Saffron 3.1

Multiple Alignment Zones

January 26, 2007

Alignment Zones

- Saffron 3.0 and earlier versions:
  - Grid fitting for Latin, Arabic, Devanagari, Hebrew, Thai
  - No grid fitting support for CJK

Multiple Alignment Zones

- Saffron 3.1 adds grid fitting support for CJK
  - Supports both outlines and strokes
  - Completely automatic

Comparison

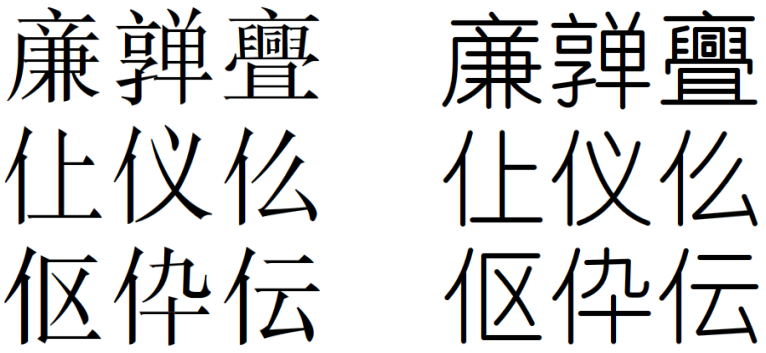
- Standard Alignment Zones
  - Detect alignment zones in a preprocess
  - Use alignment zones to grid fit at runtime
  - Used for Latin, Arabic, Devanagari, Hebrew, Thai
- Multiple Alignment Zones
  - No preprocess
  - Detection and grid fitting are performed on-the-fly at runtime
  - Used only for CJK

Comparison (cont.)

- Standard Alignment Zones
  - Ignores traditional hints embedded in typefaces
  - Requires 64 bits per glyph to store alignment zone data
- Multiple Alignment Zones
  - Ignores traditional hints embedded in typefaces
  - Requires no additional storage (no preprocessing step)
- Small footprint in both cases

Outlines and Strokes

- Different algorithms for outlines and strokes



Outlines and Strokes

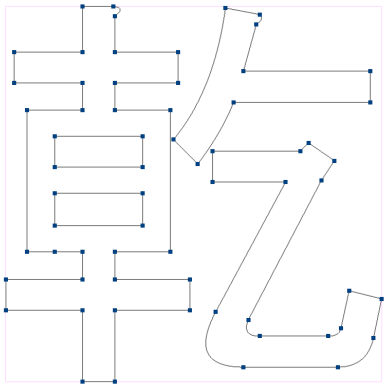
- Algorithm similarities:
  - Detect and align horizontal and vertical features
  - Grid fit horizontal and vertical features independently
  - Interpolate non-aligned features
  - No regularization
- Recall: grid fitting is performed on-the-fly
  - Process includes both feature detection and alignment
  - Keep operations simple for efficiency

Outline Algorithm Overview

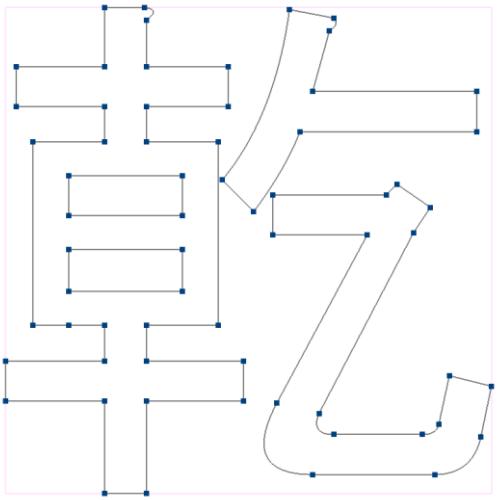
- Strategies:
  - Grid fit horizontal and vertical features independently
  - Grid fit independently in groups called "radicals"
  - Align coordinates to half-integers
  - Preserve original bar and stem widths (with some exceptions)

Outline Algorithm Inputs

- Any outline-based CJK glyph
  - Raw unhinted coordinates



Points and Contours



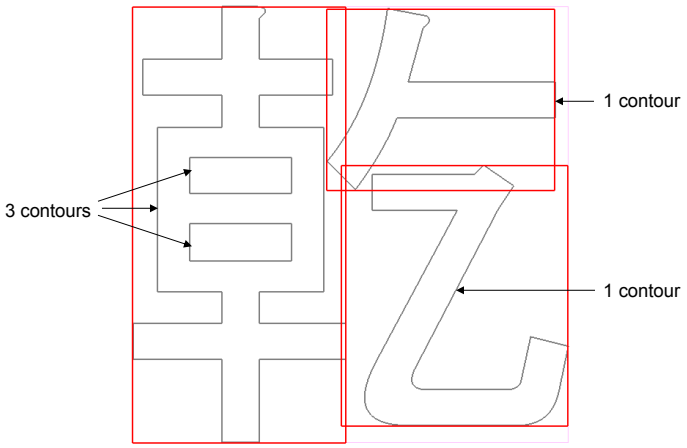
72 points, 5 contours

Outline Algorithm Parts

- Two parts, both performed at runtime:
  - Feature detection: steps 1 through 4
  - Grid fitting: step 5

Step 1: Make Radicals

- Organize contours into groups called "radicals"



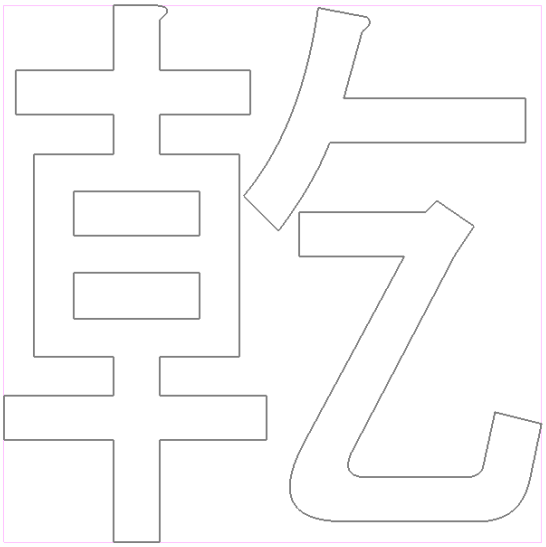
Step 1: Make Radicals

- Organize contours into groups called "radicals"
  - Each radical is a group of related glyph components
  - No connection to Chinese "radicals"
  - Grid fitting is done separately for each radical

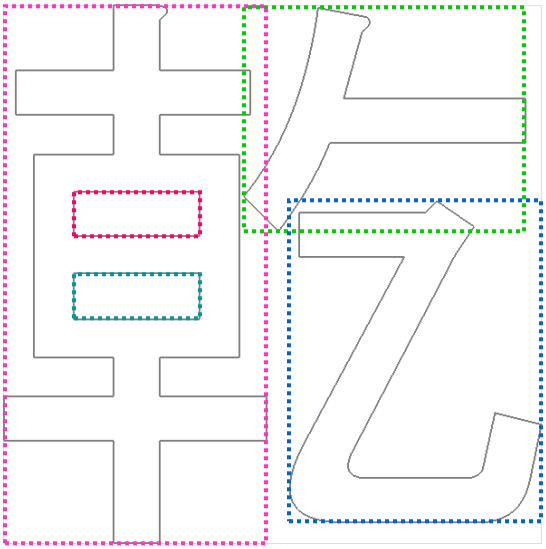
Step 1: Make Radicals

- Definitions:
  - A contour C has parent P if P is the contour with the smallest bounding box that contains C's bounding box. If no such P exists, then C has no parent.
  - Contours with no parents are called root contours
  - All other contours are called internal contours

