Higher Quality 2D Text Rendering

Nicolas P. Rougier

Higher Quality 2D Text Rendering

Nicolas P. Rougier
INRIA

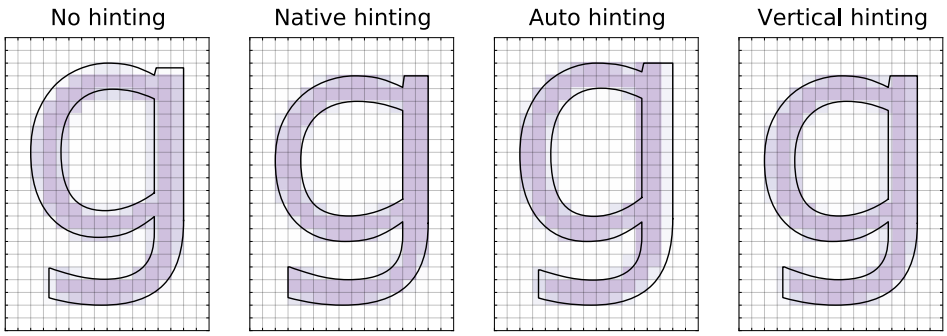

Figure 1. When displaying text on low-resolution devices (DPI < 150), one typically has to decide if one wants to respect the pixel grid (e.g., Cleartype technology / Microsoft / native hinting) for crisp rendering or, to privilege glyph shapes (Quartz technology / Apple / no hinting) at the cost of blurring. There is, however, a third way that may combine the best of the two technologies (vertical hinting).

Abstract

Even though text is pervasive in most 3D applications, there is surprisingly no native support for text rendering in OpenGL. To cope with this absence, Mark Kilgard introduced the use of texture fonts [Kilgard 1997]. This technique is well known and widely used and ensures both good performances and a decent quality in most situations. However, the quality may degrade strongly in orthographic mode (screen space) due to pixelation effects at large sizes and to legibility problems at small sizes due to incorrect hinting and positioning of glyphs. In this paper, we consider font-texture rendering to develop methods to ensure the highest quality in orthographic mode. The method used allows for both the accurate rendering and positioning of any glyph on the screen. While the method is compatible with complex shaping and/or layout (e.g., the Arabic alphabet), these specific cases are not studied in this article.

Figure 2. From left to right: bitmap font (`GLUT_BITMAP_HELVETICA_12`), stroke font (`GLUT_STROKE_ROMAN`), texture font (Bitstream Vera, 64 points), signed distance field font (Bitstream Vera, 32 points), and vector font (Bitstream Vera).

1. Introduction

Mark Kilgard [1997] introduced a simple OpenGL-based API for texture-mapped text. His method packs many rasterized glyphs into a single alpha-only texture map. It then uses a lookup table to assign texture coordinates to a quadrilateral to extract a glyph when rendering. This approach yielded several advantages over the previous approaches since it allowed arbitrary rotation, scaling, and projection over a 3D surface. More importantly, it allowed for arbitrary-shaped glyphs that can be loaded directly from standard bitmap font files (e.g., X11 font files in the accompanying code), so that font designers could work with standard tools instead of writing explicit code to render glyph shapes. The primary drawback of the method was that, at high resolution, text became quite pixelated, even using bi-linear texture interpolation (see Figure 2). To cope with this problem, Chris Green [2007] refined the method by computing a (non-adaptive) signed distance field in place of the rasterized glyph that could later be alpha-tested or thresholded. Furthermore, Gustavson [2012] showed that the distance field can be computed efficiently. Unfortunately, at very high resolution, artifacts still appear and it becomes necessary to use true vector fonts to achieve flawless rendering. Before the advent of programmable shaders, this was performed by approximating Bézier curves with many line segments, which was computationally expensive. However, Loop and Blinn [2005] introduced a new approach for resolution-independent rendering of quadratic and cubic spline curves. By tessellating a glyph the proper way, they offered *de facto* a method for resolution-independent rendering of a glyph with good rendering quality (anti-aliasing is also supported). More recently, Kilgard [2012] proposed a two-step method (stencil and cover) for GPU-accelerated path rendering as an OpenGL extension. As stated by the authors, their goals are completeness, correctness, quality, and performance; the extension actually covers most of the OpenVG specifications [2008] while giving very high performances.

