Improving Response Time

Users are very sensitive to response time. Even a slight delay in providing users with feedback to their actions will give the impression of sluggishness regardless of how ªfastº your application really is. Therefore, it's worth paying attention to response time and looking for ways to improve it. The discussion that follows provides a few techniques for improving real and perceived response time.

Optimal Panel and Menu Updating

It's often the case that you must update the state of items contained within menus and panels to reflect the user actions within the application. This may need to be done after every event. How you perform this update can have an important impact on the responsiveness of your application. The key is to avoid unnecessary drawing, particularly in menus and panels that aren't visible to the user. After all, why update a menu or panel that isn't visible? The Application Kit has a built-in mechanism for allowing you to handle updates efficiently. We encourage you to take advantage of it.

The key idea is that windows and menus have an **update** method that's responsible for handling updates. This method is invoked in the following situations:

- Automatically when a window is about to be brought on-screen.
- Whenever the **updateWindows** message is sent to NXApp, **updateWindows** will send the **update** message to every on-screen window. If you send a **setAutoupdate:YES** message to NXApp, **updateWindows** will be sent after every event.
- Whenever a Command-key event is processed, the **update** message is sent to each panel and menu before the panel or window sends the **performKeyEquivalent:** message down its view hierarchy. Note that in this case the panel or menu may not be visible, but the update is necessary so you can disable or enable things

that may respond to the key equivalent.

While the default **update** method of Window and Panel does nothing but return **self**, the **update** method of Menu has been implemented to allow you to easily update menu items in response to an **update** message. To take advantage of this, you must do two things:

- The **setUpdateAction:forMenu:** message needs to be sent to each menu item that will need to be updated, passing the method you want invoked when the menu containing the menu item receives an **update** message.
- NXApp or the delegate of the menu must implement the method specified above. The method must take a single argument which is the **id** of the menu cell, and return YES if the cell needs to be redisplayed, or NO if not.

Here's a simple example of an **updateAction**:

```
-updateIt:(id)menucell
{
    if([menucell isEnabled]!=enabledFlag){
        [menucell setEnabled:enabledFlag];
        return YES;
    } else
        return NO;
}
```

When the menu's **update** method is invoked, it first disables **flushWindow**, and then goes through its item list. For each menu cell for which an **updateAction** has been specified, it disables **display**, invokes the update action, and if the update action returns YES, **display** is reenabled and the cell is redrawn. If at least one menu cell has been redrawn, the menu is flushed. It also does a number of other things to ensure optimal redraw.

Since a menu's **update** method is invoked only when the menu is visible, is about to become visible, or is about to process a key equivalent, you can be assured that the menu item is updated only when absolutely necessary, and in the most efficient manner possible. Once again, this is an example of taking a lazy approach.

The documentation also discusses a means of disabling display while updating either multiple views or multiple items within a view. This can be convenient when you want to update a bunch of controls or other views with

new values, and then redisplay the ones that have been changed. See the Window's **disableDisplay** and **displayIfNeeded** methods.

Optimal Window Flushing

When using buffered windows, you may want to think about your strategy for flushing the window. All drawing in a buffered window is done off-screen in the backing store for the window. The Window Server maintains a ªdirty rectº that represents a rectangle which encompasses all the bits that have been touched in backing store since the last time the window was flushed. Only the area of the window contained in the dirty rect is flushed when the window is asked to flush itself. It's only when the dirty rect is flushed on-screen that the user sees the new drawing. Flushing is performed automatically once a view and its subviews have been drawn, if you use the **display/drawSelf::** mechanism provided by the View class. However, if you use **lockFocus** and **unlockFocus**, you must explicitly flush the window. In either case, here are some things to think about in determining your flush strategy.

It may be desirable to flush the window at intermediate points if drawing the view (or the views) is going to take some time. For example, if you're doing a page layout system and are paginating a page, you may want to flush the window after every line, rather than wait until the entire page is done. While the overall time may be no faster, the users' perception will be that it's faster, because they can see that things are happening. This can be a very powerful technique for enhancing perceived performance.

You may also want to do intermediate flushes if you're doing localized drawing in opposite corners of your window. If you're doing localized drawing in widely different places in the window, the dirty rect will encompass many bits that haven't changed and don't need to be flushed. In this case, you might be better off doing intermediate flushes, rather than waiting and doing a single flush at the end of all your drawing.

However, if you're updating several views within a window, and they aren't in the same branch of the hierarchy *and* you don't want the user to see intermediate results (one view updated, another not), then you should send the enclosing window the **disableFlushWindow** message before drawing. At the end of drawing, send it the **reenableFlushWindow** and **flushWindow** messages to actually flush the window. The advantage of this

approach is that the user doesn't see intermediate results. The disadvantage is that if the draw time is significant, users may be waiting for a period of time to see the results of their actions.

As an alternative to intermediate flushes, you can also use the **setwindowtype** PostScript operator to temporarily change the window from a buffered to a retained window. This will allow the user to see the progress of the drawing as it happens. Afterwards, change the window type back to buffered so that controls and title bars don't draw with ugly flashing.

Coalescing Key Events

Generally speaking, the Text object should be used for handling and drawing text, but if you're responding to key-down events yourself and doing drawing in response, it's a good idea to coalesce key events as the Text object does automatically. By this, we mean that when you respond to key-down events, check whether there are any more such events in the queue, and if there are, pull them off before responding to the event so you can process them all at the same time. The Draw demo application does this with the arrow keys (that is, if Draw can't keep up with the rate at which you're moving the selection via the arrow keys, it'll combine as many of the moves as it can into one, bigger move). Note that the system does this for you in the case of mouse-dragged events.

A related idea is to periodically check within your drawing code to see whether another event has come in. For example, if the user types a character in the middle of a line that causes the rest of the paragraph to be rewrapped, then as the program goes off to rewrap that text, it can periodically check to see whether the user has typed another character, at which point the program can abort rewrapping and start wrapping from the insertion point again.

Both of these techniques can contribute to the user's perception of a more responsive application, even if the absolute performance for an individual redraw isn't changed.

PSobscurecursor()

If you are doing your own text processing, when the user begins typing use **PSobscurecursor()** to hide the cursor rather than **PShidecursor()** and **PSshowcursor()**. The response latency of will be better, since the

Window Server will make the cursor appear the moment the user moves the mouse, without requiring the intervention of the application.

NXPing()

Communication between the Window Server and the application is buffered at both ends. While this is done for efficiency reasons, it can cause unwanted latency, particularly when PostScript code is sent to the Server at a faster rate than the Server can consume it. **NXPing()** can be used to flush the buffers on both sides of the connection. Specifically, **NXPing()** waits until all PostScript code thus far generated has been executed. Since **NXPing()** involves a round trip to the Server, its use lowers overall throughput, but in certain cases its use can reduce latency and thus improve perceived performance. If you're doing a minimal operation in a tracking loop, like rubber-banding out a rectangle in response to a mouse-dragged event, **NXPing()** is unnecessary, as the Server can keep up with the volume of PostScript code being sent to it on each iteration. However, if you're generating a lot of PostScript code on each iteration, the Server may still be working on the PostScript code from the last iteration when the application starts sending it PostScript code from the next iteration. Using **NXPing()** in this case can help keep the application and the Server in synch. Note that any operator that involves a round trip to the Server will have the same effect as **NXPing()**.