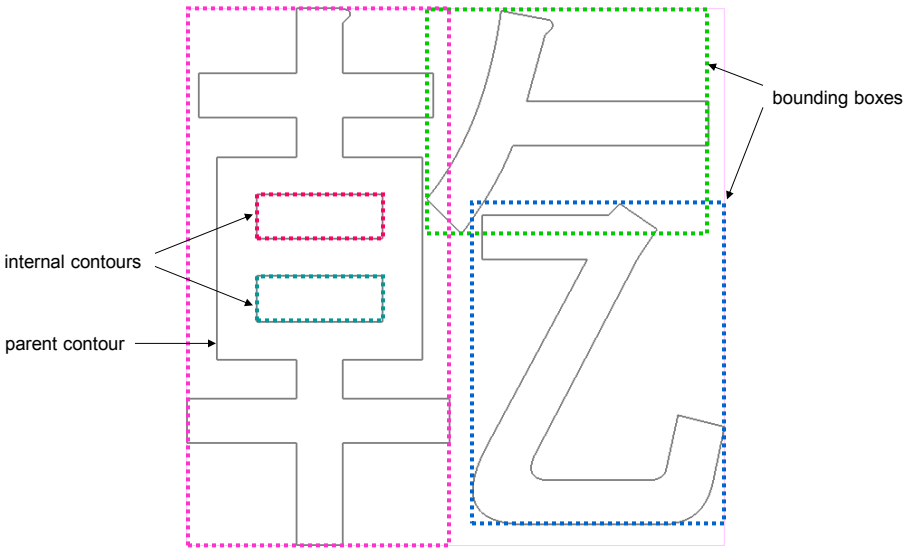
Example



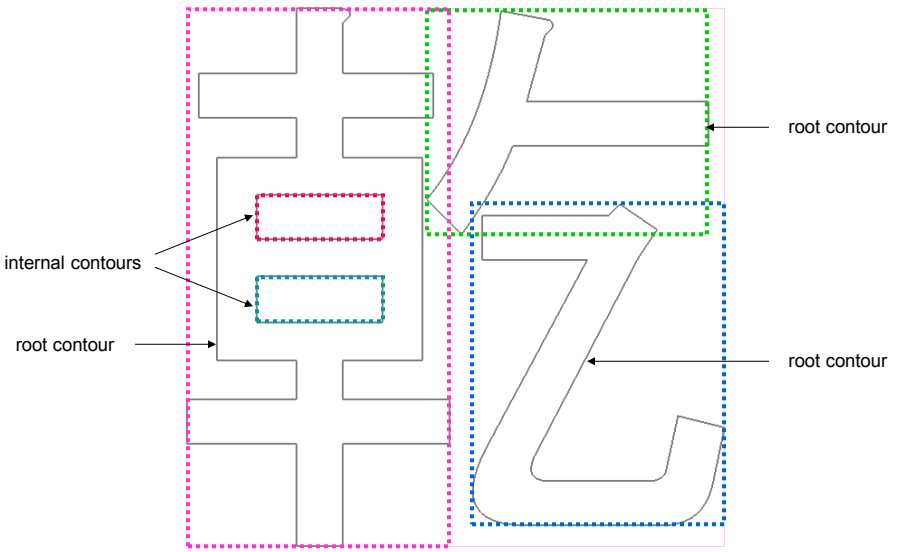
Example



Example



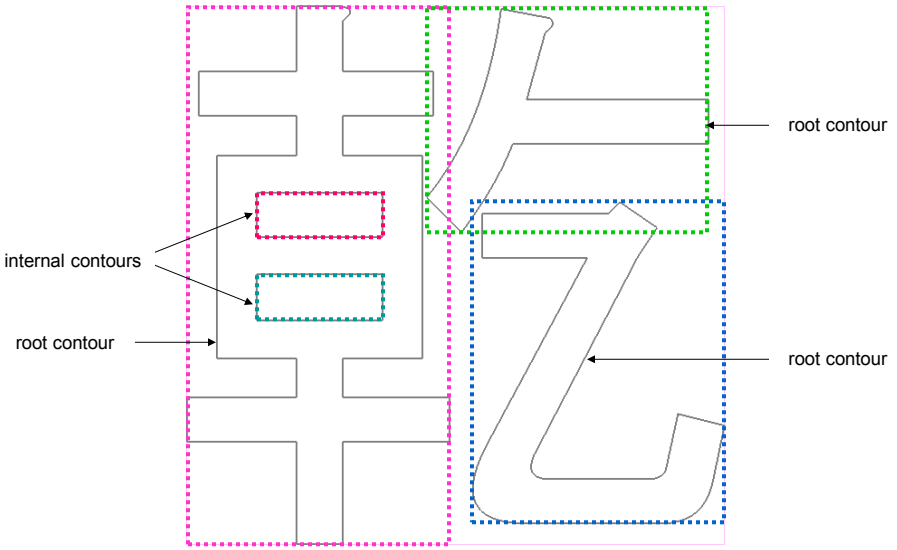
Example



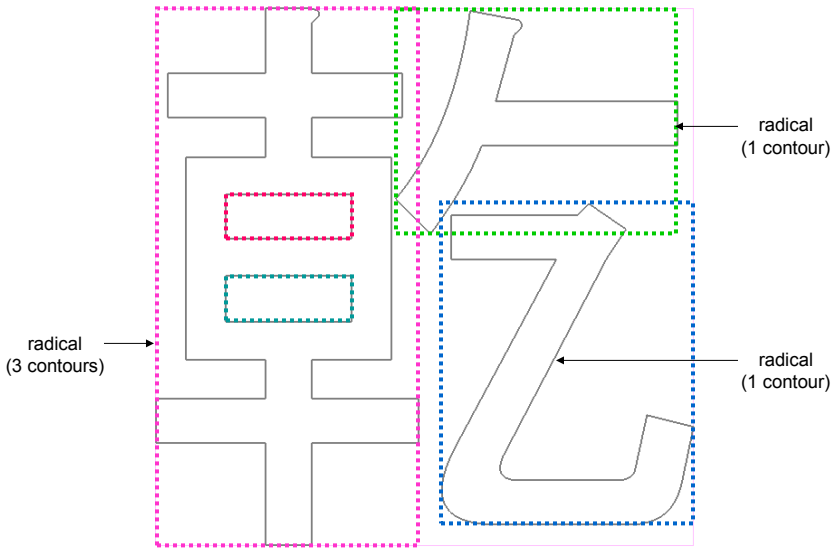
Step 1: Make Radicals

- Definition:
  - A radical is a set of contours: a single root contour R and all internal contours whose root ancestor is R

Example



Example



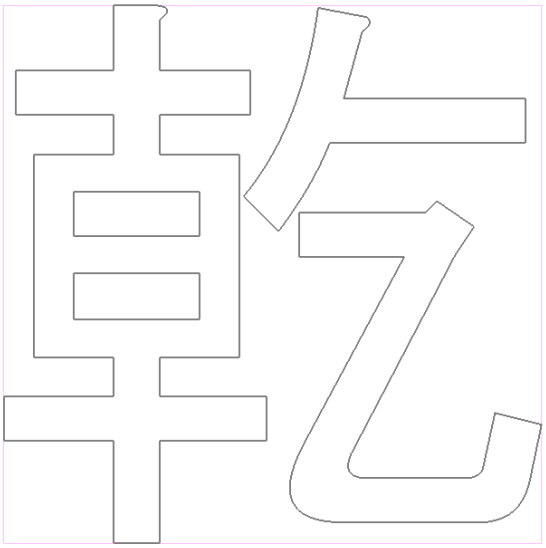
Radical Independence

- Each radical is grid fit independently
  - Allows features (bars, stems) to be detected more accurately
  - More efficient
- Descriptions on following slides are per-radical
  - i.e., apply the following steps for each radical

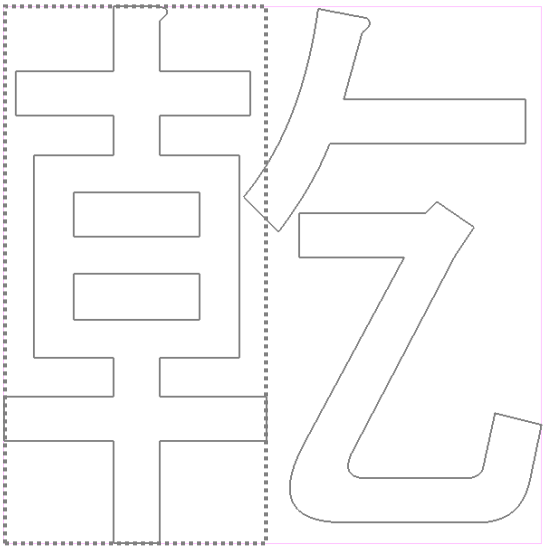
Step 2: Find Simple Segments

- Definitions:
  - A vertical simple segment has two adjacent points with equal x values
  - A horizontal simple segment has two adjacent points with equal y values
- In practice:
  - Consider two values within epsilon to be equal

