Text Rendering

zhuyie
zhuyie@gmail.com

Agenda

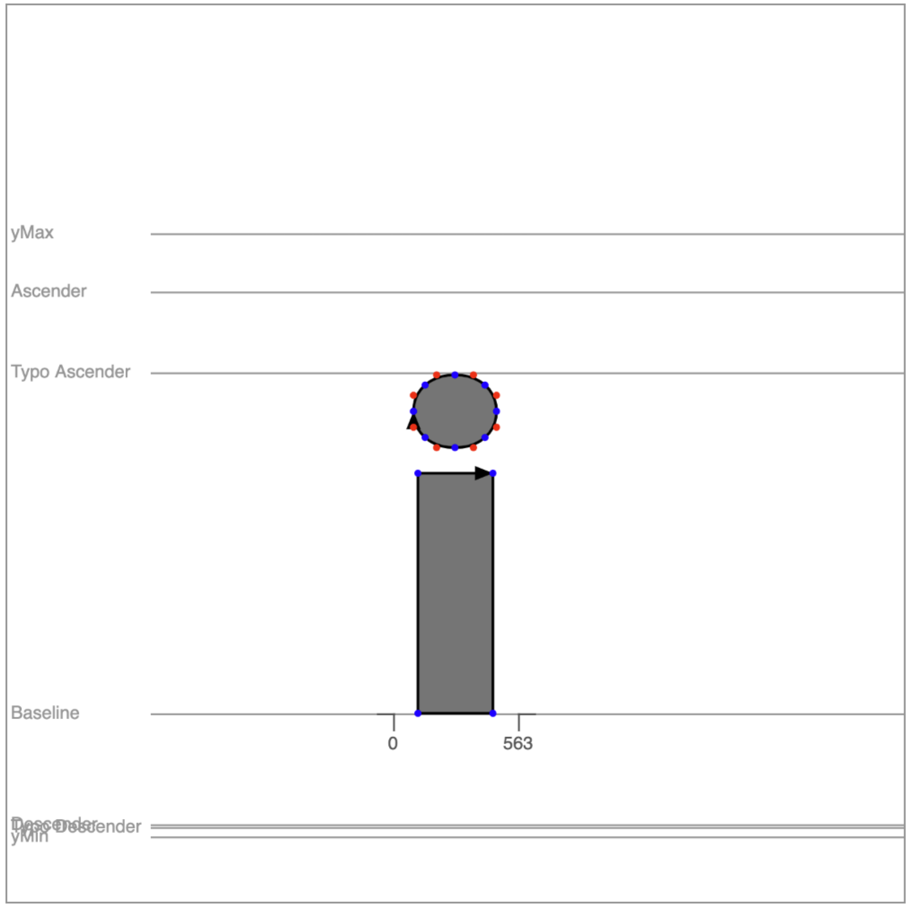
- Font basics
- Naïve path renderer
- Bitmap renderer (texture renderer)
- Hinting
- SDF renderer
- New GPU-based approaches

Font types

- A font is a collection of glyphs.
- **Bitmap** fonts consist of a matrix of dots or pixels representing the image of each glyph in each face and size.
- **Vector** fonts (outline fonts) use Bézier curves, drawing instructions and mathematical formulae to describe each glyph, which make the character outlines scalable to any size.
  - TrueType uses quadratic bezier curves.
  - OpenType(CFF) uses cubic bezier curves.

Glyph internals

Roboto-Black.ttf



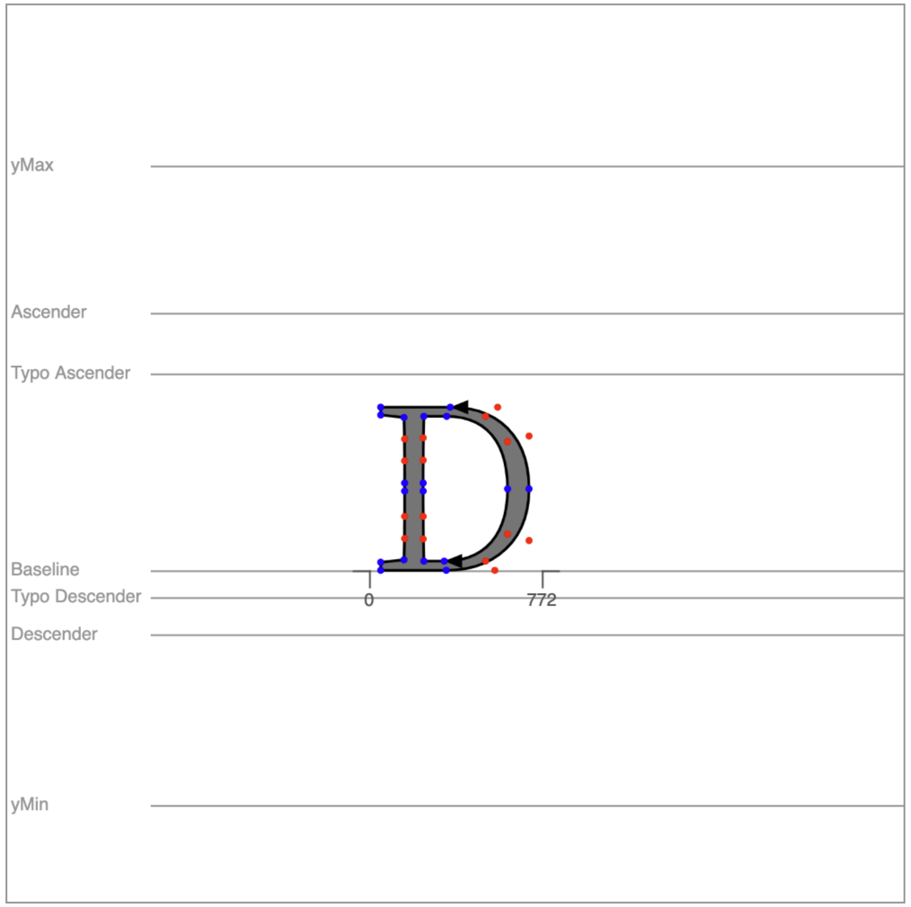
name	undefined
unicode	0069
index	77
xMin	92
xMax	466
yMin	0
yMax	1525
advanceWidth	563
leftSideBearing	92

contours:

```
x=450 y=0
x=112 y=0
x=112 y=1082
x=450 y=1082

x=92 y=1361
x=92 y=1433
x=196 y=1525
x=362 y=1525
x=466 y=1433
x=466 y=1289
x=362 y=1197
x=196 y=1197
x=92 y=1289
```

Glyph internals

NotoSerifSC-Regular.otf



name	gid37
unicode	0044
index	37
xMin	undefined
xMax	undefined
yMin	undefined
yMax	undefined
advanceWidth	772
leftSideBearing	52

path:

```
  M x=245 y=41
  C x=242 y=354 x1=242 y1=140 x2=242 y2=241
  L x=242 y=390
  C x=245 y=688 x1=242 y1=492 x2=242 y2=592
  L x=347 y=688
  C x=619 y=364 x1=521 y1=688 x2=619 y2=575
  C x=336 y=41 x1=619 y1=162 x2=521 y2=41
  Z
  M x=52 y=729
  L x=52 y=694
  L x=156 y=683
  C x=159 y=390 x1=159 y1=587 x2=159 y2=489
  L x=159 y=354
  C x=156 y=47 x1=159 y1=241 x2=159 y2=142
  L x=52 y=36
  L x=52 y=0
  L x=345 y=0
  C x=715 y=364 x1=562 y1=0 x2=715 y2=133
  C x=363 y=729 x1=715 y1=600 x2=575 y2=729
  Z
```

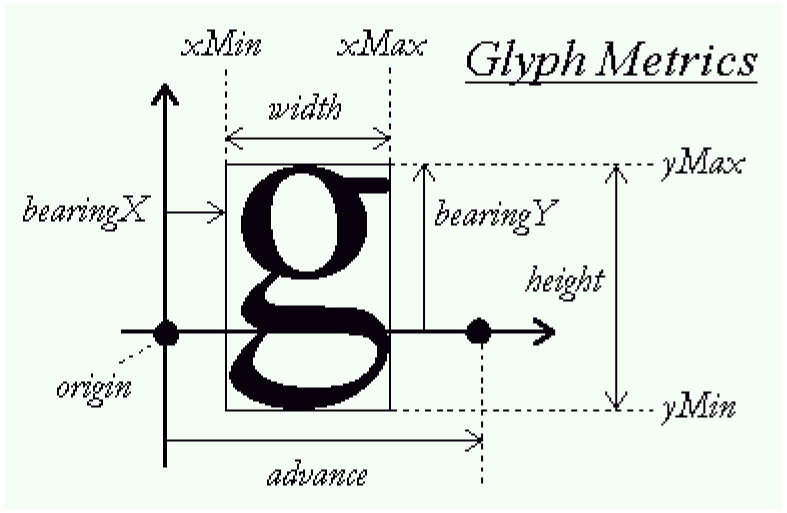
EM & font size

- In digital type, the **EM square** is a grid of arbitrary resolution that is used as the design space of a digital font.
- A 2D coordinate system, which have an **origin** and a **unit length**.
- All points in glyph description are based on this coordinate system.

EM & font size

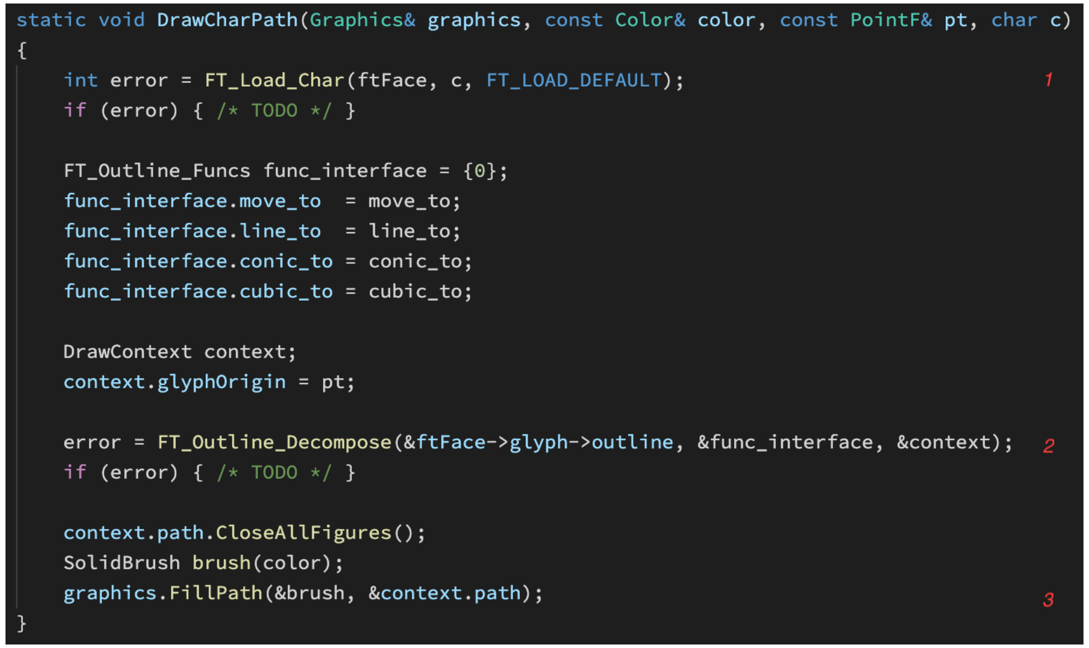
- Different fonts may have different EM size, eg. 1000 or 2048.
- 2 inches **font size** means 1 EM is 2 inches large.
- The most common used font size unit is **point**, which is 1/72 inch.

