GPU Font Rendering
Current State of the Art

Eric Lengyel, Ph.D.
Terathon Software
GPU Font Rendering: Current State of the Art

May 7, 2018
Irvine, California

About the speaker

- Working in game/graphics dev since 1994
  - Previously at Sierra, Apple, Naughty Dog

- Current projects:
  - Slug Library, Tombstone Engine, The 31st, FGED
About this talk

- Unicode
- Glyphs
- TrueType
- Font Rendering
- Typography
Unicode

- Defines character codes
- Originally 16-bit
- Now has range `0x000000` – `0x10FFFF`
- Divided into 17 “planes”
Basic Multilingual Plane

- 0x0000 – 0xFFFF
- First 128 code points are ASCII
- Lots of other common scripts/languages
- Lots of common symbols
### Basic Multilingual Plane

<table>
<thead>
<tr>
<th>Code Point</th>
<th>Character Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>00-0F</td>
<td>Latin script</td>
</tr>
<tr>
<td>10-1F</td>
<td>Non-Latin European scripts</td>
</tr>
<tr>
<td>20-2F</td>
<td>African scripts</td>
</tr>
<tr>
<td>30-3F</td>
<td>Middle Eastern and Southwest Asian scripts</td>
</tr>
<tr>
<td>40-4F</td>
<td>South and Central Asian scripts</td>
</tr>
<tr>
<td>50-5F</td>
<td>Southeast Asian scripts</td>
</tr>
<tr>
<td>60-6F</td>
<td>East Asian scripts</td>
</tr>
<tr>
<td>70-7F</td>
<td>CJK characters</td>
</tr>
<tr>
<td>80-8F</td>
<td>Indonesian and Oceanic scripts</td>
</tr>
<tr>
<td>90-9F</td>
<td>American scripts</td>
</tr>
<tr>
<td>A0-AF</td>
<td>Notational systems</td>
</tr>
<tr>
<td>B0-BF</td>
<td>Symbols</td>
</tr>
<tr>
<td>C0-CF</td>
<td>Private use</td>
</tr>
<tr>
<td>D0-DF</td>
<td>UTF-16 surrogates</td>
</tr>
<tr>
<td>E0-EF</td>
<td>Unallocated code points</td>
</tr>
</tbody>
</table>

As of Unicode 10.0
Supplementary Multilingual Plane

- 0x010000 – 0x01FFFF
- Rare characters from many languages
- Rare scripts like Cuneiform and Hieroglyphs
- Mathematical symbols and bold / italic
- Emoticons 😁
# Supplementary Multilingual Plane

<table>
<thead>
<tr>
<th>Non-Latin European scripts</th>
<th>African scripts</th>
<th>Middle Eastern and Southwest Asian scripts</th>
<th>South and Central Asian scripts</th>
<th>Southeast Asian scripts</th>
<th>East Asian scripts</th>
<th>American scripts</th>
<th>Hieroglyphs</th>
<th>Notational systems</th>
<th>Symbols</th>
<th>Unallocated code points</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
<td>108</td>
<td>109</td>
<td>110</td>
</tr>
<tr>
<td>110</td>
<td>111</td>
<td>112</td>
<td>113</td>
<td>114</td>
<td>115</td>
<td>116</td>
<td>117</td>
<td>118</td>
<td>119</td>
<td>120</td>
</tr>
<tr>
<td>120</td>
<td>121</td>
<td>122</td>
<td>123</td>
<td>124</td>
<td>125</td>
<td>126</td>
<td>127</td>
<td>128</td>
<td>129</td>
<td>130</td>
</tr>
<tr>
<td>130</td>
<td>131</td>
<td>132</td>
<td>133</td>
<td>134</td>
<td>135</td>
<td>136</td>
<td>137</td>
<td>138</td>
<td>139</td>
<td>140</td>
</tr>
<tr>
<td>140</td>
<td>141</td>
<td>142</td>
<td>143</td>
<td>144</td>
<td>145</td>
<td>146</td>
<td>147</td>
<td>148</td>
<td>149</td>
<td>150</td>
</tr>
<tr>
<td>150</td>
<td>151</td>
<td>152</td>
<td>153</td>
<td>154</td>
<td>155</td>
<td>156</td>
<td>157</td>
<td>158</td>
<td>159</td>
<td>160</td>
</tr>
<tr>
<td>160</td>
<td>161</td>
<td>162</td>
<td>163</td>
<td>164</td>
<td>165</td>
<td>166</td>
<td>167</td>
<td>168</td>
<td>169</td>
<td>170</td>
</tr>
<tr>
<td>170</td>
<td>171</td>
<td>172</td>
<td>173</td>
<td>174</td>
<td>175</td>
<td>176</td>
<td>177</td>
<td>178</td>
<td>179</td>
<td>180</td>
</tr>
<tr>
<td>180</td>
<td>181</td>
<td>182</td>
<td>183</td>
<td>184</td>
<td>185</td>
<td>186</td>
<td>187</td>
<td>188</td>
<td>189</td>
<td>190</td>
</tr>
<tr>
<td>190</td>
<td>191</td>
<td>192</td>
<td>193</td>
<td>194</td>
<td>195</td>
<td>196</td>
<td>197</td>
<td>198</td>
<td>199</td>
<td>200</td>
</tr>
<tr>
<td>200</td>
<td>201</td>
<td>202</td>
<td>203</td>
<td>204</td>
<td>205</td>
<td>206</td>
<td>207</td>
<td>208</td>
<td>209</td>
<td>210</td>
</tr>
</tbody>
</table>