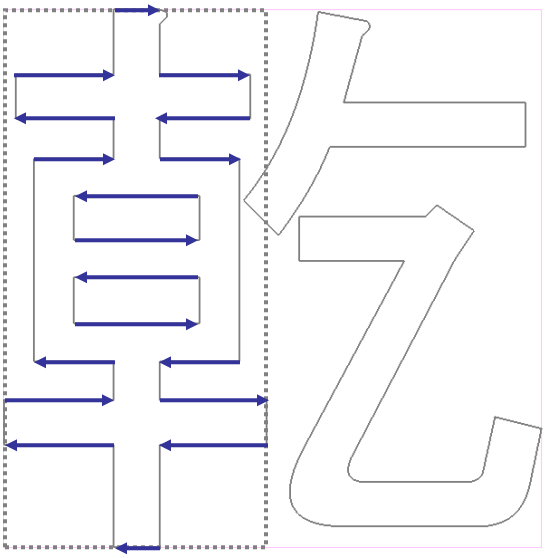
Example



Example



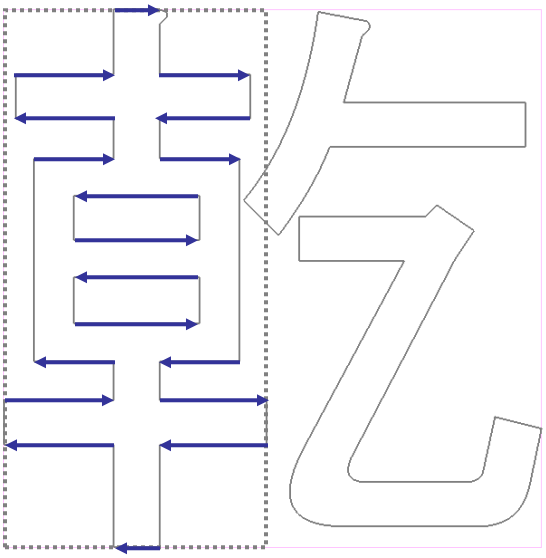
Example



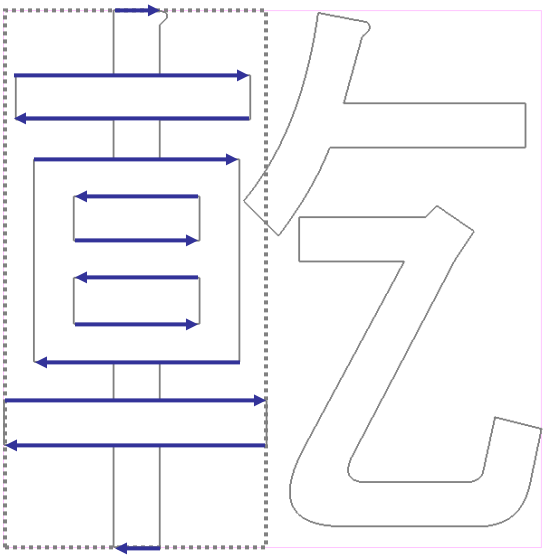
Step 3: Merge Simple Segments

- Rules for merging simple segments:
  - aligned (e.g., same y coordinate for horizontal segments)
  - same orientation (e.g., left-to-right)

Example



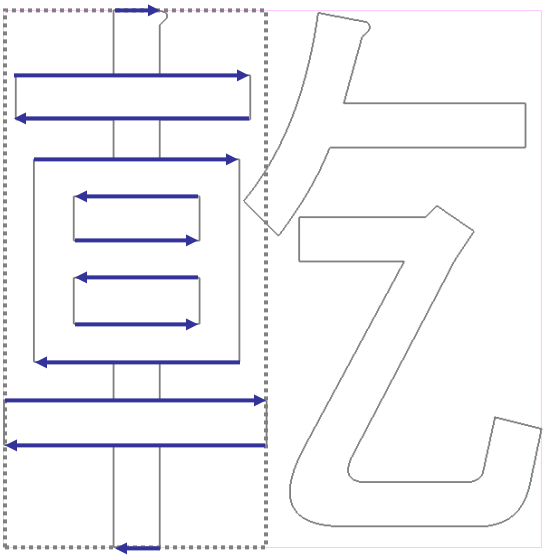
Example



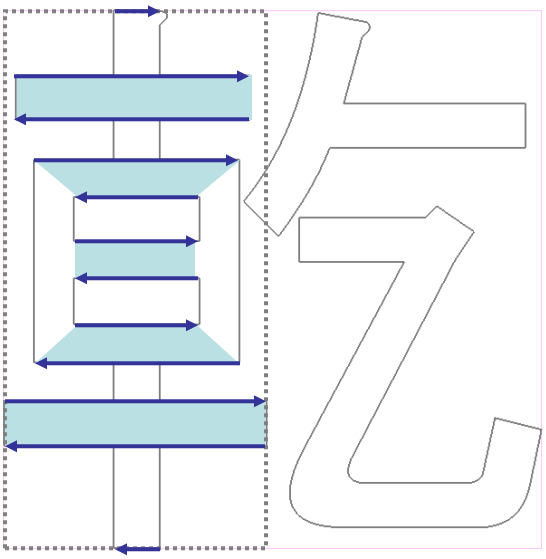
Step 4: Create Pairs

- A segment pair consists of two merged segments
- Requirements:
  - sufficient overlap
  - not too wide
  - segments have opposite orientations
- Multiple pair configurations possible
  - optimize for configuration with thinnest pairs
  - not all segments will be paired

