Published in the July/August 1993 issue of *The X Journal.*

# Programming X Overlay Windows

Mark J. Kilgard*
*Silicon Graphics Inc.*
*Revision* : 1.12

September 2, 1993

## Abstract

Overlay planes provide an alternate set of frame buffer bitplanes which can be preferentially displayed instead of the normal set of bitplanes. Overlay planes have been common in high-end graphics systems for some time. Recently, work has been done by Silicon Graphics to integrate overlay plane support into the X Window System. A standard convention proposed and implemented by Silicon Graphics allows X client writers to create windows in the overlay planes. This article describes how to write programs to utilize overlay planes.

## 1 Introduction

Overlay planes are a common feature of high-end graphics systems. Overlay planes provide an alternate set of frame buffer bitplanes which can be preferentially displayed instead of the standard set of bitplanes (often called the normal planes). Overlay planes allow avoiding damage to an image retained in the normal planes by instead rendering into the overlay planes. Pop up menus, dialogs, rubber banding, and text annotation are all functions which can make use of overlay planes to minimize screen damage. Overlays can also be used to achieve transparency effects.

Silicon Graphics Inc. (abbreviated SGI) supports overlay planes across its full range of graphics hardware. A large amount of effort was expended by SGI to seamlessly integrate overlay planes support with the X Window System [6, 1, 5].

It is worth noting that overlay planes can be generalized to an arbitrary number of frame buffer *layers*. Underlay planes are bitplanes that can defer their pixel values to the pixel values in the normal planes. Additionally, multiple sets of overlay and underlay planes are possible. In theory, the notion of normal planes is a relative concept. In practice, it is useful to differentiate some layer of bitplanes by calling them the normal planes. Layers above the normal planes are considered overlays. Layers below the normal planes are underlays.

For the purpose of this article, we discuss overlay planes because they tend to be the most useful and only the most recent SGI graphics hardware is well-suited for supporting underlay planes for X windows. Keep in mind the same basic mechanisms described can also be used to support underlay planes. Other workstation vendors like Hewlett-Packard also support X overlay windows.

This article describes support for overlay planes from an X client writer's perspective, detailing how to create and manage windows created in the overlay planes using a convention promoted as a standard by SGI. The following section further explores the usefulness of overlay planes integrated with a window system like X. The third section details SGI's Server Overlay Visuals proposal [4] and provides Xlib programming examples of how to utilize overlay planes in X. The fourth section discusses various pitfalls and portability issues when using overlay planes. The fifth section explores the future possibilities for better supporting overlays.

## 2 Utility of Overlays

It is important to recognize the advantages overlay planes allow. Overlay planes allow screen repainting to be reduced and the implementation of efficient transparency effects. Rob Gabbard of SDRC [2] has also presented a strong case that overlays can significantly improve the responsiveness of user interfaces for 3D applications.

### 2.1 Minimizing Screen Repaint Costs

Anyone familiar with X or most any other window system understands that programs must be able to "repair" damaged or freshly exposed regions of their windows. For most simple applications, regenerating a portion of a window is

*Mark graduated with B.A. in Computer Science from Rice University and is currently a Member of the Technical Staff at Silicon Graphics and can be reached by electronic mail addressed to `mjk@sgi.com`

not very arduous. But the fact still remains that a significant share of compute cycles and graphics bandwidth is spent regenerating window contents.

The core X protocol supports a notion of backing store which can minimize the client costs of regenerating windows. But it greatly increases the X server's burden by asking the server to maintain in off-screen memory the contents of obscured or unmapped windows. A related core X protocol functionality known as save-unders saves the contents of windows obscured by a window using save-unders.

Overlay planes provide another means of minimizing window regeneration. It is notable that a significant amount of window damage is due to "popping up" transient windows such as pop-up or pull-down menus and dialog boxes. By creating such transient windows in the overlay planes, damage to underlying more permanent windows is avoided.

Notice that overlay planes gain advantage in the same way save-unders do. They both avoid damage to underlying windows.

But if we examine the two mechanisms closely, we see that overlay planes have some decided advantages over save-unders and backing store. First, because hardware is used to implement overlay planes, there is little overhead to using them. Backing store and save-unders both have high software overhead. For save-unders even though the client does not need to be aware that damage is happening to windows, the server must still save the pixels in the frame buffer and be ready to rewrite them back to the frame buffer. But in the overlay case, the pixels in the underlying window *never* need to be copied or redrawn even by the server.

Second, rendering to the obscured window while the window is still obscured can utilize the graphics hardware's faster rendering path instead of attempting to divert rendering into often slower off-screen pixmaps as backing store does.

Third, with modern graphics hardware, particularly hardware which supports 3D, the frame buffer holds much more information than just the pixel's color. It can also contain alpha buffer, accumulation buffer, stencil buffer, and depth buffer information. So called "deep" frame buffers magnify the costs of moving data off-screen then back into the frame buffer again. And in the case of direct hardware access graphics libraries, the server is not involved in the rendering making it impossible to effectively retain backing store for such windows.