Glyph metrics



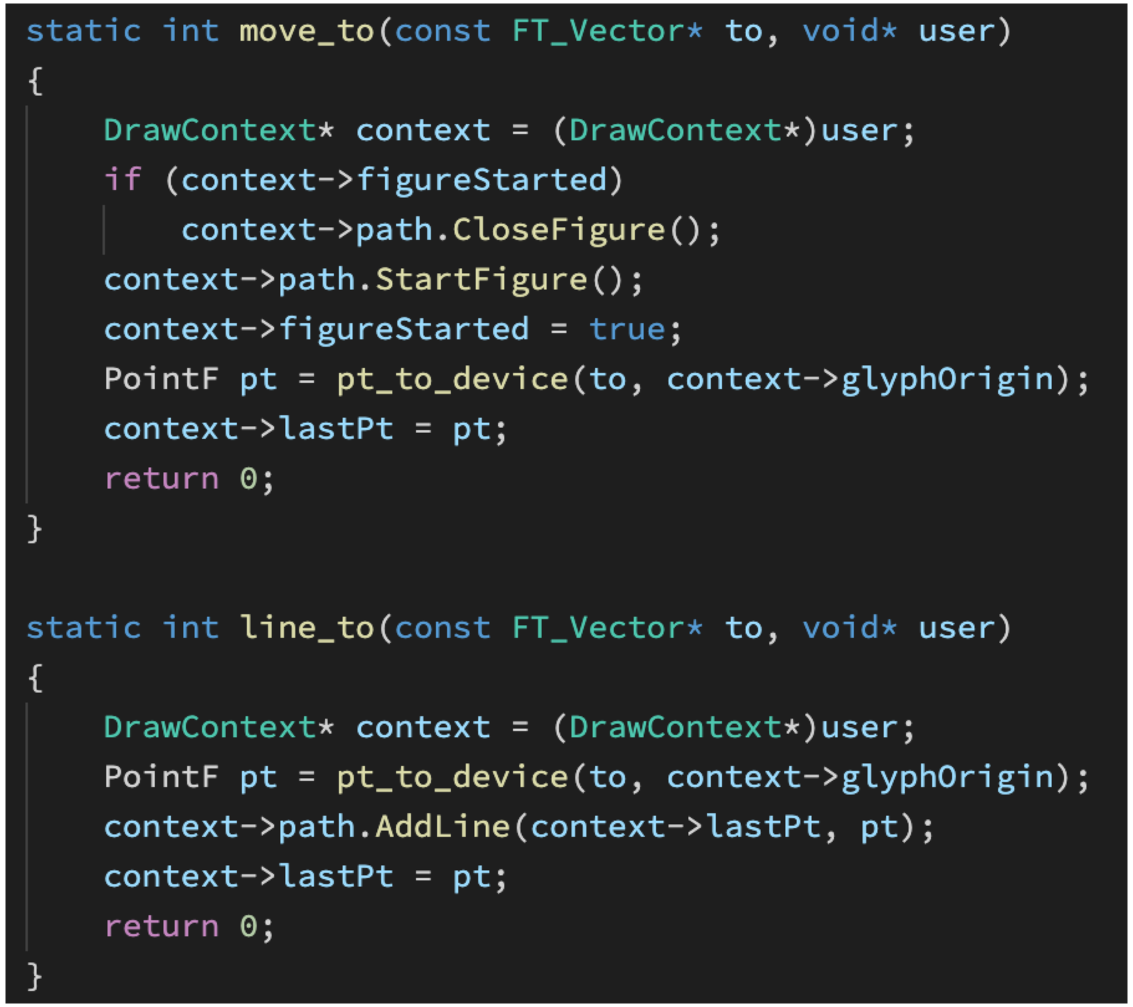
Naïve path renderer

(1) Load the glyph.
(2) Build a **path** object by calling FT_Outline_Decompose.
(3) Draw the filled path object.



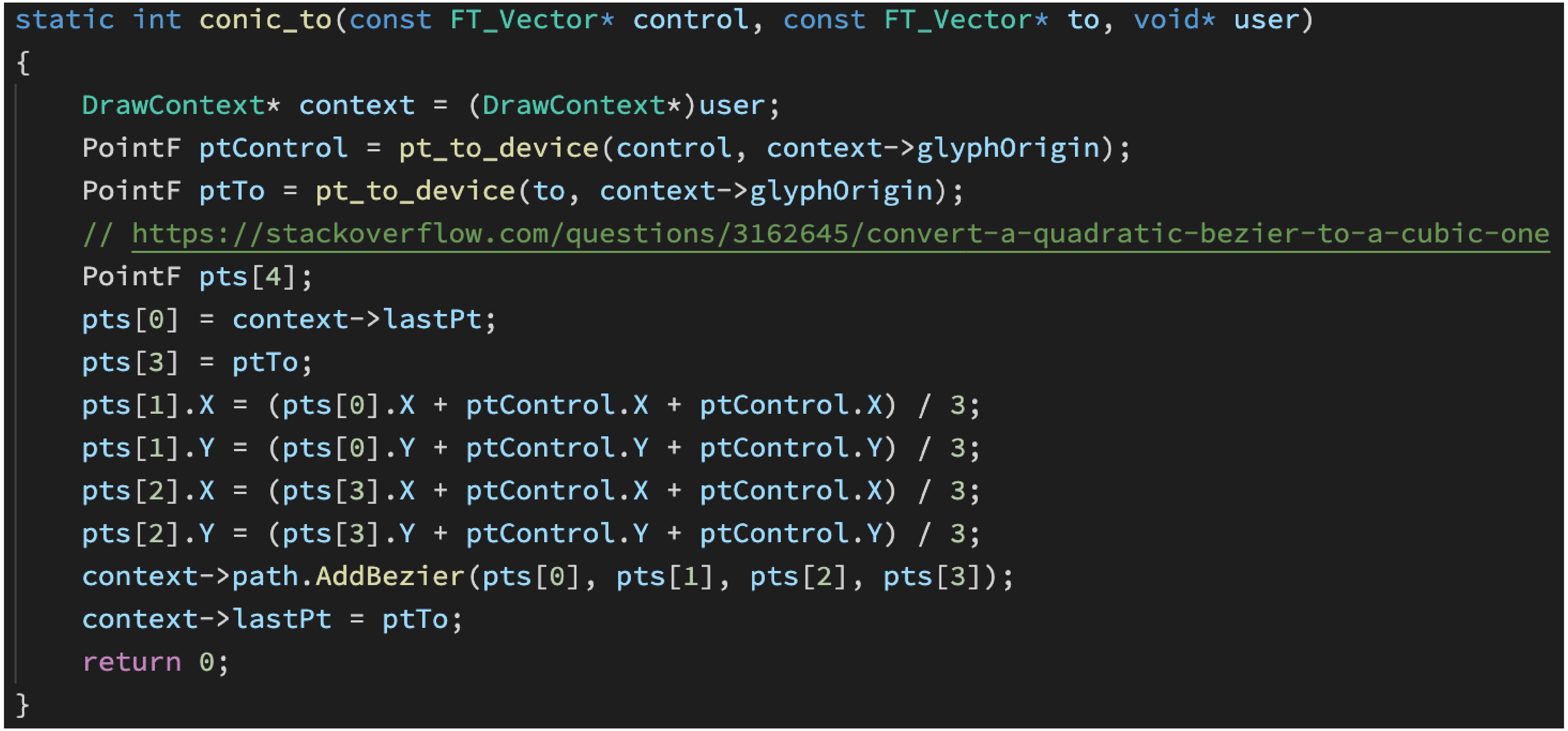
```
static void DrawCharPath(Graphics& graphics, const Color& color, const PointF& pt, char c)
{
    int error = FT_Load_Char(ftFace, c, FT_LOAD_DEFAULT);                          1
    if (error) { /* TODO */ }

    FT_Outline_Funcs func_interface = {0};
    func_interface.move_to  = move_to;
    func_interface.line_to  = line_to;
    func_interface.conic_to = conic_to;
    func_interface.cubic_to = cubic_to;

    DrawContext context;
    context.glyphOrigin = pt;

    error = FT_Outline_Decompose(&ftFace->glyph->outline, &func_interface, &context);  2
    if (error) { /* TODO */ }

    context.path.CloseAllFigures();
    SolidBrush brush(color);
    graphics.FillPath(&brush, &context.path);                                      3
}
```

Naïve path renderer



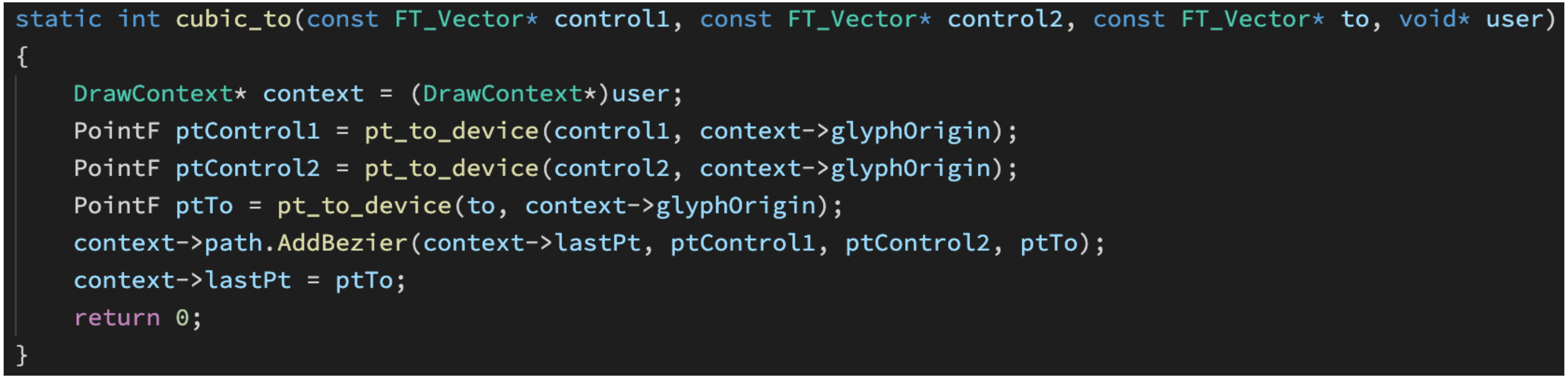
```
static int move_to(const FT_Vector* to, void* user)
{
    DrawContext* context = (DrawContext*)user;
    if (context->figureStarted)
        context->path.CloseFigure();
    context->path.StartFigure();
    context->figureStarted = true;
    PointF pt = pt_to_device(to, context->glyphOrigin);
    context->lastPt = pt;
    return 0;
}

static int line_to(const FT_Vector* to, void* user)
{
    DrawContext* context = (DrawContext*)user;
    PointF pt = pt_to_device(to, context->glyphOrigin);
    context->path.AddLine(context->lastPt, pt);
    context->lastPt = pt;
    return 0;
}
```

Naïve path renderer



```
static int conic_to(const FT_Vector* control, const FT_Vector* to, void* user)
{
    DrawContext* context = (DrawContext*)user;
    PointF ptControl = pt_to_device(control, context->glyphOrigin);
    PointF ptTo = pt_to_device(to, context->glyphOrigin);
    // https://stackoverflow.com/questions/3162645/convert-a-quadratic-bezier-to-a-cubic-one
    PointF pts[4];
    pts[0] = context->lastPt;
    pts[3] = ptTo;
    pts[1].X = (pts[0].X + ptControl.X + ptControl.X) / 3;
    pts[1].Y = (pts[0].Y + ptControl.Y + ptControl.Y) / 3;
    pts[2].X = (pts[3].X + ptControl.X + ptControl.X) / 3;
    pts[2].Y = (pts[3].Y + ptControl.Y + ptControl.Y) / 3;
    context->path.AddBezier(pts[0], pts[1], pts[2], pts[3]);
    context->lastPt = ptTo;
    return 0;
}
```

