Using Compugraphic Typefaces with Bullet

by John Orr

One of the improvements made to the Amiga's operating system for Workbench Release 2.1 is programmatic control of AGFA's IntelliFont® scaling engine. With this engine, application programs can fully utilize Compugraphic (CG) typefaces installed by *Fountain* (the CG typeface install that comes with 2.04 and 2.1). Some of the features that the font scaling engine offers include:

- ❏ Rasterization of a typeface to arbitrary vertical and horizontal resolutions.
- ❏ Baseline rotation of glyphs to an arbitrary angle.
- ❏ Glyph shearing (italicizing) to any angle from -45 to 45 degrees (inclusive).
- ❏ Access to kerning tables.
- ❏ Algorithmic emboldening.

Starting the Engine

There are several steps involved in using a font outline on the Amiga.

1. Open the font contents file (the ‘‘.font’’ file) and verify that it has a corresponding outline tag file (an ‘‘.otag’’ file).

All system supported fonts on the Amiga have a font contents file. From this file, an application can determine a font's type, so the application knows how to utilize the font. The font contents file is a FontContentsHeader structure (defined in *<diskfont/diskfont.h>*). The first field of that structure (fch_FileID) contains an ID identifying the font's type. If that type is `OTAG_ID` (0x0F03), the font is an outline and it should have a corresponding otag file. The otag file should be in the same directory as the font contents file.

2. Open the otag file, verify that it is valid, load its tag list into memory, and resolve any memory indirections in the tag list.

The otag file contains a tag list that describes the typeface. All of these tags are defined in *<diskfont/diskfonttag.h>* as either level 1, level 2, or level 3 outline tags. Level 1 tags are required to be present in an otag file. Level 2 and 3 tags are optional. See the include file for more information on the tag levels.

The first tag of the otag file is always `OT_FileIdent`. Its value is the size of the otag file. This tag is here to verify the validity of the otag file. If the first tag is not `OT_FileIdent`, or the `OT_FileIdent` tag value is not the size of the otag file, the otag file is invalid, so don't attempt to use it. If the file is valid, copy the entire file into a buffer.

The tags from the otag file have a special `OT_Indirect` bit. If this bit is set, the tag's value is an indirect reference to data defined elsewhere in the otag file. The tag's value is the offset to the data (in bytes) from the beginning of the otag file. For example, the otag file *fonts:CGTimes.otag* that is on the 2.04 Release disks contains the tag `OT_Family` (0x80009003), which has its `OT_Indirect` bit set. The value of the `OT_Family` tag is 195, meaning that the data for it--the NULL terminated string ‘‘CG Times’’--is located 195 bytes into the otag file.

Of course, if an application read the file *fonts:CGTimes.otag* into a memory buffer, the ‘‘CG Times’’ string would be 195 bytes from the beginning of the buffer. The `OT_Family` tag must point to the absolute address of its data, so when an application loads an otag file into memory, it has to resolve the indirection of the `OT_Indirect` tags in memory. The application can do this by adding the buffer address to each `OT_Indirect` tag value.

3. Find out the name of the typeface's scaling engine and obtain a pointer to the engine's GlyphEngine structure.

One of the level 1 outline tags is the `OT_Engine` tag. This tag refers to the name of this typeface's scaling engine. At present there is only one scaling engine available on the Amiga. It is named *Bullet*. This is the IntelliFont scaling engine. The name is left over from the original implementation of the IntelliFont engine used on the Amiga. The scaling engine itself is in its own Exec library, called *bullet.library*. To open the engine, build a complete library name by adding the string ‘‘.library’’ to the `OT_Engine` string, and open it with OpenLibrary(). Don't assume that `OT_Engine` will always be the string ‘‘bullet’’. In the future, Commodore or some third party developer may create additional scaling engines libraries that will allow the Amiga to use other types of outline typefaces (PostScript, Nimbus-Q, etc.). Using the proper library name will help ensure compatibility with future possible scaling engines.

All scaling engine libraries contain several functions:

OpenEngine()	If successful, returns a pointer to the library's GlyphEngine structure.
CloseEngine()	Releases the GlyphEngine structure obtained in OpenEngine().
SetInfo()/SetInfoA()	Sets current parameters of a scaling engine (the current typeface, the current point size, the current output resolution, etc.)
ObtainInfo()/ObtainInfoA()	Queries a scaling engine for glyph information (a glyph's bitmap, the kerning value between two glyphs, etc.).
ReleaseInfo()/ReleaseInfoA()	Releases data obtained with ObtainInfo()/ObtainInfoA().