### 2.2 Transparency

Along with minimizing screen repainting, overlay planes can be utilized to generate transparency effects efficiently. Usually overlay hardware supports a special *transparent* pixel value. If this value is drawn into the window, the pixel value in the layer below "shows through."

Imagine an application which generates annotated weather maps of the United States. The map of the United States itself is unchanging but the front lines and temperatures and other symbols which are painted on top of the static map do change.

By utilizing overlay planes, we can quickly redraw the weather map by only redrawing the meteorological annotations for the map. To do this, we draw the static map in a window located in the normal planes. Then create an overlay window as a child of the static map's window. The background pixel for this window is the overlay planes' transparent pixel. Effectively, we see through the overlay to the map. Now we can draw the annotations in the overlay window. When a new set of annotations is to be drawn, we clear only the overlay window, leaving the static map untouched, and redraw a new set of annotations.

Of course the same application could be written without overlays but overlays allow a much more efficient implementation by eliminating redrawing of the static map.

Video game style animation can also be efficiently implemented using overlays. Space ships, asteroids, and sneaker-wearing hedgehogs can be drawn into an overlay window while an intricate background window scrolls by in a normal plane window using transparency effects.

### 2.3 A Single Window Hierarchy

One thing worth noting about windows which can exist in overlay planes is that such windows should exist in the *same* window hierarchy as the normal plane windows. Input distribution should work no different for windows in different layers. A user should be able to push/pop windows at will, regardless of what layer they are in. There should be no restrictions about how windows in different layers should be parented. Layered windows should observe the same protocol semantics as normal single layer X server implementations. The only ways layered windows should affect the server is that higher layers do not clip layers beneath them and in the existence of a transparent pixel.

This mode of operation ensures a user need not be aware of what layer a window resides in and is not exposed to any layering artifacts (modulo transparency effects).

## 3 The Overlay Visuals Convention

One might expect overlay support for X would require an extension. In fact, no additional requests or events are needed to support overlays so a true X extension is unnecessary. The core X protocol's visual mechanism provides a way to create windows of different types. The only thing which is necessary is to support a way to advertise which

visuals are overlay visuals since the core X notion of a visual does not include layer information.

By leaving a properly formatted property on the root window of each screen describing what visuals are for overlay planes, a client can inspect that property knowing its format and determine which visuals are overlay visuals. The included information would also indicate what layer the visual is in (remember, there can be multiple sets of overlays) and how transparency is implemented.

It is up to the client to select the visual appropriate to the client's needs. The standard `CreateWindow` protocol request is used to create a window. If the visual specified for the `CreateWindow` request is an overlay visual, the window will be created in the visual's specified layer. An Xlib programmer can simply call `XCreateWindow` with a `Visual*` of an overlay visual to create an overlay window.

So how does an X client determine what visuals are overlay visuals? The Overlay Visuals Convention specifies that a property named `SERVER_OVERLAY_VISUALS` should be placed on the root window of each screen supporting overlays by the X server. The X server itself creates the property. The property has a standard format. It consists of elements (as described in Figure 1) which specify a visual, the type of transparency supported by the visual, what layer the visual resides in, and a transparency value which can be treated as a mask or a pixel value depending on the type of transparency supported. The `SERVER_OVERLAY_VISUALS` root window property is expected to be of type `SERVER_OVERLAY_VISUALS` and must be 32-bit in format.

The transparency type is an enumerated value indicating how transparency works for the visual. The following transparency types are possible:

**None** There are no transparent pixels. The value field should be ignored.

**TransparentPixel** The value field explicitly names a transparent pixel.

**TransparentMask** Any pixel value which has at least the same bits set as the value field is transparent.

The same visual may appear more than once in the list. In this case, the union of the pixel values described by the transparent type and value fields should all be transparent. The value of the layer field will be the same across all instances of the multiply listed visual.

## 3.1 Sample Xlib Programming Interface

The convention does not specify an Xlib interface to query the `SERVER_OVERLAY_VISUALS` property. Client programmers wishing to use overlay windows have been forced to query and decode the property without help from utility routines. This article includes routines that mimic the Xlib `XGetVisualInfo` and `XMatchVisualInfo` routines,

| Name | Type | Description |
|---|---|---|
| overlay_visual | VISUALID | Visual ID of visual. |
| transparent_type | CARD32 | None (0)<br>TransparentPixel (1)<br>TransparentMask (2) |
| value | CARD32 | Pixel value or transparency mask. |
| layer | INT32 | Which layer the visual resides in. |

Figure 1: SERVER_OVERLAY_VISUALS property entry format.

but are augmented to also provide support for querying the layering and transparency capabilities of visuals. The implementations of the routines written in C can be found in appendices A and B.

