Technical - Additional Topics

Font Optimization

Font – Optimization

- When including FreeType font library into your engines, while following online tutorials, you might be using a slow coding method
- Problem!
  - Mainly most tutorials are rendering one character at a time

Font – Optimization

- Initial phase: bind 1 texture per character (glyph)
- See sample snippet next


```
for (GLubyte c = 0; c < 128; c++)
{
    // Load character glyph
    if (FT_Load_Char(face, c, FT_LOAD_RENDER)) //tell FreeType to create an 8-bit grayscale bitmap image for us
    {
        std::cout << "Cannot load Glyph" << std::endl;
        continue;
    }

    // Generate texture
    GLuint texture;
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);

    glTexImage2D(
        GL_TEXTURE_2D,
        0,
        GL_ALPHA,
        face->glyph->bitmap.width,
        face->glyph->bitmap.rows,
        0,
        GL_ALPHA,
        GL_UNSIGNED_BYTE,
        face->glyph->bitmap.buffer
    );

    // Now store character for later use
    CharacterGlyph character;
    character.textureID = texture;
    character.size = Vector2D((float)face->glyph->bitmap.width, (float)face->glyph->bitmap.rows);
    character.bearing = Vector2D((float)face->glyph->bitmap_left, (float)face->glyph->bitmap_top);
    character.advance = (GLuint)face->glyph->advance.x;

    m_charactersGlyph.insert(std::pair<GLchar, CharacterGlyph>(c, character));

    glBindTexture(GL_TEXTURE_2D, 0);
}
```

Font – Optimization

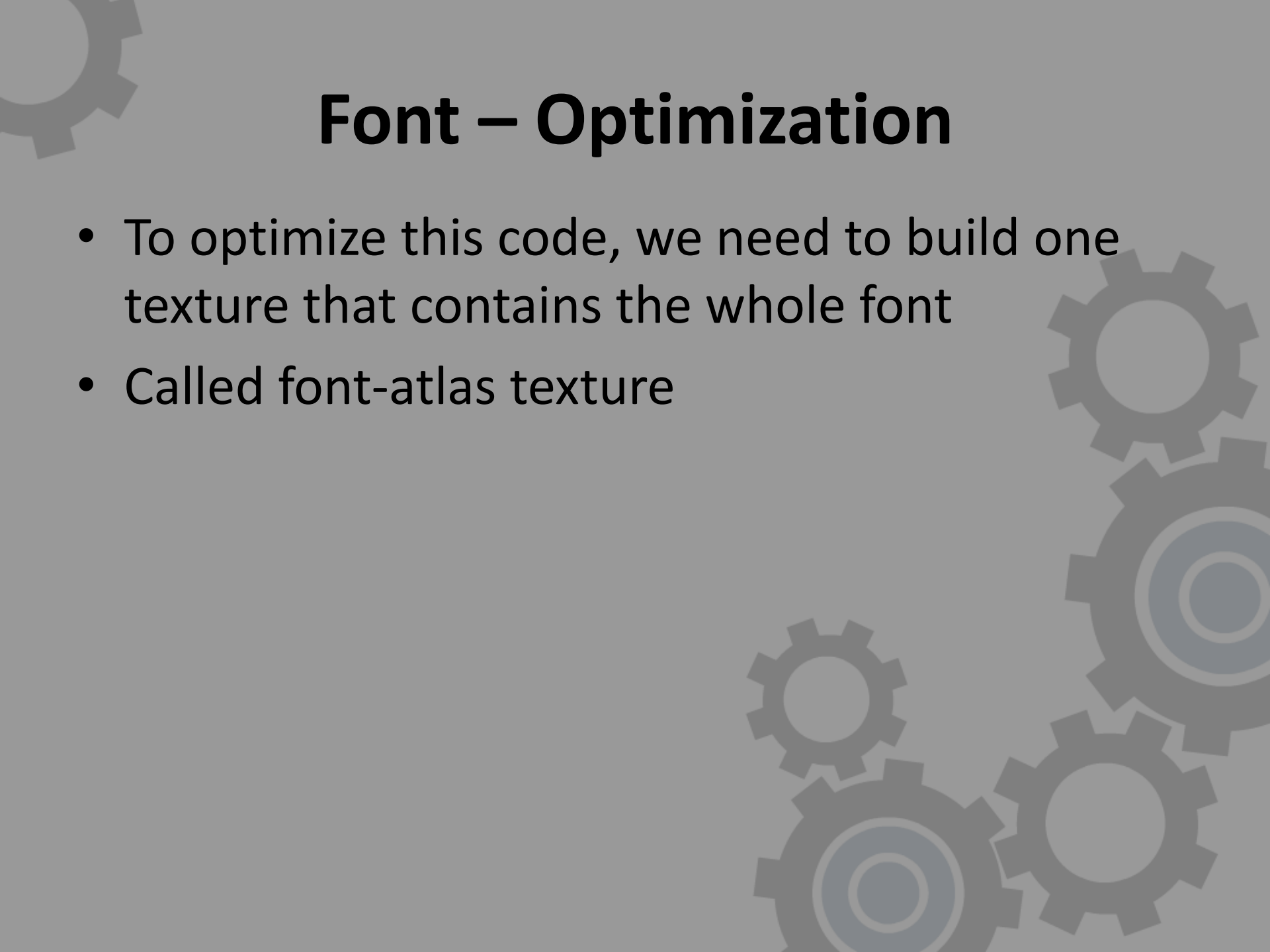
- Render phase: renders one character at a time
- See sample snippet next