While such vector methods may appear to be the definitive solution for text rendering, they suffer from one drawback that is inherent to their nature: they cannot enforce hinting. This is an expected and unavoidable effect since they promote

resolution-independence while hinting is specifically related to a given pixel grid during the rasterization process (see Section 2.2 for further explanation). Historically, hinting (together with anti-aliasing) was introduced to address very-low resolution scenarios, where pixels are visible to the unaided eye and a mechanical rasterization may produce results that are not actually indicative of letter forms to a human viewer, even though they have the mathematical optimal grayscale shades for representing the underlying shapes. For example, modern printers offer resolution up to 600 DPI (dots per inch) at which antialiasing or hinting is not required, but many screens today offer fewer than 100 PPI (pixels per inch) although some (e.g., *retina* displays) offer resolutions higher than 300 PPI. Until very-high-resolution displays are universal, hinting must be enforced for small glyphs in most applications. To cope with this problem, we must revisit software rasterization for a given pixel grid. Maxim Shemarev [2007] analyzed font rendering on different systems and software and arrived at a set of four principles:

1. Use horizontal RGB sub-pixel anti-aliasing for LCD flat panels.
2. Use vertical hinting only and completely discard the horizontal one.
3. Use accurate glyph advance values from unhinted glyph.
4. Use accurate, high-resolution values from the kerning table.

We will now explain how to enforce these principles in the context of texture fonts for simple layouts and shaping (considering primarily Western writing, in which one letter corresponds to one glyph).

2. Textured Text

The idea of textured text is well known and has been used in many projects. The main idea is to render rasterized glyphs into one or more textures and to render text using textured quads. There are many variants, since you can use a texture for a whole sentence, a word, or a glyph. Furthermore, one can either use a dedicated texture for each glyph or pack them into a single texture. We retain the latter technique since it will allow us to display a whole text using a single texture and a single vertex buffer. To access font files, we will use the well-known FreeType library [Turner et al. 1996] which is widely deployed on a number of platforms. This library provides a simple and easy-to-use API to access font content in a uniform way, independently of the file format. Furthermore, FreeType also provides a font rasterization engine that produces high-quality output. An alternative library worth noting is Barret's minimalist TrueType font rasterizer [2009] (one file) that can parse TrueType files and extract metrics and shapes with kerning information and perform subpixel rendering.

2.1. Texture Atlas

Storing glyphs efficiently in a two-dimensional texture is a bin packing problem, which is a classical problem in combinatorial optimization and is known to be NP-hard. Given a sequence of rectangles $(R_1, R_2, \ldots, R_n)$, $R_i = (w_i, h_i)$, the task is to find a packing $(P_1, P_2, \ldots, P_n)$, $P_i = (x_i, y_i)$ of these items such that no two rectangles may intersect or be contained inside one another while minimizing the number of bins of size (W, H). This problem has been studied extensively in the literature and several algorithms are known whose performances may vary depending on the conditions and parameters of the problem. In our case, the problem is *online packing* where items are to be packed immediately in the bin without knowledge of the upcoming items. We need to store new glyphs just in time, when they are actually needed. Jukka Jylänki [2010] offers an extensive survey of the two-dimensional bin-packing methods and provides benchmarks of algorithm efficiency. We chose the Skyline Bottom-Left al-