`XGetLayerVisualInfo` works like `XGetVisualInfo` but instead of using a `XVisualInfo` structure as a template and to return information on each visual, a `XLayerVisualInfo` structure is used which has the `XVisualInfo` structure embedded in it but also contains fields for layer and transparency. The normal visual information mask bits are extended to support the new fields. The routine hides all the work done to query and interpret the `SERVER_OVERLAY_VISUALS` property.

`XMatchLayerVisualInfo` works like `XMatchVisualInfo` but is extended in the same way `XGetLayerVisualInfo` is using the `XLayerVisualInfo` structure. You can supply an additional `layer` parameter for matching a visual in a specified layer.

As mentioned earlier, a single overlay visual may support several transparency values. These routines only return a single transparency type and value.

## 3.2 Example

To illustrate how to use an overlay window, Appendix C uses the routines described above to query the server for an overlay visual and then creates a normal plane window with a child in the overlay planes completely overlapping its parent. The background pixel value is set to a transparent pixel. Then like in the weather map example mentioned earlier, a semi-intricate black on white image is drawn in the normal planes while red text annotation is drawn in the overlay planes. Each time a mouse button is clicked, the overlay planes are cleared and the text is redrawn in a different position. The example demonstrates how the overlay planes can be modified without disturbing the image in the normal planes.

The example tries to find two appropriate visuals to use in making its "layer sandwich." The code first looks at the default visual and tries to find a visual "above" it with transparency supported. If such a visual is not found, the code looks for a visual "below" the default visual to use but

this requires that the default visual supports transparency. Note this strategy could fail to find an appropriate pair of visuals in some cases even when two potentially layerable visuals exist on a given server. The code is meant to be a relatively short, simple example to elucidate how to use layers in X; it is not meant to be a piece of production code.

The window created in the lower layer is the parent and the window created in the higher layer must be the child. It is important this ordering is not reversed. The layered transparency effect implies lower layers show through the transparent pixels of higher levels; not the other way around.

Because the overlay window is a child of the toplevel normal plane window, if we move the client's toplevel window, the overlay window moves too. Even though the windows are in different layers, the window hierarchy still dictates how windows move and interact.

Also consistent with the window hierarchy is the way events are distributed for overlay windows. In our example, the child covers the entire normal plane window so it receives all button clicks. Note that an overlay window receives mouse clicks even if the mouse is located on a transparent pixel. Transparency just affects how the pixel is displayed. Transparency does *not* affect the clipping region of a window. Similarly, if a transparent pixel is read from an overlay window using `XGetImage` the value read is *not* the displayed value but the transparent value for the pixel. So transparency does not change the value of the pixel in the window; it only affects the color displayed on the screen.

Appendix D presents one more simple example for a program called `xlayerinfo` that lists the visuals supported by a server including information about layering and transparency effects.

### 3.3 Vendor Support for Overlays

In IRIX 4.0,[1] overlay plane support was introduced. SGI's `4Dwm` Motif-based window manager and other SGI X clients such as the `xwsh` terminal emulator make use of the overlay planes to speed pop-up menus and support efficient rubber banding. Additionally, the IRIS Graphics Library allows rendering into the overlay planes. Sample code is provided demonstrating how to support overlay planes in Motif-based applications.

In the 5.0 release of IRIX, the SGI X product's overlay support was substantially re-engineered to improve upon the original implementation. Both overlay and underlay visuals are supported on some graphics configurations such as the Reality Engine.

Hewlett-Packard [3, 9] also supports overlays on some of their workstations in particular those with CRX24 and CRX48Z graphics hardware. Hewlett-Packard implements overlays by making the default visual be in an 8-bit overlay (supporting a transparent pixel). This approach means most X clients automatically run in the overlay planes.

Many other vendors have also implemented overlays for X or are considering implementing them.

The code presented in the appendices works correctly on Silicon Graphics and Hewlett-Packard workstations which support X overlay windows.

[1] IRIX is SGI's version of the Unix operating system.

## 4 Usage Considerations

By overloading the X notion of visual with layering information, overlays can be integrated into the X Window System without an extension or other radical restructuring of the server's operation. Even so, there are a number of considerations to keep in mind when using overlay windows.

### 4.1 Shallower Depths

It is common for the overlay planes to not be as deep as the normal planes. For example, the Indigo with Starter graphics (SGI's lowest end graphics platform) supports 8-bit deep normal planes but only provides a 2-bit deep overlay plane. Mid-range SGI graphics platforms support 4-bit overlays while the normal planes support up to 24-bit deep normal plane windows. SGI's high end Reality Engine does support an 8-bit overlay. You can expect deeper overlay planes in the future but do not be surprised by shallow overlay plane visuals.

Hewlett-Packard avoids this issue of shallow depths by have an 8-bit overlay plane.

When you write a program to use overlay planes, you should remember that the overlay planes may be substantially shallower than the default visual. In the case of a 2-bit overlay visual, keep in mind that you probably only have 3 colors to use (since one is transparent). Efficient use of color resources is important.

