic font.
`code`	appears in courier font.
`<key>`	appears in boldface courier font, surrounded by angle brackets, indicating that you press the designated key on your keyboard.
`entry`	appears in boldface courier font, indicating that you enter the information from your keyboard.

Chapter 1

X Window System and IRIS IM Programming Guidelines

This chapter outlines the some of the facilities available to the developer in the Silicon Graphics X environment. Topics discussed in this chapter include:

- building applications with X toolkits
- using X visuals and nondefault X visuals
- using colormaps

The Silicon Graphics native X Window System is a full implementation of the X Version 11 Release 5 (X11R5) standard that supports multiple visuals, overlay windows, the X Input Extension, the Display PostScript Extension, the Shape Extension, and the Shared Memory Extension.

The X Window System is a portable, network-based windowing system. The portability provided by X allows you to create an application that is capable of running on many different types of workstations. You can compile and execute portable X application code in IRIX without modification. Because X is network transparent, applications can be used in a heterogeneous environment, running and displaying on workstations from different vendors. X also allows simultaneous display on multiple graphics monitors.

You'll find most of the information you need to develop X-based applications in the references listed under "Suggestions for Further Reading" in the Introduction to this guide. The programming notes in this guide are a supplement to those references, detailing the features available in the Silicon Graphics X environment.

Programming with IRIS IM

IRIS IM interface maker is Silicon Graphics' port of the industry standard, OSF/Motif, for use on Silicon Graphics IRIS workstations.

Silicon Graphics also supplies some related software that is not part of standard OSF/Motif, including:

- Widget support for integrating toolkits with the IRIS Graphics Library.
- *4Dwm*, an enhanced version of *mwm*.
- Modified copies of many of the OSF/Motif demonstration programs, which show how IRIS IM programs can use nondefault visuals such as the popup and overlay planes.

  The techniques used in these sample programs are useful with any nondefault visual. These demos use `#ifdef OVERLAY_DEMO` statements so you can compile them without using the Silicon Graphics modifications.
- An added "Quit..." option for the *mwm* root menu to avoid an ICCCM shortcoming. This was added to both the built-in root menu and the default root menu in */usr/lib/X11/system.mwmrc*.
- Several bug fixes implemented by Silicon Graphics.

This section contains information about programming IRIS IM on an IRIS workstation. See the release notes for release-specific information regarding IRIS IM, such as product codes, sizes, version numbers, and bugs.

Topics discussed in section are:

- the advantages of using shared libraries
- specifics about building a toolkit-based program
- virtual key bindings
- default translation issues
- rendering from the IRIS Graphics Library using the GlxMDraw widget
- using nonstandard visuals that let you take advantage of the special framebuffer architecture available on Silicon Graphics workstations

Using Shared Libraries to Maximize System Resources

Generally, programs should be linked with the shared libraries. Shared libraries provide several significant advantages:

- Programs are significantly smaller—they take less disk space and less virtual memory.
- The smaller size leads to faster startup times.

  Note: If only one application is running with the shared library, memory usage and startup time can actually be worse; however, in the default system configuration two programs are running that make use of the IRIS IM shared library, namely *4Dwm* and *Toolchest*. The shared library is also used in *mwm*.
- A program linked with the shared library benefits from any bug fixes and changes in subsequent *compatible* versions of the shared library. While this is normally a benefit, it does mean that the application could be running in an untested configuration.

  Silicon Graphics tries to maintain shared library compatibility across releases; however, it is not always possible to maintain compatibility in new releases. In cases where a new library is not compatible with those of previous releases, Silicon Graphics tries to supply the old, compatible library. The older library is not updated nor are bugs in it fixed; you must relink your applications with the new library to get these benefits. To check whether your application needs relinking, refer to the compatibility section of the Release Notes.

The default system behavior is to link with shared libraries; however, sometimes you might not want to use shared libraries. You can do so only by specifying the full library name, such as *libXm.a*.

It is strongly recommended that programs linked with the IRIS IM library be linked with the other shared libraries. Programs linked with the unshared IRIS IM library can still be linked with other shared libraries.

Using Debugging Libraries to Troubleshoot Programs

Silicon Graphics provides a set of debugging libraries (for example, *libXm_d*, *libUil_d*, and *libMrm_d*) to help you troubleshoot IRIS IM programs. These

libraries are large, so they are not automatically installed with the standard IRIS IM development libraries. If you have sufficient disk space, you can install these libraries and link your programs with them to find out more information when you debug your programs.

Linking Your IRIS IM Programs

When using *Imakefile*s, the correct libraries are automatically loaded; however, when using *Makefile*s, the correct libraries must be specified.

