NintendoWare for CTR

Font and Its Derived Classes

2010/07/29

Ver. 1.0.0

PROVISIONAL TRANSLATION

The content of this document is highly confidential and should be handled accordingly.

Table of Contents

Code

Figures

1 Introduction

1.1 About This Manual

This manual describes the following classes used by NintendoWare for CTR:

- `nw::font::Font`
- `nw::font::ResFont`
- `nw::font::ArchiveFont`
- `nw::font::PackedFont`

These classes handle the fonts used to render characters and text.

For information on matters common to the drawing of characters in general, see Character Drawing Fundamentals (`DrawText_First.pdf`).

1.2 Font Licenses

NintendoWare for CTR can use any font installed on the PC to display characters on the CTR system using the `ResFont` class. However, a license for each font used must be obtained before selling such software. Be sure to obtain the proper font licenses for each game title.

No font licenses of any kind are supplied with NintendoWare for CTR.

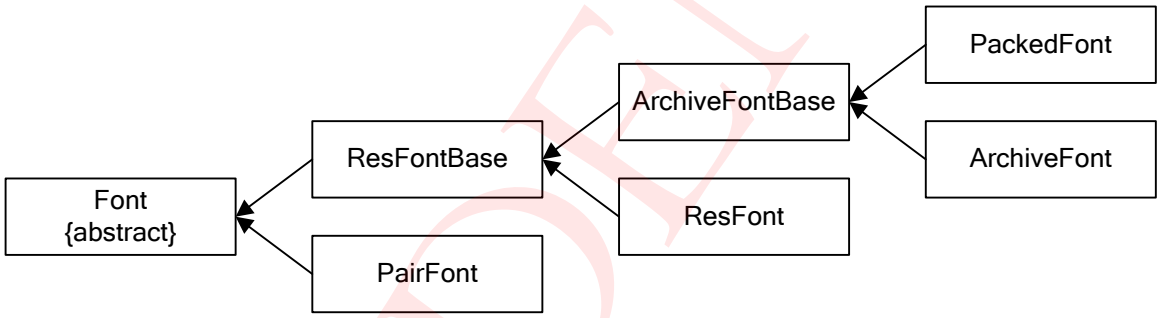
2 Font and Its Derived Classes

2.1 Font

The `Font` class is an abstract class used to define font processing on NintendoWare for CTR. Since many member functions of this class are pure virtual functions, they are implemented in classes that are derived from `Font`. Note that `Font` itself is never instantiated because it is an abstract class—classes derived from `Font` are used instead.

Three classes are derived from `Font`: `ResFont`, `ArchiveFont`, and `PackedFont` (Figure 2-1). Note that `ResFontBase` and `ArchiveFontBase`, both shown in Figure 2-1, are internal classes that cannot be used.

Figure 2-1 Font Class and Derivations



These derived classes differ only in terms of the resource data on which they are based and the method in which they are constructed and destructed. The manner in which they handle fonts is the same. Chapter 3, Constructing and Destroying Fonts, describes each of these differences. Chapter 4, Accessing Fonts, describes common features of classes derived from `Font`.

2.2 ResFont

The `ResFont` class is a derived class of the `Font` class and actually uses font resource (bcfnt) data, which can be created using NW4C FontConverter, as a font.

The font resource must be created using NW4C FontConverter before constructing this class. For details on NW4C FontConverter, see the FontConverter Manual (`FontConverter_Manual.pdf`).

2.3 ArchiveFont

The `ArchiveFont` class is a derived class of the `Font` class and is used to extract/unpack only those glyph groups the user needs from an archive font (bcfna) that can be created using NW4C FontConverter.

An archive font must be created using NW4C FontConverter before constructing this class. For details on NW4C FontConverter, see the FontConverter Manual (`FontConverter_Manual.pdf`).

2.4 PackedFont

The `PackedFont` class is a derived class of the `Font` class and can handle compressed archive fonts (bcfna) created using NW4C FontConverter as-is.

The `PackedFont` class allows you to use large fonts while consuming a only a small amount of memory by unpacking only that part of the font to be used in an internal cache based on the glyphs requested. Note, however, that the load is heavy when drawing characters due to the unpacking required to get glyphs. Because a large amount of processing is required to unpack glyphs, there is no advantage to using this class with conventional fonts. This font class is used for special application where huge fonts are being used.