Also be advised that creating windows in the overlay planes generally means using visuals other than the default visual. Most X programmers always use the default visual and are not aware of the particular rules necessary to create a window in a visual other than the default. You must always specify a colormap and a border pixel color or pixmap when creating a window not using the default visual. Otherwise a `BadMatch` protocol error will occur. `XCreateSimpleWindow` will not be sufficient; you will need to use the more general `XCreateWindow`.

### 4.2 Overlay Colormaps

Colormaps create another area of concern. Normally if you create windows using the default visual, you use the default colormap and allocate colors from it. Since most

windows use the default colormap, occasions when the colormap is not installed are rare. Since overlays can not typically use the default colormap and each client ends up needing to create a unique colormap to use with its overlay windows, colormap flashing problems are easy to create in the overlay planes. And generally overlay planes do not support multiple simultaneous colormaps.

However it is generally true that the overlay planes often have a distinct hardware colormap from the normal planes meaning a colormap for the overlay planes and a colormap for the normal planes can usually be installed simultaneously. This is the case on SGI graphics hardware.

If you are new to the practice of using non-default colormaps, you should note that `BlackPixel` and `WhitePixel` return pixel values for the *default* colormap. Colormaps created with `XCreateColormap` are not created with preallocated black and white pixels. If you need black and white pixels, you should allocate them yourself.

Also remember to use `XSetWMColormapWindows` if you have a window using a colormap not the same as the colormap of your toplevel window. This allows a window manager to install the appropriate colormaps for your client.

There exists one more caveat to using overlay colormaps. Because the transparent pixel is preallocated as pixel zero in SGI X servers, it is impossible to allocate all the colormap cells. SGI has opted to generate a `BadAlloc` error if a colormap for an overlay visual is created with the `AllocAll` parameter. Client programs should instead use `AllocNone` when creating overlay colormaps and allocate colormap cells individually.

Hewlett-Packard's implementation puts the default visual in the overlay planes making it possible to use the default colormap.

### 4.3 Window Manager Interactions

Window managers are another consideration when programming with overlays. Depending on the window manager, border decoration may or may not be created to exist in the overlay planes. If the borders are not in the overlay planes, expose events for underlying normal plane windows will still be generated when the overlay window is moved or unmapped due to the damage caused by the window manager borders. The chief advantage of creating overlay windows is negated. But if the window manager does create the borders in the overlay planes, the colormap used by the window manager will be different from the colormap for the client window nearly guaranteeing colormap flashing.

The best advice is to create toplevel overlay plane windows enabling override-redirect or without any window manager decoration. Since most uses of toplevel overlays are for transient windows, this advice is generally easy to follow.

In summary, due to the fact that overlays are generally shallower than the normal planes visuals and not the default visual, there are a number of considerations to take into account when creating overlay windows. Efforts to avoid exposes or colormap flashing also means window manager decoration should generally be avoided for overlay windows. Because overlays are used for transient windows mostly, these limitations are generally acceptable if client writers are aware of them.

## 5 Future Directions

The Server Overlay Visuals convention is only a starting point for effectively utilizing overlays in X. It exposes the base functionality but more should be done to improve on the utility of overlay visuals and windows.

### 5.1 Colormap Coordination

As discussed in the previous section, the inability to share colormaps with the window manager's border decoration makes it visually unappealing to create window manager decorated windows in the overlay plans. This problem could be solved if a convention was established where window managers would make available (probably through a property on the root window) the colormap the window manager intended to use for overlay border decorations. Such a convention might be part of a future revision to the Inter Client Communication Conventions Manual (ICCCM). Effectively, a default colormap for the overlay planes would be established. Multiple overlay windows could share the window manager's overlay colormap to avoid colormap flashing the same way the default colormap achieves this end for the default visual.

### 5.2 Centralized Visual Selection

Certainly a standard programming interface should be developed for querying overlay visuals. The sample routines provided in the appendices of this article provide a starting point for such an interface.

The problem of querying the layering capabilities of visuals is a subset of the larger *visual selection* problem in X. Sophisticated extensions such as PEX and OpenGL are overloading visuals to describe what capabilities are available for various classes of windows. The `PEXMatchRenderTargets` request in PEX 5.1 [8] determines what visuals can be used for PEX rendering. OpenGL [7] supplies the `glxChooseVisual` and `glxGetConfig` routines to perform visual selection.[2]

[2] OpenGL has a very sophisticated notion of frame buffer capabilities expressed through visuals. In fact, the frame buffer layer (*frame buffer level* in OpenGL terminology) is one of many capabilities that can be queried via the OpenGL visual selection mechanism.