Example

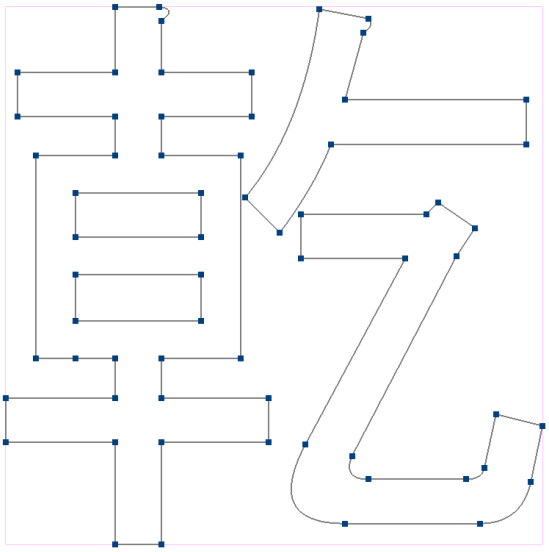


Example

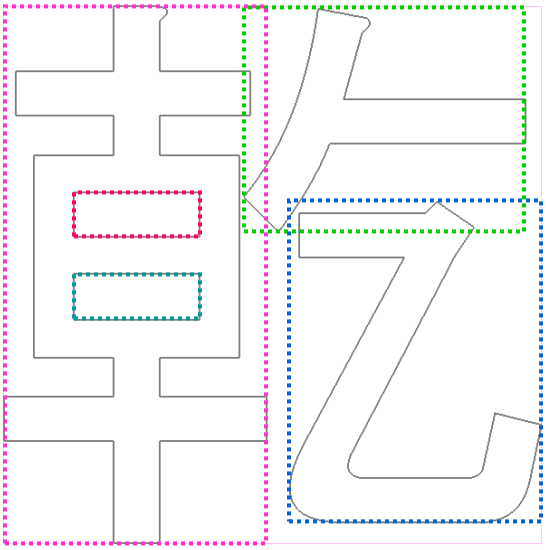


Visual Review of Part 1

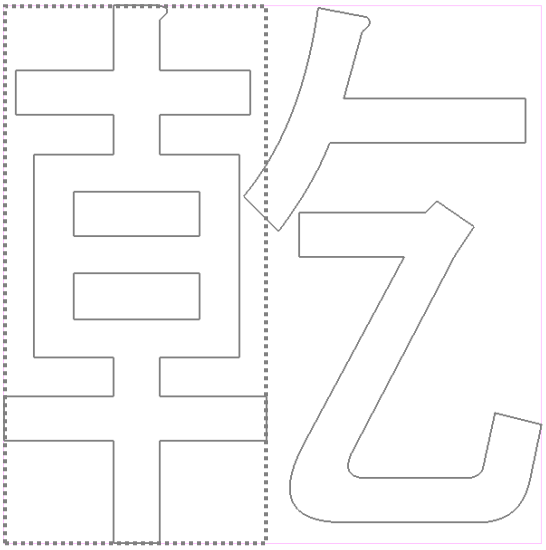
Visual Review of Part 1



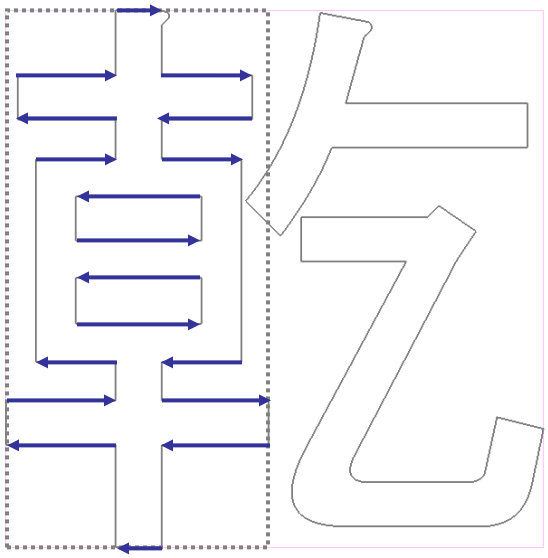
Visual Review of Part 1



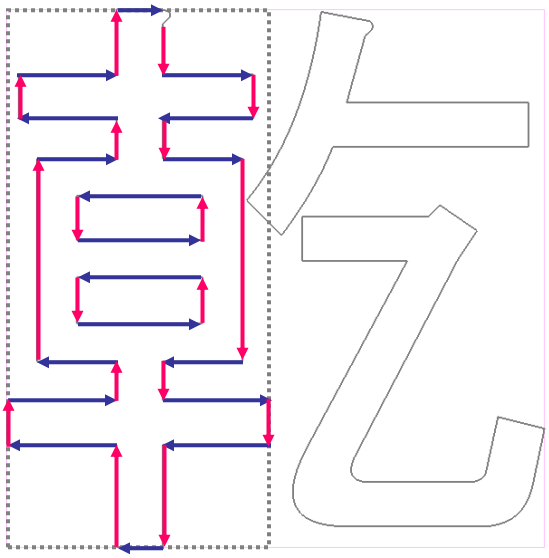
Visual Review of Part 1



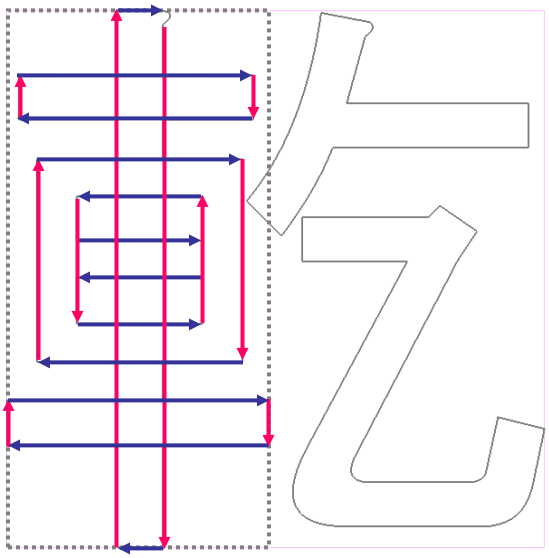
Visual Review of Part 1



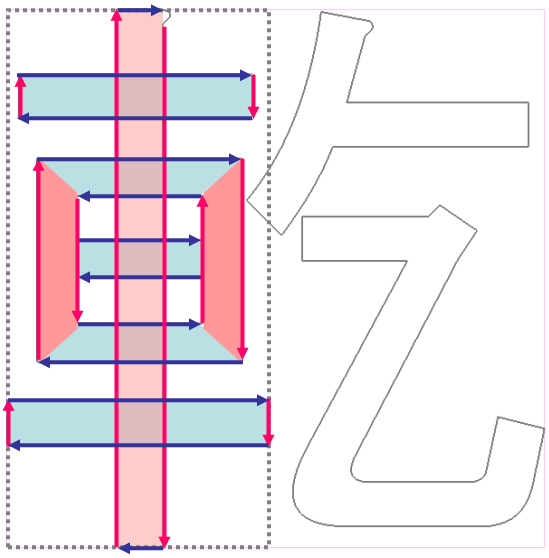
Visual Review of Part 1



Visual Review of Part 1



Visual Review of Part 1



Step 5: Grid Fit

- Perform grid fitting in the following order:
  1. Segment pairs
  2. Unpaired segments
  3. All other points

Segment Pairs

- Rules:
  - Snap bottom segment to the nearest grid point
  - Determine top segment based on adjusted pair width
- Competing goals:
  - Clarity (edge contrast)
  - Consistency (perceptually uniform stroke weights)
- Should we round or preserve pair widths?

Rounding vs. Preserving

Preserve original pair widths
(one edge sharp, one edge fuzzy)

Round pair widths to nearest integer
(both edges sharp)

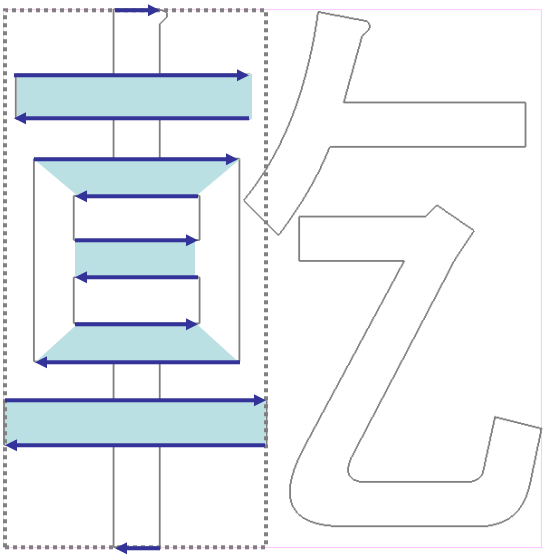
Rounding vs. Preserving

- Observations on rounding:
  - Optimizes clarity
  - Disastrous at small PPEMs (e.g., 14, 20, 24)
  - Acceptable at intermediate PPEMs (e.g., 36, 48, 60)
  - Superior at large PPEMs (about 80 and above)
- Observations on preserving:
  - Optimizes consistency
  - Features become too thin at small PPEMs (e.g., 14, 20, 24)

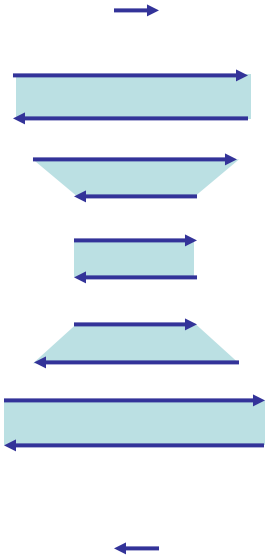
Adjusted Pair Width

- Rounding vs. preserving:
  - Preserve original pair widths at small and intermediate PPEMs
  - Round pair widths at large PPEMs (80 and above)
- Small pair widths:
  - Force pair widths to be at least 0.5 pixels
  - Boost pair widths that lie between 0.5 and 1.0 pixels
  - Boosting is based on PPEM (between 14 and 30 PPEMs)
  - Good balance between clarity and consistency