To obtain a pointer to a GlyphEngine structure for a particular scaling library, use that library's OpenEngine() routine. The function takes no arguments.

4. Tell the engine which typeface to use.

Setting a scaling engine's current typeface involves the SetInfoA() (or SetInfo()) function and the Level 0 tags from <*diskfont/diskfonttag.h*>. The SetInfoA() function takes two parameters, a pointer to the GlyphEngine structure, and a tag list of level 0 outline tags. The Level 0 tags act as commands for a scaling engine, some of which are for setting scaling engine parameters (with SetInfo() or SetInfoA()), and some of which are for querying information from a scaling engine (with ObtainInfo() or ObtainInfoA()).

Two tags set a scaling engine's current typeface: OT_OTagPath and OT_TagList. The OT_OTagPath tag points to the full path name of a typeface's otag file (for example, *fonts:CGTimes.otag*). The OT_OTagList tag points to the tag list created in step 2 above.

5. Set other scaling engine parameters.

There are three other parameters the scaling engine needs in order answer queries for information:

```
OT_DeviceDPI         OT_PointHeight        OT_GlyphCode
```

The `OT_DeviceDPI` tag refers to the resolution of the output device. The tag value's high word is the horizontal resolution and the low word is the vertical resolution. Both are unsigned words measured in dots per inch.

The `OT_PointHeight` tag refers to the height of a typeface in points. One point is approximately equal to 1/72 of an inch (AGFA/Compugraphic defines the point to be 0.01383 inches). For CG typefaces, this height is the distance between baselines from one line to the next (without any leading adjustments). The point height is represented as a fixed point, two's complement binary number, with the point situated in the middle of a long word. This means the lower word is the fractional portion and the upper word the whole number portion (the number covers the powers of two from 2^{15} through 2^{-16}).

For those who may not know, a two's complement number is a way of representing negative numbers. To find the two's complement of a negative number, find the one's complement of the absolute value of that number (change all the binary ones to zeros and all the zeros to ones) and then add one to the result. To change from two's complement, subtract one from the two's complement number and find the one's complement of the number. For example,

Before conversion to two's complement, the absolute value of the one byte decimal quantity -32 is represented as:

```
In binary         0010 0000  ($20)
One's complement  1101 1111  ($DF)
Add One           0000 0001  ($01)
-----------------------------------
Two's complement  1110 0000  ($E0)
```

So the number -32 is encoded as 1110 0000 or 0xE0 in hex. Notice that the high bit (the sign bit) of the encoded number is set if the number is negative. If the number is zero or greater, the high bit is clear. This procedure is independent of where the ‘‘point’’ is in the negative number (the ‘‘point’’ in this case is the divider between the whole portion of the number and the fractional portion). When the computer adds one to the one's complement, it does not consider where the ‘‘point’’ is in the one's complement, the computer just treats the one's complement value as a whole integer. For example, to encode the decimal quantity -531/256 as a two byte, fixed point, two's complement binary number where the point is situated in the middle of the two bytes:

```
531/256 = -(2 + 19/256) = -2.13 in hex

In binary           0000 0010.0001 0011   ($02.13 in hex, with the point)

One's complement    1111 1101 1110 1100   ($FDEC  in hex ignore the point from now on)
Add One             0000 0000 0000 0001   ($0001)
--------------------------------------
Two's complement    1111 1101 1110 1101   ($FDED)
```

Notice that the one added to the one's complement is in the 2^{-8} (= 1/256) column.

The `OT_GlyphCode` tag refers to the current glyph. When an application asks the scaling engine to rasterize a glyph, this is the glyph the engine renders. The scaling engine uses Unicode encoding to represent glyphs. Unicode is an international character encoding standard that encompasses many of the world's national scripts in a 16-bit code space. Conveniently, the Amiga's Latin-1, 8-bit character set corresponds to the same glyphs as the Unicode standard. To set the current glyph to a character from the Amiga character set, use the same Latin-1 code, but pad it out to a 16-bit value.