To handle regular expressions, the *FileSelectionBox* and related IRIS IM code use the functions **regex()** and **regcmp()**. These functions are in *libPW*. A program linked with the unshared IRIS IM library requires *-lPW* only if the file selection box is used. A program linked with the shared IRIS IM library always requires it. Thus, converting from unshared to shared libraries may result in these undefined symbols unless *-lPW* is included.

Building IRIS IM Programs with *imake*

If you use *Imakefile*s, and you depend on the configuration files in the */usr/lib/X11/config* directory, you need to generate a new *Makefile* as follows:

1. **`cd`** to the directory containing your *Imakefile*.
2. Run *mmkmf*.

The above procedure makes a new *Makefile* with the correct definitions for building IRIS IM software so that the standard *Makefile* targets, such as **`make Makefile`** or **`make Makefiles`** work properly.

Note: *mmkmf* is similar to *xmkmf*. ♦

Programs using *Imakefile*s will automatically be linked with the shared libraries.

The *Imakefile*s loaded with the demonstration software build the demonstration programs under */usr/src/X11/motif/osf_demos*. You can examine these *Imakefile*s for an example of how to build an IRIS IM program.

Building IRIS IM Programs with Conventional Makefiles

It is entirely possible to build your IRIS IM programs using conventional *Makefile*s. For example, the *Makefile*s that build the book example programs (located under */usr/src/X11/motif/book_examples*) are traditional *Makefile*s—even if they are fairly terse; however, if you use *Makefile*s to build your IRIS IM programs, you need to maintain the *Makefile*s and make sure that all definitions and targets still work from release to release. (If you use *Imakefile*s, up-to-date definitions will be provided in the configuration files with each new release.)

If you are having problems with your *Makefile*s, look at the command lines generated by building the demo programs. Since the demo *Makefile*s were generated by *imake*, these command lines will be correct.

Handling Keyboard Issues

This section touches on a few keyboard issues that may be of interest to developers.

Virtual Key Bindings

The virtual key bindings shipped with IRIS IM match the default ones compiled into *libXm*. There is no assurance that users won't change their own bindings. Virtual key bindings are not synonyms for other keysyms. Instead, virtual key bindings *replace* other keysyms.

Numeric Keypad Behavior

The OSF virtual key binding model translates a single keysym into a virtual keysym—for example, **`<Key>Left`** is translated to **`<Key>osfLeft`**. On Silicon Graphics keyboards, the arrows on the numeric keypad generate keypad arrows rather than arrows—for example, **`<Key>KP_Left`** instead of **`<Key>Left`**. The OSF virtual binding model is unable to accommodate multiple keysyms mapping to the same virtual keysym.

To accommodate the two types of arrows without introducing an incompatible syntax, Silicon Graphics has introduced internal virtual keysyms prefixed with **`_sg_KP`** (for example, **`_sg_KP_Left`**). These are

mapped internally to the same key code as the OSF equivalent, so the net result is that both the numeric keypad and the other arrow keypad will generate the *osf* virtual bindings (for example, **osfLeft**).

Because the **_sg_KP** bindings are an internal mechanism that get translated to *osf* bindings, users should not use them directly. Bindings should not appear in translations; they may be removed from a future release.

For all IRIS IM widgets except for XmText and XmTextField, the numeric keypad is treated as pure arrow keys. For the XmText and XmTextField, the numeric keypad is treated as either arrows or numbers, depending on the state of the **<Num Lock>**, and **<Shift>** keys. If either **<Shift>** or **<Num Lock>** is on, numbers are sent, otherwise arrows are sent. It should be noted, however, that **<Shift-Num Lock-KP_4>** sends a shifted left arrow rather than an unshifted left arrow, which is treated differently by the text widget.

Translation Tables and the Numeric Keypad

For writers of translations, whether in application default files or in widgets, it is important to understand how to write a translation so the numbers and the other keysyms work on the keypad.

If a translation of the form:

```
<Key>osfLeft:        action() \n \
```

is used, the action applies to both the left arrow key and to the numeric keypad **<4>** key (because it also has a left arrow on it), whether or not the **<Shift>** key is also pressed.

To permit the numeric keypad keys to function normally, the following form is recommended:

```
:<Key>KP_4:        self-insert() \n \
<Key>osfLeft:        action() \n \
```

The leading colon on the first translation states that the action “self-insert” should be executed if keypad **<4>** is pressed in such a manner that it would normally be interpreted as a digit—that is, either **<Num Lock>** or **<Shift>** is turned on, but not both. The second translation specifies that “action” should occur if **<key>osfLeft** is pressed in any other circumstances. Note that the action for **:<key>KP_4** must come first.