An archive font must be created using NW4C FontConverter before constructing this class. For details on NW4C FontConverter, see the FontConverter Manual (`FontConverter_Manual.pdf`).

2.5 PairFont

`PairFont` is a derived class of the `Font` class for maintaining two fonts internally and treating them as a single font.

The font data and sheet data of a `PairFont` match that of the wider or taller internal font. As a result, it is possible that the intended results will not be obtained if the two fonts that are used as a `PairFont` have differing character rendering information, such as the baseline position.

3 Constructing and Destroying Fonts

This chapter describes how to construct and destroy fonts using the different classes derived from `Font`.

3.1 ResFont

A `ResFont` object uses bcfnt data, created with NW4C FontConverter, as actual fonts. During this process, bcfnt data is loaded into memory by an application and associated with `ResFont`.

Use the `SetResource` member function to construct a `ResFont` object and the `RemoveResource` member function to destroy a `ResFont` object. A `SetResource` member function passes a pointer to the bcfnt data that has been loaded into memory and creates an association between `ResFont` and the resource. The `RemoveResource` member function can be used to destroy the memory, since it returns a pointer to the bcfnt that was given as an argument to the `SetResource` member function.

Code 3-1 Constructing and Destructing the ResFont Class

```
nw::font::ResFont font;

//--- constructor (when loading from a ROM)
{


   //--- Load bcfnt file from ROM
      nn::fs::FileReader fontReader(filePath);

    s32 fileSize = (s32)fontReader.GetSize();

      //--- Allocate memory from device memory.
      void* fileBuffer = DevMemAlloc(fileSize, nw::font::GlyphDataAlignment);

      fontReader.Read(fileBuffer, fileSize);

   //--- Associate bcfnt
  font.SetResource(fileBuffer);
      //--- Set buffer for drawing.
      const u32 drawBufferSize = nw::font::ResFont::GetDrawBufferSize(buffer);
      void* drawBuffer = MemAlloc(drawBufferSize, 4);
      font.SetDrawBuffer(drawBuffer);

}

/*
```

```
  Use as a Font class instance
*/

//--- destructor
{
   //--- Initialize buffer for drawing.
       void* drawBuffer = font.SetDrawBuffer(NULL);
      MemFree(drawBuffer);


   //--- Deallocate memory in which bcfnt file was loaded

      void* buffer = font.RemoveResource();
      MemFree(buffer);
}
```

3.2 ArchiveFont

`ArchiveFont` is a class used to handle archive fonts, created using NW4C FontConverter, as actual fonts. Font data is constructed based on bcfna data loaded into memory by the application from the file system or other storage location.

Unlike `ResFont`, where bcfnt data loaded into memory is used as-is, `ArchiveFont` loads bcfna data at the time of construction and creates font data in a different memory region than that in which the bcfna data is stored. This data is then used as font data. Memory for storing bcfna data and memory for `ArchiveFont` to store font data must be allocated at the time of construction. Although the size of this memory will vary depending on the bcfna to be used and the combination of glyph groups required, it can be calculated using the `GetRequireBufferSize` member function. Although memory being used to store bcfna data can be deallocated after construction is complete, the memory region given to `ArchiveFont` falls under the management of `ArchiveFont` and cannot be deallocated or rewritten until `ArchiveFont` is destroyed.

There are two methods of constructing `ArchiveFont`. The first method is to load all bcfna data into memory at once and use the `Construct` member function (see Code 3-2). Although this method is simple, it temporarily uses a large amount of memory due to the fact that memory for storing bcfna data and memory for storing constructed font data are both necessary at the same time.

The other method of constructing `ArchiveFont` is by loading bcfna data into memory a little bit at a time (see Code 3-3). In this case, call the `InitStreamingConstruct` member function to prepare for construction and then construct the font a little bit at a time by calling the `StreamingConstruct` member function each time bcfna data is loaded. This method is a bit difficult to implement, but less memory is required than when constructing using the `Construct` member function due to the fact that the size of memory used to load bcfna data does not depend on the size of bcfna data.

Regardless of which of the two methods described above is used to construct `ArchiveFont`, the `Destroy` member function is used to destroy the class. Since the `Destroy` member function returns a pointer to the memory region assigned to `ArchiveFont` at the time of construction, it can be used to perform operations including deallocating memory.