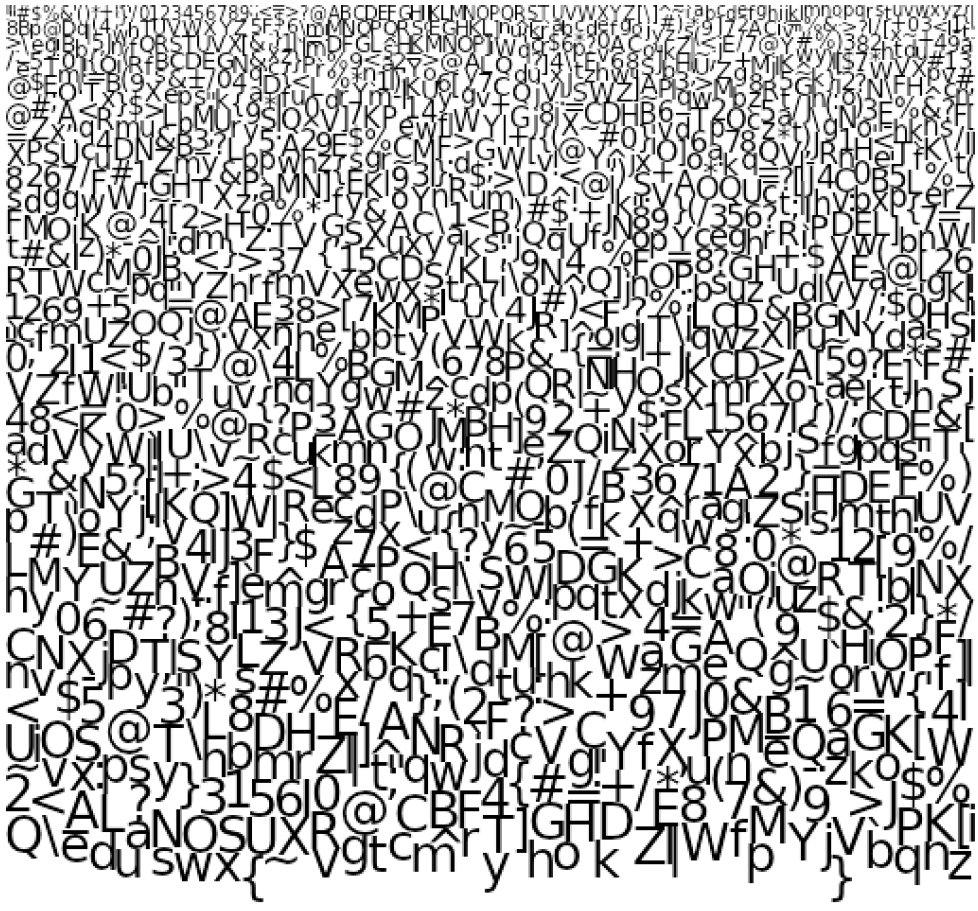

Figure 3. This 512 × 512 texture contains 1900 glyphs (20 fonts × 95 characters) from the Bitstream Vera Sans font at sizes 8 to 28 points (DPI has been set to 72).

gorithm based on this survey. In our experience, it seems to be particularly well-suited for quickly storing random glyphs in an efficient way. Figure 3 shows the result of packing more than 1900 glyphs into a single 512×512 texture, comprising the entire ASCII set for 20 different font sizes.

2.2. Hinting

Font hinting is the use of specific instructions (hints) to adjust the control points defining the outline of a glyph so that it lines up gracefully with the raster grid. At low screen resolutions, hinting is critical for producing clear and legible text as shown in Figure 4. When no hinting instructions are present in the font file, it is possible to use auto-hinting mechanisms such as the one available in FreeType[1].

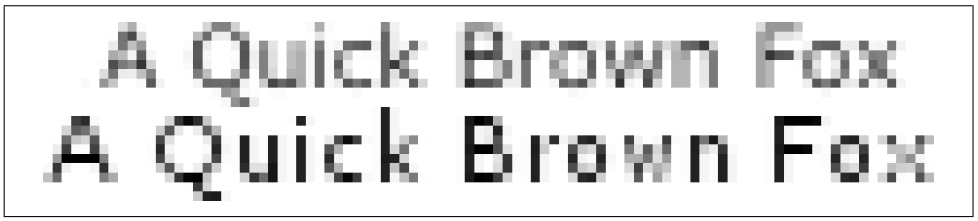

Figure 4. Unhinted text (top) and hinted text (bottom) using the Verdana font (with handcrafted hinting instructions). The hinted version is crisper, but it does not respect the glyph shapes and leads to longer lines.

To render high-quality text, Shemarev [2007] recommends using only vertical hinting and completely discarding the horizontal hints. Hinting is the responsibility of the rasterization engine (FreeType in our case) which provides no option to specifically discard horizontal hinting. In the case of the FreeType library, we can nonetheless *trick* the engine by specifying an oversized horizontal DPI (100 times the vertical) while specifying a transformation matrix that scale down the glyph as shown in Listing 1.

```
float size = 32, scale = 100.0;
FT_Matrix matrix = { (int)((1.0/scale) * 0x10000L),
                     (int)((0.0) * 0x10000L),
                     (int)((0.0) * 0x10000L),
                     (int)((1.0) * 0x10000L) };
FT_Set_Char_Size( face, (int)(32*64), 0, 72*scale, 72 );
FT_Set_Transform( face, &matrix, NULL );
```

Listing 1. Vertical hinting using FreeType.