So, if you are defining translations that affect the shifted keypad keys, the following translations should come first to also permit use of the digits from the numeric keypad:

```
:<Key>KP_0:        self-insert() \n \
:<Key>KP_1:        self-insert() \n \
:<Key>KP_2:        self-insert() \n \
:<Key>KP_3:        self-insert() \n \
:<Key>KP_4:        self-insert() \n \
:<Key>KP_5:        self-insert() \n \
:<Key>KP_6:        self-insert() \n \
:<Key>KP_7:        self-insert() \n \
:<Key>KP_8:        self-insert() \n \
:<Key>KP_9:        self-insert() \n \
:<Key>KP_Decimal:  self-insert() \n \\
```

Widgets for GL/X Mixed-Model Programming

Silicon Graphics provides the GLwMDrawingArea widget for IRIS IM programs that use OpenGL to draw to a window within an IRIS IM application. The GLwMDrawingArea widget is actually a special version of the GLwDrawingArea widget; GLwDrawingArea is a generic widget, suitable with any widget set that is based on the Xt intrinsics, whereas GLwMDrawingArea is designed specifically for use within an IRIS IM application. Similarly, the GlxMDraw widget allows programs to use IRIS GL to draw to a window within an IRIS IM application. The GlxMDraw widget is in turn a special version of the GlxDraw widget for drawing IRIS GL into any Xt-based program. All of these widgets are included in the *Sgm* IRIS widget library. They can be linked in by specifying *-lSgm* before *-lXm* on the link line.

Examples that use the mixed-model widget appear in the *4Dgifts* directory in *examples/GLX/glxwidget/demos*. The actual source to the widget appears in *examples/GLX/glxwidget/widget*, although it is recommended that programs link with the version provided in the *Sgm* library to ensure that they get future changes.

For information on OpenGL/IRIS IM mixed-model programming and the GLwMDrawingArea widget, consult *OpenGL Programming Guide, OpenGL Porting Guide*, and the **GlwMDrawingArea**(3X) manual page. For information on IRIS GL/IRIS IM mixed-model programming and the

GlxMDraw widget, consult *Graphics Library Programming Tools and Techniques* and the **GlxMDraw**(3X) manual page.

Using Silicon Graphics Extensions

Silicon Graphics has adopted the convention for IRIS IM that names beginning with the letters *SG*, in upper or lower case, are Silicon Graphics extensions that customers can use, but they do not guarantee portability. Names beginning with *_SG* are for internal use only—they are subject to change without warning in future releases. This intent of this policy is to allow Silicon Graphics to make extensions that:

- do not affect existing customer code
- let customers know when they are relying on Silicon Graphics extensions
- minimize the chance that unforeseen OSF changes might force Silicon Graphics to break compatibility

Such names appear in *4Dwm* resource names and in the *sgi_visual* module with the overlay demo programs.

Using the IRIS IM Sample Source Code

To study code that uses IRIS IM, look at the sample code under these directories:

/usr/src/X11/motif/osf_demos
: Contains the standard demonstration programs provided by OSF. All demos are unsupported.

/usr/src/X11/motif/overlay_demos
: Contains modified versions of many of the programs from */usr/src/X11/motif/osf_demos*. These demonstration programs use the popup and overlay planes to show how IRIS IM programs can use nondefault visuals.

/usr/src/X11/motif/book_examples
: Contains the code examples from the book *The X Window System: Programming and Applications with Xt*.

Most of the demonstration software subdirectories also contain *README* files that describe the individual demonstration programs. Be sure to read the *README* file with each demonstration program before trying to run that program.

Note: Be sure you actually read the *README* files in these directories. They contain important information on resource files. Simply typing **`make all`** is not sufficient. You need to first determine how the applications find their resource files. ♦

Using the IRIS IM Demonstration Programs

Your IRIS IM software includes source for the demonstration programs provided by OSF. Many of them are contributed software—they are not part of the core product. These programs and their *Imakefiles* are in the subdirectories of */usr/src/X11/motif/osf_demos*.

Here is important information about the demonstration programs provided by OSF:

- Size of the demonstration programs

  Installing the demonstration program sources takes about 3 megabytes. When built, they take about 20 megabytes. If you do not want to build them all at once, see the *README* file for advice on building individual applications.

- Building and running the demonstration programs

  The demonstration program source code is in subdirectories of */usr/src/X11/motif/osf_demos*. To build the demonstration programs, read the file */usr/src/X11/motif/osf_demos/README* and follow the instructions.

- No support is provided for the demonstration programs

  Although Silicon Graphics has fixed a number of bugs in these demonstration programs, Silicon Graphics does not claim that these programs are bug free. Because OSF does not support these programs, neither does Silicon Graphics. Bug reports are welcome, but fixes are not guaranteed.