3.3 PackedFont

Just like `ArchiveFont`, `PackedFont` is a class for handling archive fonts created by NW4C FontConverter as actual fonts.

In contrast to `ArchiveFont`, which unpacks the archive font at time of construction, `PackedFont` constructs font data based on compressed font data without unpacking it at the time of construction. `PackedFont` therefore requires less memory for font data than does `ArchiveFont`. Note, however, that a load is placed on the system as font data is unpacked when glyphs are obtained.

The interface used to construct/destruct the `PackedFont` class is the same as used with `ArchiveFont`. For details, see section 3.2, ArchiveFont. The only difference is that an argument for specifying the cache size is added to the `GetRequireBufferSize` member function. The cache is used to reduce the load on the system when retrieving glyphs for the second or subsequent time by storing unpacked font data in the cache ahead of time. Although the benefits of using the cache can be achieved for many glyphs by increasing the size of the cache, the amount of memory required may become too large, eliminating any advantage to using `PackedFont`. A suitable cache size must therefore be specified.

Code 3-2 Constructing and Destructing the ArchiveFont Class (Loaded All at Once)

```
nw::font::ArchiveFont font;

//--- constructor (when loading from a ROM)
{
    //--- Load bcfnafile form ROM
    nn::fs::FileReader fileReader(filePath);

    u32 fileSize = (u32)fileReader.GetSize();

    void* fileBuffer = MemAlloc(fileSize);

    fileReader.Read(fileBuffer, fileSize);

    //--- Calculate buffer size required for construction.
    u32 fontBufferSize = nw::font::ArchiveFont::GetRequireBufferSize(
fileBuffer, glyphGroups);

    //--- Allocate memory from device memory.
    void* fontBuffer = DevMemAlloc(fontBufferSize);
```

```
    //--- 構築
    font.Construct(fontBuffer, fontBufferSize, fileBuffer, glyphGroups);
    MemFree(fileBuffer);
}

/*
    Use as a Font class instance
*/

//--- destructor
{
    //--- deallocate memory
    void* buffer = font.Destroy();
    MemFree(buffer);
}
```

Code 3-3 Constructing/Destructing the ArchiveFont Class (Sequentially Loaded)

```
 nw::font::ArchiveFont font;

 //--- constructor (when loading from ROM)
 {
     nw::font::ArchiveFont::ConstructContext context;

     //---- Allocate buffer for successive loading.
     u32 readPos = 0;
     u32 readBufferSize = 16 * 1024;
     void* readBuffer = MemAlloc(readBufferSize);

     //--- Read header to find buffer size required for construction
     nn::fs::FileReader fileReader(filePath);
     u32 fileSize = (u32)fileReader.GetSize();
     u32 readSize = (u32)fileReader.Read(readBuffer, readBufferSize);

    //--- Calculate buffer size required for construction
     u32 fontBufferSize = nw::font::ArchiveFont::GetRequireBufferSize(
 readBuffer, glyphGroups);
     void* fontBuffer = DevMemAlloc(fontBufferSize);

    //--- Construction
```

```
    font.InitStreamingConstruct(
&context, fontBuffer, fontBufferSize, glyphGroups);
    while (readPos < fileSize)
    {
        font.StreamingConstruct(&context, readBuffer, readSize);
        readPos += readSize;
        readSize = fileReader.Read(readBuffer, readBufferSize);
    }

    MemFree(readBuffer);
}

/*
    Use as a Font class instance
*/

//--- destructor
{

    //--- Deallocate memory
    void* buffer = font.Destroy();
    MemFree(buffer);
}
```

3.4 PairFont

A `PairFont` is constructed using two already-constructed `Font` classes and their derived classes. The `PairFont` itself does not perform either the construction or destruction of these internally-maintained fonts.

Code 3-4 Constructing and Destructing the PairFont Class

```
//--- Already-constructed fonts
nw::font::ResFont primary;
nw::font::ResFont secondary;

//--- Constructor
nw::font::PairFont font(&primary, &secondary);

/*
```

```
    Use as Font
*/

//--- Destructor
//--- Destruction of primary and secondary
```

4 Accessing Fonts

The `Font` class and its derived classes are used to abstract font data. The `Font` class itself therefore has a passive existence with no effect on other resources. The only operations that can be performed on `Font` are the acquisition of font data and the rewriting of some parameters. It is also possible to use it as a utility to get the CharStrmReader corresponding to a font.