[1]It is also possible to add hints to the font file using external tools such as the true type autohinter [Lemberg and Crossland 2011]

The result, shown in Figure 1 (far right), is a glyph that is only hinted vertically. This *trick* will allows us to ensure an accurate horizontal text positioning at the price of some inaccuracy in vertical positioning.

2.3. Subpixel Positioning

The FreeType library can rasterize a glyph using sub-pixel anti-aliasing in RGB mode. However, this is only half of the problem, since we also want to achieve sub-pixel positioning for accurate placement of the glyphs. Displaying the textured quad at

```
uniform sampler2D texture;
uniform vec2 pixel;
varying float shift;
void main()
{
    vec2 uv       = gl_TexCoord[0].xy;
    vec4 current  = texture2D(texture, uv);
    vec4 previous = texture2D(texture, uv+vec2(-1,0)*pixel);
    float r = current.r;
    float g = current.g;
    float b = current.b;
    float a = current.a;
    if( shift <= 1.0/3.0 )
    {
        float z = 3.0*shift;
        r = mix(current.r, previous.b, z);
        g = mix(current.g, current.r,  z);
        b = mix(current.b, current.g,  z);
    }
    else if( shift <= 2.0/3.0 )
    {
        float z = 3.0*shift-1.0;
        r = mix(previous.b, previous.g, z);
        g = mix(current.r,  previous.b, z);
        b = mix(current.g,  current.r,  z);
    }
    else if( shift < 1.0 )
    {
        float z = 3.0*shift-2.0;
        r = mix(previous.g, previous.r, z);
        g = mix(previous.b, previous.g, z);
        b = mix(current.r,  previous.b, z);
    }
    gl_FragColor = vec3(r,g,b,a);
}
```

Listing 2. Subpixel positioning fragment shader.

$0 < s \leq 1/3, t = 3s$	$R_{\text{out}} = tB_{\text{left}} + (1-t)R$ $G_{\text{out}} = tR + (1-t)G$ $B_{\text{out}} = tG + (1-t)B$
$1/3 < s \leq 2/3, t = 3s-1$	$R_{\text{out}} = tG_{\text{left}} + (1-t)B_{\text{left}}$ $G_{\text{out}} = tB_{\text{left}} + (1-t)R$ $B_{\text{out}} = tR + (1-t)G$
$2/3 < s < 1, t = 3s-2$	$R_{\text{out}} = tR_{\text{left}} + (1-t)G_{\text{left}}$ $G_{\text{out}} = tG_{\text{left}} + (1-t)B_{\text{left}}$ $B_{\text{out}} = tB_{\text{left}} + (1-t)R$

Table 1. The three different cases of subpixel positioning depend on the amount of rightward shift, s.

fractional pixel coordinates does not solve the problem, since it only results in texture interpolation at the whole-pixel level. Instead, we want to achieve a precise shift (between 0 and 1) in the subpixel domain. This can be done in a fragment shader, by considering three different levels of rightward shift, s, as illustrated in Table 1. The corresponding fragment shader code is straightforward (see Listing 2).

Note that in some circumstances, subpixel rendering can lead to more or less severe color fringes that can be partly removed by redistributing energy among the pixels. The compromise between blurriness and color fringes can be tuned by the `FT_Library_SetLcdFilter` in the FreeType library.

2.4. Kerning and Advance Values

Kerning refers to the manual or automatic adjustment of the space between characters to achieve a pleasant visual result (see Figure 5). The kerning value is added to the advance value of a glyph, that defines the distance the (virtual) pen must advance before displaying the next glyph (see Figure 6). Kerning is specified for each (non-permutable) character pair and is given as a distance by which to increase or decrease the spacing between the two characters. Font files can have an optional *kern* table that specifies how glyph pairs should be kerned. However, the number of fonts that

Figure 5. Kerning in Roman alphabet occurs most often with capital letters.