Because the Compugraphic typefaces use their own character set, the scaling engine in the bullet.library has to map the Unicode glyph codes to Compugraphic glyph codes. Note that the Unicode standard encompasses many more glyphs than just the Latin-1 character set or the Compugraphic character set, so many of the characters in the Unicode set do not map to any glyphs in the Compugraphic set. For example, Unicode includes several Asian ideogram sets, that the Compugraphic set does not. The result is the vast majority of the Unicode characters are not available using Compugraphic typefaces. The Compugraphic character set covers roughly 400 glyphs. For more information on the UniCode standard, see the Unicode Consortium's book *The Unicode Standard, Worldwide Character Encoding* published by Addison-Wesley (ISBN 0-201-56788-1).

Rasterizing a Glyph

Once an application has set up the scaling engine, obtaining a glyph is a matter of asking for it. As was mentioned earlier, the ObtainInfoA()/ObtainInfo() function queries a scaling engine for glyph information. This function accepts a pointer to a GlyphEngine structure and a tag list. To ask for a rasterization of a glyph, pass ObtainInfo() the `OT_GlyphMap` tag. The engine expects the OT_GlyphMap value to be an address where it can place the address of a GlyphMap structure. The GlyphMap structure (defined in *<diskfont/glyph.h>*) is for reading only and looks like this:

```
struct GlyphMap {
    UWORD  glm_BMModulo;    /* # of bytes in row: always multiple of 4 */
    UWORD  glm_BMRows;      /* # of rows in bitmap */
    UWORD  glm_BlackLeft;   /* # of blank pixel columns at left of glyph */
    UWORD  glm_BlackTop;    /* # of blank rows at top of glyph */
    UWORD  glm_BlackWidth;  /* span of non-blank columns (horizontal span of the glyph) */
    UWORD  glm_BlackHeight; /* span of non-blank rows (vertical span of the glyph) */
    FIXED  glm_XOrigin;     /* distance from upper left corner of bitmap to lower */
    FIXED  glm_YOrigin;     /*   left of glyph (baseline), in fractional pixels */
    WORD   glm_X0;          /* approximation of XOrigin in whole pixels */
    WORD   glm_Y0;          /* approximation of YOrigin in whole pixels */
    WORD   glm_X1;          /* approximation of XOrigin + Width */
    WORD   glm_Y1;          /* approximation of YOrigin + Width */
    FIXED  glm_Width;       /* character advance, as fraction of em width */
    UBYTE *glm_BitMap;      /* actual glyph bitmap */
};
```

The glm_BitMap field points to a single bitplane bitmap of the glyph. This bitmap is not necessarily in Chip RAM, so if an application needs to use the blitter to render the glyph, it has to copy the bitmap to a Chip RAM buffer. The fields glm_BMModulo and glm_BMRows are the dimensions of the whole bitmap. The glyph itself does not occupy the entire bitmap area. The fields glm_BlackLeft, glm_BlackTop, glm_BlackWidth, and glm_BlackHeight describe the position and dimension of the glyph within the bitmap. The fields glm_XOrigin and glm_YOrigin are the X and Y offsets from the bitmap's upper left corner to the glyph's lower left corner. The lower left corner lies on the glyph's baseline. These X and Y offsets are in fractional pixels. The fields glm_X0 and glm_Y0 are rounded versions of glm_XOrigin and glm_YOrigin, respectively.

The glm_Width field is a measure of the width of the glyph in fractions of an em (pronounced like the letter 'M'). This value is a fixed point binary fraction. The em is a relative measurement. A distance of one em is equal to the point size of the typeface. For example, in a 36 point typeface, one em equals 36 points which is approximately equal to a half inch. For a 72 point typeface, one em equals 72 points which is approximately equal to one inch.

When an application is finished with the GlyphMap structure, it must use the ReleaseInfoA() or ReleaseInfo() function to relinquish the GlyphMap structure. This function uses the same format as ObtainInfoA()/ObtainInfo(), except the data value of the `OT_GlyphMap` tag is a pointer to the GlyphMap structure (rather than a pointer to a pointer).

Kerning

The IntelliFont scaling engine also supports two types of kerning. One type of kerning is called text kerning, which is for regular bodies of text. The other type of kerning is called design kerning, which is for more obvious displays, like headlines. The basic difference is that design kerning is generally more tightly spaced than text kerning.

Before asking for a kerning pair, an application has to tell the engine which character pair to kern. It does this using one of the SetInfo() functions to set the primary glyph, `OT_GlyphCode`, and the secondary glyph code, `OT_GlyphCode2`.

