Blowing Smoke

Michael S. Engber

This article was published in the November 1993 issue of PIE Developers magazine. For information about PIE Developers, contact Creative Digital Systems at CDS.SEM@APPLELINK.APPLE.COM or 415.621.4252.

The system view types clParagrah and clEdit automatically handle scrubbing gestures using the ever popular poof sound and dissipating smoke. This article explains how you can use the scrub effect in your own views. This article assumes that the reader has some experience using NTK to write Newton applications. The culmination of this article is a method, SmokeIt, that you define in your application's base view. You pass this method a bounds frame, in global coordinates, describing the rectangle you want "smoked."

Poof Sound
The first problem to solve is how to get the graphics and sound data. Fortunately, we can use the same bitmaps and sound the system uses. The sound is available as ROM_poof and the three bitmaps in the animation sequence are ROM_Cloud1, ROM_Cloud2, and ROM_Cloud3. These pre-defined constants and many others are provided by NTK. In the initial release of NTK, all the constant definitions are in the "GlobalData" text file. In future releases of NTK, the actual definitions will be in the various platform files, but a listing of them can be found in the "NTK Definitions" text file.

Connect the NTK Inspector and execute the following code:

```
PlaySound(ROM_poof);
```

You will hear the familiar poof. We have the sound problem solved. PlaySound plays the sound asynchronously, so all that's remains is to immediately follow the call to PlaySound with animation code.

Cloud Animation
In principle, it sounds easy. Just draw ROM_Cloud1, ROM_Cloud2, and ROM_Cloud3 in rapid succession. I tried a number of different techniques before finding one that closely resembled the way the system does it. Speed was not the problem. In fact, delays need to be inserted between drawing the clouds or they go by too quickly to be seen. We'll start by examining the bitmaps we need to draw. If we use NTK's Inspector to examine ROM_Cloud1, we see the following:

```
{
  mask: <mask, length 2632>,
  bits: <bits, length 2632>,
  bounds: {left: 0, top: 0, right: 178, bottom: 109}}
```

Bitmaps are represented as frames with a bits slot containing the raw bitmap data and bounds slot specifying the rectangle in which to draw the (scaled) bitmap. Bitmaps can optionally have a mask slot containing raw bitmap data that is used to determine which pixels from the bits data are actually drawn. The mask allows you to draw non-rectangular images. Only bits pixels whose corresponding mask pixels are "on" actually get drawn.

Bitmaps are used in three ways in the Newton:

- They can be used in graphic objects that are created with MakeShape and drawn with DrawShape;
- They can be used in the icon slots of clPicture views, in which case the view system takes care of drawing them; and
- They can be passed to the CopyBits drawing function.

Unfortunately, the first two methods won't work for us. when you pass a bitmap to MakeShape the resulting

graphic object does not retain the mask data. Since our cloud bitmaps have masks, we can't use MakeShape and DrawShape. clPicture views use the mask data. We could create a dummy clPicture view and animate it by changing its icon slot using SetValue. However, when we closed the dummy clPicture view, the entire view rectangle would be erased instead of just the area covered by clouds. A small point, but it's definitely a different look and feel from the way the system does scrubbing.

CopyBits uses the mask data if you specify modeMask as the transfer mode, and it doesn't require allocating any dummy views. That makes CopyBits the best choice of the three methods, but life is never so easy. We need to be able to draw the clouds scaled. Unlike its namesake in the Macintosh ToolBox, CopyBits only lets you specify the top-left corner of where you want to draw, not the entire rectangle. However, CopyBits does use the bounds slot of the bitmap. We can adjust the bounds slot to achieve suitable scaling. Of course, the three cloud bitmaps are in ROM, so we can't change them directly. Instead, we clone them and change the bounds of the clones. Below is the relevant code extracted from the SmokeIt function:

```
local bm1 := Clone(ROM_cloud1);
local bm2 := Clone(ROM_cloud2);
local bm3 := Clone(ROM_cloud3);

bm1.bounds := bm2.bounds := bm3.bounds :=           smokeBounds;
```

Since bitmaps are fairly large objects (we can see that the bits and mask fields occupy 2*2632 = 5264 bytes), you might be concerned that cloning consumes a lot of memory. Remember that we're using Clone, not DeepClone. Clone only copies the top level of a data structure. The only memory allocated is a new, three-slot frame. The bits and mask fields of the clone are still references to the same objects referenced by the original slots.