Using the IRIS IM Overlay Demonstration Programs

Your IRIS IM software includes source for the overlay demonstration programs. These are slightly modified versions of the standard demonstration programs. Each source change has been clearly identified with `#ifdef OVERLAY_DEMO` statements. These programs and their *Imakefiles* are in the subdirectories of */usr/src/X11/motif/overlay_demos*.

Here is important information about the overlay demonstration programs:

- These programs use the popup and overlay planes. The techniques demonstrated are appropriate to any nondefault visual.
- Size of the overlay demonstration programs

  The sources themselves take about 1.8 megabytes when installed. When built, they take about 9.8 megabytes. If you do not want to build them all at once, see the *README* file for advice on building individual applications.
- Building and running the overlay demonstration programs

  The overlay demonstration programs are built and run just as the normal demonstration programs are. Be sure to read */usr/src/X11/motif/overlay_demos/README* for the status of individual overlay demonstration programs.
- No support is provided for the demonstration programs

  These programs are unsupported. They are derived from the (unsupported) standard demonstration software. Bug reports are welcome, but fixes are not guaranteed.

Using the Reference Book Sample Programs

Your IRIS IM software includes source for the example programs in the book *The X Window System: Programming and Applications with Xt*, *OSF/Motif Edition*, Douglas A. Young, ISBN 0-13-497074-8.

These sample programs are provided to make it easier for you to use this book to learn IRIS IM. The examples also show how to use simple *Makefile*s with IRIS IM (as contrasted with the demonstration programs that show you how to use *Imakefiles*). Before you try to run an example program, be sure to read the *README* file in its directory.

The X defaults files for the example programs are in the *AppDefaults* subdirectory of */usr/src/X11/motif/book_examples*.

Here is important information about the overlay demonstration programs:

- Size of the Example Program Sources

  Installing the example program sources takes about 0.5 megabytes. When built, they take about 72 megabytes. If you don't want to build them all at once, see the file */usr/src/X11/motif/book_examples/README* for advice on building individual applications.

- Building and Running the Example Programs

  These programs and their *Makefile*s are in the subdirectories of */usr/src/X11/motif/book_examples*. Read the top level *README* file for instructions on building and running these example programs.

- No support is provided for the demonstration programs

  Silicon Graphics does not claim that these example programs are bug free. Silicon Graphics does not support these example programs. These programs are as supplied by the book's publisher, except that Silicon Graphics has fixed a few bugs and provided some missing application default files. Bug reports are welcome, but fixes are not guaranteed.

Rendering Considerations

This section highlights issues, including framebuffers and colormaps, that are related to rendering from a software and hardware perspective. All the information necessary for drawing on the screen, such as framebuffer selection, depth, and visual class (colormap type) is contained in an X11 *visual*. See "Using X11 Visuals on Silicon Graphics Workstations" on page 16 for more information about visuals on Silicon Graphics workstations.

Hardware Configurations of Silicon Graphics Workstations

This section outlines some hardware features of Silicon Graphics workstations that you should be aware of when developing your applications.

Framebuffers on Silicon Graphics Workstations

On Silicon Graphics workstations, the display bitplanes are divided into framebuffers:

Normal	Supported on all IRIS workstations. Most applications use this framebuffer to render graphics. The number of bitplanes assigned to this framebuffer can vary.
Popup	Two popup planes are supported on all IRIS workstations. The popup planes are intended for transient drawing. For example, the window manager and toolchest use this framebuffer to display such things as popup menus without damaging the graphics rendered in other framebuffers. By system convention, drawing to these planes does not necessarily use a window. If you put anything other than transient drawing here, it may get damaged. It is reasonable to put your application's menus and modal popup dialogs here.
Overlay	0, 2, 4, or 8 overlay planes are supported on IRIS workstations. The 2-bit overlay planes are used by applications for transient things, such as popup menus and dialog boxes. Silicon Graphics discourages the use of 4-bit overlay framebuffers. The 4-bit overlay framebuffer uses the same hardware as the two-bit overlay and popup framebuffers. That means that both the pixels and the colormap of the 4-bit overlay planes conflict with those of both the 2-bit overlay planes and the popup planes.
Underlay	On any Silicon Graphics hardware that has overlay planes, the hardware supports underlay planes. The X server, however, does not support visuals for the underlay framebuffers. This means that X11 (and therefore IRIS IM) programs cannot draw in the underlay planes. Any underlay planes drawing must be done by means of the Graphics Library.

Special	Other special-purpose framebuffers include the z-buffer, the stencil planes, and the alpha planes. These framebuffers are not available in all IRIS workstations and there are no X visuals that support these framebuffers.
Configurable	Some workstations, such as RealityEngine™, allow you to configure the framebuffer to suit your application needs. This is an advanced concept that affects only RealityEngine systems and similar architectures and there are no X visuals that support this feature.