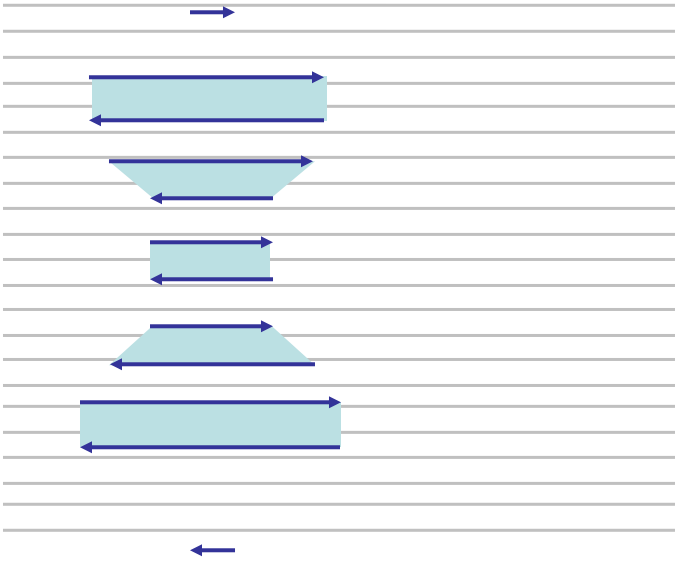
Example



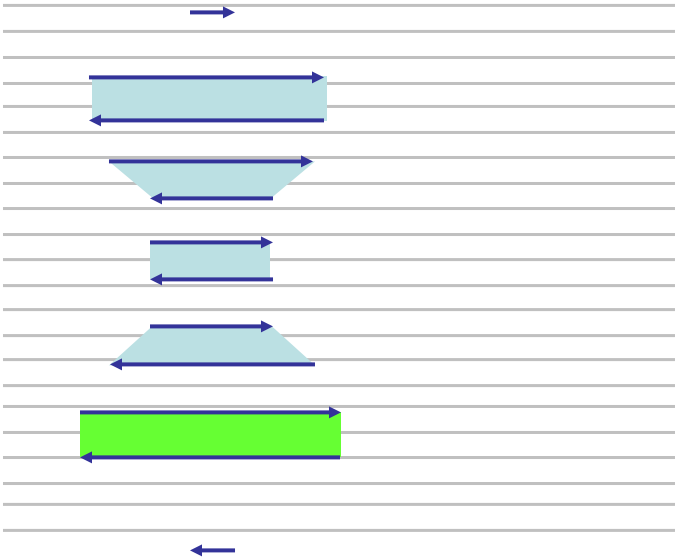
Example



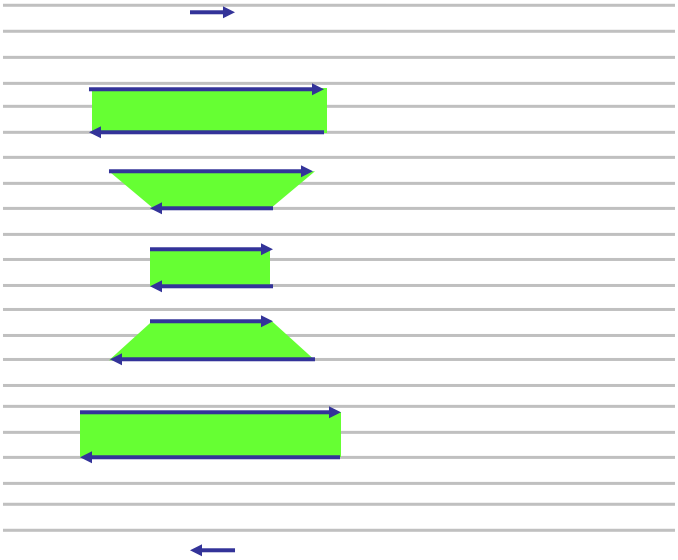
Example



Example



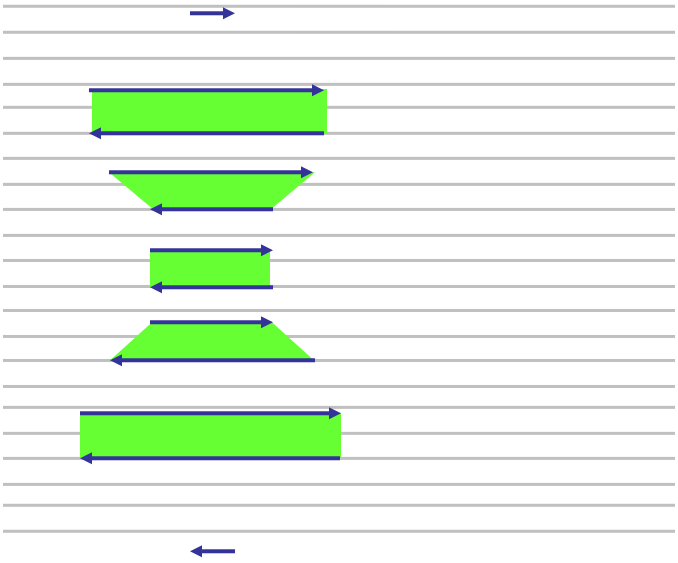
Example



Unpaired Segments

- Snap unpaired segments to the nearest grid point

Example

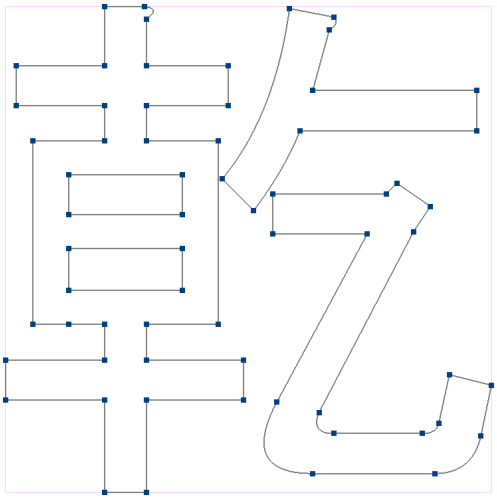


Example

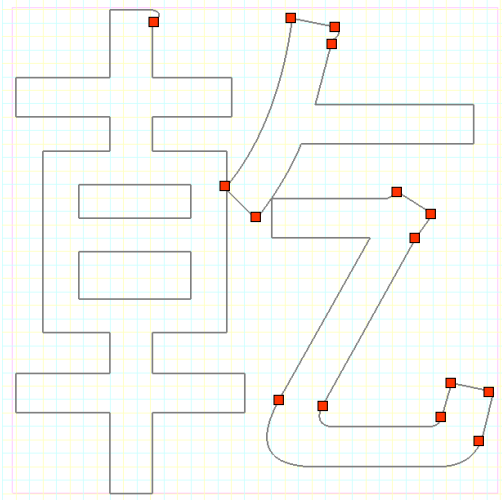
Remaining Points

- Interpolate remaining points using anchors
- Anchor points are:
  - points that have already been grid fit
  - local minima or maxima (i.e., local extrema)

Example



Example



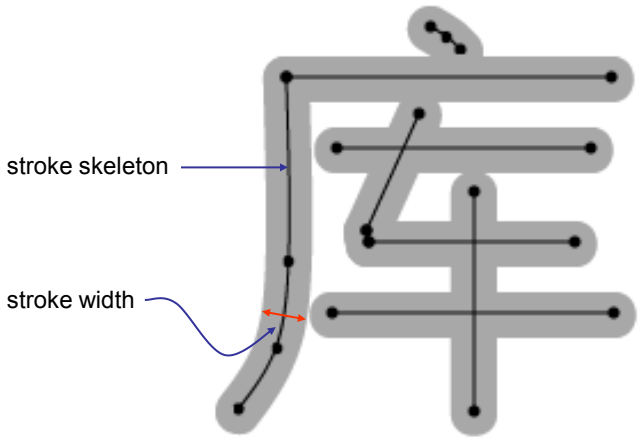
Demos

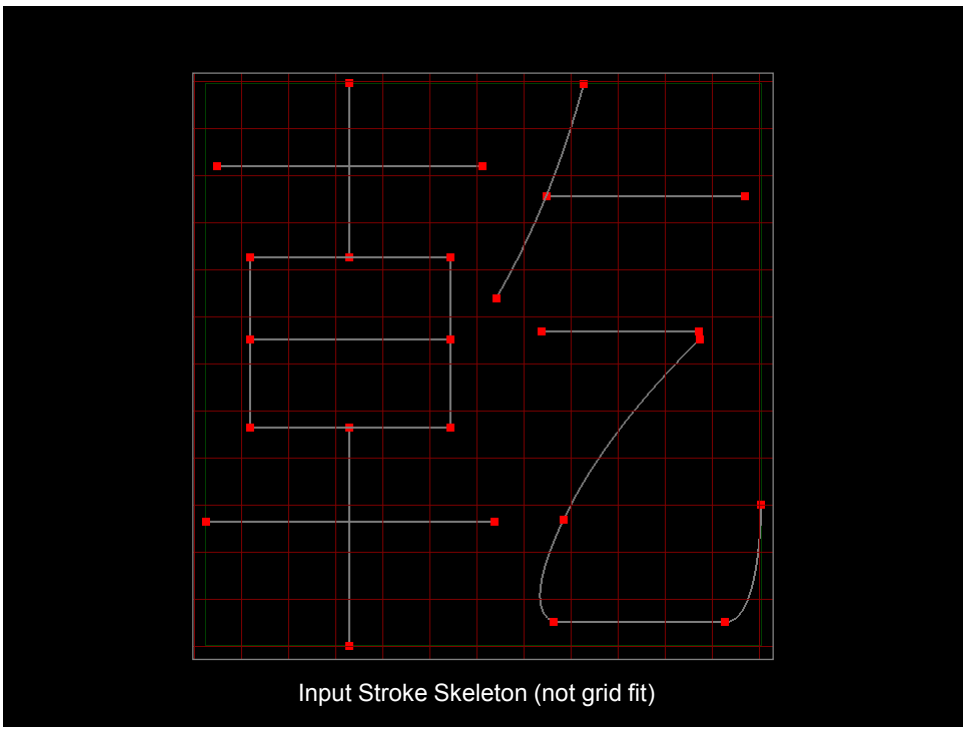
Stroke Algorithm Overview

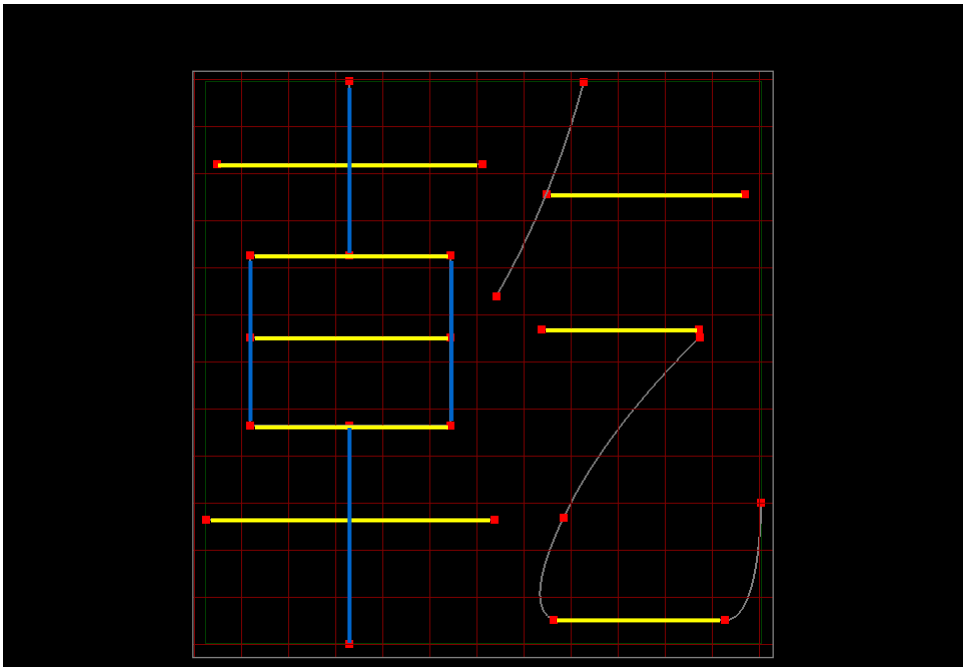
- Strategies:
  - Grid fit horizontal and vertical features independently
  - Round stroke widths to the nearest integer (at least 1 pixel)
  - Maximize clarity by placing features at least 2 pixels apart
  - Do not group into radicals