Below is the code from the single animation step which draws ROM_Cloud1. First we draw the cloud, then we delay a bit, and finally we erase the cloud. These steps are repeated for each cloud.

```
GetRoot():CopyBits(bm1,left,top,modeMask);
Sleep(2);
GetRoot():CopyBits(bm1,left,top,modeBic);
```

Just for completeness, let's consider how the two drawing techniques we abandoned would handle scaling: Graphic objects created with MakeShape can be scaled using the ScaleShape function; clPicture views scale their pictures if you have their content set to full horizontal and full vertical justification.

Forcing A Screen Update

The net result on the screen from each animation step is to erase the area covered by the cloud. After the animation we need to ensure that the screen updates properly. Presumably SmokeIt's caller takes care of making the scrubbed item disappear, by deleting the view for instance. However, smoke clouds usually occupy a larger area than the item being scrubbed. This means SmokeIt must ensure that the entire smokeBounds rectangle gets re-drawn.

If we know that smokeBounds is confined to a single view, we could simply send that view a Dirty message. Since we're writing SmokeIt as a general purpose routine, however, we can't rely on having this information. Alternately, we could send a Dirty message to the root view, but this turns out to be unacceptably slow. What's needed is something like the InvalRect call from the Macintosh ToolBox, but no such call is available.

We can achieve the same effect by creating a dummy view, opening it, and then closing it. Here is the relevant code from SmokeIt:

```
local danQuayle := BuildContext({
  viewClass:clView,
              viewFlags:vFloating,
```

```
                viewBounds:smokeBounds});
danQuayle:Open();
danQuayle:Close();
```

BuildContext takes a template as an argument and returns the view it constructed from that template. The view's parent is the root view. The template is not added to the root view's viewChildren array–you can liken the view to a free agent. BuildContext should not be treated as a general purpose method for creating views dynamically. It's only useful in special situations like the one at hand.

The template passed to BuildContext is the bare minimum amount of information you need to create a view. The only part worth discussing is the viewFlags. If vFloating isn't specified, the dummy view appears behind any floating views that happen to be around. Hence, those floating views wouldn't be forced to update when the dummy view is closed.

SmokeIt

The source code to SmokeIt is listed on the nest page. Copy this source into a script slot named SmokeIt in your application's base view. That way it will be available to all the views in your application. The smokeBounds argument is a bounds frame. That is, a frame with a left, top, bottom, and right slot. It needs to be specified in global coordinates. A typical usage might be to delete a view named deadMan. You might do this with code like:

```
:SmokeIt(deadMan:GlobalBox());
deadMan:Close();
```

In practice, you'll find that you want the rectangle to be somewhat larger than the item being deleted so that the clouds completely obscure it. I find that expanding 25% in each direction is adequate.

Some useful functions and methods for creating bounds frames are SetBounds, RelBounds, StrokeBounds, :GlobalBox, and :LocalBox. They are all documented in the *Newton Programmer's Guide.*

```
func(smokeBounds)
begin

 local top := smokeBounds.top;
 local left := smokeBounds.left;

 local bm1 := Clone(ROM_cloud1);
 local bm2 := Clone(ROM_cloud2);
 local bm3 := Clone(ROM_cloud3);

 //CopyBits draws bitmaps scaled to their bounds
 bm1.bounds := bm2.bounds := bm3.bounds := smokeBounds;

 //explained below
 local danQuayle := BuildContext({
        viewClass:clView,
        viewFlags:vFloating,
        viewBounds:smokeBounds});

 PlaySound(ROM_poof);

 GetRoot():CopyBits(bm1,left,top,modeMask);
 Sleep(2);
 GetRoot():CopyBits(bm1,left,top,modeBic);
```

```
 GetRoot():CopyBits(bm2,left,top,modeMask);
 Sleep(2);
 GetRoot():CopyBits(bm2,left,top,modeBic);

 GetRoot():CopyBits(bm3,left,top,modeMask);
 Sleep(1);
 GetRoot():CopyBits(bm3,left,top,modeBic);

//  force update (a la InvalRect in Mac ToolBox)
// If you knew what view(s) to dirty you could
//  just use :Dirty() to force the update. This
//  code is general purpose, so it doesn't
//  know (and GetRoot():Dirty() is way slow.)

 danQuayle:Open();
 danQuayle:Close();

end

p
```