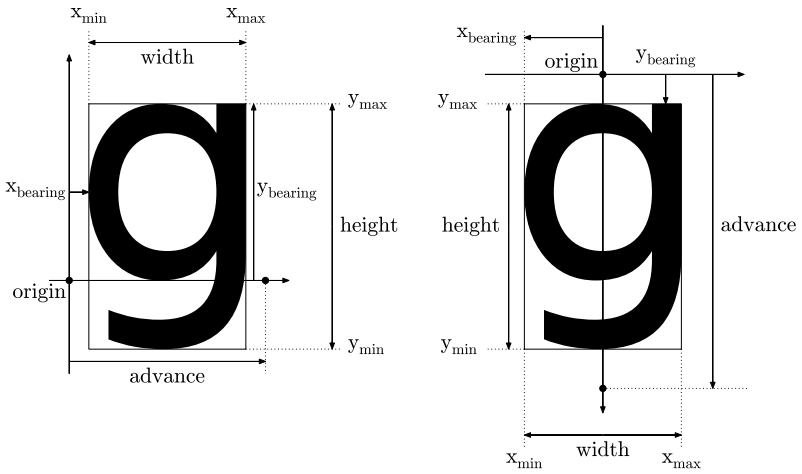

Figure 6. Glyph metrics for horizontal layout (left) and vertical layout (right).

include *kern* tables is diminishing in favor of the glyph positioning table (GPOS) specification that is more sophisticated, yet more complex to process (OpenType). We'll stick to the *kern*-table approach, since it is accessible directly from the FreeType API while the GPOS table would require an additional library such as HarfBuzz or Pango. This allows us easy access to the kerning information and advance values from high resolution unhinted glyphs as required by the proposed method.

2.5. Gamma Correction

As explained in the Gamma FAQ, *the intensity of light generated by a physical device is not usually a linear function of the applied signal*; rather, the visual perception of brightness is approximately proportional to the square root of the physical luminosity. This difference is critical for text rendering since it dramatically affects our perception of the text, especially at small sizes. Since brightness perception is not linear, a black text over a white background is not the opposite of a white text over a black background. This means that we must adjust gamma correction on a per-glyph basis and not only at the frame-buffer level.

3. Conclusion

We have shown shown how to achieve high-quality text rendering following the principles described by Shemarev [Shemanarev 2007] (see Figure 7). The implementation is both fast (two triangles/glyph) and keeps memory footprints to a minimum for a given pair of font family and size. If one needs frequent (e.g., dynamic) text scaling or needs to use text under perspective projection, signed texture fonts may be a better option. In orthographic space, a mixed approach using the two aforementioned techniques could be an ideal compromise in speed, quality, and memory. For medium

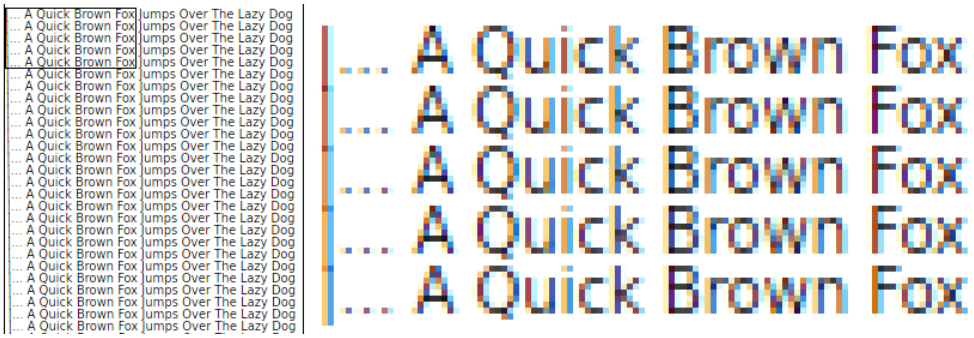

Figure 7. Left: the same text has been repeated 30 times shifted by an additional 0.1 pixels at each line. Right: a detail of the first five lines.

to high resolutions, the vector approach should be favored, especially if accuracy is a concern as in CAD software. Finally, we did not handle the case of composite glyphs, ligature or complex text layout that are mandatory in some languages. This is a complex problem that requires a full shaping engine such as the HarfBuzz or Pango library and is beyond the scope of this article.

4. Appendix: Implementation

All the screenshots from this article have been made using the freetype-gl project available from `http://code.google.com/p/freetype-gl/` under the terms of the new BSD license.