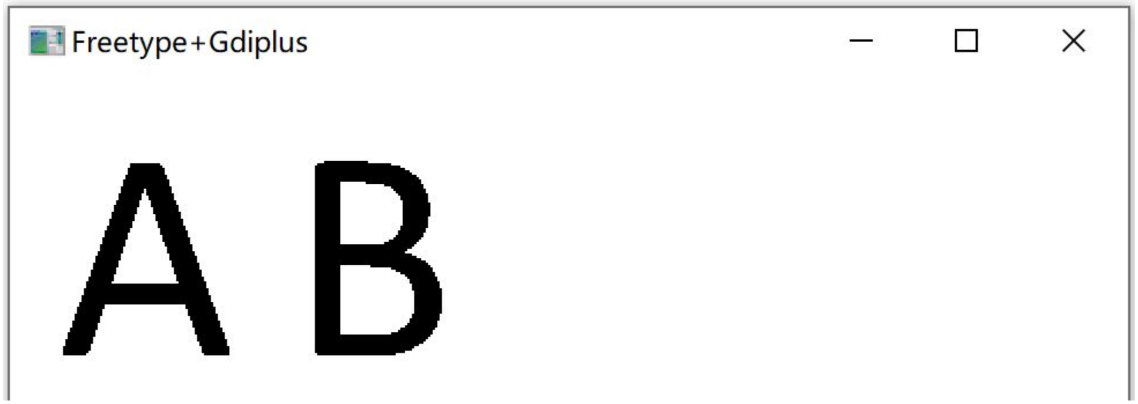
Naïve path renderer



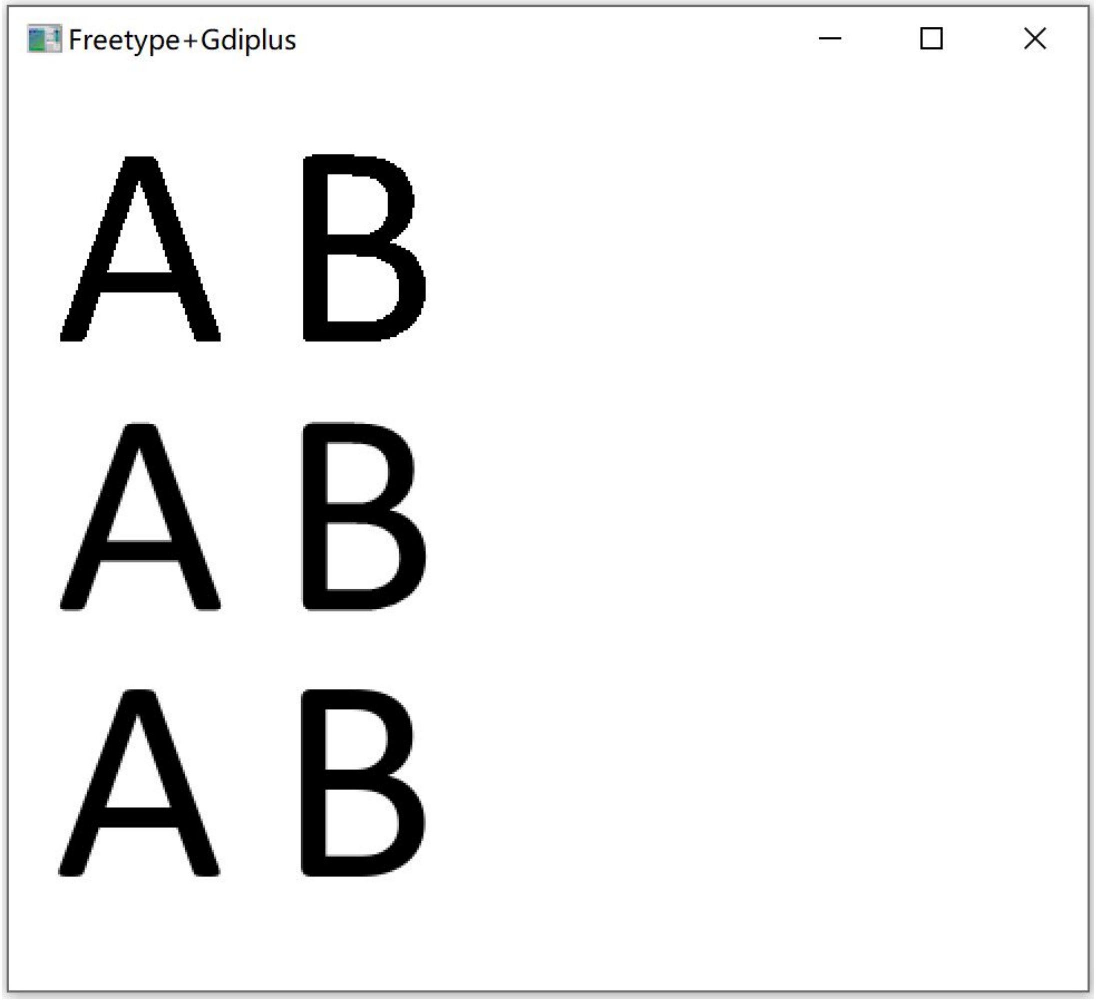
```
static int cubic_to(const FT_Vector* control1, const FT_Vector* control2, const FT_Vector* to, void* user)
{
    DrawContext* context = (DrawContext*)user;
    PointF ptControl1 = pt_to_device(control1, context->glyphOrigin);
    PointF ptControl2 = pt_to_device(control2, context->glyphOrigin);
    PointF ptTo = pt_to_device(to, context->glyphOrigin);
    context->path.AddBezier(context->lastPt, ptControl1, ptControl2, ptTo);
    context->lastPt = ptTo;
    return 0;
}
```

Naïve path renderer

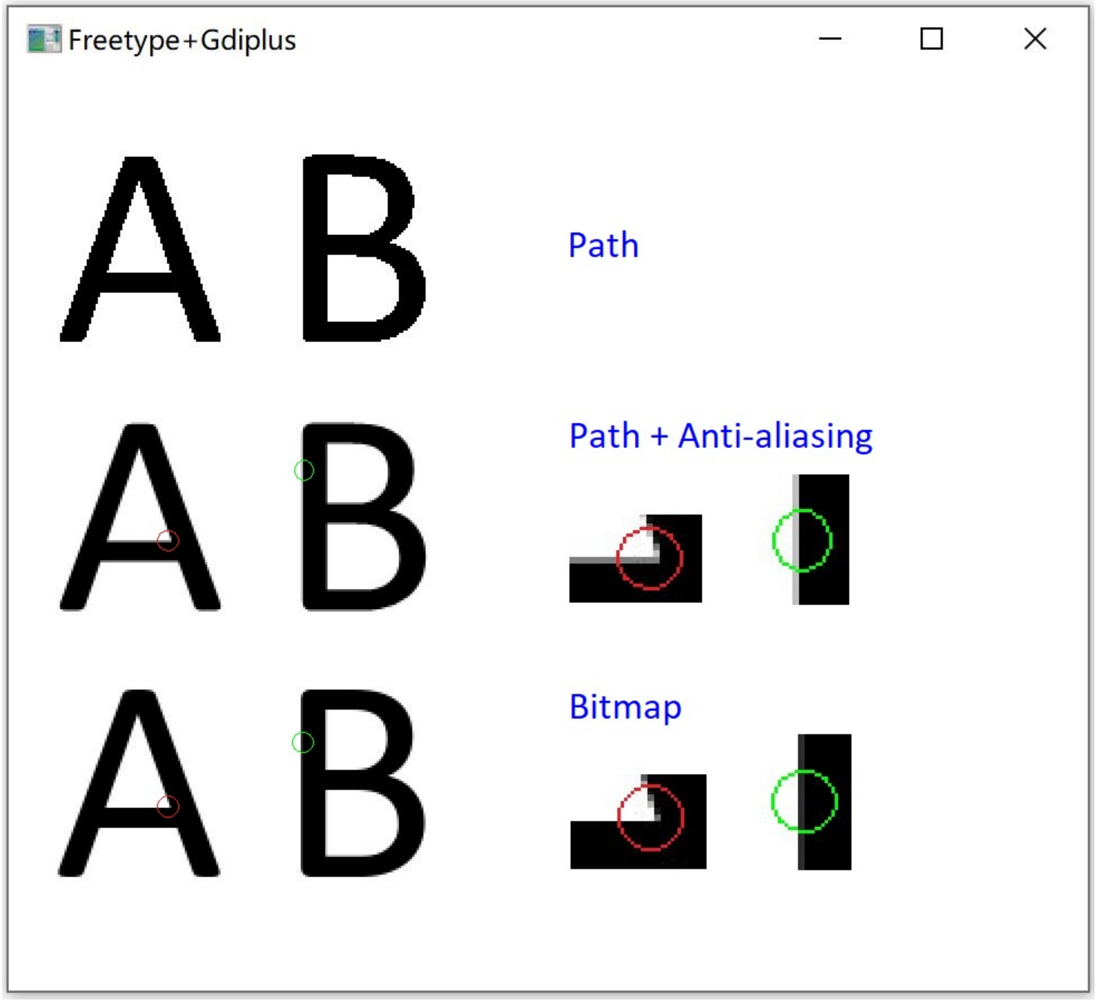
```
VOID OnPaint(HDC hdc)
{
    Graphics graphics(hdc);
    Color color(255, 0, 0, 0);

    DrawCharPath(graphics, color, PointF(20.0f, 120.0f), 'A');
    DrawCharPath(graphics, color, PointF(120.0f, 120.0f), 'B');
```



Naïve path renderer



Naïve path renderer



Naïve path renderer

- Pros
  - simple.
  - font size independent.
- Cons
  - display quality is not very good.
  - drawing lots of paths is slow.

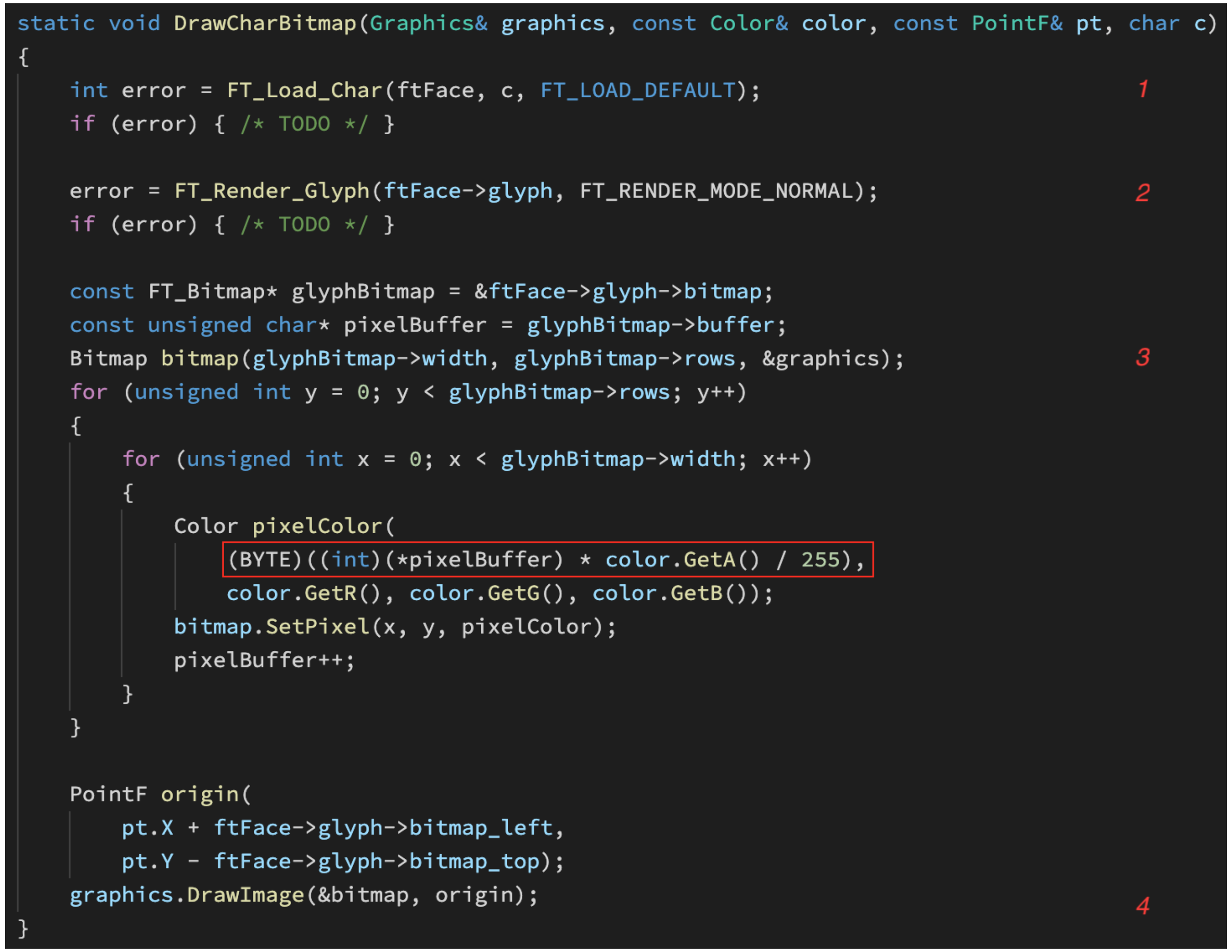
Bitmap renderer

- Although the glyph describe in **vector** form, most display devices are **raster** devices. there are always a rasterization stage.
- One glyph may displayed many instances at the same time. Use a pre-rasterized glyph and bitblt multiples times usually more efficient.
- With exact pixel boundary, we can do font hinting to improve the display quality at low screen resolutions.
- Therefore, the most common way to render glyphs is using pre-rasterized glyphs. We can call it "bitmap renderer".

