Limn

These days, fonts to be used on computers are represented in one of two ways: as a **bitmap font**, which specifies each individual pixel in the image of a character; and/or as an **outline font**, which specifies the image as a collection of mathematically-specified curves. Each method has its own advantages and disadvantages; typesetting programs, page description languages, and output devices can generally deal with both.

Limn converts a font from a bitmap to an outline by fitting curves to the pixels. Non-shape-related information in the bitmap font, such as that for the side bearings, is preserved in the outline output.

Specifically, the input is a bitmap (GF or PK) font. The output is a BZR outline font (see section BZR files), which can then be converted to (for example) Metafont or PostScript with BZRto (see section BZRto).

There is a fair amount of literature on converting bitmaps to outlines. We found three particularly helpful: Philip Schneider's Master's thesis on his system Phoenix; Michael Plass and Maureen Stone's article `Curve-fitting with piecewise parametric cubics' published in SIGGRAPH; and Jakob Gonczarowski's article `A fast approach to auto-tracing (with parametric cubics)' in the RIDT 91 conference proceedings. See the file `` `limn/README' `` for the full citations.

Limn algorithm

Limn can always (barring bugs, of course) fit some sort of outline to the bitmap input. But its default fit is likely to be far from the ideal: character features may disappear, curves distorted, straight lines turned into curves and curves into straight lines, and on and on.

To control the fitting process, you must specify options to override Limn's defaults. To describe those options, we must describe the algorithm Limn uses to do the fitting, which we do in this section. We mention the options at the appropriate point.

The next section summarizes all the options, in alphabetical order.

Here is a schematic of the algorithm. The subsections below go into detail for each step. Except for the very first step, this is implemented in `` `limn/fit.c' ``.

```
find pixel outlines
for each pixel outline:
  find corners, yielding curve lists
  for each curve list:
    remove knees
    filter
    if too small:
      fit with straight line
    otherwise fit with spline:
      set initial t values
      find tangents
      fit with one spline
      while error > reparameterize-threshold, reparameterize
      if error > error-threshold, subdivide and recurse
      if linearity < line-threshold, change to straight line
    revert bad lines
    align endpoints
```

Finding pixel outlines

The first step in the conversion from a character shape represented as a bitmap to a list of mathematical curves is to find all the cyclical outlines (i.e., closed curves) in the bitmap image. The resulting list is called a **pixel outline list**. Each **pixel outline**in the list consists of the pixel coordinates of each edge on the outline.

For example, the pixel outline list for an `` `i' `` has two elements: one for the dot, and one for the stem. The pixel outline list for an `` `o' `` also has two elements: one for the outside of the shape, and one for the inside.

But we must differentiate between an **outside outline** (whose interior is to be filled with black to render the character) and an **inside outline** (whose interior is to be filled with white). Limn's convention is to write the pixel coordinates for outside outlines in counterclockwise order, and those for inside outlines in clockwise order.

This counterclockwise movement of outside outlines is required by the Type 1 format used for PostScript fonts, which is why we adopted that convention for Limn.

For example, consider a pixel outline consisting of a single black pixel at the origin. The pixel has four corners, and hence the outline will have four coordinates. Limn looks for starting pixels from top to bottom, left to right, within a bitmap image. Thus, the list of pixel coordinates will start at (0,1) and proceed counterclockwise: (0,0) (1,0) (1,1). Here is a picture:

```
start => (0,1)<-(1,1)
           |      ^
           v      |
         (0,0)->(0,1)
```

Because finding pixel outlines does not involve approximation or estimation, there are no options to control the process. Put another way, Limn will always find the correct pixel coordinates for each outline.

Once these pixel outlines have been found, each is then processed independently; i.e., all the remaining steps, described in the following sections, operate on each pixel outline individually.

The source code for this is in `` `limn/pxl-outline.c' `` and `` `lib/edge.c' ``.

Finding corners

Recall that our final goal is to fit splines, i.e., continuous curves, to the discrete bitmap image. To that end, Limn looks for **corners** in each pixel outline (see the previous section)---points where the outline makes such a sharp turn that a single curve cannot possibly fit well. Two corners mark the endpoints of a **curve**.

