Rendering mode specific hints

Implementation guide - **DRAFT**

# Abstract

Historically, TrueType hints round outline coordinates to full-pixel boundaries. While this is useful for black and white rasterization, it is not optimal for grey scale or subpixel antialiasing. A better approach for antialiasing rasterizers is to round outlines to the fractions of a pixel that correspond to the antialiasing method. For example, if the rasterizer is using a 4x4 antialiasing technique, then text will better preserve proportions and weight if hints are rounded to ¼ of a pixel. This whitepaper is a guide to implementing this technique in font development tools that generate TrueType hints (e.g. font editors, auto-hinters, etc).

# Introduction

TrueType hints were originally conceived to improve font rendering on low resolution screens, often using pure black and white, or bi-level rasterization. To achieve the best rendering quality in this environment, hints distorted a character’s outline to align with the pixel grid. As a result, pixels were either pure black or pure white, and stem widths were full multiples of pixels. At small sizes, stems are one pixel wide. As the size increases, stems jump from one to two to three and more pixels wide, causing text to look artificially light or bold before and after one of these cutover points. However, with bi-level rendering, there were no other options.

Since then, font rasterizers have switched to use a variety of antialiasing techniques including grey scaling, ClearType color subpixel rendering, ClearType with vertical antialiasing, and monochrome variants of ClearType subpixel rendering. These techniques render pixels with some intermediate level of grey or color, proportionate to how much of the filled in letter is contained in the pixel. If only 51% of the pixel is covered by the letterform, then the pixel would be drawn with only 51% black, instead of the 100% black that bi-level rendering would use.

## Oversampling in antialiasing

Most antialiasing techniques use the technique of oversampling. Rather than generating only the number of pixels required for a given size, the rasterizer will first create a bi-level rendering with many more subpixels than is necessary, then map those black & white subpixels to an appropriate color for the overall pixel. For example, a 4x4 greyscale antialiasing technique will rasterize text with four times the number of subpixels in each direction, for a total of 16 subpixels for each real pixel. To determine the appropriate shade of grey to use for the pixel, the rasterizer essentially counts how many subpixels were filled in, divides by the total number subpixels (e.g. 16), and uses that level of black. If 13 subpixels were painted black, then the overall pixel would have a shade of 13/16 = 81% black.

Even though rasterizers can only light up full pixels in the end, oversampled antialiasing enables rasterizers to render more subtlety. The size of a pixel is still quite small at typical viewing distances, so the shades of grey or color can give the illusion of greater precision. A lowercase e that use one row of subpixels at the top will look slightly smaller than one with two rows of subpixels at the apex, despite using the same number of real pixels, because the larger e has darker pixels at the top.

{add picture here}

Unfortunately, traditional TrueType hinting prevents this from ever happening because the hints would round the outlines in both cases to the same full pixel boundary. The hints unnecessarily constrain the rendering.

## A new hinting style

This implementation guide proposes a different technique: rather than always rounding to whole pixel boundaries, hints would round to fractional pixels – i.e. the subpixels used in oversampling. For example, in 4x4 rendering, hints would round to quarter pixels in both directions. In 6x1 oversampling (used by ClearType), they would round to one sixth of a pixel in the horizontal direction, and full pixels in the vertical direction.

This style of hinting provides the consistency in weight, sharpness, and proportion that hints have always done, but also enables more fine-grained subtlety in the final rendering. Stem widths need not be constrained to full-pixel widths, thus can increase naturally with size giving better proportion and weight distribution. As an added bonus, a waterfall using this style of hinting smoothly transitions between sizes, without the weight jumps of full-pixel rounded hints. When used with a carefully-tuned GASP table, subpixel-rounded hints provide a single method for hinting that works well on both low and high-dpi screens.

{Insert pictures here}

Microsoft Sitka, which debuted in Windows 8.1, was the first of Microsoft’s fonts to make use of this technique. Microsoft Visual TrueType 6.0 (VTT) adds a new high-level VTT Talk commands to explicitly use this functionality. For example, the YLink() command rounds to full pixels, and the ResYLink() command rounds to subpixels.

Conceptually, implementing subpixel-rounding hints is easy: The first step to implementing subpixel-rounded hints is for the font’s Pre-Program to query the rasterizer to determine the rendering mode, thus the rounding granularity. It is important to dynamically determine the granularity at run time because operating systems, applications, and GASP table settings all can select different rendering modes from the rasterizer. The “Determining rounding granularity” section walks through this in more detail with references to the sample code included in this document.

Next, each font’s Glyph Program uses the above rounding granularity to round outline coordinates. Unfortunately, there are no native TrueType instructions to do this – MDRP[m<RGr] will still round to full pixel boundaries. Instead, fonts should include subroutines in their Font Program for doing the rounding. The Glyph Program should call these subroutines instead of using native TrueType alignment instructions. The “Rounding outlines” section gives more detail and explains the sample code.

### A note on sample code

Sample TrueType code accompanies this document to help illustrate these techniques. This code is released under an MIT License so that you may use this code in own implementation.

# Determining rounding granularity

Rasterizers use different levels of oversampling for each antialiasing scheme. ClearType takes advantage of color subpixels to triple the horizontal resolution, so it uses 6x1 oversampling (twice the tripled horizontal resolution). ClearType with vertical antialiasing uses a 6x5 oversampling. Windows 8 introduced two greyscale subpixel rendering modes, 8x1 and 4x4. Bi-level rendering is essentially the same as 1x1 oversampling. Because there are such a variety of antialiasing modes, font hinting software should not assume a single rounding granularity.