While most X programmers simply use the default visual, in the future X programmers can expect to choose non-default visuals to suit sophisticated needs. Imagine how difficult visual selection might be if multiple ad hoc sources of visual information had to be consulted and it was up to the client program to combine all the information and properly select the most appropriate visual. Anticipating the complexity of this task, progress should be made in developing a single, centralized mechanism for determining the capabilities of supported visuals.

In the future, querying the `SERVER_OVERLAY_VISUALS` property could be denigrated in favor of a more general mechanism. For now, the property is the standard mechanism and future servers will be expected to continue supporting it even if an improved visual selection mechanism is introduced.

### 5.3 Default Overlay Visuals

As discussed earlier in the paper, it can be reasonable to start the X server having the default visual be an overlay visual. This is the approach taken by Hewlett-Packard. This would mean common X clients automatically use the overlay planes for creating windows. The normal planes could then be used for complex, probably 3D applications with scenes that are difficult to regenerate. Interacting with basic X clients in the overlay planes would not disturb the complicated scene thus avoiding expensive redraws.

### 5.4 Hooks in DIX for Overlays

Supporting layers in the X11R5 and previous MIT release of X was difficult because the the Device Independent X (DIX) code in the server implicitly assumes that all windows reside in the same layer. For overlays, this assumption is not true. The code to do window tree validation inside the X server is complicated by layers. With X11R6 you can look forward to the proper hooks existing in DIX to support layers without vendor modification to the DIX layer.

## 6 Conclusions

Overlay visuals and windows are not common across a wide spectrum of X servers yet. With the implementation of overlay visuals now properly understood and a standard convention for advertising such visuals available, overlays in X should become a common capability of high-end X servers.

Hopefully the information in this article has provided programmers with enough information to start experimenting with overlays in X and integrating overlay support into their applications. It is also hoped X users will begin demanding overlay support in their X servers and enjoy the advantages of overlays in their everyday work.

## 7 Acknowledgements

Phil Karlton (SGI) initially proposed the server overlay visuals convention to the X Consortium. David Wiggins (Intergraph) contributed to the proposal. Todd Newman (SGI) did the first implementation of overlay visuals; Peter Daifuku (SGI) implemented SGI's current implementation for IRIX 5.0. John Marks (Hewlett-Packard) helped me ensure the example code worked on Hewlett-Packard workstations too.

# A XLayerUtil.h

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xmd.h>

/* Transparent type values */
/*      None                    0 */
#define TransparentPixel      1
#define TransparentMask       2

/* layered visual info template flags */
#define VisualLayerMask         0x200
#define VisualTransparentType   0x400
#define VisualTransparentValue  0x800
#define VisualAllLayerMask      0xFFF

/* layered visual info structure */
typedef struct _XLayerVisualInfo {
   XVisualInfo vinfo;
   int layer;
   int type;
   unsigned long value;
} XLayerVisualInfo;

/* SERVER_OVERLAY_VISUALS property element */
typedef struct _OverlayInfo {
   CARD32  overlay_visual;
   CARD32  transparent_type;
   CARD32  value;
   CARD32  layer;
} OverlayInfo;

extern XLayerVisualInfo *XGetLayerVisualInfo(Display*,long,XLayerVisualInfo*,int*);
extern Status XMatchLayerVisualInfo(Display*,int,int,int,int,XLayerVisualInfo*);
```

# B XLayerUtil.c

```
#include <stdio.h>
#include "XLayerUtil.h"

static Bool layersRead;
static Atom overlayVisualsAtom;
static OverlayInfo **overlayInfoPerScreen;
static int *numOverlaysPerScreen;

XLayerVisualInfo *
XGetLayerVisualInfo(display, lvinfo_mask, lvinfo_template, nitems_return)
   Display *display;
   long lvinfo_mask;
   XLayerVisualInfo *lvinfo_template;
   int *nitems_return;
{
   XVisualInfo *vinfo;
   XLayerVisualInfo *layerInfo;
   Window root;
   Status status;
   Atom actualType;
   unsigned long sizeData, bytesLeft;
```

```
int actualFormat, numVisuals, numScreens, count, i, j;