To ask the scaling engine for a kerning value, use one of the ObtainInfo() functions with the `OT_TextKernPair` (for text kerning) or `OT_DesignKernPair` (for design kerning) tags. The engine expects the value of the kerning tag to be an address where it can store a four byte long kerning value. The kerning value is a fixed point binary fraction of an em square (like the glm_Width field from the GlyphMap structure). This value is the distance to remove from the primary character advance (the glm_Width of `OT_GlyphCode`) when rendering the secondary glyph (`OT_GlyphCode2`) immediately following the primary glyph.

Unlike other ObtainInfo() tags, the scaling engine does not allocate any resources when answering queries about kerning values. Applications do not have to use ReleaseInfo() functions for kerning queries made with either `OT_TextKernPair` or `OT_DesignKernPair`.

Width Lists

An application can find the widths of a typeface's glyphs using the `OT_WidthList` tag with one of the ObtainInfo() functions. The engine expects the `OT_WidthList` value to be an address where it can place the address of a MinList structure. This MinList points to a list of GlyphWidthEntry structures. The GlyphWidthEntry structure (defined in <*diskfont/glyph.h*>) is for reading only and looks like this:

```
struct GlyphWidthEntry {
    struct MinNode gwe_Node;  /* on list returned by OT_WidthList inquiry */
    UWORD          gwe_Code;  /* entry's character code value */
    FIXED          gwe_Width; /* character advance, as fraction of em width */
};
```

The MinList contains an entry for each valid Unicode glyph ranging from the primary glyph, `OT_GlyphCode`, through the secondary glyph, `OT_GlyphCode2`. The engine does not create a GlyphWidthEntry structure for codes without glyphs (for example the codes before the space character in the Latin-1 character set).

When an application is finished with the width list, it must use one of the ReleaseInfo() functions to relinquish the list. This function uses the same format as the ObtainInfo() functions, except the data value of the `OT_WidthList` tag is a pointer to the MinList (rather than a pointer to a pointer). The primary and secondary code values do not have to remain constant while an application is using a width list. The engine deallocates the width list resources independently of the primary and secondary code values, so these can change after obtaining a width list.

The following code fragment asks the scaling engine, ge, for a list of character widths of the Unicode glyphs ranging from unicode (`OT_GlyphCode`) to unicode2 (`OT_GlyphCode2`), inclusive. The fragment steps through the list of widths, printing each one.

```
struct MinList *widthlist;
struct GlyphWidthEntry *widthentry;
.  .  .

if (SetInfo(ge,
            OT_GlyphCode,   unicode,
            OT_GlyphCode2,  unicode2,
            TAG_END) == OTERR_Success)
{
    if (ObtainInfo(ge, OT_WidthList, &widthlist, TAG_END) == OTERR_Success)
    {
        for (widthentry = (struct GlyphWidthEntry *) widthlist->mlh_Head;
            widthentry->gwe_Node.mln_Succ;
            widthentry = (struct GlyphWidthEntry *)
            widthentry->gwe_Node.mln_Succ)
        {
            printf("$%lx: %ld.%ld\n",
                widthentry->gwe_Code,
                widthentry->gwe_Width>>16,
                ((widthentry->gwe_Width&0xffff)*10000)>>16);
        }
        ReleaseInfo(ge, OT_WidthList, widthlist, TAG_END);
    }
}
.  .  .
```

Notice that the ObtainInfo() functions (as well as the SetInfo() functions) return an error code (the error codes are defined in <*diskfont/oterrors.h*>). If that error code is equal to `OTERR_Success`, the operation was successful.

Rotating

Taking advantage of other features of the Bullet library is a matter of setting other engine parameters using one of the SetInfo() functions with some other level 0 tags. One interesting feature of the IntelliFont engine is its ability to rotate glyphs. By setting the `OT_RotateSin` and `OT_RotateCos` values, the IntelliFont engine can rotate a glyph's baseline from the glyph origin (the glm_XOrigin and glm_YOrigin from the GlyphMap structure).