We call the result a **curve list**, i.e., a list of curves on the pixel outline: the first curve begins at that first corner and continues through the second corner; and so on, until the last, which begins with the last corner found and continues through the first corner. (Each pixel outline is cyclic by definition; again, see the previous section.)

The corner-finding algorithm described below works fairly well in practice, but you will probably need to adjust the parameters it uses. Finding good corners is perhaps the most important part of the entire fitting algorithm: missing a corner usually leads to a sharp point in the original image being rounded off to almost nothing; finding an extraneous corner usually leads to an extremely ugly blob.

Here is Limn's basic strategy for guessing if a given point @math{p} is a corner: compute the total displacement (in both @math{x} and @math{y}) for some number @math{n} of points before @math{p}; do the same for @math{n} points after @math{p}; find the angle @math{a} between those two vectors; if that angle is less than some threshold, @math{p} is a corner.

- The number @math{n} of points to consider is 4 by default; you can specify a different number with the `` `-corner-surround' `` option. If the resolution of the input font is not 300dpi, `` `-corner-surround' `` should almost certainly be changed proportionately. The threshold is 100 degrees by default;

you can change this with the `` `-corner-threshold' `` option. You can see the angles at the chosen corners via `` `-log' ``.

However, when Limn finds a point *p* whose angle is below `` `corner-threshold' ``, it won't necessarily take *p* as the corner. Instead, it continues looking for another`` `corner-surround' `` points; if it finds another point @math{q} whose angle is less than that of @math{p}, *q* will become the corner. (And then Limn looks for another `` `corner-surround' `` points beyond *q*, and so on.)

This continued searching prevents having two corners near each other, which is usually wrong, if the angles at the two would-be corners are approximately the same. On the other hand, sometimes there are extremely sharp turns in the outline within `` `corner-surround' `` pixels; in that case, one does want nearby corners after all.

So Limn has one more option, `` `-corner-always-threshold' ``. If the angle at a point is below this value (60 degrees by default), then that point is considered a corner, regardless of how close it is to other corners. The search for another corner within `` `corner-surround' `` pixels continues, however.

Removing knees

For each curve in the curve list determined by the corners on the pixel outline (see the previous section), Limn next removes **knees**---points on the inside of the outline that form a "right angle" with its predecessor and successor. That is, either (1) its predecessor differs only in @math{x}, and its successor only in @math{y}; or (2) its predecessor differs only in @math{y}, and its successor only in @math{x}.

It is hard to describe in words, but here is a picture:

```
**
 X*
  *
```

The point `` `X' `` is a knee, if we're moving in a clockwise direction.

Such a "right angle" point can be on either the inside or the outside of the outline. Points on the inside do nothing useful, they just slow things down and, more importantly, make the curve being fit less accurate. So we remove them. But points on the outside help to define the shape of the curve, so we keep those. (For example, if `` `X' `` was moved up one diagonally, we certainly want it as a part of the curve.)

Although we haven't found a case where removing knees produces an inferior result, there's no theory about it always helping. Also, you may just be curious what difference it makes (as we were when we programmed the operation). So Limn provides an option `` `-keep-knees' ``; if you specify it, Limn simply skips this step.

Filtering curves

After generating the final pixel coordinates for each curve (see the previous sections), Limn next **filters** the curves to smooth them. Before this step, all the coordinates are on integer boundaries, which makes the curves rather bumpy and difficult to fit well.

To filter a point @math{p}, Limn does the following:

1. Computes the sum of the distances of @math{n} neighbors (points before and after @math{p}) to @math{p}. These neighbors are always taken from the original curve, since we don't want a newly filtered point to affect subsequent points as we continue along the curve; that leads to strange results.
2. Multiplies that sum by a weight, and adds the result to @math{p}. The weight is one-third by default; you can change this with the `` `-filter-percent' `` option, which takes an integer between zero and 100.

Repeatedly filtering a curve leads to even more smoothing, at the expense of fidelity to the original. By default, Limn filters each curve 4 times; you can change this with the `` `-filter-iterations' `` option.

If the curve has less than five points, filtering is omitted altogether, since such a short curve tends to collapse down to a single point.

