A Fully Functional Implementation of Layered Windows

Peter Daifuku†

Abstract

Incorporating layered windows into the X server is a non trivial task, which has been attempted repeatedly in the past, with varying lack of success. We present our criteria for the proper behavior of layered windows. We show that the assumptions built into the DIX windowing code prevent the proper implementation of layered windows, proving that the current windowing code is inherently device-dependent. We propose a restructuring of the sample server, moving much of the windowing code to DDX. We show how the sample windowing model can be extended to clip layered windows, and what changes are required to other parts of the server which depend in part on knowledge of the current window tree.

Introduction

Many hardware vendors support overlays as a performance enhancement. However, the sample server does not address the issue of overlays, or of other framebuffer layers. In the past, there have been a number of attempts to incorporate overlays into the X server. Most recently, in the 1992 X Conference, Newman [1] presented a preliminary implementation of overlays for the Silicon Graphics X server. After describing some of the details of this implementation, he addressed its shortcomings, and speculated as to future directions. We are presenting a follow-on to Newman's work, in which we address many of the issues left unresolved by his implementation, and provide details of the resultant solution.

The Semantics of Layered Windows

Newman[1] presented in detail the concepts and issues surrounding the use of overlays. Therefore, we will only review those concepts that are fundamental to understanding our implementation.

---

†*Peter Daifuku is a Member of the Technical Staff at Silicon Graphics Computer Systems.*

Terminology

Overlays are implemented using a separate set of bitplanes from the main framebuffer. They are overlays, because when enabled they are displayed preferentially with respect to the pixels in the normal planes. They are often used to draw transient menus in a separate set of bitplanes from the main graphics window, thus avoiding potentially expensive damage to the user's scene. Most hardware implementations also support a transparent pixel or plane. Again, a typical example would be the use of XDrawString to draw transient text where only the glyph itself obscured the background image. SGI hardware typically supports several layers of overlays, and optionally underlays. Underlays are similar to overlays, but now the normal planes are displayed preferentially with respect to the underlay planes. Again, the main framebuffer often has a transparent pixel or mask in order to reveal the underlay. A typical use of underlays stores the static portion of a scene, so that drawing the animated portion does not require the background to be redrawn with every frame. Because we are solving the problem of multiple layers, we use the term **layered windows** in preference to overlays. We number the layers in ascending hierarchical order, starting with layer 0. Thus, in a system with one layer each of overlays and underlays, the underlay planes are in layer 0, the normal or main planes are in layer 1, and the overlays in layer 2. The following pseudocode illustrates the decision making of the typical SGI display hardware.

For each pixel on the screen:

```
for (layer = maxLayer; layer >= 0; layer--)
{
   if (layerEnabled[layer] && data[layer] != transparentValue)
   {
         display(data[layer]);
         break;
   }
}
```

As we can see, if a given pixel has valid data in multiple layers, the highest layer is displayed, subject to potential transparency.

"Mechanism, Not Policy", And... "A Window is a Window is a Window"

These two cliches familiar to the X community illustrate the fundamental assumptions at the heart of our work. We believe that layered windows should behave just as any other window, and that the server should not enforce constraints on the use of layered windows by a given application. Other implementations have distinguished overlays from normal planes, for example by treating them as separate screens. This constrains applications severely, by limiting the flexibility and functionality available to them. These concerns only become more severe with additional layers. We believe that any implementation should satisfy the following criteria:

- Input distribution should be layer independent, i.e., the window that is visually uppermost under the cursor should receive mouse/button events.
- The user should be able to push/pop windows at will, regardless of what layer they are in.

- There should be no restrictions for window parenting. I.e., a window in layer **p** can parent a window in layer **m**, and vice versa.
- There should be no restrictions as concerns the default visual. I.e., any layer can be used for the root window.
- Layered windows should adhere to proper observance of colormap semantics and protocol compliance. Layers with transparent pixels or masks should not allow the application to allocate colors/planes which conflict with these.

Previous implementations have attempted to solve the layered problem by maintaining each layer as a separate screen, each with its own window tree. As a first and immediate consequence of our criteria, we can observe that they can only be met if we maintain a single window tree encompassing all layers.

