Real-Time Grid Fitting of Typographic Outlines

David Turner
david@freetype.org

Werner Lemberg
Kl. Beurhausstr. 1
D-44137 Dortmund, Germany
wl@gnu.org

Abstract

This paper describes an auto-hinting algorithm to grid fit glyph outlines. Instead of generating new hints for font files or font editors, the computations are done in real-time while rendering the glyph, posing strong efficiency constraints.

A freely available implementation can be found in the FreeType font rendering library.

Résumé

Cet article décrit un algorithme d'auto-*hinting* pour des contours de glyphes définis sur une grille. Au lieu de générer des nouveaux *hints* pour des fichiers de fonte ou des éditeurs de fonte, les calculs sont effectués à la volée au moment de l'affichage du glyphe, ce qui pose des sérieuses contraintes d'efficacité.

On trouvera une implémentation librement disponible de cet algorithme dans la bibliothèque de sous-routines d'affichage de fonte FreeType.

Overview

The auto-hinting algorithm is roughly divided into two distinct parts.

- A *feature analysis* phase to study each glyph outline in order to detect interesting 'features' in them.
- A *grid fitting* phase to adjust the position of such features to the device pixel grid, and to align the outline points to these.

In the following we directly reference files and structures of the auto-hinter in the FreeType library, version 2.1.4. This should help in understanding how the algorithm is implemented.

Feature Analysis

To produce well-hinted glyphs it is not sufficient to improve the shape of a glyph as much as possible. It is also necessary to find font-wide parameters.

Global Analysis The first step is to find global metrics. In the following it is assumed that the font is Latin-based. Font specific details regarding how to map an input character to an output glyph are omitted for simplicity.

- Compute the standard stem widths and heights of the glyphs. This is done by loading the letter 'o' and running the feature analysis on it to get the sizes of its 'stems'.

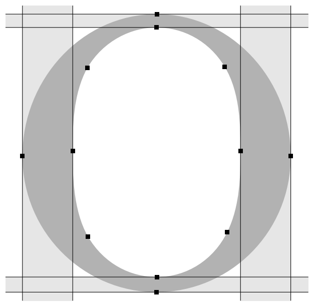
FIG. 1: The stem widths and heights of the glyph 'o'.

- Compute the *blue zones*. Outlines of certain characters such as 'Z', 'C', etc., are loaded, comparing the top-most and bottom-most coordinates to find the small horizontal zones which envelop them.

All of this is done in the C source file `ahglobals.c`. These global parameters are computed only once per font, then stored for later use during the grid fitting process.

Glyph Analysis An *outline* is the ordered set of all points defining a glyph. It consists of one or more *contours*. A contour is a closed curve; points can be control points or

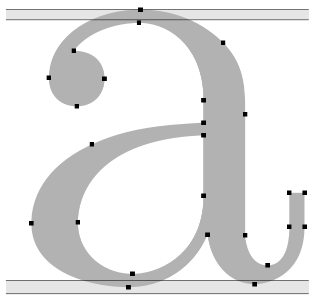

Fig. 2: Two blue zones relevant to the glyph 'a'. Vertical point coordinates of *all* glyphs within these zones are aligned.

on the contour. For example, the outline of glyph 'a' in figure 2 consists of two contours.

Each outline is decomposed into an AH_Point array.

```
struct AH_Point
{
  AH_Flags      flags;
  FT_Pos        ox, oy;
  FT_Pos        fx, fy;
  FT_Pos        x,  y;
  FT_Pos        u,  v;

  AH_Direction  in_dir;
  AH_Direction  out_dir;

  AH_Point      next;
  AH_Point      prev;
}
```

The field flags specifies the type of the current point: a point on the contour, a conic control point for a quadratic Bézier curve as used in TrueType fonts, or a cubic control point for third-order Bézier curves as used in Type 1 fonts. ox and oy are the original scaled coordinates, fx and fy the coordinates in font units, and x and y hold the hinted coordinates (which we are really interested in).

in_dir and out_dir give the direction of the vectors from the previous and to the next point in the current contour. Finally, it is easy to guess now that the fields next and prev hold pointers to the previous and next contour point.

Both the global and the feature analysis are performed twice: first vertically (modifying y coordinates and horizontal stems), then horizontally (modifying x coordinates and vertical stems). The results are independent: It is easy to disable hinting in one dimension without impacting the results of the other dimension.