The most important filtering parameter is the number @math{n} of surrounding points which are used to produce the new point. Limn has two different possibilities for this, to keep features from disappearing in the original curve. Let's call these possibilities *n* and *alt_n*; typically *alt_n* is smaller than *n*. Limn computes the total distance along the curve both coming into and going out of the point @math{p} for both *n* and *alt_n* surrounding points. Then it computes the angles between the in and out vectors for both. If those two angles differ by more than some threshold (10 degrees by default; you can change it with the `` `-filter-epsilon' `` option), then Limn uses *alt_n* to compute the new point; otherwise, it uses *n*.

Geometrically, this means that if using *n* points would result in a much different new point than using *alt_n*, use the latter, smaller number, thus (hopefully) distorting the curve less.

Limn uses 2 for *n* and 1 for *alt_n* by default. You can use the options `` `-filter-surround' `` and `` `-filter-alternative-surround' `` to change them. If the resolution of the input font is not 300dpi, you should scale them proportionately. (For a 1200dpi font, we've had good results with `` `-filter-surround=12' `` and `` `filter-alternative-surround= 6' ``.)

Fitting the bitmap curve

The steps in the previous sections are preliminary to the main fitting process. But once we have the final coordinates for each (bitmap) curve, we can proceed to fit it with some kind of continuous (mathematical) function: Limn uses both straight lines (polynomials of degree 1) and Bezier splines (degree 3).

To begin with, to use a spline the curve must have at least four points. If it has fewer, we simply use the line going through its first and last points. (There is no point in doing a fancy "best fit" for this case, since the original curve is so short.)

Otherwise, if the curve has four or more points, we try to fit it with a (piece of a) Bezier cubic spline. This spline is represented as a starting point, an ending point, and two "control points". Limn uses the endpoints of the curve as the endpoints of the spline, and adjusts the control points to try to match the curve.

A complete description of the geometric and mathematical properties of Bezier cubics is beyond the scope of this document. See a computer graphics textbook for the details.

We will use the terms "splines", "cubics", "Bezier splines", "cubic splines", and so on interchangeably, as is common practice. (Although Bezier splines are not the only kind of cubic splines, they are the only kind we use.)

The sections below describe the spline-fitting process in more detail.

Initializing @math{t}

Limn must have some way to relate the discrete curve made from the original bitmap to the continuous spline being fitted to that curve. This is done by associating another number, traditionally called @math{t}, with each point on the curve.

Imagine moving along the spline through the points on the curve. Then @math{t} for a point @math{p} corresponds to how far along the spline you have traveled to get to @math{p}. In practice, of course, the spline does not perfectly fit all the points, and so Limn adjusts the @math{t} values to improve the fit (see section Reparameterization). (It also adjusts the spline itself, as mentioned above.)

Limn initializes the @math{t} value for each point on the curve using a method called **chord-length parameterization**. The details of how this works do not affect how you use the program, so we will omit them here. (See the Plass & Stone article cited in `` `limn/README' `` if you're curious about them.)

Finding tangents

As mentioned above, Limn moves the control points on the spline to optimally fit the bitmap. But it cannot just move them arbitrarily, because it must make sure that the spline fitting one part of the bitmap blends smoothly with those fit to adjacent parts.

Technically, this smooth blending is called **continuity**, and it comes in degrees. Limn is concerned with the first two degrees: zero- and first-order. Zero-order continuity between two curves simply means the curves are connected; first-order geometric (G1) continuity means the tangents to each curve at the point of connection have the same direction. (There are other kinds of continuity besides "geometric", but they are not important for our purposes.)

Informally, this means that the final shape will not abruptly turn at the point where two splines meet. (Any computer graphics textbook will discuss the properties of tangents, continuity, and splines, if you're unfamiliar with the subject.)

To achieve G1 continuity, Limn puts the first control point of a spline on a line going in the direction of the tangent to the start of the spline; and it puts the second control point on a line in the direction of the tangent to the end of the spline. (It would be going far afield to prove that this together with the properties of Bezier splines imply G1 continuity, but they do. See Schneider's thesis referenced in`` `limn/README' `` for a complete mathematical treatment.)

For the purposes of using Limn, the important thing is that Limn must compute the tangents to the spline at the beginning and end, and must do so accurately in order to achieve a good fit to the bitmap. Since Limn has available only samples (i.e., the pixel coordinates) of the curve being fit, it cannot compute the true tangent. Instead, it must approximate the tangent by looking at some number of coordinates on either side of a point. By default, the number is 3, but you can specify a different number with the `` `-tangent-surround' `` option. If the resolution of the input font is different than 300dpi, or if the outline Limn fits to the bitmap seems off, you will want to scale it proportionately.