Note that only normal planes and popup planes are supported on all Silicon Graphics hardware. For more information on these framebuffers, including interactions between them, see the *Graphics Library Programming Guide*.

Visual Classes on Silicon Graphics Workstations

As part of the complete X visual, a hierarchy of six visual classes (colormap types) distinguishes between color/monochrome display, read-only/read-write colormaps, and color index/RGB color representation.

All currently available Silicon Graphics workstations support an 8-bit PseudoColor visual, some workstations support 12-bit visuals, and X servers running on workstations with sufficient hardware support a 24-bit TrueColor visual. Other visuals may be available from a particular X server, depending largely on the type of workstation and its graphics capabilities. Various-depth visuals are available in static, dynamic, color, or gray classes.

Some workstations, such as the RealityEngine, support a 24-bit DirectColor visual. A workstation that supports DirectColor can support any of the six visual classes. There is no difference in the number of displayable colors between a 24-bit TrueColor visual and a 24-bit DirectColor visual; however, with a DirectColor visual, you can redefine the colormap to include only one color and thus obtain varying intensities of that color. TrueColor displays colors at full intensity only.

GL uses display *modes* to access and properly display visual data in the bitplanes of a Silicon Graphics workstation. X visuals are mapped to GL modes when an X window is prepared for GL rendering. Consult the *OpenGL Programming Guide* and *The OpenGL Porting Guide* for information on OpenGL rendering in X windows. Consult the *Graphics Library*

Programming Tools and Techniques for information on IRIS GL rendering in X windows.

Colormaps on Silicon Graphics Workstations

Colormaps are managed by the X server. X employs a virtual colormap model. Each window can have a different virtual colormap, while GL maintains a fixed colormap notion. There are IRIS GL routines for manipulating the colormap, but you can use them only in pure IRIS GL programs—programs that contain no X routines. Some workstations allow more than one colormap to be loaded simultaneously.

Silicon Graphics supports two types of hardware. One type, such as the Personal IRIS™, installs only one colormap at a time; thus colormap switching must occur when input focus shifts among windows with different colormaps. The other type, such as the IRIS Indigo™, installs multiple colormaps concurrently—for example, an X colormap and an IRIS GL colormap can be loaded simultaneously.

On single colormap hardware, a single top-level colormap affects all windows on the screen (except for RGB, popup, and overlay windows, which are separate resources). On multiple colormap hardware, each window (up to a hardware dependent maximum) uses its own colormap. It's best to test your programs on both types of hardware. Some common mistakes might be seen on only one type of hardware.

Colormap Size Issues

The size of the colormap depends on the number of bitplanes associated with the visual. To express interest in a colormap, use **XSetWMColormapWindows()**. This lets the window manager know which colormap(s) to install when your application gets focus. These are some important points about colormap sizes:

- If you do decide to put widgets in one of the overlay visuals, remember that the colormap is probably smaller than the default one.
- The 2-bit popup planes colormap has only 4 entries. Color 0 is reserved for transparent, and the other three colors are available.

- The 2-bit overlay planes colormap has only 4 entries. Color 0 is reserved for transparent, and the other three colors are available.
- The 4-bit overlay planes colormap has 16 entries. Color 0 is reserved for transparent, and the other fifteen colors are available.
- The popup planes and the 2-bit overlay planes (if available) have independent colormaps. The colormap for the 4-bit overlay planes conflicts with both.

Variation across Colormaps

The same pixel location in different colormaps does not necessarily represent the same color. Be sure you use the correct pixel values for the colormap you are working with.

If you use a nondefault colormap, avoid color macros such as **BlackPixel()** and **WhitePixel()**. As is required by X11, these macros return pixel values that are correct for the default colormap. They will probably return inappropriate values for your application. The pixel is likely to represent a different color in your colormap, or worse yet, be out of range for it. If the pixel does not exist in your colormap (such as any pixel greater than three for a 2-bit overlay colormap), you will get an X protocol error.

You are most likely to make a "right index–wrong map" type of mistake if you use the macros **BlackPixel()** and **WhitePixel()**. For example, the **BlackPixel()** macro returns `0`, which is black in the default colormap. That value is always transparent (not black) in the popup and overlay colormaps.

Multiple Colormap Issues

The need to deal with multiple colormaps of various sizes raises new issues. Some of these issues do not have well-defined solutions.

