View Tips

DRAFT 3

Michael S. Engber
Apple Computer - Newton ToolBox Group

This article was (will be) published in the May 1994 issue of PIE Developers magazine. For information about PIE Developers, contact Creative Digital Systems at CDS.SEM@APPLELINK.APPLE.COM or 415.621.4252.

Introduction

This article discusses a few techniques for optimizing your view drawing performance. Often, applications are slowed down because of unnecessary redrawing of their views. Some insight into how the view system handles updates can be a big help in preventing this problem. I assume the reader has some experience using NTK to write Newton applications.

Note that this article is based on details of the view system implementation which could change in future products. This should not discourage you from using the techniques presented. They won't break in future products – it's just that improvements to the view system system may eventually make these techniques unnecessary.

Dirtying Views

When you dirty a view, normally as the result of calling SetValue, it is not redrawn immediately. Instead, the view system keeps track of what has been dirtied and periodically performs updates to the screen. The view system tries to optimize these updates by keeping track of the enclosing rectangle of all the dirtied views and only redrawing views which intersect this rectangle. I call this rectangle the update rectangle.

In addition, the view system also keeps track of a view which it uses to search for views to redraw for the update. I call this view the update view. When an update occurs, the view system starts from the update view, examining it and its descendants, looking for views that intersect the update rectangle, and therefore need to be redrawn.

Note that becoming the update view is not the same as being dirtied. The update view only affects where the search for views to update begins. Contrast this with dirtying a view, which forces the view and all its descendants to be redrawn.

Because only a single update view is maintained, when a view is dirtied the view system finds the closest common ancestor of the update view and the newly dirtied view and makes this ancestor the new update view. This has some interesting implications. Consider the simple view hierarchy shown in Figure 1.

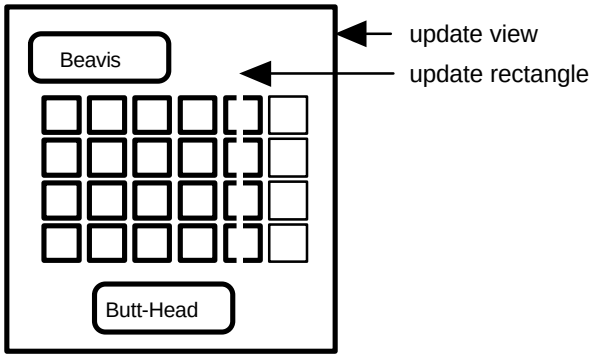

Figure 1 - Beavis & Butt-Head Update Scenario I

If we do a SetValue on Beavis immediately followed by a SetValue on Butt-Head, then their parent becomes the update view. When the update occurs Beavis, Butt-Head, their parent, and all of the children shown in heavy borders will be redrawn.

A simple work around for this is to call RefreshViews right after we call SetValue on Beavis. The RefreshViews will cause Beavis to be redrawn and will empty out the update rectangle. Then when we call SetValue on Butt-Head we won't cause his parent to become the update view and we won't end up with this large update rectangle that forces everyone to redraw. The only two views that get redrawn will be Beavis and Butt-Head precisely what we want.

Lets consider another scenario in which exactly the same problem occurs. Suppose clicking on Beavis brings up a pop-up menu and that we change Butt-Head when an item from the menu is selected. This scenario is depicted in Figure 2.

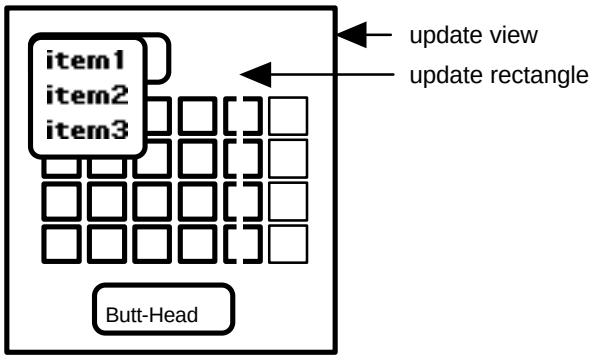

Figure 2 - Beavis & Butt-Head Update Scenario II

If the PickActionScript simply calls SetValue on Butt-Head, then once again we'll have the situation where Beavis, Butt-Head, their parent, and all of the children shown in heavy borders will be redrawn.