Finding the spline

At last, after all the preprocessing steps described in the previous sections, we can actually fit a spline to the bitmap. Subject to the tangent constraints (see the previous section), Limn finds the spline which minimizes the **error**---the overall distance to the pixel coordinates.

More precisely, Limn uses a **least-squares error metric** to measure the "goodness" of the fit. This metric minimizes the sum of the squares of the distance between each point on the bitmap curve and its corresponding point on the fitted spline. (It is appropriate to square the distance because it is equally bad for the fitted spline to diverge from the curve in a positive or negative direction.)

The correspondence between the fitted spline and the bitmap curve is defined by the @math{t} value that is associated with each point (see section Initializing @math{t}).

For a given set of @math{t} values and given endpoints on the spline, the control points which minimize the least-squares metric are unique. The formula which determines them is derived in Schneider's thesis (see the reference in `` `limn/README' ``); Limn implements that formula.

Once we have the control points, we can ask how well the resulting spline actually does fit the bitmap curve. Limn can do two things to improve the fit: change the @math{t} values (reparameterization); or break the curve into two pieces and then try to fit each piece separately (subdivision).

The following two sections describe these operations in more detail.

Reparameterization

Reparameterization changes the @math{t} value for each point @math{p} on the bitmap curve, thus changing the place on the spline which corresponds to @math{p}. Given these new @math{t} values, Limn will then fit a new spline (see the previous section) to the bitmap, one which presumably matches it more closely.

Reparameterization is almost always a win. Only if the initial fit (see section Initializing @math{t}) was truly terrible will reparameterization be a waste of time, and be omitted in favor of immediate subdivision (see the next section).

Limn sets the default threshold for not reparameterizing to be 30 "square pixels" (this number is compared to the least-squares error; see the previous section). This is usually only exceeded in cases such as that of an outline of `` `o' ``, where one spline cannot possibly fit the entire more-or-less oval outline. You can change the threshold with the option `` `-reparameterize-threshold' ``.

If the error is less than `` `reparameterize-threshold' ``, Limn reparameterizes and refits the curve until the difference in the error from the last iteration is less than some percentage (10 by default; you can change this with the option `` `-reparameterize-improve' ``).

After Limn has given up reparameterization (either because the initial fit was worse than `` `reparameterize-threshold' ``, or because the error did not change by more than `` `reparameterize-improve' ``), the final error is compared to another threshold, 2.0 by default. (You can specify this with the option `` `-error-threshold' ``.) If the error is larger, Limn subdivides the bitmap curve (see the next section) and fits each piece separately. Otherwise, Limn saves the fitted spline and goes on to the next piece of the pixel outline.

Subdivision

When Limn cannot fit a bitmap curve within the `` `error-threshold' `` (see the previous section), it must subdivide the curve into two pieces and fit each independently, applying the fitting algorithm recursively.

As a strategy to improve the fit, subdivision is inferior to reparameterization, because it increases the number of splines in the character definition. This increases the memory required to store the character, and also the time to render it. However, subdivision is unavoidable in some circumstances: for example, the outlines on an `` `o' `` cannot be fit by a single spline.

For the initial guess of the point at which to subdivide, Limn chooses the point of worst error--the point where the fitted spline is farthest from the bitmap curve. Although this is usually a good choice, minimizing the chance that further subdivision will be necessary, occasionally it is not the best: in order to preserve straight lines, it is better to subdivide at the point where a straight becomes a curve if that point is close to the worst point. For example, this happens where a serif joins the stem.

Limn has three options to control this process:

1. `` `-subdivide-search `` *`percent`* `'` specifies how far away from the worst point to search for a better subdivision point, as a percentage of the total number of points in the curve; the default is 10. If you find Limn missing a join as a subdivision point, resulting in a straight line becoming a curve, you probably need to increase this.
2. `` `-subdivide-threshold `` *`real`* `'`: if the distance between a point @math{p} (within the search range) and a straight line is less than this, subdivide at @math{p}; default is .03 pixels.