vinfo = XGetVisualInfo(display, lvinfo_mask&VisualAllMask,
   &lvinfo_template->vinfo, nitems_return);
if(vinfo == NULL)
   return NULL;
numVisuals = *nitems_return;
if(layersRead == False) {
   overlayVisualsAtom = XInternAtom(display, "SERVER_OVERLAY_VISUALS", True);
   if(overlayVisualsAtom != None) {
      numScreens = ScreenCount(display);
      overlayInfoPerScreen = (OverlayInfo**) malloc(numScreens*sizeof(OverlayInfo*));
      numOverlaysPerScreen = (int*) malloc(numScreens*sizeof(int));
      if(overlayInfoPerScreen != NULL && numOverlaysPerScreen != NULL) {
         for(i=0; i<numScreens; i++) {
            root = RootWindow(display, i);
            status = XGetWindowProperty(display, root, overlayVisualsAtom,
               0L, (long)10000, False, overlayVisualsAtom, &actualType, &actualFormat,
               &sizeData, &bytesLeft, (char**) &overlayInfoPerScreen[i]);
            if(status!=Success || actualType!=overlayVisualsAtom ||
              actualFormat!=32 || sizeData<4)
               numOverlaysPerScreen[i] = 0;
            else
               numOverlaysPerScreen[i] = sizeData / (sizeof(OverlayInfo)/4);
         }
         layersRead = True;
      } else {
         if(overlayInfoPerScreen != NULL) free(overlayInfoPerScreen);
         if(numOverlaysPerScreen != NULL) free(numOverlaysPerScreen);
      }
   }
}
layerInfo = (XLayerVisualInfo*) malloc(numVisuals * sizeof(XLayerVisualInfo));
if(layerInfo == NULL) {
   XFree(vinfo);
   return NULL;
}
count = 0;
for(i=0; i<numVisuals; i++) {
   XVisualInfo *pVinfo;
   int screen;
   OverlayInfo *overlayInfo;

   pVinfo = &vinfo[i];
   screen = pVinfo->screen;
   overlayInfo = NULL;
   if(layersRead) {
      for(j=0; j<numOverlaysPerScreen[screen]; j++)
         if(pVinfo->visualid == overlayInfoPerScreen[screen][j].overlay_visual) {
            overlayInfo = &overlayInfoPerScreen[screen][j];
            break;
         }
   }
   if(lvinfo_mask & VisualLayerMask)
      if(overlayInfo == NULL) {
         if(lvinfo_template->layer != 0) continue;
      } else
         if(lvinfo_template->layer != overlayInfo->layer) continue;
   if(lvinfo_mask & VisualTransparentType)
      if(overlayInfo == NULL) {
```

```
            if(lvinfo_template->type != None) continue;
         } else
            if(lvinfo_template->type != overlayInfo->transparent_type) continue;
      if(lvinfo_mask & VisualTransparentValue)
         if(overlayInfo == NULL)
            /* non-overlay visuals have no sense of TransparentValue */
            continue;
         else
            if(lvinfo_template->value != overlayInfo->value) continue;
      layerInfo[count].vinfo = *pVinfo;
      if(overlayInfo == NULL) {
         layerInfo[count].layer = 0;
         layerInfo[count].type = None;
         layerInfo[count].value = 0; /* meaningless */
      } else {
         layerInfo[count].layer = overlayInfo->layer;
         layerInfo[count].type = overlayInfo->transparent_type;
         layerInfo[count].value = overlayInfo->value;
      }
      count++;
   }
   XFree(vinfo);
   *nitems_return = count;
   if(count == 0) {
      XFree(layerInfo);
      return NULL;
   } else
      return layerInfo;
}

Status
XMatchLayerVisualInfo(display, screen, depth, class, layer, lvinfo_return)
Display *display;
int screen, depth, class, layer;
XLayerVisualInfo *lvinfo_return;
{
   XLayerVisualInfo *lvinfo;
   XLayerVisualInfo lvinfoTemplate;
   int nitems;

   lvinfoTemplate.vinfo.screen = screen;
   lvinfoTemplate.vinfo.depth = depth;
   lvinfoTemplate.vinfo.class = class;
   lvinfoTemplate.layer = layer;
   lvinfo = XGetLayerVisualInfo(display,
      VisualScreenMask|VisualDepthMask|VisualClassMask|VisualLayerMask,
      &lvinfoTemplate, &nitems);
   if(lvinfo != NULL && nitems > 0) {
      *lvinfo_return = *lvinfo;
      return 1;
   } else
      return 0;
}
```

# C layerdemo.c

```
/* compile: cc -o layerdemo layerdemo.c XLayerUtil.c -lX11 -lm */

/* Tested on Silicon Graphics IRIX 4.0 and IRIX 5.0 workstations and
 * Hewlett-Packard workstations with CRX24 and CRX48Z graphics hardware.
 */

#include <stdio.h>
#include <math.h>
#include "XLayerUtil.h"

#define SIZE 400 /* width and height of window */

Display *dpy;
Window root, win, overlay;
Colormap cmap;
Visual *defaultVisual;
int screen, black, white, red, nVisuals, i, status;
GC normalGC, overlayGC;
XEvent event;
XLayerVisualInfo template;
XLayerVisualInfo *otherLayerInfo, *defaultLayerInfo;
XSetWindowAttributes winattrs;
XGCValues gcvals;
XColor color, exact;
int x = 0, y = SIZE/2;

redrawNormalPlanes()
{
   /* draw a black 43 legged octopus */
   for(i=0; i<43; i++)
      XDrawLine(dpy, win, normalGC, SIZE/2, SIZE/2,
         (int) (cos(i * 0.15) * (SIZE/2-5)) + SIZE/2,
         (int) (sin(i * 0.15) * (SIZE/2-12)) + SIZE/2);
}