As before, the work around is to follow the call to SetValue with a call to RefreshViews. The views that get redrawn will be the minimal set – Beavis, Butt-Head, their parent, and the 2x3 views in the top-left corner of the grid.

As you can see, there are numerous variations on this theme. They often arise when a view appears in front of other views (like a pop-up menu or a dialog box) and makes changes to other views.

The general principle to take away from all this is that once you're done changing one part of the screen you might consider doing a RefreshViews to prevent one of your ancestors becoming the update view and to keep the update rectangle from getting unnecessarily large.

Don't start sprinkling calls to RefreshView in your code in attempt to improve performance. RefreshViews is a fairly expensive call. Calling it excessively will slow down your application. You need to think carefully about how your screen is updating (put some print statements in your ViewDrawScripts) and only call RefreshViews where it makes sense.

Filling Views

Another part of the dirtying mechanism is that when you dirty an unfilled view its closest filled ancestor becomes the update view. Recall that your application's base view is filled. So if no other ancestor is filled, your base view will become the update view.

In the simplest case, this means that when you dirty an unfilled view its parent will be redrawn as well as any children that overlap the dirtied view. Figure 3 shows this situation.

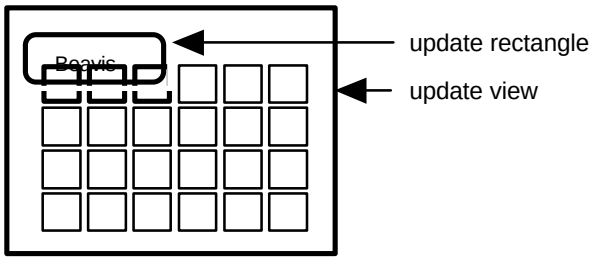

Figure 3 - Transparent Beavis Update

Beavis is a transparent view. When Beavis is dirtied his bounds become the update rectangle and his parent becomes the update view. Beavis, his parent and the three children drawn in heavy borders will be redrawn.

Since Beavis is transparent and overlaps three of his siblings, all of this redrawing is necessary to update the screen properly. This scenario is somewhat contrived. In practice, you don't normally use overlapping views. However, even if we remove the overlapping views from figure 3 we still end up with Beavis and his parent being redrawn when we really only want Beavis to be redrawn.

The work around is very simple. Give Beavis a fill pattern (anything but fill none). As with calling RefreshViews, some thought is required when deciding which views to give fill patterns. You don't want to give all your views fill patterns because this increases the time it takes to draw them. Only views which you explicitly dirty need to be considered.

Grouping Views

Another way you can help improve your view performance is by grouping your views into container views. Figure 4 shows our Beavis and Butt-Head example with the 4x6 views in the center all grouped together inside an invisible parent view – a plain old clView shown in gray.

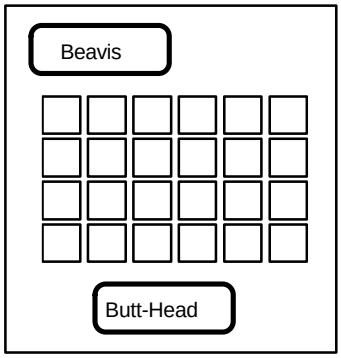

Figure 4 - Example of Grouping Views

This grouping helps in a variety of way. For example, it will speed up hit testing. When a tap is made the view hierarchy is searched for the view that contains the point. If a view contains the point, then its children are search to see if any of them contain the point. When a view doesn't contain the point, then none of its children are searched.

Without the grouping view, every a tap could require examining as many as twenty-seven views to see if they contain the point (Beavis, Butt-Head, their parent, plus the 4x6 children). With the grouping, taps outside the grouping view cause only a maximum of four views to be examined.

A similar speed-up occurs for the update process. Recall that the update view's descendants are searched to see if they intersect the update rectangle and therefore, need to be redrawn. If a view does not intersect the update rectangle then none of its children need to be further examined.

By grouping the 4x6 views together we ensure they won't be tested for redrawing in the update process when the grouping view does not intersect the update rectangle.