As of Unicode 10.0
Supplementary Ideographic Plane

- 0x020000 – 0x02FFFF
- Less common CJK ideographs
## Supplementary Ideographic Plane

<table>
<thead>
<tr>
<th>Code Point</th>
<th>CJK characters</th>
<th>Unallocated code points</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 - 20F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>210 - 21F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>220 - 22F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>230 - 23F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>240 - 24F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>250 - 25F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>260 - 26F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>270 - 27F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>280 - 28F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>290 - 29F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2A0 - 2AF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2B0 - 2BF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2C0 - 2CF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2D0 - 2DF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2E0 - 2EF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2F0 - 2FF</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As of Unicode 10.0
Other Planes

- Planes 0x03 – 0x0D unused
- Plane 0x0E contains special tags and variation selectors
- Planes 0x0F and 0x10 for private use only
Character Encoding

- ASCII
- UCS-2 (Universal Coded Character Set)
  - Always 16 bits per character
- UTF-16
  - 16 bits or 32 bits per character
- UTF-32
  - Always 32 bits per character
UTF-8

- 1 – 4 bytes per character
  - Using variable-length encoding

- Values 0x00 – 0x7F identical to ASCII

- High bit set indicates part of multi-byte sequence
UTF-8

- 1 byte: 0x00 – 0x7F
- 2 bytes: 0x0080 – 0x07FF
- 3 bytes: 0x0800 – 0xFFFF
- 4 bytes: 0x010000 – 0x10FFFF
Glyphs

- Fonts contain glyphs
- Glyphs have font-specific internal numbering
- Fonts contain tables that map character codes (Unicode values) to glyph indexes
Glyphs

- Fonts typically contain many more glyphs that are not directly mapped from characters
  - Type variations
  - Alternate styles
  - Ligatures, ZWJ sequences
  - Initial, medial, final forms (Arabic)
- More about these later
TrueType

- Contains resolution-independent representations of glyph outlines
- Has character-to-glyph mappings
- Usually contains several other tables with typographic information (e.g., kerning)
Glyph Outline

- Glyph defined by one or more closed contours
- Each contour defined by continuous sequence of quadratic Bézier curves
- Winding number determines whether a given point is inside the glyph
Winding Number

- Contours defining outer edge of glyph wound in one direction (either CW or CCW is okay)

- Contours defining a hole in the glyph wound in the opposite direction
Winding Number

- Count number of positive loops for outer contours
- Subtract number of negative loops for inner contours
- Nonzero means point inside glyph boundary
Glyph Outline / Winding Number
Font Rendering in Games

- Text rendered in lots of places
  - GUI: Buttons, menus, ...
  - HUD: Score, health, ammo, ...
  - In scene: Signs, labels, computer screens, ...
  - Debug info: Console, stats, timings, ...
Basic GPU Font Rendering

- Rasterize each glyph on CPU and store results in a texture map called an “atlas”
- Can be done for multiple font sizes at once
- Packing methods can vary in sophistication
Font Atlases

Image credit: freetype-gl
Font Atlases

- Render one quad for each glyph
- Texture map the glyph’s image from the atlas
- Very simple and stupid fast
Font Atlases

- Very limited quality
- Only looks good at originally rendered size
- Magnification looks terrible
Font Atlases

- Minification also problematic
- Mipmaps work to a degree
- Glyphs must be surrounded by empty space in atlas to prevent bleeding into neighbors
Signed Distance Fields

