Optimized Window Refreshing

by Martin Taillefer

Maintaining the graphical contents of an Amiga window can be difficult. There are many subtleties associated with the process known as window refreshing. At present, many applications refresh their windows in suboptimal ways, or fail to refresh correctly under all conditions. This article attempts to explore and resolve the window refreshing problems commonly encountered by applications.

Damaging Information

Although many people think of Intuition as the Amiga's windowing system, the lower-level Layers library actually handles most of the work involved in maintaining Intuition's windowed environment. The Layers library divides a single physical display (a BitMap) into multiple virtual displays. Each of these virtual displays is known as a layer. Intuition uses the functions in the Layers library to move, resize, and depth arrange these layers.

Normally, each Intuition window consists of a single layer. Intuition adds borders and gadgetry to the layer to give it the familiar Intuition window appearance. Intuition also takes care of monitoring user input to let the user move, resize, and depth arrange windows. When Intuition wishes to change the size or position of a window, it calls functions in the Layers library to do most of the grunt work.

One of the main reasons for a layered rendering system is to provide multiple independent logical rendering regions to applications. The layer is the basis for the Intuition Window. The functions in the Layers library allow several applications to render into the same physical display (an Intuition Screen for example) without worrying about interfering with each other. As far as the application is concerned, it has an entire display all to itself.

There is a limit to the isolation that this layered environment offers applications. Because layers can overlap, by moving, resizing, or deleting a layer, a program can uncover portions of underlying layers. These newly exposed portions are called ‘‘damage regions’’. A layer can sustain damage when a task performs layers operations on it or other layers in the same display. When the Layers library damages a layer, it sets the layer’s LAYERREFRESH bit in the Flags field of the Layer structure.

When damage occurs to a layer, the damaged region of the layer must be repaired by redrawing it. The entity responsible for redrawing the damaged region depends on the type of layer. The Layers library offers three types of layers: simple refresh, smart refresh, and superbitmap. Subsequently, Intuition bases its three types of windows on these three types of layers. The difference between each type has to do with the way in which each handles damage repair.

When a layers operation damages a simple refresh layer, the entire burden of repairing the layer’s damage rests on the application that created the layer (if the application is using Intuition windows, which are built on top of layers, Intuition shares the burden of damage repair with the application. More on this later). This is because a simple refresh layer does not preserve its contents. The advantage of this type of layer is that it doesn’t use much memory. The disadvantage is that every time a layers operation reveals a portion of a simple refresh layer, the application must explicitly rerender the exposed damaged regions.

Smart refresh layers help the application by doing some of the refreshing automatically. If a layer operation conceals a portion of a layer, that portion is automatically copied to an off-screen buffer. If a layer operation later reveals that portion of the layer, the Layers library uses the temporary buffer to update the revealed region. The Layers library does not leave the LAYERREFRESH bit set in this case because the Layers library took care of the damage. The application still needs to refresh the smart refresh layer whenever it is made larger, as the Layers library has no idea what should be in the newly exposed areas. In this case, the Layers library will leave the LAYERREFRESH bit set.

Finally, superbitmap layers totally eliminate the need for an application to refresh the display. The Layers library maintains a complete off-screen buffer representing the layer’s contents. When a layer operation exposes new portions of a layer, the Layers library automatically updates these regions by copying from the off-screen buffer. The Layers library will never leave the LAYERREFRESH bit set for a superbitmap layer.

For the application programmer, a superbitmap layer offers the simplest approach to refreshing the layer as the entire burden of repairing damaged regions falls on the Layers library. This added convenience does have a cost--because of its off-screen buffer, the superbitmap layer requires a significantly larger amount of memory compared to the simple and smart refresh layers. The Layers library allocates this memory even if the layer never sustains damage.

When To Refresh

The way in which an application determines that it needs to refresh a layer depends on whether an application deals with layers directly via the Layers library, or indirectly via Intuition. It is important to keep in mind that each Intuition window has a Layer at its core. If an application uses Intuition windows it may not use the Layers library to create, delete, move, resize, or update the window layers. The application must use the corresponding Intuition functions instead.