Note that the AH_Point structure holds two fields, u and v, which are used to store coordinates whose meaning changes depending on the context. See the function

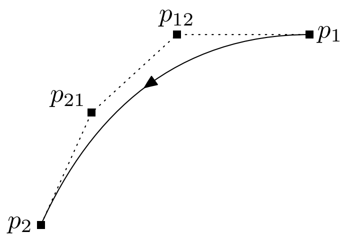

Fig. 3: A cubic Bézier curve with two control points. The 'out' direction of p_1 is AH_DIR_LEFT; all other directions are of type AH_DIR_NONE.

ah_set_uv in file ahglyph.c for details. The following 'modes' are supported:

AH_UV_FXY: (u, v) contains the original (x, y) point coordinates expressed in font units.

AH_UV_FYX: Same as above, except that 'rotated' coordinates (y, x) are used instead.

AH_UV_OXY: The pair (u, v) contains linearly scaled (ox, oy) point coordinates in 26.6 fixed-point format — not modified by grid fitting. '26.6' means that of a 32-bit integer, 26 bits are used for the integer part and 6 bits for the fractional part, providing a granularity of $\frac{1}{64}$ units.

AH_UV_OYX: Same as above, with 'rotated' (oy, ox) linear coordinates.

AH_UV_OX: A special case used during grid fitting. u corresponds to hinted coordinates x, and v to linear ones ox. This is used to perform light interpolation in the horizontal dimension.

AH_UV_OY: Same as above, but uses y and oy instead.

AH_UV_XY: (u, v) contains the scaled and hinted point coordinates (x, y). This is currently unused.

AH_UV_YX: Same as above, with 'rotated' (y, x) coordinates (also currently unused).

Directions are identified by with the following constants.

```
#define AH_DIR_NONE    4
#define AH_DIR_RIGHT   1
#define AH_DIR_LEFT   -1
#define AH_DIR_UP      2
#define AH_DIR_DOWN   -2
```

These values are carefully chosen so that $dir_1 + dir_2 = 0$ only if dir_1 and dir_2 correspond to opposite directions. This is useful in speeding up certain comparisons. In any case, these values should not be changed, since other sub-algorithms depend on them (for example, ah_outline_compute_segments assumes that AH_DIR_RIGHT is positive).

More details on the glyph analysis, like weak point flags, and major and minor directions will be explained later.

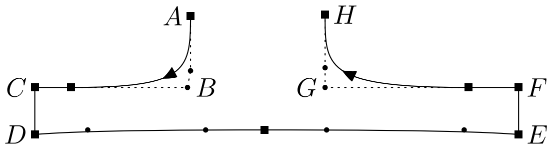

Fig. 4: A serif. Contour and control points are represented by squares and circles, respectively. The bottom 'line' DE is approximately aligned along the horizontal axis, thus it forms a segment of 7 points. Two other horizontal segments are BC and FG, while AB, CD, EF, and GH form vertical segments.

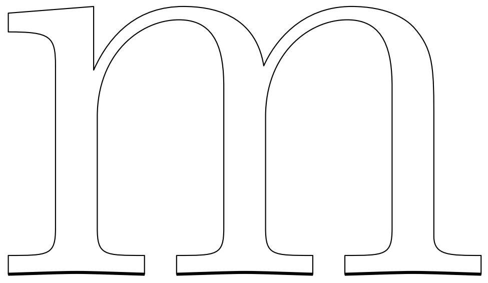

Fig. 5: The three segments marked with thick lines form an edge.

Segments `ah_outline_compute_segments` is the function used to find *segments* in an outline. A segment is a series of consecutive points that are approximately aligned along a coordinate axis. The allowed maximum deviation from a straight line is $\arctan\frac{1}{12} \approx 4.7°$ (this is a heuristic value).

A segment must have at least two points, except in the case of 'fake' segments that are generated to hint metrics appropriately, and which consist of a single point.

Each segment has a coordinate in the dimension's 'main' direction (field `pos` in the `AH_Segment` structure) and coordinates in the 'other' dimension which specifies its extrema (fields `min_coord` and `max_coord`).

As soon as segments are defined, the auto-hinter groups them into *edges* (figure 5). An edge corresponds to a single position on the main dimension that collects one or more segments (allowing for a small threshold).

The auto-hinter first tries to grid fit edges, then to align segments on the edges unless it detects that they form a *serif* (see figure 4).

Segments need to be 'linked' to other ones in order to detect *stems*. A stem is made of two segments that face each other in opposite directions and that are sufficiently close to each other. Using vocabulary from the TrueType specification, stem segments form a *black distance*. For vertical stems we have the following two cases (horizontal stems use the y axis instead for hinting):

- In TrueType fonts, the leftmost segment points upwards, and the rightmost points downwards.
- In Type 1 fonts, the reverse convention is used, i.e., the leftmost segment points downwards, and the rightmost points upwards.