Stroke Skeletons

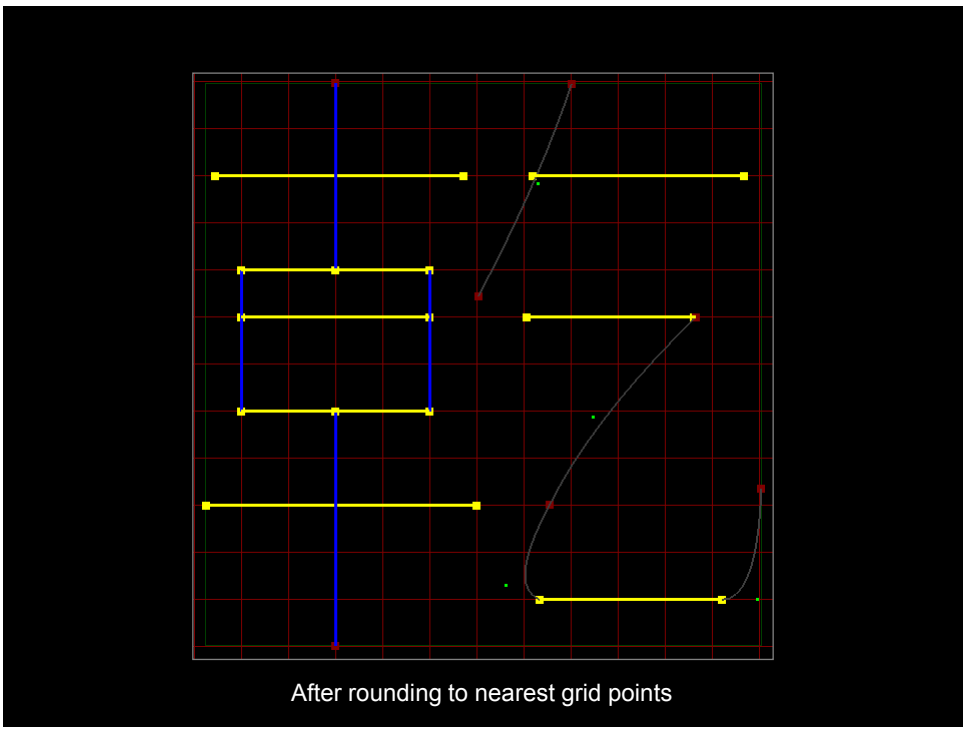
- Terminology: stroke skeletons and stroke width



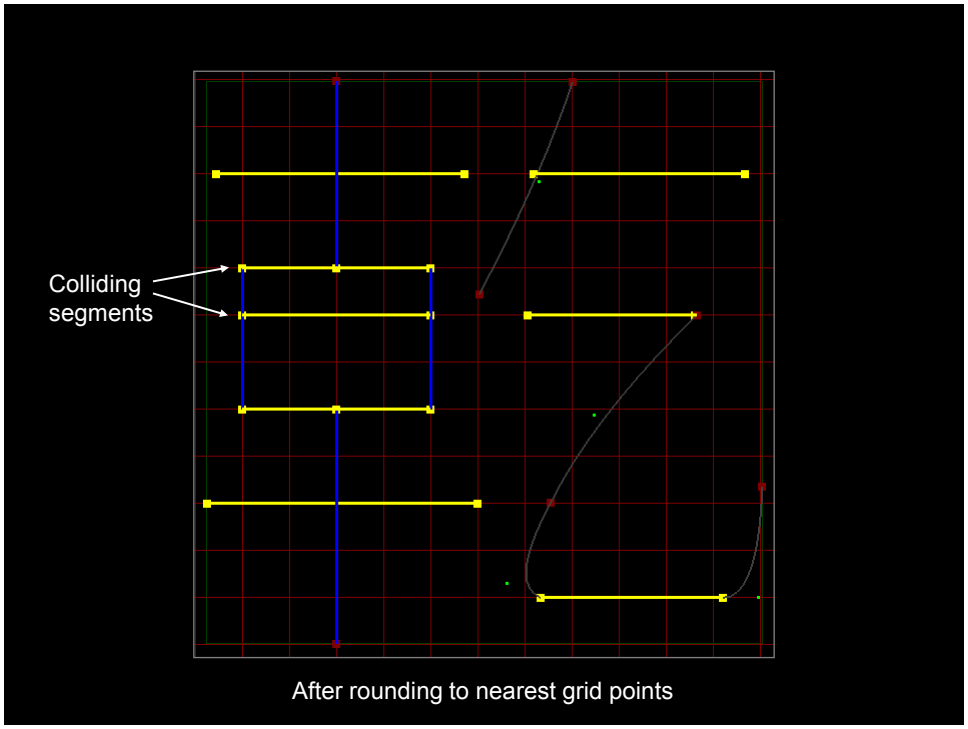




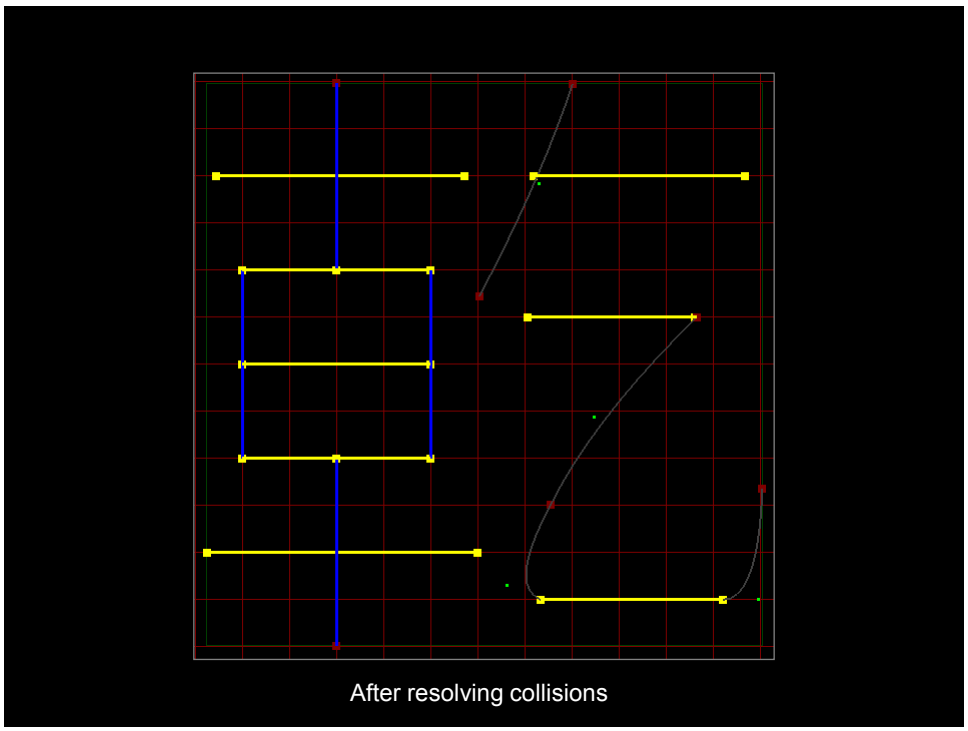
Horizontal and Vertical Segments



After rounding to nearest grid points



After rounding to nearest grid points



After resolving collisions

Rounded Stroke Width

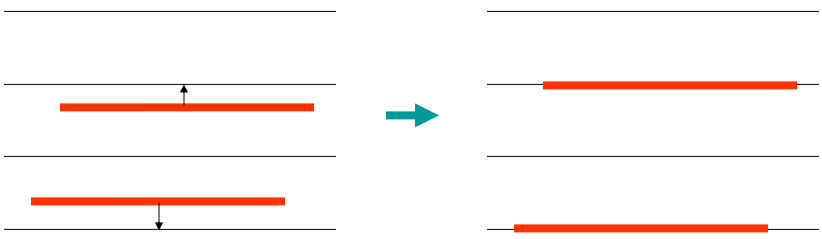
- Stroke width specified as fraction of em box
  - Typically either 3% or 5%
- Round the stroke width to the nearest integer
  - Must be at least 1 pixel
  - If even, align skeletons to half-integers (0.5, 1.5, 2.5, ...)
  - If odd, align skeletons to integers (0, 1, 2, ...)

Step 1: Find Segments

- Definitions:
  - A vertical segment has two adjacent points with equal x values
  - A horizontal segment has two adjacent points with equal y values
- In practice:
  - Consider two values within epsilon to be equal

Step 2: Grid Fit Segments

- Round each segment to the nearest grid point

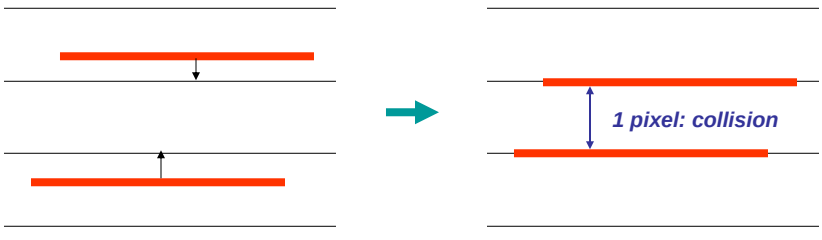


Before rounding

After rounding

Step 3: Avoid Collisions

- Segments collide if they are 1 pixel apart
- Colliding segments appear as one thick segment!