Font – Optimization

```
// Iterate through all characters
std::string::const_iterator c;
for (c = text.begin(); c != text.end(); c++)
{
    CharacterGlyph ch = m_charactersGlyph[*c];

    GLfloat xpos = advancePos.m_x + ch.bearing.m_x * scale.m_x;
    GLfloat ypos = advancePos.m_y - (ch.size.m_y - ch.bearing.m_y) * scale.m_y;

    GLfloat w = ch.size.m_x * scale.m_x;
    GLfloat h = ch.size.m_y * scale.m_y;
    // Update VBO for each character
    GLfloat vertices[6][4] = {
        { xpos,     ypos + h,   0.0, 0.0 },
        { xpos,     ypos,       0.0, 1.0 },
        { xpos + w, ypos,       1.0, 1.0 },

        { xpos,     ypos + h,   0.0, 0.0 },
        { xpos + w, ypos,       1.0, 1.0 },
        { xpos + w, ypos + h,   1.0, 0.0 }
    };
    // Render glyph texture over quad
    glBindTexture(GL_TEXTURE_2D, ch.textureID);
```

Font – Optimization

- To optimize this code, we need to build one texture that contains the whole font
- Called font-atlas texture



Font – Optimization

- Initial phase (optimized):
  - Bind one texture only
  - 2 loops
    - First loop: to compute the total width & height of the atlas
    - Second loop to build the individual glyphs buffers and combine them all into one
      - Use “glTexSubImage2D”
- See sample snippet next

Font – Optimization

```
//To build one texture for all glyphs

//Generate texture
GLuint texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);

int textureWidth = 0, textureHeight = 0;

//first loop to compute textureWidth and textureHeight
for (GLubyte c = 0; c < 128; c++)
{
    // Load character glyph
    if (FT_Load_Char(face, c, FT_LOAD_RENDER)) //tell FreeType to create an 8-bit grayscale bitmap image for us
    {
        std::cout << "Cannot load Glyph" << std::endl;
        continue;
    }

    //in this sample we are building one row texture (all character's buffers are concatenated in one row)
    textureWidth += face->glyph->bitmap.width;
    if (textureHeight < (int)face->glyph->bitmap.rows)//getting the max possible height
        textureHeight = (int)face->glyph->bitmap.rows;

    // Now store character for later use
    CharacterGlyph character;
    character.textureID = texture;//this can be removed - no need to save the same texture id for all glyphs
    character.size = Vector2D((float)face->glyph->bitmap.width, (float)face->glyph->bitmap.rows);
    character.bearing = Vector2D((float)face->glyph->bitmap_left, (float)face->glyph->bitmap_top);
    character.advance = (GLuint)face->glyph->advance.x;

    m_charactersGlyph.insert(std::pair<GLchar, CharacterGlyph>(c, character));
}
```