DIX is Device Dependent

We can now examine whether it is possible to support our model for layered window behavior using the existing windowing code in DIX. As an example, let us walk through the process of

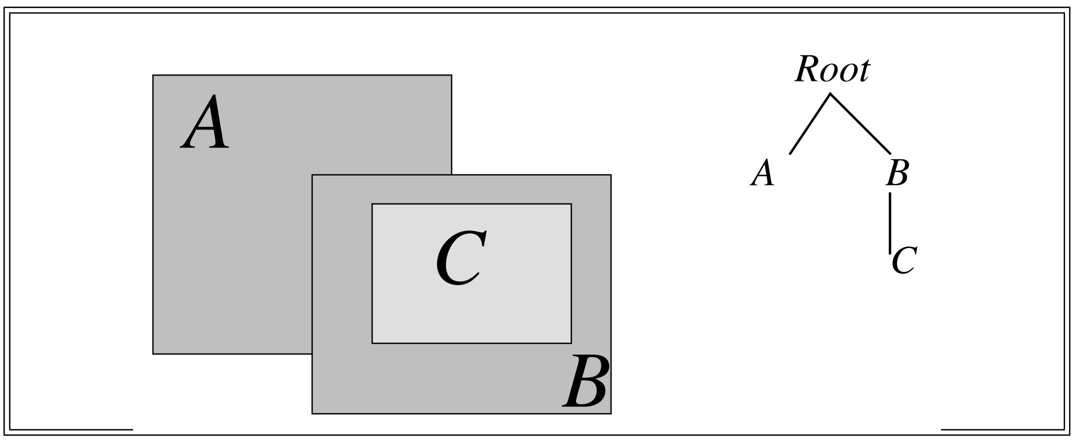

Figure 1: A typical window hierarchy

mapping a window. Figure 1 shows three windows, such that windows A and B are mapped siblings, and the window that we are mapping is window C, a child of window B. In order to calculate the new clip lists, the DIX code makes a crucial assumption, for the sake of performance: that the effect of a window operation on window C is confined to its parent B, its inferiors, and the siblings of C (and their inferiors) lower than C in the stacking order. This is a very powerful assumption, since it dramatically reduces the number of windows which need to recalculate their clip. In the example in figure 1, window A is not affected by window operations on window C, because window B shields A from any effects. Assume now that windows A and C are both in layer 0, and that window B is in layer 1. Since B is in a higher layer than A, B does not clip A. Instead, the display hardware conveys the illusion that B clips A. When window C is mapped, it clips both windows A and B. It clips window B, because window B is in a layer above it, and if left unclipped, its pixels would be displayed preferentially to C's. But C also clips window A, because window B no longer shields A from C. In a non-layered system, window B clips A, and window C can cause no further damage to A. In the layered case, since B did not damage A, and windows A and C share bitplanes, it is clear that window C must damage A. The effects of the window operation are no longer confined to the parent, but have been propagated to the grandparent. This simple example demonstrates that the fundamental assumption in the DIX windowing code is inadequate for layered windows, and hence device-dependent.

Restructuring the windowing code

We wish to propose to the X Consortium a restructuring of the windowing code, presumably for inclusion in the R6 server. Since the windowing code is device dependent, it does not belong in DIX, but rather in DDX. We propose that routines such as MapWindow, which currently are called directly from dispatch.c should instead be vectored through the screen structure, and that the current windowing routines be moved to DDX/MI. The minimum restructuring would include adding the following procedure pointers to the screen structure:

- ReparentWindow
- MapWindow
- MapSubwindows
- UnmapWindow
- UnmapSubwindows
- ConfigureWindow
- CirculateWindow
- SetShape

The procedures by the same names would be prefaced by mi (e.g., miReparentWindow), and moved to miwindow.c.

Naturally, if the consortium accepts the concept of restructuring window.c, it would be worthwhile to determine whether any other parts of DIX really belong in DDX.

Making Layered Windows Work

When a client invokes a windowing operation, the following steps must happen. First, determine which windows are affected. Second, calculate the correct clips for these windows. Third, deliver the appropriate exposures to interested clients. Backing store and save unders obviously affect this step. Finally, ensure that input events (mouse, button, etc.) are delivered to the appropriate clients. This logical sequence is mirrored in the sample server. For example, MapWindow calls MarkOverlappedWindows to flag those windows which need to be reclipped. It then calls ValidateTree to calculate the new clip lists, and determine newly exposed areas of the screen. On return from ValidateTree, MapWindow calls HandleExposures to deliver exposure events to interested clients. Finally, back in the event loop, the server determines which window should get mouse events, keyboard input, etc.