#define MESSAGE1 "This text is in the"
#define MESSAGE2 "OVERLAY PLANES"

redrawOverlayPlanes()
{
   XDrawString(dpy, overlay, overlayGC, x, y, MESSAGE1, sizeof(MESSAGE1)-1);
   XDrawString(dpy, overlay, overlayGC, x, y + 15, MESSAGE2, sizeof(MESSAGE2)-1);
}

fatalError(message)
char *message;
{
   fprintf(stderr, "layerdemo: %s\n", message);
   exit(1);
}

main(argc, argv)
int argc;
char *argv[];
{
   dpy = XOpenDisplay(NULL);
   if(dpy == NULL) fatalError("cannot open display");
   screen = DefaultScreen(dpy);
```

```
root = RootWindow(dpy, screen);
defaultVisual = DefaultVisual(dpy, screen);
/* find layer of default visual */
template.vinfo.visualid = defaultVisual->visualid;
defaultLayerInfo = XGetLayerVisualInfo(dpy, VisualIDMask, &template, &nVisuals);
/* look for visual in layer "above" default visual with transparent pixel */
template.layer = defaultLayerInfo->layer + 1;
template.vinfo.screen = screen;
template.type = TransparentPixel;
otherLayerInfo = XGetLayerVisualInfo(dpy,
   VisualScreenMask|VisualLayerMask|VisualTransparentType, &template, &nVisuals);
if(otherLayerInfo == NULL) {
   /* make sure default visual has transparent pixel */
   if(defaultLayerInfo->type == None) fatalError("unable to find expected layer visuals");
   /* visual not found "above" default visual, try looking "below" */
   template.layer = defaultLayerInfo->layer - 1;
   template.vinfo.screen = screen;
   otherLayerInfo = XGetLayerVisualInfo(dpy,
      VisualScreenMask|VisualLayerMask, &template, &nVisuals);
   if(otherLayerInfo == NULL) fatalError("unable to find layer below default visual");
   /* XCreateColormap uses AllocNone for 2 reasons:
    * 1) haven't determined class of visual, visual could have static colormap
    * and more importantly
    * 2) transparent pixel might make AllocAll impossible.
    */
   cmap = XCreateColormap(dpy, root, otherLayerInfo->vinfo.visual, AllocNone);
   /* not default colormap, must find our own black and white */
   status = XAllocNamedColor(dpy, cmap, "black", &color, &exact);
   if(status == 0) fatalError("could not allocate black");
   black = color.pixel;
   status = XAllocNamedColor(dpy, cmap, "white", &color, &exact);
   if(status == 0) fatalError("could not allocate white");
   white = color.pixel;
   winattrs.background_pixel = white;
   winattrs.border_pixel = black;
   winattrs.colormap = cmap;
   win = XCreateWindow(dpy, root, 10, 10, SIZE, SIZE, 0, otherLayerInfo->vinfo.depth,
      InputOutput, otherLayerInfo->vinfo.visual,
      CWBackPixel|CWBorderPixel|CWColormap, &winattrs);
   status = XAllocNamedColor(dpy, DefaultColormap(dpy, screen),
      "red", &color, &exact);
   if(status == 0) fatalError("could not allocate red");
   winattrs.background_pixel = defaultLayerInfo->value;
   winattrs.border_pixel = 0;
   winattrs.colormap = DefaultColormap(dpy, screen);
   overlay = XCreateWindow(dpy, win, 0, 0, SIZE, SIZE, 0, DefaultDepth(dpy, screen),
      InputOutput, defaultVisual, CWBackPixel|CWBorderPixel|CWColormap, &winattrs);
} else {
   /* create lower window using default visual */
   black = BlackPixel(dpy, screen);
   white = WhitePixel(dpy, screen);
   win = XCreateSimpleWindow(dpy, root, 10, 10, SIZE, SIZE, 1, black, white);
   /* see note above about AllocNone */
   cmap = XCreateColormap(dpy, root, otherLayerInfo->vinfo.visual, AllocNone);
   status = XAllocNamedColor(dpy, cmap, "red", &color, &exact);
   if(status == 0) fatalError("could not allocate red");
   red = color.pixel;
   winattrs.background_pixel = otherLayerInfo->value; /* use transparent pixel */
   winattrs.border_pixel = 0; /* no border but still necessary to avoid BadMatch */
   winattrs.colormap = cmap;
```

