Amiga Font Scaling and Aspect Ratio

by John Orr

[Editor's Note: This article assumes the reader is familiar with how the Amiga stores and manages bitmap fonts under pre-2.0 Releases of the operating system. For more information, see the ''Graphics: Text'' chapter of the RKM:Libraries and Devices Manual*.]*

The Release 2.0 OS offers a significant improvement over the Amiga's previous font resources: it now has the ability to scale fonts to new sizes and dimensions. This means, if the *diskfont.library* can't find the font size an application requests, it can create a new bitmap font by scaling the bitmap of a different size font in the same font family. The 2.04 release of the OS improved upon the *diskfont.library's* font scaling ability so the Amiga now can utilize AGFA/Compugraphic outline fonts, yielding scalable fonts that don't have the exaggerated jagged edges inherent in bitmap scaling.

The best thing about the Amiga's font scaling is that its addition to the system is completely invisible to an application program. Because the *diskfont.library* takes care of all the font scaling, any program that uses OpenDiskFont() to open a font can have scalable fonts available to it. For simple scaling, the programming interface is the same using Release 2.0 as it was under 1.3.

However, there is one feature of the 2.0 *diskfont.library* that the 1.3 programming interface cannot handle. When scaling a font (either from an outline or from another bitmap), the 2.0 *diskfont.library* can adjust the width of a font's glyphs according to an *aspect ratio* passed to OpenDiskFont(). A font glyph is the graphical representations associated with each symbol or character of a font.

The aspect ratio refers to the shape of the pixels that make up the bitmap that *diskfont.library* creates when it scales a font. This ratio is the width of a pixel to the height of the pixel (XWidth/YWidth). The *diskfont.library* uses this ratio to figure out how wide to make the font glyphs so that the look of a font's glyphs will be the same on display modes with very different aspect ratios. For more information on Amiga aspect ratios, see Carolyn Scheppner's article, ''Finding the Aspect Ratio'', from the September/October 1991 issue of *Amiga Mail*.

To add this new feature, several changes to the OS were necessary:

1) The OS needed to be able to store an aspect ratio for any font loaded into the system list,
2) The structures that *diskfont.library* uses to store bitmap fonts on disk had to be updated so they can store the aspect ratio a font was designed for, and
3) The way in which an application requests fonts from *diskfont.library* had to be altered so that an application could ask for a specific aspect ratio.

For the *diskfont.library* to be able to scale a font to a new aspect ratio, it needs to know what the font's current aspect ratio is. The Amiga gets the aspect ratio of a font currently in the system list from an extension to the TextFont structure called (oddly enough) TextFontExtension. Under 2.0, when the system opens a new font (and there is sufficient memory), it creates this extension.

A font's TextFont structure contains a pointer to its associated TextFontExtension. While the font is opened, the TextFont's tf_Message.mn_ReplyPort field points to a font's TextFontExtension. The <*graphics/text.h*> include file #defines tf_Message.mn_ReplyPort as tf_Extension.

The TextFontExtension structure contains only one field of interest: a pointer to a tag list associated with this font:

```
struct TagItem *tfe_Tags;/* Text Tags for the font */
```

If a font has an aspect ratio associated with it, the OS stores the aspect ratio as a tag/value pair in the tfe_Tags tag list.

The `TA_DeviceDPI` tag holds a font's aspect ratio. The data portion of the `TA_DeviceDPI` tag contains an X DPI (dots per inch) value in its upper word and a Y DPI value in its lower word. These values are unsigned words (UWORD). *At present, these values do not necessarily reflect the font's true X and Y DPI, so their specific values are not relevant. At present, only the ratio of the X aspect to the Y aspect is important* (more on this later in the article).

Notice that the X and Y DPI values are not aspect values. The X and Y aspect values are the reciprocals of the X and Y DPI values, respectively:

XDPI = 1/XAspect
YDPI = 1/YAspect

so, the aspect ratio is YDPI/XDPI, not XDPI/YDPI.

Before accessing the tag list, an application should make sure that this font has a corresponding TextFontExtension. The ExtendFont() function will return a value of `TRUE` if this font already has an extension or ExtendFont() was able to create an extension for this font.