This same sequence applies to layered window operations. We will show that solving the first two issues for layered windows largely solves the remaining ones, save for some minor modifications to the sample server. We will also address the other criteria for proper window behavior, namely input distribution, lack of restrictions as concerns the default visual, and protocol compliance for colormaps.

Determining which windows are affected

As with the sample server, we wish to determine the minimum set of windows affected by a given window operation. As we showed above, we can no longer restrict ourselves to the window's parent and its inferiors. Instead, we must first determine the top-most window which could be affected by the window operation. For example, when mapping a window pWin, we would do the following:

- Find the lowest layer N used by pWin and its inferiors. This is the layer which has the greatest ability to damage an ancestor of pWin. This ability is greatest for layer 0: none of the other layers clip it, so that a window in layer 0 will damage an ancestor in the same layer through any intermediate ancestors provided they are all in higher layers.
- Starting with pWin's parent, move up the window tree until we reach an ancestor of pWin which is impervious to windows in layer N. Layer M is impervious to N if a window in layer N cannot damage an ancestor of a window in layer M. In practice, layer M is impervious to layer N iff M <= N. In the process of moving up the window tree, we may run out of levels, in which case we select the root window. The impervious ancestor (or the root window, as the case may be) becomes the layer parent of pWin. By finding N and the resultant layer parent, we are guaranteeing that the effects of the window operation are confined to the layer parent and its inferiors.
- From the layer parent, we can now go back down the window tree, and mark those windows which overlap pWin, in the same way that the sample server does.

Figure 2 shows an example of multi-layered windows. In the diagram showing the window

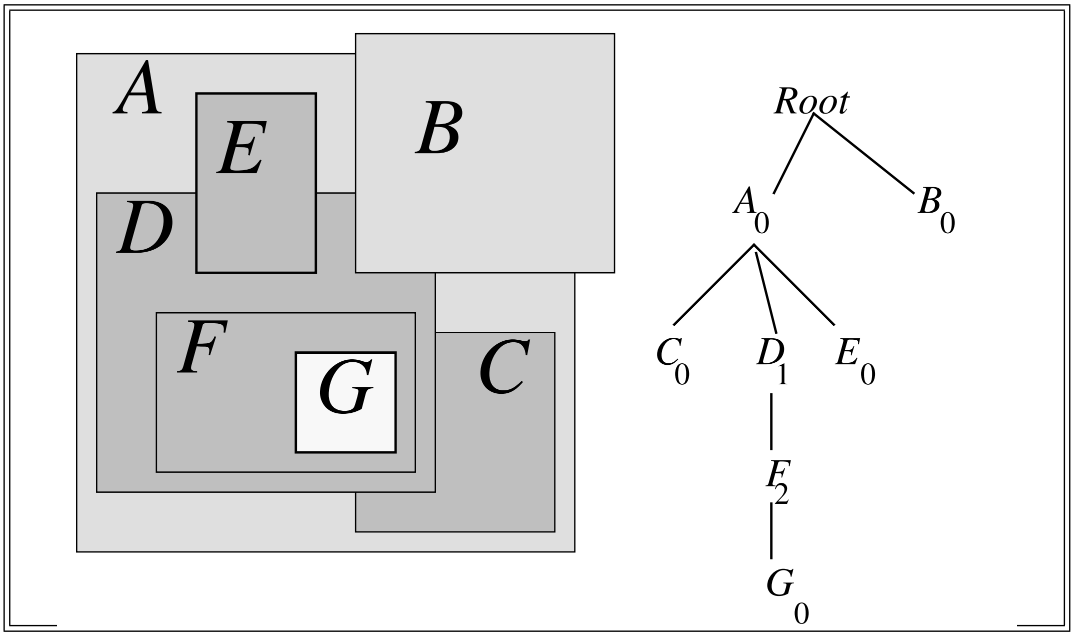

Figure 2: An example of multi-layered windows. Window A is the layer parent for windows C, D, E, F and G. If G were unmapped, F's layer parent would be window D.(subscript = layer number)

hierarchy, the subscripted number denotes each window's layer. Let's assume that we are moving window F. If window G were unmapped, the operation would be confined to windows F and D, because D is in a layer below F, and hence impervious to it. Since window G is mapped, we can see that moving window F (which also causes window G to move) will damage both windows A and C in addition to D, because window D is in a layer above window G's. And in fact, if we follow the steps above, we find that the lowest layer used by window F and its inferiors is layer 0; window D is in layer 1, which is greater than 0, so we check D's parent, A. A is in layer 0, which satisfies the requirement that the layer parent be in a layer impervious to the lowest affected child. When we mark windows for re-clipping, windows A, C, D, F and G will all be marked. Window E is not affected by the operation, because it is higher in the stacking order than its sibling D which is an ancestor of F. Similarly, window B is not affected, because it is a sibling of the layer parent.