Before rounding

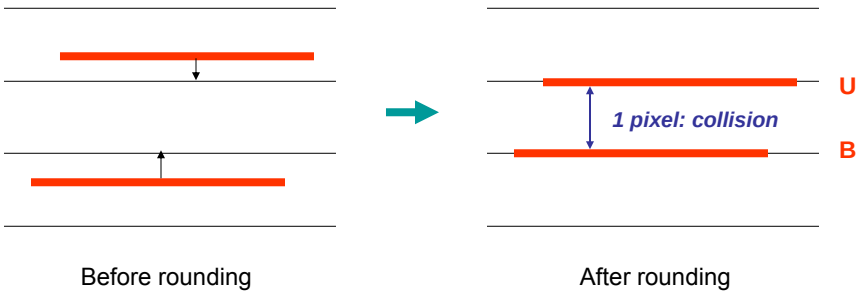
After rounding

Step 3: Avoid Collisions

- Try to space segments at least 2 pixels apart
- Collisions are resolved using three rules:
  1. Try to round down the bottom segment(s)
  2. Try to round up the upper segment
  3. Round bottom and upper segments to the same grid point

Collision Resolution Details

- Consider two segments B and U that collide
  - B is the bottom segment
  - U is the upper segment



Before rounding

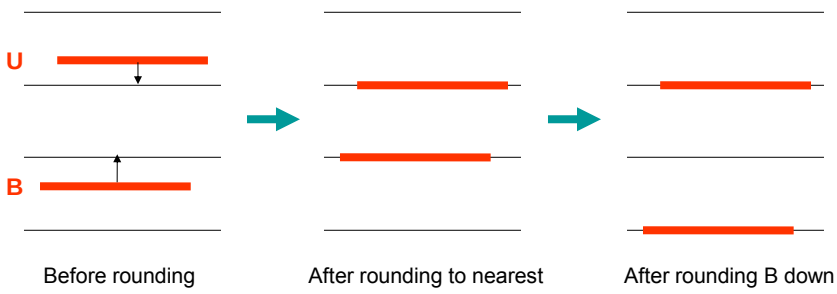
After rounding

Collision Resolution Step 1

- Try to round down B
  - B must currently be rounded up
  - Rounding down must not cause B to collide with any overlapping segments below it
  - Recursively round down all overlapping segments below B if necessary

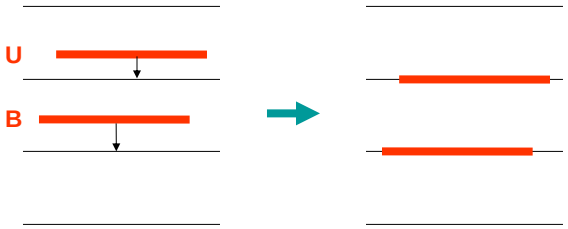
Example #1

- Situation:
  - B is rounded up
  - B has no overlapping segments below it
  - Therefore, round down B



Example #2

- Situation:
  - B is already rounded down
  - Therefore, cannot resolve collisions by rounding down B



Before rounding

After rounding to nearest

Example #3

- Situation:
  - B is rounded up
  - B has no overlapping segments below it
  - Therefore, round down B

Before rounding

After rounding to nearest

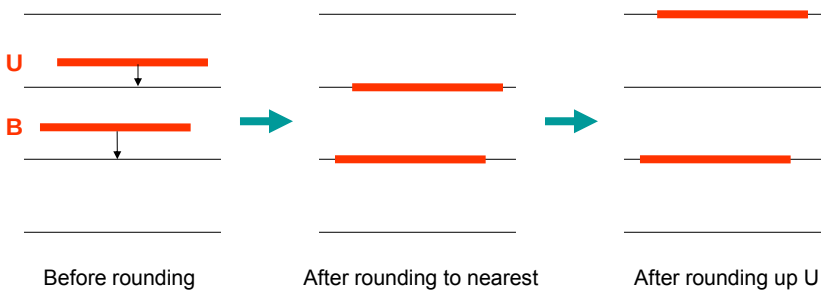
After rounding down B and the overlapping segments below

Collision Resolution Step 2

- If Step 1 fails, round up segment U
  - U must currently be rounded down

Example

- Situation:
  - B cannot be rounded down (i.e., step 1 failed)
  - Therefore, round up U instead

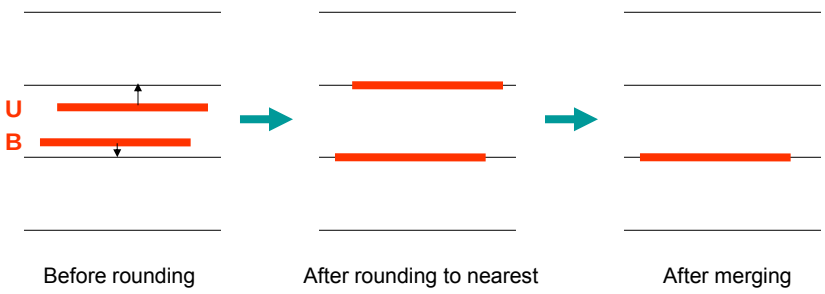


Collision Resolution Step 3

- If Step 2 fails, align segment U with segment B
  - Causes the two segments to be merged
  - U and B cannot be visually distinguished anyways

Example

- Situation:
  - B cannot be rounded down (i.e., step 1 failed)
  - U cannot be rounded up (i.e., step 2 failed)
  - Therefore, merge B and U to the same grid point

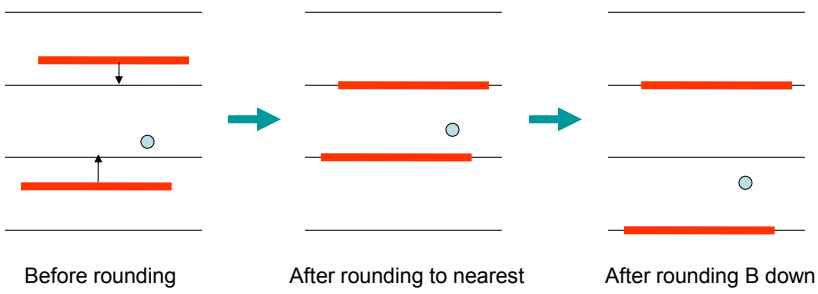


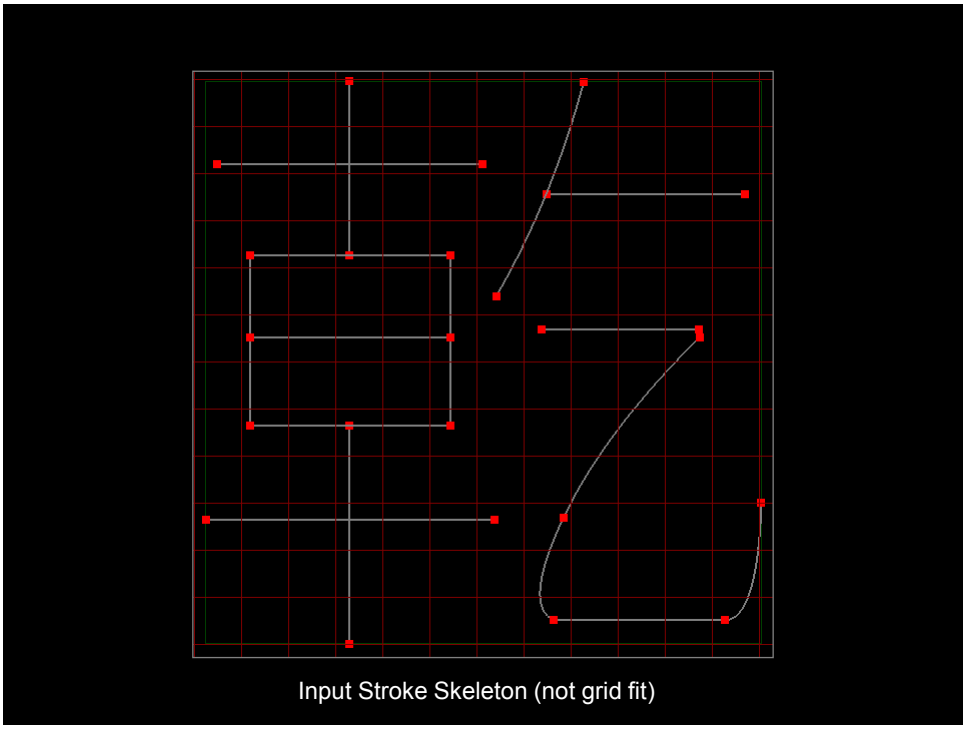
Step 4: Interpolation

- Interpolate non-segment points. For each point:
  - Find nearest segment S1 that lies above the point
  - Find nearest segment S2 that lies below the point
  - Interpolate between S1 and S2 to find new coordinates