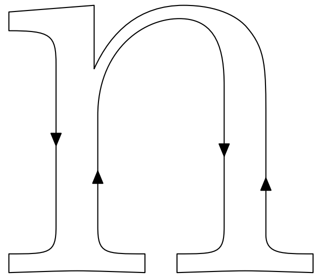

Fig. 6: The outline orientation of a glyph in a Type 1 font. TrueType fonts use the opposite direction.

Unfortunately, some fonts do not respect the fill convention of their own format, and others even contain a mixture of both conventions. To know precisely how to form black distances we thus need to analyze the glyphs, via the auxiliary function `ah_get_orientation`. It tries to guess the correct convention using the orientation of the points that make the glyph outline's bounding box.

This function is called from another function, `ah_outline_load`, which also sets two fields in the `AH_Outline` structure, namely `vert_major_dir` and `horz_major_dir`. They correspond to the direction of the leftmost or bottommost stem segments along the vertical and horizontal axis, respectively.

The algorithm to link segments is 'greedy' (that is, it will link as many segments as possible) and tries to find the 'closest' segment in the opposite direction for each candidate. Here, 'closest' means that the segments are sufficiently aligned in the 'other' dimension, and close in the 'main' one. The current code to determine this is as follows.

```
FT_Pos  min = seg1->min_coord;
FT_Pos  max = seg1->max_coord;
FT_Pos  len, dist, score;

if ( min < seg2->min_coord )
  min = seg2->min_coord;
if ( max > seg2->max_coord )
  max = seg2->max_coord;

len = max - min;
if ( len >= 8 )
{
  dist = seg2->pos - seg1->pos;
```

```
    if ( dist < 0 )
      dist = -dist;

    score = dist + 3000 / len;
    if ( score < best_score )
    {
      best_score   = score;
      best_segment = seg2;
    }
  }
```

Each segment has at most one 'best' candidate to form a black distance, or no candidate at all. Notice that two distinct segments can have the same candidate, which frequently means a serif, as in figure 4. The best candidate for both AB and CD is GH, while the best candidate for GH is AB. Similarly, the best candidate for EF and GH is AB, while the best candidate for AB is GH.

A stem is recognized by the following condition:

$$segment_{1_{best}} = segment_2 \quad \wedge \quad segment_{2_{best}} = segment_1$$

On the other hand, a serif has

$$segment_{1_{best}} = segment_2 \quad \wedge \quad segment_{2_{best}} \neq segment_1$$

where $segment_1$ corresponds to the serif segment (CD and EF in figure 4).

Stem segments store their best candidate in the `link` field of the structure `AH_Segment`, while serif segments use the field `serif` (their `link` field is set to zero).

Segments are also *round* or *flat*, depending on the series of points that define them. A segment is round if the next and previous point of an extremum (which can be either a single point or sequence of points) are both (conic or cubic) control points. Otherwise, a segment with an extremum is flat. In figure 4, the segment DE is flat because the previous point C and the next point F are both on the contour (and thus they aren't control points). In figures 1 and 2, the top segments are round.

Round and flat segments are collected for selected glyphs, then the average is taken to define blue zones and overshoot values (see figure 7).

Edges As mentioned earlier, edges are used to collect segments along coordinates in the 'main' dimension. There are three types of edges.

- Free edges which are not linked to other ones.
- Stem edges which contain at least one stem segment.
- Serif edges, which only contain one or more serif segments.

Edges can be flat (if they contain at least one flat segment) or round (if they contain only round segments).

Finally, edges can be linked to blue zones, depending on their position and the orientation of the segments they contain.

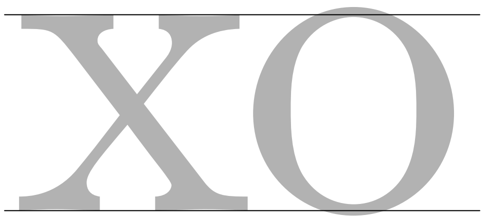

FIG. 7: The difference between the height of a glyph with a flat top segment and a glyph with a round top segment is called top *overshoot*. Analogously, the difference in depth is called bottom overshoot. Glyphs from the string 'xzroesc' are used compute the blue zone and overshoot value of the top and bottom of lowercase letters.

Grid Fitting

The collected data so far is now used to move the glyph's outline points to positions which improve the overall shape for a specific output device resolution. This complex process can be divided into the following steps.

Globals Each time the x or y scale changes, the globals are rescaled and fitted globally. This allows us to deactivate certain blue zones when they become too small or too big.