Font – Optimization

```
glTexImage2D(
    GL_TEXTURE_2D,
    0,
    GL_ALPHA,
    textureWidth,
    textureHeight,
    0,
    GL_ALPHA,
    GL_UNSIGNED_BYTE,
    NULL //can be NULL
);
```

Font – Optimization

```
// second loop to build one atlas-texture map for the whole font
for (GLubyte c = 0; c < 128; c++)
{
    if (FT_Load_Char(face, c, FT_LOAD_RENDER))
    {
        std::cout << "Cannot load Glyph" << std::endl;
        continue;
    }

    //startUV
    float uS = (float)(UJumps) / (float)(textureWidth);
    float vS = (float)(VJumps) / (float)textureHeight;

    //endUV
    float uE = (float)(UJumps + (int)face->glyph->bitmap.width);
    uE = uE / (float)(textureWidth);
    float vE = (float)(VJumps + face->glyph->bitmap.rows);
    vE = vE / (float)(textureHeight);

    //CharacterGlyph structure has now 2 additional members: (Vector2D startUV, Vector2D endUV)
    m_charactersGlyph[c].startUV = Vector2D(uS, vS);
    m_charactersGlyph[c].endUV = Vector2D(uE, vE);

    if (face->glyph->bitmap.width != 0)
    {
        glTexSubImage2D(
            GL_TEXTURE_2D,
            0,
            UJumps,
            VJumps,
            face->glyph->bitmap.width,
            face->glyph->bitmap.rows,
            GL_ALPHA,
            GL_UNSIGNED_BYTE,
            face->glyph->bitmap.buffer);
    }

    //offset along u
    UJumps += face->glyph->bitmap.width;
}
```

Font – Optimization

- Render phase (optimized):
  - Renders all characters of a string in one texture bind call
    - glBindTexture(GL_TEXTURE_2D, commonTextureId);
  - The number of elements in "glBufferData" now is multiplied by the number of characters in the string to render
- See sample snippet next