The `OT_RotateSin` and `OT_RotateCos` are, respectively, the sine and cosine of the baseline rotation angle. The engine can rotate to any angle. *The sine and cosine must correspond to the same angle and must be in the sine and cosine value range (0 through 1 inclusive)*. The engine does not do any error checking on the sine and cosine values. As a result, the engine will yield strange results if the sine and cosine are from very different angles or if these values are out of range for sines and cosines (greater than 1). By default, the engine sets these values to 0.0 and 1.0, the sine and cosine of 0 degrees. These values are encoded as fixed point binary fractions (the negative values are two's complement).

When setting the baseline rotation, an application must set both the sine and cosine. It must set `OT_RotateSin` first, then `OT_RotateCos`. An application can set both values in the same SetInfo() function, but the sine must come first. For example, to set the rotation angle to 150 degrees:

The sine of 150 degrees is 0.5 which is 0x00008000 in hex. The cosine of 150 degrees is approximately -0.866 which is approximately 0xffff224c in hex (two's complement), so:

```
. . .

if (SetInfo(ge,
            OT_RotateSin, 0x8000,
            OT_RotateCos, 0xFFFF224C,
            TAG_END) == OTERR_Success) /* If SetInfo() returns OTERR_Success, */
                                       /* it worked OK. */
{   /* The baseline rotation has been set, now the application can render it. */

    . . .
```

Shearing

Like baseline rotation, glyph shearing (also known as italicizing) is a matter of setting some Level 0 tags. The shearing tags, `OT_ShearSin` and `OT_ShearCos`, specify the shearing angle, or the angle at which the typeface is italicized. This angle refers to the angle that results from rotating the vertical axis clockwise. The angle can range from -45 to 45 degrees (inclusive). Like the rotation angle, the shearing angle is represented as a sine and cosine value that must correspond to the same angle and must fall into normal bounds for sine and cosine values. Also like the rotation angle sine and cosine tags, an application must set both the `OT_ShearSin` and `OT_ShearCos` tags, in that order. By default, the shearing value is zero degrees meaning there is no shearing (`OT_ShearSin` = 0x00000000, `OT_ShearCos` = 0x00010000).

Other Level 0 Tags

There are several other Level 0 tags:

OT_DotSize. This tag specifies the X and Y size of the target device's dots. The X and Y DPI imply a dot size. For example, at 300 X and 300 Y DPI, the resolution implied dot size is 1/300 inches by 1/300 inches. For some devices (like some dot matrix printers), the size of the output dot does not match its resolution implied size. To a degree, the IntelliFont engine can account for this. The dot size is represented as a percentage of the dot's resolution implied size. The X percentage is in the tag value's upper word, and the Y percentage is in the tag value's lower word.

OT_SetFactor. This tag distorts the width of a typeface by changing the width of the em square. The scaling engine changes the em width to this tag's value. The value is a fixed point binary fraction.

OT_EmboldenX/OT_EmboldenY. These tags specify the algorithmic emboldening factor in the X and Y direction, respectively. The tag values are fixed point two's complement binary numbers. The units are measured in ems. Emboldening values above zero embolden the typeface. Emboldening values below zero lighten the typeface. By default, both values are zero.

OT_GlyphWidth. This tag's value specifies a width for the current typeface. It is a fraction of an em represented as a fixed point binary number. If this value is set to something besides 0.0, all glyphs will have this width. To turn off the constant width, set `OT_GlyphMap` back to 0.0 (its default value).

The Otag File Tags

The Outline Tag (otag) file contains a number of tags that describe a font outline to the Diskfont library. The purpose of most of these tags is to allow Diskfont to attribute styles to a typeface when loading a font outline as a standard Amiga system font. Most applications that use the scaling engine will not need to worry about the meaning of the majority of these tags, but they are described below. The following tags are Level 1 tags and must be present in every otag file:

OT_FileIdent. Every valid otag file starts with this tag. Its value is the size of the file. It doesn't really have anything to do with the definition of the typeface, but it does serve as a way to check the validity of the otag file.

OT_Engine. This tag's value points to a string naming the font scaling engine. For example, the OT_Engine tag in *fonts:CGTimes.otag* is ‘‘bullet’’.

OT_Family. This tag's value points to a string naming the typefaces font family. For example, the OT_Family in *fonts:CGTimes.otag* is ‘‘bullet’’.

OT_SymbolSet. This tag's value is a two byte ASCII code for this typeface's symbol set. This tells the system which symbol set to use to map the Amiga character set to the Compugraphic character set. The symbol set for most CG fonts designed for use with the Amiga is ‘‘L1’’, which stands for Latin1. The exception is the CG fonts from Gold Disk, Inc. They use the ‘‘GD’’ (Gold Disk) symbol set.