Clipping the Marked Windows

Computing the Clip Universes

Now that we have marked all potentially affected windows, we need to clip them. The sample server first gathers together the areas occupied by the marked windows. This creates a clip universe, which represents the total area to be divided among the marked windows. It then walks the marked portion of the tree in both directions: top to bottom, then bottom to top. On the way down, the new border clips are calculated; on the way back up, the new clip lists. This two-pass method is largely dictated by the semantics of window borders. Since a window's border cannot be clipped by an inferior, the new border clip must be calculated (and removed) from the clip universe before inferiors can generate their own sets of clips. This is a perfect example of an initial design (namely, window borders) constraining all future implementations.

In the case of layered windows, the clipping process is necessarily more involved. Windows in layer 1 do not clip windows in layer 0, but windows in layer 0 can clip windows in layer 1. Newman[1] introduced the notion of a Render clip and of a Visibility clip in an attempt to address this issue. The Render clip would describe the area of the window that the client could actually draw to, while the Visibility clip would describe the area of the window visible to the user. It was hoped that judicious use of these two clipping lists would be sufficient to resolve the problem. Unfortunately, this approach breaks down if there are more than two layers in the system.

Instead, for each window, we maintain a clip list for each layer currently in use. We monitor the number of windows currently mapped and realized in each layer. When the first window is mapped in a given layer, all subsequent window operations will result in clips being maintained for that layer, until the last window in that layer is unmapped. The calculation of the clip universe for layer N gathers in the areas occupied by the marked inferiors of the layer parent in layers N and below. In addition, if the window operation is not a stack change, we add a contribution from the layer parent.

Whenever a marked window has a valid clip for layer N, we use that clip. If not, then the window has not participated in an operation which involved layer N since layer N first came into (re-)use. Knowing this, it is fairly straightforward to calculate its contribution to layer N. An inferior of the layer parent only contributes to layer N if it is in layer N or below. The following pseudocode best illustrates the process.

The contribution of pWin to clipUniverse[N] is:

```
if (pWin->layerClip[N].valid) use it;
else if (pWin->layer > N) layer N is not affected by pWin;
else
      for (i = N - 1; i >= pWin->layer; i--)
            if (pWin->layerClip[i].valid)
            {
                  use it;
                  break;
            }
```

From this pseudocode, we can see that pWin will always contribute to the clip universe for layer N if pWin is in a layer below N, because the layer clip is always valid for the window's own layer.

For non stack change operations, the layer parent contributes to the clip universe, regardless of what layer it is in. While the marked inferiors compete for real estate in layer N only if they are in layer N or below, the contribution from the layer parent reflects the amount of unused space available to the marked inferiors. To evaluate that contribution, we check if it has a valid clip for layer N, in which case we just use that clip. If it is in a layer below N, we are once again guaranteed that one of the layer clips from layer N-1 to its layer is valid, since the layer clip is always valid for the window's own layer. If it is in a layer above N, it can only be the root window, given our marking algorithm. In which case we examine the layer clips from N-1 down to layer 0. If any of them is valid, we use it, and we're done. If none of them are valid, there are no windows in layer N or below that affect the current window operation. In which case, we just add in the screen size to the clip universe. The following pseudocode sums up these concepts.

The contribution of layerParent to clipUniverse[N] is:

```
if (layerParent->layerClip[N].valid) use it;
else
{
      if (layerParent->layer < N)
            for (i = N - 1; i >= layerParent->layer; i--)
                  if (layerParent->layerClip[i].valid)
                  {
                        use it;
                        break;
                  }
      else        /* parent can only be root */
      {
            foundLayer = FALSE;
            for (i = N - 1; i >= 0; i--)
                  if (layerParent->layerClip[i].valid)
                  {
                        use it;
                        foundLayer = TRUE;
                        break;
                  }
            if (!foundLayer) add layerParent->winSize to clipUniverse;
      }
}
```

Once layer N is in use, it becomes necessary for each window to maintain a clip list for that layer, even if no windows in layer N are involved in the current window operation, because it may be impossible to recompute it based purely on the known clip in other layers, and because the clip information could otherwise become stale.