4.1 Getting Font Data

The following font data can be acquired. For the meaning of each separate parameter, see The Basics of Text Rendering (DrawText_First.pdf).

- Cell width and height (in pixels)
- The ascent of the font (in pixels)
- The descent of the font (in pixels)
- Font height (in pixels)
- Font width (in pixels)
- Linefeed width (in pixels)
- Maximum character width (in pixels)
- Default character width information (in pixels)
- Texture format of sheet (nw::font::FontSheetFormat type)
- Corresponding font encoding (`nw::font::FontEncoding` type)
- The character's character width information (in pixels)
- Character glyph data (`nw::font::Glyph` type)

Use the character glyph data when you render characters with font data. Refer to Chapter 6, bcfnt Sheet Format, for the `nw::font::Glyph` type used to obtain the glyph data and its meanings.

4.2 Changing Parameters

Parameters that can be changed are listed below. The changes take effect immediately.

When using `ResFont`, these parameters are changed by changing the resource. If multiple instances of `ResFont` are sharing the same resource, changing the parameters of one instance will affect the other instances as well.

- Line feed width
- Default character width information
- Substitute character

4.3 CharStrmReader

The `CharStrmReader` class hides differences in character-string encoding formats and reads character strings without regard to the encoding format.

The `GetCharStrmReader` member function of the `Font` class returns an instance of the `CharStrmReader` class. This instance is used to read text strings having the character string encoding for the corresponding font. Using `GetCharStrmReader` allows code to be written that does not depend on the particular encoding format that corresponds to the font.

Code 4-1 Example of Using CharStrmReader

```
//--- Calculate rendering width for character string
int GetStringWidth(
    const nw::font::Font& font,
    const char*           str
)
{
    //--- Always get CharStrmReader instance from the Font class
    nw::font::CharStreamReader reader = font.GetCharStrmReader();
    nw::font::CharCode code;
    int width = 0;

    //--- Set the text string stream to be read
    reader.Set(str);

    //--- Calculate character width of characters until NULL character is reached
    while( (code = reader.next()) != '¥0' )
    {
    width += font.GetCharWidth(code);
    }

    return width;
}
```

5 Cache Management

Although the PackedFont class only requires a small amount of memory by saving font data in a compressed format, a sheet must be unpacked when a glyph is requested. To reduce the associated process load, PackedFont includes a function for caching sheets that have already been unpacked. However, only a small cache can be allocated since use of a large cache would completely eliminate the advantage of using PackedFont. So rather than allowing all unpacked sheets to be stored in a cache at once, the cache is used in a way that the contents of the cache keep on being sequentially replaced.

With PackedFont, a least recently used (LRU) algorithm is applied when replacing the contents of the cache so that the contents in the least recently used area of the cache are replaced first unless caching operations are performed. Member functions for performing various caching operations have therefore been added to PackedFont in case you want to control the cache more efficiently. Functions that have been added for the cache management will be described below.

5.1 Preloading Sheets

Although sheets are usually unpacked at the time a glyph is requested, this data can also be unpacked ahead of time and stored in a cache. The load on the system when drawing characters can be reduced by loading sheets into the cache during times when extra CPU is available.

To preload data, specify a character code to the PreloadSheet member. The PreloadSheet member function will unpack the sheet including the glyph having the specified character code and store it in the cache.

5.2 Locking the Cache

Although data is usually replaced in the region of the cache that has been used least recently, the cache can be locked so that its contents are not replaced.

In a language such as Japanese, where there is a relatively low number of hiragana and a relatively large number of kanji that alternately appear in text strings, cache efficiency can be increased by caching a sheet of hiragana, but this efficiency will drop if the hiragana cache is replaced by a sheet of kanji. In such cases, cache efficiency can be increased, and the load on the system decreased, by locking the cache in which the sheet of hiragana is stored.

The cache is locked by specifying a character code to the LockSheet member function. The LockSheet member function will unpack the sheet, including the glyph having the specified character code, store it in the cache, and then lock the cache.