- Instead of storing glyph images in atlas, store distance to glyph outline at each point

Image credit: Konstantin Käfer, "Drawing Text with Signed Distance Fields in Mapbox GL", 2014.
Signed Distance Fields

- Render linear coverage by scaling distance to pixel units and clamping
- Requires derivatives in pixel shader and extra computation
- Still very fast
Signed Distance Fields

- Addresses magnification problem
- Also allows good perspective rendering

Image credit: Chris Green, "Improved Alpha-Tested Magnification for Vector Textures and Special Effects", 2007.
Signed Distance Fields

- Need high resolution to capture glyph details
- Sharp corners always rounded off
  - Can be addressed with multiple distance channels
- Minification becomes bigger problem
  - Because one distance value can’t account for multiple curves in scaled-down field
Signed Distance Fields

Resolution Independence

- Render directly from original outline data
  - Control points for quadratic Bézier curves

- No more texture atlases!
  - No resolution-dependent approximation
  - Impossible to lose detail
Loop-Blinn Method

- Creates a triangulation for each glyph using its outline control points
- Each triangle corresponds to one Bézier curve
- Simple calculation based on interpolated texture coordinates yields inside/outside state
Loop-Blinn Method

Loop-Blinn Method

- Needs further subdivision for interior triangles so they never border more than one curve

- Correct antialiasing also requires more triangles in the exterior
  - Consider a pixel intersecting the outline but without its center covered by a triangle
Loop-Blinn Method

Loop-Blinn Method

- Requires a large number of triangles for each glyph
- More complex glyphs could require 1000s!
- Calculation of triangles is complex
Loop-Blinn Method

- Produces high-quality magnification

- However, minification is poor
  - Any pixel is covered by at most one triangle
  - Each triangle corresponds to only one curve
  - Thus, it’s impossible for one pixel to consider contribution from multiple nearby curves
Dobbie Method

- Covers each glyph with a single quad

- Pixel shader considers subset of all Bézier curves to determine winding number

- Basically ray tracing glyphs
Dobbie Method

- For a given point, shoot a ray outward and count curve intersections
- An intersection makes a positive or negative contribution based on its winding direction
- Nonzero total means inside glyph boundary
Dobbie Method

- Antialiasing possible along ray direction

- If intersection occurs within pixel, it makes a fractional contribution

- Test rays in multiple directions and average to get isotropic antialiasing
Dobbie Method
Dobbie Method

- Very slow to test all Bézier curves defining the glyph for each ray

- Dobbie method divides glyph’s bounding box into cells

- Each cell has list of intersecting curves
Dobbie Method

Dobbie Method

- Pixel footprint could overlap multiple cells
  - Have to sort that out in pixel shader

- Need to precompute whether cell center inside or outside glyph boundary
  - Then trace extra ray from pixel location to cell center to fix up winding number
Dobbie Method

- There’s a serious problem:
- Numerical robustness
- Floating-point round-off error causes rendering artifacts
Dobbie Method

Sparkle / streaking artifacts
Glyphy

- Similar to Dobbie method in that a glyph is covered by a single quad

- Pixel shader determines distance to nearest Bézier curve
Glyphy

- Original outlines not preserved
- Also has numerical robustness problems
Glyphy

Straight lines rounded

Sparkle artifacts
GPU Font Rendering: Current State of the Art

May 7, 2018
Irvine, California
Slug Library

- The result of my own research begun in 2016
- Uses one quad per glyph
- Calculates winding number in pixel shader
- Has *perfect* numerical robustness
Numerical Robustness

- Round-off errors in previous methods:
  - Generally come from determining whether roots of ray-curve intersections fall in [0,1] range
  - Problems typically occur at the endpoints
  - Especially bad when ray nearly tangent to curve
  - Hacks like using epsilons or perturbing coordinates just shift the problem cases around
Numerical Robustness

- Only way to solve is to completely eliminate the [0,1] range test

- Slug introduces an equivalence class algorithm
  - Equivalence class represents control point state
  - Same actions taken for all cases in same class
Equivalence Classes

- With respect to a given ray, a particular quadratic Bézier curve is classified into one of 8 possible equivalence classes.