Figure 3 shows a typical example illustrating these issues. Windows B and C are in layer 1, window

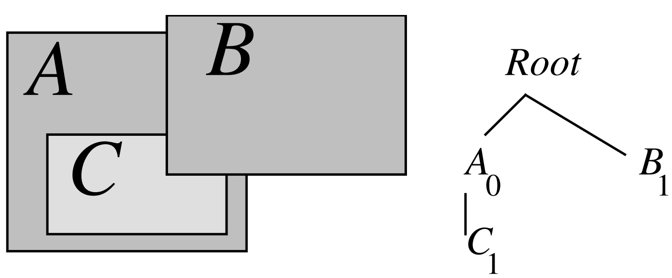

Figure 3: Illustrating the need for maintaining separate layer clips. (subscript = layer number)

A is in layer 0. Thus, A is clipped by neither B nor C. Imagine that we first unmap C, then move window A slightly, keeping it beneath window B. Only window A and the root window will be marked for this operation. A is marked because it is the window being moved. The root window is marked because moving A causes damage to the root, no matter what layer the root window is in, since A is in the lowest layer in the system. C is unmarked because it is unmapped, B because it is a sibling of A above it in the stacking order. Let's assume that, because no windows in layer 1 are involved in the operation, that we do not update A's clip for layer 1. If we now map window C, we have two choices, neither of which will generate the correct result. If we use the old value for A's clip in layer 1 as the basis for the new clip universe, C will be clipped incorrectly because the old clip region will not have been translated when A was moved. If instead we attempt to calculate the clip universe for layer 1 starting anew, we will get the wrong result, because A is unclipped by B (since B is in a layer above A), B is not a marked window (since B is above A in the stacking order), so that the resultant clip universe would include areas that are in reality overlapped by B. Thus, the only way to clip C properly is to maintain A's clip for layer 1, even when it would seem that no windows in layer 1 are involved in the current operation. Note that if both B and C were unmapped prior to moving A, then remapped following the move, it would be possible to compute a clip universe for layer 1 based on the known clips in layer 0, because we would be guaranteed that no window in layer 1 was obscuring window A during the move.

This example illustrates why clips are maintained for a given layer as long as any window in that layer is currently mapped and realized. This is certainly not the only way to solve this problem. We chose only to mark siblings that are lower in the stacking order than the layer parent, in a direct extension of the marking algorithm in the sample server. If instead we had chosen to mark all windows that overlap a portion of the active window (i.e., the window being moved, resized, etc.) regardless of stacking order, then we would have been assured that all affected layers were fully represented for the current window operation. But the only way to guarantee that all overlapping windows are marked would be to traverse the entire window tree, starting with the root window. In the simple case, this approach would involve fewer calculations. But in the complex case, when the window tree is particularly deep and intricate, this approach would result in many more windows being checked, and potentially in many more windows being marked.

Calculating the New Clip Lists

An accurate determination of the clip universe for each layer in use is critical to the clipping process. We have seen that it requires non-trivial changes to the original algorithm. In comparison, calculating the new clip lists is a much simpler extension of the sample server. We traverse the marked subtree from top to bottom and bottom to top just as the sample server does. But when we remove a window's extents from its clip universe to deny the space to other windows, we also remove those extents from the clip universes in layers above it, since once again a window clips layers above its own.

Some of the optimizations in the original clipping code cannot be extended for layered windows. For example, if none of the children of a window overlap, they do not interfere with each other, so that it is not necessary to remove their extents from the clip universe after each is clipped. Instead, it is faster to remove their union from the clip universe after they have all been processed than it is to subtract them one by one at the time that they are clipped. In a layered system, if a window has inferiors in a different layer, we cannot use this optimization, because the various extents will be removed from different clip universes.

Exposure Handling, Backing Store and Save Unders

Newman[1] reported problems with handling exposure events and providing proper backing store support. These problems resulted largely from the limitations of his clipping model. The calculation of exposed areas, and the support of backing store both depend critically (but trivially) on accurate clip calculations. We have solved the clipping problem by identifying the full set of affected windows, and by maintaining separate clip lists for all layers in use. Having achieved this, there are no further changes required within the sample server to support backing store for X clients, and to deliver proper and accurate expose events to interested clients. Since exposures and backing store events are intimately linked to the rendering process, it is clear that only the clip within a window's own layer is relevant to this process.