Where Do the X and Y DPI values Come From?

The Amiga has a place to store a font's X and Y DPI values once the font is loaded into memory, but where do these X and Y values come from? A font's X and Y DPI values can come from several sources. The X and YDPI can come from a font's disk-based representation, or it can be set by the programmer.

For the traditional Amiga bitmap fonts, in order to store the X and Y DPI values in a bitmap font's *.font* file, the structures that make up the *.font* file had to be expanded slightly. There are two changes from the structures described in the ''Graphics:Text'' chapter of the *RKM: Libraries and Devices* manual. The first is that the fch_FileID in the FontContentsHeader structure (essentially, the *.font* file) contains a different file ID (TFCH_ID from <*libraries/diskfont.h*>) from the traditional *.font* file. This tells the *diskfont.library* that this FontContentsHeader uses a special kind of FontContents structure called a TFontContents structure. The difference between these structures is that the TFontContents structure allows the OS to store tag value pairs in the extra space set aside for the font's name.

```
struct TFontContents {
    char    tfc_FileName[MAXFONTPATH-2];
    UWORD   tfc_TagCount;/* including the TAG_DONE tag */
    /*
     *    if tfc_TagCount is non-zero, tfc_FileName is overlayed with
     *    Text Tags starting at:  (struct TagItem *)
     *        &tfc_FileName[MAXFONTPATH-(tfc_TagCount*sizeof(struct TagItem))]
     */
    UWORD   tfc_YSize;
    UBYTE   tfc_Style;
    UBYTE   tfc_Flags;
};
```

Currently, out of all the system standard bitmap fonts (ones loaded from bitmaps on disk or ROM, not scaled from a bitmap or outline), only one has a built in aspect ratio: *Topaz-9*.

For the Compugraphic outline fonts, the X and Y DPI values are built into the font outline. Because the format of the Compugraphic outline fonts is proprietary, information about their layout is available only from AGFA/Compugraphic. For most people, the format of the outline fonts is not important, as the *diskfont.library* handles converting the fonts to an Amiga-usable form.

The other place where a font can get an aspect ratio is an application. When an application opens a font with OpenDiskFont(), it can supply the `TA_DeviceDPI` tag that the *diskfont.library* uses to scale (if necessary) the font according to the aspect ratio. To do so, an application has to pass OpenDiskFont() an extended version of the TextAttr structure called the TTextAttr:

```
struct TTextAttr {
    STRPTR  tta_Name;/* name of the font */
    UWORD   tta_YSize;/* height of the font */
    UBYTE   tta_Style;/* intrinsic font style */
    UBYTE   tta_Flags;/* font preferences and flags */
    struct TagItem *tta_Tags;/* extended attributes */
};
```

The TextAttr and the TTextAttr are identical except that the tta_Tags field points to a tag list where you place your `TA_DeviceDPI` tag. To tell OpenDiskFont() that it has a TTextAttr structure rather than a TextAttr structure, set the `FSF_TAGGED` bit in the tta_Style field.

For example, to ask for *Topaz-9* scaled to an aspect ratio of 75 to 50, the code might look something like this:

```
...

#define MYXDPI (75L << 16)
#define MYYDPI (50L)

struct TTextAttr mytta = {
    "topaz.font",
    9,
    FSF_TAGGED,
    0L,
    NULL
};

struct TagItem tagitem[2];
struct TextFont *myfont;
ULONG dpivalue;
struct Library *UtilityBase, *DiskfontBase, *GfxBase;
void main(void)
{
    tagitem[0].ti_Tag = TA_DeviceDPI;
    tagitem[0].ti_Data = MYXDPI | MYYDPI;
    tagitem[1].ti_Tag = TAG_END;

...
               if (myfont = OpenDiskFont(&mytta))
                {
                    dpivalue = GetTagData(TA_DeviceDPI,
                        0L,
                        ((struct TextFontExtension *)(myfont->tf_Extension))->tfe_Tags);
                    if (dpivalue) printf("XDPI = %d    YDPI = %d\n",
                                    ((dpivalue & 0xFFFF0000)>>16),
                                    (dpivalue & 0x0000FFFF));
                    /* blah, blah, render some text, blah, blah */
                    ...

                    CloseFont(myfont);
                }
...

}
```

Some Things to Look Out For

One misleading thing about the `TA_DeviceDPI` tag is that its name implies that the *diskfont.library* is going to scale the font glyphs according to an actual DPI (dots per inch) value. As far as scaling is concerned, this tag serves only as a way to specify the aspect ratio, so the actual values of the X and Y elements are not important, just the ratio of one to the other. A font glyph will look the same if the ratio is 2:1 or 200:100 as these two ratios are equal.