If an application uses layers directly, it must look at a layer's `LAYERREFRESH` bit to tell if the layer sustained damage and needs repair. Since the application created and maintains its layers, it knows when damage can occur, so it checks for damage at those times.

Intuition manages an arbitrary number of layers on behalf of any number of client applications. All windowing operations happen on Intuition's time frame. Since other applications and the user can ask Intuition to manipulate window layers around at any time, it is not possible for a window-based application to know by itself when to refresh its window. That would involve polling the window layer's `LAYERREFRESH` bit, which is a big no-no in a multitasking system. Instead, Intuition provides a refresh notification mechanism through the IDCMP system.

Whenever Intuition performs a layer operation that can damage a window layer, it checks the damage state of each layer in the current screen by inspecting each layer's `LAYERREFRESH` bit. If Intuition finds that a layer's `LAYERREFRESH` bit is set, Intuition takes care of refreshing the damaged areas of the window that are Intuition's responsibility (for example, the window borders and gadgets). Following that, Intuition looks at the `WFLG_NOCAREREFRESH` bit in the Window structure. If this bit is set, Intuition's refresh processing for the window is complete, and all damage region information is discarded. However, If the bit is clear, Intuition sends an `IDCMP_REFRESHWINDOW` message to the window's IDCMP port, essentially asking the window to refresh itself.

So, instead of having to poll the `LAYERREFRESH` bit, an application can just wait for Intuition to tell it to refresh its window. This is convenient and fits in well with the IDCMP mechanism.

Scrolling Your Life Away

A scrolling display is a part of many user interfaces. An easy way to make people think an application is slow is to give it a sluggish user interface. Regardless of the actual speed at which an application performs its job, if its user interface management is slow, the user will get the impression the whole application is slow.

The ScrollRaster() and ClipBlit() routines are the two principal ways of scrolling data within a layer. ScrollRaster() moves the data within a RastPort. ClipBlit() moves data from one RastPort to the next, but works equally well if the source and destination RastPorts are the same.

Smart refresh layers are relatively easy to scroll with either ScrollRaster() or ClipBlit(). Although ClipBlit() has a theoretical advantage over ScrollRaster(), the performance of these functions is generally equivalent. ScrollRaster() does a minimum of two Blitter operations: one to scroll the data, and a generally smaller one to erase the area that was scrolled away. ClipBlit() can do a single blit to move the data in the layer, and leaves the rest of the display alone. Thus, while scrolling large amounts of data, an application can get better CPU/Blitter processing overlap by using ClipBlit(). Consider the following pseudo-code of what happens in ScrollRaster() versus ClipBlit():

```
Within the ScrollRaster() function:
    Wait for any outstanding Blitter operation to complete
    Initiate a Blitter operation to move the data in the window
    Wait for the Blitter operation to complete
    Initiate a second small Blitter operation to clear the scrolled region
    Return to the caller while the small blit is still in progress

Within the ClipBlit() function:
    Wait for any outstanding Blitter operation to complete
    Initiate a Blitter operation to move the data in the window
    Return to the caller while the blit is still in progress
```

Since the Blitter and the CPU can run concurrently, the second approach provides the most throughput, as the Blitter performs most of the scrolling work *at the same time* the application code runs after ClipBlit(). In practice though, ScrollRaster() and ClipBlit() run at about the same speed.