- There is no general official way for multiple independent applications to cooperate to use a common colormap. There are no defaults, including a default colormap, for any visual other than the default visual.
- With multiple colormaps in use, there may be colormap flashing. An application is guaranteed to have all of its colormaps installed only when it has colormap focus. At that time, the window manager installs

all of the application's colormaps, regardless if they are all currently needed. These colormaps remain installed until another application needs to have one of them replaced. If another application gets colormap focus, the window manager installs those (possibly conflicting) colormaps. Thus, some widgets might be affected while other widgets remain unchanged. The window manager does not reinstall the colormaps for your application until your application has the colormap focus again.

- Remember to call **XSetWMColormapWindows()** to tell the window manager which colormaps your application wants installed when it gets colormap focus. Some of the IRIS IM overlay demonstration programs illustrate this.

  In particular, if any of your application's subwindows or override redirect windows (for example, pull-down and popup menus) use their own colormaps, remember to call **XSetWMColormapWindows()** for those windows. This instructs the window manager to install the colormaps you need whenever you enter your main window.

Using X11 Visuals on Silicon Graphics Workstations

An X *visual* contains all the information necessary for drawing on the screen, such as framebuffer selection, depth, and visual class (colormap type). Each visual allows drawing to only a single combination of framebuffer, depth, and colormap type.

The Silicon Graphics X11 server, *Xsgi*, provides various visuals that enable X11 drawing access to the framebuffers that are physically available. The server supports at least one visual for the overlay, popup, and normal framebuffers (except for Entry graphics, which does not support an overlay framebuffer). There are several visuals for the normal framebuffer. There is no visual for the underlay framebuffer.

Note: One shortcoming of the current X11 visual model is that it does not consider layering of framebuffers (that is, it does not explicitly consider overlays or underlays). Thus, any application code that deals with them is nonstandard. Silicon Graphics is continuing to work with the X11 Consortium to get this area standardized, so you can expect this area to evolve. ♦

Each X server can support multiple visuals, depending on the available hardware. Each server has a default visual that can be specified when the server starts. You can ascertain the default visual with the Xlib macro **DefaultVisual()**.

Because you can't predict the configuration of every X server, and you may not know what hardware configuration your program will be used on, it is best to find out what visual classes are available on a case-by-case basis. You can determine which visuals are supported on any given workstation either at the command line or from within an application.

To get X display information from the command line, enter **`xdpyinfo`**. Use *xdpyinfo* to confirm the visuals supported by your workstation and to return information regarding the capabilities of your X server. For more information, consult the manual page for *xdpyinfo*(1).

To get X display information from within your application, use the Xlib functions **XGetVisualInfo()** and **XMatchVisualInfo()** to determine what visuals are available on a given workstation. These functions allow you to indicate the visuals you prefer, returning a list of available visuals called an **XVisualInfo()** structure, from which you can match your preference.

Using Nondefault Visuals in the Normal Framebuffer

The need to use nondefault visuals on an IRIS workstation is not confined to putting a widget in the overlay or popup planes. Visuals define which bitplanes to use for a widget, and they also determine things such as whether you use 24-bit TrueColor or 8-bit PseudoColor, and so on.

Thus, even if your application draws only into the normal framebuffer, it might need to specify a nondefault visual—if only to account for the possibility that the user can change the default visual.

The sections that follow describe some issues related to using nondefault visuals when programming with Xlib, Xt, and IRIS IM.

Xlib Programming with Nondefault Visuals

When you call **XCreateWindow()**, the *depth* and *visual* resources must be consistent and correct. You cannot later change this information for the window. Other things can be set as needed.

You must also be consistent in how you set the *XSetWindowAttributes* structure members. Table 1-1 lists the requirements for these settings.

Table 1-1 *XSetWindowAttributes* Structure Settings for Nondefault Visuals

Structure Member	Requirement
background pixmap	Must be of the stated depth
background pixel	Doesn't exceed the colormap size
border pixmap	Must be of the stated depth
border pixel	Doesn't exceed the colormap size
colormap	Must match the visual
backing store	Information must be consistent with the visual and depth

If an application sets these values inconsistently, or if it allows a widget to inherit an inconsistent value, the result is an error returned by the X server.

Xt Toolkit Programming with Nondefault Visuals

The major concerns when using the nondefault visuals with Xt Toolkit intrinsics are setting visual and depth resources consistently and expressing interest in any special colormaps you want to associate with the nondefault visual.

Setting Consistent Resources for Nondefault Visuals

visual is a Shell widget resource. A Shell widget's *visual* resource is inherited by all widget descendants of that Shell widget (except for other Shell widgets), because their windows are descendants of the Shell's window.

colormap and *depth* are Core widget resources. They must always be consistent with the current visual setting. Generally, whenever you set *visual*, you also need to set *colormap* and *depth*.

Anything else that has an associated depth also needs to be consistent with the *depth* resource. The most common such consistency problem is using a pixmap at a depth different than the depth for which it was created.