Font – Optimization

```
glBindVertexArray(m_VAO);
glBindBuffer(GL_ARRAY_BUFFER, m_VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * numberOfVerticesPerGlyph * numberOfElementsIn_FontVertex * numberOfCharacters, NULL, GL_DYNAMIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, numberOfElementsIn_FontVertex, GL_FLOAT, GL_FALSE, numberOfElementsIn_FontVertex * sizeof(GL_FLOAT), 0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

Font – Optimization Demo

Font
Signed Distance Field

Font – Signed Distance Field

- Simple and efficient method that allows improving the rendering of glyphs with curved and linear elements.
- Distance field map is generated from a high-resolution image and saved at lower resolution texture.

Font – Signed Distance Field

- In the simplest case, the texture can be rendered by using alpha testing and alpha-thresholding.

Font – Signed Distance Field

- Original Problem
  - Lower resolution textures, when scaled up and magnified, results in undesirable blurry effects.
  - Having higher resolution textures, requires tons of memory and processing.
  - Line/curve art images requires very high resolution.

Font – Signed Distance Field

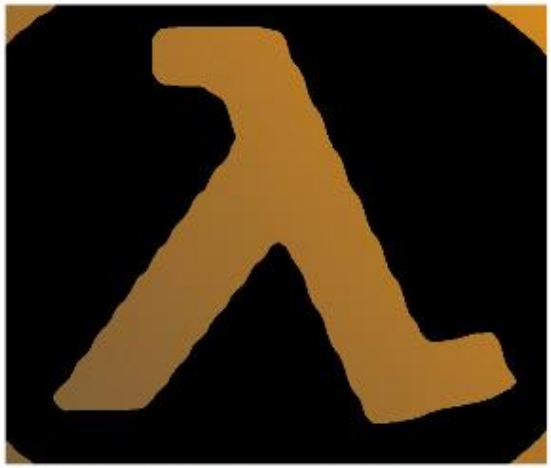
- Main solution
  - Generating alpha-tested texture maps