Edge Grid Fitting First of all, the edges are fitted to the pixel grid, in the following order.

1. In the vertical dimension, all edges linked to blue zones are aligned to the zone's position (we call them *blue edges*). This assures consistent glyph heights.
2. Stem edges are fitted to the pixel grid. The stem width and position are computed more or less independently.

   If a stem edge is linked to a blue edge, its position is directly computed. Otherwise, subtle alignments may occur, beyond our scope here.
3. Finally, serif edges are aligned.

The function `ah_hinter_hint_edges_3` in the file `ahhint.c` should be consulted for more details.

Computing the position and size of stems is very sensitive to tuning, containing a lot of heuristic constants. Most of the patches to the auto-hinter were refinements to the involved routines.

Segment Grid Fitting After aligning all edges the corresponding segments are fitted to the same position. This forces all points on these segments to adopt the edge's position. The auxiliary function `ah_hinter_align_edge_points` has more details.

Each point has a flag `AH_FLAG_DONE` which is set when it is aligned to a specific location. This is used for

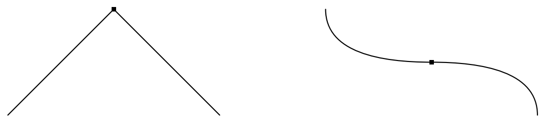

Fig. 8: An angle point (left) and an inflection point (right).

the two distinct interpolation algorithms described below.

Strong Points Experience has shown that points which are not part of an edge need to be interpolated linearly between their two closest edges, even if these are not part of the contour of those particular points. Typical candidates for this are

- *angle* points (i.e., points where the 'in' and 'out' direction differ greatly);
- *inflection* points (i.e., where the 'in' and 'out' angles are the same, but the curvature changes sign).

`ah_hinter_align_strong_points` is the function which takes care of such situations; it is equivalent to the TrueType 'IP' hinting instruction.

Weak Points Other points in the outline must be interpolated using the coordinates of their previous and next unfitted contour neighbours. These are called *weak points* and are touched by the function `ah_hinter_align_weak_points`, equivalent to the TrueType 'IUP' hinting instruction. Typical candidates are control points and points on the contour without a major direction.

The major effect is to reduce possible distortion caused by alignment of edges and strong points, thus weak points are processed after strong points.

Hinting Metrics To properly hint the advance widths of glyphs, `ah_outline_compute_segments` creates two 'fake' segments corresponding to the position of the leftmost and rightmost points in the outline (for the horizontal position only).

These segments are grid fitted, and the resulting distance between them is used to correct the scaled advance width.

See file `ahhint.c` for details; the code looks like this:

```
FT_Pos   old_advance, old_rsb, old_lsb,
         new_lsb;
/* leftmost edge */
AH_Edge  edge1 =
  outline->vert_edges;
/* rightmost edge */
AH_Edge  edge2 =
  edge1 + outline->num_vedges - 1;

old_advance = hinter->pp2.x;
old_rsb     = old_advance - edge2->opos;
old_lsb     = edge1->opos;
new_lsb     = edge1->pos;

/* round result to the
   nearest integer pixel */
hinter->pp1.x =
  ( ( new_lsb    - old_lsb ) + 32 ) & -64;
hinter->pp2.x =
  ( ( edge2->pos + old_rsb ) + 32 ) & -64;
```

where rsb and lsb represent the right and left side bearing, respectively. pp_1 and pp_2 are the auxiliary points which control the advance width. All computing is done with 26.6 fixed-point numbers.

Conclusion

The general design of FreeType's auto-hinter has been proven to be very stable; as mentioned above, only small changes have been applied, most of them to refine the computations of the position and width of stems, not the basic segment linking architecture.

In the future, we will implement additional constraints to compute more sophisticated global parameters. Here is a list of improvements which we will likely implement in the near future.

- Use more glyphs than the letter 'o' to find default stem widths and heights. This is a script dependent feature.
- Assure that the advance widths of normal digits have the same value.
- Extend blue zones to cover non-Latin languages such as Arabic. This is, select character groups for this feature dependent on the script.
- Autohinting of CJK characters needs additional analysis steps to assure constant whitespace between multiple parallel stems. Currently, only stems in 'm'-like glyphs are handled this way.

References

[1] Adobe Systems, Inc. *Adobe Type 1 Font Format.* Addison-Wesley, 3rd edition, 1993.

[2] Microsoft Corporation. OpenType specification version 1.4. Available from `http://www.microsoft.com/typography`, 2002.

[3] David Turner, Werner Lemberg, and Robert Wilhelm. The freetype 2 font rendering library. Available from `http://www.freetype.org`, 2003.