Bitmap renderer

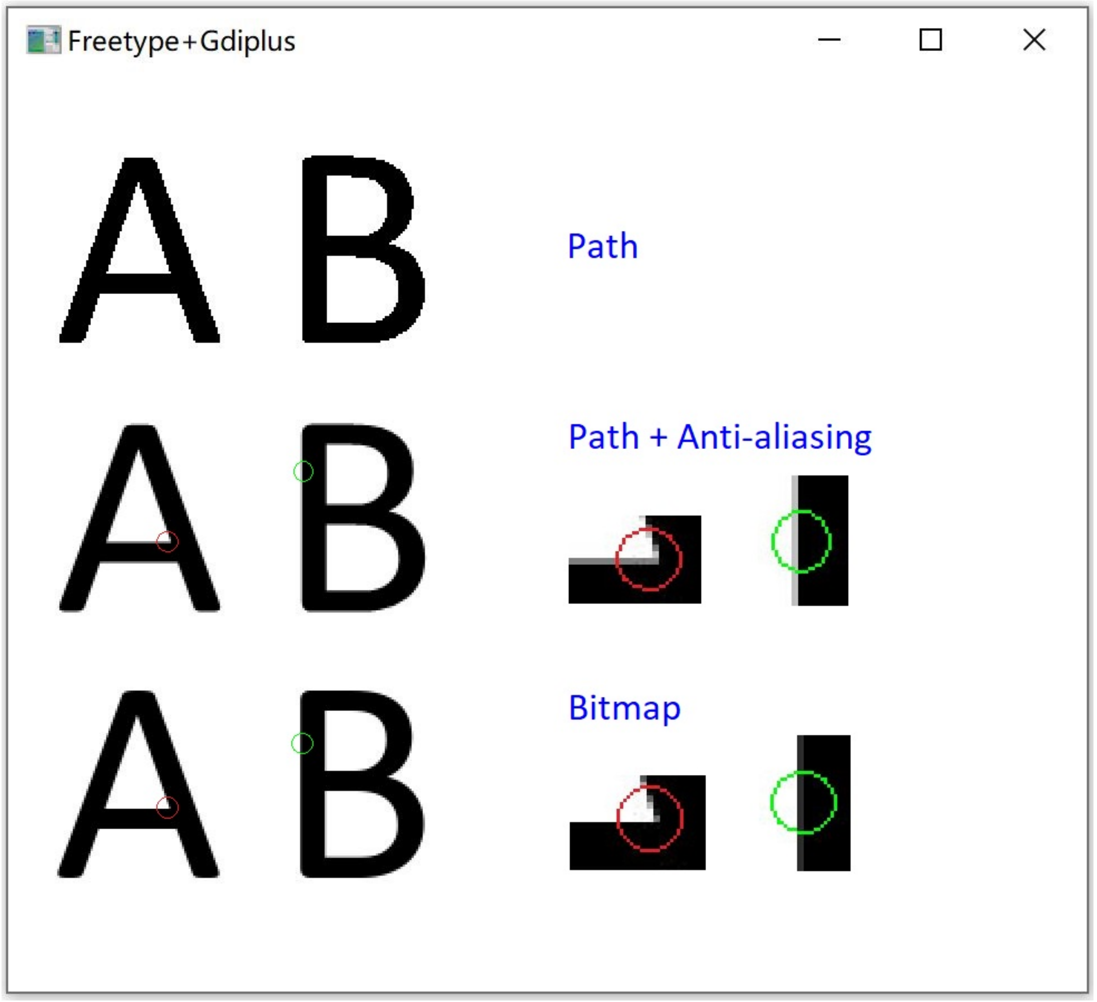
- A "toy" bitmap renderer:
    - Set font size (FT_Set_Char_Size/FT_Set_Pixel_Sizes).
    - Load the glyph (FT_Load_Char/FT_Load_Glyph).
    - Convert the given glyph image to a bitmap (FT_Render_Glyph). The default render mode renders an anti-aliased coverage bitmap with 256 gray levels (also called a **pixmap**).
    - Build a corresponding **bitmap** from the pixmap.
    - Draw the bitmap.

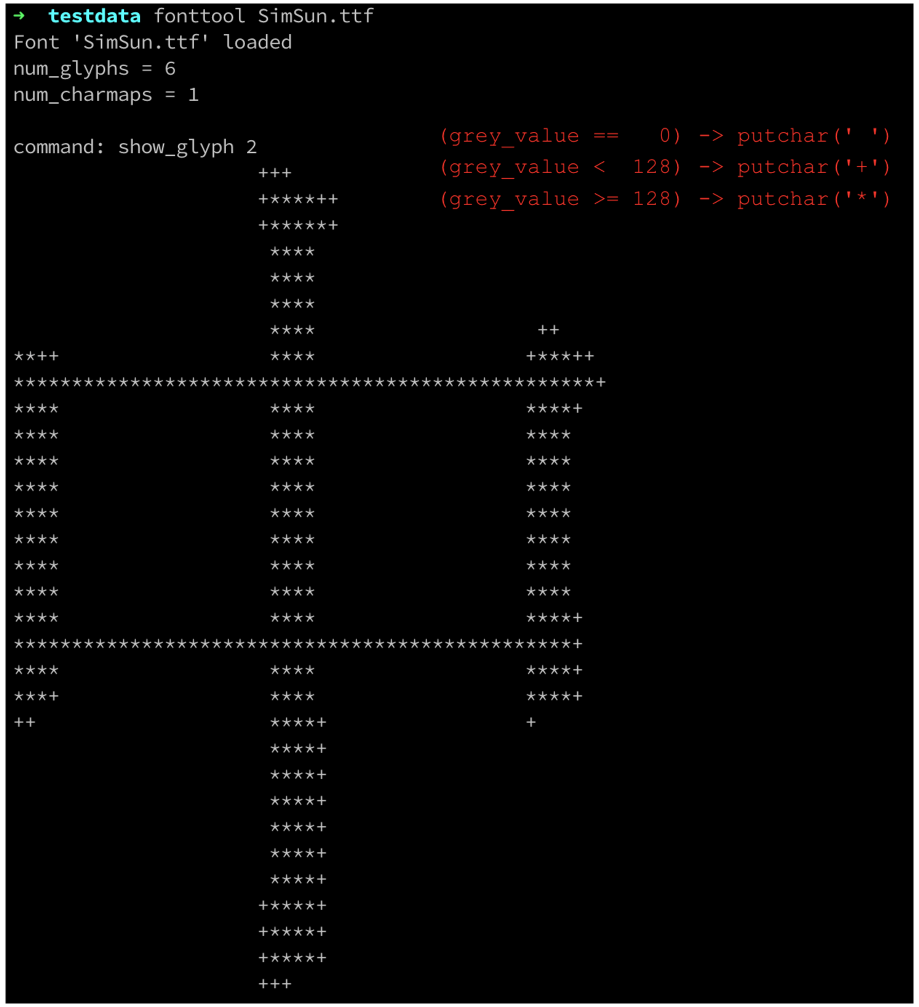
Bitmap renderer



```
static void DrawCharBitmap(Graphics& graphics, const Color& color, const PointF& pt, char c)
{
    int error = FT_Load_Char(ftFace, c, FT_LOAD_DEFAULT);                                 1
    if (error) { /* TODO */ }

    error = FT_Render_Glyph(ftFace->glyph, FT_RENDER_MODE_NORMAL);                        2
    if (error) { /* TODO */ }

    const FT_Bitmap* glyphBitmap = &ftFace->glyph->bitmap;
    const unsigned char* pixelBuffer = glyphBitmap->buffer;
    Bitmap bitmap(glyphBitmap->width, glyphBitmap->rows, &graphics);                      3
    for (unsigned int y = 0; y < glyphBitmap->rows; y++)
    {
        for (unsigned int x = 0; x < glyphBitmap->width; x++)
        {
            Color pixelColor(
                (BYTE)((int)(*pixelBuffer) * color.GetA() / 255),
                color.GetR(), color.GetG(), color.GetB());
            bitmap.SetPixel(x, y, pixelColor);
            pixelBuffer++;
        }
    }

    PointF origin(
        pt.X + ftFace->glyph->bitmap_left,
        pt.Y - ftFace->glyph->bitmap_top);
    graphics.DrawImage(&bitmap, origin);                                                  4
}
```

Bitmap renderer





```
➜  testdata fonttool SimSun.ttf
Font 'SimSun.ttf' loaded
num_glyphs = 6
num_charmaps = 1

command: show_glyph 2
                     +++
                     +****++
                     +*****+
                      ****
                      ****
                      ****
                      ****                   ++
**++                  ****                  +***++
**************************************************+
****                  ****                  ****+
****                  ****                  ****
****                  ****                  ****
****                  ****                  ****
****                  ****                  ****
****                  ****                  ****
****                  ****                  ****
****                  ****                  ****
****                  ****                  ****+
************************************************+
****                  ****                  ****+
***+                  ****                  ****+
++                    ****+                 +
                      ****+
                      ****+
                      ****+
                      ****+
                      ****+
                      ****+
                     +****+
                     +****+
                     +****+
                     +++
```

```
(grey_value ==   0) -> putchar(' ')
(grey_value <  128) -> putchar('+')
(grey_value >= 128) -> putchar('*')
```

Bitmap renderer

Scanline rendering - Wikipedia
How FreeType's rasterizer work
FreeType's new 'perfect' anti-aliasing renderer

Bitmap renderer

- In practice, we should store the generated pixmap somewhere (eg. a **LRU cache**) and query it whenever we want to render a glyph.
- The pixmap generated by FreeType is just large enough to contain the visible part of a glyph, which do not have the same size. so some **metrics** should be stored as well:
  - bitmap.width: the width of the bitmap.
  - bitmap.rows: the height of the bitmap.
  - bitmap_left: the horizontal offset (relative to the origin).
  - bitmap_top: the vertical offset (relative to the origin).

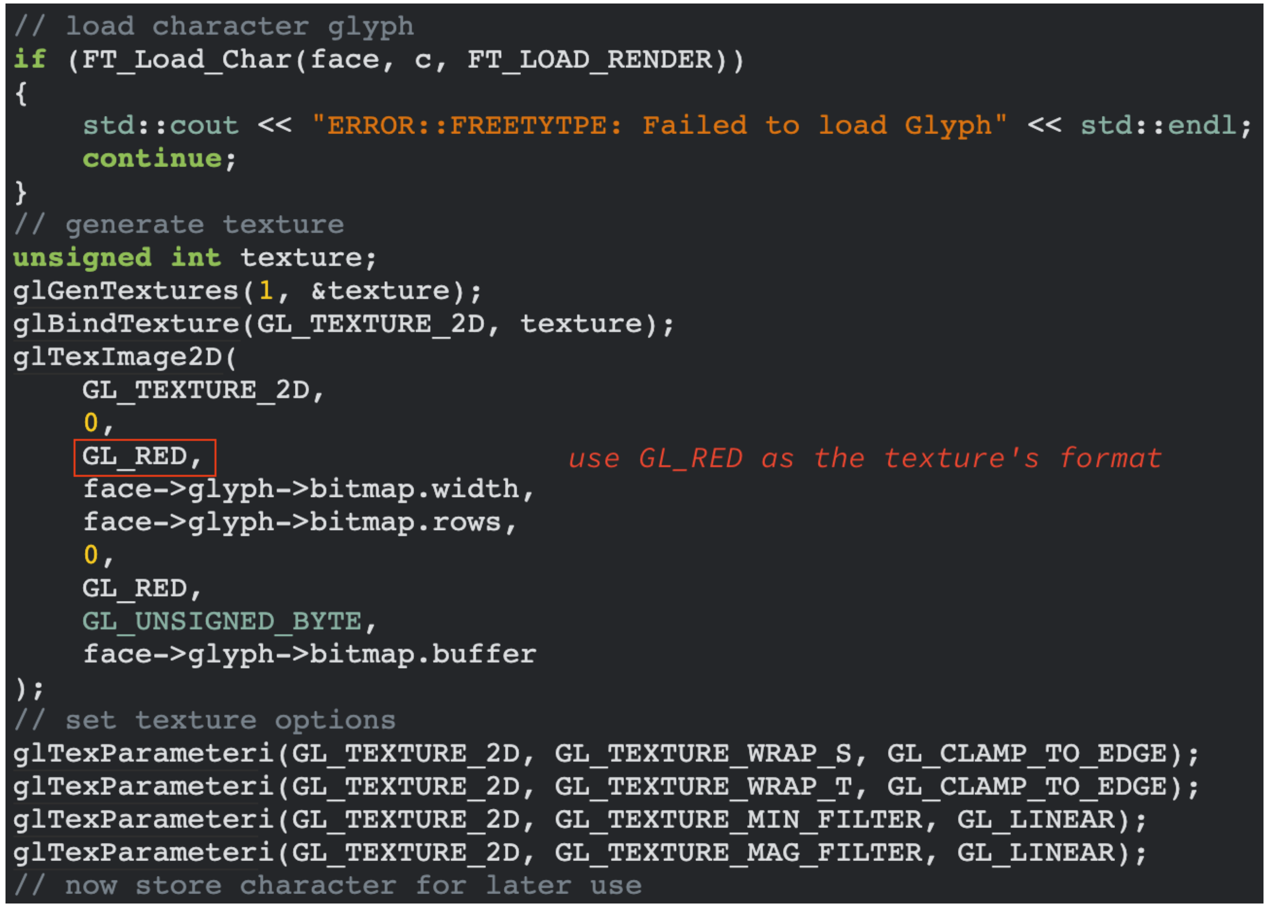
Bitmap renderer

- Modern computers have **GPU** which is very good at processing pixels and bilinear interpolation. We could use graphics API like OpenGL to implement a GPU based bitmap renderer.
- In OpenGL(DirectX/Vulkan), the most natural way to represent a glyph pixmap is using **texture**. Build a 2D texture from a pixmap, then do texture sampling in fragment shader.
- To draw a glyph quad, we can divide the rectangle into two triangles.

```
1 texture + 6 vertices (position and corresponding texture
coordinate) ==> 1 glyph quad
```