Note: Use the new IRIS IM function **XmGetPixmapByDepth()** instead of **XmGetPixmap()**. ♦

Resources Cannot Control Putting X Toolkit Components in Overlay Planes

When you put X toolkit components (for example, popup menus) into the overlay or popup planes, you must do this programmatically rather than from a resource file. The reasons are:

- To put a widget tree into the overlay or popup planes, you must put the Shell for the widget tree into the overlay or popup planes. To put your entire application in the overlay or popup planes, you must put your top-level Shell in the overlay or popup planes.

  As useful as it is, you cannot use **XtAppInitialize()** in this case. To put a Shell into the overlay or popup planes, you first need to know the display, which you cannot know before calling **XtAppInitialize()**. Instead, you must use the older piece-meal approach. Then you can determine the display before you need to create the top-level Shell. (There are demos done each way for you to look at.)

- There is no resource converter for the shell.visual resource, meaning that you can set this resource only programmatically.

  It also means that the **varargs** calls such as **XtVaCreate()** and **XmVaCreate()** cannot use an XtVaTypedArg parameter to put a Shell in a nondefault visual.

- There is no Xt resource that you can use to distinguish between the types of framebuffer (for example, there is no visualType: overlay).

Inheriting and Using Nondefault Visuals

The Shell widget inherits all of the visual attributes it needs (such as colormap and depth) from its widget parent—except for the visual itself. The Shell inherits the actual visual from its window (not widget) parent. Inheriting things that must be consistent from two different places causes trouble when using nondefault visuals because the inherited values are almost certain to be inconsistent.

In practice, this usually means that the default colormap and depth come from the widget parent, but that the visual is inherited from the root window (and is therefore the default visual). This may cause an X11 server BADMATCH error when everything else pertains to a nondefault visual.

To get around this problem, you need to do one of two things:

- Explicitly set the visual for each shell you want to put in a nondefault visual. That is, be careful to not depend on how the shell visual is inherited.
- Link with a modified *Shell.o* that inherits the visual from its widget tree instead of its window parent.

  A modified version of *Shell.c* is provided with the IRIS IM overlay demos so you can see what changes are needed.The requisite header files are not included. If you have your own X11 source, you have those header files and you can recompile it if you wish.

  A matching *Shell.o* is also supplied for those who want to link with this modified version. Many of the converted demo programs do this.

Note: This modified *Shell.c*, and the modified *Shell.o*, are supplied solely on a demonstration software basis. They are not supported products and might not appear in future releases. If you link with one of them, you do not get any fixes to *Shell.o* that might appear in future versions of *libXt*. ♦

IRIS IM Programming with Nondefault Visuals

Most of the problems you might have with putting IRIS IM widgets in the overlay or popup planes arise from the need to use nondefault visuals; however, color range issues can arise if you try to put a 4-color widget in a framebuffer only two bits deep.

Limited Color Range in Overlay and Popup Planes

Because the popup and 2-bit overlay framebuffers are two bits deep, the colormap for these framebuffers contains only four color values. In colormaps for popup or overlay framebuffers, pixel 0 is always reserved to mean transparent, which leaves room for three additional colors.

Note: On workstations with configurable framebuffers, for example workstations with RealityEngine graphics, pixel 0 is not reserved to mean transparent. You can use all color values on these workstations. ♦

An IRIS IM widget generally uses at least four colors (foreground, background, top shadow color, bottom shadow color); however, allocating a fourth color fails. Silicon Graphics has modified certain color selection algorithms in *libXm* to deal with colormap overflow by choosing the "best" available color from those already in the colormap. The color chosen is not the background color (unless you are explicitly setting the background color), nor is it transparent.

Because of the limited number of choices, the "best" available color might not be very close to the one you actually requested. To produce the best possible appearance for the widget, be sure that you select your three colors carefully.

These Silicon Graphics modifications apply only to colors set in the IRIS IM library, *libXm*. Silicon Graphics did not modify color selection in *libXt*, which sets colors for things such as *BorderColor*. If Xt cannot allocate the color you request, it returns pixel 0, which is transparent in popups and overlays.

MenuBar Uses a Common Shell for Pull-down Menus

The IRIS IM MenuBar includes an optimization that uses a common shell for all of the pull-down menus of a single MenuBar, meaning that all such menus will be in the same visual; this should cause no practical problems.

Pixmaps in Nondefault Visuals

As of Release 1.2 there is a new call, **XmGetPixmapByDepth()**, which you should use instead of **XmGetPixmap()**.

XmGetPixmap() assumes *DefaultDepth* when it creates pixmaps and does not let you set the depth resource. You cannot use **XmGetPixmap()** to create pixmaps for use in a visual whose depth differs from the default depth.