The following Skyline Bottom-Left bin packer algorithm in Python (see Listing 3) demonstrates higher-quality 2D texture font using subpixel rendering and positioning. The output of the demo is illustrated in the left part of Figure 7. The demo requires the following resources to execute:

- FreeType library with the subpixel rendering option (check for `FT_CONFIG_ OPTION_SUBPIXEL_RENDERING` in `freetype/config/ftoption.h`);
- numpy library available from `http://numpy.scipy.org`;
- OpenGL Python bindings available from `http://pyopengl.sourceforge.net`.

The demo download comprises the following source files and documentation:

- **README** Readme file
- **shader.py** Class to handle shaders
- **demo.py** Actual demo code
- **atlas.py** Texture atlas, texture font and texture glyph definition
- **freetype/** Python FreeType bindings

```
#!/usr/bin/env python
import sys

class Bin:
    '''
    Bin packer class
    ================
    The algorithm is based on the article by Jukka JylaÌĹnki : "A Thousand
    Ways to Pack the Bin - A Practical Approach to Two-Dimensional
    Rectangle Bin Packing", February 27, 2010.

    Example usage:
    --------------
    bin = Bin(512,512)
    region = bin.pack(20,20)
    region = bin.pack(10,10)
    ...
    '''

    def __init__(self, width=1024, height=1024):
        """ Initialize a new bin of given size. """
        self.width  = width
        self.height = height
        self.nodes  = [ (0,0,self.width), ]
        self.used   = 0

    def fit(self, index, width, height):
        """ Test if region (width,height) fits into self.nodes[index]"""
        node = self.nodes[index]
        x,y = node[0], node[1]
        width_left = width

        if x+width > self.width:
            return -1
        i = index
        while width_left > 0:
            node = self.nodes[i]
            y = max(y, node[1])
            if y+height > self.height:
                return -1
            width_left -= node[2]
            i += 1
        return y

    def merge(self):
        """ Merge nodes where possible """
        i = 0
        while i < len(self.nodes)-1:
            node = self.nodes[i]
            next_node = self.nodes[i+1]
            if node[1] == next_node[1]:
                self.nodes[i] = node[0], node[1], node[2]+next_node[2]
                del self.nodes[i+1]
            else:
                i += 1
```

```
def pack(self, width, height):
    """
    Try to pack the given region (width,height) of given size and return
    the newly allocated region as (x,y,width,height) or None
    """
    best_height = sys.maxint
    best_index = -1
    best_width = sys.maxint
    region = 0, 0, width, height
    for i in range(len(self.nodes)):
        y = self.fit(i, width, height)
        if y >= 0:
            node = self.nodes[i]
            if (y+height < best_height or
                (y+height == best_height and node[2] < best_width)):
                best_height = y+height
                best_index = i
                best_width = node[2]
                region = node[0], y, width, height
    if best_index == -1:
        return None
    node = region[0], region[1]+height, width
    self.nodes.insert(best_index, node)

    i = best_index+1
    while i < len(self.nodes):
        node = self.nodes[i]
        prev_node = self.nodes[i-1]
        if node[0] < prev_node[0]+prev_node[2]:
            shrink = prev_node[0]+prev_node[2] - node[0]
            x,y,w = self.nodes[i]
            self.nodes[i] = x+shrink, y, w-shrink
            if self.nodes[i][2] <= 0:
                del self.nodes[i]
                i -= 1
            else:
                break
        else:
            break
        i += 1
    self.merge()
    self.used += width*height
    return region
```

Listing 3. Skyline Bottom-Left bin packing algorithm

Glossary

advance value is the horizontal distance by which the pen position must be incremented (for left-to-right writing) or decremented (for right-to-left writing) after each glyph is rendered when processing text (from FreeType definition).

aliasing Aliasing refers to the distortion or artifacts that results when a continuous signal is sampled into discrete samples as shown in Figure 8.

Figure 8. The same glyph rendered at different resolutions (16pt, 24pt, 32pt, 48pt, 64pt, and 128pt) without antialiasing.

anti-aliasing refers to a number of methods that aim at attenuating aliasing artifacts as shown in Figure 9.

Figure 9. The same glyph rendered at different resolutions (16pt, 24pt, 32pt, 48pt, 64pt,and 128 pt) with greyscale antialiasing.

complex text layout is required for some writing systems such as, for example, the Arabic alphabet. It generalizes the concept of ligatures with quite complex transformations between text input and text display to ensure proper rendering (see Figure 10).