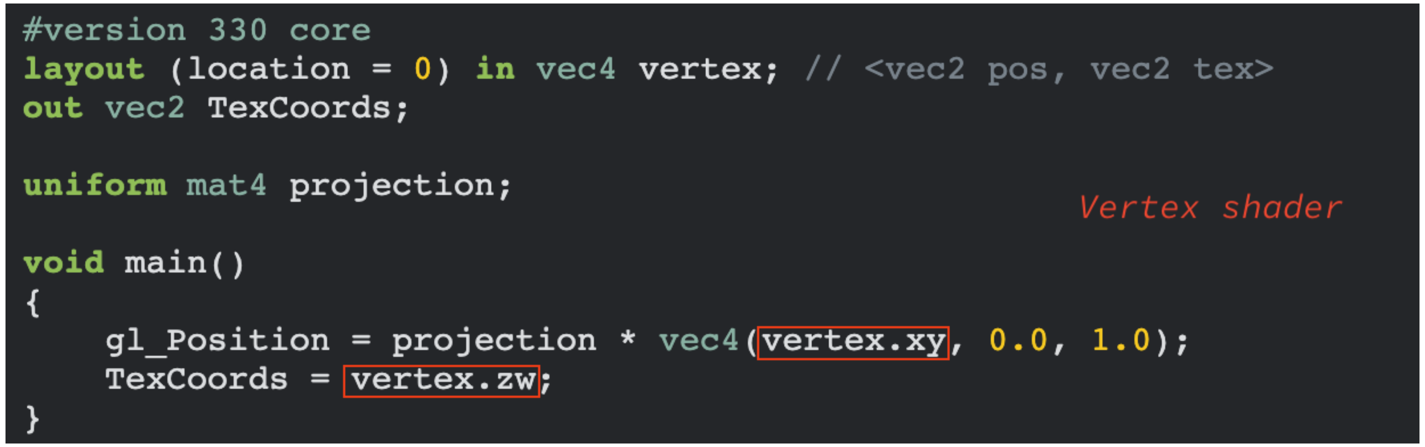
Bitmap renderer

Code snippets from OpenGL Text Rendering

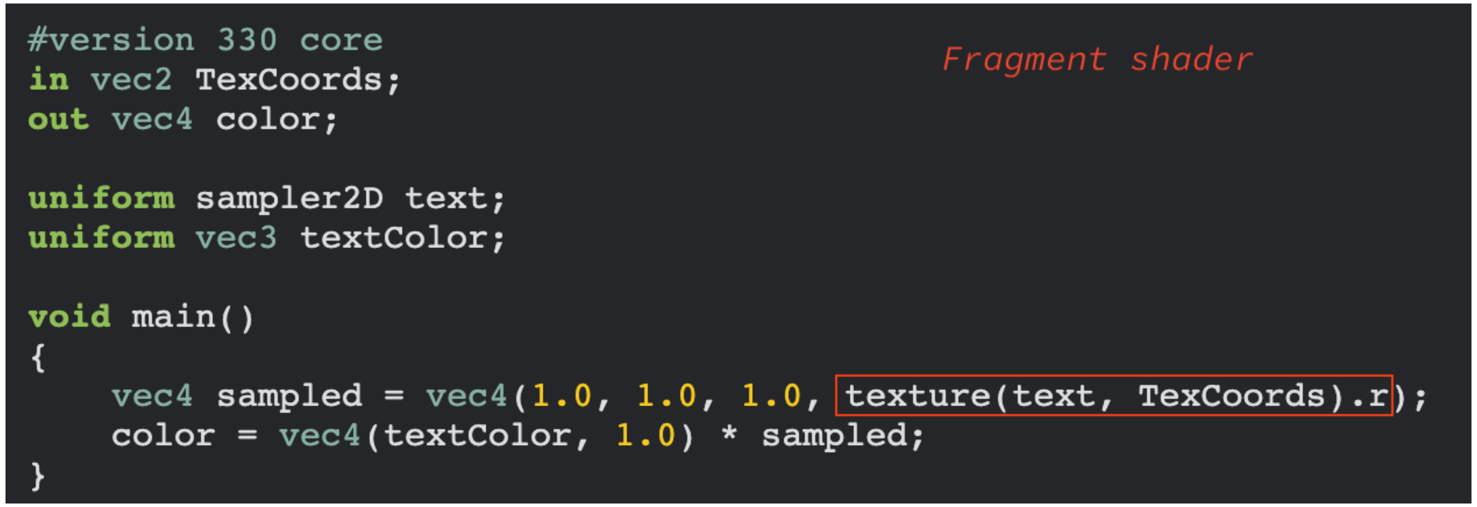


```
// load character glyph
if (FT_Load_Char(face, c, FT_LOAD_RENDER))
{
    std::cout << "ERROR::FREETYPE: Failed to load Glyph" << std::endl;
    continue;
}
// generate texture
unsigned int texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
glTexImage2D(
    GL_TEXTURE_2D,
    0,
    GL_RED,                          use GL_RED as the texture's format
    face->glyph->bitmap.width,
    face->glyph->bitmap.rows,
    0,
    GL_RED,
    GL_UNSIGNED_BYTE,
    face->glyph->bitmap.buffer
);
// set texture options
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, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// now store character for later use
```

Bitmap renderer

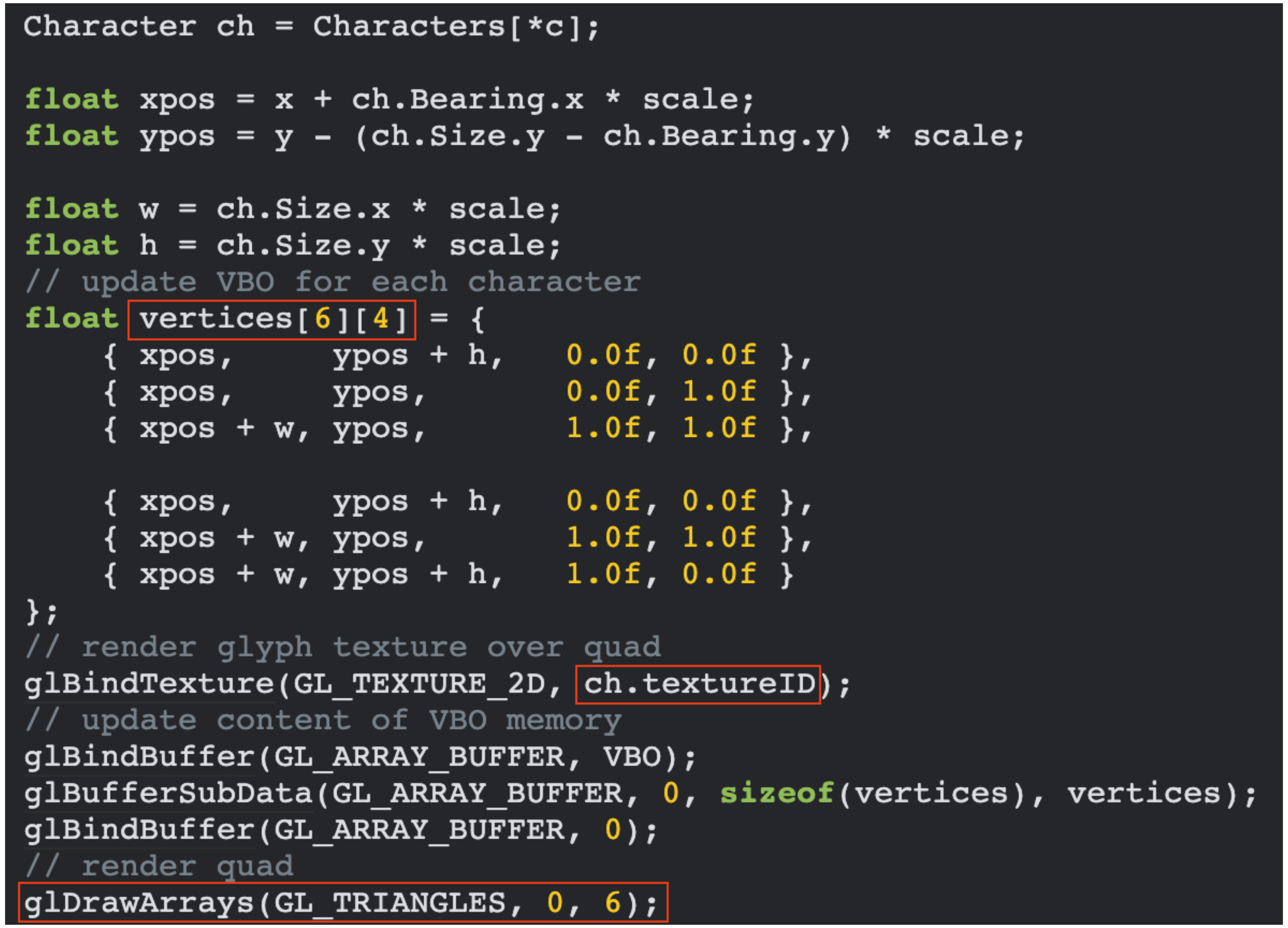


```
#version 330 core
layout (location = 0) in vec4 vertex; // <vec2 pos, vec2 tex>
out vec2 TexCoords;

uniform mat4 projection;

void main()
{
    gl_Position = projection * vec4(vertex.xy, 0.0, 1.0);
    TexCoords = vertex.zw;
}
```

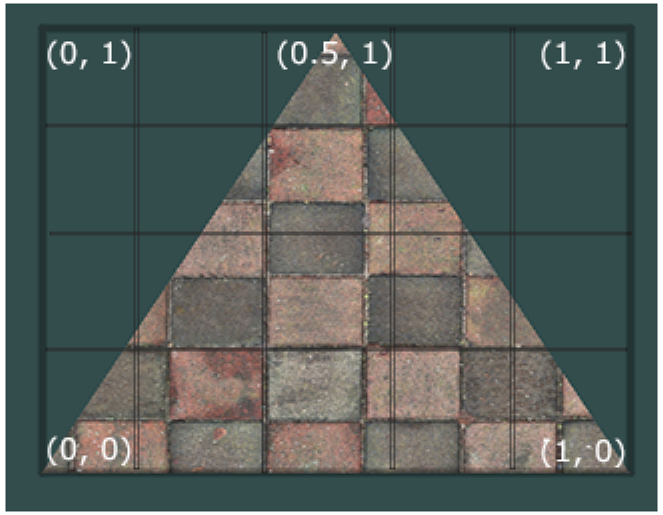


```
#version 330 core
in vec2 TexCoords;
out vec4 color;

uniform sampler2D text;
uniform vec3 textColor;

void main()
{
    vec4 sampled = vec4(1.0, 1.0, 1.0, texture(text, TexCoords).r);
    color = vec4(textColor, 1.0) * sampled;
}
```

Bitmap renderer



```
Character ch = Characters[*c];

float xpos = x + ch.Bearing.x * scale;
float ypos = y - (ch.Size.y - ch.Bearing.y) * scale;

float w = ch.Size.x * scale;
float h = ch.Size.y * scale;
// update VBO for each character
float vertices[6][4] = {
    { xpos,     ypos + h,   0.0f, 0.0f },
    { xpos,     ypos,       0.0f, 1.0f },
    { xpos + w, ypos,       1.0f, 1.0f },

    { xpos,     ypos + h,   0.0f, 0.0f },
    { xpos + w, ypos,       1.0f, 1.0f },
    { xpos + w, ypos + h,   1.0f, 0.0f }
};
// render glyph texture over quad
glBindTexture(GL_TEXTURE_2D, ch.textureID);
// update content of VBO memory
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);
glBindBuffer(GL_ARRAY_BUFFER, 0);
// render quad
glDrawArrays(GL_TRIANGLES, 0, 6);
```



Bitmap renderer

- One texture per glyph means lots of **texture switching**, which can hurt GPU performance. A better way is to have a single, large texture that contains lots of glyphs.
- A **texture atlas** is basically a big texture which contains many small images that are packed together.
- Glyphs may have different sizes, we need a **bin packing** algorithm.
- A Thousand Ways to Pack the Bin - A Practical Approach to Two-Dimensional Rectangle Bin Packing

Bitmap renderer

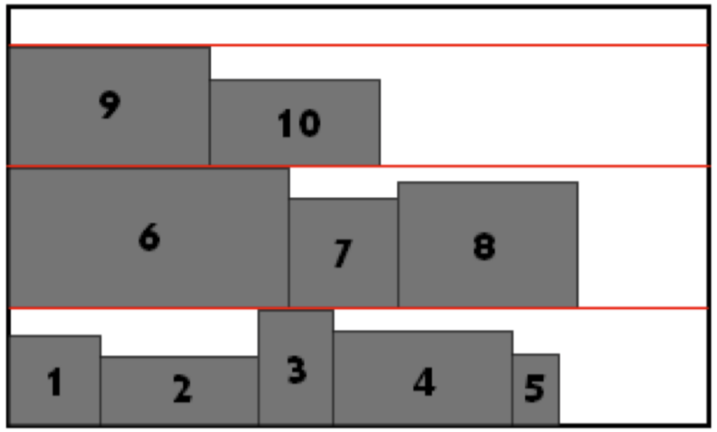


Figure 1: A sample packing produced by a Shelf algorithm.