Because the Xt resource converter uses the default depth, you cannot specify pixmaps in an application default file if they will be used in a visual whose depth is different from that of the default visual.

Beware of Visuals-related Issues in *libMrm*

The IRIS IM resource manager (*libMrm*) has many visuals-related bugs that occur when you use a nondefault visual. Silicon Graphics has not fixed them. The bugs occur because *libMrm* relies on defaults, such as default depth and the default colormap. You can still use UIL, but use it with caution.

In your UIL file, avoid:

pixmaps	*mrm* uses the default depth
bitmap files	*mrm* uses the default depth
colors	*mrm* uses the colormap index appropriate to the default colormap. That index might not even exist in your colormap (for example, there is no pixel 10 in a 4-entry colormap).

You can specify your colors in a resource file instead of the UIL file, but you must set up your bitmaps and pixmaps programmatically. The pixmap resource converter uses the default depth when converting a resource.

Using IRIS IM Interface Builders with Nondefault Visuals

Because default-file resources cannot deal with nondefault visual resource issues, it is possible that none of the interface builders work properly in this environment.

In particular, *TeleUSE* and *UIMX* do not directly work with IRIS IM overlay use. A possible workaround is to hand code anything that needed a nondefault visual and then link this to the builder-generated code. This hand code could even be initially generated by the builder tool (for a default visual) and then hand-modified.

Nondefault Visual Heuristic Routines

The files */usr/src/X11/motif/overlay_demos/lib/sgi_visual.[ch]* provide convenient routines to get legal and consistent values for use with different visuals (such as overlay planes). Some of these routines are used by some Silicon Graphics code—for example, *4Dwm*, *toolchest*—and by the overlay demonstration programs. This encapsulation is useful because:

- The functions are commonly needed by software wanting to open windows in the overlay or popup planes.
- Opening windows in the overlay or popup planes is (understandably) a mystery to many application writers.
- Unfortunately, some of these requirements have neither a standard X11 nor a standard Silicon Graphics way to do them. This encapsulation includes a number of heuristic routines that may change in time.

Note: *sgi_visual.c* is unsupported demonstration code. It is extremely likely that details of the code, and even the programming interface it presents, will be different in future releases. If you make use of this code in your own software, you should make a copy of it so that it won't be overwritten by later installation of a changed version. ♦

The routines in *sgi_visual.c* deal with consistent sets of the following values: depth, visual class (for example, PseudoColor), visual type (for example, OVERLAY), X11 visual, and colormaps. The other members of the *XSetWindowAttributes* structure must also be consistent, but they are easier for the application itself to keep track of and set correctly. Therefore, *sgi_visual.c* does not do anything with them.

The following functions are included and are among the simplest to use. These are the functions used by the overlay demo programs. Each function inserts compatible values for the following visual-related resources into the *arglist* for a toolkit widget: colormap, depth, and visual.

The difference between the routines is the visual type (popup, overlay, or normal) of the visual:

- **int SG_getPopupArgs(dpy, scr, args, n)**
- **int SG_getOverlayArgs(dpy, scr, args, n)**
- **int SG_getOverlay2Args(dpy, scr, args, n)**

- **int SG_getOverlay4Args(dpy, scr, args, n)**
- **int SG_getNormalArgs(dpy, scr, args, n)**

These functions require these arguments:

Display **dpy*	The display is used to find the root window.
int *scr*	Screen number
Arglist *args*	Must have room for the three arguments to be added by this routine; there may be more arguments in the future.
int **n*	Index into the arguments list. It will be incremented for each argument added.

If the calling program asks for a visual type that does not exist on the hardware, the function automatically returns the best one it can.

Adding Input Devices with the X Input Extension

To add an input device, you write a streams module that reformats the device events to the format used by the X server and handles the information and control *ioctls* that are used by the X server. In most cases, events go to the X server, where they are distributed based on focus policy. It is possible for a single program to own all the events generated by a device.

Follow the procedure for adding input devices used by the X Window System and the X Input Extension. The X Input Extension is an X Consortium standard under X11R5. Documentation on the X Input Extension can be obtained in the X11R5 distribution from MIT in the *mit/doc/extensions/xinput* and *mit/hardcopy/extensions/input* directories.

For more specific information about adding an input device to a workstation, refer to the subdirectory on input devices in the *4Dgifts* directory on your IRIS-4D Series workstation. This directory contains a *README* file with very detailed instructions on input devices, as well as some useful example programs that demonstrate how to add an input device. If you can't find the *4Dgifts* directory on input devices, refer to the *README* file in */usr/people/4Dgifts*. This *README* file explains the structure and contents of the *4Dgifts* directory.

Index

K

L

M

N

O

P

Q

R

Z

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-1472-020.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 415-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389