To makes things a little more complicated, when *diskfont.library* scales a bitmap font using an aspect ratio, the X and Y DPI values that the OS stores in the font's TextFontExtension *are identical* to the X and Y DPI values passed in the `TA_DeviceDPI` tag. This means the system can associate an X and Y DPI value to an open font size that is very different from the font size's actual X and Y DPI. For this reason, applications should not use these values as real DPI values. Instead, only use them to calculate a ratio.

For the Compugraphic outline fonts, things are a little different. The X and Y DPI values are built into the font outline and *reflect a true X and Y DPI.* When the *diskfont.library* creates a font from an outline, scaling it according to an application-supplied aspect ratio, *diskfont.library does not* change the YDPI setting. Instead, it calculates a new XDPI based on the font's YDPI value and the aspect ratio passed in the `TA_DeviceDPI` tag. It does this because the Amiga thinks of a font size as being a height in pixels. If an application was able to change the true Y DPI of a font, the *diskfont.library* would end up creating font sizes that were much larger or smaller than the YSize the application asked for. If an application needs to scale a font according to height as well as width, the application can adjust the value of the YSize it asks for in the TTextAttr.

As mentioned earlier, almost all of the system standard bitmap fonts *do not* have a built in aspect ratio. This means that if an application loads one of these bitmap fonts without supplying an aspect ratio, the system will not put a `TA_DeviceDPI` tag in the font's TextFontExtension: the font will not have an aspect ratio. If a font size that is already in the system font list does not have an associated X and Y DPI, the *diskfont.library* cannot create a new font of the same size with a different aspect ratio. The reason for this is the *diskfont.library* cannot tell the difference between two instances of the same font size where one has an aspect ratio and the other does not. Because *diskfont.library* cannot see this difference, when an application asks, for example, for *Topaz-8* with an aspect ratio of 2:1, OpenDiskFont() first looks through the system list to see if that font is loaded. OpenDiskFont() happens to find the ROM font *Topaz-8* in the system font list, which has no X and Y DPI. Because it cannot see the difference, *diskfont.library* thinks it has found what it was looking for, so it does not create a new *Topaz-8* with an aspect ratio of 2:1, and instead opens the *Topaz-8* with no aspect ratio.

This also causes problems for programs that do not ask for a specific aspect ratio. When an application asks for a font size without specifying an aspect ratio, OpenDiskFont() will not consider the aspect ratios of fonts in the system font list when it is looking for a matching font. If a font of the same font and style is already in the system font list, even though it may have a wildly distorted aspect ratio, OpenDiskFont() will return the font already in the system rather than creating a new one.

Due to a bug in the 2.00 through 2.04 versions of the *graphics.library* function WeighTAMatch(), the OS ignores the aspect ratio of a font when trying to determine if the TTextAttr passed to OpenDiskFont() matches a font already in the system font list. If an application asks for a font that exactly matches a font already in the font list in all ways except in aspect ratio, OpenDiskFont() will open the font in the system font list rather than creating a font with a matching aspect ratio. For example, if an application requests *CGTimes-20* with an aspect ratio of 1:2 and while scanning through the system font list OpenDiskFont() finds *CGtimes-20* with an aspect ratio of 1:1, OpenDiskFont() will open the font with a 1:1 aspect ratio. A patch that temporarily fixes this bug, *SetPatchWTAM*, is on the Denver/Milano DevCon disks.

The following example, *cliptext.c*, renders the contents of a text file to a Workbench window. This example gets the new aspect ratio for a font by asking the display database what the aspect ratio of the current display mode is.