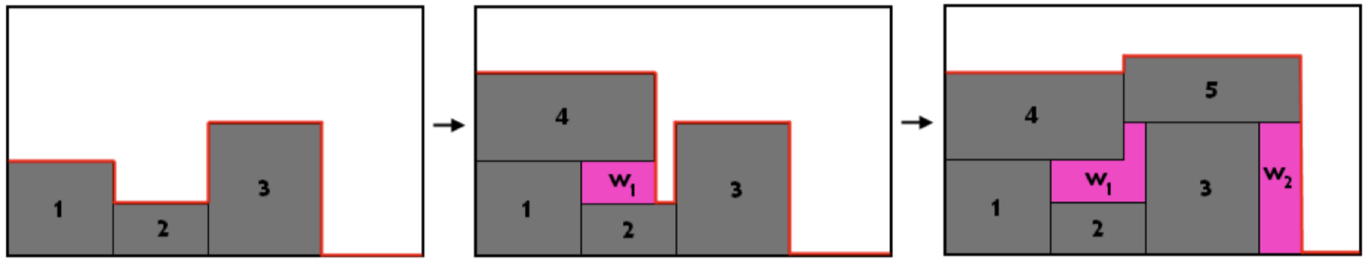
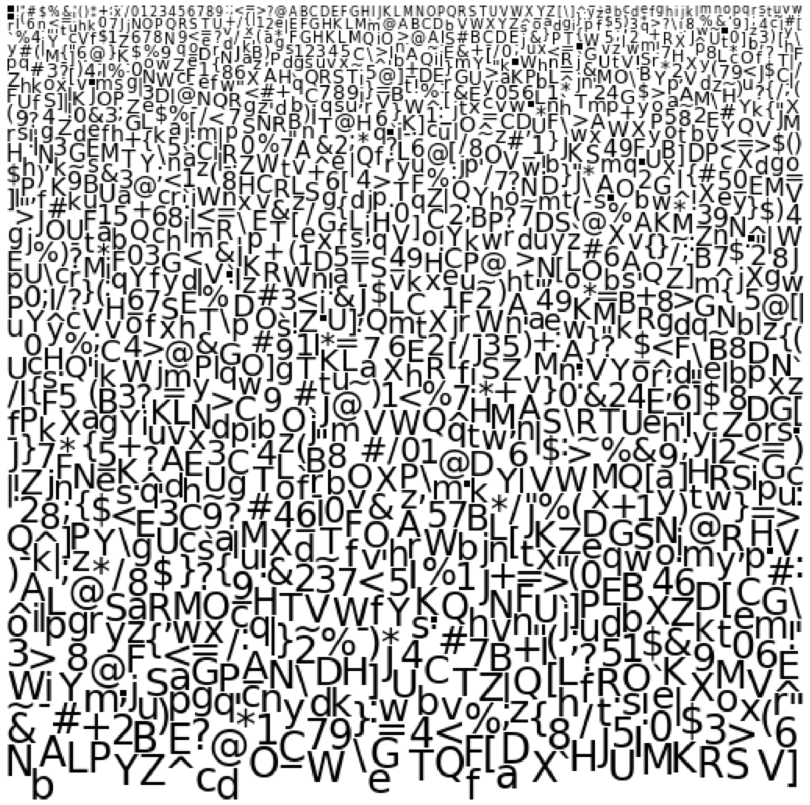


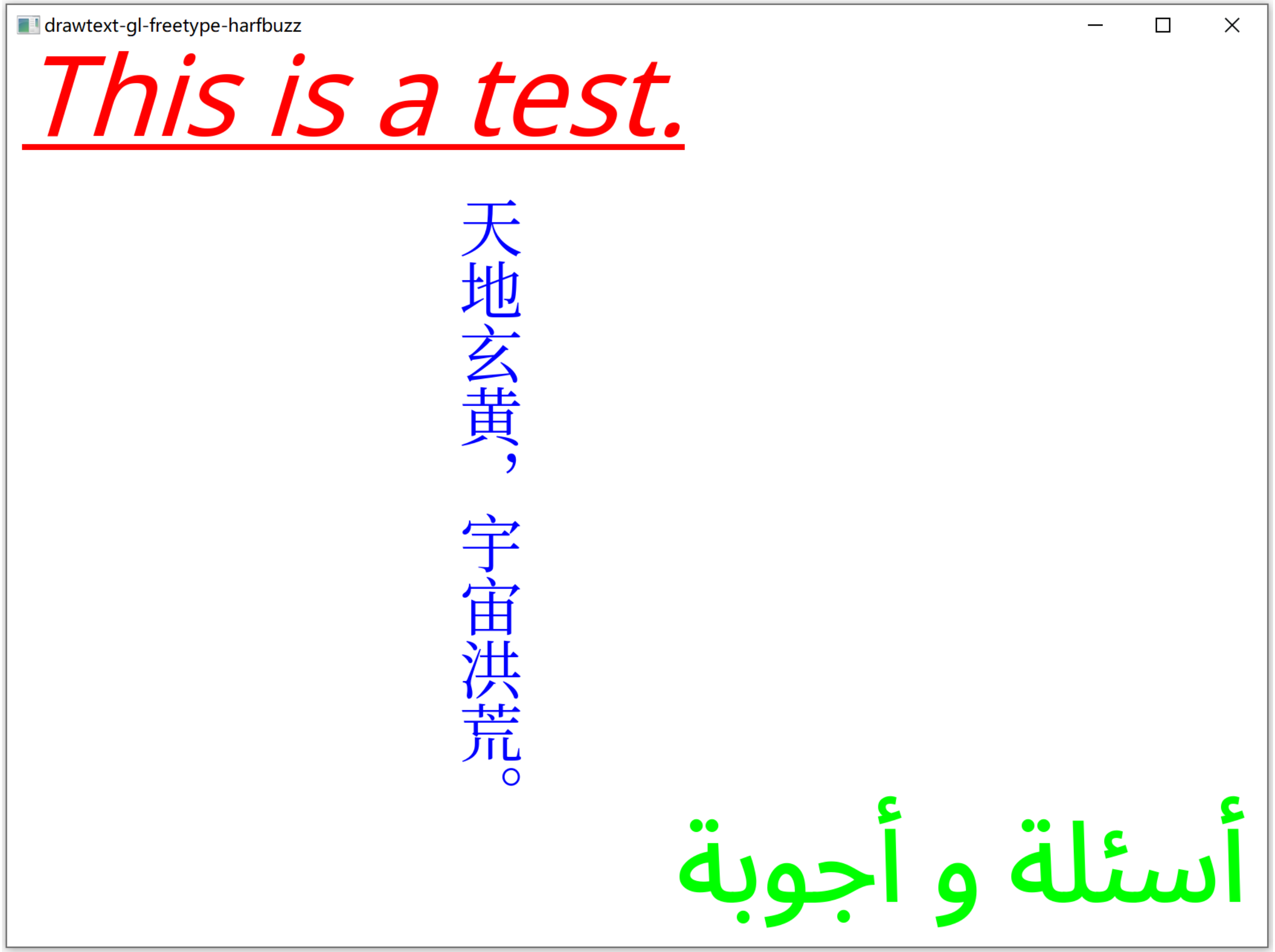
Figure 7: A sample packing produced by the SKYLINE-BL algorithm.

Bitmap renderer

Freetype-GL uses skyline bottom left.



Bitmap renderer



drawtext-gl-freetype-harfbuzz

This is a test.

天地玄黄，宇宙洪荒。

أسئلة و أجوبة

Bitmap renderer

- Pros
  - good display quality.
  - pretty fast.
- Cons
  - pre-rasterized glyphs are **font size dependent**.
  - relatively complex.

Hinting

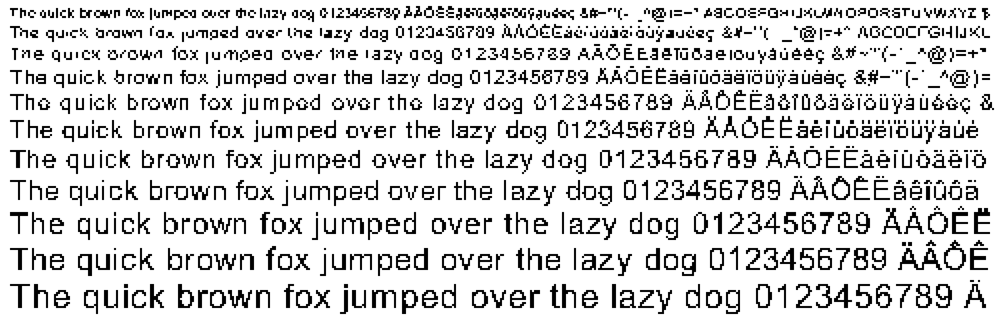
- Glyphs displayed on a **low-resolution** surface will often show numerous **unpleasant artifacts**:
    - stem widths and heights are not consistent, even in a single character image.
    - the top and bottom of certain characters do not seem to align with the top or bottom of others.
    - curves and diagonals are generally ugly.

Hinting

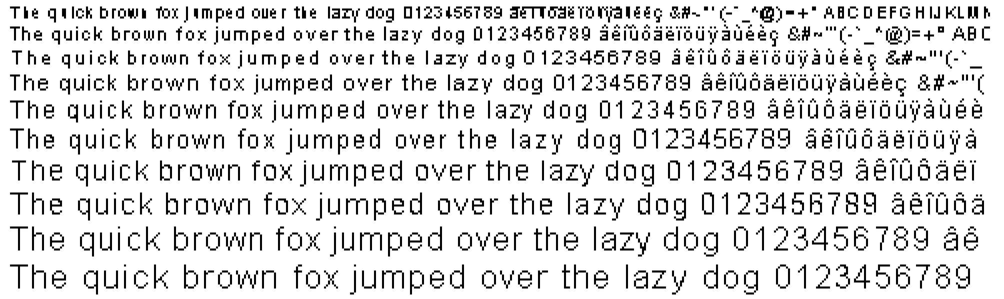
- Hinting (Grid-Fitting) is the general process of modifying glyph outlines in order to **align** some of their **important features** to the **pixel grid** in device space.
- When done correctly, the quality of the final glyph bitmaps is **massively improved**.

Hinting

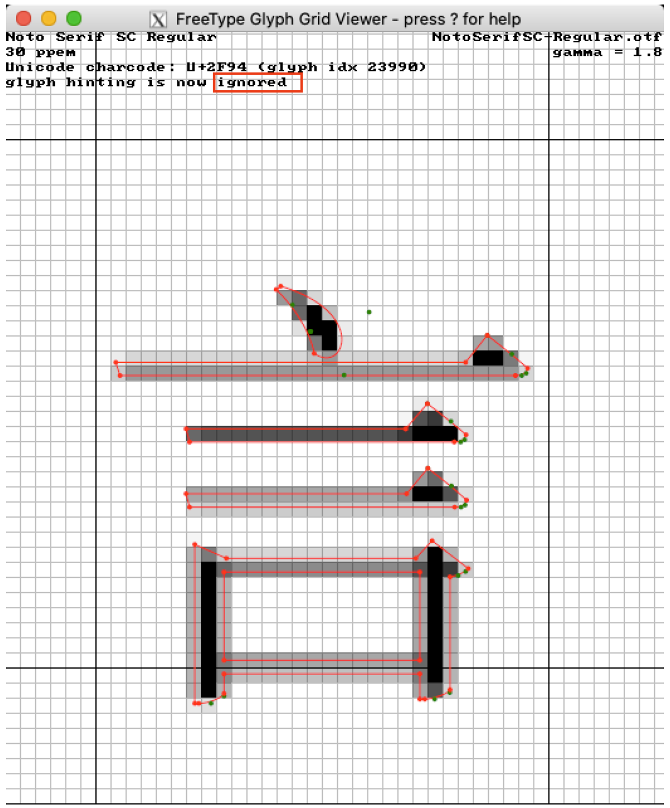
before:


The quick brown fox jumped over the lazy dog 0123456789 ÄÂÔÊËâêîûôäëïöüÿàùéèç &#~"'(-`_^@)=+* ABCDEFGHIJKLMNOPQRSTUVWXYZ $
The quick brown fox jumped over the lazy dog 0123456789 ÄÂÔÊËâêîûôäëïöüÿàùéèç &#~"'(-`_^@)=+* ABCDEFGHIJKL
The quick brown fox jumped over the lazy dog 0123456789 ÄÂÔÊËâêîûôäëïöüÿàùéèç &#~"'(-`_^@)=+*
The quick brown fox jumped over the lazy dog 0123456789 ÄÂÔÊËâêîûôäëïöüÿàùéèç &#~"'(-`_^@)=
The quick brown fox jumped over the lazy dog 0123456789 ÄÂÔÊËâêîûôäëïöüÿàùéèç &
The quick brown fox jumped over the lazy dog 0123456789 ÄÂÔÊËâêîûôäëïöüÿàùé
The quick brown fox jumped over the lazy dog 0123456789 ÄÂÔÊËâêîûôäëïö
The quick brown fox jumped over the lazy dog 0123456789 ÄÂÔÊËâêîûôä
The quick brown fox jumped over the lazy dog 0123456789 ÄÂÔÊË
The quick brown fox jumped over the lazy dog 0123456789 ÄÂÔÊ
The quick brown fox jumped over the lazy dog 0123456789 Ä