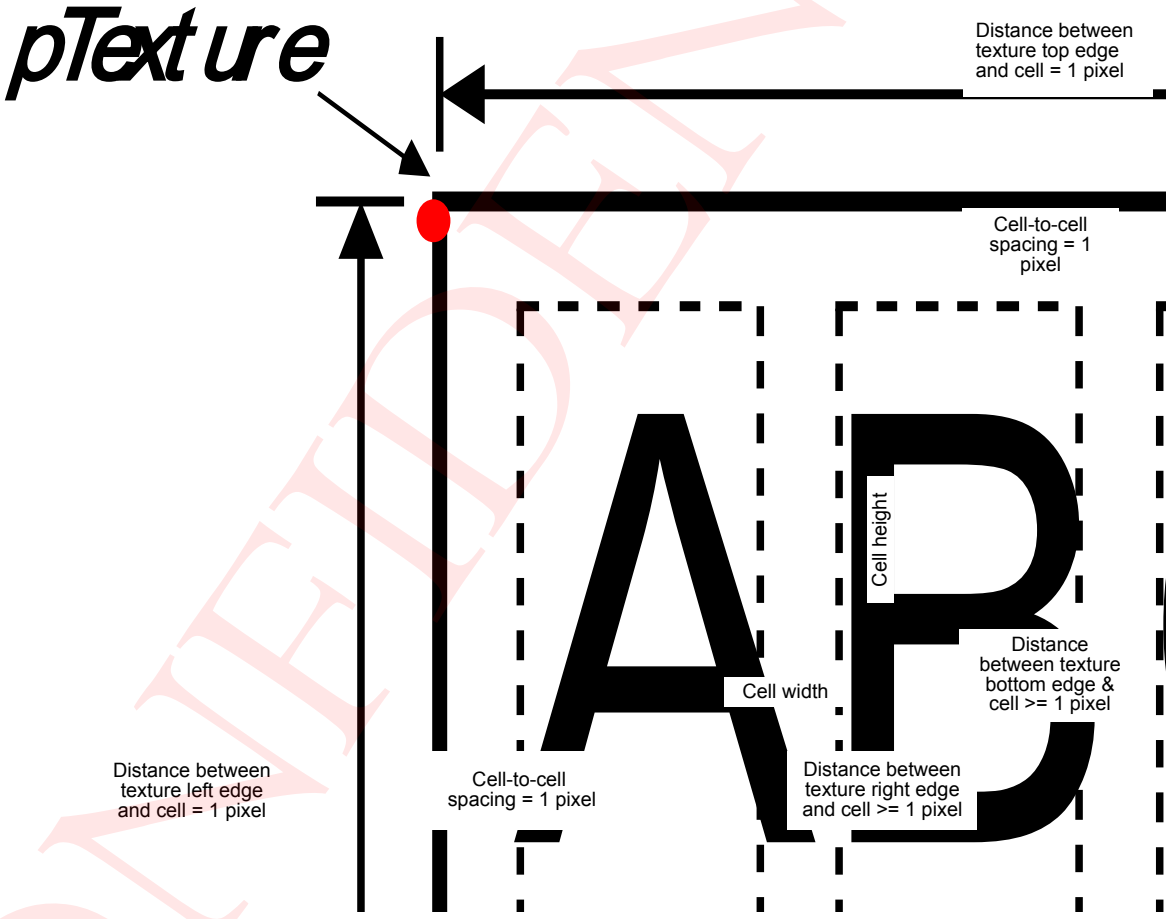
A locked cache can be unlocked using the UnlockSheet or UnlockSheetAll member function. The UnlockSheet member function unlocks the cache by specifying character code just like the LockSheet member function. Sheets that have been unlocked can also be completely eliminated from the cache using an option. The UnlockSheetAll member function unlocks all caches.

6 bcfnt Sheet Format

The glyphs in the bcfnt (font resource) are stored in the form of a "sheet," which is the name for a kind of CTR-format texture image. Normally, one bcfnt contains a number of sheets, each of which holds a number of glyph images. All of these sheets share the same texture format and size.

Figure 6-1Figure 6-1 Placement of Cells in Sheet shows how cells are positioned on sheets. In the figure, the individual cells are represented by broken-line rectangles. The cells on all of the sheets in a single bcfnt all have the same size. A glyph is drawn left-justified inside each cell. The terms written in italics in the figure correspond to items in Code 6-1.

Figure 6-1 Placement of Cells in Sheet



Use the `Font::GetGlyph` function to obtain the information needed to display the glyph images in a common format regardless of which derived class the instance represents. The definition of the `Glyph` structure used by the `GetGlyph` function is shown in Code 6-1.

