Technical Note TN1149

Smoothing Fonts

CONTENTS

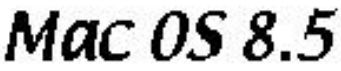
Mac OS 8.5 came with a "Smooth all fonts on screen" checkbox in the Appearance control panel, accompanied by a setting of the minimum text size required for turning on anti-aliasing. In this Technote, I briefly explain the current implementation of this feature, and discuss its limitations and the possible compatibility problems with existing applications. I then document the new APIs that allow applications to take control over anti-aliased text rendering. Finally, I explain why anti-aliased text looks so bad when highlighted (in particular with dark highlight colors), and what you can do to solve the problem.

Updated: [Jan 18 1998]

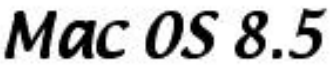
Smooth Edges

What Are They?

Here are two samples of text with and without font smoothing turned on (sample text is in Textile font, size 24):



Font smoothing OFF



Font smoothing ON

And here is the (visual) explanation of why the right hand side looks better:

When the outline of a character is scan-converted to a bitmap, the limited resolution of the screen device causes "jaggies" to appear along non-vertical and non-horizontal portions of the contour. By adding levels of gray along oblique or round edges, your eyes get tricked into perceiving smoother contours. To produce the gray-level pixmaps for anti-aliased text, the new TrueType scaler in Mac OS 8.5 still uses the grid-fitted outline for the requested size. But, before passing the outline to the traditional black-white scan converter, it conceptually replaces each pixel by a square of 4x4 sub-pixels, and applies the scan conversion to four-times the original resolution. The count of black dots in the resulting bitmap squares for each pixel (a number between 0 and 16) is then mapped to a gray level encoded in four bits (a value from 0 to 15), which becomes the pixmap's pixel value. QuickDrawText's blitter then looks to it that these gray levels blend nicely from the text's foreground color into the existing background color.

What are the issues?

So far, so good. Unfortunately, this approach comes with some potentially annoying consequences (some of which are alluded to in the "Mac OS 8.5 Read Me" file):

a) Font Smoothing ON implies `SetOutlinePreferred(TRUE)`. Because the metrics of outline fonts are in general different from the bitmap font metrics, text may reflow, or get mispositioned or clipped.

This was a deliberate design decision. Take the example of the font Times normal. It comes as an outline font, accompanied by bitmap (`NFNT`) versions for the sizes 9, 10, 12, 14, 18, and 24. By default, the `outlinePreferred` flag is off; this has been done this way (back at the time when System 7 with TrueType was introduced), to avoid reflow of existing documents that used the bitmap fonts, given that the metrics of the outline font are different from those of the bitmap fonts.

It's bad enough that since then the appearance and metrics of text goes through qualitative jumps while scaling through different sizes. Without the above decision, we would see another qualitative jump when font smoothing is turned on: sizes 11, 13, 15, 16, 17, 19, etc. would appear anti-aliased, and the bitmap sizes would stay jaggily black-white!

Still, Mac OS 8.5 is not even consistent in this matter. If the bitmap versions are hidden inside the `sfnt` format (in 'bloc' and 'bdat' tables), the current version of the TrueType scaler doesn't quite know what to do with `outlinePreferred`. In an attempt to avoid some other compatibility problems with the metrics of existing 2-byte fonts, it ignores the request for smoothed text, and returns unsmoothed glyphs, in this case - with yet another exception, if the bitmaps contained in the '`bdat`' and '`bloc`' tables are themselves stored grayscaled. This will only be the case for certain Microsoft fonts converted for use on the Macintosh platform.

The only good solutions would be either to provide gray-level versions of all the bitmap fonts, or to develop a procedural antialiasing of the existing bitmap fonts. The first option clearly is not realistic; the second was victimized by the well-known time-resource constraints of any software engineering effort. Other solutions have been proposed; they are "not good". It is in general typographically plainly impossible, from an esthetic point of view, to use the metrics of bitmap fonts together with glyphs rendered from outlines, or vice-versa.

b) Applications assume that text is always representable in a black-white bitmap.

There are good reasons to draw text in an offscreen (flicker-free editing and scrolling, etc.), and there are good reasons to try to get away with a 1-bit deep offscreen (memory footprint). Obviously, anti-aliased text cannot be rendered under these circumstances, and if the application mixes calls to direct text drawing onto the screen with updates from the offscreen, anti-aliased and non-anti-aliased text can end up next to each other.

Even if the offscreen is deep enough, i.e., at least 8 bits/pixel, the assumption of monochrome text may still hurt. Some applications use QuickDraw's `CopyBits` to transfer the offscreen text on top of some textured background, for example. But there is no `CopyBits` transfer mode that knows how to blend the gray fringe pixels into the destination, while keeping the "black" source pixels intact. The result is in general quite unsatisfactory.

c) Highlighting and unhighlighting assume monochrome text

The consequences of this fact are bad enough that they deserve their own chapter, at the end of this note.