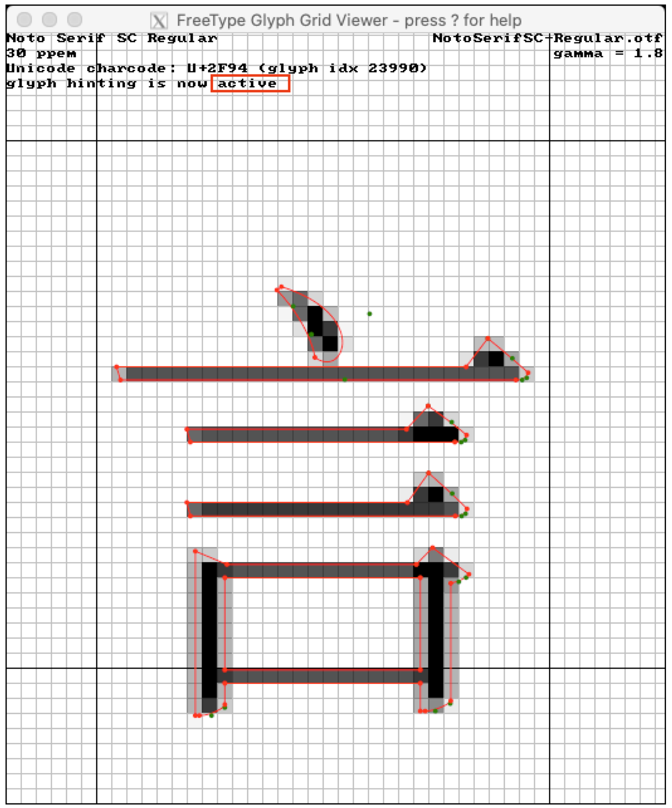
after:


The quick brown fox jumped over the lazy dog 0123456789 âêîûôäëïöüÿàùéèç &#~"'(-`_^@)=+* ABCDEFGHIJKLMN
The quick brown fox jumped over the lazy dog 0123456789 âêîûôäëïöüÿàùéèç &#~"'(-`_^@)=+* ABC
The quick brown fox jumped over the lazy dog 0123456789 âêîûôäëïöüÿàùéèç &#~"'(-`_
The quick brown fox jumped over the lazy dog 0123456789 âêîûôäëïöüÿàùéèç &#~"'(
The quick brown fox jumped over the lazy dog 0123456789 âêîûôäëïöüÿàùéè
The quick brown fox jumped over the lazy dog 0123456789 âêîûôäëïöüÿà
The quick brown fox jumped over the lazy dog 0123456789 âêîûôäëï
The quick brown fox jumped over the lazy dog 0123456789 âêîûôä
The quick brown fox jumped over the lazy dog 0123456789 âê
The quick brown fox jumped over the lazy dog 0123456789

Hinting


FreeType Glyph Grid Viewer - press ? for help

```
Noto Serif SC Regular                NotoSerifSC-Regular.otf
30 ppem                                          gamma = 1.8
Unicode charcode: U+2F94 (glyph idx 23990)
glyph hinting is now ignored
```


FreeType Glyph Grid Viewer - press ? for help

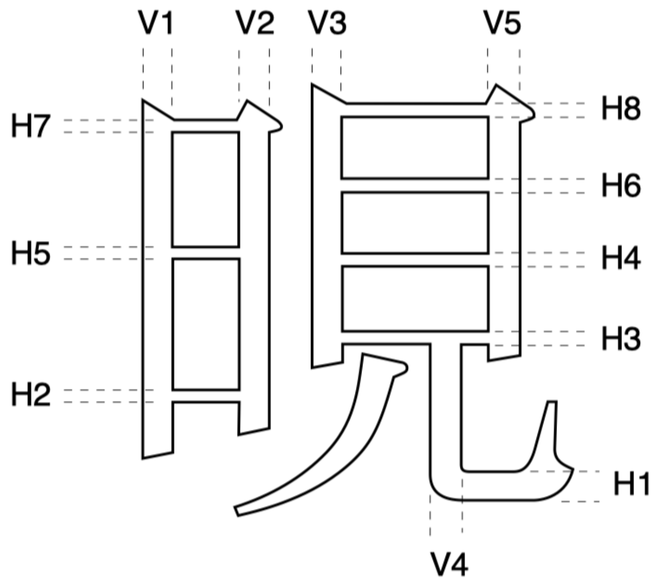
```
Noto Serif SC Regular                NotoSerifSC-Regular.otf
30 ppem                                          gamma = 1.8
Unicode charcode: U+2F94 (glyph idx 23990)
glyph hinting is now active
```

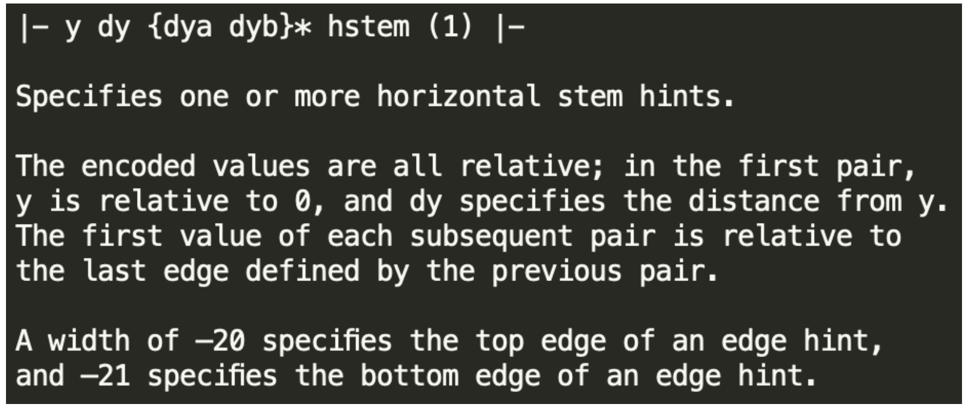
Hinting

- Grid-fitting can be decomposed in two important phases:
  - detecting important glyph and font features and **produce** control data, called hints.
  - **applying** the alignment operations described in the hints.
- Traditionally, the first pass is performed when the font file is created. For example, the TrueType format associates to each glyph outline a **optional** bytecoded program which handle hinting.
- FreeType **automatic hinting module** include both a feature-detection and alignment control pass.

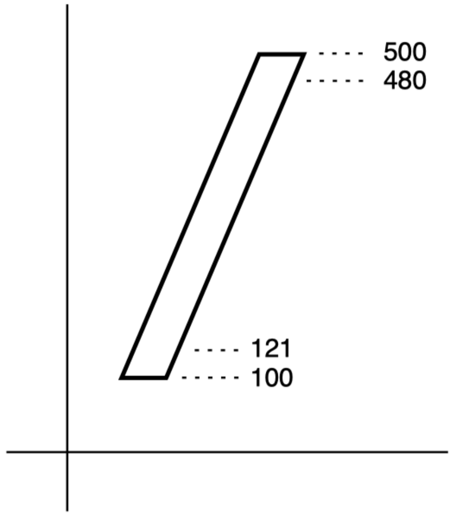
Hinting

[CFF2 CharString Hinting operators]





```
|- y dy {dya dyb}* hstem (1) |-

Specifies one or more horizontal stem hints.

The encoded values are all relative; in the first pair,
y is relative to 0, and dy specifies the distance from y.
The first value of each subsequent pair is relative to
the last edge defined by the previous pair.

A width of –20 specifies the top edge of an edge hint,
and –21 specifies the bottom edge of an edge hint.
```



121 –21 400 –20 **hstem**

Hinting

References:

Font hinting - Wikipedia

TrueType hinting

The Type 2 Charstring Format

FreeType auto-hinter

The character "热" looks bad in 15px 96dpi with fontconfig 2.10

SDF

- A signed distance field, or SDF for short, is the result of a signed distance transformation applied to a subset of N-dimensional space. It **maps** each point P of the space to a **scalar signed distance** value. A signed distance is defined as follows: If the point P **belongs** to the subset, it is the minimum distance to any point outside the subset. If it does **not belong** to the subset, it is **minus** the minimum distance to any point of the subset.

SDF

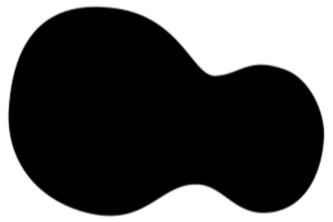


Figure 1.1: A two-dimensional shape.

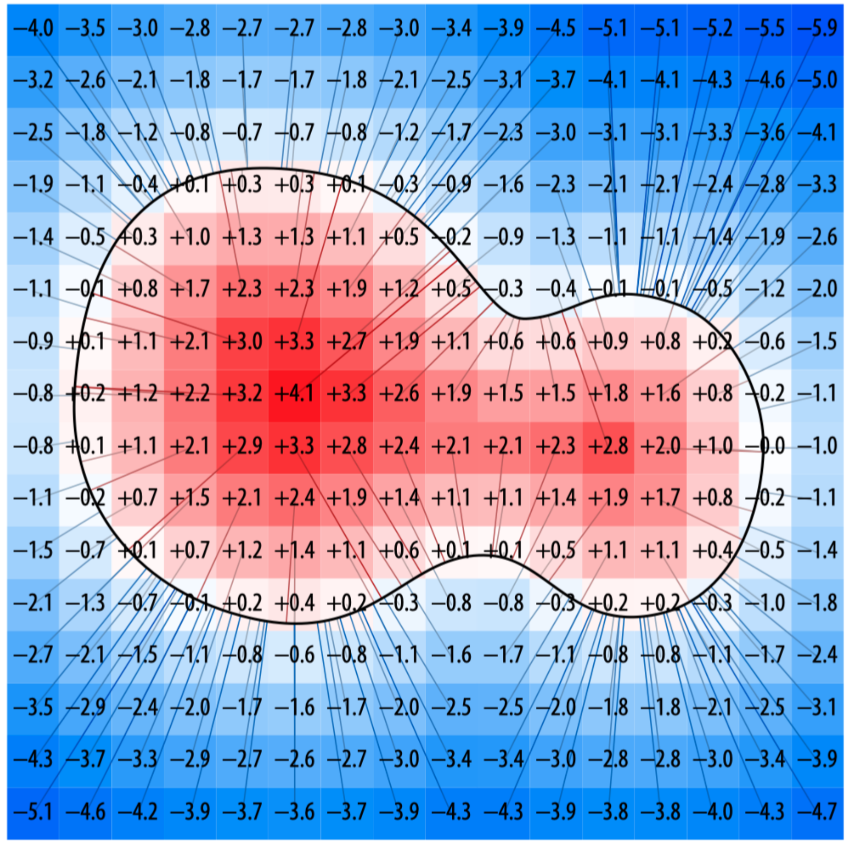
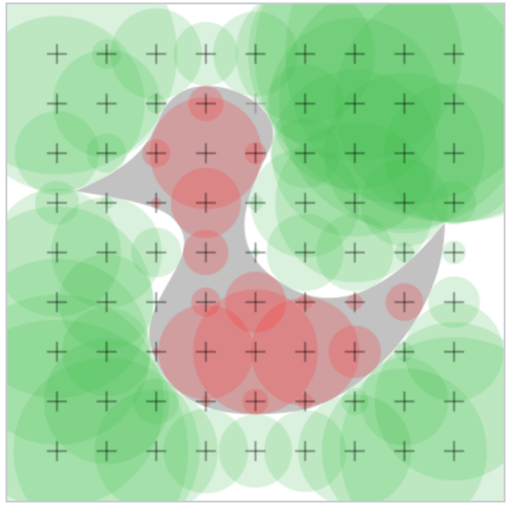


Figure 1.2: The shape's signed distance field.

SDF

- Imagine plotting a circle for each point of the grid, and using the absolute value of the signed distance as its radius.



- The collective area of the negative circles lies strictly outside the shape and the area of the positive ones inside. Only the space that does not coincide with any circle is **uncertain**.

SDF

- Assuming the shape is relatively **smooth**, the uncertain area can be reconstructed by taking the shortest or smoothest route.
- The distance values change very smoothly and predictably. using simple interpolation, the signed distance field grid can be sampled at **a much higher resolution**, and still provide a **good approximation** of the actual signed distances throughout the plane.
- One SDF for **all** font sizes.

SDF

- Clearly, our SDF only use a finite number of discrete points, and therefore **not exact**.
- To be able to render more complex shapes such as decorative fonts we need to use a higher SDF resolution.