For a smart refresh window, ScrollRaster() and ClipBlit() do not present a problem. As long as the application only scrolls the contents of the window and not the window borders, the Layers library will take care of fixing any damage. Intuition will never know what happened. For a simple refresh window however, there is a problem. The reason has to do with the following:

```
+--------------+          +--------------+
|aaaaaaaaaaaaaa|          |bbbbaaaaaabbbb|
|bbbb+----+bbbb|          |cccc+----+cccc|
|cccc|    |cccc|          |dddd|    |dddd|
|dddd+----+dddd|          |eeee+----+eeee|
|eeeeeeeeeeeeee|          |              |
+--------------+          +--------------+
```

Figure 1a **Figure 1b**

Figure 1a represents a small window that the user placed on top of a larger simple refresh window. When an application scrolls the larger window's contents upwards with ScrollRaster(), the result looks like Figure 1b. This operation exposes two portions of the larger window, the section at the bottom of the large window (the part ScrollRaster() clears) and also the area above the smaller window (which ScrollRaster() leaves intact).

The application that owns the larger window has to refresh the two portions exposed by ScrollRaster(). The application knows to refresh the bottom section of the window because the damage is a direct result of the application calling ScrollRaster(). The problem is the area above the smaller window. The application does not know that there is a window overlapping its larger window, so it does not directly know about any damage resulting from other overlapping layers. Because the application used a graphics.library function to manipulate the window, Intuition does not know about the damage either. The result is that the larger window is damaged and no one knows it.

If the larger window had been a smart refresh window, the Layers library would have cached this portion of the larger window and ScrollRaster() would have taken action to make sure the cached region got restored. For a simple refresh window, this portion only gets added to the window layer's damage list and no rendering actually occurs.

The solution to this problem is for the application to act as Intuition would if it were scrolling. When Intuition manipulates a layer, it checks the LAYERREFRESH bit of each of its layers to see if any of them was damaged. After the application calls ScrollRaster() on its window, the application has to look at the LAYERREFRESH bit of the window's layer. If the bit is set, damage exists and the application needs to repair the window. As this window is the only window that can sustain layer damage as a result of the call to ScrollRaster(), the application needs to check only its own window's layer for damage.

Note that to scroll the contents of a simple refresh window, an application has to use ScrollRaster() rather than ClipBlit(). The reason is ClipBlit() does not add anything to the window layer's damage region, so an application will never know about the damage. Also, unlike ScrollRaster(), ClipBlit() does not scroll the window layer's damage region. If the layer already has damage when an application calls ScrollRaster(), the position of the damage region will move along with the data in the layer. Note that prior to Release 2, ScrollRaster() did not scroll the layer's damage list.

Faster Rendering

When rendering to a RastPort, an application can improve its rendering performance if it renders using a limited number of colors. Each RastPort contains a mask field which specifies the writeable bitplanes of the RastPort's BitMap. If a RastPort's bitplane is write-protected, the system ignores that bitplane when rendering to the RastPort.

An application can control the BitMap mask by using the SetWrMsk() macro (defined in *<graphics/gfxmacros.h>*). This macro accepts a pointer to a RastPort, and a new mask. The mask is an 8 bit value. Each bit of the value represents a BitMap bitplane. For example, a mask of value 5, which is 00000101 in binary, restricts rendering in that BitMap to planes 0 and 2 which are the only two bits set in the binary value.

One type of application that can improve its performance by using the RastPort Mask is the text editor. Typically, a text editor needs to render its text in a single color, generally color 1. This means the editor only needs to render to bitplane 0. All other planes will always remain blank. If these planes are going to remain blank, why should the editor bother to render to them or scroll them?

When scrolling large sections of the display, ScrollRaster() can make the display look unstable because it continuously clears portions of the display. ClipBlit() eliminates this visual nuisance as it does not clear the display. Unfortunately, ClipBlit() by itself is useless for scrolling a simple refresh window because it does not deal with damage regions. The solution is sneaky but quite effective. An application can do the following each time it scrolls a simple refresh layer:

Set the RastPort Mask to limit the number of writeable bitplanes.
Use ClipBlit() to scroll the data.
Set the RastPort Mask to 0.
Use ScrollRaster() to scroll the same data.

In the procedure above, ClipBlit() scrolls the window contents but not the damage regions. ScrollRaster() scrolls the damage region, but because the RastPort mask is 0, ScrollRaster() does not affect the window contents. Setting the RastPort mask to 0 prevents the system from disturbing the data in any of the planes of the BitMap.