3. ``-subdivide-surround *unsigned*'`: when calculating the linearity of the curve surrounding a potential subdivision, use this many points; default is 4.

Because fitting a shorter curve is easier, this process will always terminate. (Eventually the curve will be short enough to fit with a straight line (see section Fitting the bitmap curve), if nothing else.)

Changing splines to lines

Upon accepting a fitted spline (see the previous sections), Limn checks if a straight line would fit the curve as well. If so, that is preferable, since it is much faster to render straight lines than cubic splines.

More precisely, after fitting a cubic spline to a particular (segment of a) curve, Limn finds the straight line between the spline's endpoints, and computes the average distance (see section Finding the spline) between the line and the curve. If the result is less than some threshold, 1 by default, then the spline is provisionally (see the next section) changed to a line.

You can change the theshold with the `` `-line-threshold' `` option.

Changing lines to splines

Once an entire curve (i.e., the bitmap outline between two corners; see section Finding corners) has been fit, Limn checks for straight lines that are adjacent to splines. Unless such lines fit the bitmap *extremely* well, they must be changed to splines.

The reason is that the point at which the line and spline meet will be a visible "bump" in the typeset character unless the two blend smoothly. Where two splines meet, the continuity is guaranteed from the way we constructed the splines (see section Finding tangents). But where a line and spline meet, nothing makes the join smooth.

For example, if the outline of a `` `o' `` has been subdivided many times (as typically happens), a spline may end up fitting just a few pixels--so few that a line would fit just as well. The actions described in the previous section will therefore change the spline to a line. But since the adjacent parts of the `` `o' `` are being fit with curves, that line will result in a noticeable flat spot in the final output. So we must change it back to a spline.

We want this reversion to be more likely for short curves than long curves, since short curves are more likely to be the result of a small piece of a curved shape. So Limn divides the total distance between the fitted line and the bitmap curve by the square of the curve length, and compares the result to a threshold, .01 by default. You can change this with the `` `-line-reversion-threshold' `` option.

Aligning endpoints

After fitting a mathematical outline of splines and lines to a pixel outline (see section Finding pixel outlines), Limn aligns the endpoints on the fitted outline. This involves simply checking each spline to see if its starting point and ending point (in either axis) are "close enough" to each other. If they are, then they are made equal to their average.

This is useful because even a slight offset of the endpoints can be produce a noticeable result, especially for straight lines and corners.

By default, "close enough" is half a pixel. You can change this with the `` `-align-threshold' `` option.

Displaying fitting online

While experimenting with the various fitting options listed in the preceding sections, you may find it useful to see the results of the fitting online. Limn can display the filtered (see section Filtering curves) bitmap and the fitted outline online if it is run under the X window system and you specify `` `-do-display' ``.

Ordinarily, Limn stops at the end of fitting every character for you to hit return, so you have a chance to examine the result. If you just want to get a brief glimpse or something, you can specify `` `-display-continue' ``. Then Limn won't stop.

If you specify `` `-do-display' ``, you must set the environment variable `DISPLAY` to the X server you want Limn to use. For example, in Bourne-compatible shells, you might do:

```
DISPLAY=:0 ; export DISPLAY
```

The output is shown on a grid, in which each square represents several pixels in the input. Corners are shown as filled squares; other pixels are shown as hollow squares.

Limn has several options that change the appearance of the online output:

- `` `-display-grid-size ``*`unsigned`*`'` The number of expanded pixels shown between the grid lines; default is 10.
- `` `-display-pixel-size ``*`unsigned`*`'` The expansion factor; i.e., each input pixel is expanded to a square this many pixels on a side; default is 9.
- `` `-display-rectangle-size ``*`unsigned`*`'` The pixel size; i.e., each pixel shown is a square this many pixels on a side; default is 6. This must be less than the `` `display-pixel-size' ``, so that black pixels don't merge into each other.

You can change the size of the window Limn creates with the `geometry` resource in your `` `.Xdefaults' `` file (see the documentation in the file `` `mit/doc/tutorials/resources.txt' `` in the X distribution if you aren't familiar with X resources). The class name is `Limn`. For example:

```
Limn*geometry: 300x400-0-0
```

makes the window 300 pixels wide, 400 pixels high, and located in the lower right corner of the screen.