    - The alpha value output from a pixel shader is thresholded.
    - Widely used in games to provide sharp edges
    - Unfortunately, because of how the images are saved, the bilinear interpolation of the sub-texel still have unpleasant artifacts for non-axis aligned edges.

Font – Signed Distance Field

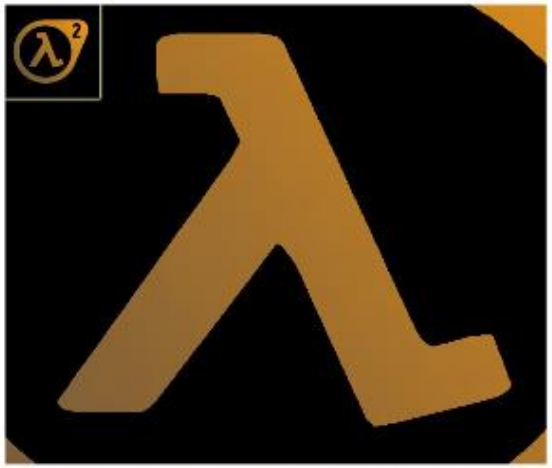
- Better solution
  - Signed Distance Field
    - Solves the artifacts problem of alpha-tested method
    - Lower memory usage: 8-bit texture format
    - As fast as standard texture mapping
    - Add at most a few instructions to the pixel shader
  - Published paper by Valve
    - Search for SIGGRAPH2007_AlphaTestedMagnification.pdf

Font – Signed Distance Field

(a) 64x64 texture, alpha-blended



(b) 64x64 texture, alpha tested

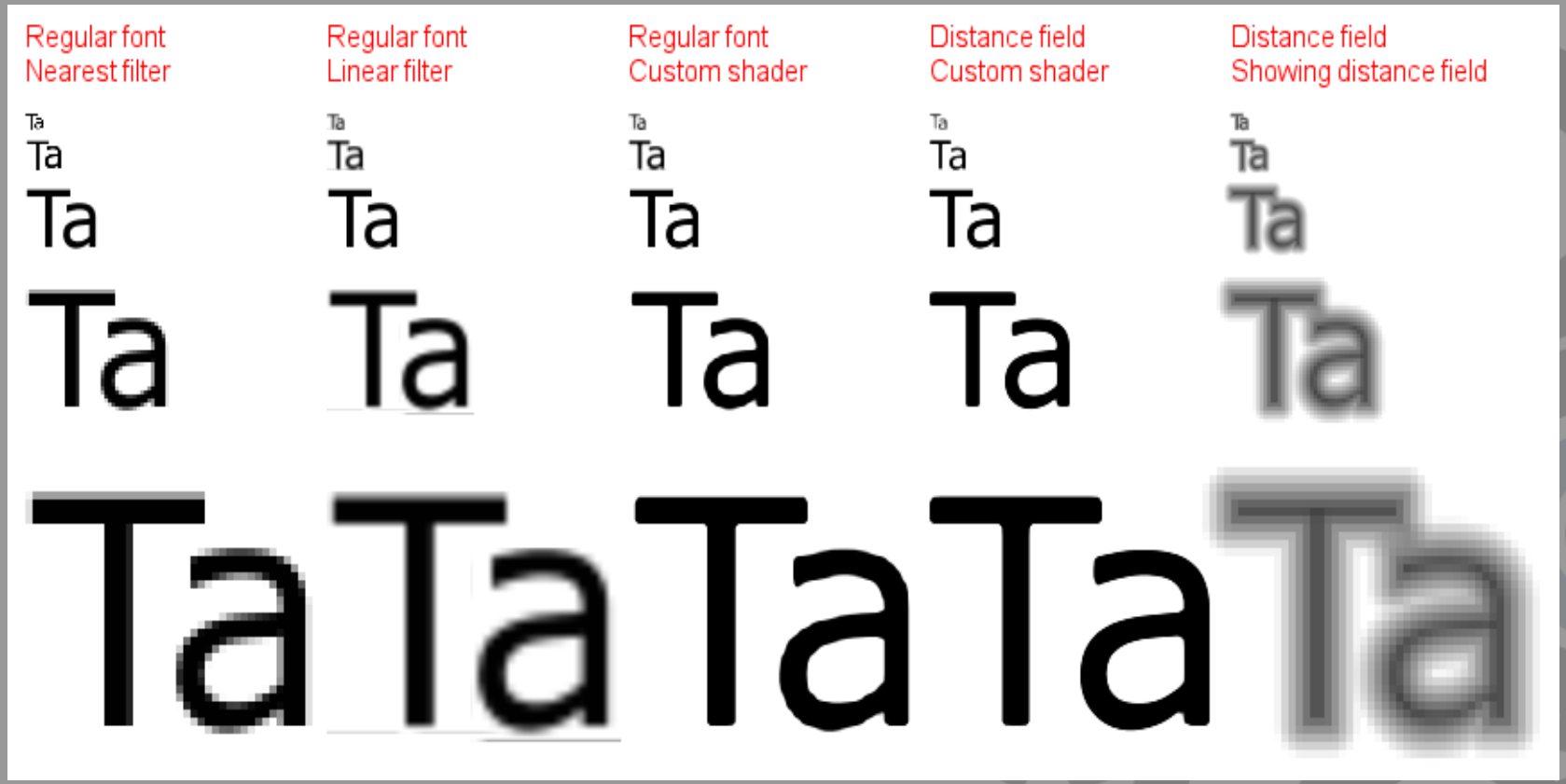


(c) 64x64 texture using our technique

Figure 1: *Vector art encoded in a 64x64 texture using (a) simple bilinear filtering (b) alpha testing and (c) our distance field technique*

Courtesy of ”Valve Software“

Font – Signed Distance Field



Courtesy of ”libgdx“

Font – Signed Distance Field

Showing
distance field
generated
out of a .ttf file

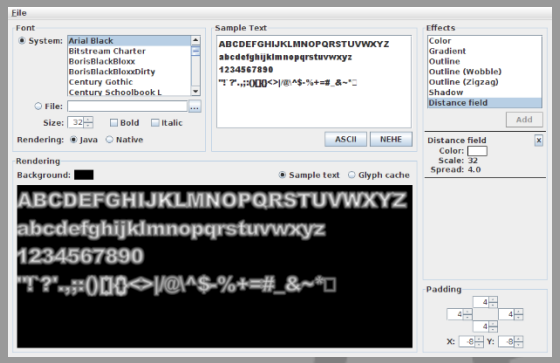
Font – Signed Distance Field