The above trick can come in quite handy. It is fairly fast as well, although it can involve some hidden overhead. Even though ScrollRaster() doesn't move data on the display, it still needs to go through all the layer's clipping regions, which can be time consuming.

Using Multiple RastPorts

A RastPort specifies attributes needed to perform many rendering operations. These include the RastPort's foreground, background, and drawing mode. The Graphics library functions SetAPen(), SetBPen(), and SetDrMd() set each of these attributes, respectively.

Although these functions appear to have fairly simple purposes in life, the SetAPen(), SetBPen(), and SetDrMd() are quite CPU intensive routines. These functions require recalculating values that the OS caches in private parts of the RastPort. If an application only requires a few different combinations of foreground/background/draw mode, it can improve its performance by using a different RastPort structure for each combination. An application sets the attributes of several RastPorts only once which is generally more efficient that setting the attributes of one RastPort every time the rendering attributes change.

Assume an application has a window in which it renders all of its data in pen 1, and clears any part of its display using color 0. Such an application can improve its rendering performance by doing the following:

```
struct RastPort dataRP;
struct RastPort clrRP;

dataRP = *window->RPort;
SetAPen(&dataRP,1);
SetBPen(&dataRP,0);

clrRP = *window->RPort;
SetAPen(&clrRP,0);

/* renders to the window’s RastPort in color 1 */
Text(&dataRP,"hello",5);

/* clears a section of the window’s RastPort in color 0 */
RectFill(&clrRPort,0,0,10,10);
```

Refreshing a Sizable Window

A sizable window is by far the trickiest to refresh correctly. The most important and often overlooked point is that an application needs to ensure the size of its window does not change while it is refreshing the window. If the application doesn’t and the user changes the window size, the application will refresh the window at the window’s old size rather than its new size. This can severely corrupt the appearance of the window.

There are two main ways an application can keep the size of its window stable while refreshing the window. The first method is to keep the size of the window locked most of the time, unlocking the window only when the user tries to size the window. The other approach is to lock the window size only while rendering to the window.

The first method is part of Intuition’s IDCMP mechanism. To lock a window’s size, an application sets the window’s `IDCMP_SIZEVERIFY` IDCMP flag. When the user attempts to size the window by clicking on the window’s sizing gadget, Intuition notifies the application by sending an `IDCMP_SIZEVERIFY` message to the window’s IDCMP port. Intuition will keep the window’s size locked until the application returns the `IDCMP_SIZEVERIFY` message.

In general this scheme works well, but it does have two problems. First, if the application is busy doing some processing, such as recalculating a spreadsheet, it may not notice that the message arrived until it is done with its current processing. The result is the user will not be able to size the window until the application is finished processing, which might more time than the user wants.

The second problem occurs when the application is waiting for input from Intuition, such as in the middle of an EasyRequest() call. If the user clicks on the sizing gadget of the application's window while the requester is up, Intuition will wait for the system will enter a deadlock. Intuition will wait for the application to send back will not see the IDCMP_SIZEVERIFY event until the user satisfies the EasyRequest(). The result will be a system deadlock. Many applications suffer from this problem.

The second problem occurs when the application is waiting for input from Intuition, such as in the middle of an EasyRequest() call. If the user clicks on the sizing gadget of the application's window while the requester is up, the system will enter a deadlock. When the user clicks the sizing gadget, Intuition sends the IDCMP_SIZEVERIFY message and waits for a reply. Because the application is already waiting for the EasyRequest() to return, the application cannot send back the reply. Many applications suffer from this problem.

As of Release 2, Intuition has adapted and avoids these deadlocks. Intuition will time out the sizing operation if the application does not process the IDCMP_SIZEVERIFY message within a given time period. Although the user can no longer deadlock the system, this situation can still confuse the user because clicking on the window's sizing gadget no longer sizes the window. That does not mean an application should rely upon Intuition to avoid the deadlock. An application should always avoid these conditions.