Not only are there many rendering modes, but they can all be used at any time on a given system. For example, applications on Windows 8.1 can request to render text either ClearType (6x1) or ClearType Grey (8x1). Further, the font’s GASP table determines whether the font uses asymmetric rendering (6x1 or 8x1) or symmetric rendering (6x5 or 4x4). Thus, a font on a Windows 8.1 system could be rendered in any of these four modes, even within the same application, at the same time. Therefore, the determination of rounding granularity must be done dynamically, preferably in the Pre-Program.

## Implementation

The Pre-Program can determine rounding granularity getting the current rasterizer configuration via the GETINFO[] instruction, and then selecting an appropriate granularity for that configuration. The following are the GETINFO[] selectors and results to query in order to determine the rasterizer configuration.

|  |  |  |
| --- | --- | --- |
| Selector | Result | Meaning |
| 32 (bit 5 set) | Bit 12 | If bit 12 is set, Greyscale is enabled |
| 64 (bit 6 set) | Bit 13 | If bit 13 is set, ClearType is enabled |
| 2048 (bit 11 set) | Bit 18 | If bit 18 is set, vertical antialiasing is enabled |
| 4096 (bit 12 set) | Bit 19 | If bit 19 is set, ClearType Grey is enabled |

Note that these selectors are only implemented in Microsoft Windows rasterizers. The Pre-Program should check the rasterizer version (GETINFO[] selector 1, result in the range 35-64, inclusive) before checking the above flags.

Having determined the rasterizer configuration, the Pre-Program should set the rounding granularity as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Greyscale | ClearType | Vert. Antialias | ClearType Grey | X granularity | Y granularity |
| True | Any | Any | Any | 4 | 4 |
| False | True | False | False | 6 | 1 |
| False | True | True | False | 6 | 5 |
| False | True | False | True | 8 | 1 |
| False | True | True | True | 4 | 4 |
| False | False | Any | Any | 1 | 1 |

## Sample code

The file get\_granularity.txt that accompanies this guide includes Function 84 from Microsoft Sitka. Function 84 has two halves:

The first half queries the rasterizer configuration for the above flags (as well as a few others that are useful, but out of the scope of this whitepaper) and sets a bitmask of the rendering mode flags, which it stores in storage location 2. This implementation is a bit more complex than necessary because it uses a nested set of IF[] statements to query specific versions of the rasterizer. These version checks are unnecessary for the above flags we are concerned with in this whitepaper since older Windows rasterizers will return 0 for any flags they do not have defined. The only version check that is necessary is to check that the current rasterizer is one of the Windows versions (35-64).

The second half of function 84 uses the bitmask generated in the first half to store the appropriate rounding granularity (samples/pixel) in locations 8 and 9, as well as saving other useful rendering parameters in locations 5, 12, and 13. The subroutines for rounding points use storage locations 8 and 9 for their granularity. Note that because math in TrueType is done in 26.6 fixed point, this implementation stores 64 for 1x1, thus 64\*4 = 256 for 4x4 subpixel granularity, etc.

# Rounding outlines

Rather than rely exclusively on native TrueType instructions, which only round to full pixel boundaries, this implementation uses subroutines to round to subpixel granularities. Hints in the Glyph Program should call these subroutines rather than using TrueType instructions like MDRP.

## Implementation

Before executing hints, the Glyph Program should first get the appropriate rounding granularity for the current projection vector (Function 79 in the sample code implements this). Since Glyph Programs generally only hint purely in the X or Y direction at any given time, then only the granularity in the current direction is needed.

Next, the Glyph Program executes its hints, calling rounding functions instead of using built-in TrueType instructions like MDAP, etc. The rounding operation is a straightforward mathematical round based on the granularity determined above. Generally, developers can implement a shared core rounding function, that is wrapped in a layer of functions that are specific to particular types of hints (e.g. Links, Anchors, etc).

## Sample code

To jumpstart development, this guide includes the file rounding.txt, which contains a complete sample implementation of rounding functions. The top-level functions exactly map to TrueType instructions like MDAP, MDRP, etc to make porting easy. These functions depend on function 84 having already been executed in the Pre- and Glyph Programs.

For example, traditional TrueType hints might execute the following:

{CODE GOES HERE}

Instead, a font using subpixel rounding would execute:

{CODE GOES HERE}

The top-level functions are listed below:

{PUT FUNCTION DESCRIPTIONS HERE}

# Hinting recommendations

Hinters can mix native TrueType hinting instructions with the above subpixel rounding functions. Generally speaking, hinters should use subpixel rounding functions for stem widths, but should anchor stems with native instructions like MIAP. More detailed recommendations for using subpixel rounded hints will appear in a forthcoming whitepaper.

For example, the following is the high-level VTT Talk for Microsoft Sitka’s lowercase e.

/\* Y direction \*/

YAnchor(6,9)

ResYLink(6,30,303)

YAnchor(16,1)

ResYLink(16,41,300)

YIPAnchor(16,35,6)

YIPAnchor(16,36,6)

ResYLink(36,26,304)

/\* X direction \*/

Smooth()

The Res\*() instructions will use subpixel rounding functions and the rest will use native TrueType. The above compiles to the following TrueType code:

SVTCA[Y]

MIAP[R], 6, 9

CALL[], 6, 30, 303, 108

MIAP[R], 16, 1

CALL[], 16, 41, 300, 108

SRP2[], 6

IP[], 35

MDAP[R], 35

SRP1[], 16

IP[], 36

MDAP[R], 36

CALL[], 36, 26, 304, 108

IUP[Y]

IUP[X]

# MIT License

The sample code provided in this document and in the supporting files is licensed under an MIT License:

The MIT License (MIT)

Copyright (c) 2015 Microsoft Corporation

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell  
copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.