Figure 10. Some language requires a complex layout with a lot of transformations of the base glyphs.

glyph is the visual representation of a letter, character, or symbol, for a specific language in a specific font and style as illustrated in Figure 11.

Figure 11. From left to right, a Maya symbol, a Japanese character (kanji), a Roman letter, and a mathematical symbol.

hinting is the use of specific instructions (hints) that modify the control points that define the outline of a glyph such that it lines up with a rasterized grid. At low screen resolutions, hinting is critical for producing a clear and legible text (see Figure 1).

kerning refers to the manual or automatic adjustment of the spacing between characters that would be otherwise too distant from each other leading to an unpleasing visual result as illustrated in Figure 12.

Figure 12. Kerning in Roman alphabet occurs most often with capital letters.

ligature is the union of two or more glyphs in order to form a single glyph as shown in Figure 13. Ligatures are generally considered purely aesthetic or ornamental and more complex combinations are handled by a complex text layout (CTL).

Figure 13. Some common ligatures found in the Roman alphabet.

subpixel rendering is a rendering method that increase the apparent resolution of a display by exploiting the arrangement of physical elements that compose a single pixel.

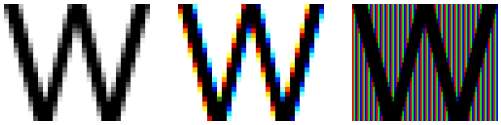

Figure 14. Subixel rendering allows for perceptually more accurate anti-aliasing. From left to right: grayscale rendering, subpixel rendering, simulated LCD display.

References

BARRETT, S., 2009. STB Truetype. http://nothings.org/stb/stb_truetype.h. 52

GREEN, C. 2007. Improved Alpha-Tested Magnification for Vector Textures and Special Effects. In *ACM SIGGRAPH 2007 courses*, ACM, New York, NY, USA, SIGGRAPH '07, 9–18. 51

GUSTAVSON, S. 2012. 2D Shape Rendering by Distance Fields. In *OpenGL Insights*, CRC Press, Boca Raton, FL, P. Cozzi and C. Riccio, Eds., 173–182. 51

JYLÄNKI, J., 2010. A Thousand Ways to Pack the Bin - A Practical Approach to Two-Dimensional Rectangle Bin Packing. http://clb.demon.fi/files/RectangleBinPack.pdf. 53

KHRONOS GROUP, 2008. OpenVG: The Standard for Vector Graphics Acceleration. http://www.khronos.org/openvg/. 51

KILGARD, M., AND BOLZ, J. 2012. GPU-Accelerated Path Rendering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2012) 31*, 6, 172:1–172:10. 51

KILGARD, M., 1997. A Simple OpenGL-based API for Texture Mapped Text, Silicon Graphics. http://reality.sgi.com/opengl/tips/TexFont/TexFont.html. 50, 51

LEMBERG, W., AND CROSSLAND, D., 2011. TTF Autohint. http://www.freetype.org/ttfautohint/doc/ttfautohint.html. 54

LOOP, C., AND BLINN, J. 2005. Resolution Independent Curve Rendering Using Programmable Graphics Hardware. In *ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, SIGGRAPH '05, 1000–1009. 51

SHEMANAREV, M., 2007. Texts Rasterization Exposures. http://www.antigrain.com/research/font_rasterization/. 52, 54, 57

TURNER, D., WILHELM, R., AND LEMBERG, W., 1996. The FreeType Project. http://www.freetype.org/. 52

Author Contact Information

Nicolas P. Rougier
Mnemosyne, INRIA Bordeaux - Sud Ouest
LaBRI, UMR 5800 CNRS, Bordeaux University
Institute of Neurodegenerative Diseases, UMR 5293
351, Cours de la Libération
33405 Talence Cedex, France
Nicolas.Rougier@inria.fr
http://www.loria.fr/~rougier

Nicolas P. Rougier, Higher Quality 2D Text Rendering, *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 1, 50–64, 2013
http://jcgt.org/published/0002/02/01/

Received:	2012-11-22		
Recommended:	2013-03-02	Corresponding Editor:	Tomer Moscovich
Published:	2013-04-30	Editor-in-Chief:	Morgan McGuire