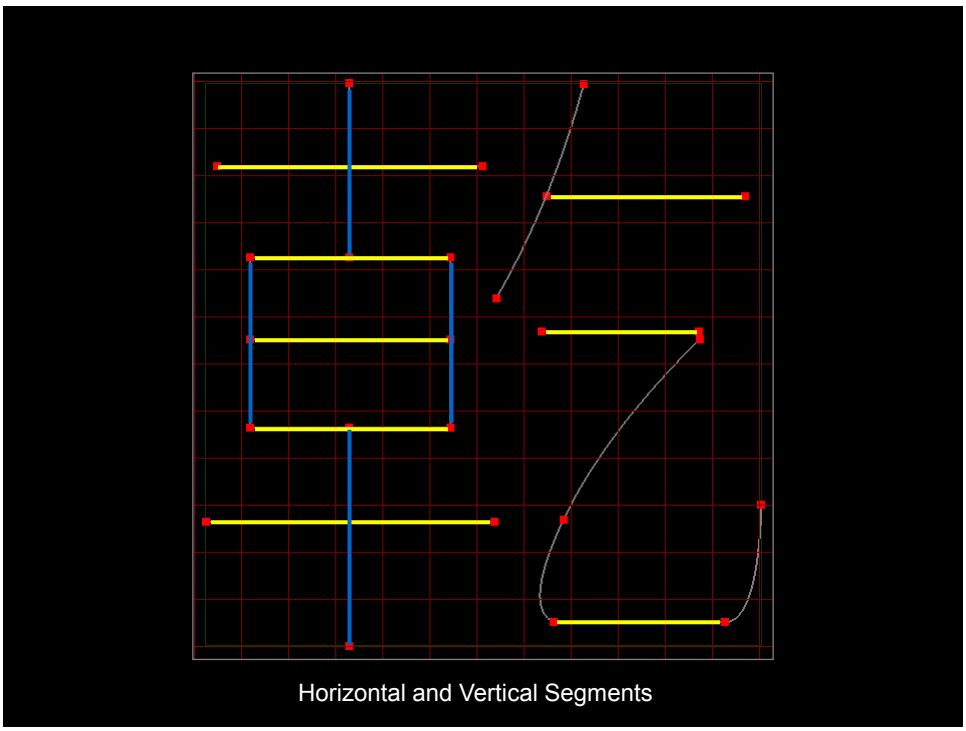
Example

- Situation:
  - Point lies between two segments
  - Final position is determined by interpolation

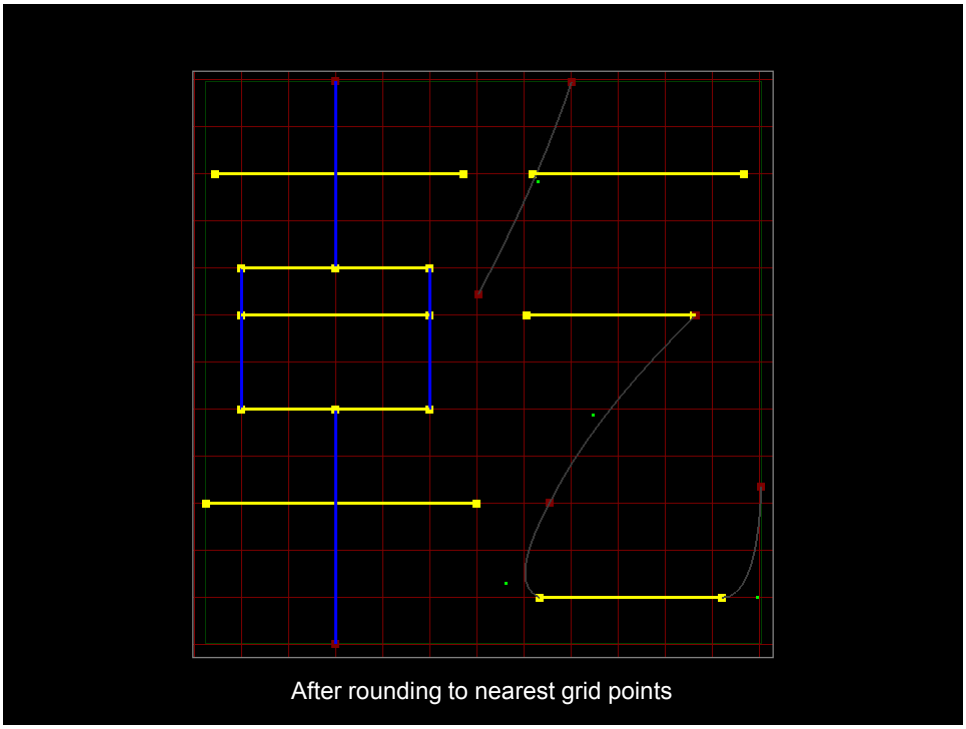




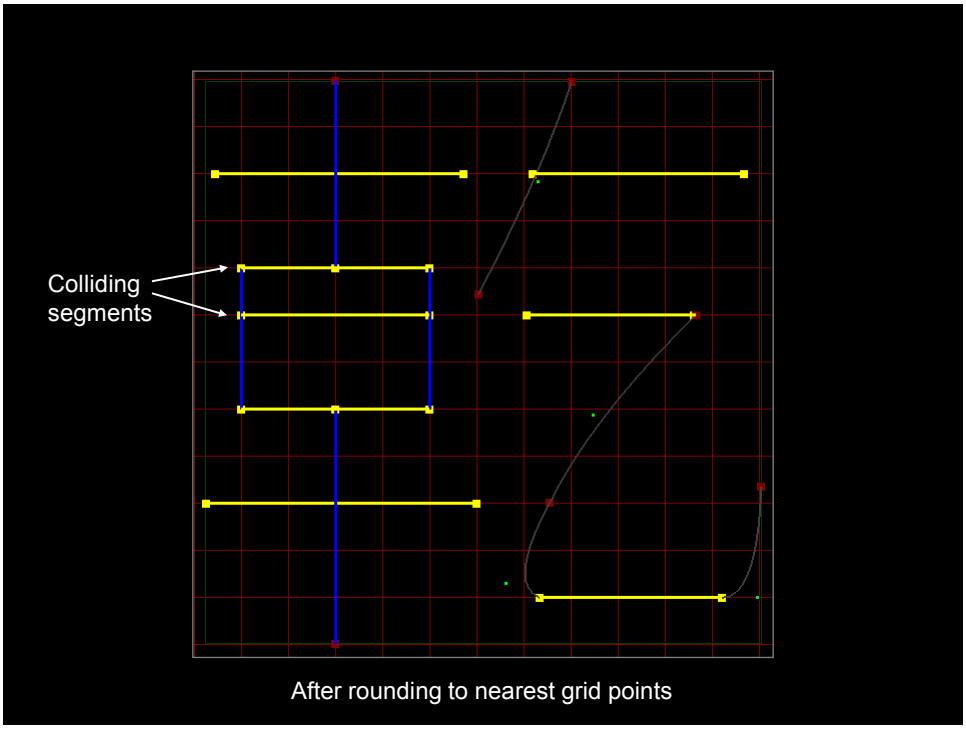
Input Stroke Skeleton (not grid fit)



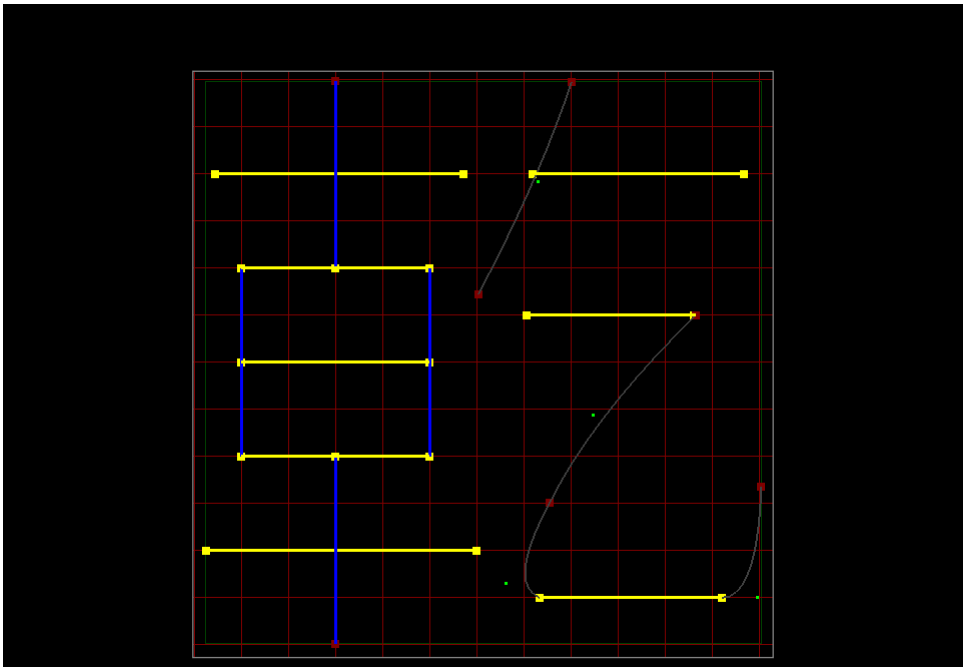
Horizontal and Vertical Segments



After rounding to nearest grid points



After rounding to nearest grid points



After resolving collisions

Demos

Performance

- Measurement system:
  - IBM T60 laptop
  - Intel Core Duo T2500 2.0 GHz
- Results:
  - Outlines: 90,000 – 110,000 glyphs/sec
  - Strokes: 190,000 – 230,000 glyphs/sec
- Fast enough for on-the-fly hinting at render time

Summary

- Multiple alignment zones:
  - Grid fitting support for CJK
  - Supports both outlines and strokes
  - Feature detection and grid fitting on-the-fly at runtime
  - Minimal footprint (does not require font-specific hints)
  - Included in Saffron 3.1 (to be released March 1, 2007)

API Support

- API is not finalized
- Current thoughts:
  - Fully automatic
  - One Boolean in API to turn multiple alignment zones on/off
  - Application is responsible for limiting use to CJK glyphs

Working Together

- Items we can share with you:
  - Saffron 3.1 pre-release library with MAZ support
  - SaffronViewer with MAZ support
  - SaffronCSMTuner with MAZ support