In general, it is simpler and safer for the application to lock a window only during the rendering process. The application can do this by surrounding all rendering operations with calls to the Layers library functions LockLayer() and UnlockLayer(). LockLayer() locks the size and position of a layer. While a window's layer is locked, an application can safely look at the window's current size and render to it without any danger of the size changing. Once the application finishes rendering, it unlocks the window's layer by calling UnlockLayer(). When using this method, an application must not set the window's IDCMP_SIZEVERIFY bit. Be careful which system functions an application calls while it has a layer locked. Only use the Graphics library rendering functions and the simple Intuition rendering functions (i.e., PrintIText(), DrawImage(), etc.). In particular, avoid calls that deal with gadgets (including RefreshGList()) and other locks (i.e., LockIBase() and LockLayerInfo()).

BeginRefresh() and EndRefresh()

To improve the performance of repairing damage to simple and smart refresh windows, Intuition can put a window into a special refresh state using Intuition's BeginRefresh() function:

```
VOID BeginRefresh(struct Window *window)
```

While a window is in this state, the system restricts attempts to render into the window to the window's damaged regions. Because the system ignores rendering operations outside the window's damage regions, an application only refreshes the parts of a window that need refreshing. This can significant decrease the amount of time necessary to refresh the window. This also reduces the possibility of visual flicker that can happen if an application has to redraw the entire contents of a window.

To end a window's special refresh state, use EndRefresh():

```
VOID EndRefresh(struct Window *window, BOOL Complete)
```

Compared to BeginRefresh(), EndRefresh() has an extra parameter, a boolean value. If this value is `TRUE`, Intuition assumes that all of the refreshing for this window is finished and removes all of the window layer's damaged regions. In most cases, this value should be `TRUE`. See the Autodoc for BeginRefresh() and EndRefresh() for more information.

Backfill Hook

Whenever a layer sustains damage, the damaged region is cleared to color 0. Clearing the region requires an often lengthy Blitter operation which is unnecessary if the application plans to redraw the damaged region anyway. Such an application can improve its performance by preventing the Layers library from clearing the damaged regions. This is what layer backfill hooks are for.

Backfill hooks are a new Layers library feature added in Release 2. A backfill hook is a custom function that an application attaches to a specific layer. Whenever damage occurs to a layer, the Layers library calls that layer's custom backfill hook. The Layers library passes the position and dimensions of the damage area to the backfill hook. The backfill hook can render into the damage area, refreshing it. Theoretically, the backfill hook can redraw a layer's damaged regions instead of clearing the damage area to color 0.

Unfortunately, refreshing a layer from a backfill hook can be quite difficult. The backfill hook code is usually called from a different task than the main application. If the backfill hook and the application do not properly arbitrate access to the application's data, dangerous race conditions can occur if the hook and the application try to access the same data. Another problem with rendering through the backfill hook is that no clipping is available. The backfill hook must do its own clipping to ensure that no rendering goes outside the dimensions specified when the hook is called, which in itself can be quite difficult.

For many situations, the best use of a backfill hook is to have it do nothing. Whenever the hook is called, it simply returns without doing any rendering. This has the effect of eliminating the extra blit done to clear the damage region to color 0. This can speed layer operations quite a bit.

There is one problem with a no-op backfill hook. If the application is busy doing some processing and damage occurs to its layer, the display will remain dirty until the application finishes its processing and notices that the layer is damaged. This damage can include remnants of system imagery, like window borders, which can confuse the user.

One way to overcome this problem is to use a backfill hook that changes its behavior depending on the state of the application. If the application is not too busy to notice damage to its window's layer, the backfill hook does not erase the display. However, if the application is too busy to refresh its damaged window, the backfill hook clears the damaged portion of the display. Using this method, the backfill hook will not waste time erasing the damage when the application will update it immediately, but the backfill hook will erase the damage if the application can't refresh for a while.