Code 6-1 Type nw::font::Glyph

```
struct Glyph
{
    const void*  pTexture;   // Pointer to sheet that contains target glyph
    CharWidths   widths;     // Character width information
    u8           height;     // Cell height
    TexFmt     texFormat;  // Sheet's texture format
    u16          texWidth;   // Sheet width
    u16          texHeight;  // Sheet height
    u16          cellX;      // X coordinate of upper left corner of cell in sheet
    u16          cellY;      // Y coordinate of upper left corner of cell in sheet
      ...

};
```

As shown in Code 6-2, you can use the values stored in the `Glyph` structure to display polygons on which glyph-image textures are mapped. This assumes that the shader and vertex attributes etc. are set appropriately,.

Code 6-2 Example: Using glyph structure data to draw characters

```
void DrawGlyph(const Font& font, const Glyph& g, f32 x, f32 y)
{
    // Load texture
    {
        const GLint mipLevel = 0;
        // When g.texFormat = FONT_SHEET_FORMAT_A4
        GLenum format = GL_ALPHA_NATIVE_DMP;
        GLenum type = GL_UNSIGNED_4BITS_DMP;
        glTexImage2D(GL_TEXTURE_2D, mipLevel, format, g.texWidth, g.texHeight, 0,
format, type, g.pTexture);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
            font.IsLinearFilterEnableAtSmall() ? GL_LINEAR: GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
            font.IsLinearFilterEnableAtLarge() ? GL_LINEAR: GL_NEAREST);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_LOD_BIAS, 0.0f);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_LOD, -1000);
    }

    // Draw polygons
```

```
    const f32 w         = g.widths.glyphWidth;
    const f32 h         = g.height;

    const f32 posLeft   =  x
        + g.widths.left;  // Add left-space width

    const f32 posTop    = y
        + font.GetAscent() - font.GetBaselinePos(); // Take ascent into
consideration

    const f32 posRight  = posLeft + w;
    const f32 posBtm    = posTop + h;

    const f32 posZ      = 0.0f;
    const f32 texLeft   = static_cast<f32>(g.cellX) / g.texWidth;
    const f32 texTop    = static_cast<f32>(g.texHeight - g.cellY) / g.texHeight;
    const f32 texRight = static_cast<f32>(g.cellX + w) / g.texWidth;
    const f32 texBtm    = static_cast<f32>(g.texHeight - (g.cellY + h))
                                / g.texHeight;

    VertexAttribute vtxAttrs[VERTEX_MAX];

    // Set the position
    {
        vtxAttrs[VERTEX_RT].pos.x = posRight;
        vtxAttrs[VERTEX_RT].pos.y = posTop;

        vtxAttrs[VERTEX_LT].pos.x = posLeft;
        vtxAttrs[VERTEX_LT].pos.y = posTop;

        vtxAttrs[VERTEX_LB].pos.x = posLeft;
        vtxAttrs[VERTEX_LB].pos.y = posBottom;

        vtxAttrs[VERTEX_RB].pos.x = posRight;
        vtxAttrs[VERTEX_RB].pos.y = posBottom;
    }

    // Set the texture coordinates
    {
        vtxAttrs[VERTEX_RT].tex.x = texRight;
        vtxAttrs[VERTEX_RT].tex.y = texTop;
```

```
        vtxAttrs[VERTEX_LT].tex.x = texLeft;
        vtxAttrs[VERTEX_LT].tex.y = texTop;

        vtxAttrs[VERTEX_LB].tex.x = texLeft;
        vtxAttrs[VERTEX_LB].tex.y = texBottom;

        vtxAttrs[VERTEX_RB].tex.x = texRight;
        vtxAttrs[VERTEX_RB].tex.y = texBottom;
    }

    glDrawArrays(GL_TRIANGLE_FAN, 0, VERTEX_MAX);

    // Comments:
    //
    //   When proceeding to the following character position:
    //     x += g.widths.charWidth;
    //
    //   When proceeding to the following character position:
    //     y += font.GetLineFeed();
}
```

7 Revision History

Version	Revision Date	Description
1.0.0	2010/07/29	• Changed format and revised sample code.
	2010/10/29	• Deleted unnecessary blank pages
	2010/01/15	• Revised the content of figures.
	2009/10/30	• Initial version.