Back to top

How to Turn Smoothing ON and OFF

With all these possible consequences of turning font smoothing on, it is clear that applications need a programmatic way to interfere with a user's setting. More positively, they should also get the possibility to selectively turn anti-aliasing on, even on systems where the user chose not to enable font smoothing.

It is equally clear that applications need to restore the system's state as soon as they are done with their text-rendering job that required a different setting. Setting and restoring the flag of font smoothing carries no performance penalty at all (in contrast to what seasoned Mac OS programmers might assume, based on experience with pre-8.5 versions of the `FontManager`). However, it is a system-wide global setting (not a per-`GrafPort` or per-context one), and as such, it affects all text redrawing of background processes and system-owned text drawing as well. It would not make a good impression to have text rendering switch unpredictably between anti-aliasing and not.

Here are the API calls, as contained in the latest versions of the *Fonts.h* include file:

```
Boolean  IsAntiAliasedTextEnabled(SInt16* outMinFontSize);

OSStatus SetAntiAliasedTextEnabled(Boolean inEnable, SInt16 inMinFontSize);
```

The first function returns a Boolean with the obvious interpretation. If you are interested in the minimum size for which anti-aliasing is enabled, `*outMinFontSize` returns this value as set by the user in the Appearance control panel, provided the function value is TRUE; if the function value is FALSE, `*outMinFontSize` is undefined. If you do not care about this minimum font size, pass NULL as parameter.

The second function allows to set the state of anti-aliased text; if `inEnable` is TRUE, the `inMinFontSize` parameter is put in the place where a user-selected minimum font size value would go. Because the rendering quality of anti-aliased text for small point sizes is currently perceived as unsatisfactory, minimum sizes below 12 are not allowed, and are, in fact, replaced by 12. The function always returns `noErr`.

Here is an example of how to use these functions to turn anti-aliased text temporarily OFF (if it's enabled):

```
//-------------------------------------------------
Boolean  userWantsSmoothText ;
SInt16 previousMinSize;

userWantsSmoothText = IsAntiAliasedTextEnabled(&previousMinSize);
if (userWantsSmoothText)
 (void)SetAntiAliasedTextEnabled(false, 0);
 // second parameter is ignored when first parameter is FALSE

// draw text the "jaggy" way, here

// and then set it back to what the user wants:

if (userWantsSmoothText)
 (void)SetAntiAliasedTextEnabled(true, previousMinSize);
//-------------------------------------------------
```

One more thing: the symbols `IsAntiAliasedTextEnabled` and `SetAntiAliasedTextEnabled` are exported from the `FontManager` code fragment in the System, but the `InterfaceLib` you are using with your development tools may not yet include them. To make the Linker happy, you may need to create a little "`FontManager`" stub library that exports these symbols and contains empty implementations of the functions, until the next revision of the `InterfaceLib`.

Back to top

What Happened to Highlighting?

The "Imaging With QuickDraw" volume of Inside Macintosh explains on pages 4-41 through 4-44 how the highlighting concept has been designed in QuickDraw. Here are the two crucial features:

a) highlighting only affects pixels with either the background or the highlight color;

b) calling the highlighting function twice returns the original state.

Clearly, there was no anticipation of anti-aliasing in the original design of QuickDraw. When a bitmap edge gets smoothed by using shades of colors that blend the foreground color into the background color, the fringe of intermediate shades does not participate in the highlighting algorithm. Assume that the foreground color is black, the background white, and the highlight color pretty saturated (like a dark blue or red). With anti-aliased text, there will be a fringe of more or less light-gray pixels around the original black bitmap, to blend the contours into white. When highlighting is applied, the white background pixels are replaced by the relatively dark highlight color, whereas the "fringe" pixels keep their original values. The effect is an ugly sparkling around the contour, which can make the text nearly unreadable in small sizes and certain type faces.

Apple tried hard to find a solution to this problem, but the evidence of the facts won. It is not possible in practice to maintain feature b) above (a requirement for compatibility with existing code), while modifying feature a) such that contours blend correctly into whatever the background pixels are (highlighted or not).

The solution has to come from a different approach to highlighting. Either the highlighted areas are first erased to the desired state (highlighted or not), before the text is drawn into the area, or we provide two different API calls for Highlighting and Unhighlighting and implement them in a way that takes anti-aliased text into account. (See the functions `ATSUHighlightText` and `ATSUUnHighlightText` in ATSUnicode.h for an example.)

For the time being, we recommend you use a pastel-like highlight color which minimizes the ugliness, or to turn off anti-aliased text altogether if you are frequently editing text in sizes and type faces that make the defect particularly annoying.

Back to top

Summary

Mac OS 8.5 introduced the feature of anti-aliased text, which provides a substantial esthetic improvement in many text drawing cases, in particular for slanted type faces. However, it became apparent that it may conflict with assumptions made in legacy code, and that it comes with other undesirable side effects. Expect that the future will bring certain improvements.

On the other hand, perhaps we should follow the recommendation "enjoy moderately and wisely" not only for some other good things in life, but also for font smoothing.

Back to top

Downloadables

 Acrobat version of this Note (K). Download

Back to top