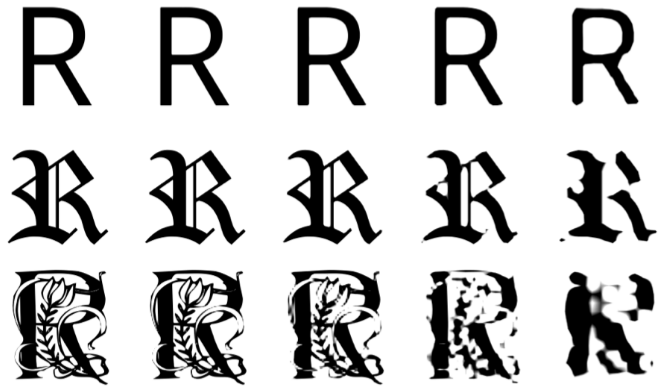
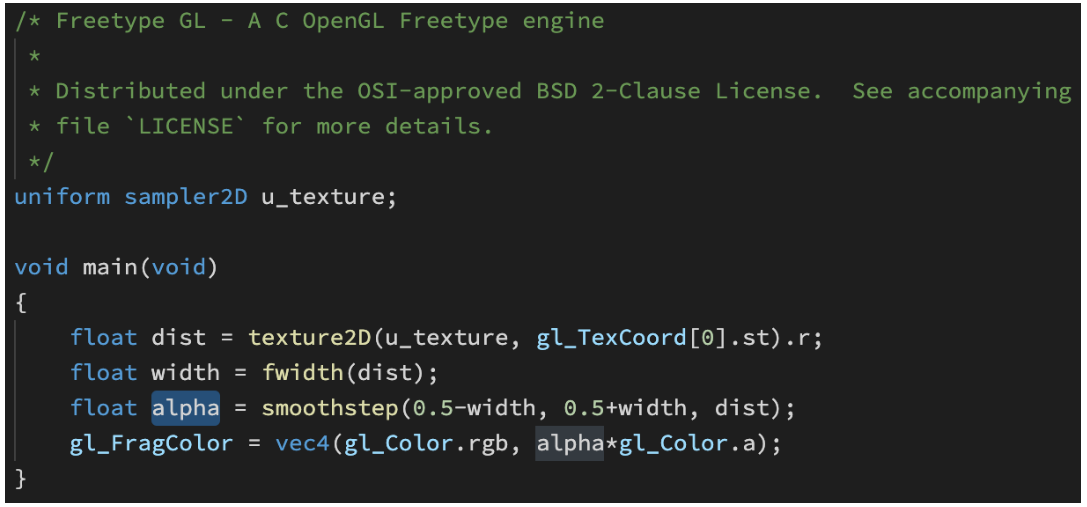


Figure 3.5: Renders using SDFs of different resolution. From left to right (256×256), (128×128), (64×64), (32×32), (16×16).

SDF

- By convention, we map the signed distance into the range 0..1, with **0** representing the maximum possible negative distance and **1.0** representing the maximum possible positive distance. hence, **0.5** is generally used for the **alpha threshold value**.
- A simple SDF renderer (OpenGL fragment shader):



```
/* Freetype GL - A C OpenGL Freetype engine
 *
 * Distributed under the OSI-approved BSD 2-Clause License.  See accompanying
 * file `LICENSE` for more details.
 */
uniform sampler2D u_texture;

void main(void)
{
    float dist = texture2D(u_texture, gl_TexCoord[0].st).r;
    float width = fwidth(dist);
    float alpha = smoothstep(0.5-width, 0.5+width, dist);
    gl_FragColor = vec4(gl_Color.rgb, alpha*gl_Color.a);
}
```

SDF

- Glyph shapes often have **sharp** corners, which cause irregularities in the distance field.

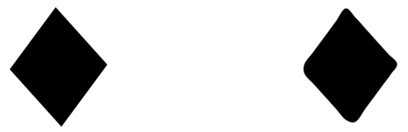
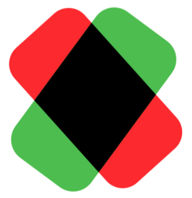


Figure 1.6: A shape with sharp corners (left) and its reconstruction from a low resolution distance field (right).

- A possible solution is to divide it into two smooth shapes, and create two separate distance fields. When reconstructing the image, one can first reconstruct the two auxiliary subshapes, and afterwards fill only those pixels that **belong in both**.

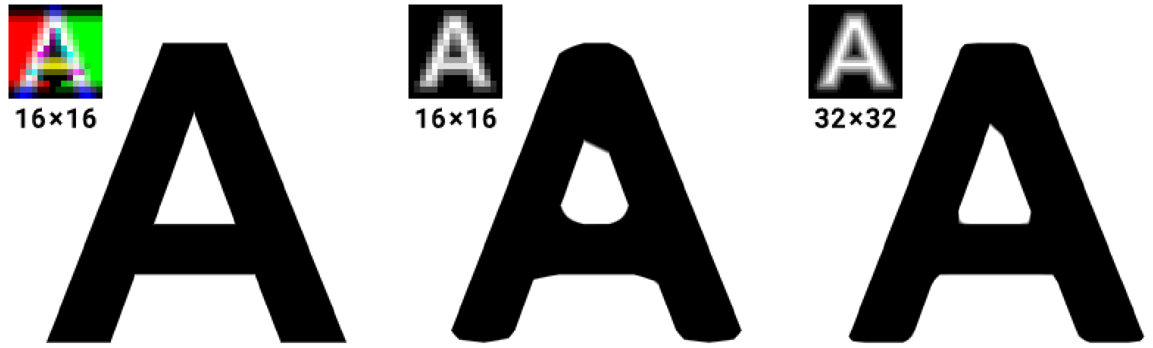


SDF

- The distance fields are usually stored as monochrome images. However, image files have the ability to hold **3 or more color channels**, making it natural to encode the two separate distance fields as one image, where each channel holds one of them.
- Viktor Chlumsky proposes the **Multi-channel Distance Fields** approach in his master's thesis.

SDF

- By utilizing all three color channels, MSDF have the ability to reproduce sharp corners almost perfectly.



- The performance impact of using MSDF is relatively small.

SDF

- There are a few minor drawbacks:
  - No support for font hinting.
  - Computing the (M)SDF is not trivially cheap.

SDF

References:

Valve paper

Drawing Text with Signed Distance Fields in Mapbox GL

Multi-channel signed distance field generator

FreeType Merges New "SDF" Renderer

Loop Blinn Curve Rendering

C. Loop, J. Blinn, "Resolution Independent Curve Rendering using Programmable Graphics Hardware"

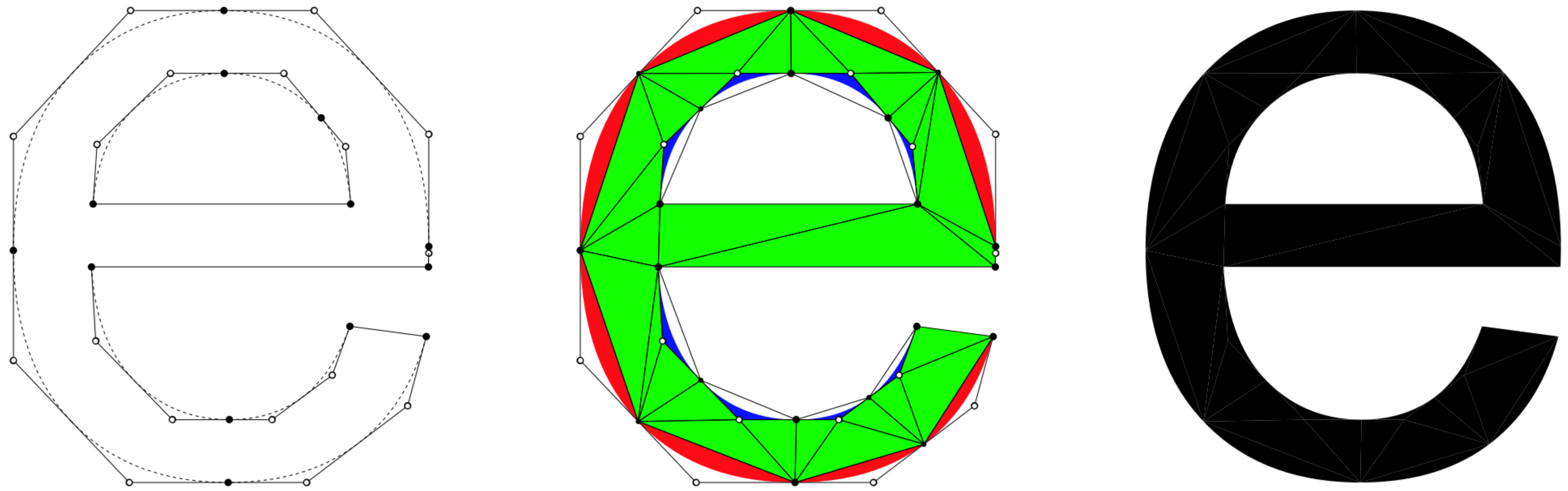


Figure 3: A two contour TrueType font outline on the left; filled dots represent on-curve points, hollow dots represent off-curve points. The outline is triangulated together with implied *on-curve* points as shown in the middle. The green triangles are interior to the shape and are entirely filled. The red (convex) and blue (concave) curves within triangles are rendered using our pixel shader program. The resulting shape on the right, can be arbitrarily transformed projectively and remains resolution independent.

Loop Blinn Curve Rendering

- The triangulation step is quite complicated and for complicated glyphs the triangle count could reach large numbers for each glyph.
- At small sizes when several of the outline curve segments intersect a pixel, the pixel shader will only use one of them to determine the color.
- Anti-aliasing requires additional triangles to be added to the outside of the glyph mesh or the use of super sampling.

Slug Algorithm

E. Lengyel, "GPU-Centered Font Rendering Directly from Glyph Outlines"

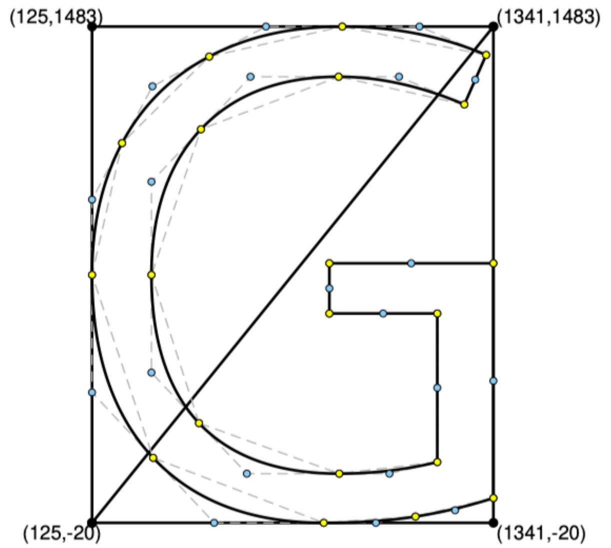


Figure 4.1: Quad with texture coordinates set to the glyphs bounding box.

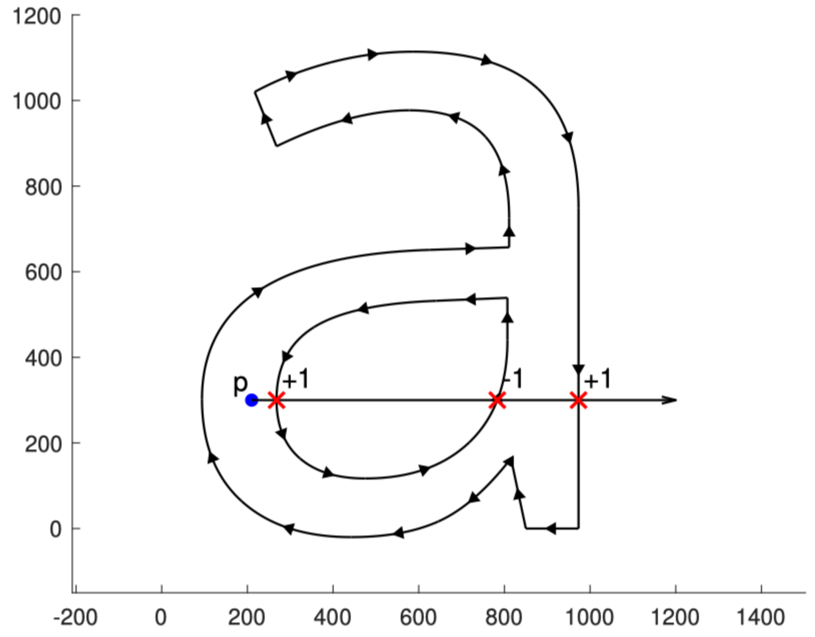


Figure 4.2: A ray is fired in positive x-direction. For every intersection the winding number is adjusted.

Slug Algorithm

- For a closed curve in a 2D plane, the **winding number** for a point p is the number of times the curve loops clockwise around the point.
- In general, a glyph shape can be **sampled** at any point by determining the winding number.
- One way to compute the winding number of a point p is to shoot a ray in any direction, originating from p.
- Then a non-zero winding number indicates that the point is inside the glyph shape, otherwise it is outside.
- Many times **slower** than the pixmap texture based methods.

Thanks