```
      overlay = XCreateWindow(dpy, win, 0, 0, SIZE, SIZE, 0, otherLayerInfo->vinfo.depth,
         InputOutput, otherLayerInfo->vinfo.visual,
         CWBackPixel|CWBorderPixel|CWColormap, &winattrs);
   }
   XSelectInput(dpy, win, ExposureMask);
   XSelectInput(dpy, overlay, ExposureMask|ButtonPressMask);
   XSetWMColormapWindows(dpy, win, &overlay, 1);
   gcvals.foreground = black;
   gcvals.line_width = 8;
   gcvals.cap_style = CapRound;
   normalGC = XCreateGC(dpy, win, GCForeground|GCLineWidth|GCCapStyle, &gcvals);
   gcvals.foreground = red;
   overlayGC = XCreateGC(dpy, overlay, GCForeground, &gcvals);
   XMapSubwindows(dpy, win);
   XMapWindow(dpy, win);
   while(1) {
      XNextEvent(dpy, &event);
      switch(event.type) {
      case Expose:
         if(event.xexpose.window == win)
            redrawNormalPlanes();
         else
            redrawOverlayPlanes();
         break;
      case ButtonPress:
         x = random() % SIZE/2;
         y = random() % SIZE;
         XClearWindow(dpy, overlay);
         redrawOverlayPlanes();
         break;
      }
   }
}
```

# D xlayerinfo.c

```
/* compile: cc -o xlayerinfo xlayerinfo.c XLayerUtil.c -lX11 */

#include <stdio.h>
#include "XLayerUtil.h"

main(argc, argv)
int argc;
char *argv[];
{
   Display *dpy;
   char *display_name, *arg, *class;
   XLayerVisualInfo template, *lvinfo;
   int nVisuals, i;

   display_name = NULL;
   for(i=1; i<argc; i++) {
      arg = argv[i];
      if(strcmp(arg, "-display") == 0 || strcmp(arg, "-d") == 0) {
         if(++i >= argc) {
            fprintf(stderr, "nmbxdemo: missing argument to -display\n");
            exit(1);
         }
         display_name = argv[i];
```

```
      }
   }
   dpy = XOpenDisplay(display_name);
   if(dpy == NULL) {
      fprintf(stderr, "xlayerinfo: cannot open display %s\n",
         XDisplayName(NULL));
      exit(1);
   }
   lvinfo = XGetLayerVisualInfo(dpy, 0L, &template, &nVisuals);
   for(i=0; i<nVisuals; i++) {
      printf("  Visual ID: 0x%x\n", lvinfo[i].vinfo.visualid);
      printf("    screen: %d\n", lvinfo[i].vinfo.screen);
      printf("    depth: %d\n", lvinfo[i].vinfo.depth);
      switch(lvinfo[i].vinfo.class) {
         case StaticGray:   class = "StaticGray"; break;
         case GrayScale:    class = "GrayScale"; break;
         case StaticColor:  class = "StaticColor"; break;
         case PseudoColor:  class = "PseudoColor"; break;
         case TrueColor:    class = "TrueColor"; break;
         case DirectColor:  class = "DirectColor"; break;
         default:           class = "Unknown"; break;
      }
      printf("    class: %s\n", class);
      switch(lvinfo[i].type) {
      case None:
         printf("    transparent type: None\n");
         break;
      case TransparentPixel:
         printf("    transparent type: TransparentPixel\n");
         printf("    pixel value: %d\n", lvinfo[i].value);
         break;
      case TransparentMask:
         printf("    transparent type: TransparentMask\n");
         printf("    transparency mask: %0x%x\n", lvinfo[i].value);
         break;
      default:
         printf("    transparent type: Unkown or invalid\n");
         break;
      }
      printf("    layer: %d\n", lvinfo[i].layer);
   }
}
```

# References


[1] Peter Daifuku, "A Fully Functional Implementation of Layered Windows," *The X Resource: Proceedings of the 7th Annual X Technical Conference*, O'Reilly & Associates, Issue 5, Winter 1993.

[2] Rob Gabbard, "Addressing the Problems Facing the Integration of 3D Applications With Modern User Interfaces," SDRC Graphics and User Interface Group, unpublished, July 1992.

[3] Steven Hiebert, John Lang, Keith Marchington, "Sharding Overlay and Image Planes in the Starbase/X11 Merge System," *Hewlett-Packard Journal*, December 1989.

[4] Phil Karlton and David Wiggins, "Describing Overlay Visuals," X Consortium communication, 1991.

[5] Mark J. Kilgard, "Going Beyond the MIT Sample Server: The Silicon Graphics X11 Server," *The X Journal*, SIGS Publications, January 1993.

[6] Todd Newman, "How Not to Implement Overlays in X," *The X Resource: Proceeding of the 6th Annual X Technical Conference*, O'Reilly & Associates, Issue 1, Winter 1992.

[7] OpenGL Architecture Review Board, *OpenGL Reference Manual: The official reference document for OpenGL, Release 1*, Addison Wesley, 1992.

[8] "PEX Protocol Specification and Encoding Version 5.1P," *The X Resource*, O'Reilly & Associates, Special Issue A, May 1992.

[9] Desi Rhoden and Chris Wilcox, "Hardware Acceleration for Window Systems," *Computer Graphics*, Association for Computing Machinery, Volume 23, Number 3, July 1989.