Save unders do require modifications to the server in a layered environment, however. The sample implementation supports save unders by turning on backing store for windows about to be obscured. As an optimization, the sample implementation restricts itself to examining the parent and lower siblings of the window effecting the window operation. This is insufficient in a layered system, where once again we need to start out with the layer parent. In figure 2, if window G has the SaveUnder attribute, we need to turn on backing store for windows A, C, D and F, where the sample implementation would restrict itself to turning on backing store for window F only. On the other hand, if the window about to be obscured is in a layer below the obscuring window's, the obscured window is unclipped, and there is no need to enable backing store for that case.

Layered Windows and Gravity

Resizing windows with gravity results in a large number of region operations in order to maintain the correct clip lists, and calculate the appropriate expose events. If the window being resized has inferiors in different layers, tracking all of the changes would require many times more region operations, costing both in code complexity and performance. For these reasons, we have chosen to limit our support of gravity. If the window being resized has inferiors in different layers, we ignore the gravity setting, and deliver expose events to clients as if the gravity were set to ForgetGravity.

Layered Windows and Input Distribution

Input distribution falls out naturally from our design. The sample server translates an x, y pointer coordinate to a window ID by walking the window tree from top to bottom. Starting with the children of root, it restricts itself to the branch which includes the pointer coordinate, stopping when it has reached the bottom-most window which contains the point. This window is then the window closest to the viewer from a visual standpoint to contain the point. Since we maintain a single window tree, this algorithm requires no modifications for layered windows.

Note that the top-most window could be transparent, visually exposing to the user windows beneath it. Input events will still be sent to the transparent window, however, because transparency is a property of the pixel, not of the window.

Layered Windows and the Default Visual

There is no architectural limitation in our model as regards placing the default visual in any of the layers in the system. In practice, some additional support is required if the root window is in a layer which supports transparency. Imagine that the root window is in layer 1, and that the default background is the transparent pixel. We now unmap a child of the root window which is in layer 0. The area that was occupied by the child now reverts to the root window (assuming that it did not overlap any other windows). The server will then clear layer 1 to root's background, in this case the transparent pixel. But it must also clear or otherwise disable the display of the pixels in layer 0 where the window used to be. Fortunately, our server already has to follow window tree validation by a validation of the window visuals, in which we update the hardware's knowledge of the visual associated with a given pixel. It is relatively simple to add to this step the clearing/disabling of layers beneath the root window.

Transparency and Colormaps

Visuals which support a transparent pixel do not allow the allocation of a color cell for the transparent pixel value. This is accomplished by having the server mark the cell as read only. This concept is readily extended for a transparent mask.

Conclusions

At the heart of our work lies the conviction that layered windows are windows like any other, save for two properties. The first, that they do not clip windows in layers beneath them. The second, which is ignored by the server except for colormaps, is the concept of transparent pixels/masks.

If we accept this initial premise, it follows logically that the DIX windowing code is built on hardware-dependent assumptions. We propose a restructuring of the windowing code within the sample server, moving the bulk of it to DDX/MI. While most implementations would still use the sample code, this would allow easier porting to more sophisticated architectures.

We have shown that we can solve the clipping of layered windows by judiciously extending the notion of which windows are affected by the current operation, and by maintaining separate clip lists for each active layer. Other issues fall out naturally from accurate clipping, e.g., exposure handling, input handling, etc.

Our implementation, although fairly general, is driven by the design of current hardware platforms. One can speculate on layered architectures where our solution would either be inadequate, or on the other hand, overly complex. But the proposed restructuring of the sample server would allow developers of new architectures to tailor the windowing code for their needs with less effort.

Acknowledgments

I profited greatly from the experience gained by Todd Newman in his initial implementation of layered windows. By exposing the various traps and pitfalls, he enabled me to focus rapidly on a solution to the problem, without having to spend much time defining the problem.

I am grateful to Luther Kitahata and Chandlee Harrell, my immediate management, for recognizing the importance of solving the layered problem, and allowing me the time to do so.

References

1. Newman, T., "How Not to Implement Overlays in X", *The X Resource*, Issue 1, pp. 49-60, Winter 1992.

Author Information

Peter Daifuku joined Silicon Graphics in March 1991, where he first worked in the core X group. He is currently a member of the Interactive Systems Division X/Graphics Kernel group, where he works on porting and supporting the X server for new platforms when not occupied with layered windows. Prior to SGI, he spent three years at Apollo Computer, Inc., and three years before that at Raster Technologies.