- Based on which side of ray each of three control points falls, positive or negative:
  - Exactly on ray is considered positive.
Equivalence Classes

A

B

C

D

E

F

G

H
Equivalence Classes

- For each Bézier curve, always calculate roots

\[ (y_1 - 2y_2 + y_3)t^2 - 2(y_1 - y_2)t + y_1 \]

\[ t_1 = \frac{b - \sqrt{b^2 - ac}}{a} \quad \text{and} \quad t_2 = \frac{b + \sqrt{b^2 - ac}}{a} \]

\[ a = y_1 - 2y_2 + y_3 \quad b = y_1 - y_2 \quad c = y_1 \]
Equivalence Classes

- A 16-bit LUT tells us what to do with roots for each equivalence class (8 classes x 2 roots)

- Action taken only when x coordinate positive at a root, meaning intersection was on ray
Winding Number

- 1 in LUT for first root means add one
- 1 in LUT for second root means subtract one

- Total after considering all curves is winding number at pixel location
- Fractional values used when roots within pixel distance of ray origin
Antialiasing

- Result is coverage value with perfect one-dimensional antialiasing
- Evaluate horizontal and vertical rays
- Combine to produce 2D antialiasing
Banding

- For best performance, we want to minimize the number of curves tested.
- Cells don’t work well:
  - Pixel footprint can cover multiple cells
  - Pixels get larger as font size decreases
Banding

- Instead of cells, use horizontal and vertical bands that extend to infinity
Banding

- Bézier curves are sorted into the bands
  - A curve can belong to multiple bands
  - When rendering, band selected based on ray origin

- Doesn’t matter how large pixel footprints get
  - Only matters in ray direction
  - Band parallel to ray extends forever
Banding

- Curves in each band are sorted to allow early exit in pixel shader

- Once right-pointing ray’s origin is beyond maximum curve $x$ coordinate, we’re done
Banding

- Curves sorted in both directions
- Ray points left or right depending on pixel position within a band
- Reduces number of curves tested
Bandings

- We want worst-case band to contain fewest curves possible

- GPU thread coherence will make shader wait for longest number of loop iterations in a group of pixels (32 or 64)
Banding

- Use large number of bands
- Merge those with equal subsets of Bézier curves
Minification

- High-quality minification achieved with adaptive supersampling
  - Based on screen-space derivatives

- Already have perfect 1D antialiasing
  - Take $n$ samples in $x$ and $y$ directions
  - Produces better than $n \times n$ supersampling
Minification
Font Data

- Two texture maps, data only (no images)
- Curve texture, 4 x 16-bit float
  - Contains all Bézier curves
- Band texture, 4 x 16-bit integer
  - Contains curve subsets for all bands
Multicolor Glyphs

- Microsoft fonts use vector data for color emoji
- Layered glyphs with color palette
- Easy to handle with loop in pixel shader
Typography

- Slug algorithm can make individual glyphs look great at any scale or from any perspective.

- Higher-level:
  Make entire lines of text look good.
Metrics
Metrics

em size
baseline
leading
baseline

ABCXYZ

cap height

ascent
descent

ex height
Kerning

- Some pairs of glyphs appear to the eye to have too much space in between.

- Fonts usually contain kerning tables to improve overall appearance.
Kerning

“Too Wavy.”

“Too Wavy.”

Kerning off
Kerning on
Ligatures

- Replaces a sequence of glyphs with one new glyph that looks better

- In some languages, ligatures that change appearance are required for correctness
Ligatures

The firefly craft.
The firefly craft.

Normal text

With ligatures
ZWJ Sequences

- Unicode has control character “zero-width joiner” (ZWJ)

- Often used by fonts for combining several glyphs into single ligature
ZWJ Sequences
Combining Marks

- Unicode defines many accents and other symbols that are designed to combine with a preceding base character.

- Fonts determine how this combination works by defining attachment points.
Combining Marks

\[ a \sim \ddot{a} \]
Alternate Substitution

- OpenType fonts define a large array of substitution features
- Independent of Unicode
- Not directly accessible through characters
Alternate Substitution

- Small caps
- Subscripts and superscripts
- Case-sensitive forms
- Stylistic alternates
- Tabular and proportional figures
- Lining and old-style figures
Small Caps

Text

Small caps alternates

Text

Scaled glyphs
Lining and Old-style Figures

0123456789

0123456789
Cursive Joining

- In languages like Arabic, letters have multiple forms depending on position in word
- Isolated, initial, medial, final forms
- Do not have separate character codes
Cursive Joining

متفقدة تقديم الخط وتخطيط النص

متقدمة تقديم الخط وتخطيط النص
Materials

- Rendering glyphs outputs coverage value
  - (Plus color for multi-color emoji)

- Can be combined with other materials in game
Materials
Contact / More Info

- lengyel@terathon.com
- Twitter: @EricLengyel
- sluglibrary.com