OT_YSizeFactor. For traditional Amiga fonts, the font size does not include any spacing on top or bottom of the typeface--the Amiga doesn't consider it part of the font. CG fonts include spacing on the top and bottom of their typefaces. This tag's value is a ratio of the point height of a typeface to its ‘‘black’’ height (the point height minus the space on the typeface's top and bottom). The high word is the point height factor and the low word is the black height factor. Note that these values form a ratio. Individually, they do not necessarily reflect any useful value.

OT_SerifFlag. If this tag's value is TRUE, this typeface has serifs.

OT_StemWeight. This tag's value can be anywhere from 0 through 255 and indicates a nominal weight or ‘‘boldness’’ to the typeface. The *<diskfont/diskfonttag.h>* include file defines a set of ratings for this tag's value. See that file for more details. When the Diskfont library opens an outline font, it uses this value to determine if a typeface is bold.

OT_SlantStyle. The *<diskfont/diskfonttag.h>* include file defines a set of three possible values for this tag's value. See that file for more details. When the Diskfont library opens an outline font, it uses this value to determine if a typeface is italicized/obliqued.

OT_HorizStyle. This tag's value can be anywhere from 0 through 255 and indicates a nominal width rating to the typeface. The *<diskfont/diskfonttag.h>* include file defines a set of ratings for this tag's value. See that file for more details. When the Diskfont library opens an outline font, it uses this value to determine if a typeface is extended.

OT_AvailSizes. This tag's value points to a sorted list of UWORDs. The first UWORD is the number of entries in the sorted list. The remaining UWORDs are the font sizes that the Diskfont library lists when calling AvailFonts().

OT_SpecCount. This tag's value is a number of spec tags that follow it. A spec tag is private to the scaling engine.

The following are Level 2 tags. They may also be in an otag file but are not required:

OT_BName. This tag points to a string naming the bold variant of this typeface. For example, the *fonts:CGTimes.otag* file lists ‘‘CGTimesBold’’ as its bold variant.

OT_IName. This tag points to a string naming the italic variant of this typeface.

OT_BIName. This tag points to a string naming the bold italic variant of this typeface.

OT_SpaceWidth. This tag's value is the width of the space character at 250 points (where there are 72.307 points in an inch). The width is in Design Window Units (DWUs). One DWU is equal to 1/2540 inches. To convert to X pixels:

```
OT_SpaceWidth   pointsize
------------- * --------- *  XDPI = spacewidth in pixels (X dots)
    2540           250
```

OT_IsFixed. If this tag's value is `TRUE`, every glyph in this typeface has the same character advance (a fixed width).

OT_InhibitAlgoStyle. This tag's value is a bitmask that is compatible with the ta_Style field of the TextAttr structure (defined in <*graphics/text.h*>). This tag tells which styles cannot be added to a typeface algorithmically. For example, if the `FSF_BOLD` bit in OT_InhibitAlgoStyle is set and a user asks for a bold version of the typeface, the *diskfont.library* (or an application) can add that style algorithmically.

At present there are no Level 3 tags.

About the Examples

This article contains two code examples. The first, *Rotate*, rotates a user-specified glyph around a central point. By default, it rotates a 36 point 'A' using the font *fonts:CGTimes.font*. If *Rotate* finds an AmigaDOS environment variable called ‘‘XYDPI’’, it will use the X and Y DPI it finds in that variable as the default target device DPI (see the description of the Level 0 `OT_DeviceDPI` tag). If that variable is not defined, Rotate will use an XDPI of 68 and a YDPI of 27 which, nominally, is the X and Y DPI of a standard Hires display.

The second example, *View*, displays a file using the same defaults as *Rotate*. *View* utilizes kerning pairs to display its glyphs. Because *View* only considers ‘‘visible’’ characters, it ignores characters that have widths but no glyph. The result is, *View* doesn't print any space characters. If *View* were smarter, it would ask the scaling engine for a width list so it could properly advance the current pointer when it comes across a space or some other character without a glyph.

Notice that *View* uses a slightly modified version of *BulletMain.c* called *BulletMainFile.c*. Only *BulletMain.c* appears in the example code. The only significant difference between the two is that *BulletMainFile.c* obtains a file name for *View* to display.