The distance, in texels, is computed from the actual texel state to the nearest opposite state.

Think about the outline of the font as its borders.

Font – Signed Distance Field

- In Practice
  - Traditional: Load a high-resolution bitmap font, do the signed distance field computation and generate a low-res 8-bit SDF texture.
  - Tools: Use a tool to generate the SDF bitmap font texture
    - » e.g., *Hiero*:



  - Use true type font library
    - » Load a *.ttf* font at high resolution
    - » Convert every character to a SDF low-res texture

Font – Signed Distance Field

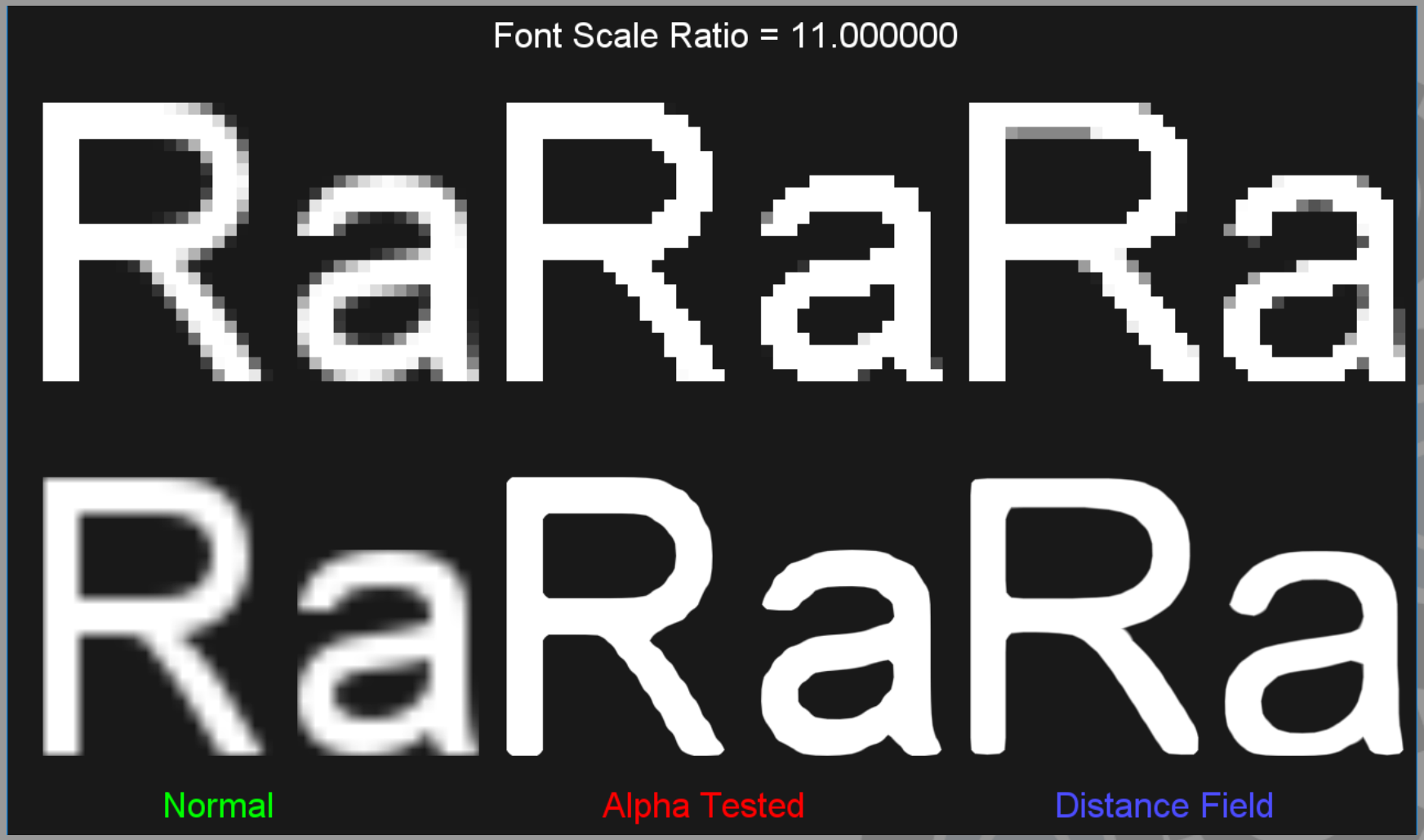
- In Practice
  - Customize your fragment/pixel shader to support smooth step alpha filtering

```
uniform sampler2D u_texture;

varying vec4 v_color;
varying vec2 v_texCoord;

const float smoothing = 1.0/16.0;

void main() {
    float distance = texture2D(u_texture, v_texCoord).a;
    float alpha = smoothstep(0.5 - smoothing, 0.5 + smoothing, distance) * v_color.a;
    gl_FragColor = vec4(v_color.rgb, alpha);
}
```

Courtesy of ”libgdx“

Font – Signed Distance Field

Demonstrated in class



Font – Signed Distance Field Demo

Font – Signed Distance Field

- Additional resources

  https://github.com/libgdx/libgdx/wiki/Distance-field-fonts